

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

```
*****
44962 Fri Dec 19 13:49:23 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
First pass at 4310
*****
_____ unchanged_portion_omitted_


1046 /*
1047 * Notes:
1048 *      Set up all device state and allocate data structures,
1049 *      mutexes, condition variables, etc. for device operation.
1050 *      Add interrupts needed.
1051 *      Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
1052 */
1053 static int
1054 mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
1055 {
1056     mptsas_t          *mpt = NULL;
1057     int                instance, i, j;
1058     int                doneq_thread_num;
1059     char               intr_added = 0;
1060     char               map_setup = 0;
1061     char               config_setup = 0;
1062     char               hba_attach_setup = 0;
1063     char               smp_attach_setup = 0;
1064     char               mutex_init_done = 0;
1065     char               event_taskq_create = 0;
1066     char               reset_taskq_create = 0;
1067 #endif /* ! codereview */
1068     char               dr_taskq_create = 0;
1069     char               doneq_thread_create = 0;
1070     char               added_watchdog = 0;
1071     scsi_hba_tran_t   *hba_tran;
1072     uint_t             mem_bar = MEM_SPACE;
1073     int                rval = DDI_FAILURE;

1074     /* CONSTCOND */
1075     ASSERT(NO_COMPETING_THREADS);

1076     if (scsi_hba_iport_unit_address(dip)) {
1077         return (mptsas_iport_attach(dip, cmd));
1078     }

1079     switch (cmd) {
1080     case DDI_ATTACH:
1081         break;

1082     case DDI_RESUME:
1083         if ((hba_tran = ddi_get_driver_private(dip)) == NULL)
1084             return (DDI_FAILURE);

1085         mpt = TRAN2MPT(hba_tran);

1086         if (!mpt) {
1087             return (DDI_FAILURE);
1088         }

1089         /*
1090          * Reset hardware and softc to "no outstanding commands"
1091          * Note that a check condition can result on first command
1092          * to a target.
1093          */
1094         mutex_enter(&mpt->m_mutex);

1095         /*
1096          * raise power.
1097          */
1098
1099
1100
1101
1102
1103
1104
```

1

```
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
*****
1105
1106     */
1107     if (mpt->m_options & MPTSAS_OPT_PM) {
1108         mutex_exit(&mpt->m_mutex);
1109         (void) pm_busy_component(dip, 0);
1110         rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
1111         if (rval == DDI_SUCCESS) {
1112             mutex_enter(&mpt->m_mutex);
1113         } else {
1114             /*
1115              * The pm_raise_power() call above failed,
1116              * and that can only occur if we were unable
1117              * to reset the hardware. This is probably
1118              * due to unhealthy hardware, and because
1119              * important filesystems(such as the root
1120              * filesystem) could be on the attached disks,
1121              * it would not be a good idea to continue,
1122              * as we won't be entirely certain we are
1123              * writing correct data. So we panic() here
1124              * to not only prevent possible data corruption,
1125              * but to give developers or end users a hope
1126              * of identifying and correcting any problems.
1127             */
1128             fm_panic("mptsas could not reset hardware "
1129                      "during resume");
1130         }
1131     }
1132     mpt->m_suspended = 0;

1133     /*
1134      * Reinitialize ioc
1135      */
1136     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1137     if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
1138         mutex_exit(&mpt->m_mutex);
1139         if (mpt->m_options & MPTSAS_OPT_PM) {
1140             (void) pm_idle_component(dip, 0);
1141         }
1142         fm_panic("mptsas init chip fail during resume");
1143     }
1144     /*
1145      * mptsas_update_driver_data needs interrupts so enable them
1146      * first.
1147      */
1148     MPTSAS_ENABLE_INTR(mpt);
1149     mptsas_update_driver_data(mpt);

1150     /*
1151      * start requests, if possible */
1152     mptsas_restart_hba(mpt);

1153     mutex_exit(&mpt->m_mutex);

1154     /*
1155      * Restart watch thread
1156      */
1157     mutex_enter(&mptsas_global_mutex);
1158     if (mptsas_timeout_id == 0) {
1159         mptsas_timeout_id = timeout(mptsas_watch, NULL,
1160                                     mptsas_tick);
1161         mptsas_timeouts_enabled = 1;
1162     }
1163     mutex_exit(&mptsas_global_mutex);

1164     /*
1165      * report idle status to pm framework */
1166     if (mpt->m_options & MPTSAS_OPT_PM) {
1167         (void) pm_idle_component(dip, 0);
1168     }
1169
1170
```

2

```

1171         }
1172
1173         return (DDI_SUCCESS);
1174
1175     default:
1176         return (DDI_FAILURE);
1177     }
1178
1179     instance = ddi_get_instance(dip);
1180
1181     /*
1182     * Allocate softc information.
1183     */
1184
1185     if (ddi_soft_state_zalloc(mptsas_state, instance) != DDI_SUCCESS) {
1186         mptsas_log(NULL, CE_WARN,
1187                     "mptsas%d: cannot allocate soft state", instance);
1188         goto fail;
1189     }
1190
1191     mpt = ddi_get_soft_state(mptsas_state, instance);
1192
1193     if (mpt == NULL) {
1194         mptsas_log(NULL, CE_WARN,
1195                     "mptsas%d: cannot get soft state", instance);
1196         goto fail;
1197     }
1198
1199     /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
1200     scsi_size_clean(dip);
1201
1202     mpt->m_dip = dip;
1203     mpt->m_instance = instance;
1204
1205     /* Make a per-instance copy of the structures */
1206     mpt->m_io_dma_attr = mptsas_dma_attrs64;
1207     if (mptsas_use_64bit_msgaddr) {
1208         mpt->m_msg_dma_attr = mptsas_dma_attrs64;
1209     } else {
1210         mpt->m_msg_dma_attr = mptsas_dmaAttrs;
1211     }
1212     mpt->m_reg_acc_attr = mptsas_dev_attr;
1213     mpt->m_dev_acc_attr = mptsas_dev_attr;
1214
1215     /*
1216     * Size of individual request sense buffer
1217     */
1218     mpt->m_req_sense_size = EXTCMDS_STATUS_SIZE;
1219
1220     /*
1221     * Initialize FMA
1222     */
1223     mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
1224                                         DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
1225                                         DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
1226                                         DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);
1227
1228     mptsas_fm_init(mpt);
1229
1230     if (mptsas_alloc_handshake_msg(mpt,
1231         sizeof (MpI2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
1232         mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
1233         goto fail;
1234     }
1235
1236     /*

```

```

1237         * Setup configuration space
1238         */
1239         if (mptsas_config_space_init(mpt) == FALSE) {
1240             mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
1241             goto fail;
1242         }
1243         config_setup++;
1244
1245         if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
1246                                0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
1247             mptsas_log(mpt, CE_WARN, "map setup failed");
1248             goto fail;
1249         }
1250         map_setup++;
1251
1252         /*
1253         * A taskq is created for dealing with the event handler
1254         */
1255         if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
1256                                                       1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1257             mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
1258             goto fail;
1259         }
1260         event_taskq_create++;
1261
1262         /*
1263         * A taskq is created for dealing with dr events
1264         */
1265         if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
1266                                                 "mptsas_dr_taskq",
1267                                                 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1268             mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
1269                         "failed");
1270             goto fail;
1271         }
1272         dr_taskq_create++;
1273
1274         /*
1275         * A taskq is created for dealing with reset events
1276         */
1277         if ((mpt->m_reset_taskq = ddi_taskq_create(dip,
1278                                                       "mptsas_reset_taskq",
1279                                                       1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1280             mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for reset "
1281                         "failed");
1282             goto fail;
1283         }
1284         reset_taskq_create++;
1285
1286 #endif /* ! codereview */
1287         mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1288                                                        0, "mptsas_doneq_thread_threshold_prop", 10);
1289         mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1290                                                        0, "mptsas_doneq_length_threshold_prop", 8);
1291         mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1292                                                    0, "mptsas_doneq_thread_n_prop", 8);
1293
1294         if (mpt->m_doneq_thread_n) {
1295             cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
1296             mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);
1297
1298             mutex_enter(&mpt->m_doneq_mutex);
1299             mpt->m_doneq_thread_id =
1300                 kmem_malloc(sizeof (mptsas_doneq_thread_list_t)
1301                             * mpt->m_doneq_thread_n, KM_SLEEP);

```

```

1303     for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1304         cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
1305                 CV_DRIVER, NULL);
1306         mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
1307                     MUTEX_DRIVER, NULL);
1308         mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1309         mpt->m_doneq_thread_id[j].flag |=
1310             MPTSAS_DONEQ_THREAD_ACTIVE;
1311         mpt->m_doneq_thread_id[j].arg.mpt = mpt;
1312         mpt->m_doneq_thread_id[j].arg.t = j;
1313         mpt->m_doneq_thread_id[j].threadap =
1314             thread_create(NULL, 0, mptsas_doneq_thread,
1315                           &mpt->m_doneq_thread_id[j].arg,
1316                           0, &p0, TS_RUN, minclsyspri);
1317         mpt->m_doneq_thread_id[j].donetail =
1318             &mpt->m_doneq_thread_id[j].doneq;
1319         mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1320     }
1321     mutex_exit(&mpt->m_doneq_mutex);
1322     doneq_thread_create++;
1323 }

1325 /*
1326  * Disable hardware interrupt since we're not ready to
1327  * handle it yet.
1328 */
1329 MPTSAS_DISABLE_INTR(mpt);
1330 if (mptsas_register_intrs(mpt) == FALSE)
1331     goto fail;
1332 intr_added++;

1334 /* Initialize mutex used in interrupt handler */
1335 mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
1336             DDI_INTR_PRI(mpt->m_intr_pri));
1337 mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
1338 mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
1339             DDI_INTR_PRI(mpt->m_intr_pri));
1340 for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1341     mutex_init(&mpt->m_phys_info[i].smhba_info.phy_mutex,
1342                 NULL, MUTEX_DRIVER,
1343                 DDI_INTR_PRI(mpt->m_intr_pri));
1344 }

1346 cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
1347 cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
1348 cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
1349 cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
1350 cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
1351 mutex_init_done++;

1353 mutex_enter(&mpt->m_mutex);
1354 /*
1355  * Initialize power management component
1356 */
1357 if (mpt->m_options & MPTSAS_OPT_PM) {
1358     if (mptsas_init_pm(mpt)) {
1359         mutex_exit(&mpt->m_mutex);
1360         mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
1361                     "failed");
1362         goto fail;
1363     }
1364 }

1366 /*
1367  * Initialize chip using Message Unit Reset, if allowed
1368 */

```

```

1369     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1370     if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
1371         mutex_exit(&mpt->m_mutex);
1372         mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
1373         goto fail;
1374     }

1376     mpt->m_targets = refhash_create(MPTSAS_TARGET_BUCKET_COUNT,
1377                                     mptsas_target_addr_hash, mptsas_target_addr_cmp,
1378                                     mptsas_target_free, sizeof(mptsas_target_t),
1379                                     offsetof(mptsas_target_t, m_link),
1380                                     offsetof(mptsas_target_t, m_addr), KM_SLEEP);

1382 /*
1383  * Fill in the phy_info structure and get the base WWID
1384 */
1385 if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
1386     mptsas_log(mpt, CE_WARN,
1387                 "mptsas_get_manufacture_page5 failed!");
1388     goto fail;
1389 }

1391 if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {
1392     mptsas_log(mpt, CE_WARN,
1393                 "mptsas_get_sas_io_unit_page_hndshk failed!");
1394     goto fail;
1395 }

1397 if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
1398     mptsas_log(mpt, CE_WARN,
1399                 "mptsas_get_manufacture_page0 failed!");
1400     goto fail;
1401 }

1403 mutex_exit(&mpt->m_mutex);

1405 /*
1406  * Register the iport for multiple port HBA
1407 */
1408 mptsas_iport_register(mpt);

1410 /*
1411  * initialize SCSI HBA transport structure
1412 */
1413 if (mptsas_hba_setup(mpt) == FALSE)
1414     goto fail;
1415 hba_attach_setup++;

1417 if (mptsas_smp_setup(mpt) == FALSE)
1418     goto fail;
1419 smp_attach_setup++;

1421 if (mptsas_cache_create(mpt) == FALSE)
1422     goto fail;

1424 mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
1425                                             dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
1426 if (mpt->m_scsi_reset_delay == 0) {
1427     mptsas_log(mpt, CE_NOTE,
1428                 "scsi_reset_delay of 0 is not recommended,"
1429                 " resetting to SCSI_DEFAULT_RESET_DELAY\n");
1430     mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
1431 }

1433 /*
1434  * Initialize the wait and done FIFO queue

```

```

1435     */
1436     mpt->m_donetail = &mpt->m_doneq;
1437     mpt->m_waitqtail = &mpt->m_waitq;
1438     mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
1439     mpt->m_tx_draining = 0;
1440
1441     /*
1442      * ioc cmd queue initialize
1443      */
1444     mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
1445     mpt->m_dev_handle = 0xFFFFF;
1446
1447     MPTSAS_ENABLE_INTR(mpt);
1448
1449     /*
1450      * enable event notification
1451      */
1452     mutex_enter(&mpt->m_mutex);
1453     if (mptsas_ioc_enable_event_notification(mpt)) {
1454         mutex_exit(&mpt->m_mutex);
1455         goto fail;
1456     }
1457     mutex_exit(&mpt->m_mutex);
1458
1459     /*
1460      * used for mptsas_watch
1461      */
1462     mptsas_list_add(mpt);
1463
1464     mutex_enter(&mptsas_global_mutex);
1465     if (mptsas_timeouts_enabled == 0) {
1466         mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
1467                                                       dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);
1468
1469         mptsas_tick = mptsas_scsi_watchdog_tick *
1470                       drv_usectohz((clock_t)1000000);
1471
1472         mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
1473         mptsas_timeouts_enabled = 1;
1474     }
1475     mutex_exit(&mptsas_global_mutex);
1476     added_watchdog++;
1477
1478     /*
1479      * Initialize PHY info for smhba.
1480      * This requires watchdog to be enabled otherwise if interrupts
1481      * don't work the system will hang.
1482      */
1483     if (mptsas_smhba_setup(mpt)) {
1484         mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
1485                    "failed");
1486         goto fail;
1487     }
1488
1489     /* Check all dma handles allocated in attach */
1490     if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
1491         != DDI_SUCCESS) ||
1492         (mptsas_check_dma_handle(mpt->m_dma_req_sense_hdl)
1493          != DDI_SUCCESS) ||
1494         (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
1495          != DDI_SUCCESS) ||
1496         (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
1497          != DDI_SUCCESS) ||
1498         (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
1499          != DDI_SUCCESS) ||
1500         (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
```

```

1501             != DDI_SUCCESS)) {
1502                 goto fail;
1503             }
1504
1505             /* Check all acc handles allocated in attach */
1506             if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
1507                 (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
1508                  != DDI_SUCCESS) ||
1509                 (mptsas_check_acc_handle(mpt->m_acc_req_sense_hdl)
1510                  != DDI_SUCCESS) ||
1511                 (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
1512                  != DDI_SUCCESS) ||
1513                 (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
1514                  != DDI_SUCCESS) ||
1515                 (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
1516                  != DDI_SUCCESS) ||
1517                 (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
1518                  != DDI_SUCCESS) ||
1519                 (mptsas_check_acc_handle(mpt->m_config_handle)
1520                  != DDI_SUCCESS)) {
1521                     goto fail;
1522                 }
1523
1524             /*
1525              * After this point, we are not going to fail the attach.
1526              */
1527
1528             /* Print message of HBA present */
1529             ddi_report_dev(dip);
1530
1531             /* report idle status to pm framework */
1532             if (mpt->m_options & MPTSAS_OPT_PM) {
1533                 (void) pm_idle_component(dip, 0);
1534             }
1535
1536             return (DDI_SUCCESS);
1537
1538 fail:
1539     mptsas_log(mpt, CE_WARN, "attach failed");
1540     mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
1541     ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
1542     if (mpt) {
1543         /* deallocate in reverse order */
1544         if (added_watchdog) {
1545             mptsas_list_del(mpt);
1546             mutex_enter(&mptsas_global_mutex);
1547
1548             if (mptsas_timeout_id && (mptsas_head == NULL)) {
1549                 timeout_id_t tid = mptsas_timeout_id;
1550                 mptsas_timeouts_enabled = 0;
1551                 mptsas_timeout_id = 0;
1552                 mutex_exit(&mptsas_global_mutex);
1553                 (void) untimeout(tid);
1554                 mutex_enter(&mptsas_global_mutex);
1555             }
1556         }
1557     }
1558
1559     mptsas_cache_destroy(mpt);
1560
1561     if (smp_attach_setup) {
1562         mptsas_smp_teardown(mpt);
1563     }
1564     if (hba_attach_setup) {
1565         mptsas_hba_teardown(mpt);
1566     }
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2100
2101
2101
2102
2102
2103
2103
2104
2104
2105
2105
2106
2106
2107
2107
2108
2108
2109
2109
2110
2110
2111
2111
2112
2112
2113
2113
2114
2114
2115
2115
2116
2116
2117
2117
2118
2118
2119
2119
2120
2120
2121
2121
2122
2122
2123
2123
2124
2124
2125
2125
2126
2126
2127
2127
2128
2128
2129
2129
2130
2130
2131
2131
2132
2132
2133
2133
2134
2134
2135
2135
2136
2136
2137
2137
2138
2138
2139
2139
2140
2140
2141
2141
2142
2142
2143
2143
2144
2144
2145
2145
2146
2146
2147
2147
2148
2148
2149
2149
2150
2150
2151
2151
2152
2152
2153
2153
2154
2154
2155
2155
2156
2156
2157
2157
2158
2158
2159
2159
2160
2160
2161
2161
2162
2162
2163
2163
2164
2164
2165
2165
2166
2166
2167
2167
2168
2168
2169
2169
2170
2170
2171
2171
2172
2172
2173
2173
2174
2174
2175
2175
2176
2176
2177
2177
2178
2178
2179
2179
2180
2180
2181
2181
2182
2182
2183
2183
2184
2184
2185
2185
2186
2186
2187
2187
2188
2188
2189
2189
2190
2190
2191
2191
2192
2192
2193
2193
2194
2194
2195
2195
2196
2196
2197
2197
2198
2198
2199
2199
2200
2200
2201
2201
2202
2202
2203
2203
2204
2204
2205
2205
2206
2206
2207
2207
2208
2208
2209
2209
2210
2210
2211
2211
2212
2212
2213
2213
2214
2214
2215
2215
2216
2216
2217
2217
2218
2218
2219
2219
2220
2220
2221
2221
2222
2222
2223
2223
2224
2224
2225
2225
2226
2226
2227
2227
2228
2228
2229
2229
2230
2230
2231
2231
2232
2232
2233
2233
2234
2234
2235
2235
2236
2236
2237
2237
2238
2238
2239
2239
2240
2240
2241
2241
2242
2242
2243
2243
2244
2244
2245
2245
2246
2246
2247
2247
2248
2248
2249
2249
2250
2250
2251
2251
2252
2252
2253
2253
2254
2254
2255
2255
2256
2256
2257
2257
2258
2258
2259
2259
2260
2260
2261
2261
2262
2262
2263
2263
2264
2264
2265
2265
2266
2266
2267
2267
2268
2268
2269
2269
2270
2270
2271
2271
2272
2272
2273
2273
2274
2274
2275
2275
2276
2276
2277
2277
2278
2278
2279
2279
2280
2280
2281
2281
2282
2282
2283
2283
2284
2284
2285
2285
2286
2286
2287
2287
2288
2288
2289
2289
2290
2290
2291
2291
2292
2292
2293
2293
2294
2294
2295
2295
2296
2296
2297
2297
2298
2298
2299
2299
2300
2300
2301
2301
2302
2302
2303
2303
2304
2304
2305
2305
2306
2306
2307
2307
2308
2308
2309
2309
2310
2310
2311
2311
2312
2312
2313
2313
2314
2314
2315
2315
2316
2316
2317
2317
2318
2318
2319
2319
2320
2320
2321
2321
2322
2322
2323
2323
2324
2324
2325
2325
2326
2326
2327
2327
2328
2328
2329
2329
2330
2330
2331
2331
2332
2332
2333
2333
2334
2334
2335
2335
2336
2336
2337
2337
2338
2338
2339
2339
2340
2340
2341
2341
2342
2342
2343
2343
2344
2344
2345
2345
2346
2346
2347
2347
2348
2348
2349
2349
2350
2350
2351
2351
2352
2352
2353
2353
2354
2354
2355
2355
2356
2356
2357
2357
2358
2358
2359
2359
2360
2360
2361
2361
2362
2362
2363
2363
2364
2364
2365
2365
2366
2366
2367
2367
2368
2368
2369
2369
2370
2370
2371
2371
2372
2372
2373
2373
2374
2374
2375
2375
2376
2376
2377
2377
2378
2378
2379
2379
2380
2380
2381
2381
2382
2382
2383
2383
2384
2384
2385
2385
2386
2386
2387
2387
2388
2388
2389
2389
2390
2390
2391
2391
2392
2392
2393
2393
2394
2394
2395
2395
2396
2396
2397
2397
2398
2398
2399
2399
2400
2400
2401
2401
2402
2402
2403
2403
2404
2404
2405
2405
2406
2406
2407
2407
2408
2408
2409
2409
2410
2410
2411
2411
2412
2412
2413
2413
2414
2414
2415
2415
2416
2416
2417
2417
2418
2418
2419
2419
2420
2420
2421
2421
2422
2422
2423
2423
2424
2424
2425
2425
2426
2426
2427
2427
2428
2428
2429
2429
2430
2430
2431
2431
2432
2432
2433
2433
2434
2434
2435
2435
2436
2436
2437
2437
2438
2438
2439
2439
2440
2440
2441
2441
2442
2442
2443
2443
2444
2444
2445
2445
2446
2446
2447
2447
2448
2448
2449
2449
2450
2450
2451
2451
2452
2452
2453
2453
2454
2454
2455
2455
2456
2456
2457
2457
2458
2458
2459
2459
2460
2460
2461
2461
2462
2462
2463
2463
2464
2464
2465
2465
2466
2466
2467
2467
2468
2468
2469
2469
2470
2470
2471
2471
2472
2472
2473
2473
2474
2474
2475
2475
2476
2476
2477
2477
2478
2478
2479
2479
2480
2480
2481
2481
2482
2482
2483
2483
2484
2484
2485
2485
2486
2486
2487
2487
2488
2488
2489
2489
2490
2490
2491
2491
2492
2492
2493
2493
2494
2494
2495
2495
2496
2496
2497
2497
2498
2498
2499
2499
2500
2500
2501
2501
2502
2502
2503
2503
2504
2504
2505
2505
2506
2506
2507
2507
2508
2508
2509
2509
2510
2510
2511
2511
2512
2512
2513
2513
2514
2514
2515
2515
2516
2516
2517
2517
2518
2518
2519
2519
2520
2520
2521
2521
2522
2522
2523
2523
2524
2524
2525
2525
2526
2526
2527
2527
2528
2528
2529
2529
2530
2530
2531
2531
2532
2532
2533
2533
2534
2534
2535
2535
2536
2536
2537
2537
2538
2538
2539
2539
2540
2540
2541
2541
2542
2542
2543
2543
2544
2544
2545
2545
2546
2546
2547
2547
2548
2548
2549
2549
2550
2550
2551
2551
2552
2552
2553
2553
2554
2554
2555
2555
2556
2556
2557
2557
2558
2558
2559
2559
2560
2560
2561
2561
2562
2562
2563
2563
2564
2564
2565
2565
2566
2566
2567
2567
2568
2568
2569
2569
2570
2570
2571
2571
2572
2572
2573
2573
2574
2574
2575
2575
2576
2576
2577
2577
2578
2578
2579
2579
2580
2580
2581
2581
2582
2582
2583
2583
2584
2584
2585
2585
2586
2586
2587
2587
2588
2588
2589
2589
2590
2590
2591
2591
2592
2592
2593
2593
2594
2594
2595
2595
2596
2596
2597
2597
2598
2598
2599
2599
2600
2600
2601
2601
2602
2602
2603
2603
2604
2604
2605
2605
2606
2606
2607
2607
2608
2608
2609
2609
2610
2610
2611
2611
2612
2612
2613
2613
2614
2614
2615
2615
2616
2616
2617
2617
2618
2618
2619
2619
2620
2620
2621
2621
2622
2622
2623
2623
2624
2624
2625
2625
2626
2626
2627
2627
2628
2628
2629
2629
2630
2630
2631
2631
2632
2632
2633
2633
2634
2634
2635
2635
2636
2636
2637
2637
2638
2638
2639
2639
2640
2640
2641
2641
2642
2642
2643
2643
2644
2644
2645
2645
2646
2646
2647
2647
2648
2648
2649
2649
2650
2650
2651
2651
2652
2652
2653
2653
2654
2654
2655
2655
2656
2656
2657
2657
2658
2658
2659
2659
2660
2660
2661
2661
2662
2662
2663
2663
2664
2664
2665
2665
2666
2666
2667
2667
2668
2668
2669
2669
2670
2670
2671
2671
2672
2672
2673
2673
2674
2674
2675
2675
2676
2676
2677
2677
2678
2678
2679
2679
2680
2680
2681
2681
2682
2682
2683
2683
2684
2684
2685
2685
2686
2686
2687
2687
2688
2688
2689
2689
2690
2690
2691
2691
2692
2692
2693
2693
2694
2694
2695
2695
2696
2696
2697
2697
2698
2698
2699
2699
2700
2700
2701
2701
2702
2702
2703
2703
2704
2704
2705
2705
2706
2706
2707
2707
2708
2708
2709
2709
2710
2710
2711
2711
2712
2712
2713
2713
2714
2714
2715
2715
2716
2716
2717
2717
2718
2718
2719
2719
2720
2720
2721
2721
2722
2722
2723
2723
2724
2724
2725
2725
2726
2726
2727
2727
2728
2728
2729
2729
2730
2730
2731
2731
2732
2732
2733
2733
2734
2734
2735
2735
2736
2736
2737
2737
2738
2738
2739
2739
2740
2740
2741
2741
2742
2742
2743
2743
2744
2744
2745
2745
2746
2746
2747
2747
2748
2748
2749
2749
2750
2750
2751
2751
2752
2752
2753
2753
2754
2754
2755
2755
2756
2756
2757
2757
2758
2758
2759
2759
2760
2760
2761
2761
2762
2762
2763
2763
2764
2764
2765
2765
2766
2766
2767
2767
2768
2768
2769
2769
2770
2770
2771
2771
2772
2772
2773
2773
2774
2774
2775
2775
2776
2776
2777
2777
2778
277
```

```

1568     if (mpt->m_targets)
1569         rehash_destroy(mpt->m_targets);
1570     if (mpt->m_smp_targets)
1571         rehash_destroy(mpt->m_smp_targets);

1573     if (mpt->m_active) {
1574         mptsas_free_active_slots(mpt);
1575     }
1576     if (intr_added) {
1577         mptsas_unregister_intrs(mpt);
1578     }

1580     if (doneq_thread_create) {
1581         mutex_enter(&mpt->m_doneq_mutex);
1582         doneq_thread_num = mpt->m_doneq_thread_n;
1583         for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1584             mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1585             mpt->m_doneq_thread_id[j].flag &=
1586                 (~MPTAS_DONEQ_THREAD_ACTIVE);
1587             cv_signal(&mpt->m_doneq_thread_id[j].cv);
1588             mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1589         }
1590         while (mpt->m_doneq_thread_n) {
1591             cv_wait(&mpt->m_doneq_thread_cv,
1592                     &mpt->m_doneq_mutex);
1593         }
1594         for (j = 0; j < doneq_thread_num; j++) {
1595             cv_destroy(&mpt->m_doneq_thread_id[j].cv);
1596             mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
1597         }
1598         kmem_free(mpt->m_doneq_thread_id,
1599                   sizeof (mptsas_doneq_thread_list_t)
1600                   * doneq_thread_num);
1601         mutex_exit(&mpt->m_doneq_mutex);
1602         cv_destroy(&mpt->m_doneq_thread_cv);
1603         mutex_destroy(&mpt->m_doneq_mutex);
1604     }
1605     if (event_taskq_create) {
1606         ddi_taskq_destroy(mpt->m_event_taskq);
1607     }
1608     if (dr_taskq_create) {
1609         ddi_taskq_destroy(mpt->m_dr_taskq);
1610     }
1611     if (reset_taskq_create) {
1612         ddi_taskq_destroy(mpt->m_reset_taskq);
1613     }
1614 #endif /* ! codereview */
1615     if (mutex_init_done) {
1616         mutex_destroy(&mpt->m_tx_waitq_mutex);
1617         mutex_destroy(&mpt->m_passthru_mutex);
1618         mutex_destroy(&mpt->m_mutex);
1619         for (i = 0; i < MPTAS_MAX_PHYS; i++) {
1620             mutex_destroy(
1621                         &mpt->m_phy_info[i].smhba_info.phy_mutex);
1622         }
1623         cv_destroy(&mpt->m_cv);
1624         cv_destroy(&mpt->m_passthru_cv);
1625         cv_destroy(&mpt->m_fw_cv);
1626         cv_destroy(&mpt->m_config_cv);
1627         cv_destroy(&mpt->m_fw_diag_cv);
1628     }
1629     if (map_setup) {
1630         mptsas_cfg_fini(mpt);
1631     }
1632 
```

```

1633     if (config_setup) {
1634         mptsas_config_space_fini(mpt);
1635     }
1636     mptsas_free_handshake_msg(mpt);
1637     mptsas_hba_fini(mpt);

1639     mptsas_fm_fini(mpt);
1640     ddi_soft_state_free(mptsas_state, instance);
1641     ddi_prop_remove_all(dip);
1642 }
1643 return (DDI_FAILURE);
1644 }

1646 static int
1647 mptsas_suspend(dev_info_t *devi)
1648 {
1649     mptsas_t          *mpt, *g;
1650     scsi_hba_tran_t  *tran;

1652     if (scsi_hba_iport_unit_address(devi)) {
1653         return (DDI_SUCCESS);
1654     }

1656     if ((tran = ddi_get_driver_private(devi)) == NULL)
1657         return (DDI_SUCCESS);

1659     mpt = TRAN2MPT(tran);
1660     if (!mpt) {
1661         return (DDI_SUCCESS);
1662     }

1664     mutex_enter(&mpt->m_mutex);

1666     if (mpt->m_suspended++) {
1667         mutex_exit(&mpt->m_mutex);
1668         return (DDI_SUCCESS);
1669     }

1671     /*
1672      * Cancel timeout threads for this mpt
1673      */
1674     if (mpt->m_quiesce_timeid) {
1675         timeout_id_t tid = mpt->m_quiesce_timeid;
1676         mpt->m_quiesce_timeid = 0;
1677         mutex_exit(&mpt->m_mutex);
1678         (void) untimout(tid);
1679         mutex_enter(&mpt->m_mutex);
1680     }

1682     if (mpt->m_restart_cmd_timeid) {
1683         timeout_id_t tid = mpt->m_restart_cmd_timeid;
1684         mpt->m_restart_cmd_timeid = 0;
1685         mutex_exit(&mpt->m_mutex);
1686         (void) untimout(tid);
1687         mutex_enter(&mpt->m_mutex);
1688     }

1690     mutex_exit(&mpt->m_mutex);

1692     (void) pm_idle_component(mpt->m_dip, 0);

1694     /*
1695      * Cancel watch threads if all mpts suspended
1696      */
1697     rw_enter(&mptsas_global_rwlock, RW_WRITER);
1698     for (g = mptsas_head; g != NULL; g = g->m_next) { 
```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c          11

1699         if (!g->m_suspended)
1700             break;
1701     }
1702     rw_exit(&mptsas_global_rwlock);
1703
1704     mutex_enter(&mptsas_global_mutex);
1705     if (g == NULL) {
1706         timeout_id_t tid;
1707
1708         mptsas_timeouts_enabled = 0;
1709         if (mptsas_timeout_id) {
1710             tid = mptsas_timeout_id;
1711             mptsas_timeout_id = 0;
1712             mutex_exit(&mptsas_global_mutex);
1713             (void) untimeout(tid);
1714             mutex_enter(&mptsas_global_mutex);
1715         }
1716         if (mptsas_reset_watch) {
1717             tid = mptsas_reset_watch;
1718             mptsas_reset_watch = 0;
1719             mutex_exit(&mptsas_global_mutex);
1720             (void) untimeout(tid);
1721             mutex_enter(&mptsas_global_mutex);
1722         }
1723     }
1724     mutex_exit(&mptsas_global_mutex);
1725
1726     mutex_enter(&mpt->m_mutex);
1727
1728     /*
1729      * If this mpt is not in full power(PM_LEVEL_D0), just return.
1730      */
1731     if ((mpt->m_options & MPTSAS_OPT_PM) &&
1732         (mpt->m_power_level != PM_LEVEL_D0)) {
1733         mutex_exit(&mpt->m_mutex);
1734         return (DDI_SUCCESS);
1735     }
1736
1737     /* Disable HBA interrupts in hardware */
1738     MPTSAS_DISABLE_INTR(mpt);
1739
1740     /* Send RAID action system shutdown to sync IR
1741      */
1742     mptsas_raid_action_system_shutdown(mpt);
1743
1744     mutex_exit(&mpt->m_mutex);
1745
1746     /* drain the taskq */
1747     ddi_taskq_wait(mpt->m_reset_taskq);
1748 #endif /* ! codereview */
1749     ddi_taskq_wait(mpt->m_event_taskq);
1750     ddi_taskq_wait(mpt->m_dr_taskq);
1751
1752     return (DDI_SUCCESS);
1753 }

1755 #ifdef __sparc
1756 /*ARGSUSED*/
1757 static int
1758 mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd)
1759 {
1760     mptsas_t        *mpt;
1761     scsi_hba_tran_t *tran;
1762
1763     /*
1764      * If this call is for iport, just return.

```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c          12

1765         */
1766         if (scsi_hba_iport_unit_address(devi))
1767             return (DDI_SUCCESS);
1768
1769         if ((tran = ddi_get_driver_private(devi)) == NULL)
1770             return (DDI_SUCCESS);
1771
1772         if ((mpt = TRAN2MPT(tran)) == NULL)
1773             return (DDI_SUCCESS);
1774
1775         /*
1776          * Send RAID action system shutdown to sync IR.  Disable HBA
1777          * interrupts in hardware first.
1778          */
1779         MPTSAS_DISABLE_INTR(mpt);
1780         mptsas_raid_action_system_shutdown(mpt);
1781
1782         return (DDI_SUCCESS);
1783     }
1784 #else /* __sparc */
1785 /*
1786  * quiesce(9E) entry point.
1787 *
1788  * This function is called when the system is single-threaded at high
1789  * PIL with preemption disabled. Therefore, this function must not be
1790  * blocked.
1791 *
1792  * This function returns DDI_SUCCESS on success, or DDI_FAILURE on failure.
1793  * DDI_FAILURE indicates an error condition and should almost never happen.
1794 */
1795 static int
1796 mptsas_quiesce(dev_info_t *devi)
1797 {
1798     mptsas_t        *mpt;
1799     scsi_hba_tran_t *tran;
1800
1801     /*
1802      * If this call is for iport, just return.
1803      */
1804     if (scsi_hba_iport_unit_address(devi))
1805         return (DDI_SUCCESS);
1806
1807     if ((tran = ddi_get_driver_private(devi)) == NULL)
1808         return (DDI_SUCCESS);
1809
1810     if ((mpt = TRAN2MPT(tran)) == NULL)
1811         return (DDI_SUCCESS);
1812
1813     /* Disable HBA interrupts in hardware */
1814     MPTSAS_DISABLE_INTR(mpt);
1815
1816     /* Send RAID action system shutdown to sync IR */
1817     mptsas_raid_action_system_shutdown(mpt);
1818
1819     return (DDI_SUCCESS);
1820 #endif /* __sparc */
1821
1822 /*
1823  * detach(9E). Remove all device allocations and system resources.
1824  * disable device interrupts.
1825  * Return DDI_SUCCESS if done; DDI_FAILURE if there's a problem.
1826  */
1827 static int
1828 mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd)
1829 {
1830     /* CONSTCOND */

```

```

1831     ASSERT(NO_COMPETING_THREADS);
1832     NDBG0(("mptsas_detach: dip=0x%p cmd=0x%p", (void *)devi, (void *)cmd));
1833
1834     switch (cmd) {
1835     case DDI_DETACH:
1836         return (mptsas_do_detach(devi));
1837
1838     case DDI_SUSPEND:
1839         return (mptsas_suspend(devi));
1840
1841     default:
1842         return (DDI_FAILURE);
1843     }
1844     /* NOTREACHED */
1845 }
1846
1847 static int
1848 mptsas_do_detach(dev_info_t *dip)
1849 {
1850     mptsas_t          *mpt;
1851     scsi_hba_tran_t  *tran;
1852     int               circ = 0;
1853     int               circ1 = 0;
1854     mdi_pathinfo_t   *pip = NULL;
1855     int               i;
1856     int               doneq_thread_num = 0;
1857
1858     NDBG0(("mptsas_do_detach: dip=0x%p", (void *)dip));
1859
1860     if ((tran = ndi_flavorv_get(dip, SCSA_FLAVOR_SCSI_DEVICE)) == NULL)
1861         return (DDI_FAILURE);
1862
1863     mpt = TRAN2MPT(tran);
1864     if (!mpt) {
1865         return (DDI_FAILURE);
1866     }
1867     /*
1868      * Still have pathinfo child, should not detach mpt driver
1869      */
1870     if (scsi_hba_iport_unit_address(dip)) {
1871         if (mpt->m_mpvio_enable) {
1872             /*
1873              * MPPIO enabled for the iport
1874              */
1875             ndi_devi_enter(scsi_vhci_dip, &circ1);
1876             ndi_devi_enter(dip, &circ);
1877             while (pip = mdi_get_next_client_path(dip, NULL)) {
1878                 if (mdi_pi_free(pip, 0) == MDI_SUCCESS) {
1879                     continue;
1880                 }
1881                 ndi_devi_exit(dip, circ);
1882                 ndi_devi_exit(scsi_vhci_dip, circ1);
1883                 NDBG12(("detach failed because of "
1884                         "outstanding path info"));
1885                 return (DDI_FAILURE);
1886             }
1887             ndi_devi_exit(dip, circ);
1888             ndi_devi_exit(scsi_vhci_dip, circ1);
1889             (void) mdi_phci_unregister(dip, 0);
1890         }
1891
1892         ddi_prop_remove_all(dip);
1893
1894     }
1895
1896     return (DDI_SUCCESS);
1897 }
```

```

1897     /* Make sure power level is D0 before accessing registers */
1898     if (mpt->m_options & MPTAS_OPT_PM) {
1899         (void) pm_busy_component(dip, 0);
1900         if (mpt->m_power_level != PM_LEVEL_D0) {
1901             if (pm_raise_power(dip, 0, PM_LEVEL_D0) !=
1902                 DDI_SUCCESS) {
1903                 mptsas_log(mpt, CE_WARN,
1904                         "mptsas%d: Raise power request failed.",
1905                         mpt->m_instance);
1906                 (void) pm_idle_component(dip, 0);
1907             }
1908         }
1909     }
1910
1911
1912     /*
1913      * Send RAID action system shutdown to sync IR. After action, send a
1914      * Message Unit Reset. Since after that DMA resource will be freed,
1915      * set ioc to READY state will avoid HBA initiated DMA operation.
1916      */
1917     mutex_enter(&mpt->m_mutex);
1918     MPTSAS_DISABLE_INTR(mpt);
1919     mptsas_raid_action_system_shutdown(mpt);
1920     mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1921     (void) mptsas_ioc_reset(mpt, FALSE);
1922     mutex_exit(mpt->m_mutex);
1923     mptsas_rem_intrs(mpt);
1924     ddi_taskq_destroy(mpt->m_reset_taskq);
1925 #endif /* ! codereview */
1926     ddi_taskq_destroy(mpt->m_event_taskq);
1927     ddi_taskq_destroy(mpt->m_dr_taskq);
1928
1929     if (mpt->m_doneq_thread_n) {
1930         mutex_enter(&mpt->m_doneq_mutex);
1931         doneq_thread_num = mpt->m_doneq_thread_n;
1932         for (i = 0; i < mpt->m_doneq_thread_n; i++) {
1933             mutex_enter(&mpt->m_doneq_thread_id[i].mutex);
1934             mpt->m_doneq_thread_id[i].flag &=
1935                 (~MPTSAS_DONEQ_THREAD_ACTIVE);
1936             cv_signal(&mpt->m_doneq_thread_id[i].cv);
1937             mutex_exit(&mpt->m_doneq_thread_id[i].mutex);
1938         }
1939         while (mpt->m_doneq_thread_n) {
1940             cv_wait(&mpt->m_doneq_thread_cv,
1941                     &mpt->m_doneq_mutex);
1942         }
1943         for (i = 0; i < doneq_thread_num; i++) {
1944             cv_destroy(&mpt->m_doneq_thread_id[i].cv);
1945             mutex_destroy(&mpt->m_doneq_thread_id[i].mutex);
1946         }
1947         kmem_free(mpt->m_doneq_thread_id,
1948                   sizeof (mptsas_doneq_thread_list_t)
1949                   * doneq_thread_num);
1950         mutex_exit(&mpt->m_doneq_mutex);
1951         cv_destroy(&mpt->m_doneq_thread_cv);
1952         mutex_destroy(&mpt->m_doneq_mutex);
1953     }
1954
1955     scsi_hba_reset_notify_tear_down(mpt->m_reset_notify_listf);
1956
1957     mptsas_list_del(mpt);
1958
1959     /*
1960      * Cancel timeout threads for this mpt
1961      */
1962     mutex_enter(&mpt->m_mutex);
```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

15

```

1963     if (mpt->m_quiesce_timeid) {
1964         timeout_id_t tid = mpt->m_quiesce_timeid;
1965         mpt->m_quiesce_timeid = 0;
1966         mutex_exit(&mpt->m_mutex);
1967         (void) untimeout(tid);
1968         mutex_enter(&mpt->m_mutex);
1969     }
1970
1971     if (mpt->m_restart_cmd_timeid) {
1972         timeout_id_t tid = mpt->m_restart_cmd_timeid;
1973         mpt->m_restart_cmd_timeid = 0;
1974         mutex_exit(&mpt->m_mutex);
1975         (void) untimeout(tid);
1976         mutex_enter(&mpt->m_mutex);
1977     }
1978
1979     mutex_exit(&mpt->m_mutex);
1980
1981     /*
1982      * last mpt? ... if active, CANCEL watch threads.
1983      */
1984     mutex_enter(&mptsas_global_mutex);
1985     if (mptsas_head == NULL) {
1986         timeout_id_t tid;
1987         /*
1988          * Clear mptsas_timeouts_enable so that the watch thread
1989          * gets restarted on DDI_ATTACH
1990          */
1991         mptsas_timeouts_enabled = 0;
1992         if (mptsas_timeout_id) {
1993             tid = mptsas_timeout_id;
1994             mptsas_timeout_id = 0;
1995             mutex_exit(&mptsas_global_mutex);
1996             (void) untimeout(tid);
1997             mutex_enter(&mptsas_global_mutex);
1998         }
1999         if (mptsas_reset_watch) {
2000             tid = mptsas_reset_watch;
2001             mptsas_reset_watch = 0;
2002             mutex_exit(&mptsas_global_mutex);
2003             (void) untimeout(tid);
2004             mutex_enter(&mptsas_global_mutex);
2005         }
2006     }
2007     mutex_exit(&mptsas_global_mutex);
2008
2009     /*
2010      * Delete Phy stats
2011      */
2012     mptsas_destroy_phy_stats(mpt);
2013
2014     mptsas_destroy_hashes(mpt);
2015
2016     /*
2017      * Delete nt_active.
2018      */
2019     mutex_enter(&mpt->m_mutex);
2020     mptsas_free_active_slots(mpt);
2021     mutex_exit(&mpt->m_mutex);
2022
2023     /* deallocate everything that was allocated in mptsas_attach */
2024     mptsas_cache_destroy(mpt);
2025
2026     mptsas_hba_fini(mpt);
2027     mptsas_cfg_fini(mpt);

```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mpptsas.c

2029     /* Lower the power informing PM Framework */
2030     if (mpt->m_options & MPTSAS_OPT_PM) {
2031         if (pm_lower_power(dip, 0, PM_LEVEL_D3) != DDI_SUCCESS)
2032             mpptsas_log(mpt, CE_WARN,
2033                         "!mpptsas%d: Lower power request failed "
2034                         "during detach, ignoring.",
2035                         mpt->m_instance);
2036     }

2038     mutex_destroy(&mpt->m_tx_waitq_mutex);
2039     mutex_destroy(&mpt->m_passthru_mutex);
2040     mutex_destroy(&mpt->m_mutex);
2041     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
2042         mutex_destroy(&mpt->m_phy_info[i].smhba_info.phy_mutex);
2043     }
2044     cv_destroy(&mpt->m_cv);
2045     cv_destroy(&mpt->m_passthru_cv);
2046     cv_destroy(&mpt->m_fw_cv);
2047     cv_destroy(&mpt->m_config_cv);
2048     cv_destroy(&mpt->m_fw_diag_cv);

2051     mpptsas_smp_teardown(mpt);
2052     mpptsas_hba_teardown(mpt);

2054     mpptsas_config_space_fini(mpt);

2056     mpptsas_free_handshake_msg(mpt);

2058     mpptsas_fm_fini(mpt);
2059     ddi_soft_state_free(mpptsas_state, ddi_get_instance(dip));
2060     ddi_prop_remove_all(dip);

2062     return (DDI_SUCCESS);
2063 }

2065 static void
2066 mpptsas_list_add(mpptsas_t *mpt)
2067 {
2068     rw_enter(&mpptsas_global_rwlock, RW_WRITER);

2070     if (mpptsas_head == NULL) {
2071         mpptsas_head = mpt;
2072     } else {
2073         mpptsas_tail->m_next = mpt;
2074     }
2075     mpptsas_tail = mpt;
2076     rw_exit(&mpptsas_global_rwlock);
2077 }

2079 static void
2080 mpptsas_list_del(mpptsas_t *mpt)
2081 {
2082     mpptsas_t *m;
2083     /*
2084      * Remove device instance from the global linked list
2085      */
2086     rw_enter(&mpptsas_global_rwlock, RW_WRITER);
2087     if (mpptsas_head == mpt) {
2088         m = mpptsas_head = mpt->m_next;
2089     } else {
2090         for (m = mpptsas_head; m != NULL; m = m->m_next) {
2091             if (m->m_next == mpt) {
2092                 m->m_next = mpt->m_next;
2093                 break;
2094             }

```

```

2095         }
2096         if (m == NULL) {
2097             mptsas_log(mpt, CE_PANIC, "Not in softc list!");
2098         }
2099     }
2100
2101     if (mptsas_tail == mpt) {
2102         mptsas_tail = m;
2103     }
2104     rw_exit(&mptsas_global_rwlock);
2105 }
2106
2107 static int
2108 mptsas_alloc_handshake_msg(mptsas_t *mpt, size_t alloc_size)
2109 {
2110     ddi_dma_attr_t task_dmaAttrs;
2111
2112     mpt->m_hshk_dma_size = 0;
2113     task_dmaAttrs = mpt->m_msg_dma_attr;
2114     task_dmaAttrs.dma_attr_sgllen = 1;
2115     task_dmaAttrs.dma_attr_granular = (uint32_t)(alloc_size);
2116
2117     /* allocate Task Management ddi_dma resources */
2118     if (mptsas_dma_addr_create(mpt, task_dmaAttrs,
2119         &mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl, &mpt->m_hshk_memp,
2120         alloc_size, NULL) == FALSE) {
2121         return (DDI_FAILURE);
2122     }
2123     mpt->m_hshk_dma_size = alloc_size;
2124
2125     return (DDI_SUCCESS);
2126 }
2127
2128 static void
2129 mptsas_free_handshake_msg(mptsas_t *mpt)
2130 {
2131     if (mpt->m_hshk_dma_size == 0)
2132         return;
2133     mptsas_dma_addr_destroy(&mpt->m_hshk_dma_hdl, &mpt->m_hshk_acc_hdl);
2134     mpt->m_hshk_dma_size = 0;
2135 }
2136
2137 static int
2138 mptsas_hba_setup(mptsas_t *mpt)
2139 {
2140     scsi_hba_tran_t          *hba_tran;
2141     int                      tran_flags;
2142
2143     /* Allocate a transport structure */
2144     hba_tran = mpt->m_tran = scsi_hba_tran_alloc(mpt->m_dip,
2145         SCSI_HBA_CANSLEEP);
2146     ASSERT(mpt->m_tran != NULL);
2147
2148     hba_tran->tran_hba_private      = mpt;
2149     hba_tran->tran_tgt_private     = NULL;
2150
2151     hba_tran->tran_tgt_init        = mptsas_scsi_tgt_init;
2152     hba_tran->tran_tgt_free        = mptsas_scsi_tgt_free;
2153
2154     hba_tran->tran_start           = mptsas_scsi_start;
2155     hba_tran->tran_reset            = mptsas_scsi_reset;
2156     hba_tran->tran_abort             = mptsas_scsi_abort;
2157     hba_tran->tran_getcap           = mptsas_scsi_getcap;
2158     hba_tran->tran_setcap           = mptsas_scsi_setcap;
2159     hba_tran->tran_init_pkt          = mptsas_scsi_init_pkt;
2160     hba_tran->tran_destroy_pkt       = mptsas_scsi_destroy_pkt;

```

```

2162     hba_tran->tran_dmafree          = mptsas_scsi_dmafree;
2163     hba_tran->tran_sync_pkt         = mptsas_scsi_sync_pkt;
2164     hba_tran->tran_reset_notify     = mptsas_scsi_reset_notify;
2165
2166     hba_tran->tran_get_bus_addr     = mptsas_get_bus_addr;
2167     hba_tran->tran_get_name          = mptsas_get_name;
2168
2169     hba_tran->tran_quiesce          = mptsas_scsi_quiesce;
2170     hba_tran->tran_unquiesce        = mptsas_scsi_unquiesce;
2171     hba_tran->tran_bus_reset         = NULL;
2172
2173     hba_tran->tran_add_eventcall    = NULL;
2174     hba_tran->tran_get_eventcookie   = NULL;
2175     hba_tran->tran_post_event        = NULL;
2176     hba_tran->tran_remove_eventcall  = NULL;
2177
2178     hba_tran->tran_bus_config        = mptsas_bus_config;
2179
2180     hba_tran->tran_interconnect_type = INTERCONNECT_SAS;
2181
2182     /*
2183      * All children of the HBA are iports. We need tran was cloned.
2184      * So we pass the flags to SCSA. SCSI_HBA_TRAN_CLONE will be
2185      * inherited to iport's tran vector.
2186      */
2187     tran_flags = (SCSI_HBA_HBA | SCSI_HBA_TRAN_CLONE);
2188
2189     if (scsi_hba_attach_setup(mpt->m_dip, &mpt->m_msg_dma_attr,
2190         hba_tran, tran_flags) != DDI_SUCCESS) {
2191         mptsas_log(mpt, CE_WARN, "hba attach setup failed");
2192         scsi_hba_tran_free(hba_tran);
2193         mpt->m_tran = NULL;
2194         return (FALSE);
2195     }
2196     return (TRUE);
2197 }
2198
2199 static void
2200 mptsas_hba_teardown(mptsas_t *mpt)
2201 {
2202     (void) scsi_hba_detach(mpt->m_dip);
2203     if (mpt->m_tran != NULL) {
2204         scsi_hba_tran_free(mpt->m_tran);
2205         mpt->m_tran = NULL;
2206     }
2207 }
2208
2209 static void
2210 mptsas_iport_register(mptsas_t *mpt)
2211 {
2212     int i, j;
2213     mptsas_phymask_t          mask = 0x0;
2214     /*
2215      * initial value of mask is 0
2216      */
2217     mutex_enter(&mpt->m_mutex);
2218     for (i = 0; i < mpt->m_num_phys; i++) {
2219         mptsas_phymask_t phy_mask = 0x0;
2220         char phy_mask_name[MPTSAS_MAX_PHYS];
2221         uint8_t current_port;
2222
2223         if (mpt->m_phy_info[i].attached_devhdl == 0)
2224             continue;
2225
2226         bzero(phy_mask_name, sizeof (phy_mask_name));

```

```

2228         current_port = mpt->m_phy_info[i].port_num;
2229
2230         if ((mask & (1 << i)) != 0)
2231             continue;
2232
2233         for (j = 0; j < mpt->m_num_phys; j++) {
2234             if (mpt->m_phy_info[j].attached_devhdl &&
2235                 (mpt->m_phy_info[j].port_num == current_port)) {
2236                 phy_mask |= (1 << j);
2237             }
2238         }
2239         mask = mask | phy_mask;
2240
2241         for (j = 0; j < mpt->m_num_phys; j++) {
2242             if ((phy_mask >> j) & 0x01) {
2243                 mpt->m_phy_info[j].phy_mask = phy_mask;
2244             }
2245         }
2246
2247         (void) sprintf(phy_mask_name, "%x", phy_mask);
2248
2249         mutex_exit(&mpt->m_mutex);
2250
2251         /*
2252          * register a iport
2253          */
2254         (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
2255         mutex_enter(&mpt->m_mutex);
2256     }
2257
2258     /*
2259      * register a virtual port for RAID volume always
2260      */
2261     (void) scsi_hba_iport_register(mpt->m_dip, "v0");
2262 }
2263
2264 static int
2265 mptsas_smp_setup(mptsas_t *mpt)
2266 {
2267     mpt->m_smprtran = smp_hba_tran_alloc(mpt->m_dip);
2268     ASSERT(mpt->m_smprtran != NULL);
2269     mpt->m_smprtran->smp_tran_hba_private = mpt;
2270     mpt->m_smprtran->smp_tran_start = mptsas_smp_start;
2271     if (smp_hba_attach_setup(mpt->m_dip, mpt->m_smprtran) != DDI_SUCCESS) {
2272         mptsas_log(mpt, CE_WARN, "smp attach setup failed");
2273         smp_hba_tran_free(mpt->m_smprtran);
2274         mpt->m_smprtran = NULL;
2275         return (FALSE);
2276     }
2277
2278     /*
2279      * Initialize smp hash table
2280      */
2281     mpt->m_smp_targets = refhash_create(MPTSAS_SMP_BUCKET_COUNT,
2282                                         mptsas_target_addr_hash, mptsas_target_addr_cmp,
2283                                         mptsas_smp_free, sizeof(mptsas_smp_t),
2284                                         offsetof(mptsas_smp_t, m_link), offsetof(mptsas_smp_t, m_addr),
2285                                         KM_SLEEP);
2286     mpt->m_smp_devhdl = 0xFFFF;
2287
2288     return (TRUE);
2289 }
2290
2291 static void
2292 mptsas_smp_teardown(mptsas_t *mpt)
2293 {

```

```

2293     (void) smp_hba_detach(mpt->m_dip);
2294     if (mpt->m_smprtran != NULL) {
2295         smp_hba_tran_free(mpt->m_smprtran);
2296         mpt->m_smprtran = NULL;
2297     }
2298     mpt->m_smp_devhdl = 0;
2299 }
2300
2301 static int
2302 mptsas_cache_create(mptsas_t *mpt)
2303 {
2304     int instance = mpt->m_instance;
2305     char buf[64];
2306
2307     /*
2308      * create kmem cache for packets
2309      */
2310     (void) sprintf(buf, "mptsas%d_cache", instance);
2311     mpt->m_kmem_cache = kmem_cache_create(buf,
2312                                             sizeof(struct mptsas_cmd) + scsi_pkt_size(), 8,
2313                                             mptsas_kmem_cache_constructor, mptsas_kmem_cache_destructor,
2314                                             NULL, (void *)mpt, NULL, 0);
2315
2316     if (mpt->m_kmem_cache == NULL) {
2317         mptsas_log(mpt, CE_WARN, "creating kmem cache failed");
2318         return (FALSE);
2319     }
2320
2321     /*
2322      * create kmem cache for extra SGL frames if SGL cannot
2323      * be accomodated into main request frame.
2324      */
2325     (void) sprintf(buf, "mptsas%d_cache_frames", instance);
2326     mpt->m_cache_frames = kmem_cache_create(buf,
2327                                             sizeof(mptsas_cache_frames_t), 8,
2328                                             mptsas_cache_frames_constructor, mptsas_cache_frames_destructor,
2329                                             NULL, (void *)mpt, NULL, 0);
2330
2331     if (mpt->m_cache_frames == NULL) {
2332         mptsas_log(mpt, CE_WARN, "creating cache for frames failed");
2333         return (FALSE);
2334     }
2335
2336     return (TRUE);
2337 }
2338
2339 static void
2340 mptsas_cache_destroy(mptsas_t *mpt)
2341 {
2342     /* deallocate in reverse order */
2343     if (mpt->m_cache_frames) {
2344         kmem_cache_destroy(mpt->m_cache_frames);
2345         mpt->m_cache_frames = NULL;
2346     }
2347     if (mpt->m_kmem_cache) {
2348         kmem_cache_destroy(mpt->m_kmem_cache);
2349         mpt->m_kmem_cache = NULL;
2350     }
2351 }
2352
2353 static int
2354 mptsas_power(dev_info_t *dip, int component, int level)
2355 {
2356 #ifndef __lock_lint
2357     _NOTE(ARGUNUSED(component))
2358 #endif

```

```

2359     mptsas_t      *mpt;
2360     int          rval = DDI_SUCCESS;
2361     int          polls = 0;
2362     uint32_t     ioc_status;
2363
2364     if (scsi_hba_iport_unit_address(dip) != 0)
2365         return (DDI_SUCCESS);
2366
2367     mpt = ddi_get_soft_state(mptsas_state, ddi_get_instance(dip));
2368     if (mpt == NULL) {
2369         return (DDI_FAILURE);
2370     }
2371
2372     mutex_enter(&mpt->m_mutex);
2373
2374     /*
2375      * If the device is busy, don't lower its power level
2376      */
2377     if (mpt->m_busy && (mpt->m_power_level > level)) {
2378         mutex_exit(&mpt->m_mutex);
2379         return (DDI_FAILURE);
2380     }
2381     switch (level) {
2382     case PM_LEVEL_D0:
2383         NDBG11(("mptsas%d: turning power ON.", mpt->m_instance));
2384         MPTSAS_POWER_ON(mpt);
2385         /*
2386          * Wait up to 30 seconds for IOC to come out of reset.
2387          */
2388         while (((ioc_status = ddi_get32(mpt->m_datap,
2389             &mpt->m_reg->Doorbell)) &
2390             MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_RESET) {
2391             if (polls++ > 3000) {
2392                 break;
2393             }
2394             delay(driv_usectohz(10000));
2395         }
2396         /*
2397          * If IOC is not in operational state, try to hard reset it.
2398          */
2399     if ((ioc_status & MPI2_IOC_STATE_MASK) !=
2400         MPI2_IOC_STATE_OPERATIONAL) {
2401         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
2402         if (mptsas_restart_lic(mpt) == DDI_FAILURE) {
2403             mptsas_log(mpt, CE_WARN,
2404                 "mptsas_power: hard reset failed");
2405             mutex_exit(&mpt->m_mutex);
2406             return (DDI_FAILURE);
2407         }
2408     }
2409     mpt->m_power_level = PM_LEVEL_D0;
2410     break;
2411     case PM_LEVEL_D3:
2412         NDBG11(("mptsas%d: turning power OFF.", mpt->m_instance));
2413         MPTSAS_POWER_OFF(mpt);
2414         break;
2415     default:
2416         mptsas_log(mpt, CE_WARN, "mptsas%d: unknown power level <%x>.",
2417             mpt->m_instance, level);
2418         rval = DDI_FAILURE;
2419         break;
2420     }
2421     mutex_exit(&mpt->m_mutex);
2422     return (rval);
2423 }
```

```

2425     /*
2426      * Initialize configuration space and figure out which
2427      * chip and revision of the chip the mpt driver is using.
2428      */
2429     static int
2430     mptsas_config_space_init(mptsas_t *mpt)
2431     {
2432         NDBG0(("mptsas_config_space_init"));
2433
2434         if (mpt->m_config_handle != NULL)
2435             return (TRUE);
2436
2437         if (pci_config_setup(mpt->m_dip,
2438             &mpt->m_config_handle) != DDI_SUCCESS) {
2439             mptsas_log(mpt, CE_WARN, "cannot map configuration space.");
2440             return (FALSE);
2441         }
2442
2443         /*
2444          * This is a workaround for a XMITS ASIC bug which does not
2445          * drive the CBE upper bits.
2446          */
2447         if (pci_config_get16(mpt->m_config_handle, PCI_CONF_STAT) &
2448             PCI_STAT_PERROR) {
2449             pci_config_put16(mpt->m_config_handle, PCI_CONF_STAT,
2450                             PCI_STAT_PERROR);
2451         }
2452
2453         mptsas_setup_cmd_reg(mpt);
2454
2455         /*
2456          * Get the chip device id:
2457          */
2458         mpt->m_devid = pci_config_get16(mpt->m_config_handle, PCI_CONF_DEVID);
2459
2460         /*
2461          * Save the revision.
2462          */
2463         mpt->m_revid = pci_config_get8(mpt->m_config_handle, PCI_CONF_REVID);
2464
2465         /*
2466          * Save the SubSystem Vendor and Device IDs
2467          */
2468         mpt->m_svrid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBVENID);
2469         mpt->m_ssrid = pci_config_get16(mpt->m_config_handle, PCI_CONF_SUBSYSID);
2470
2471         /*
2472          * Set the latency timer to 0x40 as specified by the upa -> pci
2473          * bridge chip design team. This may be done by the sparc pci
2474          * bus nexus driver, but the driver should make sure the latency
2475          * timer is correct for performance reasons.
2476          */
2477         pci_config_put8(mpt->m_config_handle, PCI_CONF_LATENCY_TIMER,
2478                         MPTSAS_LATENCY_TIMER);
2479
2480         (void) mptsas_get_pci_cap(mpt);
2481         return (TRUE);
2482     }
2483
2484     static void
2485     mptsas_config_space_fini(mptsas_t *mpt)
2486     {
2487         if (mpt->m_config_handle != NULL) {
2488             mptsas_disable_bus_master(mpt);
2489             pci_config_teardown(&mpt->m_config_handle);
2490             mpt->m_config_handle = NULL;
2491         }
2492     }

```

```

2491     }
2492 }

2494 static void
2495 mptsas_setup_cmd_reg(mptsas_t *mpt)
2496 {
2497     ushort_t      cmdreg;
2498
2499     /*
2500      * Set the command register to the needed values.
2501      */
2502     cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
2503     cmdreg |= (PCI_COMM_ME | PCI_COMM_SERR_ENABLE |
2504                PCI_COMM_PARITY_DETECT | PCI_COMM_MAE);
2505     cmdreg &= ~PCI_COMM_IO;
2506     pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
2507 }

2509 static void
2510 mptsas_disable_bus_master(mptsas_t *mpt)
2511 {
2512     ushort_t      cmdreg;
2513
2514     /*
2515      * Clear the master enable bit in the PCI command register.
2516      * This prevents any bus mastering activity like DMA.
2517      */
2518     cmdreg = pci_config_get16(mpt->m_config_handle, PCI_CONF_COMM);
2519     cmdreg &= ~PCI_COMM_MB;
2520     pci_config_put16(mpt->m_config_handle, PCI_CONF_COMM, cmdreg);
2521 }

2523 int
2524 mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_statep)
2525 {
2526     ddi_dma_attr_t attrs;
2527
2528     attrs = mpt->m_io_dma_attr;
2529     attrs.dma_attr_sgllen = 1;
2530
2531     ASSERT(dma_statep != NULL);
2532
2533     if (mptsas_dma_addr_create(mpt, attrs, &dma_statep->handle,
2534                                &dma_statep->accesssp, &dma_statep->memp, dma_statep->size,
2535                                &dma_statep->cookie) == FALSE) {
2536         return (DDI_FAILURE);
2537     }
2538
2539     return (DDI_SUCCESS);
2540 }

2542 void
2543 mptsas_dma_free(mptsas_dma_alloc_state_t *dma_statep)
2544 {
2545     ASSERT(dma_statep != NULL);
2546     mptsas_dma_addr_destroy(&dma_statep->handle, &dma_statep->accesssp);
2547     dma_statep->size = 0;
2548 }

2550 int
2551 mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)())
2552 {
2553     ddi_dma_attr_t      attrs;
2554     ddi_dma_handle_t    dma_handle;
2555     caddr_t             memp;
2556     ddi_acc_handle_t    accesssp;

```

```

2557     int          rval;
2559
2560     ASSERT(mutex_owned(&mpt->m_mutex));
2561
2562     attrs = mpt->m_msg_dma_attr;
2563     attrs.dma_attr_sgllen = 1;
2564     attrs.dma_attr_granular = size;
2565
2566     if (mptsas_dma_addr_create(mpt, attrs, &dma_handle,
2567                                &accesssp, &memp, size, NULL) == FALSE) {
2568         return (DDI_FAILURE);
2569     }
2570
2571     rval = (*callback) (mpt, memp, var, accesssp);
2572
2573     if ((mptsas_check_dma_handle(dma_handle) != DDI_SUCCESS) ||
2574         (mptsas_check_acc_handle(accesssp) != DDI_SUCCESS)) {
2575         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2576         rval = DDI_FAILURE;
2577     }
2578
2579     mptsas_dma_addr_destroy(&dma_handle, &accesssp);
2580     return (rval);
2581 }

2583 static int
2584 mptsas_alloc_request_frames(mptsas_t *mpt)
2585 {
2586     ddi_dma_attr_t      frame_dmaAttrs;
2587     caddr_t             memp;
2588     ddi_dma_cookie_t   cookie;
2589     size_t              mem_size;
2590
2591     /*
2592      * re-alloc when it has already allocoed
2593      */
2594     if (mpt->m_dma_req_frame_hdl)
2595         mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
2596                                &mpt->m_acc_req_frame_hdl);
2597
2598     /*
2599      * The size of the request frame pool is:
2600      * Number of Request Frames * Request Frame Size
2601      */
2602     mem_size = mpt->m_max_requests * mpt->m_req_frame_size;
2603
2604     /*
2605      * set the DMA attributes. System Request Message Frames must be
2606      * aligned on a 16-byte boundry.
2607      */
2608     frame_dmaAttrs = mpt->m_msg_dma_attr;
2609     frame_dmaAttrs.dma_attr_align = 16;
2610     frame_dmaAttrs.dma_attr_sgllen = 1;
2611
2612     /*
2613      * allocate the request frame pool.
2614      */
2615     if (mptsas_dma_addr_create(mpt, frame_dmaAttrs,
2616                                &mpt->m_dma_req_frame_hdl, &mpt->m_acc_req_frame_hdl, &memp,
2617                                mem_size, &cookie) == FALSE) {
2618         return (DDI_FAILURE);
2619     }
2620
2621     /*
2622      * Store the request frame memory address. This chip uses this

```

```

2623     * address to dma to and from the driver's frame. The second
2624     * address is the address mpt uses to fill in the frame.
2625     */
2626     mpt->m_req_frame_dma_addr = cookie.dmac_laddress;
2627     mpt->m_req_frame = memp;
2628
2629     /*
2630     * Clear the request frame pool.
2631     */
2632     bzero(mpt->m_req_frame, mem_size);
2633
2634     return (DDI_SUCCESS);
2635 }
2636
2637 static int
2638 mptsas_alloc_sense_bufs(mptsas_t *mpt)
2639 {
2640     ddi_dma_attr_t          sense_dma_attrs;
2641     caddr_t                 memp;
2642     ddi_dma_cookie_t        cookie;
2643     size_t                  mem_size;
2644     int                     num_extreqsense_bufs;
2645
2646     /*
2647     * re-alloc when it has already allocoed
2648     */
2649     if (mpt->m_dma_req_sense_hdl) {
2650         rmfreemap(mpt->m_ergsense_map);
2651         mptsas_dma_addr_destroy(&mpt->m_dma_req_sense_hdl,
2652                                &mpt->m_acc_req_sense_hdl);
2653     }
2654
2655     /*
2656     * The size of the request sense pool is:
2657     *   (Number of Request Frames - 2) * Request Sense Size +
2658     *   extra memory for extended sense requests.
2659     */
2660     mem_size = ((mpt->m_max_requests - 2) * mpt->m_req_sense_size) +
2661             mptsas_extreq_sense_bufsize;
2662
2663     /*
2664     * set the DMA attributes. ARQ buffers
2665     * aligned on a 16-byte boundary.
2666     */
2667     sense_dma_attrs = mpt->m_msg_dma_attr;
2668     sense_dma_attrs.dma_attr_align = 16;
2669     sense_dma_attrs.dma_attr_sgllen = 1;
2670
2671     /*
2672     * allocate the request sense buffer pool.
2673     */
2674     if (mptsas_dma_addr_create(mpt, sense_dma_attrs,
2675                               &mpt->m_dma_req_sense_hdl, &mpt->m_acc_req_sense_hdl, &memp,
2676                               mem_size, &cookie) == FALSE) {
2677         return (DDI_FAILURE);
2678     }
2679
2680     /*
2681     * Store the request sense base memory address. This chip uses this
2682     * address to dma the request sense data. The second
2683     * address is the address mpt uses to access the data.
2684     * The third is the base for the extended rqsense buffers.
2685     */
2686     mpt->m_req_sense_dma_addr = cookie.dmac_laddress;
2687     mpt->m_req_sense = memp;
2688     memp += (mpt->m_max_requests - 2) * mpt->m_req_sense_size;

```

```

2689     mpt->m_extreq_sense = memp;
2690
2691     /*
2692     * The extra memory is divided up into multiples of the base
2693     * buffer size in order to allocate via rmalloc().
2694     * Note that the rmallocmap cannot start at zero!
2695     */
2696     num_extreqsense_bufs = mptsas_extreq_sense_bufsize /
2697                           mpt->m_req_sense_size;
2698     mpt->m_ergsense_map = rmallocmap_wait(num_extreqsense_bufs);
2699     rmfree(mpt->m_ergsense_map, num_extreqsense_bufs, 1);
2700
2701     /*
2702     * Clear the pool.
2703     */
2704     bzero(mpt->m_req_sense, mem_size);
2705
2706     return (DDI_SUCCESS);
2707 }
2708
2709 static int
2710 mptsas_alloc_reply_frames(mptsas_t *mpt)
2711 {
2712     ddi_dma_attr_t          frame_dma_attrs;
2713     caddr_t                 memp;
2714     ddi_dma_cookie_t        cookie;
2715     size_t                  mem_size;
2716
2717     /*
2718     * re-alloc when it has already allocoed
2719     */
2720     if (mpt->m_dma_reply_frame_hdl) {
2721         mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
2722                                &mpt->m_acc_reply_frame_hdl);
2723     }
2724
2725     /*
2726     * The size of the reply frame pool is:
2727     *   Number of Reply Frames * Reply Frame Size
2728     */
2729     mem_size = mpt->m_max_replies * mpt->m_reply_frame_size;
2730
2731     /*
2732     * set the DMA attributes. System Reply Message Frames must be
2733     * aligned on a 4-byte boundry. This is the default.
2734     */
2735     frame_dma_attrs = mpt->m_msg_dma_attr;
2736     frame_dma_attrs.dma_attr_sgllen = 1;
2737
2738     /*
2739     * allocate the reply frame pool
2740     */
2741     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2742                               &mpt->m_dma_reply_frame_hdl, &mpt->m_acc_reply_frame_hdl, &memp,
2743                               mem_size, &cookie) == FALSE) {
2744         return (DDI_FAILURE);
2745     }
2746
2747     /*
2748     * Store the reply frame memory address. This chip uses this
2749     * address to dma to and from the driver's frame. The second
2750     * address is the address mpt uses to process the frame.
2751     */
2752     mpt->m_reply_frame_dma_addr = cookie.dmac_laddress;
2753     mpt->m_reply_frame = memp;

```

```

2755     /*
2756      * Clear the reply frame pool.
2757      */
2758     bzero(mpt->m_reply_frame, mem_size);
2759
2760     return (DDI_SUCCESS);
2761 }
2762
2763 static int
2764 mptsas_alloc_free_queue(mptsas_t *mpt)
2765 {
2766     ddi_dma_attr_t          frame_dma_attrs;
2767     caddr_t                 memp;
2768     ddi_dma_cookie_t        cookie;
2769     size_t                  mem_size;
2770
2771     /*
2772      * re-alloc when it has already allocoed
2773      */
2774     if (mpt->m_dma_free_queue_hdl) {
2775         mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
2776                                 &mpt->m_acc_free_queue_hdl);
2777     }
2778
2779     /*
2780      * The reply free queue size is:
2781      *   Reply Free Queue Depth * 4
2782      * The "4" is the size of one 32 bit address (low part of 64-bit
2783      *   address)
2784      */
2785     mem_size = mpt->m_free_queue_depth * 4;
2786
2787     /*
2788      * set the DMA attributes  The Reply Free Queue must be aligned on a
2789      * 16-byte boundry.
2790      */
2791     frame_dma_attrs = mpt->m_msg_dma_attr;
2792     frame_dma_attrs.dma_attr_align = 16;
2793     frame_dma_attrs.dma_attr_sgllen = 1;
2794
2795     /*
2796      * allocate the reply free queue
2797      */
2798     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2799                               &mpt->m_dma_free_queue_hdl, &mpt->m_acc_free_queue_hdl, &memp,
2800                               mem_size, &cookie) == FALSE) {
2801         return (DDI_FAILURE);
2802     }
2803
2804     /*
2805      * Store the reply free queue memory address. This chip uses this
2806      * address to read from the reply free queue. The second address
2807      * is the address mpt uses to manage the queue.
2808      */
2809     mpt->m_free_queue_dma_addr = cookie.dmac_laddress;
2810     mpt->m_free_queue = memp;
2811
2812     /*
2813      * Clear the reply free queue memory.
2814      */
2815     bzero(mpt->m_free_queue, mem_size);
2816
2817     return (DDI_SUCCESS);
2818 }
2819
2820 static int

```

```

2821 mptsas_alloc_post_queue(mptsas_t *mpt)
2822 {
2823     ddi_dma_attr_t          frame_dma_attrs;
2824     caddr_t                 memp;
2825     ddi_dma_cookie_t        cookie;
2826     size_t                  mem_size;
2827
2828     /*
2829      * re-alloc when it has already allocoed
2830      */
2831     if (mpt->m_dma_post_queue_hdl) {
2832         mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
2833                                 &mpt->m_acc_post_queue_hdl);
2834     }
2835
2836     /*
2837      * The reply descriptor post queue size is:
2838      *   Reply Descriptor Post Queue Depth * 8
2839      * The "8" is the size of each descriptor (8 bytes or 64 bits).
2840      */
2841     mem_size = mpt->m_post_queue_depth * 8;
2842
2843     /*
2844      * set the DMA attributes. The Reply Descriptor Post Queue must be
2845      * aligned on a 16-byte boundary.
2846      */
2847     frame_dma_attrs = mpt->m_msg_dma_attr;
2848     frame_dma_attrs.dma_attr_align = 16;
2849     frame_dma_attrs.dma_attr_sgllen = 1;
2850
2851     /*
2852      * allocate the reply post queue
2853      */
2854     if (mptsas_dma_addr_create(mpt, frame_dma_attrs,
2855                               &mpt->m_dma_post_queue_hdl, &mpt->m_acc_post_queue_hdl, &memp,
2856                               mem_size, &cookie) == FALSE) {
2857         return (DDI_FAILURE);
2858     }
2859
2860     /*
2861      * Store the reply descriptor post queue memory address. This chip
2862      * uses this address to write to the reply descriptor post queue. The
2863      * second address is the address mpt uses to manage the queue.
2864      */
2865     mpt->m_post_queue_dma_addr = cookie.dmac_laddress;
2866     mpt->m_post_queue = memp;
2867
2868     /*
2869      * Clear the reply post queue memory.
2870      */
2871     bzero(mpt->m_post_queue, mem_size);
2872
2873     return (DDI_SUCCESS);
2874 }
2875
2876 static void
2877 mptsas_alloc_reply_args(mptsas_t *mpt)
2878 {
2879     if (mpt->m_replayh_args == NULL) {
2880         mpt->m_replayh_args = kmem_zalloc(sizeof (m_replayh_arg_t) *
2881                                         mpt->m_max_replies, KM_SLEEP);
2882     }
2883 }
2884
2885 static int
2886 mptsas_alloc_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)

```

```

2887 {
2888     mptsas_cache_frames_t *frames = NULL;
2889     if (cmd->cmd_extra_frames == NULL) {
2890         frames = kmem_cache_alloc(mpt->m_cache_frames, KM_NOSLEEP);
2891         if (frames == NULL) {
2892             return (DDI_FAILURE);
2893         }
2894         cmd->cmd_extra_frames = frames;
2895     }
2896     return (DDI_SUCCESS);
2897 }

2898 static void
2899 mptsas_free_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd)
2900 {
2901     if (cmd->cmd_extra_frames) {
2902         kmem_cache_free(mpt->m_cache_frames,
2903                         (void *)cmd->cmd_extra_frames);
2904         cmd->cmd_extra_frames = NULL;
2905     }
2906 }

2907 static void
2908 mptsas_cfg_fini(mptsas_t *mpt)
2909 {
2910     NDBG0(("mptsas_cfg_fini"));
2911     ddi_regs_map_free(&mpt->m_datap);
2912 }

2913 static void
2914 mptsas_hba_fini(mptsas_t *mpt)
2915 {
2916     NDBG0(("mptsas_hba_fini"));

2917     /*
2918      * Free up any allocated memory
2919      */
2920     if (mpt->m_dma_req_frame_hdl) {
2921         mptsas_dma_addr_destroy(&mpt->m_dma_req_frame_hdl,
2922                                 &mpt->m_acc_req_frame_hdl);
2923     }

2924     if (mpt->m_dma_req_sense_hdl) {
2925         rmfreemap(mpt->m_erqsense_map);
2926         mptsas_dma_addr_destroy(&mpt->m_dma_req_sense_hdl,
2927                                 &mpt->m_acc_req_sense_hdl);
2928     }

2929     if (mpt->m_dma_reply_frame_hdl) {
2930         mptsas_dma_addr_destroy(&mpt->m_dma_reply_frame_hdl,
2931                                 &mpt->m_acc_reply_frame_hdl);
2932     }

2933     if (mpt->m_dma_free_queue_hdl) {
2934         mptsas_dma_addr_destroy(&mpt->m_dma_free_queue_hdl,
2935                                 &mpt->m_acc_free_queue_hdl);
2936     }

2937     if (mpt->m_dma_post_queue_hdl) {
2938         mptsas_dma_addr_destroy(&mpt->m_dma_post_queue_hdl,
2939                                 &mpt->m_acc_post_queue_hdl);
2940     }

2941     if (mpt->m_replyh_args != NULL) {
2942         kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
2943                   * mpt->m_max_replies);
2944     }

```

```

2945     }
2946 }

2947 static int
2948 mptsas_name_child(dev_info_t *lun_dip, char *name, int len)
2949 {
2950     int lun = 0;
2951     char *sas_wwn = NULL;
2952     int phnum = -1;
2953     int reallen = 0;
2954 }

2955 /* Get the target num */
2956 int
2957 mptsas_name_child(dev_info_t *lun_dip, char *name, int len)
2958 {
2959     int lun = 0;
2960     char *sas_wwn = NULL;
2961     int phnum = -1;
2962     int reallen = 0;
2963

2964     /* Get the target num */
2965     lun = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip, DDI_PROP_DONTPASS,
2966                           LUN_PROP, 0);
2967

2968     if ((phnum = ddi_prop_get_int(DDI_DEV_T_ANY, lun_dip,
2969                                   DDI_PROP_DONTPASS, "sata-phy", -1)) != -1) {
2970         /*
2971          * Stick in the address of form "pPHY,LUN"
2972          */
2973         reallen = snprintf(name, len, "p%u,%u", phnum, lun);
2974     } else if (ddi_prop_lookup_string(DDI_DEV_T_ANY, lun_dip,
2975                                       DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &sas_wwn)
2976                == DDI_PROP_SUCCESS) {
2977         /*
2978          * Stick in the address of the form "wWWN,LUN"
2979          */
2980         reallen = snprintf(name, len, "%s,%u", sas_wwn, lun);
2981     } else {
2982         ddi_prop_free(sas_wwn);
2983     }
2984 }

2985 ASSERT(reallen < len);
2986 if (reallen >= len) {
2987     mptsas_log(0, CE_WARN, "!mptsas_get_name: name parameter "
2988                "length too small, it needs to be %d bytes", reallen + 1);
2989 }
2990 return (DDI_SUCCESS);
2991 }

2992 */

2993 /* tran_tgt_init(9E) - target device instance initialization
2994 */
2995 static int
2996 mptsas_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
2997                       scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
2998 {
2999 #ifndef __lock_lint
3000     _NOTE(ARGUNUSED(hba_tran))
3001 #endif
3002

3003 */

3004     /*
3005      * At this point, the scsi_device structure already exists
3006      * and has been initialized.
3007      *
3008      * Use this function to allocate target-private data structures,
3009      * if needed by this HBA. Add revised flow-control and queue
3010      * properties for child here, if desired and if you can tell they
3011      * support tagged queueing by now.
3012      */
3013     mptsas_t *mpt;
3014     int lun = sd->sd_address.a_lun;
3015     mdi_pathinfo_t *pip = NULL;
3016     mptsas_tgt_private_t *tgt_private = NULL;
3017     mptsas_target_t *ptgt = NULL;
3018

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas

3

```

3019     char *psas_wwn = NULL;
3020     mptsas_phymask_t phymask = 0;
3021     uint64_t sas_wwn = 0;
3022     mptsas_target_addr_t addr;
3023     mpt = SDEV2MPT(sd);

3025     ASSERT(scси_hba_iport_unit_address(hba_dip) != 0);

3027     NDBG0(("mptsas_scси_tgt_init: hbadip=0x%p tgtdip=0x%p lun=%d",
3028            (void *)hba_dip, (void *)tgt_dip, lun));

3029     if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
3030         (void) ndi_merge_node(tgt_dip, mptsas_name_child);
3031         ddi_set_name_addr(tgt_dip, NULL);
3032         return (DDI_FAILURE);
3033     }
3034     /*
3035      * phymask is 0 means the virtual port for RAID
3036      */
3037     phymask = (mptsas_phymask_t)ddi_prop_get_int(DDI_DEV_T_ANY, hba_dip, 0,
3038                                                "phymask", 0);
3039     if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
3040         if ((pip = (void *)sd->sd_private)) == NULL) {
3041             /*
3042              * Very bad news if this occurs. Somehow scси_vhci has
3043              * lost the pathinfo node for this target.
3044              */
3045             return (DDI_NOT_WELL_FORMED);
3046         }
3047     }

3048     if (mdi_prop_lookup_int(pip, LUN_PROP, &lun) !=
3049         DDI_PROP_SUCCESS) {
3050         mptsas_log(mpt, CE_WARN, "Get lun property failed\n");
3051         return (DDI_FAILURE);
3052     }

3053     if (mdi_prop_lookup_string(pip, SCSI_ADDR_PROP_TARGET_PORT,
3054                                &psas_wwn) == MDI_SUCCESS) {
3055         if (scси_wnnstr_to_wwn(psas_wwn, &sas_wwn)) {
3056             sas_wwn = 0;
3057         }
3058         (void) mdi_prop_free(psas_wwn);
3059     }
3060     } else {
3061         lun = ddi_prop_get_int(DDI_DEV_T_ANY, tgt_dip,
3062                               DDI_PROP_DONTPASS, LUN_PROP, 0);
3063         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, tgt_dip,
3064                                   DDI_PROP_DONTPASS, SCSI_ADDR_PROP_TARGET_PORT, &psas_wwn) ==
3065             DDI_PROP_SUCCESS) {
3066             if (scси_wnnstr_to_wwn(psas_wwn, &sas_wwn)) {
3067                 sas_wwn = 0;
3068             }
3069             ddi_prop_free(psas_wwn);
3070         }
3071     } else {
3072         sas_wwn = 0;
3073     }
3074 }

3075 }

3076 ASSERT((sas_wwn != 0) || (phymask != 0));
3077 addr.mta_wwn = sas_wwn;
3078 addr.mta_phymask = phymask;
3079 mutex_enter(&mpt->m_mutex);
3080 ptgt = refresh_lookup(mpt->m_targets, &addr);
3081 mutex_exit(&mpt->m_mutex);
3082 if (ptgt == NULL) {
3083     mptsas_log(mpt, CE_WARN, "!tgt_init: target doesn't exist or "
3084

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.

```

3085         "gone already! phymask:%x, saswwn %"PRIx64, phymask,
3086         sas_wwn);
3087     }
3088     if (hba_tran->tran_tgt_private == NULL) {
3089         tgt_private = kmalloc(sizeof (mptsas_tgt_private_t),
3090                               KM_SLEEP);
3091         tgt_private->t_lun = lun;
3092         tgt_private->t_private = ptgt;
3093         hba_tran->tran_tgt_private = tgt_private;
3094     }
3095
3096     if (mdi_component_is_client(tgt_dip, NULL) == MDI_SUCCESS) {
3097         return (DDI_SUCCESS);
3098     }
3099     mutex_enter(&mpt->m_mutex);
3100
3101     if (ptgt->m_deviceinfo &
3102         (MPI2_SAS_DEVICE_INFO_SATA_DEVICE | 
3103          MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
3104         uchar_t *inq89 = NULL;
3105         int inq89_len = 0x238;
3106         int reallen = 0;
3107         int rval = 0;
3108         struct sata_id *sid = NULL;
3109         char model[SATA_ID_MODEL_LEN + 1];
3110         char fw[SATA_ID_FW_LEN + 1];
3111         char *vid, *pid;
3112         int i;
3113
3114         mutex_exit(&mpt->m_mutex);
3115         /*
3116          * According SCSI/ATA Translation -2 (SAT-2) revision 01a
3117          * chapter 12.4.2 VPD page 89h includes 512 bytes ATA IDENTIFY
3118          * DEVICE data or ATA IDENTIFY PACKET DEVICE data.
3119          */
3120         inq89 = kmalloc(inq89_len, KM_SLEEP);
3121         rval = mptsas_inquiry(mpt, ptgt, 0, 0x89,
3122                               inq89, inq89_len, &reallen, 1);
3123
3124         if (rval != 0) {
3125             if (inq89 != NULL) {
3126                 kmem_free(inq89, inq89_len);
3127             }
3128
3129             mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
3130             "0x89 for SATA target:%x failed!", ptgt->m_devhdl);
3131             return (DDI_SUCCESS);
3132         }
3133         sid = (void *)(&inq89[60]);
3134
3135         swab(sid->ai_model, model, SATA_ID_MODEL_LEN);
3136         swab(sid->ai_fw, fw, SATA_ID_FW_LEN);
3137
3138         model[SATA_ID_MODEL_LEN] = 0;
3139         fw[SATA_ID_FW_LEN] = 0;
3140
3141         /*
3142          * split model into into vid/pid
3143          */
3144         for (i = 0, pid = model; i < SATA_ID_MODEL_LEN; i++, pid++)
3145             if ((*pid == ' ') || (*pid == '\t'))
3146                 break;
3147         if (i < SATA_ID_MODEL_LEN) {
3148             vid = model;
3149             /*

```

```

3151             * terminate vid, establish pid
3152             */
3153             *pid++ = 0;
3154         } else {
3155             /*
3156             * vid will stay "ATA      ", the rule is same
3157             * as sata framework implementation.
3158             */
3159             vid = NULL;
3160             /*
3161             * model is all pid
3162             */
3163             pid = model;
3164         }
3165
3166         /*
3167         * override SCSA "inquiry-*" properties
3168         */
3169         if (vid)
3170             (void) scsi_device_prop_update_inqstring(sd,
3171                                         INQUIRY_VENDOR_ID, vid, strlen(vid));
3172         if (pid)
3173             (void) scsi_device_prop_update_inqstring(sd,
3174                                         INQUIRY_PRODUCT_ID, pid, strlen(pid));
3175         (void) scsi_device_prop_update_inqstring(sd,
3176                                         INQUIRY_REVISION_ID, fw, strlen(fw));
3177
3178         if (inq89 != NULL) {
3179             kmem_free(inq89, inq89_len);
3180         }
3181     } else {
3182         mutex_exit(&mpt->m_mutex);
3183     }
3184
3185     return (DDI_SUCCESS);
3186 }
3187 */
3188 * tran_tgt_free(9E) - target device instance deallocation
3189 */
3190 static void
3191 mptsas_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
3192                       scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
3193 {
3194 #ifndef __lock_lint
3195     _NOTE(ARGUNUSED(hba_dip, tgt_dip, hba_tran, sd))
3196 #endif
3197
3198     mptsas_tgt_private_t *tgt_private = hba_tran->tran_tgt_private;
3199
3200     if (tgt_private != NULL) {
3201         kmem_free(tgt_private, sizeof (mptsas_tgt_private_t));
3202         hba_tran->tran_tgt_private = NULL;
3203     }
3204 }
3205 */
3206 */
3207 * scsi_pkt handling
3208 *
3209 * Visible to the external world via the transport structure.
3210 */
3211 */
3212 */
3213 * Notes:
3214 *   - transport the command to the addressed SCSI target/lun device
3215 *   - normal operation is to schedule the command to be transported,
3216 *     and return TRAN_ACCEPT if this is successful.

```

```

3217     *      - if NO_INTR, tran_start must poll device for command completion
3218     */
3219     static int
3220     mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
3221 {
3222 #ifndef __lock_lint
3223     _NOTE(ARGUNUSED(ap))
3224 #endif
3225     mptsas_t *mpt = PKT2MPT(pkt);
3226     mptsas_cmd_t *cmd = PKT2CMD(pkt);
3227     int rval;
3228     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
3229
3230     NDBG1(("mptsas_scsi_start: pkt=0x%p", (void *)pkt));
3231     ASSERT(ptgt);
3232     if (ptgt == NULL)
3233         return (TRAN_FATAL_ERROR);
3234
3235     /*
3236     * prepare the pkt before taking mutex.
3237     */
3238     rval = mptsas_prepare_pkt(cmd);
3239     if (rval != TRAN_ACCEPT) {
3240         return (rval);
3241     }
3242
3243     /*
3244     * Send the command to target/lun, however your HBA requires it.
3245     * If busy, return TRAN_BUSY; if there's some other formatting error
3246     * in the packet, return TRAN_BADPKT; otherwise, fall through to the
3247     * return of TRAN_ACCEPT.
3248     *
3249     * Remember that access to shared resources, including the mptsas_t
3250     * data structure and the HBA hardware registers, must be protected
3251     * with mutexes, here and everywhere.
3252     *
3253     * Also remember that at interrupt time, you'll get an argument
3254     * to the interrupt handler which is a pointer to your mptsas_t
3255     * structure; you'll have to remember which commands are outstanding
3256     * and which scsi_pkt is the currently-running command so the
3257     * interrupt handler can refer to the pkt to set completion
3258     * status, call the target driver back through pkt_comp, etc.
3259     *
3260     * If the instance lock is held by other thread, don't spin to wait
3261     * for it. Instead, queue the cmd and next time when the instance lock
3262     * is not held, accept all the queued cmd. A extra tx_waitq is
3263     * introduced to protect the queue.
3264     *
3265     * The polled cmd will not be queued and accepted as usual.
3266     *
3267     * Under the tx_waitq mutex, record whether a thread is draining
3268     * the tx_waitq. An IO requesting thread that finds the instance
3269     * mutex contended appends to the tx_waitq and while holding the
3270     * tx_wait mutex, if the draining flag is not set, sets it and then
3271     * proceeds to spin for the instance mutex. This scheme ensures that
3272     * the last cmd in a burst be processed.
3273     *
3274     * we enable this feature only when the helper threads are enabled,
3275     * at which we think the loads are heavy.
3276     *
3277     * per instance mutex m_tx_waitq_mutex is introduced to protect the
3278     * m_tx_waitqtail, m_tx_waitq, m_tx_draining.
3279     */
3280
3281     if (mpt->m_doneq_thread_n) {
3282         if (mutex_tryenter(&mpt->m_mutex) != 0) {

```

```

3283         rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3284         mutex_exit(&mpt->m_mutex);
3285     } else if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3286         mutex_enter(&mpt->m_mutex);
3287         rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3288         mutex_exit(&mpt->m_mutex);
3289     } else {
3290         mutex_enter(&mpt->m_tx_waitq_mutex);
3291         /*
3292          * ptgt->m_dr_flag is protected by m_mutex or
3293          * m_tx_waitq_mutex. In this case, m_tx_waitq_mutex
3294          * is acquired.
3295         */
3296         if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3297             if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3298                 /*
3299                  * The command should be allowed to
3300                  * retry by returning TRAN_BUSY to
3301                  * stall the I/O's which come from
3302                  * scsi_vhci since the device/path is
3303                  * in unstable state now.
3304                 */
3305                 mutex_exit(&mpt->m_tx_waitq_mutex);
3306                 return (TRAN_BUSY);
3307             } else {
3308                 /*
3309                  * The device is offline, just fail the
3310                  * command by returning
3311                  * TRAN_FATAL_ERROR.
3312                 */
3313                 mutex_exit(&mpt->m_tx_waitq_mutex);
3314                 return (TRAN_FATAL_ERROR);
3315             }
3316         }
3317         if (mpt->m_tx_draining) {
3318             cmd->cmd_flags |= CFLAG_TXQ;
3319             *mpt->m_tx_waitqtail = cmd;
3320             mpt->m_tx_waitqtail = &cmd->cmd_linkp;
3321             mutex_exit(&mpt->m_tx_waitq_mutex);
3322         } else { /* drain the queue */
3323             mpt->m_tx_draining = 1;
3324             mutex_exit(&mpt->m_tx_waitq_mutex);
3325             mutex_enter(&mpt->m_mutex);
3326             rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3327             mutex_exit(&mpt->m_mutex);
3328         }
3329     }
3330 } else {
3331     mutex_enter(&mpt->m_mutex);
3332     /*
3333      * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
3334      * in this case, m_mutex is acquired.
3335     */
3336     if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3337         if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3338             /*
3339              * commands should be allowed to retry by
3340              * returning TRAN_BUSY to stall the I/O's
3341              * which come from scsi_vhci since the device/
3342              * path is in unstable state now.
3343             */
3344             mutex_exit(&mpt->m_mutex);
3345             return (TRAN_BUSY);
3346         } else {
3347             /*
3348              * The device is offline, just fail the
3349            */

```

```

3349         * command by returning TRAN_FATAL_ERROR.
3350         */
3351         mutex_exit(&mpt->m_mutex);
3352         return (TRAN_FATAL_ERROR);
3353     }
3354 }
3355 rval = mptsas_accept_pkt(mpt, cmd);
3356 mutex_exit(&mpt->m_mutex);
3357 }

3359 return (rval);
3360 }

3362 /*
3363  * Accept all the queued cmds(if any) before accept the current one.
3364 */
3365 static int
3366 mptsas_accept_txwq_and_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3367 {
3368     int rval;
3369     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

3371     ASSERT(mutex_owned(&mpt->m_mutex));
3372     /*
3373      * The call to mptsas_accept_tx_waitq() must always be performed
3374      * because that is where mpt->m_tx_draining is cleared.
3375     */
3376     mutex_enter(&mpt->m_tx_waitq_mutex);
3377     mptsas_accept_tx_waitq(mpt);
3378     mutex_exit(&mpt->m_tx_waitq_mutex);
3379     /*
3380      * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
3381      * in this case, m_mutex is acquired.
3382     */
3383     if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3384         if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3385             /*
3386              * The command should be allowed to retry by returning
3387              * TRAN_BUSY to stall the I/O's which come from
3388              * scsi_vhci since the device/path is in unstable state
3389              * now.
3390             */
3391             return (TRAN_BUSY);
3392         } else {
3393             /*
3394              * The device is offline, just fail the command by
3395              * return TRAN_FATAL_ERROR.
3396             */
3397         }
3398     }
3399     rval = mptsas_accept_pkt(mpt, cmd);
3400 }

3402 return (rval);
3403 }

3405 static int
3406 mptsas_accept_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3407 {
3408     int rval = TRAN_ACCEPT;
3409     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
3410     NDBG1(("mptsas_accept_pkt: cmd=0x%p", (void *)cmd));
3411     ASSERT(mutex_owned(&mpt->m_mutex));
3412 }
```

```

3415     if ((cmd->cmd_flags & CFLAG_PREPARED) == 0) {
3416         rval = mptsas_prepare_pkt(cmd);
3417         if (rval != TRAN_ACCEPT) {
3418             cmd->cmd_flags &= ~CFLAG_TRANFLAG;
3419             return (rval);
3420         }
3421     }
3422
3423     /*
3424      * reset the throttle if we were draining
3425      */
3426     if ((ptgt->m_t_ncmds == 0) &&
3427         (ptgt->m_t_throttle == DRAIN_THROTTLE)) {
3428         NDBG23(("reset throttle"));
3429         ASSERT(ptgt->m_reset_delay == 0);
3430         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
3431     }
3432
3433     /*
3434      * If HBA is being reset, the DevHandles are being re-initialized,
3435      * which means that they could be invalid even if the target is still
3436      * attached. Check if being reset and if DevHandle is being
3437      * re-initialized. If this is the case, return BUSY so the I/O can be
3438      * retried later.
3439      */
3440     if ((ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) && mpt->m_in_reset) {
3441         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
3442         if (cmd->cmd_flags & CFLAG_TXQ) {
3443             mptsas_doneq_add(mpt, cmd);
3444             mptsas_doneq_empty(mpt);
3445         } else {
3446             return (rval);
3447         }
3448     }
3449
3450     /*
3451      * If device handle has already been invalidated, just
3452      * fail the command. In theory, command from scsi_vhci
3453      * client is impossible send down command with invalid
3454      * devhdl since devhdl is set after path offline, target
3455      * driver is not suppose to select a offlined path.
3456      */
3457     if (ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) {
3458         NDBG3(("rejecting command, it might because invalid devhdl "
3459                "request."));
3460         mptsas_set_pkt_reason(mpt, cmd, CMD_DEV_GONE, STAT_TERMINATED);
3461         if (cmd->cmd_flags & CFLAG_TXQ) {
3462             mptsas_doneq_add(mpt, cmd);
3463             mptsas_doneq_empty(mpt);
3464         } else {
3465             return (rval);
3466         }
3467     }
3468 }
3469
3470 /*
3471  * The first case is the normal case. mpt gets a command from the
3472  * target driver and starts it.
3473  * Since SMID 0 is reserved and the TM slot is reserved, the actual max
3474  * commands is m_max_requests - 2.
3475  */
3476 if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
3477     (ptgt->m_t_throttle > HOLD_THROTTLE) &&
3478     (ptgt->m_t_ncmds < ptgt->m_t_throttle) &&
3479     (ptgt->m_reset_delay == 0) &&
3480     (ptgt->m_t_nwait == 0) &&

```

```

3481     ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0)) {
3482         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
3483             (void) mptsas_start_cmd(mpt, cmd);
3484         } else {
3485             mptsas_waitq_add(mpt, cmd);
3486         }
3487     } else {
3488         /*
3489          * Add this pkt to the work queue
3490          */
3491         mptsas_waitq_add(mpt, cmd);
3492
3493         if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3494             (void) mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);
3495
3496             /*
3497              * Only flush the doneq if this is not a TM
3498              * cmd. For TM cmds the flushing of the
3499              * doneq will be done in those routines.
3500              */
3501             if ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
3502                 mptsas_doneq_empty(mpt);
3503             }
3504         }
3505     }
3506     return (rval);
3507 }
3508
3509 int
3510 mptsas_save_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
3511 {
3512     mptsas_slots_t *slots = mpt->m_active;
3513     uint_t slot, start_rotor;
3514     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
3515
3516     ASSERT(MUTEX_HELD(&mpt->m_mutex));
3517
3518     /*
3519      * Account for reserved TM request slot and reserved SMID of 0.
3520      */
3521     ASSERT(slots->m_n_normal == (mpt->m_max_requests - 2));
3522
3523     /*
3524      * Find the next available slot, beginning at m_rotor. If no slot is
3525      * available, we'll return FALSE to indicate that. This mechanism
3526      * considers only the normal slots, not the reserved slot 0 nor the
3527      * task management slot m_n_normal + 1. The rotor is left to point to
3528      * the normal slot after the one we select, unless we select the last
3529      * normal slot in which case it returns to slot 1.
3530      */
3531     start_rotor = slots->m_rotor;
3532     do {
3533         slot = slots->m_rotor++;
3534         if (slots->m_rotor > slots->m_n_normal)
3535             slots->m_rotor = 1;
3536
3537         if (slots->m_slot[slot] == start_rotor)
3538             break;
3539     } while (slots->m_slot[slot] != NULL);
3540
3541     if (slots->m_slot[slot] != NULL)
3542         return (FALSE);
3543
3544     ASSERT(slot != 0 && slot <= slots->m_n_normal);
3545     cmd->cmd_slot = slot;

```

```

3547     slots->m_slot[slot] = cmd;
3548     mpt->m_ncmds++;
3549
3550     /*
3551      * only increment per target ncmds if this is not a
3552      * command that has no target associated with it (i.e. a
3553      * event acknowledgement)
3554     */
3555     if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
3556         /*
3557          * Expiration time is set in mptsas_start_cmd
3558          */
3559         ptgt->m_t_ncmds++;
3560         cmd->cmd_active_expiration = 0;
3561     } else {
3562         /*
3563          * Initialize expiration time for passthrough commands,
3564          */
3565         cmd->cmd_active_expiration = gethrtime() +
3566             (hrtime_t)cmd->cmd_pkt->pkt_time * NANOSEC;
3567     }
3568     return (TRUE);
3569 }

3570 /*
3571  * prepare the pkt:
3572  * the pkt may have been resubmitted or just reused so
3573  * initialize some fields and do some checks.
3574 */
3575
3576 static int
3577 mptsas_prepare_pkt(mptsas_cmd_t *cmd)
3578 {
3579     struct scsi_pkt *pkt = CMD2PKT(cmd);
3580
3581     NDBG1(("mptsas_prepare_pkt: cmd=0x%p", (void *)cmd));
3582
3583     /*
3584      * Reinitialize some fields that need it; the packet may
3585      * have been resubmitted
3586      */
3587     pkt->pkt_reason = CMD_CMPLT;
3588     pkt->pkt_state = 0;
3589     pkt->pkt_statistics = 0;
3590     pkt->pkt_resid = 0;
3591     cmd->cmd_age = 0;
3592     cmd->cmd_pkt_flags = pkt->pkt_flags;
3593
3594     /*
3595      * zero status byte.
3596      */
3597     *(pkt->pkt_scbp) = 0;
3598
3599     if (cmd->cmd_flags & CFLAG_DMavalid) {
3600         pkt->pkt_resid = cmd->cmd_dmacount;
3601
3602         /*
3603          * consistent packets need to be sync'ed first
3604          * (only for data going out)
3605          */
3606         if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
3607             (cmd->cmd_flags & CFLAG_DMASEND)) {
3608             (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
3609                               DDI_DMA_SYNC_FORDEV);
3610         }
3611     }

```

```

3613     cmd->cmd_flags =
3614         (cmd->cmd_flags & ~(CFLAG_TRANFLAG)) |
3615         CFLAG_PREPARED | CFLAG_IN_TRANSPORT;
3616
3617     return (TRAN_ACCEPT);
3618 }

3619 /*
3620  * tran_init_pkt(9E) - allocate scsi_pkt(9S) for command
3621  */
3622
3623 /* One of three possibilities:
3624  *   - allocate scsi_pkt
3625  *   - allocate scsi_pkt and DMA resources
3626  *   - allocate DMA resources to an already-allocated pkt
3627  */
3628 static struct scsi_pkt *
3629 mptsas_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
3630                      struct buf *bp, int cmdlen, int statuslen, int tgtlen, int flags,
3631                      int (*callback)(), caddr_t arg)
3632 {
3633     mptsas_cmd_t           *cmd, *new_cmd;
3634     mptsas_t                *mpt = ADDR2MPT(ap);
3635     int                     failure = 1;
3636     uint_t                  oldcookiec;
3637     mptsas_target_t         *ptgt = NULL;
3638     int                     rval;
3639     mptsas_tgt_private_t   *tgt_private;
3640     int                     kf;
3641
3642     kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;
3643
3644     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
3645                 tran_tgt_private;
3646     ASSERT(tgt_private != NULL);
3647     if (tgt_private == NULL) {
3648         return (NULL);
3649     }
3650     ptgt = tgt_private->t_private;
3651     ASSERT(ptgt != NULL);
3652     if (ptgt == NULL)
3653         return (NULL);
3654     ap->a_target = ptgt->m_devhdl;
3655     ap->a_lun = tgt_private->t_lun;
3656
3657     ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);
3658 #ifdef MPTSAS_TEST_EXTRN_ALLOC
3659     statuslen *= 100; tgtlen *= 4;
3660 #endif
3661     NDBG3(("mptsas_scsi_init_pkt:\n"
3662            "\tttgt=%d in=0x%p bp=0x%p clen=%d slen=%d tlen=%d flags=%x",
3663            ap->a_target, (void *)pkt, (void *)bp,
3664            cmdlen, statuslen, tgtlen, flags));
3665
3666     /*
3667      * Allocate the new packet.
3668      */
3669     if (pkt == NULL) {
3670         ddi_dma_handle_t        save_dma_handle;
3671
3672         cmd = kmem_cache_alloc(mpt->m_kmem_cache, kf);
3673
3674         if (cmd) {
3675             save_dma_handle = cmd->cmd_dmahandle;
3676             bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3677             cmd->cmd_dmahandle = save_dma_handle;
3678
3679         }
3680     }
3681
3682     if (cmd) {
3683         save_dma_handle = cmd->cmd_dmahandle;
3684         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3685         cmd->cmd_dmahandle = save_dma_handle;
3686
3687     }
3688
3689     if (cmd) {
3690         save_dma_handle = cmd->cmd_dmahandle;
3691         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3692         cmd->cmd_dmahandle = save_dma_handle;
3693
3694     }
3695
3696     if (cmd) {
3697         save_dma_handle = cmd->cmd_dmahandle;
3698         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3699         cmd->cmd_dmahandle = save_dma_handle;
3700
3701     }
3702
3703     if (cmd) {
3704         save_dma_handle = cmd->cmd_dmahandle;
3705         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3706         cmd->cmd_dmahandle = save_dma_handle;
3707
3708     }
3709
3710     if (cmd) {
3711         save_dma_handle = cmd->cmd_dmahandle;
3712         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3713         cmd->cmd_dmahandle = save_dma_handle;
3714
3715     }
3716
3717     if (cmd) {
3718         save_dma_handle = cmd->cmd_dmahandle;
3719         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3720         cmd->cmd_dmahandle = save_dma_handle;
3721
3722     }
3723
3724     if (cmd) {
3725         save_dma_handle = cmd->cmd_dmahandle;
3726         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3727         cmd->cmd_dmahandle = save_dma_handle;
3728
3729     }
3730
3731     if (cmd) {
3732         save_dma_handle = cmd->cmd_dmahandle;
3733         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3734         cmd->cmd_dmahandle = save_dma_handle;
3735
3736     }
3737
3738     if (cmd) {
3739         save_dma_handle = cmd->cmd_dmahandle;
3740         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3741         cmd->cmd_dmahandle = save_dma_handle;
3742
3743     }
3744
3745     if (cmd) {
3746         save_dma_handle = cmd->cmd_dmahandle;
3747         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3748         cmd->cmd_dmahandle = save_dma_handle;
3749
3750     }
3751
3752     if (cmd) {
3753         save_dma_handle = cmd->cmd_dmahandle;
3754         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3755         cmd->cmd_dmahandle = save_dma_handle;
3756
3757     }
3758
3759     if (cmd) {
3760         save_dma_handle = cmd->cmd_dmahandle;
3761         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3762         cmd->cmd_dmahandle = save_dma_handle;
3763
3764     }
3765
3766     if (cmd) {
3767         save_dma_handle = cmd->cmd_dmahandle;
3768         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3769         cmd->cmd_dmahandle = save_dma_handle;
3770
3771     }
3772
3773     if (cmd) {
3774         save_dma_handle = cmd->cmd_dmahandle;
3775         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3776         cmd->cmd_dmahandle = save_dma_handle;
3777
3778     }
3779
3780     if (cmd) {
3781         save_dma_handle = cmd->cmd_dmahandle;
3782         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3783         cmd->cmd_dmahandle = save_dma_handle;
3784
3785     }
3786
3787     if (cmd) {
3788         save_dma_handle = cmd->cmd_dmahandle;
3789         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3790         cmd->cmd_dmahandle = save_dma_handle;
3791
3792     }
3793
3794     if (cmd) {
3795         save_dma_handle = cmd->cmd_dmahandle;
3796         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3797         cmd->cmd_dmahandle = save_dma_handle;
3798
3799     }
3800
3801     if (cmd) {
3802         save_dma_handle = cmd->cmd_dmahandle;
3803         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3804         cmd->cmd_dmahandle = save_dma_handle;
3805
3806     }
3807
3808     if (cmd) {
3809         save_dma_handle = cmd->cmd_dmahandle;
3810         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3811         cmd->cmd_dmahandle = save_dma_handle;
3812
3813     }
3814
3815     if (cmd) {
3816         save_dma_handle = cmd->cmd_dmahandle;
3817         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3818         cmd->cmd_dmahandle = save_dma_handle;
3819
3820     }
3821
3822     if (cmd) {
3823         save_dma_handle = cmd->cmd_dmahandle;
3824         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3825         cmd->cmd_dmahandle = save_dma_handle;
3826
3827     }
3828
3829     if (cmd) {
3830         save_dma_handle = cmd->cmd_dmahandle;
3831         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3832         cmd->cmd_dmahandle = save_dma_handle;
3833
3834     }
3835
3836     if (cmd) {
3837         save_dma_handle = cmd->cmd_dmahandle;
3838         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3839         cmd->cmd_dmahandle = save_dma_handle;
3840
3841     }
3842
3843     if (cmd) {
3844         save_dma_handle = cmd->cmd_dmahandle;
3845         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3846         cmd->cmd_dmahandle = save_dma_handle;
3847
3848     }
3849
3850     if (cmd) {
3851         save_dma_handle = cmd->cmd_dmahandle;
3852         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3853         cmd->cmd_dmahandle = save_dma_handle;
3854
3855     }
3856
3857     if (cmd) {
3858         save_dma_handle = cmd->cmd_dmahandle;
3859         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3860         cmd->cmd_dmahandle = save_dma_handle;
3861
3862     }
3863
3864     if (cmd) {
3865         save_dma_handle = cmd->cmd_dmahandle;
3866         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3867         cmd->cmd_dmahandle = save_dma_handle;
3868
3869     }
3870
3871     if (cmd) {
3872         save_dma_handle = cmd->cmd_dmahandle;
3873         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3874         cmd->cmd_dmahandle = save_dma_handle;
3875
3876     }
3877
3878     if (cmd) {
3879         save_dma_handle = cmd->cmd_dmahandle;
3880         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3881         cmd->cmd_dmahandle = save_dma_handle;
3882
3883     }
3884
3885     if (cmd) {
3886         save_dma_handle = cmd->cmd_dmahandle;
3887         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3888         cmd->cmd_dmahandle = save_dma_handle;
3889
3890     }
3891
3892     if (cmd) {
3893         save_dma_handle = cmd->cmd_dmahandle;
3894         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3895         cmd->cmd_dmahandle = save_dma_handle;
3896
3897     }
3898
3899     if (cmd) {
3900         save_dma_handle = cmd->cmd_dmahandle;
3901         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3902         cmd->cmd_dmahandle = save_dma_handle;
3903
3904     }
3905
3906     if (cmd) {
3907         save_dma_handle = cmd->cmd_dmahandle;
3908         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3909         cmd->cmd_dmahandle = save_dma_handle;
3910
3911     }
3912
3913     if (cmd) {
3914         save_dma_handle = cmd->cmd_dmahandle;
3915         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3916         cmd->cmd_dmahandle = save_dma_handle;
3917
3918     }
3919
3920     if (cmd) {
3921         save_dma_handle = cmd->cmd_dmahandle;
3922         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3923         cmd->cmd_dmahandle = save_dma_handle;
3924
3925     }
3926
3927     if (cmd) {
3928         save_dma_handle = cmd->cmd_dmahandle;
3929         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3930         cmd->cmd_dmahandle = save_dma_handle;
3931
3932     }
3933
3934     if (cmd) {
3935         save_dma_handle = cmd->cmd_dmahandle;
3936         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3937         cmd->cmd_dmahandle = save_dma_handle;
3938
3939     }
3940
3941     if (cmd) {
3942         save_dma_handle = cmd->cmd_dmahandle;
3943         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3944         cmd->cmd_dmahandle = save_dma_handle;
3945
3946     }
3947
3948     if (cmd) {
3949         save_dma_handle = cmd->cmd_dmahandle;
3950         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3951         cmd->cmd_dmahandle = save_dma_handle;
3952
3953     }
3954
3955     if (cmd) {
3956         save_dma_handle = cmd->cmd_dmahandle;
3957         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3958         cmd->cmd_dmahandle = save_dma_handle;
3959
3960     }
3961
3962     if (cmd) {
3963         save_dma_handle = cmd->cmd_dmahandle;
3964         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3965         cmd->cmd_dmahandle = save_dma_handle;
3966
3967     }
3968
3969     if (cmd) {
3970         save_dma_handle = cmd->cmd_dmahandle;
3971         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3972         cmd->cmd_dmahandle = save_dma_handle;
3973
3974     }
3975
3976     if (cmd) {
3977         save_dma_handle = cmd->cmd_dmahandle;
3978         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3979         cmd->cmd_dmahandle = save_dma_handle;
3980
3981     }
3982
3983     if (cmd) {
3984         save_dma_handle = cmd->cmd_dmahandle;
3985         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3986         cmd->cmd_dmahandle = save_dma_handle;
3987
3988     }
3989
3990     if (cmd) {
3991         save_dma_handle = cmd->cmd_dmahandle;
3992         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3993         cmd->cmd_dmahandle = save_dma_handle;
3994
3995     }
3996
3997     if (cmd) {
3998         save_dma_handle = cmd->cmd_dmahandle;
3999         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4000         cmd->cmd_dmahandle = save_dma_handle;
4001
4002     }
4003
4004     if (cmd) {
4005         save_dma_handle = cmd->cmd_dmahandle;
4006         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4007         cmd->cmd_dmahandle = save_dma_handle;
4008
4009     }
4010
4011     if (cmd) {
4012         save_dma_handle = cmd->cmd_dmahandle;
4013         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4014         cmd->cmd_dmahandle = save_dma_handle;
4015
4016     }
4017
4018     if (cmd) {
4019         save_dma_handle = cmd->cmd_dmahandle;
4020         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4021         cmd->cmd_dmahandle = save_dma_handle;
4022
4023     }
4024
4025     if (cmd) {
4026         save_dma_handle = cmd->cmd_dmahandle;
4027         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4028         cmd->cmd_dmahandle = save_dma_handle;
4029
4030     }
4031
4032     if (cmd) {
4033         save_dma_handle = cmd->cmd_dmahandle;
4034         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4035         cmd->cmd_dmahandle = save_dma_handle;
4036
4037     }
4038
4039     if (cmd) {
4040         save_dma_handle = cmd->cmd_dmahandle;
4041         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4042         cmd->cmd_dmahandle = save_dma_handle;
4043
4044     }
4045
4046     if (cmd) {
4047         save_dma_handle = cmd->cmd_dmahandle;
4048         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4049         cmd->cmd_dmahandle = save_dma_handle;
4050
4051     }
4052
4053     if (cmd) {
4054         save_dma_handle = cmd->cmd_dmahandle;
4055         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4056         cmd->cmd_dmahandle = save_dma_handle;
4057
4058     }
4059
4060     if (cmd) {
4061         save_dma_handle = cmd->cmd_dmahandle;
4062         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4063         cmd->cmd_dmahandle = save_dma_handle;
4064
4065     }
4066
4067     if (cmd) {
4068         save_dma_handle = cmd->cmd_dmahandle;
4069         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4070         cmd->cmd_dmahandle = save_dma_handle;
4071
4072     }
4073
4074     if (cmd) {
4075         save_dma_handle = cmd->cmd_dmahandle;
4076         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4077         cmd->cmd_dmahandle = save_dma_handle;
4078
4079     }
4080
4081     if (cmd) {
4082         save_dma_handle = cmd->cmd_dmahandle;
4083         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4084         cmd->cmd_dmahandle = save_dma_handle;
4085
4086     }
4087
4088     if (cmd) {
4089         save_dma_handle = cmd->cmd_dmahandle;
4090         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4091         cmd->cmd_dmahandle = save_dma_handle;
4092
4093     }
4094
4095     if (cmd) {
4096         save_dma_handle = cmd->cmd_dmahandle;
4097         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4098         cmd->cmd_dmahandle = save_dma_handle;
4099
4100     }
4101
4102     if (cmd) {
4103         save_dma_handle = cmd->cmd_dmahandle;
4104         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4105         cmd->cmd_dmahandle = save_dma_handle;
4106
4107     }
4108
4109     if (cmd) {
4110         save_dma_handle = cmd->cmd_dmahandle;
4111         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4112         cmd->cmd_dmahandle = save_dma_handle;
4113
4114     }
4115
4116     if (cmd) {
4117         save_dma_handle = cmd->cmd_dmahandle;
4118         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4119         cmd->cmd_dmahandle = save_dma_handle;
4120
4121     }
4122
4123     if (cmd) {
4124         save_dma_handle = cmd->cmd_dmahandle;
4125         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4126         cmd->cmd_dmahandle = save_dma_handle;
4127
4128     }
4129
4130     if (cmd) {
4131         save_dma_handle = cmd->cmd_dmahandle;
4132         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4133         cmd->cmd_dmahandle = save_dma_handle;
4134
4135     }
4136
4137     if (cmd) {
4138         save_dma_handle = cmd->cmd_dmahandle;
4139         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4140         cmd->cmd_dmahandle = save_dma_handle;
4141
4142     }
4143
4144     if (cmd) {
4145         save_dma_handle = cmd->cmd_dmahandle;
4146         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4147         cmd->cmd_dmahandle = save_dma_handle;
4148
4149     }
4150
4151     if (cmd) {
4152         save_dma_handle = cmd->cmd_dmahandle;
4153         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4154         cmd->cmd_dmahandle = save_dma_handle;
4155
4156     }
4157
4158     if (cmd) {
4159         save_dma_handle = cmd->cmd_dmahandle;
4160         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4161         cmd->cmd_dmahandle = save_dma_handle;
4162
4163     }
4164
4165     if (cmd) {
4166         save_dma_handle = cmd->cmd_dmahandle;
4167         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4168         cmd->cmd_dmahandle = save_dma_handle;
4169
4170     }
4171
4172     if (cmd) {
4173         save_dma_handle = cmd->cmd_dmahandle;
4174         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4175         cmd->cmd_dmahandle = save_dma_handle;
4176
4177     }
4178
4179     if (cmd) {
4180         save_dma_handle = cmd->cmd_dmahandle;
4181         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4182         cmd->cmd_dmahandle = save_dma_handle;
4183
4184     }
4185
4186     if (cmd) {
4187         save_dma_handle = cmd->cmd_dmahandle;
4188         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4189         cmd->cmd_dmahandle = save_dma_handle;
4190
4191     }
4192
4193     if (cmd) {
4194         save_dma_handle = cmd->cmd_dmahandle;
4195         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4196         cmd->cmd_dmahandle = save_dma_handle;
4197
4198     }
4199
4200     if (cmd) {
4201         save_dma_handle = cmd->cmd_dmahandle;
4202         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4203         cmd->cmd_dmahandle = save_dma_handle;
4204
4205     }
4206
4207     if (cmd) {
4208         save_dma_handle = cmd->cmd_dmahandle;
4209         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4210         cmd->cmd_dmahandle = save_dma_handle;
4211
4212     }
4213
4214     if (cmd) {
4215         save_dma_handle = cmd->cmd_dmahandle;
4216         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4217         cmd->cmd_dmahandle = save_dma_handle;
4218
4219     }
4220
4221     if (cmd) {
4222         save_dma_handle = cmd->cmd_dmahandle;
4223         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4224         cmd->cmd_dmahandle = save_dma_handle;
4225
4226     }
4227
4228     if (cmd) {
4229         save_dma_handle = cmd->cmd_dmahandle;
4230         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4231         cmd->cmd_dmahandle = save_dma_handle;
4232
4233     }
4234
4235     if (cmd) {
4236         save_dma_handle = cmd->cmd_dmahandle;
4237         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4238         cmd->cmd_dmahandle = save_dma_handle;
4239
4240     }
4241
4242     if (cmd) {
4243         save_dma_handle = cmd->cmd_dmahandle;
4244         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4245         cmd->cmd_dmahandle = save_dma_handle;
4246
4247     }
4248
4249     if (cmd) {
4250         save_dma_handle = cmd->cmd_dmahandle;
4251         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4252         cmd->cmd_dmahandle = save_dma_handle;
4253
4254     }
4255
4256     if (cmd) {
4257         save_dma_handle = cmd->cmd_dmahandle;
4258         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4259         cmd->cmd_dmahandle = save_dma_handle;
4260
4261     }
4262
4263     if (cmd) {
4264         save_dma_handle = cmd->cmd_dmahandle;
4265         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4266         cmd->cmd_dmahandle = save_dma_handle;
4267
4268     }
4269
4270     if (cmd) {
4271         save_dma_handle = cmd->cmd_dmahandle;
4272         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4273         cmd->cmd_dmahandle = save_dma_handle;
4274
4275     }
4276
4277     if (cmd) {
4278         save_dma_handle = cmd->cmd_dmahandle;
4279         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4280         cmd->cmd_dmahandle = save_dma_handle;
4281
4282     }
4283
4284     if (cmd) {
4285         save_dma_handle = cmd->cmd_dmahandle;
4286         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4287         cmd->cmd_dmahandle = save_dma_handle;
4288
4289     }
4290
4291     if (cmd) {
4292         save_dma_handle = cmd->cmd_dmahandle;
4293         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4294         cmd->cmd_dmahandle = save_dma_handle;
4295
4296     }
4297
4298     if (cmd) {
4299         save_dma_handle = cmd->cmd_dmahandle;
4300         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4301         cmd->cmd_dmahandle = save_dma_handle;
4302
4303     }
4304
4305     if (cmd) {
4306         save_dma_handle = cmd->cmd_dmahandle;
4307         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4308         cmd->cmd_dmahandle = save_dma_handle;
4309
4310     }
4311
4312     if (cmd) {
4313         save_dma_handle = cmd->cmd_dmahandle;
4314         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4315         cmd->cmd_dmahandle = save_dma_handle;
4316
4317     }
4318
4319     if (cmd) {
4320         save_dma_handle = cmd->cmd_dmahandle;
4321         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4322         cmd->cmd_dmahandle = save_dma_handle;
4323
4324     }
4325
4326     if (cmd) {
4327         save_dma_handle = cmd->cmd_dmahandle;
4328         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4329         cmd->cmd_dmahandle = save_dma_handle;
4330
4331     }
4332
4333     if (cmd) {
4334         save_dma_handle = cmd->cmd_dmahandle;
4335         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4336         cmd->cmd_dmahandle = save_dma_handle;
4337
4338     }
4339
4340     if (cmd) {
4341         save_dma_handle = cmd->cmd_dmahandle;
4342         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4343         cmd->cmd_dmahandle = save_dma_handle;
4344
4345     }
4346
4347     if (cmd) {
4348         save_dma_handle = cmd->cmd_dmahandle;
4349         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4350         cmd->cmd_dmahandle = save_dma_handle;
4351
4352     }
4353
4354     if (cmd) {
4355         save_dma_handle = cmd->cmd_dmahandle;
4356         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4357         cmd->cmd_dmahandle = save_dma_handle;
4358
4359     }
4360
4361     if (cmd) {
4362         save_dma_handle = cmd->cmd_dmahandle;
4363         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4364         cmd->cmd_dmahandle = save_dma_handle;
4365
4366     }
4367
4368     if (cmd) {
4369         save_dma_handle = cmd->cmd_dmahandle;
4370         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4371         cmd->cmd_dmahandle = save_dma_handle;
4372
4373     }
4374
4375     if (cmd) {
4376         save_dma_handle = cmd->cmd_dmahandle;
4377         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4378         cmd->cmd_dmahandle = save_dma_handle;
4379
4380     }
4381
4382     if (cmd) {
4383         save_dma_handle = cmd->cmd_dmahandle;
4384         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4385         cmd->cmd_dmahandle = save_dma_handle;
4386
4387     }
4388
4389     if (cmd) {
4390         save_dma_handle = cmd->cmd_dmahandle;
4391         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4392         cmd->cmd_dmahandle = save_dma_handle;
4393
4394     }
4395
4396     if (cmd) {
4397         save_dma_handle = cmd->cmd_dmahandle;
4398         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4399         cmd->cmd_dmahandle = save_dma_handle;
4400
4401     }
4402
4403     if (cmd) {
4404         save_dma_handle = cmd->cmd_dmahandle;
4405         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4406         cmd->cmd_dmahandle = save_dma_handle;
4407
4408     }
4409
4410     if (cmd) {
4411         save_dma_handle = cmd->cmd_dmahandle;
4412         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4413         cmd->cmd_dmahandle = save_dma_handle;
4414
4415     }
4416
4417     if (cmd) {
4418         save_dma_handle = cmd->cmd_dmahandle;
4419         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4420         cmd->cmd_dmahandle = save_dma_handle;
4421
4422     }
4423
4424     if (cmd) {
4425         save_dma_handle = cmd->cmd_dmahandle;
4426         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4427         cmd->cmd_dmahandle = save_dma_handle;
4428
4429     }
4430
4431     if (cmd) {
4432         save_dma_handle = cmd->cmd_dmahandle;
4433         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4434         cmd->cmd_dmahandle = save_dma_handle;
4435
4436     }
4437
4438     if (cmd) {
4439         save_dma_handle = cmd->cmd_dmahandle;
4440         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4441         cmd->cmd_dmahandle = save_dma_handle;
4442
4443     }
4444
4445     if (cmd) {
4446         save_dma_handle = cmd->cmd_dmahandle;
4447         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4448         cmd->cmd_dmahandle = save_dma_handle;
4449
4450     }
4451
4452     if (cmd) {
4453         save_dma_handle = cmd->cmd_dmahandle;
4454         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4455         cmd->cmd_dmahandle = save_dma_handle;
4456
4457     }
4458
4459     if (cmd) {
4460         save_dma_handle = cmd->cmd_dmahandle;
4461         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4462         cmd->cmd_dmahandle = save_dma_handle;
4463
4464     }
4465
4466     if (cmd) {
4467         save_dma_handle = cmd->cmd_dmahandle;
4468         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4469         cmd->cmd_dmahandle = save_dma_handle;
4470
4471     }
4472
4473     if (cmd) {
4474         save_dma_handle = cmd->cmd_dmahandle;
4475         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4476         cmd->cmd_dmahandle = save_dma_handle;
4477
4478     }
4479
4480     if (cmd) {
4481         save_dma_handle = cmd->cmd_dmahandle;
4482         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4483         cmd->cmd_dmahandle = save_dma_handle;
4484
4485     }
4486
4487     if (cmd) {
4488         save_dma_handle = cmd->cmd_dmahandle;
4489         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4490         cmd->cmd_dmahandle = save_dma_handle;
4491
4492     }
4493
4494     if (cmd) {
4495         save_dma_handle = cmd->cmd_dmahandle;
4496         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
4497         cmd->cmd_dmahandle = save_dma_handle;
4498
4499     }
45
```

```

3679         pkt = (void *)((uchar_t *)cmd +
3680                         sizeof (struct mptsas_cmd));
3681         pkt->pkt_ha_private = (opaque_t)cmd;
3682         pkt->pkt_address = *ap;
3683         pkt->pkt_private = (opaque_t)cmd->cmd_pkt_private;
3684         pkt->pkt_scbp = (opaque_t)&cmd->cmd_scb;
3685         cmd->cmd_cdbp = (opaque_t)&cmd->cmd_cdb;
3686         cmd->cmd_pkt = (struct scsi_pkt *)pkt;
3687         cmd->cmd_cdblen = (uchar_t)cmdlen;
3688         cmd->cmd_rqrlen = statuslen;
3689         cmd->cmd_length = SENSE_LENGTH;
3690         cmd->cmd_tgt_addr = ptgt;
3691         failure = 0;
3692     }
3693
3694     if (failure || (cmdlen > sizeof (cmd->cmd_cdb)) ||
3695         (tgtlen > PKT_PRIV_LEN) ||
3696         (statuslen > EXTCMDS_STATUS_SIZE)) {
3697         if (failure == 0) {
3698             /*
3699             * if extern alloc fails, all will be
3700             * deallocated, including cmd
3701             */
3702             failure = mptsas_pkt_alloc_extern(mpt, cmd,
3703                                              cmdlen, tgtlen, statuslen, kf);
3704         }
3705         if (failure) {
3706             /*
3707             * if extern allocation fails, it will
3708             * deallocate the new pkt as well
3709             */
3710             return (NULL);
3711         }
3712     }
3713     new_cmd = cmd;
3714
3715 } else {
3716     cmd = PKT2CMD(pkt);
3717     new_cmd = NULL;
3718 }
3719
3720 /* grab cmd->cmd_cookiec here as oldcookiec */
3721 oldcookiec = cmd->cmd_cookiec;
3722
3723 /*
3724 * If the dma was broken up into PARTIAL transfers cmd_nwin will be
3725 * greater than 0 and we'll need to grab the next dma window
3726 */
3727 /*
3728 * SLM-not doing extra command frame right now; may add later
3729 */
3730
3731 if (cmd->cmd_nwin > 0) {
3732
3733     /*
3734     * Make sure we havn't gone past the the total number
3735     * of windows
3736     */
3737     if (++cmd->cmd_winindex >= cmd->cmd_nwin) {
3738         return (NULL);
3739     }
3740     if (ddi_dma_getwin(cmd->cmd_dmahandle, cmd->cmd_winindex,
3741                        &cmd->cmd_dma_offset, &cmd->cmd_dma_len,
3742                        &cmd->cmd_cookie, &cmd->cmd_cookiec) == DDI_FAILURE) {

```

```

3745                     return (NULL);
3746                 }
3747             goto get_dma_cookies;
3748         }
3749
3750         if (flags & PKT_XARQ) {
3751             cmd->cmd_flags |= CFLAG_XARQ;
3752         }
3753
3754         /*
3755          * DMA resource allocation. This version assumes your
3756          * HBA has some sort of bus-mastering or onboard DMA capability, with a
3757          * scatter-gather list of length MPTSAS_MAX_DMA_SEGS, as given in the
3758          * ddi_dma_attr_t structure and passed to scsi_impl_dmaget.
3759          */
3760         if (bp && (bp->b_bcount != 0) &&
3761             (cmd->cmd_flags & CFLAG_DMavalid) == 0) {
3762
3763             int cnt, dma_flags;
3764             mptti_t *dmap; /* ptr to the S/G list */
3765
3766             /*
3767              * Set up DMA memory and position to the next DMA segment.
3768              */
3769             ASSERT(cmd->cmd_dmahandle != NULL);
3770
3771             if (bp->b_flags & B_READ) {
3772                 dma_flags = DDI_DMA_READ;
3773                 cmd->cmd_flags &= ~CFLAG_DMASEND;
3774             } else {
3775                 dma_flags = DDI_DMA_WRITE;
3776                 cmd->cmd_flags |= CFLAG_DMASEND;
3777             }
3778             if (flags & PKT_CONSISTENT) {
3779                 cmd->cmd_flags |= CFLAG_CMDIOPB;
3780                 dma_flags |= DDI_DMA_CONSISTENT;
3781             }
3782
3783             if (flags & PKT_DMA_PARTIAL) {
3784                 dma_flags |= DDI_DMA_PARTIAL;
3785             }
3786
3787             /*
3788              * workaround for byte hole issue on psycho and
3789              * schizo pre 2.1
3790              */
3791             if ((bp->b_flags & B_READ) && ((bp->b_flags &
3792                                             (B_PAGEIO|B_REMAPPED)) != B_PAGEIO) &&
3793                 ((uintptr_t)bp->b_un.b_addr & 0x7)) {
3794                 dma_flags |= DDI_DMA_CONSISTENT;
3795             }
3796
3797             rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
3798                                           dma_flags, callback, arg,
3800                                           &cmd->cmd_cookie, &cmd->cmd_cookiec);
3801             if (rval == DDI_DMA_PARTIAL_MAP) {
3802                 (void) ddi_dma_numwin(cmd->cmd_dmahandle,
3803                                       &cmd->cmd_nwin);
3804                 cmd->cmd_winindex = 0;
3805                 (void) ddi_dma_getwin(cmd->cmd_dmahandle,
3806                                       cmd->cmd_winindex, &cmd->cmd_dma_offset,
3807                                       &cmd->cmd_dma_len, &cmd->cmd_cookie,
3808                                       &cmd->cmd_cookiec);
3809             } else if (rval && (rval != DDI_DMA_MAPPED)) {
3810                 switch (rval) {

```

```

3811             case DDI_DMA_NORESOURCES:
3812                 bioerror(bp, 0);
3813                 break;
3814             case DDI_DMA_BADATTR:
3815             case DDI_DMA_NOMAPPING:
3816                 bioerror(bp, EFAULT);
3817                 break;
3818             case DDI_DMA_TOOBIG:
3819             default:
3820                 bioerror(bp, EINVAL);
3821                 break;
3822         }
3823         cmd->cmd_flags &= ~CFLAG_DMAVALID;
3824         if (new_cmd) {
3825             mptsas_scsi_destroy_pkt(ap, pkt);
3826         }
3827     }
3828 }

3830 get_dma_cookies:
3831     cmd->cmd_flags |= CFLAG_DMAVALID;
3832     ASSERT(cmd->cmd_cookiec > 0);

3833     if (cmd->cmd_cookiec > MPTSAS_MAX_CMD_SEGS) {
3834         mptsas_log(mpt, CE_NOTE, "large cookiec received %d\n",
3835                     cmd->cmd_cookiec);
3836         bioerror(bp, EINVAL);
3837         if (new_cmd) {
3838             mptsas_scsi_destroy_pkt(ap, pkt);
3839         }
3840     }
3841     return ((struct scsi_pkt *)NULL);
3842 }

3843 /*
3844 * Allocate extra SGL buffer if needed.
3845 */
3846 if ((cmd->cmd_cookiec > MPTSAS_MAX_FRAME_SGES64(mpt)) &&
3847     (cmd->cmd_extra_frames == NULL)) {
3848     if (mptsas_alloc_extra_sgl_frame(mpt, cmd) ==
3849         DDI_FAILURE) {
3850         mptsas_log(mpt, CE_WARN, "MPT SGL mem alloc "
3851                     "failed");
3852         bioerror(bp, ENOMEM);
3853         if (new_cmd) {
3854             mptsas_scsi_destroy_pkt(ap, pkt);
3855         }
3856     }
3857     return ((struct scsi_pkt *)NULL);
3858 }
3859 }

3860 /*
3861 * Always use scatter-gather transfer
3862 * Use the loop below to store physical addresses of
3863 * DMA segments, from the DMA cookies, into your HBA's
3864 * scatter-gather list.
3865 * We need to ensure we have enough kmem alloc'd
3866 * for the sg entries since we are no longer using an
3867 * array inside mptsas_cmd_t.
3868 *
3869 * We check cmd->cmd_cookiec against oldcookiec so
3870 * the scatter-gather list is correctly allocated
3871 */
3872
3873 if (oldcookiec != cmd->cmd_cookiec) {
3874     if (cmd->cmd_sg != (mptti_t *)NULL) {
3875         kmem_free(cmd->cmd_sg, sizeof (mptti_t) *

```

```

3877                                     oldcookiec);
3878         cmd->cmd_sg = NULL;
3879     }
3880 }

3881 if (cmd->cmd_sg == (mptti_t *)NULL) {
3882     cmd->cmd_sg = kmem_alloc(sizeof (size_t)*(sizeof (mptti_t)*
3883                               cmd->cmd_cookiec), kf);
3884 }

3885 if (cmd->cmd_sg == (mptti_t *)NULL) {
3886     mptsas_log(mpt, CE_WARN,
3887                 "unable to kmem_alloc enough memory "
3888                 "for scatter/gather list");
3889 */
3890 /* if we have an ENOMEM condition we need to behave
3891 * the same way as the rest of this routine
3892 */
3893

3894 bioerror(bp, ENOMEM);
3895 if (new_cmd) {
3896     mptsas_scsi_destroy_pkt(ap, pkt);
3897 }
3898 return ((struct scsi_pkt *)NULL);
3899 }

3900 dmap = cmd->cmd_sg;
3901 ASSERT(cmd->cmd_cookie.dmac_size != 0);

3902 /*
3903 * store the first segment into the S/G list
3904 */
3905 dmap->count = cmd->cmd_cookie.dmac_size;
3906 dmap->addr.address64.Low = (uint32_t)
3907     (cmd->cmd_cookie.dmac_laddress & 0xfffffffffull);
3908 dmap->addr.address64.High = (uint32_t)
3909     (cmd->cmd_cookie.dmac_laddress >> 32);

3910 /*
3911 * dmacount counts the size of the dma for this window
3912 * (if partial dma is being used). totaldmaccount
3913 * keeps track of the total amount of dma we have
3914 * transferred for all the windows (needed to calculate
3915 * the resid value below).
3916 */
3917 cmd->cmd_dmacount = cmd->cmd_cookie.dmac_size;
3918 cmd->cmd_totaldmaccount += cmd->cmd_cookie.dmac_size;

3919 /*
3920 * We already stored the first DMA scatter gather segment,
3921 * start at 1 if we need to store more.
3922 */
3923 for (cnt = 1; cnt < cmd->cmd_cookiec; cnt++) {
3924     /*
3925      * Get next DMA cookie
3926      */
3927     ddi_dma_nextcookie(cmd->cmd_dmahandle,
3928                         &cmd->cmd_cookie);
3929     dmap++;

3930     cmd->cmd_dmacount += cmd->cmd_cookie.dmac_size;
3931     cmd->cmd_totaldmaccount += cmd->cmd_cookie.dmac_size;

3932     /*
3933      * store the segment parms into the S/G list
3934      */
3935
3936
3937
3938
3939
3940
3941
3942

```

```

3943             */
3944             dmap->count = cmd->cmd_cookie.dmac_size;
3945             dmap->addr.address64.Low = (uint32_t)
3946                 (cmd->cmd_cookie.dmac_laddress & 0xfffffffffull);
3947             dmap->addr.address64.High = (uint32_t)
3948                 (cmd->cmd_cookie.dmac_laddress >> 32);
3949         }
3950
3951         /*
3952          * If this was partially allocated we set the resid
3953          * the amount of data NOT transferred in this window
3954          * If there is only one window, the resid will be 0
3955          */
3956         pkt->pkt_resid = (bp->b_bcount - cmd->cmd_totaldmacount);
3957         NDBG3(("mptsas_scси_init_pkt: cmd_dmacount=%d.",
3958                cmd->cmd_dmacount));
3959     }
3960     return (pkt);
3961 }
3962 */
3963 * tran_destroy_pkt(9E) - scsi_pkt(9s) deallocation
3964 *
3965 * Notes:
3966 *   - also frees DMA resources if allocated
3967 *   - implicit DMA synchronization
3968 */
3969 static void
3970 mptsas_scси_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
3971 {
3972     mptsas_cmd_t      *cmd = PKT2CMD(pkt);
3973     mptsas_t           *mpt = ADDR2MPT(ap);
3974
3975     NDBG3(("mptsas_scси_destroy_pkt: target=%d pkt=0x%p",
3976            ap->a_target, (void *)pkt));
3977
3978     if (cmd->cmd_flags & CFLAG_DMavalid) {
3979         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
3980         cmd->cmd_flags &= ~CFLAG_DMavalid;
3981     }
3982
3983     if (cmd->cmd_sg) {
3984         kmem_free(cmd->cmd_sg, sizeof (mptti_t) * cmd->cmd_cookiec);
3985         cmd->cmd_sg = NULL;
3986     }
3987
3988     mptsas_free_extra_sgl_frame(mpt, cmd);
3989
3990     if ((cmd->cmd_flags &
3991         (CFLAG_FREE | CFLAG_CDBEXTERN | CFLAG_PRIVEXTERN |
3992          CFLAG_SCBEXTERN)) == 0) {
3993         cmd->cmd_flags = CFLAG_FREE;
3994         kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
3995     } else {
3996         mptsas_pkt_destroy_extern(mpt, cmd);
3997     }
3998 }
3999 */
4000 */
4001 * kmem cache constructor and destructor:
4002 * When constructing, we bzero the cmd and allocate the dma handle
4003 * When destructing, just free the dma handle
4004 */
4005 static int
4006 mptsas_kmem_cache_constructor(void *buf, void *cdrarg, int kmflags)
4007 {

```

```

4009     mptsas_cmd_t      *cmd = buf;
4010     mptsas_t           *mpt = cdrarg;
4011     int                  (*callback)(caddr_t);
4012
4013     callback = (kmflags == KM_SLEEP)? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;
4014
4015     NDBG4(("mptsas_kmem_cache_constructor"));
4016
4017     /*
4018      * allocate a dma handle
4019      */
4020     if ((ddi_dma_alloc_handle(mpt->m_dip, &mpt->m_io_dma_attr, callback,
4021                               NULL, &cmd->cmd_dmahandle)) != DDI_SUCCESS) {
4022         cmd->cmd_dmahandle = NULL;
4023         return (-1);
4024     }
4025     return (0);
4026 }
4027
4028 static void
4029 mptsas_kmem_cache_destructor(void *buf, void *cdrarg)
4030 {
4031 #ifndef _lock_lint
4032     _NOTE(ARGUNUSED(cdrarg))
4033 #endif
4034     mptsas_cmd_t      *cmd = buf;
4035
4036     NDBG4(("mptsas_kmem_cache_destructor"));
4037
4038     if (cmd->cmd_dmahandle) {
4039         ddi_dma_free_handle(&cmd->cmd_dmahandle);
4040         cmd->cmd_dmahandle = NULL;
4041     }
4042 }
4043
4044 static int
4045 mptsas_cache_frames_constructor(void *buf, void *cdrarg, int kmflags)
4046 {
4047     mptsas_cache_frames_t  *p = buf;
4048     mptsas_t           *mpt = cdrarg;
4049     ddi_dma_attr_t        frame_dma_attr;
4050     size_t                  mem_size, alloc_len;
4051     ddi_dma_cookie_t       cookie;
4052     uint_t                  ncookie;
4053     int (*callback)(caddr_t) = (kmflags == KM_SLEEP)?
4054                             ? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;
4055
4056     frame_dma_attr = mpt->m_msg_dma_attr;
4057     frame_dma_attr.dma_attr_align = 0x10;
4058     frame_dma_attr.dma_attr_sgllen = 1;
4059
4060     if (ddi_dma_alloc_handle(mpt->m_dip, &frame_dma_attr, callback, NULL,
4061                               &p->m_dma_hdl) != DDI_SUCCESS) {
4062         mptsas_log(mpt, CE_WARN, "Unable to allocate dma handle for"
4063                    " extra SGL.");
4064         return (DDI_FAILURE);
4065     }
4066
4067     mem_size = (mpt->m_max_request_frames - 1) * mpt->m_req_frame_size;
4068
4069     if (ddi_dma_mem_alloc(p->m_dma_hdl, mem_size, &mpt->m_dev_acc_attr,
4070                           DDI_DMA_CONSISTENT, callback, NULL, (caddr_t *)p->m_frames_addr,
4071                           &alloc_len, &p->m_acc_hdl) != DDI_SUCCESS) {
4072         ddi_dma_free_handle(&p->m_dma_hdl);
4073         p->m_dma_hdl = NULL;
4074         mptsas_log(mpt, CE_WARN, "Unable to allocate dma memory for"
4075                    " extra SGL.");
4076     }

```

```

4075         " extra SGL.");
4076     return (DDI_FAILURE);
4077 }
4078
4079 if (ddi_dma_addr_bind_handle(p->m_dma_hdl, NULL, p->m_frames_addr,
4080     alloc_len, DDI_DMA_RDWR | DDI_DMA_CONSISTENT, callback, NULL,
4081     &cookie, &ncookie) != DDI_DMA_MAPPED) {
4082     (void) ddi_dma_mem_free(&p->m_acc_hdl);
4083     ddi_dma_free_handle(&p->m_dma_hdl);
4084     p->m_dma_hdl = NULL;
4085     mptsas_log(mpt, CE_WARN, "Unable to bind DMA resources for"
4086     " extra SGL");
4087     return (DDI_FAILURE);
4088 }
4089
4090 /*
4091 * Store the SGL memory address. This chip uses this
4092 * address to dma to and from the driver. The second
4093 * address is the address mpt uses to fill in the SGL.
4094 */
4095 p->m_phys_addr = cookie.dmac_laddress;
4096
4097 return (DDI_SUCCESS);
4098 }
4099
4100 static void
4101 mptsas_cache_frames_destructor(void *buf, void *cdrarg)
4102 {
4103 #ifndef __lock_lint
4104     _NOTE(ARGUNUSED(cdrarg))
4105 #endif
4106     mptsas_cache_frames_t *p = buf;
4107     if (p->m_dma_hdl != NULL) {
4108         (void) ddi_dma_unbind_handle(p->m_dma_hdl);
4109         (void) ddi_dma_mem_free(&p->m_acc_hdl);
4110         ddi_dma_free_handle(&p->m_dma_hdl);
4111         p->m_phys_addr = NULL;
4112         p->m_frames_addr = NULL;
4113         p->m_dma_hdl = NULL;
4114         p->m_acc_hdl = NULL;
4115     }
4116
4117 }
4118
4119 /*
4120 * Figure out if we need to use a different method for the request
4121 * sense buffer and allocate from the map if necessary.
4122 */
4123 static boolean_t
4124 mptsas_cmddargsize(mptsas_t *mpt, mptsas_cmd_t *cmd, size_t senselength, int kf)
4125 {
4126     if (senselength > mpt->m_req_sense_size) {
4127         unsigned long i;
4128
4129         /* Sense length is limited to an 8 bit value in MPI Spec. */
4130         if (senselength > 255)
4131             senselength = 255;
4132         cmd->cmd_extrqschunks = (senselength +
4133             (mpt->m_req_sense_size - 1))/mpt->m_req_sense_size;
4134         i = (kf == KM_SLEEP ? rmalloc_wait : rmalloc)
4135             (mpt->m_ergsense_map, cmd->cmd_extrqschunks);
4136
4137         if (i == 0)
4138             return (B_FALSE);
4139
4140     cmd->cmd_extrqslen = (uint16_t)senselength;

```

```

4141         cmd->cmd_extrqsidx = i - 1;
4142         cmd->cmd_arg_buf = mpt->m_extreq_sense +
4143             (cmd->cmd_extrqsidx * mpt->m_req_sense_size);
4144     } else {
4145         cmd->cmd_rqslen = (uchar_t)senselength;
4146     }
4147
4148     return (B_TRUE);
4149 }
4150
4151 /*
4152 * allocate and deallocate external pkt space (ie. not part of mptsas_cmd)
4153 * for non-standard length cdb, pkt_private, status areas
4154 * if allocation fails, then deallocate all external space and the pkt
4155 */
4156 /* ARGSUSED */
4157 static int
4158 mptsas_pkt_alloc_extern(mptsas_t *mpt, mptsas_cmd_t *cmd,
4159     int cmdlen, int tgtlen, int statuslen, int kf)
4160 {
4161     caddr_t                 cdbp, scbp, tgt;
4162
4163     NDBG3(("mptsas_pkt_alloc_extern: "
4164         "cmd=0x%p cmdlen=%d tgtlen=%d statuslen=%d kf=%x",
4165         (void *)cmd, cmdlen, tgtlen, statuslen, kf));
4166
4167     tgt = cdbp = scbp = NULL;
4168     cmd->cmd_scrlen          = statuslen;
4169     cmd->cmd_privlen         = (uchar_t)tgtlen;
4170
4171     if (cmdlen > sizeof(cmd->cmd_cdb)) {
4172         if ((cdbp = kmalloc((size_t)cmdlen, kf)) == NULL) {
4173             goto fail;
4174         }
4175         cmd->cmd_pkt->pkt_cdbp = (opaque_t)cdbp;
4176         cmd->cmd_flags |= CFLAG_CDBEXTERN;
4177     }
4178     if (tgtlen > PKT_PRIV_LEN) {
4179         if ((tgt = kmalloc((size_t)tgtlen, kf)) == NULL) {
4180             goto fail;
4181         }
4182         cmd->cmd_flags |= CFLAG_PRIVEXTERN;
4183         cmd->cmd_pkt->pkt_private = tgt;
4184     }
4185     if (statuslen > EXTCMDS_STATUS_SIZE) {
4186         if ((scbp = kmalloc((size_t)statuslen, kf)) == NULL) {
4187             goto fail;
4188         }
4189         cmd->cmd_flags |= CFLAG_SCBEXTERN;
4190         cmd->cmd_pkt->pkt_scbp = (opaque_t)scbp;
4191
4192         /* allocate sense data buf for DMA */
4193         if (mptsas_cmddargsize(mpt, cmd, statuslen -
4194             MPTSA_GET_ITEM_OFF(struct scsi_arg_status, sts_sensedata),
4195             kf) == B_FALSE)
4196             goto fail;
4197     }
4198 fail:
4199     return (0);
4200
4201     mptsas_pkt_destroy_extern(mpt, cmd);
4202
4203     /*
4204      * deallocate external pkt space and deallocate the pkt
4205     */

```

```

4207 static void
4208 mptsas_pkt_destroy_extern(mptsas_t *mpt, mptsas_cmd_t *cmd)
4209 {
4210     NDBG3(("mptsas_pkt_destroy_extern: cmd=0x%p", (void *)cmd));
4211
4212     if (cmd->cmd_flags & CFLAG_FREE) {
4213         mptsas_log(mpt, CE_PANIC,
4214             "mptsas_pkt_destroy_extern: freeing free packet");
4215         _NOTE(NOT_REACHED)
4216         /* NOTREACHED */
4217     }
4218     if (cmd->cmd_extrqslen != 0) {
4219         rmfree(mpt->m_errqsense_map, cmd->cmd_extrqschunks,
4220                cmd->cmd_extrqsidx + 1);
4221     }
4222     if (cmd->cmd_flags & CFLAG_CDBEXTERN) {
4223         kmem_free(cmd->cmd_pkt->pkt_cdbp, (size_t)cmd->cmd_cdblen);
4224     }
4225     if (cmd->cmd_flags & CFLAG_SCBEXTERN) {
4226         kmem_free(cmd->cmd_pkt->pkt_scbp, (size_t)cmd->cmd_scblen);
4227     }
4228     if (cmd->cmd_flags & CFLAG_PRIVEXTERN) {
4229         kmem_free(cmd->cmd_pkt->pkt_private, (size_t)cmd->cmd_privlen);
4230     }
4231     cmd->cmd_flags = CFLAG_FREE;
4232     kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
4233 }
4235 /*
4236  * tran_sync_pkt(9E) - explicit DMA synchronization
4237 */
4238 /*ARGSUSED*/
4239 static void
4240 mptsas_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
4241 {
4242     mptsas_cmd_t *cmd = PKT2CMD(pkt);
4244
4245     NDBG3(("mptsas_scsi_sync_pkt: target=%d, pkt=0x%p",
4246            ap->a_target, (void *)pkt));
4247
4248     if (cmd->cmd_dmahandle) {
4249         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
4250                            (cmd->cmd_flags & CFLAG_DMASEND) ?
4251                                DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
4252     }
4254 /*
4255  * tran_dmafree(9E) - deallocate DMA resources allocated for command
4256 */
4257 /*ARGSUSED*/
4258 static void
4259 mptsas_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
4260 {
4261     mptsas_cmd_t *cmd = PKT2CMD(pkt);
4262     mptsas_t *mpt = ADDR2MPT(ap);
4264
4265     NDBG3(("mptsas_scsi_dmafree: target=%d pkt=0x%p",
4266            ap->a_target, (void *)pkt));
4267
4268     if (cmd->cmd_flags & CFLAG_DMavalid) {
4269         (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
4270         cmd->cmd_flags &= ~CFLAG_DMavalid;
4271     }
4272
4273     mptsas_free_extra_sgl_frame(mpt, cmd);

```

```

4273 }
4275 static void
4276 mptsas_pkt_comp(struct scsi_pkt *pkt, mptsas_cmd_t *cmd)
4277 {
4278     if ((cmd->cmd_flags & CFLAG_CMDIOPB) &&
4279         (!((cmd->cmd_flags & CFLAG_DMASEND))) {
4280         (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
4281                            DDI_DMA_SYNC_FORCPU);
4282     }
4283     (*pkt->pkt_comp)(pkt);
4284 }
4286 static void
4287 mptsas_sge_mainframe(mptsas_cmd_t *cmd, pMpI2SCSIIORequest_t frame,
4288                       ddi_acc_handle_t acc_hdl, uint_t cookiec, uint32_t end_flags)
4289 {
4290     pMpI2SGESimple64_t sge;
4291     mppti_t *dmap;
4292     uint32_t flags;
4294
4295     dmap = cmd->cmd_sg;
4296     sge = (pMpI2SGESimple64_t)(&frame->SGL);
4297     while (cookiec--) {
4298         ddi_put32(acc_hdl,
4299                    &sge->Address.Low, dmap->addr.address64.Low);
4300         ddi_put32(acc_hdl,
4301                    &sge->Address.High, dmap->addr.address64.High);
4302         ddi_put32(acc_hdl, &sge->FlagsLength,
4303                    dmap->count);
4304         flags = ddi_get32(acc_hdl, &sge->FlagsLength);
4305         flags |= ((uint32_t)
4306                    (MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4307                     MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4308                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4309                     MPI2_SGE_FLAGS_SHIFT);
4311
4312         /*
4313          * If this is the last cookie, we set the flags
4314          * to indicate so
4315         */
4316         if (cookiec == 0) {
4317             flags |= end_flags;
4318         }
4319         if (cmd->cmd_flags & CFLAG_DMASEND) {
4320             flags |= (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4321                         MPI2_SGE_FLAGS_SHIFT);
4322         } else {
4323             flags |= (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4324                         MPI2_SGE_FLAGS_SHIFT);
4325         }
4326         ddi_put32(acc_hdl, &sge->FlagsLength, flags);
4327         dmap++;
4328         sge++;
4329     }
4331 static void
4332 mptsas_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4333                    pMpI2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4334 {
4335     pMpI2SGESimple64_t sge;
4336     pMpI2SGEChain64_t sgechain;
4337     uint64_t nframe_phys_addr;
4338     uint_t cookiec;

```

```

4339     mptti_t          *dmap;
4340     uint32_t          flags;
4341
4342     /*
4343      * Save the number of entries in the DMA
4344      * Scatter/Gather list
4345      */
4346     cookiec = cmd->cmd_cookiec;
4347
4348     /*
4349      * Hereby we start to deal with multiple frames.
4350      * The process is as follows:
4351      * 1. Determine how many frames are needed for SGL element
4352      * storage; Note that all frames are stored in contiguous
4353      * memory space and in 64-bit DMA mode each element is
4354      * 3 double-words (12 bytes) long.
4355      * 2. Fill up the main frame. We need to do this separately
4356      * since it contains the SCSI IO request header and needs
4357      * dedicated processing. Note that the last 4 double-words
4358      * of the SCSI IO header is for SGL element storage
4359      * (MPI2_SGE_IO_UNION).
4360      * 3. Fill the chain element in the main frame, so the DMA
4361      * engine can use the following frames.
4362      * 4. Enter a loop to fill the remaining frames. Note that the
4363      * last frame contains no chain element. The remaining
4364      * frames go into the mpt SGL buffer allocated on the fly,
4365      * not immediately following the main message frame, as in
4366      * Gen1.
4367      * Some restrictions:
4368      * 1. For 64-bit DMA, the simple element and chain element
4369      * are both of 3 double-words (12 bytes) in size, even
4370      * though all frames are stored in the first 4G of mem
4371      * range and the higher 32-bits of the address are always 0.
4372      * 2. On some controllers (like the 1064/1068), a frame can
4373      * hold SGL elements with the last 1 or 2 double-words
4374      * (4 or 8 bytes) un-used. On these controllers, we should
4375      * recognize that there's not enough room for another SGL
4376      * element and move the sge pointer to the next frame.
4377      */
4378     int             i, j, k, l, frames, sgemax;
4379     int             temp;
4380     uint8_t         chainflags;
4381     uint16_t        chainlength;
4382     mptsas_cache_frames_t *p;
4383
4384     /*
4385      * Sgemax is the number of SGE's that will fit
4386      * each extra frame and frames is total
4387      * number of frames we'll need. 1 sge entry per
4388      * frame is reseverd for the chain element thus the -1 below.
4389      */
4390     sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_SGE_SIMPLE64))
4391               - 1);
4392     temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;
4393
4394     /*
4395      * A little check to see if we need to round up the number
4396      * of frames we need
4397      */
4398     if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
4399           sgemax) > 1) {
4400         frames = (temp + 1);
4401     } else {
4402         frames = temp;
4403     }
4404     dmap = cmd->cmd_sg;

```

```

4405     sge = (pMp12SGESimple64_t)(&frame->SGL);
4406
4407     /*
4408      * First fill in the main frame
4409      */
4410     j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
4411     mptsas_sge_mainframe(cmd, frame, acc_hdl, j,
4412                           ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4413                            MPI2_SGE_FLAGS_SHIFT));
4414     dmap += j;
4415     sge += j;
4416     j++;
4417
4418     /*
4419      * Fill in the chain element in the main frame.
4420      * About calculation on ChainOffset:
4421      * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4422      * in the end reserved for SGL element storage
4423      * (MPI2_SGE_IO_UNION); we should count it in our
4424      * calculation. See its definition in the header file.
4425      * 2. Constant j is the counter of the current SGL element
4426      * that will be processed, and (j - 1) is the number of
4427      * SGL elements that have been processed (stored in the
4428      * main frame).
4429      * 3. ChainOffset value should be in units of double-words (4
4430      * bytes) so the last value should be divided by 4.
4431      */
4432     ddi_put8(acc_hdl, &frame->ChainOffset,
4433               (sizeof (MPI2_SCSI_IO_REQUEST) -
4434                sizeof (MPI2_SGE_IO_UNION) +
4435                (j - 1) * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
4436     sgechain = (pMp12SGEChain64_t)sge;
4437     chainflags = (MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4438                   MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4439                   MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4440     ddi_put8(acc_hdl, &sgechain->Flags, chainflags);
4441
4442     /*
4443      * The size of the next frame is the accurate size of space
4444      * (in bytes) used to store the SGL elements. j is the counter
4445      * of SGL elements. (j - 1) is the number of SGL elements that
4446      * have been processed (stored in frames).
4447      */
4448     if (frames >= 2) {
4449       ASSERT(mpt->m_req_frame_size >= sizeof (MPI2_SGE_SIMPLE64));
4450       chainlength = mpt->m_req_frame_size /
4451                     sizeof (MPI2_SGE_SIMPLE64) *
4452                     sizeof (MPI2_SGE_SIMPLE64);
4453     } else {
4454       chainlength = ((cookiec - (j - 1)) *
4455                     sizeof (MPI2_SGE_SIMPLE64));
4456     }
4457     p = cmd->cmd_extra_frames;
4458
4459     ddi_put16(acc_hdl, &sgechain->Length, chainlength);
4460     ddi_put32(acc_hdl, &sgechain->Address.Low, p->m_phys_addr);
4461     ddi_put32(acc_hdl, &sgechain->Address.High, p->m_phys_addr >> 32);
4462
4463     /*
4464      * If there are more than 2 frames left we have to
4465      * fill in the next chain offset to the location of
4466      * the chain element in the next frame.
4467      * sgemax is the number of simple elements in an extra
4468      * frame. Note that the value NextChainOffset should be
4469      * in double-words (4 bytes).
4470

```

```

4471     /*
4472      if (frames >= 2) {
4473          ddi_put8(acc_hdl, &sgechain->NextChainOffset,
4474                  (sgemax * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
4475      } else {
4476          ddi_put8(acc_hdl, &sgechain->NextChainOffset, 0);
4477      }
4478
4479      /*
4480      * Jump to next frame;
4481      * Starting here, chain buffers go into the per command SGL.
4482      * This buffer is allocated when chain buffers are needed.
4483      */
4484      sge = (pMpI2SGESimple64_t)p->m_frames_addr;
4485      i = cookiec;
4486
4487      /*
4488      * Start filling in frames with SGE's. If we
4489      * reach the end of frame and still have SGE's
4490      * to fill we need to add a chain element and
4491      * use another frame. j will be our counter
4492      * for what cookie we are at and i will be
4493      * the total cookiec. k is the current frame
4494      */
4495      for (k = 1; k <= frames; k++) {
4496          for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {
4497
4498              /*
4499              * If we have reached the end of frame
4500              * and we have more SGE's to fill in
4501              * we have to fill the final entry
4502              * with a chain element and then
4503              * continue to the next frame
4504              */
4505              if ((l == (sgemax + 1)) && (k != frames)) {
4506                  sgechain = (pMpI2SGEChain64_t)sge;
4507                  j--;
4508                  chainflags = (
4509                      MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4510                      MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4511                      MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4512                  ddi_put8(p->m_acc_hdl,
4513                          &sgechain->Flags, chainflags);
4514
4515                  /*
4516                  * k is the frame counter and (k + 1)
4517                  * is the number of the next frame.
4518                  * Note that frames are in contiguous
4519                  * memory space.
4520                  */
4521                  nframe_phys_addr = p->m_phys_addr +
4522                      (mpt->m_req_frame_size * k);
4523                  ddi_put32(p->m_acc_hdl,
4524                          &sgechain->Address.Low,
4525                          nframe_phys_addr);
4526                  ddi_put32(p->m_acc_hdl,
4527                          &sgechain->Address.High,
4528                          nframe_phys_addr >> 32);
4529
4530                  /*
4531                  * If there are more than 2 frames left
4532                  * we have to next chain offset to
4533                  * the location of the chain element
4534                  * in the next frame and fill in the
4535                  * length of the next chain
4536                  */
4537                  if ((frames - k) >= 2) {

```

```

4537                     ddi_put8(p->m_acc_hdl,
4538                         &sgechain->NextChainOffset,
4539                         (sgemax *
4540                         sizeof (MPI2_SGE_SIMPLE64))
4541                         >> 2);
4542                     ddi_put16(p->m_acc_hdl,
4543                         &sgechain->Length,
4544                         mpt->m_req_frame_size /
4545                         sizeof (MPI2_SGE_SIMPLE64) *
4546                         sizeof (MPI2_SGE_SIMPLE64));
4547                 } else {
4548                     /*
4549                     * This is the last frame. Set
4550                     * the NextChainOffset to 0 and
4551                     * Length is the total size of
4552                     * all remaining simple elements
4553                     */
4554                     ddi_put8(p->m_acc_hdl,
4555                         &sgechain->NextChainOffset,
4556                         0);
4557                     ddi_put16(p->m_acc_hdl,
4558                         &sgechain->Length,
4559                         (cookiec - j) *
4560                         sizeof (MPI2_SGE_SIMPLE64));
4561                 }
4562
4563                 /*
4564                 * Jump to the next frame */
4565                 sge = (pMpI2SGESimple64_t)
4566                     ((char *)p->m_frames_addr +
4567                     (int)mpt->m_req_frame_size * k);
4568
4569             continue;
4570         }
4571
4572         ddi_put32(p->m_acc_hdl,
4573             &sge->Address.Low,
4574             dmap->addr.address64.Low);
4575         ddi_put32(p->m_acc_hdl,
4576             &sge->Address.High,
4577             dmap->addr.address64.High);
4578         ddi_put32(p->m_acc_hdl,
4579             &sge->FlagsLength, dmap->count);
4580         flags = ddi_get32(p->m_acc_hdl,
4581             &sge->FlagsLength);
4582         flags |= ((uint32_t)(
4583             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4584             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4585             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4586             MPI2_SGE_FLAGS_SHIFT);
4587
4588         /*
4589         * If we are at the end of the frame and
4590         * there is another frame to fill in
4591         * we set the last simple element as last
4592         * element
4593         */
4594         if ((l == sgemax) && (k != frames)) {
4595             flags |= ((uint32_t)
4596                 (MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4597                 MPI2_SGE_FLAGS_SHIFT);
4598         }
4599
4600         /*
4601         * If this is the final cookie we
4602         * indicate it by setting the flags
4603         */

```

```

4603     if (j == i) {
4604         flags |= ((uint32_t)
4605             (MPI2_SGE_FLAGS_LAST_ELEMENT |
4606             MPI2_SGE_FLAGS_END_OF_BUFFER |
4607             MPI2_SGE_FLAGS_END_OF_LIST) <<
4608             MPI2_SGE_FLAGS_SHIFT);
4609     }
4610     if (cmd->cmd_flags & CFLAG_DMASEND) {
4611         flags |=
4612             (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4613             MPI2_SGE_FLAGS_SHIFT);
4614     } else {
4615         flags |=
4616             (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4617             MPI2_SGE_FLAGS_SHIFT);
4618     }
4619     ddi_put32(p->m_acc_hdl,
4620               &sge->FlagsLength, flags);
4621     dmap++;
4622     sge++;
4623 }
4624 }

4625 /*
4626  * Sync DMA with the chain buffers that were just created
4627  */
4628 (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4629 }

4630 }

4631 static void
4632 mptsas_ieee_sge_mainframe(mptsas_cmd_t *cmd, pMpI2SCSIIORequest_t frame,
4633 ddi_acc_handle_t acc_hdl, uint_t cookiec, uint8_t end_flag)
4634 {
4635     pMpI2IeeeSgeSimple64_t ieesesge;
4636     mptti_t *dmap;
4637     uint8_t flags;
4638

4639     dmap = cmd->cmd_sg;
4640

4641     NDBG1(("mptsas_ieee_sge_mainframe: cookiec=%d, %s", cookiec,
4642            cmd->cmd_flags & CFLAG_DMASEND?"Out":"In"));

4643     ieesesge = (pMpI2IeeeSgeSimple64_t)(&frame->SGL);
4644     while (cookiec--) {
4645         ddi_put32(acc_hdl,
4646                   &ieeesesge->Address.Low, dmap->addr.address64.Low);
4647         ddi_put32(acc_hdl,
4648                   &ieeesesge->Address.High, dmap->addr.address64.High);
4649         ddi_put32(acc_hdl, &ieeesesge->Length,
4650                   dmap->count);
4651         NDBG1(("mptsas_ieee_sge_mainframe: len=%d", dmap->count));
4652         flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4653                  MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);

4654         /*
4655          * If this is the last cookie, we set the flags
4656          * to indicate so
4657          */
4658         if (cookiec == 0) {
4659             flags |= end_flag;
4660         }

4661         ddi_put8(acc_hdl, &ieeesesge->Flags, flags);
4662         dmap++;
4663         ieesesge++;
4664     }
4665
4666
4667
4668 }

```

```

4669 }

4670 static void
4671 mptsas_ieee_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4672                         pMpI2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4673 {
4674     pMpI2IeeeSgeSimple64_t ieesesge;
4675     pMpI2IeeeSgeChain64_t ieesesgechain;
4676     uint64_t nframe_phys_addr;
4677     uint_t cookiec;
4678     mptti_t *dmap;
4679     uint8_t flags;

4680     /*
4681      * Save the number of entries in the DMA
4682      * Scatter/Gather list
4683      */
4684     cookiec = cmd->cmd_cookiec;

4685     NDBG1(("mptsas_ieee_sge_chain: cookiec=%d", cookiec));

4686     /*
4687      * Hereby we start to deal with multiple frames.
4688      * The process is as follows:
4689      * 1. Determine how many frames are needed for SGL element
4690      * storage; Note that all frames are stored in contiguous
4691      * memory space and in 64-bit DMA mode each element is
4692      * 4 double-words (16 bytes) long.
4693      * 2. Fill up the main frame. We need to do this separately
4694      * since it contains the SCSI IO request header and needs
4695      * dedicated processing. Note that the last 4 double-words
4696      * of the SCSI IO header is for SGL element storage
4697      * (MPI2_SGE_IO_UNION).
4698      * 3. Fill the chain element in the main frame, so the DMA
4699      * engine can use the following frames.
4700      * 4. Enter a loop to fill the remaining frames. Note that the
4701      * last frame contains no chain element. The remaining
4702      * frames go into the mpt_sgL buffer allocated on the fly,
4703      * not immediately following the main message frame, as in
4704      * Gen1.
4705      * Restrictions:
4706      * For 64-bit DMA, the simple element and chain element
4707      * are both of 4 double-words (16 bytes) in size, even
4708      * though all frames are stored in the first 4G of mem
4709      * range and the higher 32-bits of the address are always 0.
4710      */
4711     int i, j, k, l, frames, sgemax;
4712     int temp;
4713     uint8_t chainflags;
4714     uint32_t chainlength;
4715     mptsas_cache_frames_t *p;

4716     /*
4717      * Sgemax is the number of SGE's that will fit
4718      * each extra frame and frames is total
4719      * number of frames we'll need. 1 sge entry per
4720      * frame is reserved for the chain element thus the -1 below.
4721      */
4722     sgemax = ((mpt->m_req_frame_size / sizeof(MPI2_IEEE_SGE_SIMPLE64))
4723               - 1);
4724     temp = (cookiec - (MPTAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;

4725     /*
4726      * A little check to see if we need to round up the number
4727      * of frames we need
4728      */
4729
4730
4731
4732
4733
4734

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

5

```

4735     if (((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
4736             sgemax) > 1) {
4737         frames = (temp + 1);
4738     } else {
4739         frames = temp;
4740     }
4741     NDBG1(("mptsas_ieee_sge_chain: temp=%d, frames=%d", temp, frames));
4742     dmap = cmd->cmd_sg;
4743     ieeesge = (pMpI2IeeeSgeSimple64_t)(&frame->SGL);

4745     /*
4746      * First fill in the main frame
4747      */
4748     j = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
4749     mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl, j, 0);
4750     dmap += j;
4751     ieeesge += j;
4752     j++;

4754     /*
4755      * Fill in the chain element in the main frame.
4756      * About calculation on ChainOffset:
4757      * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4758      *      in the end reserved for SGL element storage
4759      *      (MPI2_SGE_IO_UNION); we should count it in our
4760      *      calculation. See its definition in the header file.
4761      * 2. Constant j is the counter of the current SGL element
4762      *      that will be processed, and (j - 1) is the number of
4763      *      SGL elements that have been processed (stored in the
4764      *      main frame).
4765      * 3. ChainOffset value should be in units of quad-words (16
4766      *      bytes) so the last value should be divided by 16.
4767      */
4768     ddi_put8(acc_hdl, &frame->ChainOffset,
4769             (sizeof (MPI2_SCSI_IO_REQUEST) -
4770              sizeof (MPI2_SGE_IO_UNION) +
4771              (j - 1) * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4772     ieeesgechain = (pMpI2IeeeSgeChain64_t)ieeesge;
4773     chainflags = (MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4774                   MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4775     ddi_put8(acc_hdl, &ieeesgechain->Flags, chainflags);

4777     /*
4778      * The size of the next frame is the accurate size of space
4779      * (in bytes) used to store the SGL elements. j is the counter
4780      * of SGL elements. (j - 1) is the number of SGL elements that
4781      * have been processed (stored in frames).
4782      */
4783     if (frames >= 2) {
4784         ASSERT(mpt->m_req_frame_size >=
4785                 sizeof (MPI2_IEEE_SGE_SIMPLE64));
4786         chainlength = mpt->m_req_frame_size /
4787                     sizeof (MPI2_IEEE_SGE_SIMPLE64) *
4788                     sizeof (MPI2_IEEE_SGE_SIMPLE64);
4789     } else {
4790         chainlength = ((cookiec - (j - 1)) *
4791                         sizeof (MPI2_IEEE_SGE_SIMPLE64));
4792     }

4794     p = cmd->cmd_extra_frames;

4796     ddi_put32(acc_hdl, &ieeesgechain->Length, chainlength);
4797     ddi_put32(acc_hdl, &ieeesgechain->Address.Low, p->m_phys_addr);
4798     ddi_put32(acc_hdl, &ieeesgechain->Address.High, p->m_phys_addr >> 32);

4800     /*

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

```

4801 * If there are more than 2 frames left we have to
4802 * fill in the next chain offset to the location of
4803 * the chain element in the next frame.
4804 * sgemax is the number of simple elements in an extra
4805 * frame. Note that the value NextChainOffset should be
4806 * in double-words (4 bytes).
4807 */
4808 if (frames >= 2) {
4809     ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset,
4810             (sgemax * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4811 } else {
4812     ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset, 0);
4813 }
4814 /*
4815 * Jump to next frame;
4816 * Starting here, chain buffers go into the per command SGL.
4817 * This buffer is allocated when chain buffers are needed.
4818 */
4819 ieeesge = (pMpI2IeeeSgeSimple64_t)p->m_frames_addr;
4820 i = cookiec;
4821 /*
4822 * Start filling in frames with SGE's.  If we
4823 * reach the end of frame and still have SGE's
4824 * to fill we need to add a chain element and
4825 * use another frame.  j will be our counter
4826 * for what cookie we are at and i will be
4827 * the total cookiec. k is the current frame
4828 */
4829 for (k = 1; k <= frames; k++) {
4830     for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {
4831         /*
4832         * If we have reached the end of frame
4833         * and we have more SGE's to fill in
4834         * we have to fill the final entry
4835         * with a chain element and then
4836         * continue to the next frame
4837         */
4838 if ((l == (sgemax + 1)) && (k != frames)) {
4839     ieeesgechain = (pMpI25IeeeSgeChain64_t)ieeesge;
4840     j--;
4841     chainflags =
4842         MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4843         MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR;
4844     ddi_put8(p->m_acc_hdl,
4845             &ieeesgechain->Flags, chainflags);
4846     /*
4847     * k is the frame counter and (k + 1)
4848     * is the number of the next frame.
4849     * Note that frames are in contiguous
4850     * memory space.
4851     */
4852     nframe_phys_addr = p->m_phys_addr +
4853     (mpt->m_req_frame_size * k);
4854     ddi_put32(p->m_acc_hdl,
4855             &ieeesgechain->Address.Low,
4856             nframe_phys_addr);
4857     ddi_put32(p->m_acc_hdl,
4858             &ieeesgechain->Address.High,
4859             nframe_phys_addr >> 32);
4860     /*
4861     * If there are more than 2 frames left
4862     * we have to next chain offset to

```

```

4867             * the location of the chain element
4868             * in the next frame and fill in the
4869             * length of the next chain
4870             */
4871     if ((frames - k) >= 2) {
4872         ddi_put8(p->m_acc_hdl,
4873                 &ieeesgechain->NextChainOffset,
4874                 (sgemax *
4875                  sizeof (MPI2_IEEE_SGE_SIMPLE64))
4876                  >> 4);
4877     ASSERT(mpt->m_req_frame_size >=
4878            sizeof (MPI2_IEEE_SGE_SIMPLE64));
4879     ddi_put32(p->m_acc_hdl,
4880                 &ieeesgechain->Length,
4881                 mpt->m_req_frame_size /
4882                 sizeof (MPI2_IEEE_SGE_SIMPLE64) *
4883                 sizeof (MPI2_IEEE_SGE_SIMPLE64));
4884 } else {
4885     /*
4886      * This is the last frame. Set
4887      * the NextChainOffset to 0 and
4888      * Length is the total size of
4889      * all remaining simple elements
4890      */
4891     ddi_put8(p->m_acc_hdl,
4892                 &ieeesgechain->NextChainOffset,
4893                 0);
4894     ddi_put32(p->m_acc_hdl,
4895                 &ieeesgechain->Length,
4896                 (cookiec - j) *
4897                 sizeof (MPI2_IEEE_SGE_SIMPLE64));
4898 }

4899 /* Jump to the next frame */
4900 ieeesge = (pMpI2IeeeSgeSimple64_t)
4901     ((char *)p->m_frames_addr +
4902      (int)mpt->m_req_frame_size * k);
4903
4904     continue;
4905 }
4906
4907 ddi_put32(p->m_acc_hdl,
4908             &ieeesge->Address.Low,
4909             dmap->addr.address64.Low);
4910 ddi_put32(p->m_acc_hdl,
4911             &ieeesge->Address.High,
4912             dmap->addr.address64.High);
4913 ddi_put32(p->m_acc_hdl,
4914             &ieeesge->Length, dmap->count);
4915 flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4916           MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4917
4918 /*
4919  * If we are at the end of the frame and
4920  * there is another frame to fill in
4921  * do we need to do anything?
4922  * if ((l == sgemax) && (k != frames)) {
4923  * }
4924  */
4925
4926 /*
4927  * If this is the final cookie set end of list.
4928  */
4929 if (j == i) {
4930     flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
4931 }

```

```

4934             ddi_put8(p->m_acc_hdl, &ieeesge->Flags, flags);
4935             dmap++;
4936             ieeesge++;
4937         }
4938     }
4939
4940     /*
4941      * Sync DMA with the chain buffers that were just created
4942      */
4943     (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4944 }
4945
4946 static void
4947 mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd, uint32_t *control,
4948 pMpI2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4949 {
4950     ASSERT(cmd->cmd_flags & CFLAG_DMAVALID);
4951
4952     NDBG1(("mptsas_sge_setup: cookiec=%d", cmd->cmd_cookiec));
4953
4954     /*
4955      * Set read/write bit in control.
4956      */
4957     if (cmd->cmd_flags & CFLAG_DMASEND) {
4958         *control |= MPI2_SCSIIO_CONTROL_WRITE;
4959     } else {
4960         *control |= MPI2_SCSIIO_CONTROL_READ;
4961     }
4962
4963     ddi_put32(acc_hdl, &frame->DataLength, cmd->cmd_dmacount);
4964
4965     /*
4966      * We have 4 cases here. First where we can fit all the
4967      * SG elements into the main frame, and the case
4968      * where we can't. The SG element is also different when using
4969      * MPI2.5 interface.
4970      * If we have more cookies than we can attach to a frame
4971      * we will need to use a chain element to point
4972      * a location of memory where the rest of the S/G
4973      * elements reside.
4974      */
4975     if (cmd->cmd_cookiec <= MPTSAS_MAX_FRAME_SGES64(mpt)) {
4976         if (mpt->m_MPI25) {
4977             mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl,
4978                                         cmd->cmd_cookiec,
4979                                         MPI25_IEEE_SGE_FLAGS_END_OF_LIST);
4980         } else {
4981             mptsas_sge_mainframe(cmd, frame, acc_hdl,
4982                                         cmd->cmd_cookiec,
4983                                         ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT
4984                                         | MPI2_SGE_FLAGS_END_OF_BUFFER
4985                                         | MPI2_SGE_FLAGS_END_OF_LIST) <<
4986                                         MPI2_SGE_FLAGS_SHIFT));
4987         }
4988     } else {
4989         if (mpt->m_MPI25) {
4990             mptsas_ieee_sge_chain(mpt, cmd, frame, acc_hdl);
4991         } else {
4992             mptsas_sge_chain(mpt, cmd, frame, acc_hdl);
4993         }
4994     }
4995 }
4996
4997 /* Interrupt handling

```

```

4999 * Utility routine. Poll for status of a command sent to HBA
5000 * without interrupts (a FLAG_NOINTR command).
5001 */
5002 int
5003 mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
5004 {
5005     int rval = TRUE;
5006
5007     NDBG5(("mptsas_poll: cmd=0x%p", (void *)poll_cmd));
5008
5009     if ((poll_cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
5010         mptsas_restart_hba(mpt);
5011     }
5012
5013     /*
5014      * Wait, using drv_usecwait(), long enough for the command to
5015      * reasonably return from the target if the target isn't
5016      * "dead". A polled command may well be sent from scsi_poll, and
5017      * there are retries built in to scsi_poll if the transport
5018      * accepted the packet (TRAN_ACCEPT). scsi_poll waits 1 second
5019      * and retries the transport up to scsi_poll_busycnt times
5020      * (currently 60) if
5021      * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
5022      * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
5023
5024      * limit the waiting to avoid a hang in the event that the
5025      * cmd never gets started but we are still receiving interrupts
5026      */
5027     while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
5028         if (mptsas_wait_intr(mpt, polltime) == FALSE) {
5029             NDBG5(("mptsas_poll: command incomplete"));
5030             rval = FALSE;
5031             break;
5032         }
5033     }
5034
5035     if (rval == FALSE) {
5036
5037         /*
5038          * this isn't supposed to happen, the hba must be wedged
5039          * Mark this cmd as a timeout.
5040          */
5041         mptsas_set_pkt_reason(mpt, poll_cmd, CMD_TIMEOUT,
5042                               (STAT_TIMEOUT|STAT_ABORTED));
5043
5044         if (poll_cmd->cmd_queued == FALSE) {
5045
5046             NDBG5(("mptsas_poll: not on waitq"));
5047
5048             poll_cmd->cmd_pkt->pkt_state |=
5049                         (STATE_GOT_BUS|STATE_GOT_TARGET|STATE_SENT_CMD);
5050         } else {
5051
5052             /* find and remove it from the waitq */
5053             NDBG5(("mptsas_poll: delete from waitq"));
5054             mptsas_waitq_delete(mpt, poll_cmd);
5055         }
5056
5057     }
5058     mptsas_fma_check(mpt, poll_cmd);
5059     NDBG5(("mptsas_poll: done"));
5060     return (rval);
5061 }
5062
5063 */
5064 * Used for polling cmds and TM function

```

```

5065 */
5066 static int
5067 mptsas_wait_intr(mptsas_t *mpt, int polltime)
5068 {
5069     int cnt;
5070     pMpI2ReplyDescriptorsUnion_t reply_desc_union;
5071     uint32_t int_mask;
5072
5073     NDBG5(("mptsas_wait_intr"));
5074
5075     mpt->m_polled_intr = 1;
5076
5077     /*
5078      * Get the current interrupt mask and disable interrupts. When
5079      * re-enabling ints, set mask to saved value.
5080      */
5081     int_mask = ddi_get32(mpt->m_datap, &mpt->m_reg->HostInterruptMask);
5082     MPTSAS_DISABLE_INTR(mpt);
5083
5084     /*
5085      * Keep polling for at least (polltime * 1000) seconds
5086      */
5087     for (cnt = 0; cnt < polltime; cnt++) {
5088         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5089                             DDI_DMA_SYNC_FORCPU);
5090
5091         reply_desc_union = (pMpI2ReplyDescriptorsUnion_t)
5092                           MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);
5093
5094         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5095                       &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5096             ddi_get32(mpt->m_acc_post_queue_hdl,
5097                       &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5098             drv_usecwait(1000);
5099             continue;
5100         }
5101
5102         /*
5103          * The reply is valid, process it according to its
5104          * type.
5105          */
5106         mptsas_process_intr(mpt, reply_desc_union);
5107
5108         if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5109             mpt->m_post_index = 0;
5110         }
5111
5112         /*
5113          * Update the global reply index
5114          */
5115         ddi_put32(mpt->m_datap,
5116                   &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
5117         mpt->m_polled_intr = 0;
5118
5119         /*
5120          * Re-enable interrupts and quit.
5121          */
5122         ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask,
5123                   int_mask);
5124         return (TRUE);
5125     }
5126
5127     /*
5128      * Clear polling flag, re-enable interrupts and quit.
5129      */
5130

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

63

```

5131     mpt->m_polled_intr = 0;
5132     ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, int_mask);
5133     return (FALSE);
5134 }

5136 static void
5137 mptsas_handle_scsi_io_success(mptsas_t *mpt,
5138     pMpI2ReplyDescriptorsUnion_t reply_desc)
5139 {
5140     pMpI2SCSIIOSuccessReplyDescriptor_t      scsi_io_success;
5141     uint16_t                                SMID;
5142     mptsas_slots_t                          *slots = mpt->m_active;
5143     mptsas_cmd_t                           *cmd = NULL;
5144     struct scsi_pkt                         *pkt;
5145
5146     ASSERT(mutex_owned(&mpt->m_mutex));
5147
5148     scsi_io_success = (pMpI2SCSIIOSuccessReplyDescriptor_t)reply_desc;
5149     SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &scsi_io_success->SMID);
5150
5151     /*
5152      * This is a success reply so just complete the IO. First, do a sanity
5153      * check on the SMID. The final slot is used for TM requests, which
5154      * would not come into this reply handler.
5155      */
5156     if ((SMID == 0) || (SMID > slots->m_n_normal)) {
5157         mptsas_log(mpt, CE_WARN, "?Received invalid SMID of %d\n",
5158                     SMID);
5159         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5160         return;
5161     }
5162
5163     cmd = slots->m_slot[SMID];
5164
5165     /*
5166      * print warning and return if the slot is empty
5167      */
5168     if (cmd == NULL) {
5169         mptsas_log(mpt, CE_WARN, "?NULL command for successful SCSI IO
5170                               in slot %d", SMID);
5171         return;
5172     }
5173
5174     pkt = CMD2PKT(cmd);
5175     pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET | STATE_SENT_CMD |
5176                         STATE_GOT_STATUS);
5177     if (cmd->cmd_flags & CFLAG_DMAVALID) {
5178         pkt->pkt_state |= STATE_XFERRED_DATA;
5179     }
5180     pkt->pkt_resid = 0;
5181
5182     if (cmd->cmd_flags & CFLAG_PASSTHRU) {
5183         cmd->cmd_flags |= CFLAG_FINISHED;
5184         cv_broadcast(&mpt->m_passthru_cv);
5185         return;
5186     } else {
5187         mptsas_remove_cmd(mpt, cmd);
5188     }
5189
5190     if (cmd->cmd_flags & CFLAG_RETRY) {
5191         /*
5192          * The target returned QFULL or busy, do not add tihs
5193          * pkt to the doneq since the hba will retry
5194          * this cmd.
5195          *
5196          * The pkt has already been resubmitted in

```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c 64

5197             * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
5198             * Remove this cmd_flag here.
5199             */
5200         cmd->cmd_flags &= ~CFLAG_RETRY;
5201     } else {
5202         mptsas_doneq_add(mpt, cmd);
5203     }
5204 }

5206 static void
5207 mptsas_handle_address_reply(mptsas_t *mpt,
5208     pMpI2ReplyDescriptorsUnion_t reply_desc)
5209 {
5210     pMpI2AddressDescriptor_t address_reply;
5211     pMpI2DefaultReply_t reply;
5212     mptsas_fw_diagnostic_buffer_t *pBuffer;
5213     uint32_t reply_addr, reply_frame_dma_baseaddr, SMID, iocstatus;
5214     uint16_t *slots = mpt->m_active;
5215     mptsas_slots_t *cmd = NULL;
5216     mptsas_cmd_t function, buffer_type;
5217     uint8_t *args;
5218     m_replyh_arg_t reply_frame_no;
5219     int ASSERT(mutex_owned(&mpt->m_mutex));

5221     address_reply = (pMpI2AddressReplyDescriptor_t)reply_desc;
5222     reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
5223         &address_reply->ReplyFrameAddress);
5224     SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

5225     /*
5226      * If reply frame is not in the proper range we should ignore this
5227      * message and exit the interrupt handler.
5228      */
5229     reply_frame_dma_baseaddr = mpt->m_reply_frame_dma_addr & 0xfffffffffu;
5230     if ((reply_addr < reply_frame_dma_baseaddr) ||
5231         (reply_addr >= (reply_frame_dma_baseaddr +
5232             (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
5233         ((reply_addr - reply_frame_dma_baseaddr) %
5234             mpt->m_reply_frame_size != 0)) {
5235         mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
5236             "address 0x%x\n", reply_addr);
5237         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5238         return;
5239     }

5240     (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
5241         DDI_DMA_SYNC_FORCPU);
5242     reply = (pMpI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
5243         reply_frame_dma_baseaddr));
5244     function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

5245     NDBG31(("mptsas_handle_address_reply: function 0x%x, reply_addr=0x%x",
5246             function, reply_addr));

5247     /*
5248      * don't get slot information and command for events since these values
5249      * don't exist
5250      */
5251     if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
5252         (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
5253         /*
5254          * This could be a TM reply, which use the last allocated SMID,
5255          * so allow for that.
5256          */
5257 
```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
5263         if ((SMID == 0) || (SMID > (slots->m_n_normal + 1))) {
5264             mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
5265                         "%d\n", SMID);
5266             ddi_fn_service_impact(mpt->m_dip,
5267                                     DDI_SERVICE_UNAFFECTED);
5268             return;
5269         }
5271
5271     cmd = slots->m_slot[SMID];
5273
5274     /*
5275      * print warning and return if the slot is empty
5276      */
5277     if (cmd == NULL) {
5278         mptsas_log(mpt, CE_WARN, "?NULL command for address "
5279                     "reply in slot %d", SMID);
5280         return;
5281     }
5282     if ((cmd->cmd_flags &
5283          (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG))) {
5284         cmd->cmd_rfm = reply_addr;
5285         cmd->cmd_flags |= CFLAG_FINISHED;
5286         cv_broadcast(&mpt->m_passthru_cv);
5287         cv_broadcast(&mpt->m_config_cv);
5288         cv_broadcast(&mpt->m_fw_diag_cv);
5289         return;
5290     } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
5291         mptsas_remove_cmd(mpt, cmd);
5292     }
5293     NDBG31(("\\t\\tmpptsas_process_intr: slot=%d", SMID));
5294 }
5295 /*
5296  * Depending on the function, we need to handle
5297  * the reply frame (and cmd) differently.
5298 */
5299 switch (function) {
5300 case MPI2_FUNCTION_SCSI_IO_REQUEST:
5301     mptsas_check_scsi_io_error(mpt, (pMpI2SCSIIOReply_t)reply, cmd);
5302     break;
5303 case MPI2_FUNCTION_SCSI_TASK_MGMT:
5304     cmd->cmd_rfm = reply_addr;
5305     mptsas_check_task_mgt(mpt, (pMpI2SCSITaskManagementReply_t)reply,
5306                           cmd);
5307     break;
5308 case MPI2_FUNCTION_FW_DOWNLOAD:
5309     cmd->cmd_flags |= CFLAG_FINISHED;
5310     cv_signal(&mpt->m_fw_cv);
5311     break;
5312 case MPI2_FUNCTION_EVENT_NOTIFICATION:
5313     reply_frame_no = (reply_addr - reply_frame_dma_baseaddr) /
5314                 mpt->m_reply_frame_size;
5315     args = &mpt->m_replayh_args[reply_frame_no];
5316     args->mpt = (void *)mpt;
5317     args->rfm = reply_addr;
5318
5319     /*
5320      * Record the event if its type is enabled in
5321      * this mpt instance by ioctl.
5322      */
5323     mptsas_record_event(args);
5324
5325     /*
5326      * Handle time critical events
5327      * NOT_RESPONDING/ADDED only now
5328      */
5329     if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {

```

6

```
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c 66

5329
5330         /*
5331          * Would not return main process,
5332          * just let taskq resolve ack action
5333          * and ack would be sent in taskq thread
5334          */
5335      NDBG20(("send mptsas_handle_event_sync success"));
5336  }

5337  if (mpt->m_in_reset) {
5338      NDBG20(("dropping event received during reset"));
5339      return;
5340  }

5341  if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
5342      (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
5343      mptsas_log(mpt, CE_WARN, "No memory available"
5344      "for dispatch taskq");
5345      /*
5346       * Return the reply frame to the free queue.
5347       */
5348      ddi_put32(mpt->m_acc_free_queue_hdl,
5349      &((uint32_t *))(void *)
5350      mpt->m_free_queue)[mpt->m_free_index], reply_addr);
5351      (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
5352      DDI_DMA_SYNC_FORDEV);
5353      if (++mpt->m_free_index == mpt->m_free_queue_depth) {
5354          mpt->m_free_index = 0;
5355      }
5356  }

5357  ddi_put32(mpt->m_datap,
5358      &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
5359  }
5360  return;
5361 case MPI2_FUNCTION_DIAG_BUFFER_POST:
5362 /*
5363  * If SMID is 0, this implies that the reply is due to a
5364  * release function with a status that the buffer has been
5365  * released. Set the buffer flags accordingly.
5366  */
5367  if (SMID == 0) {
5368      iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
5369      &reply->IOCStatus);
5370      buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
5371      &((pMpI2DiagBufferPostReply_t)reply)->BufferType));
5372      if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
5373          pBuffer =
5374              &mpt->m_fw_diag_buffer_list[buffer_type];
5375          pBuffer->valid_data = TRUE;
5376          pBuffer->owned_by_firmware = FALSE;
5377          pBuffer->immediate = FALSE;
5378      }
5379  } else {
5380  /*
5381   * Normal handling of diag post reply with SMID.
5382   */
5383  cmd = slots->m_slot[SMID];
5384  }

5385  /*
5386   * print warning and return if the slot is empty
5387   */
5388  if (cmd == NULL) {
5389      mptsas_log(mpt, CE_WARN, "?NULL command for "
5390      "address reply in slot %d", SMID);
5391      return;
5392  }
5393  cmd->cmd_rfmp = reply_addr;
```

```

5395         cmd->cmd_flags |= CFLAG_FINISHED;
5396         cv_broadcast(&mpt->m_fw_diag_cv);
5397     }
5398     return;
5399 default:
5400     mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
5401     break;
5402 }

5403 /*
5404  * Return the reply frame to the free queue.
5405  */
5406 ddi_put32(mpt->m_acc_free_queue_hdl,
5407   &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
5408   reply_addr);
5409 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
5410   DDI_DMA_SYNC_FORDEV);
5411 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
5412     mpt->m_free_index = 0;
5413 }
5414 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
5415   mpt->m_free_index);

5416 if (cmd->cmd_flags & CFLAG_FW_CMD)
5417     return;

5418 if (cmd->cmd_flags & CFLAG_RETRY) {
5419     /*
5420      * The target returned QFULL or busy, do not add this
5421      * pkt to the doneq since the hba will retry
5422      * this cmd.
5423      *
5424      * The pkt has already been resubmitted in
5425      * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
5426      * Remove this cmd_flag here.
5427      */
5428     cmd->cmd_flags &= ~CFLAG_RETRY;
5429 } else {
5430     mptsas_doneq_add(mpt, cmd);
5431 }
5432

5433 #ifdef MPTSAS_DEBUG
5434 static uint8_t mptsas_last_sense[256];
5435 #endif

5441 static void
5442 mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
5443   mptsas_cmd_t *cmd)
5444 {
5445     uint8_t          scsi_status, scsi_state;
5446     uint16_t         ioc_status, cmd_rqs_len;
5447     uint32_t         xferred, sensecount, respondedata, loginfo = 0;
5448     struct scsi_pkt *pkt;
5449     struct scsi_arq_status *arqstat;
5450     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
5451     uint8_t          *sensedata = NULL;
5452     uint64_t         sas_wwn;
5453     uint8_t          phy;
5454     char             wnn_str[MPTSAS_WWN_STRLEN];

5455     scsi_status = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIStatus);
5456     ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
5457     scsi_state = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSISState);
5458     xferred = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->TransferCount);
5459     sensecount = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->SenseCount);

```

```

5461     respondedata = ddi_get32(mpt->m_acc_reply_frame_hdl,
5462       &reply->ResponseInfo);
5463

5464     if (ioc_status & MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
5465         sas_wwn = ptgt->m_addr.mta_wwn;
5466         phy = ptgt->m_physnum;
5467         if (sas_wwn == 0) {
5468             (void) sprintf(wwn_str, "p%x", phy);
5469         } else {
5470             (void) sprintf(wwn_str, "w%016PRIx64, sas_wwn");
5471         }
5472         loginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
5473       &reply->IOCLogInfo);
5474         mptsas_log(mpt, CE_NOTE,
5475           "?Log info 0x%x received for target %d %s.\n"
5476           "\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
5477           loginfo, Tgt(cmd), wnn_str, scsi_status, ioc_status,
5478           scsi_state);
5479     }

5480     NDBG31(("t\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
5481   scsi_status, ioc_status, scsi_state));

5482     pkt = CMD2PKT(cmd);
5483     *(pkt->pkt_scbp) = scsi_status;

5484     if (loginfo == 0x31170000) {
5485         /*
5486          * if loginfo PL_LOGININFO_CODE_IO_DEVICE_MISSING_DELAY_RETRY
5487          * 0x31170000 comes, that means the device missing delay
5488          * is in progressing, the command need retry later.
5489          */
5490         *(pkt->pkt_scbp) = STATUS_BUSY;
5491         return;
5492     }

5493     if ((scsi_state & MPI2_SCSI_STATE_NO_SCSI_STATUS) ||
5494       ((ioc_status & MPI2_IOCSTATUS_MASK) ==
5495        MPI2_IOCSTATUS_SCSI_DEVICE_NOT THERE)) {
5496         pkt->pkt_reason = CMD_INCOMPLETE;
5497         pkt->pkt_state |= STATE_GOT_BUS;
5498         if (ptgt->m_reset_delay == 0) {
5499             mptsas_set_throttle(mpt, ptgt,
5500               DRAIN_THROTTLE);
5501         }
5502         return;
5503     }

5504     if (scsi_state & MPI2_SCSI_STATE_RESPONSE_INFO_VALID) {
5505         respondedata &= 0x000000FF;
5506         if (respondedata & MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF) {
5507             mptsas_log(mpt, CE_NOTE, "Do not support the TLR\n");
5508             pkt->pkt_reason = CMD_TLR_OFF;
5509             return;
5510         }
5511     }

5512     switch (scsi_status) {
5513     case MPI2_SCSI_STATUS_CHECK_CONDITION:
5514         pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5515         arqstat = (void*)(pkt->pkt_scbp);
5516         arqstat->sts_rqpkt_status = *((struct scsi_status *)
5517           (pkt->pkt_scbp));
5518         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET |
5519           STATE_SENT_CMD | STATE_GOT_STATUS | STATE_ARQ_DONE);
5520     }

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

69

```

5527     if (cmd->cmd_flags & CFLAG_XARQ) {
5528         pkt->pkt_state |= STATE_XARQ_DONE;
5529     }
5530     if (pkt->pkt_resid != cmd->cmd_dmacount) {
5531         pkt->pkt_state |= STATE_XFERRED_DATA;
5532     }
5533     argstat->sts_rqpkt_reason = pkt->pkt_reason;
5534     argstat->sts_rqpkt_state = pkt->pkt_state;
5535     argstat->sts_rqpkt_state |= STATE_XFERRED_DATA;
5536     argstat->sts_rqpkt_statistics = pkt->pkt_statistics;
5537     sensedata = (uint8_t *)argstat->sts_sensedata;
5538     cmd_rqs_len = cmd->cmd_extrqlen ?
5539         cmd->cmd_extrqlen : cmd->cmd_rqslen;
5540     (void) ddi_dma_sync(mpt->m_dma_req_sense_hdl, 0, 0,
5541         DDI_DMA_SYNC_FORKERNEL);
5542 #ifdef MPTSAS_DEBUG
5543     bcopy(cmd->cmd_arg_buf, mptsas_last_sense,
5544         ((cmd_rqs_len >= sizeof (mptsas_last_sense)) ?
5545             sizeof (mptsas_last_sense):cmd_rqs_len));
5546 #endif
5547     bcopy((uchar_t *)cmd->cmd_arg_buf, sensedata,
5548         ((cmd_rqs_len >= sensecount) ? sensecount :
5549             cmd_rqs_len));
5550     argstat->sts_rqpkt_resid = (cmd_rqs_len - sensecount);
5551     cmd->cmd_flags |= CFLAG_CMDARQ;
5552     /*
5553      * Set proper status for pkt if autosense was valid
5554      */
5555     if (scsi_state & MPI2_SCSI_STATE_AUTOSENSE_VALID) {
5556         struct scsi_status zero_status = { 0 };
5557         argstat->sts_rqpkt_status = zero_status;
5558     }
5559
5560     /*
5561      * ASC=0x47 is parity error
5562      * ASC=0x48 is initiator detected error received
5563      */
5564     if ((scsi_sense_key(sensedata) == KEY_ABORTED_COMMAND) &&
5565         ((scsi_sense_asc(sensedata) == 0x47) ||
5566          (scsi_sense_asc(sensedata) == 0x48))) {
5567         mptsas_log(mpt, CE_NOTE, "Aborted_command!");
5568     }
5569
5570     /*
5571      * ASC/ASCQ=0x3F/0x0E means report_luns data changed
5572      * ASC/ASCQ=0x25/0x00 means invalid lun
5573      */
5574     if (((scsi_sense_key(sensedata) == KEY_UNIT_ATTENTION) &&
5575         (scsi_sense_asc(sensedata) == 0x3F) &&
5576         (scsi_sense_ascq(sensedata) == 0x0E)) ||
5577         ((scsi_sense_key(sensedata) == KEY_ILLEGAL_REQUEST) &&
5578         (scsi_sense_asc(sensedata) == 0x25) &&
5579         (scsi_sense_ascq(sensedata) == 0x00))) {
5580         mptsas_topo_change_list_t *topo_node = NULL;
5581
5582         topo_node = kmalloc_zalloc(
5583             sizeof (mptsas_topo_change_list_t),
5584             KM_NOSLEEP);
5585         if (topo_node == NULL) {
5586             mptsas_log(mpt, CE_NOTE, "No memory"
5587                         "resource for handle SAS dynamic"
5588                         "reconfigure.\n");
5589             break;
5590         }
5591         topo_node->mpt = mpt;
5592         topo_node->event = MPTSAS_DR_EVENT_RECONFIG_TARGET;

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.

70

```

topo_node->un.phymask = ptgt->m_addr.mta_phymask;
topo_node->devhdl = ptgt->m_devhdl;
topo_node->object = (void *)ptgt;
topo_node->flags = MPTSAS_TOPO_FLAG_LUN_ASSOCIATED;

if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
    mptsas_handle_dr,
    (void *)topo_node,
    DDI_NOSLEEP)) != DDI_SUCCESS) {
    kmem_free(topo_node,
        sizeof (mptsas_topo_change_list_t));
    mptsas_log(mpt, CE_NOTE, "mptsas start taskq"
        "for handle SAS dynamic reconfigure"
        "failed. \n");
}
}
break;
case MPI2_SCSI_STATUS_GOOD:
switch (ioc_status & MPI2_IOCSTATUS_MASK) {
case MPI2_IOCSTATUS_SCSI_DEVICE_NOT THERE:
    pkt->pkt_reason = CMD_DEV_GONE;
    pkt->pkt_state |= STATE_GOT_BUS;
    if (ptgt->m_reset_delay == 0) {
        mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
    }
    NDBG31(("lost disk for target%d, command:%x",
        Tgt(cmd), pkt->pkt_cdbp[0]));
    break;
case MPI2_IOCSTATUS_SCSI_DATA_OVERRUN:
    NDBG31(("data overrun: xferred=%d", xferred));
    NDBG31(("dmacount=%d", cmd->cmd_dmacount));
    pkt->pkt_reason = CMD_DATA_OVR;
    pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
        | STATE_SENT_CMD | STATE_GOT_STATUS
        | STATE_XFERRED_DATA);
    pkt->pkt_resid = 0;
    break;
case MPI2_IOCSTATUS_SCSI_RESIDUAL_MISMATCH:
case MPI2_IOCSTATUS_SCSI_DATA_UNDERRUN:
    NDBG31(("data underrun: xferred=%d", xferred));
    NDBG31(("dmacount=%d", cmd->cmd_dmacount));
    pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
        | STATE_SENT_CMD | STATE_GOT_STATUS);
    pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
    if (pkt->pkt_resid != cmd->cmd_dmacount) {
        pkt->pkt_state |= STATE_XFERRED_DATA;
    }
    break;
case MPI2_IOCSTATUS_SCSI_TASK_TERMINATED:
    if (cmd->cmd_active_expiration <= gethrtime()) {
        /*
         * When timeout requested, propagate
         * proper reason and statistics to
         * target drivers.
         */
        mptsas_set_pkt_reason(mpt, cmd, CMD_TIMEOUT,
            STAT_BUS_RESET | STAT_TIMEOUT);
    } else {
        mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
            STAT_BUS_RESET);
    }
    break;
case MPI2_IOCSTATUS_SCSI_IOC_TERMINATED:
case MPI2_IOCSTATUS_SCSI_EXT_TERMINATED:
    mptsas_set_pkt_reason(mpt,
        cmd, CMD_RESET, STAT_DEV_RESET);
}
}

```

```

5659         break;
5660     case MPI2_IOCSTATUS_SCSI_IO_DATA_ERROR:
5661     case MPI2_IOCSTATUS_SCSI_PROTOCOL_ERROR:
5662         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET);
5663         mptsas_set_pkt_reason(mpt,
5664             cmd, CMD_TERMINATED, STAT_TERMINATED);
5665         break;
5666     case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
5667     case MPI2_IOCSTATUS_BUSY:
5668         /*
5669          * set throttles to drain
5670          */
5671         for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
5672             ptgt = refhash_next(mpt->m_targets, ptgt)) {
5673             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5674         }
5675         /*
5676          * retry command
5677          */
5678         cmd->cmd_flags |= CFLAG_RETRY;
5679         cmd->cmd_pkt_flags |= FLAG_HEAD;
5680
5681         (void) mptsas_accept_pkt(mpt, cmd);
5682         break;
5683     default:
5684         mptsas_log(mpt, CE_WARN,
5685             "unknown ioc_status = %x\n", ioc_status);
5686         mptsas_log(mpt, CE_CONT, "scsi_state = %x, transfer "
5687             "count = %x, scsi_status = %x", scsi_state,
5688             xferred, scsi_status);
5689         break;
5690     }
5691     break;
5692 case MPI2_SCSI_STATUS_TASK_SET_FULL:
5693     mptsas_handle_qfull(mpt, cmd);
5694     break;
5695 case MPI2_SCSI_STATUS_BUSY:
5696     NDBG31(("scsi_status busy received"));
5697     break;
5698 case MPI2_SCSI_STATUS_RESERVATION_CONFLICT:
5699     NDBG31(("scsi_status reservation conflict received"));
5700     break;
5701 default:
5702     mptsas_log(mpt, CE_WARN, "scsi_status=%x, ioc_status=%x\n",
5703                 scsi_status, ioc_status);
5704     mptsas_log(mpt, CE_WARN,
5705                 "mptsas_process_intr: invalid scsi status\n");
5706     break;
5707 }
5708 }
5709 }

5711 static void
5712 mptsas_check_task_mgt(mptsas_t *mpt, pMpI2SCSIManagementReply_t reply,
5713     mptsas_cmd_t *cmd)
5714 {
5715     uint8_t          task_type;
5716     uint16_t         ioc_status;
5717     uint32_t         log_info;
5718     uint16_t         dev_handle;
5719     struct scsi_pkt *pkt = CMD2PKT(cmd);

5721     task_type = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->TaskType);
5722     ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
5723     log_info = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->IOCLogInfo);
5724     dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->DevHandle);

```

```

5726     if (ioc_status != MPI2_IOCSTATUS_SUCCESS) {
5727         mptsas_log(mpt, CE_WARN, "mptsas_check_task_mgt: Task 0x%x "
5728             "failed. IOCStatus=0x%x IOCLogInfo=0x%x target=%d\n",
5729             task_type, ioc_status, log_info, dev_handle);
5730         pkt->pkt_reason = CMD_INCOMPLETE;
5731         return;
5732     }

5733     switch (task_type) {
5734     case MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK:
5735     case MPI2_SCSITASKMGMT_TASKTYPE_CLEAR_TASK_SET:
5736     case MPI2_SCSITASKMGMT_TASKTYPE_QUERY_TASK:
5737     case MPI2_SCSITASKMGMT_TASKTYPE_CLR_ACA:
5738     case MPI2_SCSITASKMGMT_TASKTYPE_QRY_TASK_SET:
5739     case MPI2_SCSITASKMGMT_TASKTYPE_QRY_UNIT_ATTENTION:
5740         break;
5741     case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
5742     case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
5743     case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
5744         /*
5745          * Check for invalid DevHandle of 0 in case application
5746          * sends bad command. DevHandle of 0 could cause problems.
5747          */
5748         if (dev_handle == 0) {
5749             mptsas_log(mpt, CE_WARN, "!Can't flush target with"
5750                         " DevHandle of 0.");
5751         } else {
5752             mptsas_flush_target(mpt, dev_handle, Lun(cmd),
5753                                 task_type);
5754         }
5755         break;
5756     default:
5757         mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
5758                     task_type);
5759         mptsas_log(mpt, CE_WARN, "ioc status = %x", ioc_status);
5760         break;
5761     }
5762 }

5763 }

5764 static void
5765 mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg)
5766 {
5767     mptsas_t          *mpt = arg->mpt;
5768     uint64_t           t = arg->t;
5769     mptsas_cmd_t       *cmd;
5770     struct scsi_pkt    *pkt;
5771     mptsas_doneq_thread_list_t *item = &mpt->m_doneq_thread_id[t];
5772

5773     mutex_enter(&item->mutex);
5774     while (item->flag & MPTSAS_DONEQ_THREAD_ACTIVE) {
5775         if (!item->doneq) {
5776             cv_wait(&item->cv, &item->mutex);
5777         }
5778         pkt = NULL;
5779         if ((cmd = mptsas_doneq_thread_rm(mpt, t)) != NULL) {
5780             cmd->cmd_flags |= CFLAG_COMPLETED;
5781             pkt = CMD2PKT(cmd);
5782         }
5783         mutex_exit(&item->mutex);
5784         if (pkt) {
5785             mptsas_pkt_comp(pkt, cmd);
5786         }
5787         mutex_enter(&item->mutex);
5788     }
5789     mutex_exit(&item->mutex);
5790 }

```

```

5791     mutex_enter(&mpt->m_doneq_mutex);
5792     mpt->m_doneq_thread_n--;
5793     cv_broadcast(&mpt->m_doneq_thread_cv);
5794     mutex_exit(&mpt->m_doneq_mutex);
5795 }

5798 /*
5799  * mpt interrupt handler.
5800 */
5801 static uint_t
5802 mptsas_intr(caddr_t arg1, caddr_t arg2)
5803 {
5804     mptsas_t          *mpt = (void *)arg1;
5805     pMpI2ReplyDescriptorsUnion_t    reply_desc_union;
5806     uchar_t           did_reply = FALSE;
5807
5808     NDBG1(("mptsas_intr: arg1 0x%p arg2 0x%p", (void *)arg1, (void *)arg2));
5809
5810     mutex_enter(&mpt->m_mutex);
5811
5812     /*
5813      * If interrupts are shared by two channels then check whether this
5814      * interrupt is genuinely for this channel by making sure first the
5815      * chip is in high power state.
5816      */
5817     if ((mpt->m_options & MPTSAS_OPT_PM) &&
5818         (mpt->m_power_level != PM_LEVEL_D0)) {
5819         mutex_exit(&mpt->m_mutex);
5820         return (DDI_INTR_UNCLAIMED);
5821     }
5822
5823     /*
5824      * If polling, interrupt was triggered by some shared interrupt because
5825      * IOC interrupts are disabled during polling, so polling routine will
5826      * handle any replies. Considering this, if polling is happening,
5827      * return with interrupt unclaimed.
5828      */
5829     if (mpt->m_polled_intr) {
5830         mutex_exit(&mpt->m_mutex);
5831         mptsas_log(mpt, CE_WARN, "mpt_sas: Unclaimed interrupt");
5832         return (DDI_INTR_UNCLAIMED);
5833     }
5834
5835     /*
5836      * Read the istat register.
5837      */
5838     if ((INTPENDING(mpt)) != 0) {
5839         /*
5840          * read fifo until empty.
5841          */
5842 #ifndef __lock_lint
5843         _NOTE(CONSTCOND)
5844 #endif
5845         while (TRUE) {
5846             (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5847                                 DDI_DMA_SYNC_FORCPU);
5848             reply_desc_union = (pMpI2ReplyDescriptorsUnion_t)
5849                           MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);
5850
5851             if (ddi_get32(mpt->m_acc_post_queue_hdl,
5852                           &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5853                 ddi_get32(mpt->m_acc_post_queue_hdl,
5854                           &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5855                 break;
5856             }
5857         }
5858     }
5859 }

```

```

5858 /*
5859  * The reply is valid, process it according to its
5860  * type. Also, set a flag for updating the reply index
5861  * after they've all been processed.
5862  */
5863 did_reply = TRUE;

5864 mptsas_process_intr(mpt, reply_desc_union);

5865 /*
5866  * Increment post index and roll over if needed.
5867  */
5868 if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5869     mpt->m_post_index = 0;
5870 }
5871
5872 /*
5873  * Update the global reply index if at least one reply was
5874  * processed.
5875  */
5876 if (did_reply) {
5877     ddi_put32(mpt->m_datap,
5878               &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
5879 }
5880 else {
5881     mutex_exit(&mpt->m_mutex);
5882     return (DDI_INTR_UNCLAIMED);
5883 }
5884 NDBG1(("mptsas_intr complete"));

5885 /*
5886  * If no helper threads are created, process the doneq in ISR. If
5887  * helpers are created, use the doneq length as a metric to measure the
5888  * load on the interrupt CPU. If it is long enough, which indicates the
5889  * load is heavy, then we deliver the IO completions to the helpers.
5890  * This measurement has some limitations, although it is simple and
5891  * straightforward and works well for most of the cases at present.
5892  */
5893 if (!mpt->m_doneq_thread_n ||
5894     (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)) {
5895     mptsas_doneq_empty(mpt);
5896 } else {
5897     mptsas_deliver_doneq_thread(mpt);
5898 }
5899
5900 /*
5901  * If there are queued cmd, start them now.
5902  */
5903 if (mpt->m_waitq != NULL) {
5904     mptsas_restart_waitq(mpt);
5905 }
5906
5907 mutex_exit(&mpt->m_mutex);
5908 return (DDI_INTR_CLAIMED);
5909 }

5910 static void
5911 mptsas_process_intr(mptsas_t *mpt,
5912                      pMpI2ReplyDescriptorsUnion_t reply_desc_union)
5913 {
5914     uint8_t reply_type;
5915
5916     ASSERT(mutex_owned(&mpt->m_mutex));
5917
5918     {
5919         uint8_t reply_type;
5920
5921         ASSERT(mutex_owned(&mpt->m_mutex));
5922
5923         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5924                       &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5925             ddi_get32(mpt->m_acc_post_queue_hdl,
5926                       &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5927             break;
5928         }
5929     }
5930 }

```

```

5923     /*
5924      * The reply is valid, process it according to its
5925      * type. Also, set a flag for updated the reply index
5926      * after they've all been processed.
5927     */
5928     reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5929     &reply_desc_union->Default.ReplyFlags);
5930     reply_type &= MPI2_RPY_DESCRIPTOR_FLAGS_TYPE_MASK;
5931     if (reply_type == MPI2_RPY_DESCRIPTOR_FLAGS_SCSI_IO_SUCCESS ||
5932         reply_type == MPI25_RPY_DESCRIPTOR_FLAGS_FAST_PATH_SCSI_IO_SUCCESS) {
5933         mptsas_handle_scsi_io_success(mpt, reply_desc_union);
5934     } else if (reply_type == MPI2_RPY_DESCRIPTOR_FLAGS_ADDRESS_REPLY) {
5935         mptsas_handle_address_reply(mpt, reply_desc_union);
5936     } else {
5937         mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
5938         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5939     }
5940
5941     /*
5942      * Clear the reply descriptor for re-use and increment
5943      * index.
5944     */
5945     ddi_put64(mpt->m_acc_post_queue_hdl,
5946     &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
5947     0xFFFFFFFFFFFFFF);
5948     (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5949     DDI_DMA_SYNC_FORDEV);
5950 }
5951
5952 /*
5953  * handle qfull condition
5954 */
5955 static void
5956 mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd)
5957 {
5958     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
5959
5960     if ((++cmd->cmd_qfull_retries > ptgt->m_qfull_retries) ||
5961         (ptgt->m_qfull_retries == 0)) {
5962         /*
5963          * We have exhausted the retries on QFULL, or,
5964          * the target driver has indicated that it
5965          * wants to handle QFULL itself by setting
5966          * qfull-retries capability to 0. In either case
5967          * we want the target driver's QFULL handling
5968          * to kick in. We do this by having pkt_reason
5969          * as CMD_CMPLT and pkt_scqp as STATUS_QFULL.
5970         */
5971         mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5972     } else {
5973         if (ptgt->m_reset_delay == 0) {
5974             ptgt->m_t_throttle =
5975                 max((ptgt->m_t_ncmds - 2), 0);
5976         }
5977
5978         cmd->cmd_pkt_flags |= FLAG_HEAD;
5979         cmd->cmd_flags &= ~(CFLAG_TRANFLAG);
5980         cmd->cmd_flags |= CFLAG_RETRY;
5981
5982         (void) mptsas_accept_pkt(mpt, cmd);
5983
5984         /*
5985          * when target gives queue full status with no commands
5986          * outstanding (m_t_ncmds == 0), throttle is set to 0
5987          * (HOLD_THROTTLE), and the queue full handling start
5988          * (see psarc/1994/313); if there are commands outstanding,

```

```

5989             * throttle is set to (m_t_ncmds - 2)
5990             */
5991             if (ptgt->m_t_throttle == HOLD_THROTTLE) {
5992                 /*
5993                  * By setting throttle to QFULL_THROTTLE, we
5994                  * avoid submitting new commands and in
5995                  * mptsas_restart_cmd find out slots which need
5996                  * their throttles to be cleared.
5997                 */
5998                 mptsas_set_throttle(mpt, ptgt, QFULL_THROTTLE);
5999                 if (mpt->m_restart_cmd_timeid == 0) {
6000                     mpt->m_restart_cmd_timeid =
6001                         timeout(mptsas_restart_cmd, mpt,
6002                                 ptgt->m_qfull_retry_interval);
6003                 }
6004             }
6005         }
6006     }
6007
6008 mptsas_phymask_t
6009 mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport)
6010 {
6011     mptsas_phymask_t phy_mask = 0;
6012     uint8_t i = 0;
6013
6014     NDBG20(("mptsas%d physport_to_phymask enter", mpt->m_instance));
6015
6016     ASSERT(mutex_owned(&mpt->m_mutex));
6017
6018     /*
6019      * If physport is 0xFF, this is a RAID volume. Use phymask of 0.
6020     */
6021     if (physport == 0xFF) {
6022         return (0);
6023     }
6024
6025     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
6026         if (mpt->m_phy_info[i].attached_devhdl &&
6027             (mpt->m_phy_info[i].phy_mask != 0) &&
6028             (mpt->m_phy_info[i].port_num == physport)) {
6029             phy_mask = mpt->m_phy_info[i].phy_mask;
6030             break;
6031         }
6032     }
6033     NDBG20(("mptsas%d physport_to_phymask:physport :%x phymask :%x, ",
6034             mpt->m_instance, physport, phy_mask));
6035
6036     return (phy_mask);
6037
6038 /*
6039  * mpt free device handle after device gone, by use of passthrough
6040 */
6041 static int
6042 mptsas_free_devhdl(mptsas_t *mpt, uint16_t devhdl)
6043 {
6044     Mpi2SasIoUnitControlRequest_t req;
6045     Mpi2SasIoUnitControlReply_t rep;
6046     int ret;
6047
6048     ASSERT(mutex_owned(&mpt->m_mutex));
6049
6050     /*
6051      * Need to compose a SAS IO Unit Control request message
6052      * and call mptsas_do_passthru() function
6053     */
6054     bzero(&req, sizeof (req));

```

```

6055     bzero(&rep, sizeof(rep));
6056
6057     req.Function = MPI2_FUNCTION_SAS_IO_UNIT_CONTROL;
6058     req.Operation = MPI2_SAS_OP_REMOVE_DEVICE;
6059     req.DevHandle = LE_16(devhdl);
6060
6061     ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
6062                               sizeof(req), sizeof(rep), NULL, 0, NULL, 0, 60, FKIOCTL);
6063     if (ret != 0) {
6064         cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
6065                 "Control error %d", ret);
6066         return (DDI_FAILURE);
6067     }
6068
6069     /* do passthrough success, check the ioc status */
6070     if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
6071         cmn_err(CE_WARN, "mptsas_free_devhdl: passthru SAS IO Unit "
6072                 "Control IOCStatus %d", LE_16(rep.IOCStatus));
6073         return (DDI_FAILURE);
6074     }
6075
6076     return (DDI_SUCCESS);
6077 }
6078
6079 static void
6080 mptsas_update_phymask(mptsas_t *mpt)
6081 {
6082     mptsas_phymask_t mask = 0, phy_mask;
6083     char          *phy_mask_name;
6084     uint8_t        current_port;
6085     int           i, j;
6086
6087     NDBG20(("mptsas%d update phymask ", mpt->m_instance));
6088
6089     ASSERT(mutex_owned(&mpt->m_mutex));
6090
6091     (void) mptsas_get_sas_io_unit_page(mpt);
6092
6093     phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
6094
6095     for (i = 0; i < mpt->m_num_phys; i++) {
6096         phy_mask = 0x00;
6097
6098         if (mpt->m_phy_info[i].attached_devhdl == 0)
6099             continue;
6100
6101         bzero(phy_mask_name, sizeof(phy_mask_name));
6102
6103         current_port = mpt->m_phy_info[i].port_num;
6104
6105         if ((mask & (1 << i)) != 0)
6106             continue;
6107
6108         for (j = 0; j < mpt->m_num_phys; j++) {
6109             if (mpt->m_phy_info[j].attached_devhdl &&
6110                 (mpt->m_phy_info[j].port_num == current_port)) {
6111                 phy_mask |= (1 << j);
6112             }
6113         }
6114         mask = mask | phy_mask;
6115
6116         for (j = 0; j < mpt->m_num_phys; j++) {
6117             if ((phy_mask >> j) & 0x01) {
6118                 mpt->m_phy_info[j].phy_mask = phy_mask;
6119             }
6120         }
6121     }
}

```

```

6122
6123     (void) sprintf(phy_mask_name, "%x", phy_mask);
6124     mutex_exit(&mpt->m_mutex);
6125
6126     /*
6127      * register a iport, if the port has already been existed
6128      * * SCSA will do nothing and just return.
6129      */
6130     (void) scsi_hba_iport_register(mpt->m_dip, phy_mask_name);
6131     mutex_enter(&mpt->m_mutex);
6132
6133     kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
6134     NDBG20(("mptsas%d update phymask return", mpt->m_instance));
6135
6136     /*
6137      * mptsas_handle_dr is a task handler for DR, the DR action includes:
6138      * 1. Directly attached Device Added/Removed.
6139      * 2. Expander Device Added/Removed.
6140      * 3. Indirectly Attached Device Added/Expander.
6141      * 4. LUNs of a existing device status change.
6142      * 5. RAID volume created/deleted.
6143      * 6. Member of RAID volume is released because of RAID deletion.
6144      * 7. Physical disks are removed because of RAID creation.
6145      */
6146     static void
6147     mptsas_handle_dr(void *args) {
6148         mptsas_topo_change_list_t    *topo_node = NULL;
6149         mptsas_topo_change_list_t    *save_node = NULL;
6150         mptsas_t                    *mpt;
6151         dev_info_t                  *parent = NULL;
6152         mptsas_phymask_t            *mpt;
6153         char                         phy_mask_name;
6154         uint8_t                      flags = 0, physport = 0xff;
6155         uint8_t                      port_update = 0;
6156         uint_t                        event;
6157
6158         topo_node = (mptsas_topo_change_list_t *)args;
6159
6160         mpt = topo_node->mpt;
6161         event = topo_node->event;
6162         flags = topo_node->flags;
6163
6164         phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
6165
6166         NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));
6167
6168         switch (event) {
6169         case MPTSAS_DR_EVENT_RECONFIG_TARGET:
6170             if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6171                 (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
6172                 (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6173                 /*
6174                  * Direct attached or expander attached device added
6175                  * into system or a Phys Disk that is being unhidden.
6176                  */
6177                 port_update = 1;
6178             }
6179             break;
6180         case MPTSAS_DR_EVENT_RECONFIG_SMP:
6181             /*
6182              * New expander added into system, it must be the head
6183              * of topo_change_list_t
6184              */
6185             port_update = 1;
6186             break;
}

```

```

6187     default:
6188         port_update = 0;
6189         break;
6190     }
6191     /*
6192      * All cases port_update == 1 may cause initiator port form change
6193      */
6194     mutex_enter(&mpt->m_mutex);
6195     if (mpt->m_port_chng && port_update) {
6196         /*
6197          * mpt->m_port_chng flag indicates some PHYS of initiator
6198          * port have changed to online. So when expander added or
6199          * directly attached device online event come, we force to
6200          * update port information by issuing SAS IO Unit Page and
6201          * update PHYMASKs.
6202         */
6203         (void) mptsas_update_phymask(mpt);
6204         mpt->m_port_chng = 0;
6205     }
6206     mutex_exit(&mpt->m_mutex);
6207     while (topo_node) {
6208         phy mask = 0;
6209         if (parent == NULL) {
6210             physport = topo_node->un.physport;
6211             event = topo_node->event;
6212             flags = topo_node->flags;
6213             if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
6214                           MPTSAS_DR_EVENT_OFFLINE_SMP)) {
6215                 /*
6216                  * For all offline events, phy mask is known
6217                  */
6218                 phy mask = topo_node->un.phymask;
6219                 goto find_parent;
6220             }
6221             if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6222                 goto handle_topo_change;
6223             }
6224             if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
6225                 phy mask = topo_node->un.phymask;
6226                 goto find_parent;
6227             }
6228
6229             if ((flags ==
6230                           MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
6231                           (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
6232                 /*
6233                  * There is no any field in IR_CONFIG_CHANGE
6234                  * event indicate physport/phynum, let's get
6235                  * parent after SAS Device Page0 request.
6236                  */
6237                 goto handle_topo_change;
6238             }
6239
6240             mutex_enter(&mpt->m_mutex);
6241             if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6242                 /*
6243                  * If the direct attached device added or a
6244                  * phys disk is being unhidden, argument
6245                  * physport actually is PHY#, so we have to get
6246                  * phy mask according PHY#.
6247                 */
6248                 physport = mpt->m_phy_info[physport].port_num;
6249             }
6250
6251         /*

```

```

6253             * Translate physport to phy mask so that we can search
6254             * parent dip.
6255             */
6256             phy mask = mptsas_physport_to_phymask(mpt,
6257                                           physport);
6258             mutex_exit(&mpt->m_mutex);
6259
6260     find_parent:
6261     bzero(phy mask_name, MPTSAS_MAX_PHYS);
6262     /*
6263      * For RAID topology change node, write the iport name
6264      * as v0.
6265     */
6266     if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
6267         (void) sprintf(phy mask_name, "v0");
6268     } else {
6269         /*
6270          * phy mask can be 0 if the drive has been
6271          * pulled by the time an add event is
6272          * processed. If phy mask is 0, just skip this
6273          * event and continue.
6274         */
6275         if (phy mask == 0) {
6276             mutex_enter(&mpt->m_mutex);
6277             save_node = topo_node;
6278             topo_node = topo_node->next;
6279             ASSERT(save_node);
6280             kmem_free(save_node,
6281                       sizeof (mptsas topo_change_list_t));
6282             mutex_exit(&mpt->m_mutex);
6283
6284             parent = NULL;
6285             continue;
6286         }
6287         (void) sprintf(phy mask_name, "%x", phy mask);
6288     }
6289     parent = scsi_hba_iport_find(mpt->m_dip,
6290                                  phy mask_name);
6291     if (parent == NULL) {
6292         mptsas_log(mpt, CE_WARN, "Failed to find an "
6293                    "iport, should not happen!");
6294     }
6295
6296     ASSERT(parent);
6297
6298     handle_topo_change:
6299
6300     mutex_enter(&mpt->m_mutex);
6301     /*
6302      * If HBA is being reset, don't perform operations depending
6303      * on the IOC. We must free the topo list, however.
6304     */
6305     if (!mpt->m_in_reset)
6306         mptsas_handle_topo_change(topo_node, parent);
6307     else
6308         NDBG20(("skipping topo change received during reset"));
6309     save_node = topo_node;
6310     topo_node = topo_node->next;
6311     ASSERT(save_node);
6312     kmem_free(save_node, sizeof (mptsas topo_change_list_t));
6313
6314     if (((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6315          (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
6316          (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) ||
6317
6318

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c

81

```

6319         /*
6320          * If direct attached device associated, make sure
6321          * reset the parent before start the next one. But
6322          * all devices associated with expander shares the
6323          * parent. Also, reset parent if this is for RAID.
6324          */
6325         parent = NULL;
6326     }
6327 }
6328 out:
6329     kmem_free(phy_mask_name, MPTAS_MAX_PHYS);
6330 }

6332 static void
6333 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
6334     dev_info_t *parent)
6335 {
6336     mptsas_target_t *ptgt = NULL;
6337     mptsas_smp_t *psmp = NULL;
6338     mptsas_t *mpt = (void *)topo_node->mpt;
6339     uint16_t devhdl;
6340     uint16_t attached_devhdl;
6341     uint64_t sas_wwn = 0;
6342     int rval = 0;
6343     uint32_t page_address;
6344     uint8_t phy, flags;
6345     char *addr = NULL;
6346     dev_info_t *lunidp;
6347     int circ = 0, circ1 = 0;
6348     char attached_wwnstr[MPTAS_WWN_STRLEN];

6350     NDBG20(("mptsas%d handle_topo_change enter, devhdl 0x%x",
6351         "event 0x%lx, flags 0x%lx", mpt->m_instance, topo_node->devhdl,
6352         topo_node->event, topo_node->flags));

6354     ASSERT(mutex_owned(&mpt->m_mutex));

6356     switch (topo_node->event) {
6357     case MPTAS_DR_EVENT_RECONFIG_TARGET:
6358     {
6359         char *phy_mask_name;
6360         mptsas_phymask_t phymask = 0;

6362         if (topo_node->flags == MPTAS_TOPO_FLAG_RAID_ASSOCIATED) {
6363             /*
6364              * Get latest RAID info.
6365              */
6366             (void) mptsas_get_raid_info(mpt);
6367             ptgt = rehash_linear_search(mpt->m_targets,
6368                                         mptsas_target_eval_devhdl, &topo_node->devhdl);
6369             if (ptgt == NULL)
6370                 break;
6371         } else {
6372             ptgt = (void *)topo_node->object;
6373         }

6375         if (ptgt == NULL) {
6376             /*
6377              * If a Phys Disk was deleted, RAID info needs to be
6378              * updated to reflect the new topology.
6379              */
6380             (void) mptsas_get_raid_info(mpt);

6382             /*
6383              * Get sas device page 0 by DevHandle to make sure if
6384              * SSP/SATA end device exist.

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.

82

```

6451      */
6452      if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6453          bzero(attached_wwnstr,
6454              sizeof (attached_wwnstr));
6455          (void) sprintf(attached_wwnstr, "w%016"PRIx64,
6456                          ptgt->m_addr.mta_wwn);
6457          if (ddi_prop_update_string(DDI_DEV_T_NONE,
6458                                      parent,
6459                                      SCSI_ADDR_PROP_ATTACHED_PORT,
6460                                      attached_wwnstr)
6461                                      != DDI_PROP_SUCCESS) {
6462              (void) ddi_prop_remove(DDI_DEV_T_NONE,
6463                                      parent,
6464                                      SCSI_ADDR_PROP_ATTACHED_PORT);
6465              mptsas_log(mpt, CE_WARN, "Failed to"
6466                         "attach-ported props");
6467              return;
6468          }
6469          if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6470                                  MPTSAS_NUM_PHYS, 1) !=
6471              DDI_PROP_SUCCESS) {
6472              (void) ddi_prop_remove(DDI_DEV_T_NONE,
6473                                      parent, MPTSAS_NUM_PHYS);
6474              mptsas_log(mpt, CE_WARN, "Failed to"
6475                         "create num-phys props");
6476              return;
6477          }
6478          /*
6479          * Update PHY info for smhba
6480          */
6481          mutex_enter(&mpt->m_mutex);
6482          if (mptsas_smhba_phy_init(mpt)) {
6483              mutex_exit(&mpt->m_mutex);
6484              mptsas_log(mpt, CE_WARN, "mptsas phy"
6485                         " update failed");
6486              return;
6487          }
6488          mutex_exit(&mpt->m_mutex);
6489
6490          /*
6491          * topo_node->un.physport is really the PHY#
6492          * for direct attached devices
6493          */
6494          mptsas_smhba_set_one_phy_props(mpt, parent,
6495                                          topo_node->un.physport, &attached_devhdl);
6496
6497          if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6498                                  MPTSAS_VIRTUAL_PORT, 0) !=
6499              DDI_PROP_SUCCESS) {
6500              (void) ddi_prop_remove(DDI_DEV_T_NONE,
6501                                      parent, MPTSAS_VIRTUAL_PORT);
6502              mptsas_log(mpt, CE_WARN,
6503                         "mptsas virtual-port"
6504                         " port prop update failed");
6505              return;
6506          }
6507      }
6508      mutex_enter(&mpt->m_mutex);
6509
6510      NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
6511              "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
6512              ptgt->m_addr.mta_phymask));
6513      break;
6514
6515  }

```

```

6517      case MPTSAS_DR_EVENT_OFFLINE_TARGET:
6518      {
6519          devhdl = topo_node->devhdl;
6520          ptgt = rehash_linear_search(mpt->m_targets,
6521                                      mptsas_target_eval_devhdl, &devhdl);
6522          if (ptgt == NULL)
6523              break;
6524
6525          sas_wwn = ptgt->m_addr.mta_wwn;
6526          phy = ptgt->m_physnum;
6527
6528          addr = kmalloc(SCSI_MAXNAMELEN, KM_SLEEP);
6529
6530          if (sas_wwn) {
6531              (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
6532          } else {
6533              (void) sprintf(addr, "p%x", phy);
6534          }
6535          ASSERT(ptgt->m_devhdl == devhdl);
6536
6537          if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
6538              (topo_node->flags ==
6539               MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6540              /*
6541              * Get latest RAID info if RAID volume status changes
6542              * or Phys Disk status changes
6543              */
6544              (void) mptsas_get_raid_info(mpt);
6545
6546          /*
6547          * Abort all outstanding command on the device
6548          */
6549          rval = mptsas_do_scsi_reset(mpt, devhdl);
6550          if (rval) {
6551              NDBG20(("mptsas%d handle_topo_change to reset target "
6552                     "before offline devhdl:%x, phymask:%x, rval:%x",
6553                     mpt->m_instance, ptgt->m_devhdl,
6554                     ptgt->m_addr.mta_phymask, rval));
6555
6556          mutex_exit(&mpt->m_mutex);
6557
6558          ndi_devi_enter(scsi_vhci_dip, &circ);
6559          ndi_devi_enter(parent, &circ1);
6560          rval = mptsas_offline_target(parent, addr);
6561          ndi_devi_exit(parent, circ1);
6562          ndi_devi_exit(scsi_vhci_dip, circ);
6563
6564          NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6565                     "phymask:%x, rval:%x", mpt->m_instance,
6566                     ptgt->m_devhdl, ptgt->m_addr.mta_phymask, rval));
6567
6568          kmem_free(addr, SCSI_MAXNAMELEN);
6569
6570          /*
6571          * Clear parent's props for SMHBA support
6572          */
6573          flags = topo_node->flags;
6574          if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6575              bzero(attached_wwnstr, sizeof (attached_wwnstr));
6576              if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6577                                         SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6578                  DDI_PROP_SUCCESS) {
6579                  (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6580                                         SCSI_ADDR_PROP_ATTACHED_PORT);
6581                  mptsas_log(mpt, CE_WARN, "mptsas attached port "
6582                             "prop update failed");
6583              }
6584          }
6585      }
6586
6587  }

```



```

6715             DDI_PROP_SUCCESS) {
6716                 (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6717                     MPTSAS_VIRTUAL_PORT);
6718                 mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6719                         "prop update failed");
6720                 return;
6721             }
6722             /*
6723              * Check whether the smp connected to the iport,
6724              */
6725             if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6726                     MPTSAS_NUM_PHYS, 0) != DDI_PROP_SUCCESS) {
6727                 (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6728                     MPTSAS_NUM_PHYS);
6729                 mptsas_log(mpt, CE_WARN, "mptsas num phys"
6730                         "prop update failed");
6731                 return;
6732             }
6733             /*
6734              * Clear parent's attached-port props
6735              */
6736             bzero(attached_wwnstr, sizeof (attached_wwnstr));
6737             if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6738                     SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) != DDI_PROP_SUCCESS) {
6739                 (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6740                     SCSI_ADDR_PROP_ATTACHED_PORT);
6741                 mptsas_log(mpt, CE_WARN, "mptsas attached port "
6742                         "prop update failed");
6743                 return;
6744             }
6745         }
6746     }
6747
6748     mutex_enter(&mpt->m_mutex);
6749     NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6750             "'rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
6751     if (rval == DDI_SUCCESS) {
6752         refhash_remove(mpt->m_smp_targets, psmp);
6753     } else {
6754         psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
6755     }
6756
6757     bzero(attached_wwnstr, sizeof (attached_wwnstr));
6758
6759     break;
6760 }
6761 default:
6762     return;
6763 }
6764 }
6765 }

6766 /*
6767  * Record the event if its type is enabled in mpt instance by ioctl.
6768  */
6769 */
6770 static void
6771 mptsas_record_event(void *args)
6772 {
6773     m_replayh_arg_t          *replayh_arg;
6774     PMpi2EventNotificationReply_t eventreply;
6775     uint32_t                  event, rfm;
6776     mptsas_t                  *mpt;
6777     int                       i, j;
6778     uint16_t                  event_data_len;
6779     boolean_t                 sendAEN = FALSE;

```

```

6781     replayh_arg = (m_replayh_arg_t *)args;
6782     rfm = replayh_arg->rfm;
6783     mpt = replayh_arg->mpt;
6784
6785     eventreply = (pMpi2EventNotificationReply_t)
6786         (mpt->m_reply_frame + (rfm -
6787             (mpt->m_reply_frame_dma_addr & 0xffffffff)));
6788     event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6789     /*
6790      * Generate a system event to let anyone who cares know that a
6791      * LOG_ENTRY_ADDED event has occurred. This is sent no matter what the
6792      * event mask is set to.
6793      */
6794     if (event == MPI2_EVENT_LOG_ENTRY_ADDED) {
6795         sendAEN = TRUE;
6796     }

6797     /*
6798      * Record the event only if it is not masked. Determine which dword
6799      * and bit of event mask to test.
6800      */
6801     i = (uint8_t)(event / 32);
6802     j = (uint8_t)(event % 32);
6803
6804     if ((i < 4) && ((1 << j) & mpt->m_event_mask[i])) {
6805         i = mpt->m_event_index;
6806         mpt->m_events[i].Type = event;
6807         mpt->m_events[i].Number = ++mpt->m_event_number;
6808         bzero(mpt->m_events[i].Data, MPTSAS_MAX_EVENT_DATA_LENGTH * 4);
6809         event_data_len = ddi_get16(mpt->m_acc_reply_frame_hdl,
6810             &eventreply->EventDataLength);

6811         if (event_data_len > 0) {
6812             /*
6813              * Limit data to size in m_event entry
6814              */
6815             if (event_data_len > MPTSAS_MAX_EVENT_DATA_LENGTH) {
6816                 event_data_len = MPTSAS_MAX_EVENT_DATA_LENGTH;
6817             }
6818             for (j = 0; j < event_data_len; j++) {
6819                 mpt->m_events[i].Data[j] =
6820                     ddi_get32(mpt->m_acc_reply_frame_hdl,
6821                         &(eventreply->EventData[j]));
6822             }
6823
6824             /*
6825              * check for index wrap-around
6826              */
6827             if (++i == MPTSAS_EVENT_QUEUE_SIZE) {
6828                 i = 0;
6829             }
6830             mpt->m_event_index = (uint8_t)i;
6831
6832             /*
6833              * Set flag to send the event.
6834              */
6835             sendAEN = TRUE;
6836         }
6837     }
6838
6839     }
6840 }

6841 /*
6842  * Generate a system event if flag is set to let anyone who cares know
6843  * that an event has occurred.
6844  */
6845 if (sendAEN) {
6846

```

```

6847             (void) ddi_log_sysevent(mpt->m_dip, DDI_VENDOR_LSI, "MPT_SAS",
6848                           "SAS", NULL, NULL, DDI_NOSLEEP);
6849         }
6850     }

6852 #define SMP_RESET_IN_PROGRESS MPI2_EVENT_SAS_TOPO_LR_SMP_RESET_IN_PROGRESS
6853 /* handle sync events from ioc in interrupt
6854 * return value:
6855 * DDI_SUCCESS: The event is handled by this func
6856 * DDI_FAILURE: Event is not handled
6857 */
6858 static int
6859 mptsas_handle_event_sync(void *args)
6860 {
6861     m_replyh_arg_t          *replyh_arg;
6862     pMpI2EventNotificationReply_t eventreply;
6863     uint32_t                 event, rfm;
6864     mptsas_t                 *mpt;
6865     uint_t                   iocstatus;
6866

6868     replyh_arg = (m_replyh_arg_t *)args;
6869     rfm = replyh_arg->rfm;
6870     mpt = replyh_arg->mpt;

6872     ASSERT(mutex_owned(&mpt->m_mutex));

6874     eventreply = (pMpI2EventNotificationReply_t)
6875         (mpt->m_reply_frame + (rfm -
6876           (mpt->m_reply_frame_dma_addr & 0xfffffffffu)));
6877     event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6879     if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
6880       &eventreply->IOCStatus)) {
6881         if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
6882             mptsas_log(mpt, CE_WARN,
6883                         "mptsas_handle_event_sync: event 0x%x, "
6884                         "IOCStatus=0x%x, "
6885                         "IOCLogInfo=0x%x", event, iocstatus,
6886                         ddi_get32(mpt->m_acc_reply_frame_hdl,
6887                         &eventreply->IOCLogInfo));
6888         } else {
6889             mptsas_log(mpt, CE_WARN,
6890                         "mptsas_handle_event_sync: event 0x%x, "
6891                         "IOCStatus=0x%x, "
6892                         "(IOCLogInfo=0x%x)", event, iocstatus,
6893                         ddi_get32(mpt->m_acc_reply_frame_hdl,
6894                         &eventreply->IOCLogInfo));
6895         }
6896     }

6898     /*
6899      * figure out what kind of event we got and handle accordingly
6900      */
6901     switch (event) {
6902     case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
6903     {
6904         pMpI2EventDataSasTopologyChangeList_t sas_topo_change_list;
6905         uint8_t          num_entries, expstatus, phy;
6906         uint8_t          phystatus, physport, state, i;
6907         uint8_t          start_phy_num, link_rate;
6908         uint16_t         dev_handle, reason_code;
6909         uint16_t         enc_handle, expd_handle;
6910         char             string[80], curr[80], prev[80];
6911         mptsas_topo_change_list_t *topo_head = NULL;
6912         mptsas_topo_change_list_t *topo_tail = NULL;

```

```

6913         mptsas_topo_change_list_t    *topo_node = NULL;
6914         mptsas_target_t            *ptgt;
6915         mptsas_smp_t               *psmp;
6916         uint8_t                    flags = 0, exp_flag;
6917         smhba_info_t              *pSmhba = NULL;

6919     NDBG20(("mptsas_handle_event_sync: SAS topology change"));

6921     sas_topo_change_list = (pMpI2EventDataSasTopologyChangeList_t)
6922         eventreply->EventData;

6924     enc_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6925                           &sas_topo_change_list->EnclosureHandle);
6926     expd_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6927                           &sas_topo_change_list->ExpanderDevHandle);
6928     num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
6929                           &sas_topo_change_list->NumEntries);
6930     start_phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
6931                           &sas_topo_change_list->StartPhyNum);
6932     expstatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
6933                           &sas_topo_change_list->ExpStatus);
6934     physport = ddi_get8(mpt->m_acc_reply_frame_hdl,
6935                           &sas_topo_change_list->PhysicalPort);

6937     string[0] = 0;
6938     if (expd_handle) {
6939         flags = MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED;
6940         switch (expstatus) {
6941             case MPI2_EVENT_SAS_TOPO_ES_ADDED:
6942                 (void) sprintf(string, " added");
6943                 /* New expander device added */
6944                 /* */
6945                 mpt->m_port_chng = 1;
6946                 topo_node = kmem_zalloc(
6947                     sizeof (mptsas_topo_change_list_t),
6948                     KM_SLEEP);
6949                 topo_node->mpt = mpt;
6950                 topo_node->event = MPTSAS_DR_EVENT_RECONFIG_SMP;
6951                 topo_node->un.physport = physport;
6952                 topo_node->devhdl = expd_handle;
6953                 topo_node->flags = flags;
6954                 topo_node->object = NULL;
6955                 if (topo_head == NULL) {
6956                     topo_head = topo_tail = topo_node;
6957                 } else {
6958                     topo_tail->next = topo_node;
6959                     topo_tail = topo_node;
6960                 }
6961             break;
6962         case MPI2_EVENT_SAS_TOPO_ES_NOT_RESPONDING:
6963             (void) sprintf(string, " not responding, "
6964                           "removed");
6965             psmp = rehash_linear_search(mpt->m_smp_targets,
6966                                         mptsas_smp_eval_devhdl, &expd_handle);
6967             if (psmp == NULL)
6968                 break;
6969

6971         topo_node = kmem_zalloc(
6972             sizeof (mptsas_topo_change_list_t),
6973             KM_SLEEP);
6974         topo_node->mpt = mpt;
6975         topo_node->un.phymask =
6976             psmp->m_addr.mta_phymask;
6977         topo_node->event = MPTSAS_DR_EVENT_OFFLINE_SMP;
6978         topo_node->devhdl = expd_handle;

```

```

6979         topo_node->flags = flags;
6980         topo_node->object = NULL;
6981         if (topo_head == NULL) {
6982             topo_head = topo_tail = topo_node;
6983         } else {
6984             topo_tail->next = topo_node;
6985             topo_tail = topo_node;
6986         }
6987         break;
6988     case MPI2_EVENT_SAS_TOPO_ES_RESPONDING:
6989         break;
6990     case MPI2_EVENT_SAS_TOPO_ES_DELAY_NOT_RESPONDING:
6991         (void) sprintf(string, " not responding, "
6992                       "delaying removal");
6993         break;
6994     default:
6995         break;
6996     }
6997 } else {
6998     flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
6999 }

7000 NDBG20(("SAS TOPOLOGY CHANGE for enclosure %x expander %x%s\n",
7001         enc_handle, expd_handle, string));
7002 for (i = 0; i < num_entries; i++) {
7003     phy = i + start_phy_num;
7004     phystatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
7005                           &sas_topo_change_list->PHY[i].PhyStatus);
7006     dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7007                            &sas_topo_change_list->PHY[i].AttachedDevHandle);
7008     reason_code = phystatus & MPI2_EVENT_SAS_TOPO_RC_MASK;
7009     /*
7010      * Filter out processing of Phy Vacant Status unless
7011      * the reason code is "Not Responding". Process all
7012      * other combinations of Phy Status and Reason Codes.
7013     */
7014     if ((phystatus &
7015         MPI2_EVENT_SAS_TOPO_PHYSTATUS_VACANT) &&
7016         (reason_code !=
7017          MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING)) {
7018         continue;
7019     }
7020     curr[0] = 0;
7021     prev[0] = 0;
7022     string[0] = 0;
7023     switch (reason_code) {
7024     case MPI2_EVENT_SAS_TOPO_RC_TARG_ADDED:
7025     {
7026         NDBG20(("mptsas%d phy %d physical_port %d "
7027                 "dev_handle %d added", mpt->m_instance, phy,
7028                 physport, dev_handle));
7029         link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
7030                               &sas_topo_change_list->PHY[i].LinkRate);
7031         state = (link_rate &
7032                   MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
7033                   MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
7034         switch (state) {
7035         case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
7036             (void) sprintf(curr, "is disabled");
7037             break;
7038         case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
7039             (void) sprintf(curr, "is offline, "
7040                           "failed speed negotiation");
7041             break;
7042         case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
7043             (void) sprintf(curr, "SATA OOB "
7044

```

```

7045             "complete");
7046         break;
7047     case SMP_RESET_IN_PROGRESS:
7048         (void) sprintf(curr, "SMP reset in "
7049                       "progress");
7050         break;
7051     case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
7052         (void) sprintf(curr, "is online at "
7053                       "1.5 Gbps");
7054         break;
7055     case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
7056         (void) sprintf(curr, "is online at 3.0 "
7057                       "Gbps");
7058         break;
7059     case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
7060         (void) sprintf(curr, "is online at 6.0 "
7061                       "Gbps");
7062         break;
7063     case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
7064         (void) sprintf(curr,
7065                         "is online at 12.0 Gbps");
7066         break;
7067     default:
7068         (void) sprintf(curr, "state is "
7069                       "unknown");
7070         break;
7071     }
7072     /*
7073      * New target device added into the system.
7074      * Set association flag according to if an
7075      * expander is used or not.
7076     */
7077     exp_flag =
7078         MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
7079     if (flags ==
7080         MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
7081         flags = exp_flag;
7082     }
7083     topo_node = kmalloc(sizeof(mptsas_topo_change_list_t),
7084                          KM_SLEEP);
7085     topo_node->mpt = mpt;
7086     topo_node->event =
7087         MPTSAS_DR_EVENT_RECONFIG_TARGET;
7088     if (expd_handle == 0) {
7089         /*
7090          * Per MPI 2, if expander dev handle
7091          * is 0, it's a directly attached
7092          * device. So driver use PHY to decide
7093          * which iport is associated
7094          */
7095         physport = phy;
7096         mpt->m_port_chng = 1;
7097     }
7098     topo_node->un.physport = physport;
7099     topo_node->devhdl = dev_handle;
7100     topo_node->flags = flags;
7101     topo_node->object = NULL;
7102     if (topo_head == NULL) {
7103         topo_head = topo_tail = topo_node;
7104     } else {
7105         topo_tail->next = topo_node;
7106         topo_tail = topo_node;
7107     }
7108     break;
7109 }
7110

```

```

7111
7112     case MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING:
7113     {
7114         NDBG20(("mptsas%d phy %d physical_port %d "
7115                 "dev_handle %d removed", mpt->m_instance,
7116                 phy, physport, dev_handle));
7117         /*
7118          * Set association flag according to if an
7119          * expander is used or not.
7120          */
7121         exp_flag =
7122             MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
7123         if (flags ==
7124             MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
7125             flags = exp_flag;
7126         }
7127         /*
7128          * Target device is removed from the system
7129          * Before the device is really offline from
7130          * from system.
7131          */
7132         ptgt = rehash_linear_search(mpt->m_targets,
7133                                     mptsas_target_eval_devhdl, &dev_handle);
7134         /*
7135          * If ptgt is NULL here, it means that the
7136          * DevHandle is not in the hash table. This is
7137          * reasonable sometimes. For example, if a
7138          * disk was pulled, then added, then pulled
7139          * again, the disk will not have been put into
7140          * the hash table because the add event will
7141          * have an invalid phymask. BUT, this does not
7142          * mean that the DevHandle is invalid. The
7143          * controller will still have a valid DevHandle
7144          * that must be removed. To do this, use the
7145          * MPTSAS_TOPO_FLAG_REMOVE_HANDLE event.
7146          */
7147         if (ptgt == NULL) {
7148             topo_node = kmem_zalloc(
7149                 sizeof(mptsas_topo_change_list_t),
7150                 KM_SLEEP);
7151             topo_node->mpt = mpt;
7152             topo_node->un.phymask = 0;
7153             topo_node->event =
7154                 MPTSAS_TOPO_FLAG_REMOVE_HANDLE;
7155             topo_node->devhdl = dev_handle;
7156             topo_node->flags = flags;
7157             topo_node->object = NULL;
7158             if (topo_head == NULL) {
7159                 topo_head = topo_tail =
7160                     topo_node;
7161             } else {
7162                 topo_tail->next = topo_node;
7163                 topo_tail = topo_node;
7164             }
7165             break;
7166         }
7167         /*
7168          * Update DR flag immediately avoid I/O failure
7169          * before failover finish. Pay attention to the
7170          * mutex protect, we need grab m_tx_waitq_mutex
7171          * during set m_dr_flag because we won't add
7172          * the following command into waitq, instead,
7173          * we need return TRAN_BUSY in the tran_start
7174          * context.
7175          */
7176         mutex_enter(&mpt->m_tx_waitq_mutex);

```

```

7177
7178
7179
7180
7181
7182
7183
7184
7185
7186
7187
7188
7189
7190
7191
7192
7193
7194
7195
7196
7197
7198
7199
7200
7201
7202
7203
7204
7205
7206
7207
7208
7209
7210
7211
7212
7213
7214
7215
7216
7217
7218
7219
7220
7221
7222
7223
7224
7225
7226
7227
7228
7229
7230
7231
7232
7233
7234
7235
7236
7237
7238
7239
7240
7241
7242

    ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
    mutex_exit(&mpt->m_tx_waitq_mutex);

    topo_node = kmem_zalloc(
        sizeof(mptsas_topo_change_list_t),
        KM_SLEEP);
    topo_node->mpt = mpt;
    topo_node->un.phymask =
        ptgt->m_addr.mta_phymask;
    topo_node->event =
        MPTSAS_DR_EVENT_OFFLINE_TARGET;
    topo_node->devhdl = dev_handle;
    topo_node->flags = flags;
    topo_node->object = NULL;
    if (topo_head == NULL) {
        topo_head = topo_tail = topo_node;
    } else {
        topo_tail->next = topo_node;
        topo_tail = topo_node;
    }
    break;
}

case MPI2_EVENT_SAS_TOPO_RC_PHY_CHANGED:
    link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
        &sas_topo_change_list->PHY[i].LinkRate);
    state = (link_rate &
        MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
        MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
    pSmhba = &mpt->m_phy_info[i].smhba_info;
    pSmhba->negotiated_link_rate = state;
    switch (state) {
        case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
            (void) sprintf(curr, "is disabled");
            mptsas_smhba_log_sysevent(mpt,
                ESC_SAS_PHY_EVENT,
                SAS_PHY_REMOVE,
                &mpt->m_phy_info[i].smhba_info);
            mpt->m_phy_info[i].smhba_info =
                ox1;
            break;
        case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
            (void) sprintf(curr, "is offline, "
                "failed speed negotiation");
            mptsas_smhba_log_sysevent(mpt,
                ESC_SAS_PHY_EVENT,
                SAS_PHY_OFFLINE,
                &mpt->m_phy_info[i].smhba_info);
            break;
        case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
            (void) sprintf(curr, "SATA OOB "
                "complete");
            break;
        case SMP_RESET_IN_PROGRESS:
            (void) sprintf(curr, "SMP reset in "
                "progress");
            break;
        case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
            (void) sprintf(curr, "is online at "
                "1.5 Gbps");
            if ((expd_handle == 0) &&
                (enc_handle == 1)) {
                mpt->m_port_chng = 1;
            }
            mptsas_smhba_log_sysevent(mpt,
                ESC_SAS_PHY_EVENT,

```

```

7243             SAS_PHY_ONLINE,
7244             &mpt->m_phy_info[i].smhba_info);
7245             break;
7246     case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
7247         (void) sprintf(curr, "is online at 3.0 "
7248                     "Gbps");
7249         if ((expd_handle == 0) &&
7250             (enc_handle == 1)) {
7251             mpt->m_port_chng = 1;
7252         }
7253         mptsas_smhba_log_sysevent(mpt,
7254             ESC_SAS_PHY_EVENT,
7255             SAS_PHY_ONLINE,
7256             &mpt->m_phy_info[i].smhba_info);
7257             break;
7258     case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
7259         (void) sprintf(curr, "is online at "
7260                     "6.0 Gbps");
7261         if ((expd_handle == 0) &&
7262             (enc_handle == 1)) {
7263             mpt->m_port_chng = 1;
7264         }
7265         mptsas_smhba_log_sysevent(mpt,
7266             ESC_SAS_PHY_EVENT,
7267             SAS_PHY_ONLINE,
7268             &mpt->m_phy_info[i].smhba_info);
7269             break;
7270     case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
7271         (void) sprintf(curr, "is online at "
7272                     "12.0 Gbps");
7273         if ((expd_handle == 0) &&
7274             (enc_handle == 1)) {
7275             mpt->m_port_chng = 1;
7276         }
7277         mptsas_smhba_log_sysevent(mpt,
7278             ESC_SAS_PHY_EVENT,
7279             SAS_PHY_ONLINE,
7280             &mpt->m_phy_info[i].smhba_info);
7281             break;
7282     default:
7283         (void) sprintf(curr, "state is "
7284                     "unknown");
7285             break;
7286     }

    state = (link_rate &
7288     MPI2_EVENT_SAS_TOPO_LR_PREV_MASK) >>
7289     MPI2_EVENT_SAS_TOPO_LR_PREV_SHIFT;
7290 switch (state) {
7291     case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
7292         (void) sprintf(prev, ", was disabled");
7293         break;
7294     case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
7295         (void) sprintf(prev, ", was offline, "
7296                     "failed speed negotiation");
7297         break;
7298     case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
7299         (void) sprintf(prev, ", was SATA OOB "
7300                     "complete");
7301         break;
7302     case SMP_RESET_IN_PROGRESS:
7303         (void) sprintf(prev, ", was SMP reset "
7304                     "in progress");
7305         break;
7306     case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
7307         (void) sprintf(prev, ", was online at "
7308

```

```

7309             "1.5 Gbps");
7310             break;
7311     case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
7312         (void) sprintf(prev, ", was online at "
7313                     "3.0 Gbps");
7314             break;
7315     case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
7316         (void) sprintf(prev, ", was online at "
7317                     "6.0 Gbps");
7318             break;
7319     case MPI25_EVENT_SAS_TOPO_LR_RATE_12_0:
7320         (void) sprintf(prev, ", was online at "
7321                     "12.0 Gbps");
7322             break;
7323     default:
7324         break;
7325     }
7326     (void) sprintf(&string[strlen(string)], "link "
7327                     "changed, ");
7328             break;
7329     case MPI2_EVENT_SAS_TOPO_RC_NO_CHANGE:
7330         continue;
7331     case MPI2_EVENT_SAS_TOPO_RC_DELAY_NOT_RESPONDING:
7332         (void) sprintf(&string[strlen(string)], "
7333                     "target not responding, delaying "
7334                     "removal");
7335             break;
7336     }
7337     NDBG20(("mptsas%d phy %d DevHandle %x, %s%s%\n",
7338             mpt->m_instance, phy, dev_handle, string, curr,
7339             prev));
7340 if (topo_head != NULL) {
7341     /*
7342      * Launch DR taskq to handle topology change
7343      */
7344     if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
7345             mptsas_handle_dr, (void *)topo_head,
7346             DDI_NOSLEEP)) != DDI_SUCCESS) {
7347         while (topo_head != NULL) {
7348             topo_node = topo_head;
7349             topo_head = topo_head->next;
7350             kmem_free(topo_node,
7351                         sizeof (mptsas_topo_change_list_t));
7352         }
7353         mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
7354                     "for handle SAS DR event failed. \n");
7355     }
7356 }
7357 break;
7358 case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7359 {
7360     Mpi2EventDataIrConfigChangeList_t          *irChangeList;
7361     mptsas_topo_change_list_t                 *topo_head = NULL;
7362     mptsas_topo_change_list_t                 *topo_tail = NULL;
7363     mptsas_topo_change_list_t                 *topo_node = NULL;
7364     mptsas_target_t                          *ptgt;
7365     uint8_t                                num_entries, i, reason;
7366     uint16_t                               volhandle, diskhandle;
7367
7368     irChangeList = (pMpi2EventDataIrConfigChangeList_t)
7369                     eventreply->EventData;
7370     num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
7371                           &irChangeList->NumElements);
7372
7373

```

```

7375     NDBG20(("mptsas%d IR_CONFIGURATION_CHANGE_LIST event received",
7376             mpt->m_instance));
7377
7378     for (i = 0; i < num_entries; i++) {
7379         reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
7380                           &irChangeList->ConfigElement[i].ReasonCode);
7381         volhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7382                               &irChangeList->ConfigElement[i].VolDevHandle);
7383         diskhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7384                               &irChangeList->ConfigElement[i].PhysDiskDevHandle);
7385
7386         switch (reason) {
7387         case MPI2_EVENT_IR_CHANGE_RC_ADDED:
7388         case MPI2_EVENT_IR_CHANGE_RC_VOLUME_CREATED:
7389         {
7390             NDBG20(("mptsas %d volume added\n",
7391                     mpt->m_instance));
7392
7393             topo_node = kmem_zalloc(
7394                 sizeof (mptsas_topo_change_list_t),
7395                 KM_SLEEP);
7396
7397             topo_node->mpt = mpt;
7398             topo_node->event =
7399                 MPTSAS_DR_EVENT_RECONFIG_TARGET;
7400             topo_node->un.physport = 0xff;
7401             topo_node->devhdl = volhandle;
7402             topo_node->flags =
7403                 MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
7404             topo_node->object = NULL;
7405             if (topo_head == NULL) {
7406                 topo_head = topo_tail = topo_node;
7407             } else {
7408                 topo_tail->next = topo_node;
7409                 topo_tail = topo_node;
7410             }
7411             break;
7412         }
7413         case MPI2_EVENT_IR_CHANGE_RC_REMOVED:
7414         case MPI2_EVENT_IR_CHANGE_RC_VOLUME_DELETED:
7415         {
7416             NDBG20(("mptsas %d volume deleted\n",
7417                     mpt->m_instance));
7418             ptgt = rehash_linear_search(mpt->m_targets,
7419                                         mptsas_target_eval_devhdl, &volhandle);
7420             if (ptgt == NULL)
7421                 break;
7422
7423             /*
7424             * Clear any flags related to volume
7425             */
7426             (void) mptsas_delete_volume(mpt, volhandle);
7427
7428             /*
7429             * Update DR flag immediately avoid I/O failure
7430             */
7431             mutex_enter(&mpt->m_tx_waitq_mutex);
7432             ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
7433             mutex_exit(&mpt->m_tx_waitq_mutex);
7434
7435             topo_node = kmem_zalloc(
7436                 sizeof (mptsas_topo_change_list_t),
7437                 KM_SLEEP);
7438             topo_node->mpt = mpt;
7439             topo_node->un.phymask =
7440                 ptgt->m_addr.mta_phymask;

```

```

7441             topo_node->event =
7442                 MPTSAS_DR_EVENT_OFFLINE_TARGET;
7443             topo_node->devhdl = volhandle;
7444             topo_node->flags =
7445                 MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
7446             topo_node->object = (void *)ptgt;
7447             if (topo_head == NULL) {
7448                 topo_head = topo_tail = topo_node;
7449             } else {
7450                 topo_tail->next = topo_node;
7451                 topo_tail = topo_node;
7452             }
7453             break;
7454         }
7455         case MPI2_EVENT_IR_CHANGE_RC_PD_CREATED:
7456         case MPI2_EVENT_IR_CHANGE_RC_HIDE:
7457         {
7458             ptgt = rehash_linear_search(mpt->m_targets,
7459                                         mptsas_target_eval_devhdl, &diskhandle);
7460             if (ptgt == NULL)
7461                 break;
7462
7463             /*
7464             * Update DR flag immediately avoid I/O failure
7465             */
7466             mutex_enter(&mpt->m_tx_waitq_mutex);
7467             ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
7468             mutex_exit(&mpt->m_tx_waitq_mutex);
7469
7470             topo_node = kmem_zalloc(
7471                 sizeof (mptsas_topo_change_list_t),
7472                 KM_SLEEP);
7473             topo_node->mpt = mpt;
7474             topo_node->un.phymask =
7475                 ptgt->m_addr.mta_phymask;
7476             topo_node->event =
7477                 MPTSAS_DR_EVENT_OFFLINE_TARGET;
7478             topo_node->devhdl = diskhandle;
7479             topo_node->flags =
7480                 MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
7481             topo_node->object = (void *)ptgt;
7482             if (topo_head == NULL) {
7483                 topo_head = topo_tail = topo_node;
7484             } else {
7485                 topo_tail->next = topo_node;
7486                 topo_tail = topo_node;
7487             }
7488             break;
7489         }
7490         case MPI2_EVENT_IR_CHANGE_RC_UNHIDE:
7491         case MPI2_EVENT_IR_CHANGE_RC_PD_DELETED:
7492         {
7493             /*
7494             * The physical drive is released by a IR
7495             * volume. But we cannot get the the physport
7496             * or phnum from the event data, so we only
7497             * can get the physport/phnum after SAS
7498             * Device Page0' request for the devhdl.
7499             */
7500             topo_node = kmem_zalloc(
7501                 sizeof (mptsas_topo_change_list_t),
7502                 KM_SLEEP);
7503             topo_node->mpt = mpt;
7504             topo_node->un.phymask = 0;
7505             topo_node->event =
7506                 MPTSAS_DR_EVENT_RECONFIG_TARGET;

```

```

7507     topo_node->devhdl = diskhandle;
7508     topo_node->flags =
7509         MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
7510     topo_node->object = NULL;
7511     mpt->m_port_chng = 1;
7512     if (topo_head == NULL) {
7513         topo_head = topo_tail = topo_node;
7514     } else {
7515         topo_tail->next = topo_node;
7516         topo_tail = topo_node;
7517     }
7518     break;
7519 }
7520 default:
7521     break;
7522 }
7523 }

7524 if (topo_head != NULL) {
7525     /*
7526      * Launch DR taskq to handle topology change
7527      */
7528     if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
7529                             mptsas_handle_dr, (void *)topo_head,
7530                             DDI_NOSLEEP) != DDI_SUCCESS) {
7531         while (topo_head != NULL) {
7532             topo_node = topo_head;
7533             topo_head = topo_head->next;
7534             kmem_free(topo_node,
7535                         sizeof (mptsas_topo_change_list_t));
7536         }
7537         mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
7538                     "for handle SAS DR event failed. \n");
7539     }
7540     }
7541     break;
7542 }
7543 default:
7544     return (DDI_FAILURE);
7545 }
7546 }

7547 return (DDI_SUCCESS);
7548 }

7549 */

7550 /* handle events from ioc
7551 */
7552 static void
7553 mptsas_handle_event(void *args)
7554 {
7555     m_replyh_arg_t          *replyh_arg;
7556     pMpI2EventNotificationReply_t eventreply;
7557     uint32_t                 event, iocloginfo, rfm;
7558     uint32_t                 status;
7559     uint8_t                  port;
7560     uint8_t                  *mpt;
7561     mptsas_t                 *mpt;
7562     uint_t                   iocstatus;
7563

7564     replyh_arg = (m_replyh_arg_t *)args;
7565     rfm = replyh_arg->rfm;
7566     mpt = replyh_arg->mpt;
7567

7568     mutex_enter(&mpt->m_mutex);
7569     /*
7570      * If HBA is being reset, drop incoming event.
7571      */
7572

```

```

7573     if (mpt->m_in_reset) {
7574         NDBG20(("dropping event received prior to reset"));
7575         mutex_exit(&mpt->m_mutex);
7576         return;
7577     }
7578

7579     eventreply = (pMpI2EventNotificationReply_t)
7580         (mpt->m_reply_frame + (rfm -
7581         (mpt->m_reply_frame_dma_addr & 0xffffffffu)));
7582     event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);
7583

7584     if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
7585         &eventreply->IOCStatus)) {
7586         if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
7587             mptsas_log(mpt, CE_WARN,
7588                 "!mptsas_handle_event: IOCStatus=0x%x, "
7589                 "IOCLogInfo=0x%x", iocstatus,
7590                 ddi_get32(mpt->m_acc_reply_frame_hdl,
7591                         &eventreply->IOCLogInfo));
7592         } else {
7593             mptsas_log(mpt, CE_WARN,
7594                 "mptsas_handle_event: IOCStatus=0x%x, "
7595                 "IOCLogInfo=0x%x", iocstatus,
7596                 ddi_get32(mpt->m_acc_reply_frame_hdl,
7597                         &eventreply->IOCLogInfo));
7598         }
7599     }

7600     /*
7601      * figure out what kind of event we got and handle accordingly
7602      */
7603     switch (event) {
7604         case MPI2_EVENT_LOG_ENTRY_ADDED:
7605             break;
7606         case MPI2_EVENT_LOG_DATA:
7607             iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7608                 &eventreply->IOCLogInfo);
7609             NDBG20(("mptsas%d log info %x received.\n", mpt->m_instance,
7610                     iocloginfo));
7611             break;
7612         case MPI2_EVENT_STATE_CHANGE:
7613             NDBG20(("mptsas%d state change.", mpt->m_instance));
7614             break;
7615         case MPI2_EVENT_HARD_RESET_RECEIVED:
7616             NDBG20(("mptsas%d event change.", mpt->m_instance));
7617             break;
7618         case MPI2_EVENT_SAS_DISCOVERY:
7619             {
7620                 MPI2_EVENT_DATA_SAS_DISCOVERY    *sasdiscovery;
7621                 char                           string[80];
7622                 uint8_t                         rc;
7623
7624                 sasdiscovery =
7625                     (pMpI2EventDataSasDiscovery_t)eventreply->EventData;
7626
7627                 rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7628                     &sasdiscovery->ReasonCode);
7629                 port = ddi_get8(mpt->m_acc_reply_frame_hdl,
7630                     &sasdiscovery->PhysicalPort);
7631                 status = ddi_get32(mpt->m_acc_reply_frame_hdl,
7632                     &sasdiscovery->DiscoveryStatus);
7633
7634                 string[0] = 0;
7635                 switch (rc) {
7636                     case MPI2_EVENT_SAS_DISC_RC_STARTED:
7637                         (void) sprintf(string, "STARTING");
7638

```

```

7639         break;
7640     case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
7641         (void) sprintf(string, "COMPLETED");
7642         break;
7643     default:
7644         (void) sprintf(string, "UNKNOWN");
7645         break;
7646     }
7647
7648     NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
7649             port, status));
7650
7651     break;
7652 }
7653 case MPI2_EVENT_EVENT_CHANGE:
7654     NDBG20(("mptsas%d event change.", mpt->m_instance));
7655     break;
7656 case MPI2_EVENT_TASK_SET_FULL:
7657 {
7658     pMpI2EventDataTaskSetFull_t taskfull;
7659
7660     taskfull = (pMpI2EventDataTaskSetFull_t)eventreply->EventData;
7661
7662     NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
7663             mpt->m_instance, ddi_get16(mpt->m_acc_reply_frame_hdl,
7664             &taskfull->CurrentDepth)));
7665     break;
7666 }
7667 case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
7668 {
7669     /*
7670      * SAS TOPOLOGY CHANGE LIST Event has already been handled
7671      * in mptsas_handle_event_sync() of interrupt context
7672      */
7673     break;
7674 }
7675 case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
7676 {
7677     pMpI2EventDataSasEnclDevStatusChange_t encstatus;
7678     uint8_t rc;
7679     char string[80];
7680
7681     encstatus = (pMpI2EventDataSasEnclDevStatusChange_t)
7682         eventreply->EventData;
7683
7684     rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7685             &encstatus->ReasonCode);
7686     switch (rc) {
7687     case MPI2_EVENT_SAS_ENCL_RC_ADDED:
7688         (void) sprintf(string, "added");
7689         break;
7690     case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
7691         (void) sprintf(string, ", not responding");
7692         break;
7693     default:
7694         break;
7695     }
7696     NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure "
7697             "%x%s\n", mpt->m_instance,
7698             ddi_get16(mpt->m_acc_reply_frame_hdl,
7699             &encstatus->EnclosureHandle), string));
7700
7701 }
7702 */
7703 /* MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by

```

```

7705     * mptsas_handle_event_sync,in here just send ack message.
7706     */
7707     case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
7708     {
7709         pMpI2EventDataSasDeviceStatusChange_t statuschange;
7710         uint8_t rc;
7711         devhdl;
7712         uint64_t wwn = 0;
7713         uint32_t wwn_lo, wwn_hi;
7714
7715         statuschange = (pMpI2EventDataSasDeviceStatusChange_t)
7716             eventreply->EventData;
7717         rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7718             &statuschange->ReasonCode);
7719         wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7720             (uint32_t *) (void *) &statuschange->SASAddress);
7721         wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
7722             (uint32_t *) (void *) &statuschange->SASAddress + 1);
7723         wnn = ((uint64_t) wnn_hi << 32) | wnn_lo;
7724         devhdl = ddi_get16(mpt->m_acc_reply_frame_hdl,
7725             &statuschange->DevHandle);
7726
7727         NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wnn is %"PRIx64,
7728                 wnn));
7729
7730         switch (rc) {
7731         case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
7732             NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
7733                     ddi_get8(mpt->m_acc_reply_frame_hdl,
7734                     &statuschange->ASC),
7735                     ddi_get8(mpt->m_acc_reply_frame_hdl,
7736                     &statuschange->ASCQ)));
7737             break;
7738
7739         case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
7740             NDBG20(("Device not supported"));
7741             break;
7742
7743         case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
7744             NDBG20(("IOC internally generated the Target Reset "
7745                     "for devhdl:%x", devhdl));
7746             break;
7747
7748         case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
7749             NDBG20(("IOC's internally generated Target Reset "
7750                     "completed for devhdl:%x", devhdl));
7751             break;
7752
7753         case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
7754             NDBG20(("IOC internally generated Abort Task"));
7755             break;
7756
7757         case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
7758             NDBG20(("IOC's internally generated Abort Task "
7759                     "completed"));
7760             break;
7761
7762         case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
7763             NDBG20(("IOC internally generated Abort Task Set"));
7764             break;
7765
7766         case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
7767             NDBG20(("IOC internally generated Clear Task Set"));
7768             break;
7769
7770         case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
7771             break;
7772     }
7773 }
7774
7775
7776
7777
7778
7779
7780
7781
7782
7783
7784
7785
7786
7787
7788
7789
7790
7791
7792
7793
7794
7795
7796
7797
7798
7799
7800
7801
7802
7803
7804
7805
7806
7807
7808
7809
7810
7811
7812
7813
7814
7815
7816
7817
7818
7819
7820
7821
7822
7823
7824
7825
7826
7827
7828
7829
7830
7831
7832
7833
7834
7835
7836
7837
7838
7839
7840
7841
7842
7843
7844
7845
7846
7847
7848
7849
7850
7851
7852
7853
7854
7855
7856
7857
7858
7859
7860
7861
7862
7863
7864
7865
7866
7867
7868
7869
7870
7871
7872
7873
7874
7875
7876
7877
7878
7879
7880
7881
7882
7883
7884
7885
7886
7887
7888
7889
7890
7891
7892
7893
7894
7895
7896
7897
7898
7899
7900
7901
7902
7903
7904
7905
7906
7907
7908
7909
7910
7911
7912
7913
7914
7915
7916
7917
7918
7919
7920
7921
7922
7923
7924
7925
7926
7927
7928
7929
7930
7931
7932
7933
7934
7935
7936
7937
7938
7939
7940
7941
7942
7943
7944
7945
7946
7947
7948
7949
7950
7951
7952
7953
7954
7955
7956
7957
7958
7959
7960
7961
7962
7963
7964
7965
7966
7967
7968
7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
8000
8001
8002
8003
8004
8005
8006
8007
8008
8009
8010
8011
8012
8013
8014
8015
8016
8017
8018
8019
8020
8021
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
8035
8036
8037
8038
8039
8040
8041
8042
8043
8044
8045
8046
8047
8048
8049
8050
8051
8052
8053
8054
8055
8056
8057
8058
8059
8060
8061
8062
8063
8064
8065
8066
8067
8068
8069
8070
8071
8072
8073
8074
8075
8076
8077
8078
8079
8080
8081
8082
8083
8084
8085
8086
8087
8088
8089
8090
8091
8092
8093
8094
8095
8096
8097
8098
8099
8100
8101
8102
8103
8104
8105
8106
8107
8108
8109
8110
8111
8112
8113
8114
8115
8116
8117
8118
8119
8120
8121
8122
8123
8124
8125
8126
8127
8128
8129
8130
8131
8132
8133
8134
8135
8136
8137
8138
8139
8140
8141
8142
8143
8144
8145
8146
8147
8148
8149
8150
8151
8152
8153
8154
8155
8156
8157
8158
8159
8160
8161
8162
8163
8164
8165
8166
8167
8168
8169
8170
8171
8172
8173
8174
8175
8176
8177
8178
8179
8180
8181
8182
8183
8184
8185
8186
8187
8188
8189
8190
8191
8192
8193
8194
8195
8196
8197
8198
8199
8200
8201
8202
8203
8204
8205
8206
8207
8208
8209
8210
8211
8212
8213
8214
8215
8216
8217
8218
8219
8220
8221
8222
8223
8224
8225
8226
8227
8228
8229
8230
8231
8232
8233
8234
8235
8236
8237
8238
8239
8240
8241
8242
8243
8244
8245
8246
8247
8248
8249
8250
8251
8252
8253
8254
8255
8256
8257
8258
8259
8260
8261
8262
8263
8264
8265
8266
8267
8268
8269
8270
8271
8272
8273
8274
8275
8276
8277
8278
8279
8280
8281
8282
8283
8284
8285
8286
8287
8288
8289
8290
8291
8292
8293
8294
8295
8296
8297
8298
8299
8300
8301
8302
8303
8304
8305
8306
8307
8308
8309
8310
8311
8312
8313
8314
8315
8316
8317
8318
8319
8320
8321
8322
8323
8324
8325
8326
8327
8328
8329
8330
8331
8332
8333
8334
8335
8336
8337
8338
8339
8340
8341
8342
8343
8344
8345
8346
8347
8348
8349
8350
8351
8352
8353
8354
8355
8356
8357
8358
8359
8360
8361
8362
8363
8364
8365
8366
8367
8368
8369
8370
8371
8372
8373
8374
8375
8376
8377
8378
8379
8380
8381
8382
8383
8384
8385
8386
8387
8388
8389
8390
8391
8392
8393
8394
8395
8396
8397
8398
8399
8400
8401
8402
8403
8404
8405
8406
8407
8408
8409
8410
8411
8412
8413
8414
8415
8416
8417
8418
8419
8420
8421
8422
8423
8424
8425
8426
8427
8428
8429
8430
8431
8432
8433
8434
8435
8436
8437
8438
8439
8440
8441
8442
8443
8444
8445
8446
8447
8448
8449
8450
8451
8452
8453
8454
8455
8456
8457
8458
8459
8460
8461
8462
8463
8464
8465
8466
8467
8468
8469
8470
8471
8472
8473
8474
8475
8476
8477
8478
8479
8480
8481
8482
8483
8484
8485
8486
8487
8488
8489
8490
8491
8492
8493
8494
8495
8496
8497
8498
8499
8500
8501
8502
8503
8504
8505
8506
8507
8508
8509
8510
8511
8512
8513
8514
8515
8516
8517
8518
8519
8520
8521
8522
8523
8524
8525
8526
8527
8528
8529
8530
8531
8532
8533
8534
8535
8536
8537
8538
8539
8540
8541
8542
8543
8544
8545
8546
8547
8548
8549
8550
8551
8552
8553
8554
8555
8556
8557
8558
8559
8560
8561
8562
8563
8564
8565
8566
8567
8568
8569
8570
8571
8572
8573
8574
8575
8576
8577
8578
8579
8580
8581
8582
8583
8584
8585
8586
8587
8588
8589
8590
8591
8592
8593
8594
8595
8596
8597
8598
8599
8600
8601
8602
8603
8604
8605
8606
8607
8608
8609
8610
8611
8612
8613
8614
8615
8616
8617
8618
8619
8620
8621
8622
8623
8624
8625
8626
8627
8628
8629
8630
8631
8632
8633
8634
8635
8636
8637
8638
8639
8640
8641
8642
8643
8644
8645
8646
8647
8648
8649
8650
8651
8652
8653
8654
8655
8656
8657
8658
8659
8660
8661
8662
8663
8664
8665
8666
8667
8668
8669
8670
8671
8672
8673
8674
8675
8676
8677
8678
8679
8680
8681
8682
8683
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693
8694
8695
8696
8697
8698
8699
8700
8701
8702
8703
8704
8705
8706
8707
8708
8709
8710
8711
8712
8713
8714
8715
8716
8717
8718
8719
8720
8721
8722
8723
8724
8725
8726
8727
8728
8729
8730
8731
8732
8733
8734
8735
8736
8737
8738
8739
8740
8741
8742
8743
8744
8745
8746
8747
8748
8749
8750
8751
8752
8753
8754
8755
8756
8757
8758
8759
8760
8761
8762
8763
8764
8765
8766
8767
8768
8769
8770
8771
8772
8773
8774
8775
8776
8777
8778
8779
8780
8781
8782
8783
8784
8785
8786
8787
8788
8789
8790
8791
8792
8793
8794
8795
8796
8797
8798
8799
8800
8801
8802
8803
8804
8805
8806
8807
8808
8809
8810
8811
8812
8813
8814
8815
8816
8817
8818
8819
8820
8821
8822
8823
8824
8825
8826
8827
8828
8829
8830
8831
8832
8833
8834
8835
8836
8837
8838
8839
8840
8841
8842
8843
8844
8845
8846
8847
8848
8849
8850
8851
8852
8853
8854
8855
8856
8857
8858
8859
8860
8861
8862
8863
8864
8865
8866
8867
8868
8869
8870
8871
8872
8873
8874
8875
8876
8877
8878
8879
8880
8881
8882
8883
8884
8885
8886
8887
8888
8889
8890
8891
8892
8893
8894
8895
8896
8897
8898
8899
8900
8901
8902
8903
8904
8905
8906
8907
8908
8909
8910
8911
8912
8913
8914
8915
8916
8917
8918
8919
8920
8921
8922
8923
8924
8925
8926
8927
8928
8929
8930
8931
8932
8933
8934
8935
8936
8937
8938
8939
8940
8941
8942
8943
8944
8945
8946
8947
8948
8949
8950
8951
8952
8953
8954
8955
8956
8957
8958
8959
8960
8961
8962
8963
8964
8965
8966
8967
8968
8969
8970
8971
8972
8973
8974
8975
8976
8977
8978
8979
8980
8981
8982
8983
8984
8985
8986
8987
8988
8989
8990
8991
8992
8993
8994
8995
8996
8997
8998
8999
9000
9001
9002
9003
9004
9005
9006
9007
9008
9009
90010
90011
90012
90013
90014
90015
90016
90017
90018
90019
90020
90021
90022
90023
90024
90025
90026
90027
90028
90029
90030
90031
90032
90033
90034
90035
90036
90037
90038
90039
90040
90041
90042
90043
90044
90045
90046
90047
90048
90049
90050
90051
90052
90053
90054
90055
90056
90057
90058
90059
90060
90061
90062
90063
90064
90065
90066
90067
90068
90069
90070
90071
90072
90073
90074
90075
90076
90077
90078
90079
90080
90081
90082
90083
90084
90085
90086
90087
90088
90089
90090
90091
90092
90093
90094
90095
90096
90097
90098
90099
900100
900101
900102
900103
900104
900105
900106
900107
900108
900109
900110
900111
900112
900113
900114
900115
900116
900117
900118
900119
900120
900121
900122
900123
900124
900125
900126
900127
900128
900129
900130
900131
900132
900133
900134
900135
900136
900137
900138
900139
900140
900141
900142
900143
900144
900145
900146
900147
900148
900149
900150
900151
900152
900153
900154
900155
900156
900157
900158
900159
900160
900161
900162
900163
900164
900165
900166
900167
900168
900169
900170
900171
900172
900173
900174
900175
900176
900177
900178
900179
900180
900181
900182
900183
900184
900185
900186
900187
900188
900189
900190
900191
900192
900193
900194
900195
900196
900197
900198
900199
900200
900201
900202
900203
900204
900205
900206
900207
900208
900209
900210
900211
900212
900213
900214
900215
900216
900217
900218
900219
900220
900221
900222
900223
900224
900225
900226
900227
900228
900229
900230
900231
900232
900233
900234
900235
900236
900237
900238
900239
900240
900241
900242
900243
900244
900245
900246
900247
900248
900249
900250
900251
900252
900253
900254
900255
900256
900257
900258
900259
900260
900261
900262
900263
900264
900265
900266
900267
900268
900269
900270
900271
900272
900273
900274
900275
900276
900277
900278
900279
900280
900281
900282
900283
900284
900285
900286
900287
900288
900289
900290
900291
900292
900293
900294
900295
900296
900297
900298
900299
900300
900301
900302
900303
900304
900305
900306
900307
900308
900309
900310
900311
900312
900313
900314
900315
900316
900317
900318
900319
900320
900321
900322
900323
900324
900325
900326
900327
900328
900329
900330
900331
900332
900333
900334
900335
900336
900337
900338
900339
900340
900341
900342
900343
900344
900345
900346
900347
900348
900349
900350
900351
900352
900353
900354
900355
900356
900357
900358
900359
900360
900361
900362
900363
900364
900365
900366
900367
900368
900369
900370
900371
900372
900373
900374
900375
900376
900377
900378
900379
900380
900381
900382
900383
900384
900385
900386
900387
900388
900389
900390
900391
900392
900393
900394
900395
900396
900397
900398
900399
900400
900401
900402
900403
900404
900405
900406
900407
900408
900409
900410
900411
900412
900413
900414
900415
900416
900417
900418
900419
900420
900421
900422
900423
900424
900425
900426
900427
900428
900429
900430
900431
900432
900433
900434
900435
900436
900437
900438
900439
900440
900441
900442
900443
900444
900445
900446
900447
900448
900449
900450
900451
900452
900453
900454
900455
900456
900457
900458
900459
900460
900461
900462
900463
900464
900465
900466
900467
900468
900469
900470
900471
900472
900473
900474
900475
900476
900477
900478
900479
900480
900481
900482
900483
900484
900485
900486
900487
900488
```

```

7771             NDBG20(("IOC internally generated Query Task"));
7772             break;
7773
7774     case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
7775         NDBG20(("Device sent an Asynchronous Notification"));
7776         break;
7777
7778     default:
7779         break;
7780     }
7781     break;
7782 }
7783 case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7784 {
7785     /*
7786      * IR TOPOLOGY CHANGE LIST Event has already been handled
7787      * in mpt_handle_event_sync() of interrupt context
7788      */
7789     break;
7790 }
7791 case MPI2_EVENT_IR_OPERATION_STATUS:
7792 {
7793     Mpi2EventDataIrOperationStatus_t    *irOpStatus;
7794     char                                reason_str[80];
7795     uint8_t                             rc, percent;
7796     uint16_t                            handle;
7797
7798     irOpStatus = (pMpi2EventDataIrOperationStatus_t)
7799         eventreply->EventData;
7800     rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7801                   &irOpStatus->RAIDOperation);
7802     percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
7803                          &irOpStatus->PercentComplete);
7804     handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7805                         &irOpStatus->VolDevHandle);
7806
7807     switch (rc) {
7808         case MPI2_EVENT_IR_RAIDOP_RESYNC:
7809             (void) sprintf(reason_str, "resync");
7810             break;
7811         case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
7812             (void) sprintf(reason_str, "online capacity "
7813                           "expansion");
7814             break;
7815         case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
7816             (void) sprintf(reason_str, "consistency check");
7817             break;
7818         default:
7819             (void) sprintf(reason_str, "unknown reason %x",
7820                           rc);
7821     }
7822
7823     NDBG20(("mptsas%d raid operational status: (%s)"
7824             "\n\thandle(0x%04x), percent complete(%d)\n",
7825             mpt->m_instance, reason_str, handle, percent));
7826     break;
7827 }
7828 case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
7829 {
7830     pMpi2EventDataSasBroadcastPrimitive_t    sas_broadcast;
7831     uint8_t                                phy_num;
7832     uint8_t                                primitive;
7833
7834     sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
7835         eventreply->EventData;

```

```

7837     phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
7838                         &sas_broadcast->PhyNum);
7839     primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
7840                         &sas_broadcast->Primitive);
7841
7842     switch (primitive) {
7843         case MPI2_EVENT_PRIMITIVE_CHANGE:
7844             mptsas_smhba_log_sysevent(mpt,
7845                                         ESC_SAS_HBA_PORT_BROADCAST,
7846                                         SAS_PORT_BROADCAST_CHANGE,
7847                                         &mpt->m_phy_info[phy_num].smhba_info);
7848             break;
7849         case MPI2_EVENT_PRIMITIVE_SES:
7850             mptsas_smhba_log_sysevent(mpt,
7851                                         ESC_SAS_HBA_PORT_BROADCAST,
7852                                         SAS_PORT_BROADCAST_SES,
7853                                         &mpt->m_phy_info[phy_num].smhba_info);
7854             break;
7855         case MPI2_EVENT_PRIMITIVE_EXPANDER:
7856             mptsas_smhba_log_sysevent(mpt,
7857                                         ESC_SAS_HBA_PORT_BROADCAST,
7858                                         SAS_PORT_BROADCAST_D01_4,
7859                                         &mpt->m_phy_info[phy_num].smhba_info);
7860             break;
7861         case MPI2_EVENT_PRIMITIVE_ASYNCHRONOUS_EVENT:
7862             mptsas_smhba_log_sysevent(mpt,
7863                                         ESC_SAS_HBA_PORT_BROADCAST,
7864                                         SAS_PORT_BROADCAST_D04_7,
7865                                         &mpt->m_phy_info[phy_num].smhba_info);
7866             break;
7867         case MPI2_EVENT_PRIMITIVE_RESERVED3:
7868             mptsas_smhba_log_sysevent(mpt,
7869                                         ESC_SAS_HBA_PORT_BROADCAST,
7870                                         SAS_PORT_BROADCAST_D16_7,
7871                                         &mpt->m_phy_info[phy_num].smhba_info);
7872             break;
7873         case MPI2_EVENT_PRIMITIVE_RESERVED4:
7874             mptsas_smhba_log_sysevent(mpt,
7875                                         ESC_SAS_HBA_PORT_BROADCAST,
7876                                         SAS_PORT_BROADCAST_D29_7,
7877                                         &mpt->m_phy_info[phy_num].smhba_info);
7878             break;
7879         case MPI2_EVENT_PRIMITIVE_CHANGE0_RESERVED:
7880             mptsas_smhba_log_sysevent(mpt,
7881                                         ESC_SAS_HBA_PORT_BROADCAST,
7882                                         SAS_PORT_BROADCAST_D24_0,
7883                                         &mpt->m_phy_info[phy_num].smhba_info);
7884             break;
7885         case MPI2_EVENT_PRIMITIVE_CHANGE1_RESERVED:
7886             mptsas_smhba_log_sysevent(mpt,
7887                                         ESC_SAS_HBA_PORT_BROADCAST,
7888                                         SAS_PORT_BROADCAST_D27_4,
7889                                         &mpt->m_phy_info[phy_num].smhba_info);
7890             break;
7891         default:
7892             NDBG16(("mptsas%d: unknown BROADCAST PRIMITIVE"
7893                     " %x received",
7894                     mpt->m_instance, primitive));
7895             break;
7896     }
7897     NDBG16(("mptsas%d sas broadcast primitive: "
7898                     "\n\tprimitive(0x%04x), phy(%d) complete\n",
7899                     mpt->m_instance, primitive, phy_num));
7900     break;
7901 }
7902 case MPI2_EVENT_IR_VOLUME:

```

```

7903     {
7904         Mpi2EventDataIrVolume_t          *irVolume;
7905         uint16_t                         devhandle;
7906         uint32_t                         state;
7907         int                               config, vol;
7908         uint8_t                          found = FALSE;
7909
7910         irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
7911         state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7912                           &irVolume->NewValue);
7913         devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7914                           &irVolume->VolDevHandle);
7915
7916         NDBG20(("EVENT_IR_VOLUME event is received"));
7917
7918         /*
7919          * Get latest RAID info and then find the DevHandle for this
7920          * event in the configuration. If the DevHandle is not found
7921          * just exit the event.
7922         */
7923         (void) mptsas_get_raid_info(mpt);
7924         for (config = 0; (config < mpt->m_num_raid_configs) &&
7925              (!found); config++) {
7926             for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
7927               if (mpt->m_raidconfig[config].m_raidvol[vol].
7928                   m_raidhandle == devhandle) {
7929                 found = TRUE;
7930                 break;
7931               }
7932             }
7933           if (!found) {
7934             break;
7935           }
7936
7937         switch (irVolume->ReasonCode) {
7938         case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
7939         {
7940             uint32_t i;
7941             mpt->m_raidconfig[config].m_raidvol[vol].m_settings =
7942                 state;
7943
7944             i = state & MPI2_RAIDVOL_SETTING_MASK_WRITE_CACHING;
7945             mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
7946                         ", auto-config of hot-swap drives is %s"
7947                         ", write caching is %s"
7948                         ", hot-spare pool mask is %02x\n",
7949                         vol, state &
7950                         MPI2_RAIDVOL_SETTING_AUTO_CONFIG_HSWAP_DISABLE
7951                         ? "disabled" : "enabled",
7952                         i == MPI2_RAIDVOL_SETTING_UNCHANGED
7953                         ? "controlled by member disks" :
7954                         i == MPI2_RAIDVOL_SETTING_DISABLE_WRITE_CACHING
7955                         ? "disabled" :
7956                         i == MPI2_RAIDVOL_SETTING_ENABLE_WRITE_CACHING
7957                         ? "enabled" :
7958                         "incorrectly set",
7959                         (state >> 16) & 0xff);
7960             break;
7961           }
7962         case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
7963         {
7964             mpt->m_raidconfig[config].m_raidvol[vol].m_state =
7965                 (uint8_t)state;
7966
7967             mptsas_log(mpt, CE_NOTE,
7968

```

```

7969
7970
7971
7972
7973
7974
7975
7976
7977
7978
7979
7980
7981
7982
7983
7984
7985
7986
7987
7988
7989
7990
7991
7992
7993
7994
7995
7996
7997
7998
7999
8000
8001
8002
8003
8004
8005
8006
8007
8008
8009
8010
8011
8012
8013
8014
8015
8016
8017
8018
8019
8020
8021
8022
8023
8024
8025
8026
8027
8028
8029
8030
8031
8032
8033
8034
    "Volume %d is now %s\n", vol,
    state == MPI2_RAID_VOL_STATE_OPTIMAL
    ? "optimal" :
    state == MPI2_RAID_VOL_STATE_DEGRADED
    ? "degraded" :
    state == MPI2_RAID_VOL_STATE_ONLINE
    ? "online" :
    state == MPI2_RAID_VOL_STATE_INITIALIZING
    ? "initializing" :
    state == MPI2_RAID_VOL_STATE_FAILED
    ? "failed" :
    state == MPI2_RAID_VOL_STATE_MISSING
    ? "missing" :
    "state unknown");
    break;
}
case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
{
    mpt->m_raidconfig[config].m_raidvol[vol].
        m_statusflags = state;

    mptsas_log(mpt, CE_NOTE,
        " Volume %d is now %s%s%s%s%s%s%s%s\n",
        vol,
        state & MPI2_RAIDVOL_STATUS_FLAG_ENABLED
        ? ", enabled" : ", disabled",
        state & MPI2_RAIDVOL_STATUS_FLAG QUIESCED
        ? ", quiesced" : "",
        state & MPI2_RAIDVOL_STATUS_FLAG_VOLUME_INACTIVE
        ? ", inactive" : ", active",
        state &
        MPI2_RAIDVOL_STATUS_FLAG_BAD_BLOCK_TABLE_FULL
        ? ", bad block table is full" : "",
        state &
        MPI2_RAIDVOL_STATUS_FLAG_RESYNC_IN_PROGRESS
        ? ", resync in progress" : "",
        state & MPI2_RAIDVOL_STATUS_FLAG_BACKGROUND_INIT
        ? ", background initialization in progress" : "",
        state &
        MPI2_RAIDVOL_STATUS_FLAG_CAPACITY_EXPANSION
        ? ", capacity expansion in progress" : "",
        state &
        MPI2_RAIDVOL_STATUS_FLAG_CONSISTENCY_CHECK
        ? ", consistency check in progress" : "",
        state & MPI2_RAIDVOL_STATUS_FLAG_DATA_SCRUB
        ? ", data scrub in progress" : "");
    break;
}
default:
    break;
}
break;
}
case MPI2_EVENT_IR_PHYSICAL_DISK:
{
    Mpi2EventDataIrPhysicalDisk_t *irPhysDisk;
    uint16_t                      devhandle, enhandle, slot;
    uint32_t                      status, state;
    uint8_t                        physdisknum, reason;

    irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
        eventreply->EventData;
    physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,
        &irPhysDisk->PhysDiskNum);
    devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
        &irPhysDisk->PhysDiskDevHandle);
}

```

```

8035     enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
8036         &irPhysDisk->EnclosureHandle);
8037     slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
8038         &irPhysDisk->Slot);
8039     state = ddi_get32(mpt->m_acc_reply_frame_hdl,
8040         &irPhysDisk->NewValue);
8041     reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
8042         &irPhysDisk->ReasonCode);

8044     NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));

8046     switch (reason) {
8047     case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
8048         mptsas_log(mpt, CE_NOTE,
8049             " PhysDiskNum %d with DevHandle 0x%x in slot %d "
8050             "for enclosure with handle 0x%x is now in hot "
8051             "%s%s%s%s\n", physdisknum, devhandle, slot,
8052             (state >> 16) & 0xff);
8053         break;

8056     case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
8057         status = state;
8058         mptsas_log(mpt, CE_NOTE,
8059             " PhysDiskNum %d with DevHandle 0x%x in slot %d "
8060             "for enclosure with handle 0x%x is now "
8061             "%s%s%s%s\n", physdisknum, devhandle, slot,
8062             enchandle,
8063             status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
8064             ? ", inactive" : ", active",
8065             status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
8066             ? ", out of sync" : "",
8067             status & MPI2_PHYSDISK0_STATUS_FLAG QUIESCED
8068             ? ", quiesced" : "",
8069             status &
8070             MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
8071             ? ", write cache enabled" : "",
8072             status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
8073             ? ", capacity expansion target" : "");
8074         break;

8076     case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
8077         mptsas_log(mpt, CE_NOTE,
8078             " PhysDiskNum %d with DevHandle 0x%x in slot %d "
8079             "for enclosure with handle 0x%x is now %s\n",
8080             physdisknum, devhandle, slot, enchandle,
8081             state == MPI2_RAID_PD_STATE_OPTIMAL
8082             ? "optimal" :
8083             state == MPI2_RAID_PD_STATE_REBUILDING
8084             ? "rebuilding" :
8085             state == MPI2_RAID_PD_STATE_DEGRADED
8086             ? "degraded" :
8087             state == MPI2_RAID_PD_STATE_HOT_SPARE
8088             ? "a hot spare" :
8089             state == MPI2_RAID_PD_STATE_ONLINE
8090             ? "online" :
8091             state == MPI2_RAID_PD_STATE_OFFLINE
8092             ? "offline" :
8093             state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
8094             ? "not compatible" :
8095             state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
8096             ? "not configured" :
8097             "state unknown");
8098         break;
8099     }
8100     break;

```

```

8101     }
8102     default:
8103         NDBG20(("mptsas%d: unknown event %x received",
8104             mpt->m_instance, event));
8105         break;
8106     }

8108     /*
8109      * Return the reply frame to the free queue.
8110      */
8111     ddi_put32(mpt->m_acc_free_queue_hdl,
8112         &(uint32_t *)mpt->m_free_queue)[mpt->m_free_index], rfm);
8113     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
8114         DDI_DMA_SYNC_FORDEV);
8115     if (++mpt->m_free_index == mpt->m_free_queue_depth) {
8116         mpt->m_free_index = 0;
8117     }
8118     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
8119         mpt->m_free_index);
8120     mutex_exit(&mpt->m_mutex);
8121 }

8123 /* invoked from timeout() to restart qfull cmd with throttle == 0
8124 */
8125 static void
8126 mptsas_restart_cmd(void *arg)
8127 {
8128     mptsas_t *mpt = arg;
8129     mptsas_target_t *ptgt = NULL;
8130
8131     mutex_enter(&mpt->m_mutex);
8132
8133     mpt->m_restart_cmd_timeid = 0;
8134
8135     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
8136         ptgt = refhash_next(mpt->m_targets, ptgt)) {
8137         if (ptgt->m_reset_delay == 0) {
8138             if (ptgt->m_t_throttle == QFULL_THROTTLE) {
8139                 mptsas_set_throttle(mpt, ptgt,
8140                     MAX_THROTTLE);
8141             }
8142         }
8143     }
8144     mptsas_restart_hba(mpt);
8145     mutex_exit(&mpt->m_mutex);
8146
8147 }

8148 void
8149 mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
8150 {
8151     int slot;
8152     mptsas_slots_t *slots = mpt->m_active;
8153     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8154
8155     ASSERT(cmd != NULL);
8156     ASSERT(cmd->cmd_queued == FALSE);
8157
8158     /*
8159      * Task Management cmds are removed in their own routines. Also,
8160      * we don't want to modify timeout based on TM cmds.
8161      */
8162     if (cmd->cmd_flags & CFLAG_TM_CMD) {
8163         return;
8164     }
8165 }

```

```

8167     slot = cmd->cmd_slot;
8168
8169     /*
8170      * remove the cmd.
8171      */
8172     if (cmd == slots->m_slot[slot]) {
8173         NDBG31(("mptsas_remove_cmd: removing cmd=0x%p, flags "
8174                 "0x%lx", (void *)cmd, cmd->cmd_flags));
8175         slots->m_slot[slot] = NULL;
8176         mpt->m_ncmds--;
8177
8178         /*
8179          * only decrement per target ncmds if command
8180          * has a target associated with it.
8181          */
8182         if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
8183             ptgt->m_t_ncmds--;
8184             /*
8185              * reset throttle if we just ran an untagged command
8186              * to a tagged target
8187              */
8188             if ((ptgt->m_t_ncmds == 0) &&
8189                 ((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0)) {
8190                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
8191             }
8192
8193             /*
8194              * Remove this command from the active queue.
8195              */
8196             if (cmd->cmd_active_expiration != 0) {
8197                 TAILQ_REMOVE(&ptgt->m_active_cmdq, cmd,
8198                             cmd_active_link);
8199                 cmd->cmd_active_expiration = 0;
8200             }
8201         }
8202     }
8203
8204     /*
8205      * This is all we need to do for ioc commands.
8206      */
8207     if (cmd->cmd_flags & CFLAG_CMDIOC) {
8208         mptsas_return_to_pool(mpt, cmd);
8209         return;
8210     }
8211
8212     ASSERT(cmd != slots->m_slot[cmd->cmd_slot]);
8213 }
8214
8215 */
8216 /* accept all cmds on the tx_waitq if any and then
8217 * start a fresh request from the top of the device queue.
8218 *
8219 * since there are always cmds queued on the tx_waitq, and rare cmds on
8220 * the instance waitq, so this function should not be invoked in the ISR,
8221 * the mptsas_restart_waitq() is invoked in the ISR instead. otherwise, the
8222 * burden belongs to the IO dispatch CPUs is moved the interrupt CPU.
8223 */
8224 static void
8225 mptsas_restart_hba(mptsas_t *mpt)
8226 {
8227     ASSERT(mutex_owned(&mpt->m_mutex));
8228
8229     mutex_enter(&mpt->m_tx_waitq_mutex);
8230     if (mpt->m_tx_waitq) {
8231         mptsas_accept_tx_waitq(mpt);
8232     }

```

```

8233     mutex_exit(&mpt->m_tx_waitq_mutex);
8234     mptsas_restart_waitq(mpt);
8235 }
8236
8237 /*
8238  * start a fresh request from the top of the device queue
8239  */
8240 static void
8241 mptsas_restart_waitq(mptsas_t *mpt)
8242 {
8243     mptsas_cmd_t *cmd, *next_cmd;
8244     mptsas_target_t *ptgt = NULL;
8245
8246     NDBG1(("mptsas_restart_waitq: mpt=0x%p", (void *)mpt));
8247
8248     ASSERT(mutex_owned(&mpt->m_mutex));
8249
8250     /*
8251      * If there is a reset delay, don't start any cmds. Otherwise, start
8252      * as many cmds as possible.
8253      * Since SMID 0 is reserved and the TM slot is reserved, the actual max
8254      * commands is m_max_requests - 2.
8255      */
8256     cmd = mpt->m_waitq;
8257
8258     while (cmd != NULL) {
8259         next_cmd = cmd->cmd_linkp;
8260         if (cmd->cmd_flags & CFLAG_PASSTHRU) {
8261             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8262                 /*
8263                  * passthru command get slot need
8264                  * set CFLAG_PREPARED.
8265                  */
8266                 cmd->cmd_flags |= CFLAG_PREPARED;
8267                 mptsas_waitq_delete(mpt, cmd);
8268                 mptsas_start_passthru(mpt, cmd);
8269             }
8270             cmd = next_cmd;
8271             continue;
8272         }
8273         if (cmd->cmd_flags & CFLAG_CONFIG) {
8274             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8275                 /*
8276                  * Send the config page request and delete it
8277                  * from the waitq.
8278                  */
8279                 cmd->cmd_flags |= CFLAG_PREPARED;
8280                 mptsas_waitq_delete(mpt, cmd);
8281                 mptsas_start_config_page_access(mpt, cmd);
8282             }
8283             cmd = next_cmd;
8284             continue;
8285         }
8286         if (cmd->cmd_flags & CFLAG_FW_DIAG) {
8287             if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8288                 /*
8289                  * Send the FW Diag request and delete if from
8290                  * the waitq.
8291                  */
8292                 cmd->cmd_flags |= CFLAG_PREPARED;
8293                 mptsas_waitq_delete(mpt, cmd);
8294                 mptsas_start_diag(mpt, cmd);
8295             }
8296             cmd = next_cmd;
8297             continue;
8298         }

```

```

8300
8301     ptgt = cmd->cmd_tgt_addr;
8302     if (ptgt && (ptgt->m_t_throttle == DRAIN_THROTTLE) &&
8303         (ptgt->m_t_ncmds == 0)) {
8304         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
8305     }
8306     if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
8307         (ptgt && (ptgt->m_reset_delay == 0)) &&
8308         (ptgt && (ptgt->m_t_ncmds <
8309             ptgt->m_t_throttle))) {
8310         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
8311             mptsas_waitq_delete(mpt, cmd);
8312             (void) mptsas_start_cmd(mpt, cmd);
8313         }
8314     }
8315     cmd = next_cmd;
8316 }
8317 */
8318 * Cmds are queued if tran_start() doesn't get the m_mutexlock(no wait).
8319 * Accept all those queued cmd's before new cmd is accept so that the
8320 * cmd's are sent in order.
8321 */
8322 static void
8323 mptsas_accept_tx_waitq(mptsas_t *mpt)
8324 {
8325     mptsas_cmd_t *cmd;
8326
8327     ASSERT(mutex_owned(&mpt->m_mutex));
8328     ASSERT(mutex_owned(&mpt->m_tx_waitq_mutex));
8329
8330     /*
8331      * A Bus Reset could occur at any time and flush the tx_waitq,
8332      * so we cannot count on the tx_waitq to contain even one cmd.
8333      * And when the m_tx_waitq_mutex is released and run
8334      * mptsas_accept_pkt(), the tx_waitq may be flushed.
8335     */
8336     cmd = mpt->m_tx_waitq;
8337     for (;;) {
8338         if ((cmd = mpt->m_tx_waitq) == NULL) {
8339             mpt->m_tx_draining = 0;
8340             break;
8341         }
8342         if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL) {
8343             mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
8344         }
8345         cmd->cmd_linkp = NULL;
8346         mutex_exit(&mpt->m_tx_waitq_mutex);
8347         if (mptsas_accept_pkt(mpt, cmd) != TRAN_ACCEPT)
8348             cmn_err(CE_WARN, "mpt: mptsas_accept_tx_waitq: failed "
8349                     "to accept cmd on queue\n");
8350         mutex_enter(&mpt->m_tx_waitq_mutex);
8351     }
8352 }
8353
8354 */
8355 * mpt tag type lookup
8356 */
8357 static char mptsas_tag_lookup[] =
8358     {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};
8359
8360 static int
8361 mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
8362 {
8363     struct scsi_pkt
8364         *pkt = CMD2PKT(cmd);

```

```

8365     uint32_t
8366     caddr_t
8367     PMpi2SCSIIORequest_t
8368     ddi_dma_handle_t
8369     ddi_acc_handle_t
8370     mptsas_target_t
8371     uint16_t
8372     uint8_t
8373     uint64_t
8374     uint32_t
8375     mptsas_cmd_t
8376
8377     control = 0;
8378     mem, arsbuf;
8379     io_request;
8380     dma_hdl = mpt->m_dma_req_frame_hdl;
8381     acc_hdl = mpt->m_acc_req_frame_hdl;
8382     *ptgt = cmd->cmd_tgt_addr;
8383     SMID, io_flags = 0;
8384     ars_size;
8385     request_desc;
8386     ars_dmaaddrlow;
8387     *c;
8388
8389     NDBG1(("mptsas_start_cmd: cmd=0x%p, flags 0x%x", (void *)cmd,
8390            cmd->cmd_flags));
8391
8392     /*
8393      * Set SMID and increment index. Rollover to 1 instead of 0 if index
8394      * is at the max. 0 is an invalid SMID, so we call the first index 1.
8395     */
8396     SMID = cmd->cmd_slot;
8397
8398     /*
8399      * It is possible for back to back device reset to
8400      * happen before the reset delay has expired. That's
8401      * ok, just let the device reset go out on the bus.
8402     */
8403     if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
8404         ASSERT(ptgt->m_reset_delay == 0);
8405     }
8406
8407     /*
8408      * if a non-tagged cmd is submitted to an active tagged target
8409      * then drain before submitting this cmd; SCSI-2 allows RQSENSE
8410      * to be untagged
8411     */
8412     if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
8413         (ptgt->m_t_ncmds > 1) &&
8414         ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
8415         (*(*cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
8416         if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
8417             NDBG23(("target=%d, untagged cmd, start draining\n",
8418                    ptgt->m_devhdl));
8419
8420         if (ptgt->m_reset_delay == 0) {
8421             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
8422         }
8423
8424         mptsas_remove_cmd(mpt, cmd);
8425         cmd->cmd_pkt_flags |= FLAG_HEAD;
8426         mptsas_waitq_add(mpt, cmd);
8427     }
8428     return (DDI_FAILURE);
8429 }
8430
8431     /*
8432      * Set correct tag bits.
8433     */
8434     if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
8435         switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
8436             FLAG_TAGMASK) >> 12)]) {
8437             case MSG_SIMPLE_QTAG:
8438                 control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
8439                 break;
8440             case MSG_HEAD_QTAG:
8441                 control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
8442                 break;
8443         }
8444     }

```

```

8431     case MSG_ORDERED_QTAG:
8432         control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
8433         break;
8434     default:
8435         mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
8436         break;
8437     } else {
8438         if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
8439             ptgt->m_t_throttle = 1;
8440         }
8441         control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
8442     }
8443
8444     if (cmd->cmd_pkt_flags & FLAG_TLR) {
8445         control |= MPI2_SCSIIO_CONTROL_TLR_ON;
8446     }
8447
8448     mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
8449     io_request = (pmPi2SCSIIORequest_t)mem;
8450     if (cmd->cmd_extrqslen != 0) {
8451         /*
8452          * Mapping of the buffer was done in mptsas_pkt_alloc_extern().
8453          * Calculate the DMA address with the same offset.
8454          */
8455         arsbuf = cmd->cmd_arq_buf;
8456         ars_size = cmd->cmd_extrqslen;
8457         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
8458             ((uintptr_t)arsbuf - (uintptr_t)mpt->m_req_sense)) &
8459             0xfffffffffu;
8460     } else {
8461         arsbuf = mpt->m_req_sense + (mpt->m_req_sense_size * (SMID-1));
8462         cmd->cmd_arq_buf = arsbuf;
8463         ars_size = mpt->m_req_sense_size;
8464         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
8465             (mpt->m_req_sense_size * (SMID-1))) &
8466             0xfffffffffu;
8467     }
8468     bzero(io_request, sizeof (MpI2SCSIIORequest_t));
8469     bzero(arsbuf, ars_size);
8470
8471     ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
8472         (MPI2_SCSI_IO_REQUEST, SGL) / 4);
8473     mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
8474         MPI2_FUNCTION_SCSI_IO_REQUEST);
8475
8476     (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
8477         io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);
8478
8479     io_flags = cmd->cmd_cdblen;
8480     if (mptsas_use_fastpath &&
8481         ptgt->m_io_flags & MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
8482         io_flags |= MPI25_SCSIIO_IOFLAGS_FAST_PATH;
8483         request_desc = MPI25_REQ_DESCRIPTOR_FLAGS_FAST_PATH_SCSI_IO;
8484     } else {
8485         request_desc = MPI2_REQ_DESCRIPTOR_FLAGS_SCSI_IO;
8486     }
8487     ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
8488     /*
8489      * setup the Scatter/Gather DMA list for this request
8490      */
8491     if (cmd->cmd_cookiec > 0) {
8492         mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
8493     } else {
8494         ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
8495             ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
```

```

8497             MPI2_SGE_FLAGS_END_OF_BUFFER |
8498             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
8499             MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
8500     }
8501
8502     /* save ARQ information
8503      */
8504     ddi_put8(acc_hdl, &io_request->SenseBufferLength, ars_size);
8505     ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress, ars_dmaaddrlow);
8506
8507     ddi_put32(acc_hdl, &io_request->Control, control);
8508
8509     NDBG31(("starting message=%d(0x%p), with cmd=0x%p",
8510             SMID, (void *)io_request, (void *)cmd));
8511
8512     (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
8513     (void) ddi_dma_sync(ptgt->m_dma_req_sense_hdl, 0, 0,
8514         DDI_DMA_SYNC_FORDEV);
8515
8516     /*
8517      * Build request descriptor and write it to the request desc post reg.
8518      */
8519     request_desc |= (SMID << 16);
8520     request_desc |= (uint64_t)ptgt->m_devhdl << 48;
8521     MPTSAS_START_CMD(mpt, request_desc);
8522
8523     /*
8524      * Start timeout.
8525      */
8526     cmd->cmd_active_expiration =
8527         gethrtime() + (hrtime_t)pkt->pkt_time * NANOSEC;
8528 #ifdef MPTSAS_TEST
8529     /*
8530      * Force timeouts to happen immediately.
8531      */
8532     if (mptsas_test_timeouts)
8533         cmd->cmd_active_expiration = gethrtime();
8534 #endif
8535     c = TAILQ_FIRST(&ptgt->m_active_cmdq);
8536     if (c == NULL ||
8537         c->cmd_active_expiration < cmd->cmd_active_expiration) {
8538         /*
8539          * Common case is that this is the last pending expiration
8540          * (or queue is empty). Insert at head of the queue.
8541          */
8542         TAILQ_INSERT_HEAD(&ptgt->m_active_cmdq, cmd, cmd_active_link);
8543     } else {
8544         /*
8545          * Queue is not empty and first element expires later than
8546          * this command. Search for element expiring sooner.
8547          */
8548         while ((c = TAILQ_NEXT(c, cmd_active_link)) != NULL) {
8549             if (c->cmd_active_expiration <
8550                 cmd->cmd_active_expiration) {
8551                 TAILQ_INSERT_BEFORE(c, cmd, cmd_active_link);
8552                 break;
8553             }
8554         }
8555     }
8556     if (c == NULL) {
8557         /*
8558          * No element found expiring sooner, append to
8559          * non-empty queue.
8560          */
8561     TAILQ_INSERT_TAIL(&ptgt->m_active_cmdq, cmd,
8562         cmd_active_link);
```

```

8563         }
8564     }
8565     if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
8566         (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
8567         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8568         return (DDI_FAILURE);
8569     }
8570     return (DDI_SUCCESS);
8571 }
8572 }

8574 /*
8575 * Select a helper thread to handle current doneq
8576 */
8577 static void
8578 mptsas_deliver_doneq_thread(mptsas_t *mpt)
8579 {
8580     uint64_t t, i;
8581     uint32_t min = 0xffffffff;
8582     mptsas_doneq_thread_list_t *item;
8583
8584     for (i = 0; i < mpt->m_doneq_thread_n; i++) {
8585         item = &mpt->m_doneq_thread_id[i];
8586         /*
8587          * If the completed command on help thread[i] less than
8588          * doneq_thread_threshold, then pick the thread[i]. Otherwise
8589          * pick a thread which has least completed command.
8590         */
8591
8592         mutex_enter(&item->mutex);
8593         if (item->len < mpt->m_doneq_thread_threshold) {
8594             t = i;
8595             mutex_exit(&item->mutex);
8596             break;
8597         }
8598         if (item->len < min) {
8599             min = item->len;
8600             t = i;
8601         }
8602         mutex_exit(&item->mutex);
8603     }
8604     mutex_enter(&mpt->m_doneq_thread_id[t].mutex);
8605     mptsas_doneq_mv(mpt, t);
8606     cv_signal(&mpt->m_doneq_thread_id[t].cv);
8607     mutex_exit(&mpt->m_doneq_thread_id[t].mutex);
8608 }

8610 /*
8611 * move the current global doneq to the doneq of thread[t]
8612 */
8613 static void
8614 mptsas_doneq_mv(mptsas_t *mpt, uint64_t t)
8615 {
8616     mptsas_cmd_t *cmd;
8617     mptsas_doneq_thread_list_t *item = &mpt->m_doneq_thread_id[t];
8618
8619     ASSERT(mutex_owned(&item->mutex));
8620     while ((cmd = mpt->m_doneq) != NULL) {
8621         if ((mpt->m_doneq = cmd->cmd_linkp) == NULL) {
8622             mpt->m_donetail = &mpt->m_doneq;
8623         }
8624         cmd->cmd_linkp = NULL;
8625         *item->donetail = cmd;
8626         item->donetail = &cmd->cmd_linkp;
8627         mpt->m_doneq_len--;
8628         item->len++;
8629     }
8630 }

```

```

8629     }
8630 }

8632 void
8633 mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd)
8634 {
8635     struct scsi_pkt *pkt = CMD2PKT(cmd);

8636     /* Check all acc and dma handles */
8637     if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
8638         (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) != DDI_SUCCESS) ||
8639         (mptsas_check_acc_handle(mpt->m_acc_req_sense_hdl) != DDI_SUCCESS) ||
8640         (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) != DDI_SUCCESS) ||
8641         (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) != DDI_SUCCESS) ||
8642         (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) != DDI_SUCCESS) ||
8643         (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) != DDI_SUCCESS) ||
8644         (mptsas_check_acc_handle(mpt->m_config_handle) != DDI_SUCCESS)) {
8645         ddi_fm_service_impact(mpt->m_dip,
8646                               DDI_SERVICE_UNAFFECTED);
8647         ddi_fm_acc_err_clear(mpt->m_config_handle,
8648                               DDI_FME_VERO);
8649         pkt->pkt_reason = CMD_TRAN_ERR;
8650         pkt->pkt_statistics = 0;
8651     }
8652     if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) != DDI_SUCCESS) ||
8653         (mptsas_check_dma_handle(mpt->m_dma_req_sense_hdl) != DDI_SUCCESS) ||
8654         (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) != DDI_SUCCESS) ||
8655         (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) != DDI_SUCCESS) ||
8656         (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) != DDI_SUCCESS) ||
8657         (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) != DDI_SUCCESS)) {
8658         ddi_fm_service_impact(mpt->m_dip,
8659                               DDI_SERVICE_UNAFFECTED);
8660         pkt->pkt_reason = CMD_TRAN_ERR;
8661         pkt->pkt_statistics = 0;
8662     }
8663     if (cmd->cmd_dmahandle &&
8664         (mptsas_check_dma_handle(cmd->cmd_dmahandle) != DDI_SUCCESS)) {
8665         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8666         pkt->pkt_reason = CMD_TRAN_ERR;
8667         pkt->pkt_statistics = 0;
8668     }
8669     if ((cmd->cmd_extra_frames &&
8670         ((mptsas_check_dma_handle(cmd->cmd_extra_frames->m_dma_hdl) != DDI_SUCCESS) ||
8671         (mptsas_check_acc_handle(cmd->cmd_extra_frames->m_acc_hdl) != DDI_SUCCESS))) {
8672         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8673         pkt->pkt_reason = CMD_TRAN_ERR;
8674         pkt->pkt_statistics = 0;
8675     }
8676 }
8677
8678
8679
8680
8681
8682
8683
8684
8685
8686
8687
8688
8689
8690
8691
8692
8693

```

```

8695 /*
8696  * These routines manipulate the queue of commands that
8697  * are waiting for their completion routines to be called.
8698  * The queue is usually in FIFO order but on an MP system
8699  * it's possible for the completion routines to get out
8700  * of order. If that's a problem you need to add a global
8701  * mutex around the code that calls the completion routine
8702  * in the interrupt handler.
8703 */
8704 static void
8705 mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8706 {
8707     struct scsi_pkt *pkt = CMD2PKT(cmd);
8708
8709     NDBG31(("mptsas_doneq_add: cmd=0x%p", (void *)cmd));
8710
8711     ASSERT((cmd->cmd_flags & CFLAG_COMPLETED) == 0);
8712     cmd->cmd_linkp = NULL;
8713     cmd->cmd_flags |= CFLAG_FINISHED;
8714     cmd->cmd_flags &= ~CFLAG_IN_TRANSPORT;
8715
8716     mptsas_fma_check(mpt, cmd);
8717
8718     /*
8719      * only add scsi pkts that have completion routines to
8720      * the doneq. no intr cmd's do not have callbacks.
8721      */
8722     if (pkt && (pkt->pkt_comp)) {
8723         *mpt->m_donetail = cmd;
8724         mpt->m_donetail = &cmd->cmd_linkp;
8725         mpt->m_doneq_len++;
8726     }
8727 }
8728 static mptsas_cmd_t *
8729 mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t)
8730 {
8731     mptsas_cmd_t          *cmd;
8732     mptsas_doneq_thread_list_t    *item = &mpt->m_doneq_thread_id[t];
8733
8734     /* pop one off the done queue */
8735     if ((cmd = item->doneq) != NULL) {
8736         /* if the queue is now empty fix the tail pointer */
8737         NDBG31(("mptsas_doneq_thread_rm: cmd=0x%p", (void *)cmd));
8738         if ((item->doneq = cmd->cmd_linkp) == NULL) {
8739             item->donetail = &item->doneq;
8740         }
8741         cmd->cmd_linkp = NULL;
8742         item->len--;
8743     }
8744     return (cmd);
8745 }
8746
8747 static void
8748 mptsas_doneq_empty(mptsas_t *mpt)
8749 {
8750     if (mpt->m_doneq && !mpt->m_in_callback) {
8751         mptsas_cmd_t *cmd, *next;
8752         struct scsi_pkt *pkt;
8753
8754         mpt->m_in_callback = 1;
8755         cmd = mpt->m_doneq;
8756         mpt->m_doneq = NULL;
8757         mpt->m_donetail = &mpt->m_doneq;
8758         mpt->m_doneq_len = 0;
8759     }

```

```

8761         mutex_exit(&mpt->m_mutex);
8762
8763         /*
8764          * run the completion routines of all the
8765          * completed commands
8766          */
8767         while (cmd != NULL) {
8768             next = cmd->cmd_linkp;
8769             cmd->cmd_linkp = NULL;
8770             /* run this command's completion routine */
8771             cmd->cmd_flags |= CFLAG_COMPLETED;
8772             pkt = CMD2PKT(cmd);
8773             mptsas_pkt_comp(pkt, cmd);
8774             cmd = next;
8775         }
8776         mutex_enter(&mpt->m_mutex);
8777         mpt->m_in_callback = 0;
8778     }
8779
8780     /*
8781      * These routines manipulate the target's queue of pending requests
8782      */
8783     void
8784     mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8785     {
8786         NDBG7(("mptsas_waitq_add: cmd=0x%p", (void *)cmd));
8787         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8788         cmd->cmd_queued = TRUE;
8789         if (ptgt)
8790             ptgt->m_t_nwait++;
8791         if (cmd->cmd_pkt_flags & FLAG_HEAD) {
8792             if ((cmd->cmd_linkp = mpt->m_waitq) == NULL) {
8793                 mpt->m_waitqtail = &cmd->cmd_linkp;
8794             }
8795             mpt->m_waitq = cmd;
8796         } else {
8797             cmd->cmd_linkp = NULL;
8798             *(mpt->m_waitqtail) = cmd;
8799             mpt->m_waitqtail = &cmd->cmd_linkp;
8800         }
8801     }
8802
8803     static mptsas_cmd_t *
8804     mptsas_waitq_rm(mptsas_t *mpt)
8805     {
8806         mptsas_cmd_t          *cmd;
8807         mptsas_target_t *ptgt;
8808         NDBG7(("mptsas_waitq_rm"));
8809
8810         MPTSAS_WAITQ_RM(mpt, cmd);
8811
8812         NDBG7(("mptsas_waitq_rm: cmd=0x%p", (void *)cmd));
8813         if (cmd) {
8814             ptgt = cmd->cmd_tgt_addr;
8815             if (ptgt) {
8816                 ptgt->m_t_nwait--;
8817                 ASSERT(ptgt->m_t_nwait >= 0);
8818             }
8819         }
8820         return (cmd);
8821     }
8822
8823 /*
8824  * remove specified cmd from the middle of the wait queue.
8825  */
8826     static void

```

```

8827 mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8828 {
8829     mptsas_cmd_t *prevp = mpt->m_waitq;
8830     mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8831
8832     NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8833            (void *)mpt, (void *)cmd));
8834     if (ptgt) {
8835         ptgt->m_t_nwait--;
8836         ASSERT(ptgt->m_t_nwait >= 0);
8837     }
8838
8839     if (prevp == cmd) {
8840         if ((mpt->m_waitq = cmd->cmd_linkp) == NULL)
8841             mpt->m_waitqtail = &mpt->m_waitq;
8842
8843         cmd->cmd_linkp = NULL;
8844         cmd->cmd_queued = FALSE;
8845         NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8846                (void *)mpt, (void *)cmd));
8847         return;
8848     }
8849
8850     while (prevp != NULL) {
8851         if (prevp->cmd_linkp == cmd) {
8852             if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8853                 mpt->m_waitqtail = &prevp->cmd_linkp;
8854
8855             cmd->cmd_linkp = NULL;
8856             cmd->cmd_queued = FALSE;
8857             NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8858                    (void *)mpt, (void *)cmd));
8859             return;
8860         }
8861         prevp = prevp->cmd_linkp;
8862     }
8863     cmn_err(CE_PANIC, "mpt: mptsas_waitq_delete: queue botch");
8864 }
8865
8866 static mptsas_cmd_t *
8867 mptsas_tx_waitq_rm(mptsas_t *mpt)
8868 {
8869     mptsas_cmd_t *cmd;
8870     NDBG7(("mptsas_tx_waitq_rm"));
8871
8872     MPTSAWS_TX_WAITQ_RM(mpt, cmd);
8873
8874     NDBG7(("mptsas_tx_waitq_rm: cmd=0x%p", (void *)cmd));
8875
8876     return (cmd);
8877 }
8878
8879 */
8880 /* remove specified cmd from the middle of the tx_waitq.
8881 */
8882 static void
8883 mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8884 {
8885     mptsas_cmd_t *prevp = mpt->m_tx_waitq;
8886
8887     NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8888            (void *)mpt, (void *)cmd));
8889
8890     if (prevp == cmd) {
8891         if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL)
8892             mpt->m_tx_waitqtail = &mpt->m_tx_waitq;

```

```

8893
8894     cmd->cmd_linkp = NULL;
8895     cmd->cmd_queued = FALSE;
8896     NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8897            (void *)mpt, (void *)cmd));
8898     return;
8899 }
8900
8901     while (prevp != NULL) {
8902         if (prevp->cmd_linkp == cmd) {
8903             if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8904                 mpt->m_tx_waitqtail = &prevp->cmd_linkp;
8905
8906             cmd->cmd_linkp = NULL;
8907             cmd->cmd_queued = FALSE;
8908             NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8909                    (void *)mpt, (void *)cmd));
8910             return;
8911         }
8912         prevp = prevp->cmd_linkp;
8913     }
8914     cmn_err(CE_PANIC, "mpt: mptsas_tx_waitq_delete: queue botch");
8915 }
8916
8917 /*
8918  * device and bus reset handling
8919  *
8920  * Notes:
8921  * - RESET_ALL:    reset the controller
8922  * - RESET_TARGET: reset the target specified in scsi_address
8923  */
8924 static int
8925 mptsas_scsi_reset(struct scsi_address *ap, int level)
8926 {
8927     mptsas_t *mpt = ADDR2MPT(ap);
8928     int rval;
8929     mptsas_tgt_private_t *tgt_private;
8930     mptsas_target_t *ptgt = NULL;
8931
8932     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->tran_tgt_private;
8933     ptgt = tgt_private->t_private;
8934     if (ptgt == NULL) {
8935         return (FALSE);
8936     }
8937     NDBG22(("mptsas_scsi_reset: target=%d level=%d", ptgt->m_devhdl,
8938             level));
8939
8940     mutex_enter(&mpt->m_mutex);
8941     /*
8942      * if we are not in panic set up a reset delay for this target
8943      */
8944     if (!ddi_in_panic()) {
8945         mptsas_setup_bus_reset_delay(mpt);
8946     } else {
8947         drv_usecwait(mpt->m_scsi_reset_delay * 1000);
8948     }
8949     rval = mptsas_do_scsi_reset(mpt, ptgt->m_devhdl);
8950     mutex_exit(&mpt->m_mutex);
8951
8952     /*
8953      * The transport layer expect to only see TRUE and
8954      * FALSE. Therefore, we will adjust the return value
8955      * if mptsas_do_scsi_reset returns FAILED.
8956      */
8957     if (rval == FAILED)
8958         rval = FALSE;

```

```

8959     return (rval);
8960 }

8962 static int
8963 mptsas_do_scsi_reset(mptsas_t *mpt, uint16_t devhdl)
8964 {
8965     int          rval = FALSE;
8966     uint8_t      config, disk;
8968     ASSERT(mutex_owned(&mpt->m_mutex));

8970     if (mptsas_debug_resets) {
8971         mptsas_log(mpt, CE_WARN, "mptsas_do_scsi_reset: target=%d",
8972                     devhdl);
8973     }

8975     /*
8976      * Issue a Target Reset message to the target specified but not to a
8977      * disk making up a raid volume. Just look through the RAID config
8978      * Phys Disk list of DevHandles. If the target's DevHandle is in this
8979      * list, then don't reset this target.
8980      */
8981     for (config = 0; config < mpt->m_num_raid_configs; config++) {
8982         for (disk = 0; disk < MPTAS_MAX_DISKS_IN_CONFIG; disk++) {
8983             if (devhdl == mpt->m_raidconfig[config].
8984                 m_physdisk_devhdl[disk]) {
8985                 return (TRUE);
8986             }
8987         }
8988     }

8990     rval = mptsas_ioc_task_management(mpt,
8991         MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET, devhdl, 0, NULL, 0, 0);

8993     mptsas_doneq_empty(mpt);
8994     return (rval);
8995 }

8997 static int
8998 mptsas_scsi_reset_notify(struct scsi_address *ap, int flag,
8999     void (*callback)(caddr_t), caddr_t arg)
9000 {
9001     mptsas_t      *mpt = ADDR2MPT(ap);

9003     NDBG22(("mptsas_scsi_reset_notify: tgt=%d", ap->a_target));

9005     return (scsi_hba_reset_notify_setup(ap, flag, callback, arg,
9006         &mpt->m_mutex, &mpt->m_reset_notify_listf));
9007 }

9009 static int
9010 mptsas_get_name(struct scsi_device *sd, char *name, int len)
9011 {
9012     dev_info_t    *lun_dip = NULL;

9014     ASSERT(sd != NULL);
9015     ASSERT(name != NULL);
9016     lun_dip = sd->sd_dev;
9017     ASSERT(lun_dip != NULL);

9019     if (mptsas_name_child(lun_dip, name, len) == DDI_SUCCESS) {
9020         return (1);
9021     } else {
9022         return (0);
9023     }
9024 }

```

```

9026 static int
9027 mptsas_get_bus_addr(struct scsi_device *sd, char *name, int len)
9028 {
9029     return (mptsas_get_name(sd, name, len));
9030 }

9032 void
9033 mptsas_set_throttle(mptsas_t *mpt, mptsas_target_t *ptgt, int what)
9034 {
9036     NDBG25(("mptsas_set_throttle: throttle=%x", what));

9038     /*
9039      * if the bus is draining/quiesced, no changes to the throttles
9040      * are allowed. Not allowing change of throttles during draining
9041      * limits error recovery but will reduce draining time
9042      *
9043      * all throttles should have been set to HOLD_THROTTLE
9044      */
9045     if (mpt->m_softstate & (MPTSAS_SS QUIESCED | MPTSAS_SS DRAINING)) {
9046         return;
9047     }

9049     if (what == HOLD_THROTTLE) {
9050         ptgt->m_t_throttle = HOLD_THROTTLE;
9051     } else if (ptgt->m_reset_delay == 0) {
9052         ptgt->m_t_throttle = what;
9053     }
9054 }

9056 /*
9057  * Clean up from a device reset.
9058  * For the case of target reset, this function clears the waitq of all
9059  * commands for a particular target. For the case of abort task set, this
9060  * function clears the waitq of all commands for a particular target/lun.
9061 */
9062 static void
9063 mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
9064 {
9065     mptsas_slots_t *slots = mpt->m_active;
9066     mptsas_cmd_t   *cmd, *next_cmd;
9067     int            slot;
9068     uchar_t        reason;
9069     uint_t          stat;
9070     hrtime_t       timestamp;

9072     NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));

9074     timestamp = gethrtime();

9076     /*
9077      * Make sure the I/O Controller has flushed all cmd
9078      * that are associated with this target for a target reset
9079      * and target/lun for abort task set.
9080      * Account for TM requests, which use the last SMID.
9081      */
9082     for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
9083         if ((cmd = slots->m_slot[slot]) == NULL)
9084             continue;
9085         reason = CMD_RESET;
9086         stat = STAT_DEV_RESET;
9087         switch (tasktype) {
9088             case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
9089                 if (Tgt(cmd) == target) {
9090                     if (cmd->cmd_active_expiration <= timestamp) {

```

```

9091             /*
9092              * When timeout requested, propagate
9093              * proper reason and statistics to
9094              * target drivers.
9095              */
9096             reason = CMD_TIMEOUT;
9097             stat |= STAT_TIMEOUT;
9098         }
9099         NDBG25(("mptsas_flush_target discovered non-"
9100                 "NULL cmd in slot %d, tasktype 0x%x", slot,
9101                 tasktype));
9102         mptsas_dump_cmd(mpt, cmd);
9103         mptsas_remove_cmd(mpt, cmd);
9104         mptsas_set_pkt_reason(mpt, cmd, reason, stat);
9105         mptsas_doneq_add(mpt, cmd);
9106     }
9107     break;
9108 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
9109     reason = CMD_ABORTED;
9110     stat = STAT_ABORTED;
9111     /*FALLTHROUGH*/
9112 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
9113     if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9114
9115         NDBG25(("mptsas_flush_target discovered non-"
9116                 "NULL cmd in slot %d, tasktype 0x%x", slot,
9117                 tasktype));
9118         mptsas_dump_cmd(mpt, cmd);
9119         mptsas_remove_cmd(mpt, cmd);
9120         mptsas_set_pkt_reason(mpt, cmd, reason,
9121                               stat);
9122         mptsas_doneq_add(mpt, cmd);
9123     }
9124     break;
9125 default:
9126     break;
9127 }
9128 }

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

reason = CMD_RESET;
stat = STAT_DEV_RESET;

switch (tasktype) {
case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
    while (cmd != NULL) {
        next_cmd = cmd->cmd_linkp;
        if (Tgt(cmd) == target) {
            mptsas_waitq_delete(mpt, cmd);
            mptsas_set_pkt_reason(mpt, cmd,
                                  reason, stat);
            mptsas_doneq_add(mpt, cmd);
        }
        cmd = next_cmd;
    }
    mutex_enter(&mpt->m_tx_waitq_mutex);
    cmd = mpt->m_tx_waitq;
    while (cmd != NULL) {
        next_cmd = cmd->cmd_linkp;
        if (Tgt(cmd) == target) {
            mptsas_tx_waitq_delete(mpt, cmd);
            mutex_exit(&mpt->m_tx_waitq_mutex);
        }
    }
}

```

```

9157             mptsas_set_pkt_reason(mpt, cmd,
9158                         reason, stat);
9159             mptsas_doneq_add(mpt, cmd);
9160             mutex_enter(&mpt->m_tx_waitq_mutex);
9161         }
9162         cmd = next_cmd;
9163     }
9164     mutex_exit(&mpt->m_tx_waitq_mutex);
9165     break;
9166 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
9167     reason = CMD_ABORTED;
9168     stat = STAT_ABORTED;
9169     /*FALLTHROUGH*/
9170 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
9171     while (cmd != NULL) {
9172         next_cmd = cmd->cmd_linkp;
9173         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9174             mptsas_waitq_delete(mpt, cmd);
9175             mptsas_set_pkt_reason(mpt, cmd,
9176                                   reason, stat);
9177             mptsas_doneq_add(mpt, cmd);
9178         }
9179         cmd = next_cmd;
9180     }
9181     mutex_enter(&mpt->m_tx_waitq_mutex);
9182     cmd = mpt->m_tx_waitq;
9183     while (cmd != NULL) {
9184         next_cmd = cmd->cmd_linkp;
9185         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
9186             mptsas_tx_waitq_delete(mpt, cmd);
9187             mutex_exit(&mpt->m_tx_waitq_mutex);
9188             mptsas_set_pkt_reason(mpt, cmd,
9189                                   reason, stat);
9190             mptsas_doneq_add(mpt, cmd);
9191             mutex_enter(&mpt->m_tx_waitq_mutex);
9192         }
9193         cmd = next_cmd;
9194     }
9195     mutex_exit(&mpt->m_tx_waitq_mutex);
9196     break;
9197 default:
9198     mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
9199                tasktype);
9200     break;
9201 }
9202 }

9204 /*
9205  * Clean up hba state, abort all outstanding command and commands in waitq
9206  * reset timeout of all targets.
9207  */
9208 static void
9209 mptsas_flush_hba(mptsas_t *mpt)
9210 {
9211     mptsas_slots_t *slots = mpt->m_active;
9212     mptsas_cmd_t *cmd;
9213     int slot;

9215     NDBG25(("mptsas_flush_hba"));

9217     /*
9218      * The I/O Controller should have already sent back
9219      * all commands via the scsi I/O reply frame. Make
9220      * sure all commands have been flushed.
9221      * Account for TM request, which use the last SMID.
9222      */

```

```

9223     for (slot = 0; slot <= mpt->m_active->m_n_normal; slot++) {
9224         if ((cmd = slots->m_slot[slot]) == NULL)
9225             continue;
9226
9227         if (cmd->cmd_flags & CFLAG_CMDIOC) {
9228             /*
9229              * Need to make sure to tell everyone that might be
9230              * waiting on this command that it's going to fail. If
9231              * we get here, this command will never timeout because
9232              * the active command table is going to be re-allocated,
9233              * so there will be nothing to check against a time out.
9234              * Instead, mark the command as failed due to reset.
9235
9236             */
9237             mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
9238                         STAT_BUS_RESET);
9239             if ((cmd->cmd_flags &
9240                  (CFLAG_PASSTHRU | CFLAG_CONFIG | CFLAG_FW_DIAG)) {
9241                 cmd->cmd_flags |= CFLAG_FINISHED;
9242                 cv_broadcast(&mpt->m_passthru_cv);
9243                 cv_broadcast(&mpt->m_config_cv);
9244                 cv_broadcast(&mpt->m_fw_diag_cv);
9245             }
9246             continue;
9247         }
9248
9249         NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
9250                slot));
9251         mptsas_dump_cmd(mpt, cmd);
9252
9253         mptsas_remove_cmd(mpt, cmd);
9254         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9255         mptsas_doneq_add(mpt, cmd);
9256     }
9257
9258     /*
9259      * Flush the waitq.
9260      */
9261     while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
9262         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9263         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
9264             (cmd->cmd_flags & CFLAG_CONFIG) ||
9265             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
9266             cmd->cmd_flags |= CFLAG_FINISHED;
9267             cv_broadcast(&mpt->m_passthru_cv);
9268             cv_broadcast(&mpt->m_config_cv);
9269             cv_broadcast(&mpt->m_fw_diag_cv);
9270         } else {
9271             mptsas_doneq_add(mpt, cmd);
9272         }
9273
9274     /*
9275      * Flush the tx_waitq
9276      */
9277     mutex_enter(&mpt->m_tx_waitq_mutex);
9278     while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
9279         mutex_exit(&mpt->m_tx_waitq_mutex);
9280         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
9281         mptsas_doneq_add(mpt, cmd);
9282         mutex_enter(&mpt->m_tx_waitq_mutex);
9283     }
9284     mutex_exit(&mpt->m_tx_waitq_mutex);
9285
9286     /*
9287      * Drain the taskqs prior to reallocating resources.
9288      */

```

```

9289     mutex_exit(&mpt->m_mutex);
9290     ddi_taskq_wait(mpt->m_event_taskq);
9291     ddi_taskq_wait(mpt->m_dr_taskq);
9292     mutex_enter(&mpt->m_mutex);
9293 }
9294
9295 /*
9296  * set pkt_reason and OR in pkt_statistics flag
9297 */
9298 static void
9299 mptsas_set_pkt_reason(mptsas_t *mpt, mptsas_cmd_t *cmd, uchar_t reason,
9300                         uint_t stat)
9301 {
9302 #ifndef __lock_lint
9303     _NOTE(ARGUNUSED(mpt))
9304 #endif
9305
9306     NDBG25(("mptsas_set_pkt_reason: cmd=0x%p reason=%x stat=%x",
9307             (void *)cmd, reason, stat));
9308
9309     if (cmd) {
9310         if (cmd->cmd_pkt->pkt_reason == CMD_CMPLT) {
9311             cmd->cmd_pkt->pkt_reason = reason;
9312         }
9313         cmd->cmd_pkt->pkt_statistics |= stat;
9314     }
9315 }
9316
9317 static void
9318 mptsas_start_watch_reset_delay()
9319 {
9320     NDBG22(("mptsas_start_watch_reset_delay"));
9321
9322     mutex_enter(&mptsas_global_mutex);
9323     if (mptsas_reset_watch == NULL && mptsas_timeouts_enabled) {
9324         mptsas_reset_watch = timeout(mptsas_watch_reset_delay, NULL,
9325                                     drv_usectohz((clock_t)
9326                                     MPTAS_WATCH_RESET_DELAY_TICK * 1000));
9327         ASSERT(mptsas_reset_watch != NULL);
9328     }
9329     mutex_exit(&mptsas_global_mutex);
9330 }
9331
9332 static void
9333 mptsas_setup_bus_reset_delay(mptsas_t *mpt)
9334 {
9335     mptsas_target_t *ptgt = NULL;
9336
9337     ASSERT(MUTEX_HELD(&mpt->m_mutex));
9338
9339     NDBG22(("mptsas_setup_bus_reset_delay"));
9340     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
9341          ptgt = refhash_next(mpt->m_targets, ptgt)) {
9342         mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
9343         ptgt->m_reset_delay = mpt->m_scsi_reset_delay;
9344     }
9345
9346     mptsas_start_watch_reset_delay();
9347 }
9348
9349 /*
9350  * mptsas_watch_reset_delay(_subr) is invoked by timeout() and checks every
9351  * mpt instance for active reset delays
9352 */
9353 static void
9354 mptsas_watch_reset_delay(void *arg)

```

```

9355 {
9356 #ifndef __lock_lint
9357     _NOTE(ARGUNUSED(arg))
9358 #endif
9359
9360     mptsas_t      *mpt;
9361     int           not_done = 0;
9362
9363     NDBG22(("mptsas_watch_reset_delay"));
9364
9365     mutex_enter(&mptsas_global_mutex);
9366     mptsas_reset_watch = 0;
9367     mutex_exit(&mptsas_global_mutex);
9368     rw_enter(&mptsas_global_rwlock, RW_READER);
9369     for (mpt = mptsas_head; mpt != NULL; mpt = mpt->m_next) {
9370         if (mpt->m_tran == 0) {
9371             continue;
9372         }
9373         mutex_enter(&mpt->m_mutex);
9374         not_done += mptsas_watch_reset_delay_subr(mpt);
9375         mutex_exit(&mpt->m_mutex);
9376     }
9377     rw_exit(&mptsas_global_rwlock);
9378
9379     if (not_done) {
9380         mptsas_start_watch_reset_delay();
9381     }
9382 }
9383
9384 static int
9385 mptsas_watch_reset_delay_subr(mptsas_t *mpt)
9386 {
9387     int          done = 0;
9388     int          restart = 0;
9389     mptsas_target_t *ptgt = NULL;
9390
9391     NDBG22(("mptsas_watch_reset_delay_subr: mpt=0x%p", (void *)mpt));
9392
9393     ASSERT(mutex_owned(&mpt->m_mutex));
9394
9395     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
9396          ptgt = rehash_next(mpt->m_targets, ptgt)) {
9397         if (ptgt->m_reset_delay != 0) {
9398             ptgt->m_reset_delay -=
9399                         MPTSAS_WATCH_RESET_DELAY_TICK;
9400             if (ptgt->m_reset_delay <= 0) {
9401                 ptgt->m_reset_delay = 0;
9402                 mptsas_set_throttle(mpt, ptgt,
9403                                     MAX_THROTTLE);
9404                 restart++;
9405             } else {
9406                 done = -1;
9407             }
9408         }
9409     }
9410
9411     if (restart > 0) {
9412         mptsas_restart_hba(mpt);
9413     }
9414     return (done);
9415 }
9416
9417 #ifdef MPTSAES_TEST
9418 static void
9419 mptsas_test_reset(mptsas_t *mpt, int target)
9420 {

```

```

9421     mptsas_target_t    *ptgt = NULL;
9422
9423     if (mptsas_rtest == target) {
9424         if (mptsas_do_scsi_reset(mpt, target) == TRUE) {
9425             mptsas_rtest = -1;
9426         }
9427         if (mptsas_rtest == -1) {
9428             NDBG22(("mptsas_test_reset success"));
9429         }
9430     }
9431 }
9432 #endif
9433
9434 /*
9435  * abort handling:
9436  *
9437  * Notes:
9438  *   - if pkt is not NULL, abort just that command
9439  *   - if pkt is NULL, abort all outstanding commands for target
9440  */
9441 static int
9442 mptsas_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
9443 {
9444     mptsas_t          *mpt = ADDR2MPT(ap);
9445     int               rval;
9446     mptsas_tgt_private_t *tgt_private;
9447     int               target, lun;
9448
9449     tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
9450                 tran_tgt_private;
9451     ASSERT(tgt_private != NULL);
9452     target = tgt_private->t_private->m_devhdl;
9453     lun = tgt_private->t_lun;
9454
9455     NDBG23(("mptsas_scsi_abort: target=%d.%d", target, lun));
9456
9457     mutex_enter(&mpt->m_mutex);
9458     rval = mptsas_do_scsi_abort(mpt, target, lun, pkt);
9459     mutex_exit(&mpt->m_mutex);
9460     return (rval);
9461 }
9462
9463 static int
9464 mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun, struct scsi_pkt *pkt)
9465 {
9466     mptsas_cmd_t      *sp = NULL;
9467     mptsas_slots_t   *slots = mpt->m_active;
9468     int               rval = FALSE;
9469
9470     ASSERT(mutex_owned(&mpt->m_mutex));
9471
9472     /*
9473      * Abort the command pkt on the target/lun in ap. If pkt is
9474      * NULL, abort all outstanding commands on that target/lun.
9475      * If you can abort them, return 1, else return 0.
9476      * Each packet that's aborted should be sent back to the target
9477      * driver through the callback routine, with pkt_reason set to
9478      * CMD_ABORTED.
9479      *
9480      * abort cmd pkt on HBA hardware; clean out of outstanding
9481      * command lists, etc.
9482      */
9483     if (pkt != NULL) {
9484         /* abort the specified packet */
9485         sp = PKT2CMD(pkt);

```

```

9487     if (sp->cmd_queued) {
9488         NDBG23(("mptsas_do_scsi_abort: queued sp=0x%p aborted",
9489                 (void *)sp));
9490         mptsas_waitq_delete(mpt, sp);
9491         mptsas_set_pkt_reason(mpt, sp, CMD_ABORTED,
9492                                STAT_ABORTED);
9493         mptsas_doneq_add(mpt, sp);
9494         rval = TRUE;
9495         goto done;
9496     }
9497
9498     /*
9499      * Have mpt firmware abort this command
9500     */
9501
9502     if (slots->m_slot[sp->cmd_slot] != NULL) {
9503         rval = mptsas_ioc_task_management(mpt,
9504                                         MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK, target,
9505                                         lun, NULL, 0, 0);
9506
9507     /*
9508      * The transport layer expects only TRUE and FALSE.
9509      * Therefore, if mptsas_ioc_task_management returns
9510      * FAILED we will return FALSE.
9511     */
9512     if (rval == FAILED)
9513         rval = FALSE;
9514     goto done;
9515 }
9516
9517 /*
9518  * If pkt is NULL then abort task set
9519 */
9520 rval = mptsas_ioc_task_management(mpt,
9521                                         MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET, target, lun, NULL, 0, 0);
9522
9523 /*
9524  * The transport layer expects only TRUE and FALSE.
9525  * Therefore, if mptsas_ioc_task_management returns
9526  * FAILED we will return FALSE.
9527 */
9528 if (rval == FAILED)
9529     rval = FALSE;
9530
9531 #ifdef MPTSAS_TEST
9532     if (rval && mptsas_test_stop) {
9533         debug_enter("mptsas_do_scsi_abort");
9534     }
9535 #endif
9536
9537 done:
9538     mptsas_doneq_empty(mpt);
9539     return (rval);
9540 }
9541
9542 /*
9543  * capability handling:
9544  * (*tran_getcap). Get the capability named, and return its value.
9545 */
9546 static int
9547 mptsas_scsi_getcap(struct scsi_address *ap, char *cap, int tgtonly)
9548 {
9549     mptsas_t          *mpt = ADDR2MPT(ap);
9550     int                ckey;
9551     int                rval = FALSE;

```

```

9554     NDBG24(("mptsas_scsi_getcap: target=%d, cap=%s tgtonly=%x",
9555             ap->a_target, cap, tgtonly));
9556
9557     mutex_enter(&mpt->m_mutex);
9558
9559     if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9560         mutex_exit(&mpt->m_mutex);
9561         return (UNDEFINED);
9562     }
9563
9564     switch (ckey) {
9565     case SCSI_CAP_DMA_MAX:
9566         rval = (int)mpt->m_msg_dma_attr.dma_attr_maxxfier;
9567         break;
9568     case SCSI_CAP_ARQ:
9569         rval = TRUE;
9570         break;
9571     case SCSI_CAP_MSG_OUT:
9572     case SCSI_CAP_PARITY:
9573     case SCSI_CAP_UNTAGGED_QING:
9574         rval = TRUE;
9575         break;
9576     case SCSI_CAP_TAGGED_QING:
9577         rval = TRUE;
9578         break;
9579     case SCSI_CAP_RESET_NOTIFICATION:
9580         rval = TRUE;
9581         break;
9582     case SCSI_CAP_LINKED_CMDS:
9583         rval = FALSE;
9584         break;
9585     case SCSI_CAP_QFULL_RETRY_RETRIES:
9586         rval = ((mptsas_tgt_private_t *) (ap->a_hba_tran->
9587                                         tran_tgt_private))->t_private->m_qfull_retry_retries;
9588         break;
9589     case SCSI_CAP_QFULL_RETRY_INTERVAL:
9590         rval = drv_hztousec((mptsas_tgt_private_t *)
9591                               (ap->a_hba_tran->tran_tgt_private))->
9592                               t_private->m_qfull_retry_interval) / 1000;
9593         break;
9594     case SCSI_CAP_CDB_LEN:
9595         rval = CDB_GROUP4;
9596         break;
9597     case SCSI_CAP_INTERCONNECT_TYPE:
9598         rval = INTERCONNECT_SAS;
9599         break;
9600     case SCSI_CAP_TRAN_LAYER_RETRIES:
9601         if (mpt->m_ioc_capabilities &
9602             MPI2_IOCFACTS_CAPABILITY_TLR)
9603             rval = TRUE;
9604         else
9605             rval = FALSE;
9606         break;
9607     default:
9608         rval = UNDEFINED;
9609         break;
9610     }
9611
9612     NDBG24(("mptsas_scsi_getcap: %s, rval=%x", cap, rval));
9613
9614     mutex_exit(&mpt->m_mutex);
9615     return (rval);
9616 }
9617 */

```

```

9619 * (*tran_setcap). Set the capability named to the value given.
9620 */
9621 static int
9622 mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value, int tgtonly)
9623 {
9624     mptsas_t      *mpt = ADDR2MPT(ap);
9625     int          ckey;
9626     int          rval = FALSE;
9627
9628     NDBG24(("mptsas_scsi_setcap: target=%d, cap=%s value=%x tgtonly=%x",
9629             ap->a_target, cap, value, tgtonly));
9630
9631     if (!tgtonly) {
9632         return (rval);
9633     }
9634
9635     mutex_enter(&mpt->m_mutex);
9636
9637     if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9638         mutex_exit(&mpt->m_mutex);
9639         return (UNDEFINED);
9640     }
9641
9642     switch (ckey) {
9643     case SCSI_CAP_DMA_MAX:
9644     case SCSI_CAP_MSG_OUT:
9645     case SCSI_CAP_PARITY:
9646     case SCSI_CAP_INITIATOR_ID:
9647     case SCSI_CAP_LINKED_CMDS:
9648     case SCSI_CAP_UNTAGGED_QING:
9649     case SCSI_CAP_RESET_NOTIFICATION:
9650         /*
9651          * None of these are settable via
9652          * the capability interface.
9653         */
9654         break;
9655     case SCSI_CAP_ARQ:
9656         /*
9657          * We cannot turn off arq so return false if asked to
9658          */
9659         if (value) {
9660             rval = TRUE;
9661         } else {
9662             rval = FALSE;
9663         }
9664         break;
9665     case SCSI_CAP_TAGGED_QING:
9666         mptsas_set_throttle(mpt, ((mptsas_tgt_private_t *)
9667             (ap->a_hba_tran->tran_tgt_private))->t_private,
9668             MAX_THROTTLE);
9669         rval = TRUE;
9670         break;
9671     case SCSI_CAP_QFULL_RETRY_INTERVAL:
9672         ((mptsas_tgt_private_t *) (ap->a_hba_tran->tran_tgt_private))->
9673             t_private->m_qfull_retry_interval =
9674             drv_usecöhz(value * 1000);
9675         rval = TRUE;
9676         break;
9677     default:
9678         rval = UNDEFINED;
9679         break;
9680     }
9681
9682     mptsas_scsi_setcap(ap, cap, value, tgtonly);
9683
9684 }

```

```

9685     }
9686     mutex_exit(&mpt->m_mutex);
9687     return (rval);
9688 }
9689 */
9690 /* Utility routine for mptsas_ifsetcap/ifgetcap
9691 */
9692 /*ARGSUSED*/
9693 static int
9694 mptsas_scsi_capchk(char *cap, int tgtonly, int *cidxp)
9695 {
9696     NDBG24(("mptsas_scsi_capchk: cap=%s", cap));
9697
9698     if (!cap)
9699         return (FALSE);
9700
9701     *cidxp = scsi_hba_lookup_capstr(cap);
9702     return (TRUE);
9703 }
9704
9705 static int
9706 mptsas_alloc_active_slots(mptsas_t *mpt, int flag)
9707 {
9708     mptsas_slots_t *old_active = mpt->m_active;
9709     mptsas_slots_t *new_active;
9710     size_t           size;
9711
9712     /*
9713      * if there are active commands, then we cannot
9714      * change size of active slots array.
9715     */
9716     ASSERT(mpt->m_ncmds == 0);
9717
9718     size = MPTSAS_SLOTS_SIZE(mpt);
9719     new_active = kmalloc(size, flag);
9720     if (new_active == NULL) {
9721         NDBG1(("new active alloc failed"));
9722         return (-1);
9723     }
9724
9725     /*
9726      * Since SMID 0 is reserved and the TM slot is reserved, the
9727      * number of slots that can be used at any one time is
9728      * m_max_requests - 2.
9729     */
9730     new_active->m_n_normal = (mpt->m_max_requests - 2);
9731     new_active->m_size = size;
9732     new_active->m_rotor = 1;
9733     if (old_active)
9734         mptsas_free_active_slots(mpt);
9735     mpt->m_active = new_active;
9736
9737 }
9738
9739
9740 static void
9741 mptsas_free_active_slots(mptsas_t *mpt)
9742 {
9743     mptsas_slots_t *active = mpt->m_active;
9744     size_t           size;
9745
9746     if (active == NULL)
9747         return;
9748     size = active->m_size;
9749     kmem_free(active, size);
9750     mpt->m_active = NULL;

```

```

9751 }
9753 /*
9754 * Error logging, printing, and debug print routines.
9755 */
9756 static char *mptsas_label = "mpt_sas";
9758 /*PRINTFLIKE3*/
9759 void
9760 mptsas_log(mptsas_t *mpt, int level, char *fmt, ...)
9761 {
9762     dev_info_t      *dev;
9763     va_list         ap;
9765
9766     if (mpt) {
9767         dev = mpt->m_dip;
9768     } else {
9769         dev = 0;
9770     }
9771
9772     mutex_enter(&mptsas_log_mutex);
9773
9774     va_start(ap, fmt);
9775     (void) vsprintf(mptsas_log_buf, fmt, ap);
9776     va_end(ap);
9777
9778     if (level == CE_CONT) {
9779         scsi_log(dev, mptsas_label, level, "%s\n", mptsas_log_buf);
9780     } else {
9781         scsi_log(dev, mptsas_label, level, "%s", mptsas_log_buf);
9782     }
9783
9784     mutex_exit(&mptsas_log_mutex);
9785 }
9786 #ifdef MPTSAS_DEBUG
9787 /*
9788 * Use a circular buffer to log messages to private memory.
9789 * Increment idx atomically to minimize risk to miss lines.
9790 * It's fast and does not hold up the proceedings too much.
9791 */
9792 static const size_t mptsas_dbglog_linecnt = MPTSAS_DBGLOG_LINECNT;
9793 static const size_t mptsas_dbglog_linelen = MPTSAS_DBGLOG_LINELEN;
9794 static char mptsas_dbglog_bufs[MPTSAS_DBGLOG_LINECNT][MPTSAS_DBGLOG_LINELEN];
9795 static uint32_t mptsas_dbglog_idx = 0;
9796
9797 /*PRINTFLIKE1*/
9798 void
9799 mptsas_debug_log(char *fmt, ...)
9800 {
9801     va_list         ap;
9802     uint32_t        idx;
9803
9804     idx = atomic_inc_32_nv(&mptsas_dbglog_idx) &
9805           (mptsas_dbglog_linecnt - 1);
9806
9807     va_start(ap, fmt);
9808     (void) vsnprintf(mptsas_dbglog_bufs[idx],
9809                      mptsas_dbglog_linelen, fmt, ap);
9810     va_end(ap);
9811 }
9813 /*PRINTFLIKE1*/
9814 void
9815 mptsas_printf(char *fmt, ...)
9816 {

```

```

9817     dev_info_t      *dev = 0;
9818     va_list         ap;
9819
9820     mutex_enter(&mptsas_log_mutex);
9821
9822     va_start(ap, fmt);
9823     (void) vsprintf(mptsas_log_buf, fmt, ap);
9824     va_end(ap);
9825
9826 #ifdef PROM_PRINTF
9827     prom_printf("%s:\t%s\n", mptsas_label, mptsas_log_buf);
9828 #else
9829     scsi_log(dev, mptsas_label, CE_CONT, "!%s\n", mptsas_log_buf);
9830 #endif
9831     mutex_exit(&mptsas_log_mutex);
9832 }
9833#endif
9834
9835 /*
9836 * timeout handling
9837 */
9838 static void
9839 mptsas_watch(void *arg)
9840 {
9841 #ifndef __lock_lint
9842     _NOTE(ARGUNUSED(arg))
9843 #endif
9844
9845     mptsas_t      *mpt;
9846     uint32_t       doorbell;
9847
9848     NDBG30(("mptsas_watch"));
9849
9850     rw_enter(&mptsas_global_rwlock, RW_READER);
9851     for (mpt = mptsas_head; mpt != (mptsas_t *)NULL; mpt = mpt->m_next) {
9852
9853         mutex_enter(&mpt->m_mutex);
9854
9855         /* Skip device if not powered on */
9856         if (mpt->m_options & MPTSAS_OPT_PM) {
9857             if (mpt->m_power_level == PM_LEVEL_D0) {
9858                 (void) pm_busy_component(mpt->m_dip, 0);
9859                 mpt->m_busy = 1;
9860             } else {
9861                 mutex_exit(&mpt->m_mutex);
9862                 continue;
9863             }
9864         }
9865
9866         /*
9867         * Check if controller is in a FAULT state. If so, reset it.
9868         */
9869         doorbell = ddi_get32(mpt->m_datap, &mpt->m_reg->Doorbell);
9870         if ((doorbell & MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_FAULT) {
9871             doorbell &= MPI2_DOORBELL_DATA_MASK;
9872             mptsas_log(mpt, CE_WARN, "MPT Firmware Fault, "
9873                         "code: %04x", doorbell);
9874             mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
9875             if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
9876                 mptsas_log(mpt, CE_WARN, "Reset failed"
9877                             "after fault was detected");
9878             }
9879         }
9880
9881         /*
9882         * For now, always call mptsas_watchsubr.
9883

```

```

9883         */
9884         mptsas_watchsubr(mpt);
9885
9886         if (mpt->m_options & MPTSAS_OPT_PM) {
9887             mpt->m_busy = 0;
9888             (void) pm_idle_component(mpt->m_dip, 0);
9889         }
9890
9891         mutex_exit(&mpt->m_mutex);
9892     } rw_exit(&mptsas_global_rwlock);
9893
9894     mutex_enter(&mptsas_global_mutex);
9895     if (mptsas_timeouts_enabled)
9896         mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
9897     mutex_exit(&mptsas_global_mutex);
9898
9899 }
9900
9901 static void
9902 mptsas_watchsubr(mptsas_t *mpt)
9903 {
9904     int i;
9905     mptsas_cmd_t *cmd;
9906     mptsas_target_t *ptgt = NULL;
9907     hrtimer_t timestamp = gethrtime();
9908
9909     ASSERT(MUTEX_HELD(&mpt->m_mutex));
9910
9911     NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));
9912
9913 #ifdef MPTSAS_TEST
9914     if (mptsas_enable_untagged) {
9915         mptsas_test_untagged++;
9916     }
9917 #endif
9918
9919     /*
9920      * Check for commands stuck in active slot
9921      * Account for TM requests, which use the last SMID.
9922      */
9923     for (i = 0; i <= mpt->m_active->m_n_normal; i++) {
9924         if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9925             if (cmd->cmd_active_expiration <= timestamp) {
9926                 if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9927                     /*
9928                      * There seems to be a command stuck
9929                      * in the active slot. Drain throttle.
9930                      */
9931                     mptsas_set_throttle(mpt,
9932                         cmd->cmd_tgt_addr,
9933                         DRAIN_THROTTLE);
9934             } else if (cmd->cmd_flags &
9935                         (CFLAG_PASSTHRU | CFLAG_CONFIG |
9936                         CFLAG_FW_DIAG)) {
9937                 /*
9938                  * passthrough command timeout
9939                  */
9940                 cmd->cmd_flags |= (CFLAG_FINISHED |
9941                                 CFLAG_TIMEOUT);
9942                 cv_broadcast(&mpt->m_passthru_cv);
9943                 cv_broadcast(&mpt->m_config_cv);
9944                 cv_broadcast(&mpt->m_fw_diag_cv);
9945             }
9946         }
9947     }
9948 }

```

```

9950     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
9951         ptgt = refhash_next(mpt->m_targets, ptgt)) {
9952         /*
9953          * If we were draining due to a qfull condition,
9954          * go back to full throttle.
9955          */
9956         if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9957             (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9958             (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9959             mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9960             mptsas_restart_hba(mpt);
9961         }
9962
9963         cmd = TAILQ_LAST(&ptgt->m_active_cmdq, mptsas_active_cmdq);
9964         if (cmd == NULL)
9965             continue;
9966
9967         if (cmd->cmd_active_expiration <= timestamp) {
9968             /*
9969              * Earliest command timeout expired. Drain throttle.
9970              */
9971             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
9972
9973             /*
9974              * Check for remaining commands.
9975              */
9976             cmd = TAILQ_FIRST(&ptgt->m_active_cmdq);
9977             if (cmd->cmd_active_expiration > timestamp) {
9978                 /*
9979                  * Wait for remaining commands to complete or
9980                  * time out.
9981                  */
9982                 NDBG23(("command timed out, pending drain"));
9983                 continue;
9984             }
9985
9986             /*
9987              * All command timeouts expired.
9988              */
9989             mptsas_log(mpt, CE_NOTE, "Timeout of %d seconds "
9990                         "expired with %d commands on target %d lun %d.",
9991                         cmd->cmd_pkt->pkt_time, ptgt->m_t_ncmds,
9992                         ptgt->m_devhdl, Lun(cmd));
9993
9994             mptsas_cmd_timeout(mpt, ptgt);
9995         } else if (cmd->cmd_active_expiration <=
9996                     timestamp + (hrtimer_t)mptsas_scsi_watchdog_tick * NANOSEC) {
9997             NDBG23(("pending timeout"));
9998             mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
9999         }
10000     }
10001 }
10002
10003 /*
10004  * timeout recovery
10005  */
10006 static void
10007 mptsas_cmd_timeout(mptsas_t *mpt, mptsas_target_t *ptgt)
10008 {
10009     uint16_t devhdl;
10010     uint64_t sas_wwn;
10011     uint8_t phy;
10012     char wwn_str[MPTSAS_WWN_STRLEN];
10013
10014     devhdl = ptgt->m_devhdl;

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsa

137

```

10015     sas_wwn = pttgt->m_addr.mta_wwn;
10016     phy = pttgt->m_physnum;
10017     if (sas_wwn == 0) {
10018         (void) sprintf(wwn_str, "p%u", phy);
10019     } else {
10020         (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
10021     }
10022
10023 NDBG29(("mptsas_cmd_timeout: target=%d", devhdl));
10024 mptsas_log(mpt, CE_WARN, "Disconnected command timeout for "
10025     "target %d %s, enclosure %u", devhdl, wwn_str,
10026     pttgt->m_enclosure);
10027
10028 /* Abort all outstanding commands on the device.
10029 */
10030 NDBG29(("mptsas_cmd_timeout: device reset"));
10031 if (mptsas_do_scsi_reset(mpt, devhdl) != TRUE) {
10032     mptsas_log(mpt, CE_WARN, "Target %d reset for command timeout "
10033         "recovery failed!", devhdl);
10034 }
10035
10036 }
10037
10038 /* Device / Hotplug control
10039 */
10040 static int
10041 mptsas_scsi_quiesce(dev_info_t *dip)
10042 {
10043     mptsas_t           *mpt;
10044     scsi_hba_tran_t   *tran;
10045
10046     tran = ddi_get_driver_private(dip);
10047     if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
10048         return (-1);
10049
10050     return (mptsas_quiesce_bus(mpt));
10051 }
10052
10053 static int
10054 mptsas_scsi_unquiesce(dev_info_t *dip)
10055 {
10056     mptsas_t           *mpt;
10057     scsi_hba_tran_t   *tran;
10058
10059     tran = ddi_get_driver_private(dip);
10060     if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
10061         return (-1);
10062
10063     return (mptsas_unquiesce_bus(mpt));
10064 }
10065
10066 static int
10067 mptsas_quiesce_bus(mptsas_t *mpt)
10068 {
10069     mptsas_target_t *pttgt = NULL;
10070
10071     NDBG28(("mptsas_quiesce_bus"));
10072     mutex_enter(&mpt->m_mutex);
10073
10074     /* Set all the throttles to zero */
10075     for (pttgt = rehash_first(mpt->m_targets); pttgt != NULL;
10076         pttgt = rehash_next(mpt->m_targets, pttgt)) {
10077         mptsas_set_throttle(mpt, pttgt, HOLD_THROTTLE);
10078     }
10079 }
```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c 138

10081     /* If there are any outstanding commands in the queue */
10082     if (mpt->m_ncmds) {
10083         mpt->m_softstate |= MPTSAS_SS_DRAINING;
10084         mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10085                                         mpt, (MPTSAS QUIESCE_TIMEOUT * drv_usectohz(1000000)));
10086         if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
10087             /*
10088             * Quiesce has been interrupted
10089             */
10090             mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10091             for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10092                  ptgt = refhash_next(mpt->m_targets, ptgt)) {
10093                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10094             }
10095             mptsas_restart_hba(mpt);
10096             if (mpt->m_quiesce_timeid != 0) {
10097                 timeout_id_t tid = mpt->m_quiesce_timeid;
10098                 mpt->m_quiesce_timeid = 0;
10099                 mutex_exit(&mpt->m_mutex);
10100                 (void) untmeout(tid);
10101                 return (-1);
10102             }
10103             mutex_exit(&mpt->m_mutex);
10104             return (-1);
10105         } else {
10106             /* Bus has been quiesced */
10107             ASSERT(mpt->m_quiesce_timeid == 0);
10108             mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
10109             mpt->m_softstate |= MPTSAS_SS QUIESCED;
10110             mutex_exit(&mpt->m_mutex);
10111             return (0);
10112         }
10113     }
10114     /* Bus was not busy - QUIESCED */
10115     mutex_exit(&mpt->m_mutex);

10117     return (0);
10118 }

10120 static int
10121 mptsas_unquiesce_bus(mptsas_t *mpt)
10122 {
10123     mptsas_target_t *ptgt = NULL;

10125     NDBG28(("mptsas_unquiesce_bus"));
10126     mutex_enter(&mpt->m_mutex);
10127     mpt->m_softstate &= ~MPTSAS_SS QUIESCED;
10128     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10129          ptgt = refhash_next(mpt->m_targets, ptgt)) {
10130         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10131     }
10132     mptsas_restart_hba(mpt);
10133     mutex_exit(&mpt->m_mutex);
10134     return (0);
10135 }

10137 static void
10138 mptsas_ncmds_checkdrain(void *arg)
10139 {
10140     mptsas_t           *mpt = arg;
10141     mptsas_target_t   *ptgt = NULL;

10143     mutex_enter(&mpt->m_mutex);
10144     if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
10145         mpt->m_quiesce_timeid = 0;
10146         if (mpt->m_ncmds == 0) {

```

```

10147             /* Command queue has been drained */
10148             cv_signal(&mpt->m_cv);
10149         } else {
10150             /*
10151             * The throttle may have been reset because
10152             * of a SCSI bus reset
10153             */
10154             for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
10155                 ptgt = refhash_next(mpt->m_targets, ptgt)) {
10156                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10157             }
10158
10159             mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
10160                                             mpt, (MPTSAS QUIESCE TIMEOUT *
10161                                             drv_usectohz(1000000)));
10162
10163         }
10164         mutex_exit(&mpt->m_mutex);
10165     }
10166 /*ARGSUSED*/
10167 static void
10168 mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
10169 {
10170     int i;
10171     uint8_t *cp = (uchar_t *)cmd->cmd_pkt->pkt_cdbp;
10172     char buf[128];
10173
10174     buf[0] = '\0';
10175     NDBG25(("Cmd (0x%p) dump for Target %d Lun %d:\n", (void *)cmd,
10176             Tgt(cmd), Lun(cmd)));
10177     (void) sprintf(&buf[0], "\tcdb=[");
10178     for (i = 0; i < (int)cmd->cmd_cdblen; i++) {
10179         (void) sprintf(&buf[strlen(buf)], " 0x%x", *cp++);
10180     }
10181     (void) sprintf(&buf[strlen(buf)], " ]");
10182     NDBG25(("?%s\n", buf));
10183     NDBG25(("?pkt_flags=0x%x pkt_statistics=0x%x pkt_state=0x%x\n",
10184             cmd->cmd_pkt->pkt_flags, cmd->cmd_pkt->pkt_statistics,
10185             cmd->cmd_pkt->pkt_state));
10186     NDBG25(("?pkt_scbp=0x%x cmd_flags=0x%x\n", cmd->cmd_pkt->pkt_scbp ?
10187             *(cmd->cmd_pkt->pkt_scbp) : 0, cmd->cmd_flags));
10188
10189 }
10190
10191 static void
10192 mptsas_passthru_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10193                      pMpI2SGEsimple64_t sgep)
10194 {
10195     uint32_t          sge_flags;
10196     uint32_t          data_size, dataout_size;
10197     ddi_dma_cookie_t data_cookie;
10198     ddi_dma_cookie_t dataout_cookie;
10199
10200     data_size = pt->data_size;
10201     dataout_size = pt->dataout_size;
10202     data_cookie = pt->data_cookie;
10203     dataout_cookie = pt->dataout_cookie;
10204
10205     if (dataout_size) {
10206         sge_flags = dataout_size |
10207             ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10208             MPI2_SGE_FLAGS_END_OF_BUFFER |
10209             MPI2_SGE_FLAGS_HOST_TO_IOC |
10210             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10211             MPI2_SGE_FLAGS_SHIFT);
10212         ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);

```

```

10213             ddi_put32(acc_hdl, &sgep->Address.Low,
10214                     (uint32_t)(dataout_cookie.dmac_laddress &
10215                     0xfffffffffull));
10216             ddi_put32(acc_hdl, &sgep->Address.High,
10217                     (uint32_t)(dataout_cookie.dmac_laddress
10218                     >> 32));
10219             sgep++;
10220
10221             sge_flags = data_size;
10222             sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10223             MPI2_SGE_FLAGS_LAST_ELEMENT |
10224             MPI2_SGE_FLAGS_END_OF_BUFFER |
10225             MPI2_SGE_FLAGS_END_OF_LIST |
10226             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10227             MPI2_SGE_FLAGS_SHIFT);
10228             if (pt->direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10229                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
10230                             MPI2_SGE_FLAGS_SHIFT);
10231             } else {
10232                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
10233                             MPI2_SGE_FLAGS_SHIFT);
10234             }
10235             ddi_put32(acc_hdl, &sgep->FlagsLength,
10236                     sge_flags);
10237             ddi_put32(acc_hdl, &sgep->Address.Low,
10238                     (uint32_t)(data_cookie.dmac_laddress &
10239                     0xfffffffffull));
10240             ddi_put32(acc_hdl, &sgep->Address.High,
10241                     (uint32_t)(data_cookie.dmac_laddress >> 32));
10242
10243 static void
10244 mptsas_passthru_ieee_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10245                           pMpI2IeeeSgeSimple64_t ieeesgep)
10246 {
10247     uint8_t          sge_flags;
10248     uint32_t          data_size, dataout_size;
10249     ddi_dma_cookie_t data_cookie;
10250     ddi_dma_cookie_t dataout_cookie;
10251
10252     data_size = pt->data_size;
10253     dataout_size = pt->dataout_size;
10254     data_cookie = pt->data_cookie;
10255     dataout_cookie = pt->dataout_cookie;
10256
10257     sge_flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
10258                 MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
10259     if (dataout_size) {
10260         ddi_put32(acc_hdl, &ieeesgep->Length, dataout_size);
10261         ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10262                     (uint32_t)(dataout_cookie.dmac_laddress &
10263                     0xfffffffffull));
10264         ddi_put32(acc_hdl, &ieeesgep->Address.High,
10265                     (uint32_t)(dataout_cookie.dmac_laddress >> 32));
10266         ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
10267         ieeesgep++;
10268
10269     }
10270     sge_flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
10271     ddi_put32(acc_hdl, &ieeesgep->Length, data_size);
10272     ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10273                     (uint32_t)(data_cookie.dmac_laddress & 0xfffffffffull));
10274     ddi_put32(acc_hdl, &ieeesgep->Address.High,
10275                     (uint32_t)(data_cookie.dmac_laddress >> 32));
10276     ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
10277 }

```

```

10279 static void
10280 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
10281 {
10282     caddr_t         memp;
10283     pMPI2RequestHeader_t request_hdrp;
10284     struct scsi_pkt    *pkt = cmd->cmd_pkt;
10285     mptsas_pt_request_t *pt = pkt->pkt_ha_private;
10286     uint32_t          request_size;
10287     uint32_t          i;
10288     uint64_t          request_desc = 0;
10289     uint8_t           desc_type;
10290     uint16_t          SMID;
10291     uint8_t           *request_function;
10292     ddi_dma_handle_t  dma_hdl = mpt->m_dma_req_frame_hdl;
10293     ddi_acc_handle_t  acc_hdl = mpt->m_acc_req_frame_hdl;
10294
10295     desc_type = MPI2_REQ_DESCRIPTOR_FLAGS_DEFAULT_TYPE;
10296
10297     request = pt->request;
10298     request_size = pt->request_size;
10299
10300     SMID = cmd->cmd_slot;
10301
10302     /*
10303      * Store the passthrough message in memory location
10304      * corresponding to our slot number
10305      */
10306     memp = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
10307     request_hdrp = (pMPI2RequestHeader_t)memp;
10308     bzero(memp, mpt->m_req_frame_size);
10309
10310    for (i = 0; i < request_size; i++) {
10311        bcopy(request + i, memp + i, 1);
10312    }
10313
10314    NDBG15(("mptsas_start_passthru: Func 0x%x, MsgFlags 0x%x, "
10315            "size=%d, in %d, out %d, SMID %d", request_hdrp->Function,
10316            request_hdrp->MsgFlags, request_size,
10317            pt->data_size, pt->dataout_size, SMID));
10318
10319    /*
10320     * Add an SGE, even if the length is zero.
10321     */
10322    if (mpt->m_MPI25 && pt->simple == 0) {
10323        mptsas_passthru_ieee_sge(acc_hdl, pt,
10324            (pMpI2IeeeSgeSimple64_t)
10325            ((uint8_t *)request_hdrp + pt->sgl_offset));
10326    } else {
10327        mptsas_passthru_sge(acc_hdl, pt,
10328            (pMpI2SGESimple64_t)
10329            ((uint8_t *)request_hdrp + pt->sgl_offset));
10330    }
10331
10332    function = request_hdrp->Function;
10333    if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10334        (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10335        pMpI2SCSIIORequest_t scsi_io_req;
10336        caddr_t             arsbuf;
10337        uint8_t             ars_size;
10338        uint32_t            ars_dmaaddrlow;
10339
10340        NDBG15(("mptsas_start_passthru: Is SCSI IO Req"));
10341        scsi_io_req = (pMpI2SCSIIORequest_t)request_hdrp;
10342
10343        if (cmd->cmd_extrqslen != 0) {
10344            /*

```

```

10345                                         * Mapping of the buffer was done in
10346                                         * mptsas_do_passthru().
10347                                         * Calculate the DMA address with the same offset.
10348                                         */
10349                                         arsbuf = cmd->cmd_arq_buf;
10350                                         ars_size = cmd->cmd_extrqslen;
10351                                         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
10352                                         ((uintptr_t)arsbuf - (uintptr_t)mpt->m_req_sense)) &
10353                                         0xffffffffffu;
10354
10355     } else {
10356         arsbuf = mpt->m_req_sense +
10357             (mpt->m_req_sense_size * (SMID-1));
10358         cmd->cmd_arq_buf = arsbuf;
10359         ars_size = mpt->m_req_sense_size;
10360         ars_dmaaddrlow = (mpt->m_req_sense_dma_addr +
10361             (mpt->m_req_sense_size * (SMID-1))) &
10362             0xffffffffffu;
10363
10364         bzero(arsbuf, ars_size);
10365
10366         ddi_put8(acc_hdl, &scsi_io_req->SenseBufferLength, ars_size);
10367         ddi_put32(acc_hdl, &scsi_io_req->SenseBufferLowAddress,
10368             ars_dmaaddrlow);
10369
10370         /*
10371          * Put SGE for data and data_out buffer at the end of
10372          * scsi_io_request message header.(64 bytes in total)
10373          * Set SGLOffset0 value
10374          */
10375         ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
10376             offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);
10377
10378         /*
10379          * Setup descriptor info. RAID passthrough must use the
10380          * default request descriptor which is already set, so if this
10381          * is a SCSI IO request, change the descriptor to SCSI IO.
10382          */
10383         if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
10384             desc_type = MPI2_REQ_DESCRIPTOR_FLAGS_SCSI_IO;
10385             request_desc = ((uint64_t)ddi_get16(acc_hdl,
10386                 &scsi_io_req->DevHandle) << 48);
10387
10388             (void) ddi_dma_sync(mpt->m_dma_req_sense_hdl, 0, 0,
10389                 DDI_DMA_SYNC_FORDEV);
10390
10391         /*
10392          * We must wait till the message has been completed before
10393          * beginning the next message so we wait for this one to
10394          * finish.
10395          */
10396         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
10397         request_desc |= (SMID << 16) + desc_type;
10398         cmd->cmd_rfim = NULL;
10399         MPTAS_START_CMD(mpt, request_desc);
10400
10401         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
10402             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
10403             ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10404         }
10405
10406     typedef void (mptsas_pre_f)(mptsas_t *, mptsas_pt_request_t *);
10407     static mptsas_pre_f    mpi_pre_ioc_facts;
10408     static mptsas_pre_f    mpi_pre_port_facts;
10409     static mptsas_pre_f    mpi_pre_fw_download;
10410     static mptsas_pre_f    mpi_pre_fw_25_download;

```

```

10411 static mptsas_pre_f    mpi_pre_fw_upload;
10412 static mptsas_pre_f    mpi_pre_fw_25_upload;
10413 static mptsas_pre_f    mpi_pre_sata_passthrough;
10414 static mptsas_pre_f    mpi_pre_smp_passthrough;
10415 static mptsas_pre_f    mpi_pre_config;
10416 static mptsas_pre_f    mpi_pre_sas_io_unit_control;
10417 static mptsas_pre_f    mpi_pre_scsi_io_req;

10419 /*
10420  * Prepare the pt for a SAS2 FW_DOWNLOAD request.
10421 */
10422 static void
10423 mpi_pre_fw_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10424 {
10425     pMpi2FWDownloadTCSGE_t tcsge;
10426     pMpi2FWDownloadRequest req;

10428     /*
10429      * If SAS3, call separate function.
10430      */
10431     if (mpt->m_MPI25) {
10432         mpi_pre_fw_25_download(mpt, pt);
10433         return;
10434     }

10436     /*
10437      * User requests should come in with the Transaction
10438      * context element where the SGL will go. Putting the
10439      * SGL after that seems to work, but don't really know
10440      * why. Other drivers tend to create an extra SGL and
10441      * refer to the TCE through that.
10442      */
10443     req = (pMpi2FWDownloadRequest)pt->request;
10444     tcsge = (pMpi2FWDownloadTCSGE_t)&req->SGL;
10445     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10446         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10447         mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10448     }

10450     pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL) +
10451         sizeof (*tcsge);
10452     if (pt->request_size != pt->sgl_offset)
10453         NDBG15(("mpi_pre_fw_download(): Incorrect req size, "
10454             "0x%lx, should be 0x%lx, dataoutsz 0x%lx",
10455             (int)pt->request_size, (int)pt->sgl_offset,
10456             (int)pt->dataout_size));
10457     if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10458         NDBG15(("mpi_pre_fw_download(): Incorrect rep size, "
10459             "0x%lx, should be 0x%lx", pt->data_size,
10460             (int)sizeof (MPI2_FW_DOWNLOAD_REPLY)));
10461 }

10463 /*
10464  * Prepare the pt for a SAS3 FW_DOWNLOAD request.
10465 */
10466 static void
10467 mpi_pre_fw_25_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10468 {
10469     pMpi2FWDownloadTCSGE_t tcsge;
10470     pMpi2FWDownloadRequest req2;
10471     pMpi25FWDownloadRequest req25;

10473     /*
10474      * User requests should come in with the Transaction
10475      * context element where the SGL will go. The new firmware
10476      * Doesn't use TCE and has space in the main request for

```

```

10477             * this information. So move to the right place.
10478             */
10479     req2 = (pMpi2FWDownloadRequest)pt->request;
10480     req25 = (pMpi25FWDownloadRequest)pt->request;
10481     tcsge = (pMpi2FWDownloadTCSGE_t)&req2->SGL;
10482     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10483         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10484         mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10485     }
10486     req25->ImageOffset = tcsge->ImageOffset;
10487     req25->ImageSize = tcsge->ImageSize;

10489     pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL);
10490     if (pt->request_size != pt->sgl_offset)
10491         NDBG15(("mpi_pre_fw_25_download(): Incorrect req size, "
10492             "0x%lx, should be 0x%lx, dataoutsz 0x%lx",
10493             pt->request_size, pt->sgl_offset,
10494             pt->dataout_size));
10495     if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10496         NDBG15(("mpi_pre_fw_25_download(): Incorrect rep size, "
10497             "0x%lx, should be 0x%lx", pt->data_size,
10498             (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10499 }

10501 /*
10502  * Prepare the pt for a SAS2 FW_UPLOAD request.
10503 */
10504 static void
10505 mpi_pre_fw_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10506 {
10507     pMpi2FWUploadTCSGE_t tcsge;
10508     pMpi2FWUploadRequest_t req;

10510     /*
10511      * If SAS3, call separate function.
10512      */
10513     if (mpt->m_MPI25) {
10514         mpi_pre_fw_25_upload(mpt, pt);
10515         return;
10516     }

10518     /*
10519      * User requests should come in with the Transaction
10520      * context element where the SGL will go. Putting the
10521      * SGL after that seems to work, but don't really know
10522      * why. Other drivers tend to create an extra SGL and
10523      * refer to the TCE through that.
10524      */
10525     req = (pMpi2FWUploadRequest_t)pt->request;
10526     tcsge = (pMpi2FWUploadTCSGE_t)&req->SGL;
10527     if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10528         tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10529         mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10530     }

10532     pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL) +
10533         sizeof (*tcsge);
10534     if (pt->request_size != pt->sgl_offset)
10535         NDBG15(("mpi_pre_fw_upload(): Incorrect req size, "
10536             "0x%lx, should be 0x%lx, dataoutsz 0x%lx",
10537             pt->request_size, pt->sgl_offset,
10538             pt->dataout_size));
10539     if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10540         NDBG15(("mpi_pre_fw_upload(): Incorrect rep size, "
10541             "0x%lx, should be 0x%lx", pt->data_size,
10542             (int)sizeof (MPI2_FW_UPLOAD_REPLY)));

```

```

10543 }
10545 /* Prepare the pt a SAS3 FW_UPLOAD request.
10546 */
10548 static void
10549 mpi_pre_fw_25_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10550 {
10551     pMpi2FWUploadTCSGE_t tcsge;
10552     pMpi2FWUploadRequest_t req2;
10553     pMpi25FWUploadRequest_t req25;
10555 /*
10556     * User requests should come in with the Transaction
10557     * context element where the SGL will go. The new firmware
10558     * Doesn't use TCE and has space in the main request for
10559     * this information. So move to the right place.
10560 */
10561 req2 = (pMpi2FWUploadRequest_t)pt->request;
10562 req25 = (pMpi25FWUploadRequest_t)pt->request;
10563 tcsge = (pMpi2FWUploadTCSGE_t)&req2->SGL;
10564 if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10565     tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10566     mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10567 }
10568 req25->ImageOffset = tcsge->ImageOffset;
10569 req25->ImageSize = tcsge->ImageSize;
10571 pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL);
10572 if (pt->request_size != pt->sgl_offset)
10573     NDBG15(("mpi_pre_fw_25_upload(): Incorrect req size, "
10574             "0x%x, should be 0x%x, dataoutsz 0x%x",
10575             pt->request_size, pt->sgl_offset,
10576             pt->dataout_size));
10577 if (pt->data_size < sizeof(MPI2_FW_UPLOAD_REPLY))
10578     NDBG15(("mpi_pre_fw_25_upload(): Incorrect rep size, "
10579             "0x%x, should be 0x%x", pt->data_size,
10580             (int)sizeof(MPI2_FW_UPLOAD_REPLY)));
10581 }

10583 /*
10584     * Prepare the pt for an IOC_FACTS request.
10585 */
10586 static void
10587 mpi_pre_IOC_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10588 {
10589 #ifndef __lock_lint
10590     _NOTE(ARGUNUSED(mpt))
10591 #endif
10592     if (pt->request_size != sizeof(MPI2_IOC_FACTS_REQUEST))
10593         NDBG15(("mpi_pre_IOC_facts(): Incorrect req size, "
10594                 "0x%x, should be 0x%x, dataoutsz 0x%x",
10595                 pt->request_size,
10596                 (int)sizeof(MPI2_IOC_FACTS_REQUEST),
10597                 pt->dataout_size));
10598     if (pt->data_size != sizeof(MPI2_IOC_FACTS_REPLY))
10599         NDBG15(("mpi_pre_IOC_facts(): Incorrect rep size, "
10600                 "0x%x, should be 0x%x", pt->data_size,
10601                 (int)sizeof(MPI2_IOC_FACTS_REPLY)));
10602     pt->sgl_offset = (uint16_t)pt->request_size;
10603 }

10605 /*
10606     * Prepare the pt for a PORT_FACTS request.
10607 */
10608 static void

```

```

10609 mpi_pre_port_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10610 {
10611 #ifndef __lock_lint
10612     _NOTE(ARGUNUSED(mpt))
10613 #endif
10614     if (pt->request_size != sizeof(MPI2_PORT_FACTS_REQUEST))
10615         NDBG15(("mpi_pre_port_facts(): Incorrect req size, "
10616                 "0x%x, should be 0x%x, dataoutsz 0x%x",
10617                 pt->request_size,
10618                 (int)sizeof(MPI2_PORT_FACTS_REQUEST),
10619                 pt->dataout_size));
10620     if (pt->data_size != sizeof(MPI2_PORT_FACTS_REPLY))
10621         NDBG15(("mpi_pre_port_facts(): Incorrect rep size, "
10622                 "0x%x, should be 0x%x", pt->data_size,
10623                 (int)sizeof(MPI2_PORT_FACTS_REPLY)));
10624     pt->sgl_offset = (uint16_t)pt->request_size;
10625 }

10627 /*
10628     * Prepare pt for a SATA_PASSTHROUGH request.
10629 */
10630 static void
10631 mpi_pre_sata_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10632 {
10633 #ifndef __lock_lint
10634     _NOTE(ARGUNUSED(mpt))
10635 #endif
10636     pt->sgl_offset = offsetof(MPI2_SATA_PASSTHROUGH_REQUEST, SGL);
10637     if (pt->request_size != pt->sgl_offset)
10638         NDBG15(("mpi_pre_sata_passthrough(): Incorrect req size, "
10639                 "0x%x, should be 0x%x, dataoutsz 0x%x",
10640                 pt->request_size, pt->sgl_offset,
10641                 pt->dataout_size));
10642     if (pt->data_size != sizeof(MPI2_SATA_PASSTHROUGH_REPLY))
10643         NDBG15(("mpi_pre_sata_passthrough(): Incorrect rep size, "
10644                 "0x%x, should be 0x%x", pt->data_size,
10645                 (int)sizeof(MPI2_SATA_PASSTHROUGH_REPLY)));
10646 }

10648 static void
10649 mpi_pre_smp_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10650 {
10651 #ifndef __lock_lint
10652     _NOTE(ARGUNUSED(mpt))
10653 #endif
10654     pt->sgl_offset = offsetof(MPI2_SMP_PASSTHROUGH_REQUEST, SGL);
10655     if (pt->request_size != pt->sgl_offset)
10656         NDBG15(("mpi_pre_smp_passthrough(): Incorrect req size, "
10657                 "0x%x, should be 0x%x, dataoutsz 0x%x",
10658                 pt->request_size, pt->sgl_offset,
10659                 pt->dataout_size));
10660     if (pt->data_size != sizeof(MPI2_SMP_PASSTHROUGH_REPLY))
10661         NDBG15(("mpi_pre_smp_passthrough(): Incorrect rep size, "
10662                 "0x%x, should be 0x%x", pt->data_size,
10663                 (int)sizeof(MPI2_SMP_PASSTHROUGH_REPLY)));
10664 }

10666 /*
10667     * Prepare pt for a CONFIG request.
10668 */
10669 static void
10670 mpi_pre_config(mptsas_t *mpt, mptsas_pt_request_t *pt)
10671 {
10672 #ifndef __lock_lint
10673     _NOTE(ARGUNUSED(mpt))
10674 #endif

```

```

10675     pt->sgl_offset = offsetof(MPI2_CONFIG_REQUEST, PageBufferSGE);
10676     if (pt->request_size != pt->sgl_offset)
10677         NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10678                 "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10679                 pt->sgl_offset, pt->dataout_size));
10680     if (pt->data_size != sizeof(MPI2_CONFIG_REPLY))
10681         NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10682                 "should be 0x%x", pt->data_size,
10683                 (int)sizeof(MPI2_CONFIG_REPLY)));
10684     pt->simple = 1;
10685 }

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

10705 /* Prepare the mptsas_cmd for a SAS_IO_UNIT_CONTROL request.
10706 */
10707 static void
10708 mpi_pre_sas_io_unit_control(mptsas_t *mpt, mptsas_pt_request_t *pt)
10709 {
10710 #ifndef __lock_lint
10711     __NOTE(ARGUNUSED(mpt))
10712 #endif
10713     pt->sgl_offset = (uint16_t)pt->request_size;
10714 }

10715 /* A set of functions to prepare an mptsas_cmd for the various
10716 supported requests.
10717 */
10718 static struct mptsas_func {
10719     U8             Function;
10720     char           *Name;
10721     mptsas_pre_f   *f_pre;
10722 } mptsas_func_list[] = {
10723     { MPI2_FUNCTION_IOC_FACTS, "IOC_FACTS",      mpi_pre_ioc_facts },
10724     { MPI2_FUNCTION_PORT_FACTS, "PORT_FACTS",    mpi_pre_port_facts },
10725     { MPI2_FUNCTION_FW_DOWNLOAD, "FW_DOWNLOAD",   mpi_pre_fw_download },
10726     { MPI2_FUNCTION_FW_UPLOAD, "FW_UPLOAD",      mpi_pre_fw_upload },
10727     { MPI2_FUNCTION_SATA_PASSTHROUGH, "SATA_PASSTHROUGH",
10728         mpi_pre_sata_passthrough },
10729     { MPI2_FUNCTION_SMP_PASSTHROUGH, "SMP_PASSTHROUGH",
10730         mpi_pre_smp_passthrough },
10731     { MPI2_FUNCTION_SCSI_IO_REQUEST, "SCSI_IO_REQUEST",
10732         mpi_pre_scsi_io_req },
10733     { MPI2_FUNCTION_CONFIG, "CONFIG",            mpi_pre_config },
10734     { MPI2_FUNCTION_SAS_IO_UNIT_CONTROL, "SAS_IO_UNIT_CONTROL",
10735         mpi_pre_sas_io_unit_control }
10736 };

```

```

10741             mpi_pre_sas_io_unit_control },
10742             { 0xFF, NULL,                                     NULL } /* list end */
10743 };

10744 static void
10745 mptsas_prep_sgl_offset(mptsas_t *mpt, mptsas_pt_request_t *pt)
10746 {
10747     pMPI2RequestHeader_t     hdr;
10748     struct mptsas_func       *f;
10749

10750     hdr = (pMPI2RequestHeader_t)pt->request;

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

10765 static int
10766 mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
10767                      uint8_t *data, uint32_t request_size, uint32_t reply_size,
10768                      uint32_t data_size, uint32_t direction, uint8_t *dataout,
10769                      uint32_t dataout_size, short timeout, int mode)
10770 {
10771     mptsas_pt_request_t          pt;
10772     mptsas_dma_alloc_state_t    data_dma_state;
10773     mptsas_dma_alloc_state_t    dataout_dma_state;
10774     caddr_t                      memp;
10775     mptsas_cmd_t                *cmd = NULL;
10776     struct scsi_pkt              *pkt;
10777     uint32_t                     reply_len = 0, sense_len = 0;
10778     pMPI2RequestHeader_t         request_hdrp;
10779     pMPI2DefaultReply_t          request_msg;
10780     pMPI2DefaultReply_t          reply_msg;
10781     Mpi2SCSIIOReply_t           rep_msg;
10782     int                          rvalue;
10783     int                          i, status = 0, pt_flags = 0, rv = 0;
10784     uint8_t                      function;

10785     ASSERT(mutex_owned(&mpt->m_mutex));

10786     reply_msg = (pMPI2DefaultReply_t)(&rep_msg);
10787     bzero(reply_msg, sizeof(MPI2_DEFAULT_REPLY));
10788     request_msg = kmalloc(request_size, KM_SLEEP);

10789     mutex_exit(&mpt->m_mutex);
10790     /*
10791      * copy in the request buffer since it could be used by
10792      * another thread when the pt request into waitq
10793      */
10794     if (ddi_copyin(request, request_msg, request_size, mode)) {
10795         mutex_enter(&mpt->m_mutex);
10796         status = EFAULT;
10797         mptsas_log(mpt, CE_WARN, "failed to copy request data");
10798         goto out;
10799     }
10800 }

10801     if (ddi_copyin(request, request_msg, request_size, mode)) {
10802         mutex_enter(&mpt->m_mutex);
10803         status = EFAULT;
10804         mptsas_log(mpt, CE_WARN, "failed to copy request data");
10805         goto out;
10806     }

```

```

10807     NDBG27(("mptsas_do_passthru: mode 0x%x, size 0x%x, Func 0x%x",
10808         mode, request_size, request_msg->Function));
10809     mutex_enter(&mpt->m_mutex);

10811     function = request_msg->Function;
10812     if (function == MPI2_FUNCTION_SCSI_TASK_MGMT) {
10813         pMpI2SCSITaskManagementRequest_t task;
10814         task = (pMpI2SCSITaskManagementRequest_t)request_msg;
10815         mptsas_setup_bus_reset_delay(mpt);
10816         rv = mptsas_ioc_task_management(mpt, task->TaskType,
10817             task->DevHandle, (int)task->LUN[1], reply, reply_size,
10818             mode);

10820         if (rv != TRUE) {
10821             status = EIO;
10822             mptsas_log(mpt, CE_WARN, "task management failed");
10823         }
10824         goto out;
10825     }

10827     if (data_size != 0) {
10828         data_dma_state.size = data_size;
10829         if (mptsas_dma_alloc(mpt, &data_dma_state) != DDI_SUCCESS) {
10830             status = ENOMEM;
10831             mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10832                 "resource");
10833             goto out;
10834         }
10835         pt_flags |= MPTSAS_DATA_ALLOCATED;
10836         if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10837             mutex_exit(&mpt->m_mutex);
10838             for (i = 0; i < data_size; i++) {
10839                 if (ddi_copyin(data + i, (uint8_t *)data_dma_state.memp + i, 1, mode)) {
10840                     mutex_enter(&mpt->m_mutex);
10841                     status = EFAULT;
10842                     mptsas_log(mpt, CE_WARN, "failed to "
10843                         "copy read data");
10844                     goto out;
10845                 }
10846             }
10847             mutex_enter(&mpt->m_mutex);
10848         }
10849     } else {
10850         bzero(&data_dma_state, sizeof (data_dma_state));
10851     }

10854     if (dataout_size != 0) {
10855         dataout_dma_state.size = dataout_size;
10856         if (mptsas_dma_alloc(mpt, &dataout_dma_state) != DDI_SUCCESS) {
10857             status = ENOMEM;
10858             mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10859                 "resource");
10860             goto out;
10861         }
10862         pt_flags |= MPTSAS_DATAOUT_ALLOCATED;
10863         mutex_exit(&mpt->m_mutex);
10864         for (i = 0; i < dataout_size; i++) {
10865             if (ddi_copyin(dataout + i, (uint8_t *)dataout_dma_state.memp + i, 1, mode)) {
10866                 mutex_enter(&mpt->m_mutex);
10867                 mptsas_log(mpt, CE_WARN, "failed to copy out"
10868                     " data");
10869                 status = EFAULT;
10870                 goto out;
10871             }
10872         }

```

```

10873         }
10874         mutex_enter(&mpt->m_mutex);
10875     } else {
10876         bzero(&dataout_dma_state, sizeof (dataout_dma_state));
10877     }

10878     if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
10879         status = EAGAIN;
10880         mptsas_log(mpt, CE_NOTE, "event ack command pool is full");
10881         goto out;
10882     }
10883     pt_flags |= MPTSAS_REQUEST_POOL_CMD;
10884

10885     bzero((caddr_t)cmd, sizeof (*cmd));
10886     bzero((caddr_t)pkt, scsi_pkt_size());
10887     bzero((caddr_t)&pt, sizeof (pt));

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

10889     pt.request = (uint8_t *)request_msg;
10890     pt.direction = direction;
10891     pt.simple = 0;
10892     pt.request_size = request_size;
10893     pt.data_size = data_size;
10894     pt.dataout_size = dataout_size;
10895     pt.data_cookie = data_dma_state.cookie;
10896     pt.dataout_cookie = dataout_dma_state.cookie;
10897     mptsas_prep_sgl_offset(mpt, &pt);

10898     /*
10899      * Form a blank cmd/pkt to store the acknowledgement message
10900      */
10901     pkt->pkt_cdbp = (opaque_t)&cmd->cmd_cdb[0];
10902     pkt->pkt_scbp = (opaque_t)&cmd->cmd_scb;
10903     pkt->pkt_ha_private = (opaque_t)&pt;
10904     pkt->pkt_flags = FLAG_HEAD;
10905     pkt->pkt_time = timeout;
10906     cmd->cmd_pkt = pkt;
10907     cmd->cmd_flags = CFLAG_CMDIOC | CFLAG_PASSTHRU;

10908     if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10909         (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10910         uint8_t com, cdb_group_id;
10911         boolean_t ret;

10912         pkt->pkt_cdbp = ((pMpI2SCSIIOResponse_t)request_msg)->CDB.CDB32;
10913         com = pkt->pkt_cdb[0];
10914         cdb_group_id = CDB_GROUPID(com);
10915         switch (cdb_group_id) {
10916             case CDB_GROUPID_0: cmd->cmd_cdblen = CDB_GROUP0; break;
10917             case CDB_GROUPID_1: cmd->cmd_cdblen = CDB_GROUP1; break;
10918             case CDB_GROUPID_2: cmd->cmd_cdblen = CDB_GROUP2; break;
10919             case CDB_GROUPID_4: cmd->cmd_cdblen = CDB_GROUP4; break;
10920             case CDB_GROUPID_5: cmd->cmd_cdblen = CDB_GROUP5; break;
10921             default:
10922                 NDBG27(("mptsas_do_passthru: SCSI_IO, reserved "
10923                     "CDBGROUP 0x%x requested!", cdb_group_id));
10924                 break;
10925         }
10926         reply_len = sizeof (MPI2_SCSI_IO_REPLY);
10927         sense_len = reply_size - reply_len;
10928         ret = mptsas_cmddarqsize(mpt, cmd, sense_len, KM_SLEEP);
10929         VERIFY(ret == B_TRUE);
10930     } else {
10931         reply_len = reply_size;
10932     }

```

```

10939         sense_len = 0;
10940     }
10941
10942     NDBG27(("mptsas_do_passthru: %s, dsz 0x%x, dosz 0x%x, replen 0x%x, "
10943             "snslen 0x%lx",
10944             (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE)?"Write":"Read",
10945             data_size, dataout_size, reply_len, sense_len));
10946
10947     /*
10948      * Save the command in a slot
10949      */
10950     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
10951         /*
10952          * Once passthru command get slot, set cmd_flags
10953          * CFLAG_PREPARED.
10954          */
10955         cmd->cmd_flags |= CFLAG_PREPARED;
10956         mptsas_start_passthru(mpt, cmd);
10957     } else {
10958         mptsas_waitq_add(mpt, cmd);
10959     }
10960
10961     while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
10962         cv_wait(&mpt->m_passthru_cv, &mpt->m_mutex);
10963     }
10964
10965     NDBG27(("mptsas_do_passthru: Cmd complete, flags 0x%lx, rfm 0x%lx "
10966             "pktreason 0x%lx", cmd->cmd_flags, cmd->cmd_rfm,
10967             pkt->pkt_reason));
10968
10969     if (cmd->cmd_flags & CFLAG_PREPARED) {
10970         memp = mpt->m_req_frame + (mpt->m_req_frame_size *
10971             cmd->cmd_slot);
10972         request_hdrv = (pMPI2RequestHeader_t)memp;
10973     }
10974
10975     if (cmd->cmd_flags & CFLAG_TIMEOUT) {
10976         status = ETIMEDOUT;
10977         mptsas_log(mpt, CE_WARN, "passthrough command timeout");
10978         pt_flags |= MPTSAS_CMD_TIMEOUT;
10979         goto out;
10980     }
10981
10982     if (cmd->cmd_rfm) {
10983         /*
10984          * cmd_rfm is zero means the command reply is a CONTEXT
10985          * reply and no PCI Write to post the free reply SMFA
10986          * because no reply message frame is used.
10987          * cmd_rfm is non-zero means the reply is a ADDRESS
10988          * reply and reply message frame is used.
10989          */
10990     pt_flags |= MPTSAS_ADDRESS_REPLY;
10991     (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
10992         DDI_DMA_SYNC_FORCPU);
10993     reply_msg = (pMPI2DefaultReply_t)
10994         (mpt->m_reply_frame + (cmd->cmd_rfm -
10995             (mpt->m_reply_frame_dma_addr & 0xfffffffffu)));
10996 }
10997
10998 mptsas_fma_check(mpt, cmd);
10999 if (pkt->pkt_reason == CMD_TRAN_ERR) {
11000     status = EAGAIN;
11001     mptsas_log(mpt, CE_WARN, "passthru fma error");
11002     goto out;
11003 }
11004 if (pkt->pkt_reason == CMD_RESET) {

```

```

11005         status = EAGAIN;
11006         mptsas_log(mpt, CE_WARN, "ioc reset abort passthru");
11007         goto out;
11008     }
11009
11010     if (pkt->pkt_reason == CMD_INCOMPLETE) {
11011         status = EIO;
11012         mptsas_log(mpt, CE_WARN, "passthrough command incomplete");
11013         goto out;
11014     }
11015
11016     mutex_exit(&mpt->m_mutex);
11017     if (cmd->cmd_flags & CFLAG_PREPARED) {
11018         function = request_hdrv->Function;
11019         if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
11020             (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
11021             reply_len = sizeof(MPI2_SCSI_IO_REPLY);
11022             sense_len = cmd->cmd_extrqslen ?
11023                 min(sense_len, cmd->cmd_extrqslen) :
11024                 min(sense_len, cmd->cmd_rqslen);
11025         } else {
11026             reply_len = reply_size;
11027             sense_len = 0;
11028         }
11029
11030         for (i = 0; i < reply_len; i++) {
11031             if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
11032                             mode)) {
11033                 mutex_enter(&mpt->m_mutex);
11034                 status = EFAULT;
11035                 mptsas_log(mpt, CE_WARN, "failed to copy out "
11036                             "reply data");
11037                 goto out;
11038             }
11039         }
11040         for (i = 0; i < sense_len; i++) {
11041             if (ddi_copyout((uint8_t *)request_hdrv + 64 + i,
11042                             reply + reply_len + i, 1, mode)) {
11043                 mutex_enter(&mpt->m_mutex);
11044                 status = EFAULT;
11045                 mptsas_log(mpt, CE_WARN, "failed to copy out "
11046                             "sense data");
11047                 goto out;
11048             }
11049         }
11050     }
11051
11052     if (data_size) {
11053         if (direction != MPTSAS_PASS_THRU_DIRECTION_WRITE) {
11054             (void) ddi_dma_sync(data_dma_state.handle, 0, 0,
11055                 DDI_DMA_SYNC_FORCPU);
11056             for (i = 0; i < data_size; i++) {
11057                 if (ddi_copyout((uint8_t *)(
11058                     data_dma_state.memp + i), data + i, 1,
11059                     mode)) {
11060                     mutex_enter(&mpt->m_mutex);
11061                     status = EFAULT;
11062                     mptsas_log(mpt, CE_WARN, "failed to "
11063                         "copy out the reply data");
11064                     goto out;
11065                 }
11066             }
11067         }
11068     }
11069     mutex_enter(&mpt->m_mutex);
11070 out:

```

```

11071     /*
11072      * Put the reply frame back on the free queue, increment the free
11073      * index, and write the new index to the free index register. But only
11074      * if this reply is an ADDRESS reply.
11075      */
11076     if (pt_flags & MPTSAS_ADDRESS_REPLY) {
11077         ddi_put32(mpt->m_acc_free_queue_hdl,
11078             &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
11079                 cmd->cmd_rfim);
11080         (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11081             DDI_DMA_SYNC_FORDEV);
11082         if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11083             mpt->m_free_index = 0;
11084         }
11085         ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11086             mpt->m_free_index);
11087     }
11088     if (cmd) {
11089         if (cmd->cmd_extrqslen != 0) {
11090             rmfree(mpt->m_ersense_map, cmd->cmd_extrqschunks,
11091                 cmd->cmd_extrqsidx + 1);
11092         }
11093         if (cmd->cmd_flags & CFLAG_PREPARED) {
11094             mptsas_remove_cmd(mpt, cmd);
11095             pt_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11096         }
11097     }
11098     if (pt_flags & MPTSAS_REQUEST_POOL_CMD)
11099         mptsas_return_to_pool(mpt, cmd);
11100     if (pt_flags & MPTSAS_DATA_ALLOCATED) {
11101         if (mptsas_check_dma_handle(data_dma_state.handle) !=
11102             DDI_SUCCESS) {
11103             ddi_fm_service_impact(mpt->m_dip,
11104                 DDI_SERVICE_UNAFFECTED);
11105             status = EFAULT;
11106         }
11107         mptsas_dma_free(&data_dma_state);
11108     }
11109     if (pt_flags & MPTSAS_DATAOUT_ALLOCATED) {
11110         if (mptsas_check_dma_handle(dataout_dma_state.handle) !=
11111             DDI_SUCCESS) {
11112             ddi_fm_service_impact(mpt->m_dip,
11113                 DDI_SERVICE_UNAFFECTED);
11114             status = EFAULT;
11115         }
11116         mptsas_dma_free(&dataout_dma_state);
11117     }
11118     if (pt_flags & MPTSAS_CMD_TIMEOUT) {
11119         if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
11120             mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
11121         }
11122     }
11123     if (request_msg)
11124         kmem_free(request_msg, request_size);
11125     NDBG27(("mptsas_do_passthru: Done status 0x%x", status));
11126
11127     return (status);
11128 }

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

```

```

11137             data->Timeout = MPTSAS_PASS_THRU_TIME_DEFAULT;
11138         }
11139
11140         if (((data->DataSize == 0) &&
11141             (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_NONE)) ||
11142             ((data->DataSize != 0) &&
11143             ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_READ) ||
11144             (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_WRITE) ||
11145             (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) &&
11146             (data->DataOutSize != 0)))) {
11147             if (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) {
11148                 data->DataDirection = MPTSAS_PASS_THRU_DIRECTION_READ;
11149             } else {
11150                 data->DataOutSize = 0;
11151             }
11152             /*
11153              * Send passthru request messages
11154             */
11155             return (mptsas_do_passthru(mpt,
11156                 (uint8_t *)((uintptr_t)data->PtrRequest),
11157                 (uint8_t *)((uintptr_t)data->PtrReply),
11158                 (uint8_t *)((uintptr_t)data->PtrData),
11159                 data->RequestSize, data->ReplySize,
11160                 data->DataSize, data->DataDirection,
11161                 (uint8_t *)((uintptr_t)data->PtrDataOut),
11162                 data->DataOutSize, data->Timeout, mode));
11163         } else {
11164             return (EINVAL);
11165         }
11166     }

11167     static uint8_t
11168     mptsas_get_fw_diag_buffer_number(mptsas_t *mpt, uint32_t unique_id)
11169     {
11170         uint8_t index;
11171
11172         for (index = 0; index < MPI2_DIAG_BUF_TYPE_COUNT; index++) {
11173             if (mpt->m_fw_diag_buffer_list[index].unique_id == unique_id) {
11174                 return (index);
11175             }
11176         }
11177
11178         return (MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND);
11179     }
11180 }

11181 static void
11182 mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
11183 {
11184     pMpI2DiagBufferPostRequest_t    pDiag_post_msg;
11185     pMpI2DiagReleaseRequest_t      pDiag_release_msg;
11186     struct scsi_pkt                *pkt = cmd->cmd_pkt;
11187     mptsas_diag_request_t          *diag = pkt->pkt_ha_private;
11188     uint32_t                         i;
11189     uint64_t                         request_desc;
11190
11191     ASSERT(mutex_owned(&mpt->m_mutex));
11192
11193     /*
11194      * Form the diag message depending on the post or release function.
11195      */
11196     if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
11197         pDiag_post_msg = (pMpI2DiagBufferPostRequest_t)
11198             (mpt->m_req_frame + (mpt->m_req_frame_size *
11199                 cmd->cmd_slot));
11200         bzero(pDiag_post_msg, mpt->m_req_frame_size);
11201         ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
11202

```

```

11203         diag->function);
11204     ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
11205     diag->pBuffer->buffer_type);
11206     ddi_put8(mpt->m_acc_req_frame_hdl,
11207     &pDiag_post_msg->ExtendedType,
11208     diag->pBuffer->extended_type);
11209     ddi_put32(mpt->m_acc_req_frame_hdl,
11210     &pDiag_post_msg->BufferLength,
11211     diag->pBuffer->buffer_data.size);
11212     for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);
11213     i++) {
11214         ddi_put32(mpt->m_acc_req_frame_hdl,
11215         &pDiag_post_msg->ProductSpecific[i],
11216         diag->pBuffer->product_specific[i]);
11217     }
11218     ddi_put32(mpt->m_acc_req_frame_hdl,
11219     &pDiag_post_msg->BufferAddress.Low,
11220     (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11221     & 0xffffffff));
11222     ddi_put32(mpt->m_acc_req_frame_hdl,
11223     &pDiag_post_msg->BufferAddress.High,
11224     (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
11225     >> 32));
11226 } else {
11227     pDiag_release_msg = (pMpI2DiagReleaseRequest_t)
11228     (mpt->m_req_frame + (mpt->m_req_frame_size *
11229     cmd->cmd_slot));
11230     bzzero(pDiag_release_msg, mpt->m_req_frame_size);
11231     ddi_put8(mpt->m_acc_req_frame_hdl,
11232     &pDiag_release_msg->Function, diag->function);
11233     ddi_put8(mpt->m_acc_req_frame_hdl,
11234     &pDiag_release_msg->BufferType,
11235     diag->pBuffer->buffer_type);
11236 }
11237 */
11238 /* Send the message
11239 */
11240 (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
11241     DDI_DMA_SYNC_FORDEV);
11242 request_desc = (cmd->cmd_slot << 16) +
11243     MPI2_REQ_DESCRIPTOR_FLAGS_DEFAULT_TYPE;
11244 cmd->cmd_rfm = NULL;
11245 MPTSAS_START_CMD(mpt, request_desc);
11246 if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
11247     DDI_SUCCESS) ||
11248     (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
11249     DDI_SUCCESS)) {
11250     ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11251 }
11252 */
11253 }
11254 static int
11255 mptsas_post_fw_diag_buffer(mptsas_t *mpt,
11256     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code)
11257 {
11258     mptsas_diag_request_t      diag;
11259     int                         status, slot_num, post_flags = 0;
11260     mptsas_cmd_t                *cmd = NULL;
11261     struct scsi_pkt              *pkt;
11262     pMpI2DiagBufferPostReply_t   reply;
11263     uint16_t                      iocstatus;
11264     uint32_t                      iocloginfo, transfer_length;
11265
11266     /*
11267     * If buffer is not enabled, just leave.
11268

```

```

11269         */
11270     *return_code = MPTSAS_FW_DIAG_ERROR_POST_FAILED;
11271     if (!pBuffer->enabled) {
11272         status = DDI_FAILURE;
11273         goto out;
11274     }
11275
11276     /*
11277     * Clear some flags initially.
11278     */
11279     pBuffer->force_release = FALSE;
11280     pBuffer->valid_data = FALSE;
11281     pBuffer->owned_by_firmware = FALSE;
11282
11283     /*
11284     * Get a cmd buffer from the cmd buffer pool
11285     */
11286     if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11287         status = DDI_FAILURE;
11288         mptsas_log(mpt, CE_NOTE, "command pool is full: Post FW Diag");
11289         goto out;
11290     }
11291     post_flags |= MPTSAS_REQUEST_POOL_CMD;
11292
11293     bzero((caddr_t)cmd, sizeof (*cmd));
11294     bzero((caddr_t)pkt, scsi_pkt_size());
11295
11296     cmd->ioc_cmd_slot = (uint32_t)(slot_num);
11297
11298     diag.pBuffer = pBuffer;
11299     diag.function = MPI2_FUNCTION_DIAG_BUFFER_POST;
11300
11301     /*
11302     * Form a blank cmd/pkt to store the acknowledgement message
11303     */
11304     pkt->pkt_ha_private = (opaque_t)&diag;
11305     pkt->pkt_flags = FLAG_HEAD;
11306     pkt->pkt_time = 60;
11307     cmd->cmd_pkt = pkt;
11308     cmd->cmd_flags = CFLAG_CMDIOC | CFLAG_FW_DIAG;
11309
11310     /*
11311     * Save the command in a slot
11312     */
11313     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11314         /*
11315         * Once passthru command get slot, set cmd_flags
11316         * CFLAG_PREPARED.
11317         */
11318     cmd->cmd_flags |= CFLAG_PREPARED;
11319     mptsas_start_diag(mpt, cmd);
11320     } else {
11321         mptsas_waitq_add(mpt, cmd);
11322     }
11323
11324     while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11325         cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11326     }
11327
11328     if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11329         status = DDI_FAILURE;
11330         mptsas_log(mpt, CE_WARN, "Post FW Diag command timeout");
11331         goto out;
11332     }
11333
11334     /*

```

```

11335     * cmd_rfm points to the reply message if a reply was given. Check the
11336     * IOCStatus to make sure everything went OK with the FW diag request
11337     * and set buffer flags.
11338 */
11339 if (cmd->cmd_rfm) {
11340     post_flags |= MPTSAS_ADDRESS_REPLY;
11341     (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11342                         DDI_DMA_SYNC_FORCPU);
11343     reply = (pMpi2DiagBufferPostReply_t)(mpt->m_reply_frame +
11344                                         (cmd->cmd_rfm -
11345                                         (mpt->m_reply_frame_dma_addr & 0xfffffffffu)));
11346
11347     /*
11348      * Get the reply message data
11349      */
11350     iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11351                           &reply->IOCStatus);
11352     iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11353                            &reply->IOCLogInfo);
11354     transfer_length = ddi_get32(mpt->m_acc_reply_frame_hdl,
11355                               &reply->TransferLength);
11356
11357     /*
11358      * If post failed quit.
11359      */
11360     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
11361         status = DDI_FAILURE;
11362         NDBG13(("post FW Diag Buffer failed: IOCStatus=0x%x, "
11363                 "IOCLogInfo=0x%xx, TransferLength=0x%xx", iocstatus,
11364                 iocloginfo, transfer_length));
11365         goto out;
11366     }
11367
11368     /*
11369      * Post was successful.
11370      */
11371     pBuffer->valid_data = TRUE;
11372     pBuffer->owned_by_firmware = TRUE;
11373     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11374     status = DDI_SUCCESS;
11375 }
11376
11377 out:
11378 /*
11379  * Put the reply frame back on the free queue, increment the free
11380  * index, and write the new index to the free index register. But only
11381  * if this reply is an ADDRESS reply.
11382 */
11383 if (post_flags & MPTSAS_ADDRESS_REPLY) {
11384     ddi_put32(mpt->m_acc_free_queue_hdl,
11385                &(uint32_t *) (void *) mpt->m_free_queue)[mpt->m_free_index],
11386                cmd->cmd_rfm);
11387     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11388                         DDI_DMA_SYNC_FORDEV);
11389     if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11390         mpt->m_free_index = 0;
11391     }
11392     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11393                mpt->m_free_index);
11394 }
11395 if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11396     mptsas_remove_cmd(mpt, cmd);
11397     post_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11398 }
11399 if (post_flags & MPTSAS_REQUEST_POOL_CMD) {
11400     mptsas_return_to_pool(mpt, cmd);

```

```

11401     }
11402     return (status);
11403 }
11404
11405 static int
11406 mptsas_release_fw_diag_buffer(mptsas_t *mpt,
11407                                mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
11408                                uint32_t diag_type)
11409 {
11410     mptsas_diag_request_t diag;
11411     int status, slot_num, rel_flags = 0;
11412     mptsas_cmd_t *cmd = NULL;
11413     struct scsi_pkt *pkt;
11414     pMpi2DiagReleaseReply_t reply;
11415     uint16_t iocstatus;
11416     uint32_t iocloginfo;
11417
11418     /*
11419      * If buffer is not enabled, just leave.
11420      */
11421     *return_code = MPTSAS_FW_DIAG_ERROR_RELEASE_FAILED;
11422     if (!pBuffer->enabled) {
11423         mptsas_log(mpt, CE_NOTE, "This buffer type is not supported "
11424                    "by the IOC");
11425         status = DDI_FAILURE;
11426         goto out;
11427     }
11428
11429     /*
11430      * Clear some flags initially.
11431      */
11432     pBuffer->force_release = FALSE;
11433     pBuffer->valid_data = FALSE;
11434     pBuffer->owned_by_firmware = FALSE;
11435
11436     /*
11437      * Get a cmd buffer from the cmd buffer pool
11438      */
11439     if ((slot_num = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
11440         status = DDI_FAILURE;
11441         mptsas_log(mpt, CE_NOTE, "command pool is full: Release FW "
11442                    "Diag");
11443         goto out;
11444     }
11445     rel_flags |= MPTSAS_REQUEST_POOL_CMD;
11446
11447     bzero((caddr_t)cmd, sizeof (*cmd));
11448     bzero((caddr_t)pkt, scsi_pkt_size());
11449
11450     cmd->ioc_cmd_slot = (uint32_t)(slot_num);
11451
11452     diag.pBuffer = pBuffer;
11453     diag.function = MPI2_FUNCTION_DIAG_RELEASE;
11454
11455     /*
11456      * Form a blank cmd/pkt to store the acknowledgement message
11457      */
11458     pkt->pkt_ha_private = (opaque_t)&diag;
11459     pkt->pkt_flags = FLAG_HEAD;
11460     pkt->pkt_time = 60;
11461     cmd->cmd_pkt = pkt;
11462     cmd->cmd_flags = CFLAG_CMDIOC | CFLAG_FW_DIAG;
11463
11464     /*
11465      * Save the command in a slot
11466

```

```

11467     */
11468     if (mptsas_save_cmd(mpt, cmd) == TRUE) {
11469         /*
11470          * Once passthru command get slot, set cmd_flags
11471          * CFLAG_PREPARED.
11472          */
11473         cmd->cmd_flags |= CFLAG_PREPARED;
11474         mptsas_start_diag(mpt, cmd);
11475     } else {
11476         mptsas_waitq_add(mpt, cmd);
11477     }
11478
11479     while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
11480         cv_wait(&mpt->m_fw_diag_cv, &mpt->m_mutex);
11481     }
11482
11483     if (cmd->cmd_flags & CFLAG_TIMEOUT) {
11484         status = DDI_FAILURE;
11485         mptsas_log(mpt, CE_WARN, "Release FW Diag command timeout");
11486         goto out;
11487     }
11488
11489     /*
11490      * cmd_rfm points to the reply message if a reply was given. Check the
11491      * IOCStatus to make sure everything went OK with the FW diag request
11492      * and set buffer flags.
11493      */
11494     if (cmd->cmd_rfm) {
11495         rel_flags |= MPTSAS_ADDRESS_REPLY;
11496         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
11497                           DDI_DMA_SYNC_FORCPU);
11498         reply = (pMpi2DiagReleaseReply_t)(mpt->m_reply_frame +
11499                                         (cmd->cmd_rfm -
11500                                         (mpt->m_reply_frame_dma_addr & 0xfffffffffu)));
11501
11502     /*
11503      * Get the reply message data
11504      */
11505     iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
11506                           &reply->IOCStatus);
11507     iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
11508                           &reply->IOCLogInfo);
11509
11510     /*
11511      * If release failed quit.
11512      */
11513     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) ||
11514         pBuffer->owned_by_firmware) {
11515         status = DDI_FAILURE;
11516         NDBG13(("release FW Diag Buffer failed: "
11517                 "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
11518                 iocloginfo));
11519         goto out;
11520     }
11521
11522     /*
11523      * Release was successful.
11524      */
11525     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11526     status = DDI_SUCCESS;
11527
11528     /*
11529      * If this was for an UNREGISTER diag type command, clear the
11530      * unique ID.
11531      */
11532     if (diag_type == MPTSAS_FW_DIAG_TYPE_UNREGISTER) {

```

```

11533             pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11534         }
11535     }
11536
11537     out:
11538     /*
11539      * Put the reply frame back on the free queue, increment the free
11540      * index, and write the new index to the free index register. But only
11541      * if this reply is an ADDRESS reply.
11542      */
11543     if (rel_flags & MPTSAS_ADDRESS_REPLY) {
11544         ddi_put32(mpt->m_acc_free_queue_hdl,
11545                   &(uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
11546                   cmd->cmd_rfm);
11547         (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
11548                           DDI_DMA_SYNC_FORDEV);
11549         if (++mpt->m_free_index == mpt->m_free_queue_depth) {
11550             mpt->m_free_index = 0;
11551         }
11552         ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
11553                   mpt->m_free_index);
11554     }
11555     if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
11556         mptsas_remove_cmd(mpt, cmd);
11557         rel_flags &= (~MPTSAS_REQUEST_POOL_CMD);
11558     }
11559     if (rel_flags & MPTSAS_REQUEST_POOL_CMD) {
11560         mptsas_return_to_pool(mpt, cmd);
11561     }
11562
11563     return (status);
11564 }
11565
11566 static int
11567 mptsas_diag_register(mptsas_t *mpt, mptsas_fw_diag_register_t *diag_register,
11568                       uint32_t *return_code)
11569 {
11570     mptsas_fw_diagnostic_buffer_t    *pBuffer;
11571     uint8_t                          extended_type, buffer_type, i;
11572     uint32_t                         buffer_size;
11573     uint32_t                         unique_id;
11574     int                             status;
11575
11576     ASSERT(mutex_owned(&mpt->m_mutex));
11577
11578     extended_type = diag_register->ExtendedType;
11579     buffer_type = diag_register->BufferType;
11580     buffer_size = diag_register->RequestedBufferSize;
11581     unique_id = diag_register->UniqueId;
11582
11583     /*
11584      * Check for valid buffer type
11585      */
11586     if (buffer_type >= MPI2_DIAG_BUF_TYPE_COUNT) {
11587         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11588         return (DDI_FAILURE);
11589     }
11590
11591     /*
11592      * Get the current buffer and look up the unique ID. The unique ID
11593      * should not be found. If it is, the ID is already in use.
11594      */
11595     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11596     pBuffer = &mpt->m_fw_diag_buffer_list[buffer_type];
11597     if (i != MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11598         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;

```

```

11599         return (DDI_FAILURE);
11600     }
11601
11602     /*
11603      * The buffer's unique ID should not be registered yet, and the given
11604      * unique ID cannot be 0.
11605     */
11606     if ((pBuffer->unique_id != MPTSAS_FW_DIAG_INVALID_UID) ||
11607         (unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11608         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11609         return (DDI_FAILURE);
11610     }
11611
11612     /*
11613      * If this buffer is already posted as immediate, just change owner.
11614     */
11615     if (pBuffer->immediate && pBuffer->owned_by_firmware &&
11616         (pBuffer->unique_id == MPTSAS_FW_DIAG_INVALID_UID)) {
11617         pBuffer->immediate = FALSE;
11618         pBuffer->unique_id = unique_id;
11619         return (DDI_SUCCESS);
11620     }
11621
11622     /*
11623      * Post a new buffer after checking if it's enabled. The DMA buffer
11624      * that is allocated will be contiguous (sgl_len = 1).
11625     */
11626     if (!pBuffer->enabled) {
11627         *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11628         return (DDI_FAILURE);
11629     }
11630     bzero(&pBuffer->buffer_data, sizeof (mptsas_dma_alloc_state_t));
11631     pBuffer->buffer_data.size = buffer_size;
11632     if (mptsas_dma_alloc(mpt, &pBuffer->buffer_data) != DDI_SUCCESS) {
11633         mptsas_log(mpt, CE_WARN, "Failed to alloc DMA resource for "
11634             "diag buffer: size = %d bytes", buffer_size);
11635         *return_code = MPTSAS_FW_DIAG_ERROR_NO_BUFFER;
11636         return (DDI_FAILURE);
11637     }
11638
11639     /*
11640      * Copy the given info to the diag buffer and post the buffer.
11641     */
11642     pBuffer->buffer_type = buffer_type;
11643     pBuffer->immediate = FALSE;
11644     if (buffer_type == MPI2_DIAG_BUF_TYPE_TRACE) {
11645         for (i = 0; i < (sizeof (pBuffer->product_specific) / 4);
11646             i++) {
11647             pBuffer->product_specific[i] =
11648                 diag_register->ProductSpecific[i];
11649         }
11650     }
11651     pBuffer->extended_type = extended_type;
11652     pBuffer->unique_id = unique_id;
11653     status = mptsas_post_fw_diag_buffer(mpt, pBuffer, return_code);
11654
11655     if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11656         DDI_SUCCESS) {
11657         mptsas_log(mpt, CE_WARN, "Check of DMA handle failed in "
11658             "mptsas_diag_register.");
11659         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
11660         status = DDI_FAILURE;
11661     }
11662
11663     /*
11664      * In case there was a failure, free the DMA buffer.
11665 
```

```

11665         */
11666     if (status == DDI_FAILURE) {
11667         mptsas_dma_free(&pBuffer->buffer_data);
11668     }
11669
11670     return (status);
11671 }
11672
11673 static int
11674 mptsas_diag_unregister(mptsas_t *mpt,
11675                         mptsas_fw_diag_unregister_t *diag_unregister,
11676                         uint32_t *return_code)
11677 {
11678     mptsas_fw_diagnostic_buffer_t    *pBuffer;
11679     uint8_t                          i;
11680     uint32_t                         unique_id;
11681     int                             status;
11682
11683     ASSERT(mutex_owned(&mpt->m_mutex));
11684
11685     unique_id = diag_unregister->UniqueId;
11686
11687     /*
11688      * Get the current buffer and look up the unique ID. The unique ID
11689      * should be there.
11690     */
11691     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11692     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11693         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11694         return (DDI_FAILURE);
11695     }
11696
11697     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11698
11699     /*
11700      * Try to release the buffer from FW before freeing it. If release
11701      * fails, don't free the DMA buffer in case FW tries to access it
11702      * later. If buffer is not owned by firmware, can't release it.
11703     */
11704     if (!pBuffer->owned_by_firmware) {
11705         status = DDI_SUCCESS;
11706     } else {
11707         status = mptsas_release_fw_diag_buffer(mpt, pBuffer,
11708                                              return_code, MPTSAS_FW_DIAG_TYPE_UNREGISTER);
11709     }
11710
11711     /*
11712      * At this point, return the current status no matter what happens with
11713      * the DMA buffer.
11714     */
11715     pBuffer->unique_id = MPTSAS_FW_DIAG_INVALID_UID;
11716     if (status == DDI_SUCCESS) {
11717         if (mptsas_check_dma_handle(pBuffer->buffer_data.handle) !=
11718             DDI_SUCCESS) {
11719             mptsas_log(mpt, CE_WARN, "Check of DMA handle failed in "
11720                     "in mptsas_diag_unregister.");
11721             ddi_fm_service_impact(mpt->m_dip,
11722                                   DDI_SERVICE_UNAFFECTED);
11723         }
11724         mptsas_dma_free(&pBuffer->buffer_data);
11725     }
11726
11727     return (status);
11728 }
11729
11730 static int
11731 mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
```

```

11731     uint32_t *return_code)
11732 {
11733     mptsas_fw_diagnostic_buffer_t *pBuffer;
11734     uint8_t i;
11735     uint32_t unique_id;
11736
11737     ASSERT(mutex_owned(&mpt->m_mutex));
11738
11739     unique_id = diag_query->UniqueId;
11740
11741     /*
11742      * If ID is valid, query on ID.
11743      * If ID is invalid, query on buffer type.
11744      */
11745     if (unique_id == MPTSAS_FW_DIAG_INVALID_UID) {
11746         i = diag_query->BufferType;
11747         if (i >= MPI2_DIAG_BUF_TYPE_COUNT) {
11748             *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11749             return (DDI_FAILURE);
11750     }
11751 } else {
11752     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11753     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11754         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11755         return (DDI_FAILURE);
11756     }
11757 }
11758
11759     /*
11760      * Fill query structure with the diag buffer info.
11761      */
11762     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11763     diag_query->BufferType = pBuffer->buffer_type;
11764     diag_query->ExtendedType = pBuffer->extended_type;
11765     if (diag_query->BufferType == MPI2_DIAG_BUF_TYPE_TRACE) {
11766         for (i = 0; i < (sizeof (diag_query->ProductSpecific) / 4);
11767             i++) {
11768             diag_query->ProductSpecific[i] =
11769                 pBuffer->product_specific[i];
11770         }
11771     }
11772     diag_query->TotalBufferSize = pBuffer->buffer_data.size;
11773     diag_query->DriverAddedBufferSize = 0;
11774     diag_query->UniqueId = pBuffer->unique_id;
11775     diag_query->ApplicationFlags = 0;
11776     diag_query->DiagnosticFlags = 0;
11777
11778     /*
11779      * Set/Clear application flags
11780      */
11781     if (pBuffer->immediate) {
11782         diag_query->ApplicationFlags &= ~MPTSAS_FW_DIAG_FLAG_APP OWNED;
11783     } else {
11784         diag_query->ApplicationFlags |= MPTSAS_FW_DIAG_FLAG_APP OWNED;
11785     }
11786     if (pBuffer->valid_data || pBuffer->owned_by_firmware) {
11787         diag_query->ApplicationFlags |=
11788             MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11789     } else {
11790         diag_query->ApplicationFlags &=
11791             ~MPTSAS_FW_DIAG_FLAG_BUFFER_VALID;
11792     }
11793     if (pBuffer->owned_by_firmware) {
11794         diag_query->ApplicationFlags |=
11795             MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11796 } else {

```

```

11797             diag_query->ApplicationFlags &=
11798                 ~MPTSAS_FW_DIAG_FLAG_FW_BUFFER_ACCESS;
11799         }
11800     }
11801     return (DDI_SUCCESS);
11802 }
11803
11804 static int
11805 mptsas_diag_read_buffer(mptsas_t *mpt,
11806     mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
11807     uint32_t *return_code, int ioctl_mode)
11808 {
11809     mptsas_fw_diagnostic_buffer_t *pBuffer;
11810     uint8_t i, *pData;
11811     uint32_t unique_id, byte;
11812     int status;
11813
11814     ASSERT(mutex_owned(&mpt->m_mutex));
11815
11816     unique_id = diag_read_buffer->UniqueId;
11817
11818     /*
11819      * Get the current buffer and look up the unique ID. The unique ID
11820      * should be there.
11821      */
11822     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11823     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11824         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11825         return (DDI_FAILURE);
11826     }
11827
11828     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11829
11830     /*
11831      * Make sure requested read is within limits
11832      */
11833     if (diag_read_buffer->StartingOffset + diag_read_buffer->BytesToRead >
11834         pBuffer->buffer_data.size) {
11835         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11836         return (DDI_FAILURE);
11837     }
11838
11839     /*
11840      * Copy the requested data from DMA to the diag_read_buffer. The DMA
11841      * buffer that was allocated is one contiguous buffer.
11842      */
11843     pData = (uint8_t *) (pBuffer->buffer_data.memp +
11844         diag_read_buffer->StartingOffset);
11845     (void) ddi_dma_sync(pBuffer->buffer_data.handle, 0, 0,
11846         DDI_DMA_SYNC_FORCPU);
11847     for (byte = 0; byte < diag_read_buffer->BytesToRead; byte++) {
11848         if (ddi_copyout(pData + byte, ioctl_buf + byte, 1, ioctl_mode)
11849             != 0) {
11850             return (DDI_FAILURE);
11851         }
11852     }
11853     diag_read_buffer->Status = 0;
11854
11855     /*
11856      * Set or clear the Force Release flag.
11857      */
11858     if (pBuffer->force_release) {
11859         diag_read_buffer->Flags |= MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11860     } else {
11861         diag_read_buffer->Flags &= ~MPTSAS_FW_DIAG_FLAG_FORCE_RELEASE;
11862     }

```

```

11864      /*
11865       * If buffer is to be reregistered, make sure it's not already owned by
11866       * firmware first.
11867       */
11868     status = DDI_SUCCESS;
11869     if (!pBuffer->owned_by_firmware) {
11870         if (diag_read_buffer->Flags & MPTSAS_FW_DIAG_FLAG_RREGISTER) {
11871             status = mptsas_post_fw_diag_buffer(mpt, pBuffer,
11872                                                 return_code);
11873         }
11874     }
11875     return (status);
11877 }

11879 static int
11880 mptsas_diag_release(mptsas_t *mpt, mptsas_fw_diag_release_t *diag_release,
11881                      uint32_t *return_code)
11882 {
11883     mptsas_fw_diagnostic_buffer_t    *pBuffer;
11884     uint8_t                          i;
11885     uint32_t                         unique_id;
11886     int                             status;
11888
11889     ASSERT(mutex_owned(&mpt->m_mutex));
11890
11891     unique_id = diag_release->UniqueId;
11892
11893     /*
11894      * Get the current buffer and look up the unique ID.  The unique ID
11895      * should be there.
11896      */
11897     i = mptsas_get_fw_diag_buffer_number(mpt, unique_id);
11898     if (i == MPTSAS_FW_DIAGNOSTIC_UID_NOT_FOUND) {
11899         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_UID;
11900         return (DDI_FAILURE);
11901     }
11902
11903     pBuffer = &mpt->m_fw_diag_buffer_list[i];
11904
11905     /*
11906      * If buffer is not owned by firmware, it's already been released.
11907      */
11908     if (!pBuffer->owned_by_firmware) {
11909         *return_code = MPTSAS_FW_DIAG_ERROR_ALREADY_RELEASED;
11910         return (DDI_FAILURE);
11911     }
11912
11913     /*
11914      * Release the buffer.
11915      */
11916     status = mptsas_release_fw_diag_buffer(mpt, pBuffer, return_code,
11917                                           MPTSAS_FW_DIAG_TYPE_RELEASE);
11918
11919
11920 static int
11921 mptsas_do_diag_action(mptsas_t *mpt, uint32_t action, uint8_t *diag_action,
11922                        uint32_t length, uint32_t *return_code, int ioctl_mode)
11923 {
11924     mptsas_fw_diag_register_t        diag_register;
11925     mptsas_fw_diag_unregister_t     diag_unregister;
11926     mptsas_fw_diag_query_t          diag_query;
11927     mptsas_diag_read_buffer_t       diag_read_buffer;
11928     mptsas_fw_diag_release_t        diag_release;

```

```

11929     int                                status = DDI_SUCCESS;
11930     uint32_t                           original_return_code, read_buf_len;
11932
11933     ASSERT(mutex_owned(&mpt->m_mutex));
11934
11935     original_return_code = *return_code;
11936     *return_code = MPTSAS_FW_DIAG_ERROR_SUCCESS;
11937
11938     switch (action) {
11939         case MPTSAS_FW_DIAG_TYPE_REGISTER:
11940             if (!length) {
11941                 *return_code =
11942                     MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11943                 status = DDI_FAILURE;
11944                 break;
11945             }
11946             if (ddi_copyin(diag_action, &diag_register,
11947                            sizeof (diag_register), ioctl_mode) != 0) {
11948                 return (DDI_FAILURE);
11949             }
11950             status = mptsas_diag_register(mpt, &diag_register,
11951                                           return_code);
11952             break;
11953
11954         case MPTSAS_FW_DIAG_TYPE_UNREGISTER:
11955             if (length < sizeof (diag_unregister)) {
11956                 *return_code =
11957                     MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11958                 status = DDI_FAILURE;
11959                 break;
11960             }
11961             if (ddi_copyin(diag_action, &diag_unregister,
11962                            sizeof (diag_unregister), ioctl_mode) != 0) {
11963                 return (DDI_FAILURE);
11964             }
11965             status = mptsas_diag_unregister(mpt, &diag_unregister,
11966                                           return_code);
11967             break;
11968
11969         case MPTSAS_FW_DIAG_TYPE_QUERY:
11970             if (length < sizeof (diag_query)) {
11971                 *return_code =
11972                     MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
11973                 status = DDI_FAILURE;
11974                 break;
11975             }
11976             if (ddi_copyin(diag_action, &diag_query,
11977                            sizeof (diag_query), ioctl_mode) != 0) {
11978                 return (DDI_FAILURE);
11979             }
11980             status = mptsas_diag_query(mpt, &diag_query,
11981                                       return_code);
11982             if (status == DDI_SUCCESS) {
11983                 if (ddi_copyout(&diag_query, diag_action,
11984                               sizeof (diag_query), ioctl_mode) != 0) {
11985                     return (DDI_FAILURE);
11986                 }
11987             }
11988             break;
11989
11990         case MPTSAS_FW_DIAG_TYPE_READ_BUFFER:
11991             if (ddi_copyin(diag_action, &diag_read_buffer,
11992                            sizeof (diag_read_buffer) - 4, ioctl_mode) != 0) {
11993                 return (DDI_FAILURE);
11994             }
11995             read_buf_len = sizeof (diag_read_buffer) -

```

```

11995         sizeof (diag_read_buffer.DataBuffer) +
11996         diag_read_buffer.BytesToRead;
11997     if (length < read_buf_len) {
11998         *return_code =
11999             MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
12000     status = DDI_FAILURE;
12001     break;
12002 }
12003     status = mptsas_diag_read_buffer(mpt,
12004         &diag_read_buffer, diag_action +
12005         sizeof (diag_read_buffer) - 4, return_code,
12006         ioctl_mode);
12007     if (status == DDI_SUCCESS) {
12008         if (ddi_copyout(&diag_read_buffer, diag_action,
12009             sizeof (diag_read_buffer) - 4, ioctl_mode)
12010             != 0) {
12011             return (DDI_FAILURE);
12012         }
12013     }
12014     break;

12016     case MPTSAS_FW_DIAG_TYPE_RELEASE:
12017         if (length < sizeof (diag_release)) {
12018             *return_code =
12019                 MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
12020             status = DDI_FAILURE;
12021             break;
12022         }
12023         if (ddi_copyin(diag_action, &diag_release,
12024             sizeof (diag_release), ioctl_mode) != 0) {
12025             return (DDI_FAILURE);
12026         }
12027         status = mptsas_diag_release(mpt, &diag_release,
12028             return_code);
12029         break;

12031     default:
12032         *return_code = MPTSAS_FW_DIAG_ERROR_INVALID_PARAMETER;
12033         status = DDI_FAILURE;
12034         break;
12035     }

12037     if ((status == DDI_FAILURE) &&
12038         (original_return_code == MPTSAS_FW_DIAG_NEW) &&
12039         (*return_code != MPTSAS_FW_DIAG_ERROR_SUCCESS)) {
12040         status = DDI_SUCCESS;
12041     }
12043     return (status);
12044 }

12046 static int
12047 mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *user_data, int mode)
12048 {
12049     int                     status;
12050     mptsas_diag_action_t   driver_data;
12052
12053     ASSERT(mutex_owned(&mpt->m_mutex));

12054     /*
12055      * Copy the user data to a driver data buffer.
12056      */
12057     if (ddi_copyin(user_data, &driver_data, sizeof (mptsas_diag_action_t),
12058         mode) == 0) {
12059         /*
12060          * Send diag action request if Action is valid

```

```

12061     */
12062     if (driver_data.Action == MPTSAS_FW_DIAG_TYPE_REGISTER ||
12063         driver_data.Action == MPTSAS_FW_DIAG_TYPE_UNREGISTER ||
12064         driver_data.Action == MPTSAS_FW_DIAG_TYPE_QUERY ||
12065         driver_data.Action == MPTSAS_FW_DIAG_TYPE_READ_BUFFER ||
12066         driver_data.Action == MPTSAS_FW_DIAG_TYPE_RELEASE) {
12067         status = mptsas_do_diag_action(mpt, driver_data.Action,
12068             (void *)(uintptr_t)driver_data.PtrDiagAction,
12069             driver_data.Length, &driver_data.ReturnCode,
12070             mode);
12071     if (status == DDI_SUCCESS) {
12072         if (ddi_copyout(&driver_data.ReturnCode,
12073             &user_data->ReturnCode,
12074             sizeof (user_data->ReturnCode), mode)
12075             != 0) {
12076             status = EFAULT;
12077         } else {
12078             status = 0;
12079         }
12080     } else {
12081         status = EIO;
12082     }
12083 } else {
12084     status = EINVAL;
12085 }
12086 } else {
12087     status = EFAULT;
12088 }

12089 return (status);
12090 }

12093 /*
12094  * This routine handles the "event query" ioctl.
12095 */
12096 static int
12097 mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data, int mode,
12098                      int *rval)
12099 {
12100     int                     status;
12101     mptsas_event_query_t   driverdata;
12102     uint8_t                  i;

12104     driverdata.Entries = MPTSAS_EVENT_QUEUE_SIZE;

12106     mutex_enter(&mpt->m_mutex);
12107     for (i = 0; i < 4; i++) {
12108         driverdata.Types[i] = mpt->m_event_mask[i];
12109     }
12110     mutex_exit(&mpt->m_mutex);

12112     if (ddi_copyout(&driverdata, data, sizeof (driverdata), mode) != 0) {
12113         status = EFAULT;
12114     } else {
12115         *rval = MPTIOCTL_STATUS_GOOD;
12116         status = 0;
12117     }

12119 return (status);
12120 }

12122 /*
12123  * This routine handles the "event enable" ioctl.
12124 */
12125 static int
12126 mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data, int mode,
```

```

12127     int *rval)
12128 {
12129     int          status;
12130     mptsas_event_enable_t   driverdata;
12131     uint8_t        i;
12132
12133     if (ddi_copyin(data, &driverdata, sizeof (driverdata), mode) == 0) {
12134         mutex_enter(&mpt->m_mutex);
12135         for (i = 0; i < 4; i++) {
12136             mpt->m_event_mask[i] = driverdata.Types[i];
12137         }
12138         mutex_exit(&mpt->m_mutex);
12139
12140         *rval = MPTIOCTL_STATUS_GOOD;
12141         status = 0;
12142     } else {
12143         status = EFAULT;
12144     }
12145     return (status);
12146 }
12147 */
12148 /* This routine handles the "event report" ioctl.
12149 */
12150 static int
12151 mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data, int mode,
12152                      int *rval)
12153 {
12154     int          status;
12155     mptsas_event_report_t   driverdata;
12156
12157     mutex_enter(&mpt->m_mutex);
12158
12159     if (ddi_copyin(&data->Size, &driverdata.Size, sizeof (driverdata.Size),
12160                   mode) == 0) {
12161         if (driverdata.Size >= sizeof (mpt->m_events)) {
12162             if (ddi_copyout(mpt->m_events, data->Events,
12163                            sizeof (mpt->m_events), mode) != 0) {
12164                 status = EFAULT;
12165             } else {
12166                 if (driverdata.Size > sizeof (mpt->m_events)) {
12167                     driverdata.Size =
12168                         sizeof (mpt->m_events);
12169                     if (ddi_copyout(&driverdata.Size,
12170                                   &data->Size,
12171                                   sizeof (driverdata.Size),
12172                                   mode) != 0) {
12173                         status = EFAULT;
12174                     } else {
12175                         *rval = MPTIOCTL_STATUS_GOOD;
12176                         status = 0;
12177                     }
12178                 } else {
12179                     *rval = MPTIOCTL_STATUS_GOOD;
12180                     status = 0;
12181                 }
12182             } else {
12183                 *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12184                 status = 0;
12185             }
12186         } else {
12187             status = EFAULT;
12188         }
12189     }
12190
12191     mutex_exit(&mpt->m_mutex);

```

```

12193         return (status);
12194     }
12195
12196     static void
12197     mptsas_lookup_pci_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12198     {
12199         int          *reg_data;
12200         uint_t        reglen;
12201
12202         /*
12203          * Lookup the 'reg' property and extract the other data
12204          */
12205         if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12206                                      DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
12207             DDI_PROP_SUCCESS) {
12208             /*
12209              * Extract the PCI data from the 'reg' property first DWORD.
12210              * The entry looks like the following:
12211              * First DWORD:
12212              * Bits 0 - 7 8-bit Register number
12213              * Bits 8 - 10 3-bit Function number
12214              * Bits 11 - 15 5-bit Device number
12215              * Bits 16 - 23 8-bit Bus number
12216              * Bits 24 - 25 2-bit Address Space type identifier
12217              *
12218              */
12219             adapter_data->PciInformation.u.bits.BusNumber =
12220                 (reg_data[0] & 0x00FF0000) >> 16;
12221             adapter_data->PciInformation.u.bits.DeviceNumber =
12222                 (reg_data[0] & 0x0000F800) >> 11;
12223             adapter_data->PciInformation.u.bits.FunctionNumber =
12224                 (reg_data[0] & 0x00000700) >> 8;
12225             ddi_prop_free((void *)reg_data);
12226         } else {
12227             /*
12228              * If we can't determine the PCI data then we fill in FF's for
12229              * the data to indicate this.
12230              */
12231             adapter_data->PCIDeviceHwId = 0xFFFFFFFF;
12232             adapter_data->MpiPortNumber = 0xFFFFFFFF;
12233             adapter_data->PciInformation.u.AsDWORD = 0xFFFFFFFF;
12234         }
12235
12236         /*
12237          * Saved in the mpt->m_firmwareversion
12238          */
12239         adapter_data->MpiFirmwareVersion = mpt->m_firmwareversion;
12240     }
12241
12242     static void
12243     mptsas_read_adapter_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
12244     {
12245         char          *driver_verstr = MPTAS_MOD_STRING;
12246
12247         mptsas_lookup_pci_data(mpt, adapter_data);
12248         adapter_data->AdapterType = mpt->m_MPI25 ?
12249             MPTIOCTL_ADAPTER_TYPE_SAS3 :
12250             MPTIOCTL_ADAPTER_TYPE_SAS2;
12251         adapter_data->PCIDeviceHwId = (uint32_t)mpt->m_devid;
12252         adapter_data->PCIDeviceHwRev = (uint32_t)mpt->m_revid;
12253         adapter_data->SubSystemId = (uint32_t)mpt->m_ssid;
12254         adapter_data->SubSystemVendorId = (uint32_t)mpt->m_svrid;
12255         (void) strcpy((char *)&adapter_data->DriverVersion[0], driver_verstr);
12256         adapter_data->BiosVersion = 0;
12257         (void) mptsas_get_bios_page3(mpt, &adapter_data->BiosVersion);
12258     }

```

```

12260 static void
12261 mptsas_read_pci_info(mptsas_t *mpt, mptsas_pci_info_t *pci_info)
12262 {
12263     int      *reg_data, i;
12264     uint_t    reglen;
12265
12266     /*
12267      * Lookup the 'reg' property and extract the other data
12268      */
12269     if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, mpt->m_dip,
12270         DDI_PROP_DONTPASS, "reg", &reg_data, &reglen) ==
12271         DDI_PROP_SUCCESS) {
12272         /*
12273          * Extract the PCI data from the 'reg' property first DWORD.
12274          * The entry looks like the following:
12275          * First DWORD:
12276          * Bits 8 - 10 3-bit Function number
12277          * Bits 11 - 15 5-bit Device number
12278          * Bits 16 - 23 8-bit Bus number
12279          */
12280         pci_info->BusNumber = (reg_data[0] & 0x00FF0000) >> 16;
12281         pci_info->DeviceNumber = (reg_data[0] & 0x0000F800) >> 11;
12282         pci_info->FunctionNumber = (reg_data[0] & 0x00000700) >> 8;
12283         ddi_prop_free((void *)reg_data);
12284     } else {
12285         /*
12286          * If we can't determine the PCI info then we fill in FF's for
12287          * the data to indicate this.
12288          */
12289         pci_info->BusNumber = 0xFFFFFFFF;
12290         pci_info->DeviceNumber = 0xFF;
12291         pci_info->FunctionNumber = 0xFF;
12292     }
12293
12294     /*
12295      * Now get the interrupt vector and the pci header. The vector can
12296      * only be 0 right now. The header is the first 256 bytes of config
12297      * space.
12298      */
12299     pci_info->InterruptVector = 0;
12300     for (i = 0; i < sizeof(pci_info->PciHeader); i++) {
12301         pci_info->PciHeader[i] = pci_config_get8(mpt->m_config_handle,
12302             i);
12303     }
12304 }
12305 static int
12306 mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data, int mode)
12307 {
12308     int      status = 0;
12309     mptsas_reg_access_t driverdata;
12310
12311     mutex_enter(&mpt->m_mutex);
12312     if (ddi_copyin(data, &driverdata, sizeof(driverdata), mode) == 0) {
12313         switch (driverdata.Command) {
12314             /*
12315              * IO access is not supported.
12316              */
12317             case REG_IO_READ:
12318             case REG_IO_WRITE:
12319                 mptsas_log(mpt, CE_WARN, "IO access is not "
12320                            "supported. Use memory access.");
12321                 status = EINVAL;
12322                 break;

```

```

12325     case REG_MEM_READ:
12326         driverdata.RegData = ddi_get32(mpt->m_datap,
12327             (uint32_t *)(void *)mpt->m_reg +
12328             driverdata.RegOffset);
12329         if (ddi_copyout(&driverdata.RegData,
12330             &data->RegData,
12331             sizeof(driverdata.RegData), mode) != 0) {
12332             mptsas_log(mpt, CE_WARN, "Register "
12333                           "Read Failed");
12334             status = EFAULT;
12335         }
12336         break;
12337
12338     case REG_MEM_WRITE:
12339         ddi_put32(mpt->m_datap,
12340             (uint32_t *)(void *)mpt->m_reg +
12341             driverdata.RegOffset,
12342             driverdata.RegData);
12343         break;
12344
12345     default:
12346         status = EINVAL;
12347         break;
12348     } else {
12349         status = EFAULT;
12350     }
12351 }
12352 mutex_exit(&mpt->m_mutex);
12353 return (status);
12354 }
12355
12356 static int
12357 led_control(mptsas_t *mpt, intptr_t data, int mode)
12358 {
12359     int ret = 0;
12360     mptsas_led_control_t lc;
12361     mptsas_target_t *ptgt;
12362
12363     if (ddi_copyin((void *)data, &lc, sizeof(lc), mode) != 0) {
12364         return (EFAULT);
12365     }
12366
12367     if ((lc.Command != MPTSAS_LEDCTL_FLAG_SET &&
12368         lc.Command != MPTSAS_LEDCTL_FLAG_GET) ||
12369         lc.Led < MPTSAS_LEDCTL_LED_MIN ||
12370         lc.Led > MPTSAS_LEDCTL_LED_MAX ||
12371         (lc.Command == MPTSAS_LEDCTL_FLAG_SET && lc.LedStatus != 0 &&
12372         lc.LedStatus != 1)) {
12373         return (EINVAL);
12374     }
12375
12376     if ((lc.Command == MPTSAS_LEDCTL_FLAG_SET && (mode & FWRITE) == 0) ||
12377         (lc.Command == MPTSAS_LEDCTL_FLAG_GET && (mode & FREAD) == 0))
12378         return (EACCES);
12379
12380     /* Locate the target we're interrogating... */
12381     mutex_enter(&mpt->m_mutex);
12382     ptgt = refhash_linear_search(mpt->m_targets,
12383         mptsas_target_eval_slot, &lc);
12384     if (ptgt == NULL) {
12385         /* We could not find a target for that enclosure/slot. */
12386         mutex_exit(&mpt->m_mutex);
12387         return (ENOENT);
12388     }
12389 }
```

```

12391     if (lc.Command == MPTSAS_LEDCTL_FLAG_SET) {
12392         /* Update our internal LED state. */
12393         ptgt->m_led_status &= ~(1 << (lc.Led - 1));
12394         ptgt->m_led_status |= lc.LedStatus << (lc.Led - 1);
12395
12396         /* Flush it to the controller. */
12397         ret = mptsas_flush_led_status(mpt, ptgt);
12398         mutex_exit(&mpt->m_mutex);
12399         return (ret);
12400     }
12401
12402     /* Return our internal LED state. */
12403     lc.LedStatus = (ptgt->m_led_status >> (lc.Led - 1)) & 1;
12404     mutex_exit(&mpt->m_mutex);
12405
12406     if (ddi_copyout(&lc, (void *)data, sizeof (lc), mode) != 0) {
12407         return (EFAULT);
12408     }
12409
12410     return (0);
12411 }
12412
12413 static int
12414 get_disk_info(mptsas_t *mpt, intptr_t data, int mode)
12415 {
12416     uint16_t i = 0;
12417     uint16_t count = 0;
12418     int ret = 0;
12419     mptsas_target_t *ptgt;
12420     mptsas_disk_info_t *di;
12421     STRUCT_DECL(mptsas_get_disk_info, gdi);
12422
12423     if ((mode & FREAD) == 0)
12424         return (EACCES);
12425
12426     STRUCT_INIT(gdi, get_udatamodel());
12427
12428     if (ddi_copyin((void *)data, STRUCT_BUF(gdi), STRUCT_SIZE(gdi),
12429                     mode) != 0) {
12430         return (EFAULT);
12431     }
12432
12433     /* Find out how many targets there are. */
12434     mutex_enter(&mpt->m_mutex);
12435     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
12436         ptgt = rehash_next(mpt->m_targets, ptgt)) {
12437         count++;
12438     }
12439     mutex_exit(&mpt->m_mutex);
12440
12441     /*
12442      * If we haven't been asked to copy out information on each target,
12443      * then just return the count.
12444      */
12445     STRUCT_FSET(gdi, DiskCount, count);
12446     if (STRUCT_FGETP(gdi, PtrDiskInfoArray) == NULL)
12447         goto copy_out;
12448
12449     /*
12450      * If we haven't been given a large enough buffer to copy out into,
12451      * let the caller know.
12452      */
12453     if (STRUCT_FGET(gdi, DiskInfoArraySize) <
12454         count * sizeof (mptsas_disk_info_t)) {
12455         ret = ENOSPC;
12456         goto copy_out;

```

```

12457     }
12458
12459     di = kmalloc(count * sizeof (mptsas_disk_info_t), KM_SLEEP);
12460
12461     mutex_enter(&mpt->m_mutex);
12462     for (ptgt = rehash_first(mpt->m_targets); ptgt != NULL;
12463         ptgt = rehash_next(mpt->m_targets, ptgt)) {
12464         if (i >= count) {
12465             /*
12466              * The number of targets changed while we weren't
12467              * looking, so give up.
12468              */
12469             rehash_rele(mpt->m_targets, ptgt);
12470             mutex_exit(&mpt->m_mutex);
12471             kmem_free(di, count * sizeof (mptsas_disk_info_t));
12472             return (EAGAIN);
12473         }
12474         di[i].Instance = mpt->m_instance;
12475         di[i].Enclosure = ptgt->m_enclosure;
12476         di[i].Slot = ptgt->m_slot_num;
12477         di[i].SasAddress = ptgt->m_addr.mta_wwn;
12478         i++;
12479     }
12480     mutex_exit(&mpt->m_mutex);
12481     STRUCT_FSET(gdi, DiskCount, i);
12482
12483     /* Copy out the disk information to the caller. */
12484     if (ddi_copyout((void *)di, STRUCT_FGETP(gdi, PtrDiskInfoArray),
12485                     i * sizeof (mptsas_disk_info_t), mode) != 0) {
12486         ret = EFAULT;
12487     }
12488
12489     kmem_free(di, count * sizeof (mptsas_disk_info_t));
12490
12491 copy_out:
12492     if (ddi_copyout(STRUCT_BUF(gdi), (void *)data, STRUCT_SIZE(gdi),
12493                     mode) != 0) {
12494         ret = EFAULT;
12495     }
12496
12497     return (ret);
12498 }
12499
12500 static int
12501 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *cred,
12502                 int *rval)
12503 {
12504     int status = 0;
12505     mptsas_t *mpt;
12506     mptsas_update_flash_t flashdata;
12507     mptsas_pass_thru_t passthru_data;
12508     mptsas_adapter_data_t adapter_data;
12509     mptsas_pci_info_t pci_info;
12510     int copylen;
12511
12512     int iport_flag = 0;
12513     dev_info_t *dip = NULL;
12514     mptsas_phymask_t phymask = 0;
12515     struct devctl_iocdata *dcp = NULL;
12516     char *addr = NULL;
12517     mptsas_target_t *ptgt = NULL;
12518
12519     *rval = MPTIOCTL_STATUS_GOOD;
12520     if (secpolicy_sys_config(cred, B_FALSE) != 0) {
12521         return (EPERM);
12522     }

```

```

12524
12525     mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
12526     if (mpt == NULL) {
12527         /*
12528          * Called from iport node, get the states
12529          */
12530         iport_flag = 1;
12531         dip = mptsas_get_dip_from_dev(dev, &phymask);
12532         if (dip == NULL) {
12533             return (ENXIO);
12534         }
12535         mpt = DIP2MPT(dip);
12536     } /* Make sure power level is D0 before accessing registers */
12537     mutex_enter(&mpt->m_mutex);
12538     if (mpt->m_options & MPTSAS_OPT_PM) {
12539         (void) pm_busy_component(mpt->m_dip, 0);
12540         if (mpt->m_power_level != PM_LEVEL_D0) {
12541             mutex_exit(&mpt->m_mutex);
12542             if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
12543                 DDI_SUCCESS) {
12544                 mptsas_log(mpt, CE_WARN,
12545                             "mptsas%d: mptsas_ioctl: Raise power "
12546                             "request failed.", mpt->m_instance);
12547                 (void) pm_idle_component(mpt->m_dip, 0);
12548                 return (ENXIO);
12549             }
12550         } else {
12551             mutex_exit(&mpt->m_mutex);
12552         }
12553     } else {
12554         mutex_exit(&mpt->m_mutex);
12555     }
12556
12557     if (iport_flag) {
12558         status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
12559         if (status != 0) {
12560             goto out;
12561         }
12562         /*
12563          * The following code control the OK2RM LED, it doesn't affect
12564          * the ioctl return status.
12565          */
12566         if ((cmd == DEVCTL_DEVICE_ONLINE) ||
12567             (cmd == DEVCTL_DEVICE_OFFLINE)) {
12568             if (ndi_dc_allochdl((void *)data, &dcp) !=
12569                 NDI_SUCCESS) {
12570                 goto out;
12571             }
12572             addr = ndi_dc_getaddr(dcp);
12573             ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
12574             if (ptgt == NULL) {
12575                 NDBG14(("mptsas_ioctl led control: tgt %s not "
12576                         "found", addr));
12577                 ndi_dc_freehdl(dcp);
12578                 goto out;
12579             }
12580             mutex_enter(&mpt->m_mutex);
12581             if (cmd == DEVCTL_DEVICE_ONLINE) {
12582                 ptgt->m_tgt_unconfigured = 0;
12583             } else if (cmd == DEVCTL_DEVICE_OFFLINE) {
12584                 ptgt->m_tgt_unconfigured = 1;
12585             }
12586             if (cmd == DEVCTL_DEVICE_OFFLINE) {
12587                 ptgt->m_led_status |=
12588                     (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12589             }
12590         }
12591     }
12592     iport_flag = 0;
12593     dip = mptsas_get_dip_from_dev(dev, &phymask);
12594     if (dip == NULL) {
12595         return (ENXIO);
12596     }
12597     goto out;
12598
12599     switch (cmd) {
12600         case MPTIOCTL_GET_DISK_INFO:
12601             status = get_disk_info(mpt, data, mode);
12602             break;
12603         case MPTIOCTL_LED_CONTROL:
12604             status = led_control(mpt, data, mode);
12605             break;
12606         case MPTIOCTL_UPDATE_FLASH:
12607             if (ddi_copyin((void *)data, &flashdata,
12608                           sizeof(struct mptsas_update_flash), mode)) {
12609                 status = EFAULT;
12610                 break;
12611             }
12612             mutex_enter(&mpt->m_mutex);
12613             if (mptsas_update_flash(mpt,
12614                                     (caddr_t)(long)flashdata.PtrBuffer,
12615                                     flashdata.ImageSize, flashdata.ImageType, mode)) {
12616                 status = EFAULT;
12617             }
12618
12619         /*
12620          * Reset the chip to start using the new
12621          * firmware.  Reset if failed also.
12622          */
12623         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12624         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
12625             status = EFAULT;
12626         }
12627         mutex_exit(&mpt->m_mutex);
12628         break;
12629     case MPTIOCTL_PASS_THRU:
12630         /*
12631          * The user has requested to pass through a command to
12632          * be executed by the MPT firmware.  Call our routine
12633          * which does this. Only allow one passthru IOCTL at
12634          * one time. Other threads will block on
12635          * m_passthru_mutex, which is of adaptive variant.
12636          */
12637         if (ddi_copyin((void *)data, &passthru_data,
12638                       sizeof(mptsas_pass_thru_t), mode)) {
12639             status = EFAULT;
12640             break;
12641         }
12642         mutex_enter(&mpt->m_passthru_mutex);
12643         mutex_enter(&mpt->m_mutex);
12644         status = mptsas_pass_thru(mpt, &passthru_data, mode);
12645         mutex_exit(&mpt->m_mutex);
12646         mutex_exit(&mpt->m_passthru_mutex);
12647
12648         break;
12649     case MPTIOCTL_GET_ADAPTER_DATA:
12650         /*
12651          * The user has requested to read adapter data.  Call
12652          * our routine which does this.
12653          */
12654

```

```

12655         }
12656         ptgt->m_led_status &=
12657             ~(1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
12658     }
12659     (void) mptsas_flush_led_status(mpt, ptgt);
12660     mutex_exit(&mpt->m_mutex);
12661     ndi_dc_freehdl(dcp);
12662 }
12663
12664 switch (cmd) {
12665     case MPTIOCTL_GET_DISK_INFO:
12666         status = get_disk_info(mpt, data, mode);
12667         break;
12668     case MPTIOCTL_LED_CONTROL:
12669         status = led_control(mpt, data, mode);
12670         break;
12671     case MPTIOCTL_UPDATE_FLASH:
12672         if (ddi_copyin((void *)data, &flashdata,
12673                       sizeof(struct mptsas_update_flash), mode)) {
12674             status = EFAULT;
12675             break;
12676         }
12677         mutex_enter(&mpt->m_mutex);
12678         if (mptsas_update_flash(mpt,
12679                                 (caddr_t)(long)flashdata.PtrBuffer,
12680                                 flashdata.ImageSize, flashdata.ImageType, mode)) {
12681             status = EFAULT;
12682         }
12683
12684     /*
12685      * Reset the chip to start using the new
12686      * firmware.  Reset if failed also.
12687      */
12688     mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12689     if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
12690         status = EFAULT;
12691     }
12692     mutex_exit(&mpt->m_mutex);
12693     break;
12694 }
12695
12696 case MPTIOCTL_PASS_THRU:
12697     /*
12698      * The user has requested to pass through a command to
12699      * be executed by the MPT firmware.  Call our routine
12700      * which does this. Only allow one passthru IOCTL at
12701      * one time. Other threads will block on
12702      * m_passthru_mutex, which is of adaptive variant.
12703      */
12704     if (ddi_copyin((void *)data, &passthru_data,
12705                   sizeof(mptsas_pass_thru_t), mode)) {
12706         status = EFAULT;
12707         break;
12708     }
12709     mutex_enter(&mpt->m_passthru_mutex);
12710     mutex_enter(&mpt->m_mutex);
12711     status = mptsas_pass_thru(mpt, &passthru_data, mode);
12712     mutex_exit(&mpt->m_mutex);
12713     mutex_exit(&mpt->m_passthru_mutex);
12714
12715     break;
12716
12717 case MPTIOCTL_GET_ADAPTER_DATA:
12718     /*
12719      * The user has requested to read adapter data.  Call
12720      * our routine which does this.
12721      */
12722

```

```

12655     bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
12656     if (ddi_copyin((void *)data, (void *)&adapter_data,
12657         sizeof (mptsas_adapter_data_t), mode)) {
12658         status = EFAULT;
12659         break;
12660     }
12661     if (adapter_data.StructureLength >=
12662         sizeof (mptsas_adapter_data_t)) {
12663         adapter_data.StructureLength = (uint32_t)
12664             sizeof (mptsas_adapter_data_t);
12665         copylen = sizeof (mptsas_adapter_data_t);
12666         mutex_enter(&mpt->m_mutex);
12667         mptsas_read_adapter_data(mpt, &adapter_data);
12668         mutex_exit(&mpt->m_mutex);
12669     } else {
12670         adapter_data.StructureLength = (uint32_t)
12671             sizeof (mptsas_adapter_data_t);
12672         copylen = sizeof (adapter_data.StructureLength);
12673         *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
12674     }
12675     if (ddi_copyout((void *)(&adapter_data), (void *)data,
12676         copylen, mode) != 0) {
12677         status = EFAULT;
12678     }
12679     break;
12680 case MPTIOCTL_GET_PCI_INFO:
12681     /*
12682      * The user has requested to read pci info. Call
12683      * our routine which does this.
12684      */
12685     bzero(&pci_info, sizeof (mptsas_pci_info_t));
12686     mutex_enter(&mpt->m_mutex);
12687     mptsas_read_pci_info(mpt, &pci_info);
12688     mutex_exit(&mpt->m_mutex);
12689     if (ddi_copyout((void *)(&pci_info), (void *)data,
12690         sizeof (mptsas_pci_info_t), mode) != 0) {
12691         status = EFAULT;
12692     }
12693     break;
12694 case MPTIOCTL_RESET_ADAPTER:
12695     mutex_enter(&mpt->m_mutex);
12696     mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
12697     if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
12698         mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
12699                     "failed");
12700         status = EFAULT;
12701     }
12702     mutex_exit(&mpt->m_mutex);
12703     break;
12704 case MPTIOCTL_DIAG_ACTION:
12705     /*
12706      * The user has done a diag buffer action. Call our
12707      * routine which does this. Only allow one diag action
12708      * at one time.
12709      */
12710     mutex_enter(&mpt->m_mutex);
12711     if (mpt->m_diag_action_in_progress) {
12712         mutex_exit(&mpt->m_mutex);
12713         return (EBUSY);
12714     }
12715     mpt->m_diag_action_in_progress = 1;
12716     status = mptsas_diag_action(mpt,
12717         (mptsas_diag_action_t *)data, mode);
12718     mpt->m_diag_action_in_progress = 0;
12719     mutex_exit(&mpt->m_mutex);
12720     break;

```

```

12721     case MPTIOCTL_EVENT_QUERY:
12722     /*
12723      * The user has done an event query. Call our routine
12724      * which does this.
12725      */
12726     status = mptsas_event_query(mpt,
12727         (mptsas_event_query_t *)data, mode, rval);
12728     break;
12729 case MPTIOCTL_EVENT_ENABLE:
12730     /*
12731      * The user has done an event enable. Call our routine
12732      * which does this.
12733      */
12734     status = mptsas_event_enable(mpt,
12735         (mptsas_event_enable_t *)data, mode, rval);
12736     break;
12737 case MPTIOCTL_EVENT_REPORT:
12738     /*
12739      * The user has done an event report. Call our routine
12740      * which does this.
12741      */
12742     status = mptsas_event_report(mpt,
12743         (mptsas_event_report_t *)data, mode, rval);
12744     break;
12745 case MPTIOCTL_REG_ACCESS:
12746     /*
12747      * The user has requested register access. Call our
12748      * routine which does this.
12749      */
12750     status = mptsas_reg_access(mpt,
12751         (mptsas_reg_access_t *)data, mode);
12752     break;
12753 default:
12754     status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
12755                             rval);
12756     break;
12757 }

12758 out:
12759     return (status);
12760 }
12761 */

12762 /* Dirty wrapper for taskq */
12763 void
12764 mptsas_handle_restart_ioc(void *mpt) {
12765     mptsas_restart_ioc((mptsas_t *) mpt);
12766 }
12767 */

12768 #endif /* ! codereview */
12769 int
12770 mptsas_restart_ioc(mptsas_t *mpt)
12771 {
12772     int          rval = DDI_SUCCESS;
12773     mptsas_target_t *ptgt = NULL;
12774
12775     ASSERT(mutex_owned(&mpt->m_mutex));
12776
12777     /*
12778      * Set a flag telling I/O path that we're processing a reset. This is
12779      * needed because after the reset is complete, the hash table still
12780      * needs to be rebuilt. If I/Os are started before the hash table is
12781      * rebuilt, I/O errors will occur. This flag allows I/Os to be marked
12782      * so that they can be retried.
12783      */
12784     mpt->m_in_reset = TRUE;
12785

```

```

12787     /*
12788      * Set all throttles to HOLD
12789      */
12790     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12791         ptgt = refhash_next(mpt->m_targets, ptgt)) {
12792         mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
12793     }
12794
12795     /*
12796      * Disable interrupts
12797      */
12798     MPTSAS_DISABLE_INTR(mpt);
12799
12800     /*
12801      * Abort all commands: outstanding commands, commands in waitq and
12802      * tx_waitq.
12803      */
12804     mptsas_flush_hba(mpt);
12805
12806     /*
12807      * Reinitialize the chip.
12808      */
12809     if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
12810         rval = DDI_FAILURE;
12811     }
12812
12813     /*
12814      * Enable interrupts again
12815      */
12816     MPTSAS_ENABLE_INTR(mpt);
12817
12818     /*
12819      * If mptsas_init_chip was successful, update the driver data.
12820      */
12821     if (rval == DDI_SUCCESS) {
12822         mptsas_update_driver_data(mpt);
12823     }
12824
12825     /*
12826      * Reset the throttles
12827      */
12828     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
12829         ptgt = refhash_next(mpt->m_targets, ptgt)) {
12830         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
12831     }
12832
12833     mptsas_doneq_empty(mpt);
12834     mptsas_restart_hba(mpt);
12835
12836     if (rval != DDI_SUCCESS) {
12837         mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
12838         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
12839     }
12840
12841     /*
12842      * Clear the reset flag so that I/Os can continue.
12843      */
12844     mpt->m_in_reset = FALSE;
12845
12846     return (rval);
12847 }
12848
12849 static int
12850 mptsas_init_chip(mptsas_t *mpt, int first_time)
12851 {
12852     ddi_dma_cookie_t cookie;

```

```

12853     uint32_t i;
12854     int rval;
12855
12856     /*
12857      * Check to see if the firmware image is valid
12858      */
12859     if (ddi_get32(mpt->m_datap, &mpt->m_reg->HostDiagnostic) &
12860         MPI2_DIAG_FLASH_BAD_SIG) {
12861         mptsas_log(mpt, CE_WARN, "mptsas bad flash signature!");
12862         goto fail;
12863     }
12864
12865     /*
12866      * Reset the chip
12867      */
12868     rval = mptsas_ioc_reset(mpt, first_time);
12869     if (rval == MPTSAS_RESET_FAIL) {
12870         mptsas_log(mpt, CE_WARN, "hard reset failed!");
12871         goto fail;
12872     }
12873
12874     if ((rval == MPTSAS_SUCCESS_MUR) && (!first_time)) {
12875         goto mur;
12876     }
12877
12878     /*
12879      * Setup configuration space
12880      */
12881     if (mptsas_config_space_init(mpt) == FALSE) {
12882         mptsas_log(mpt, CE_WARN, "mptsas_config_space_init "
12883                     "failed!");
12884         goto fail;
12885
12886     /*
12887      * IOC facts can change after a diag reset so all buffers that are
12888      * based on these numbers must be de-allocated and re-allocated.  Get
12889      * new IOC facts each time chip is initialized.
12890      */
12891     if (mptsas_ioc_get_facts(mpt) == DDI_FAILURE) {
12892         mptsas_log(mpt, CE_WARN, "mptsas_ioc_get_facts failed");
12893         goto fail;
12894     }
12895
12896     if (mptsas_alloc_active_slots(mpt, KM_SLEEP)) {
12897         goto fail;
12898     }
12899
12900     /*
12901      * Allocate request message frames, reply free queue, reply descriptor
12902      * post queue, and reply message frames using latest IOC facts.
12903      */
12904     if (mptsas_alloc_request_frames(mpt) == DDI_FAILURE) {
12905         mptsas_log(mpt, CE_WARN, "mptsas_alloc_request_frames failed");
12906         goto fail;
12907     }
12908     if (mptsas_alloc_sense_bufs(mpt) == DDI_FAILURE) {
12909         mptsas_log(mpt, CE_WARN, "mptsas_alloc_sense_bufs failed");
12910         goto fail;
12911     }
12912     if (mptsas_alloc_free_queue(mpt) == DDI_FAILURE) {
12913         mptsas_log(mpt, CE_WARN, "mptsas_alloc_free_queue failed!");
12914         goto fail;
12915     }
12916     if (mptsas_alloc_post_queue(mpt) == DDI_FAILURE) {
12917         mptsas_log(mpt, CE_WARN, "mptsas_alloc_post_queue failed!");
12918         goto fail;
12919     }

```

```

12919     if (mptsas_alloc_reply_frames(mpt) == DDI_FAILURE) {
12920         mptsas_log(mpt, CE_WARN, "mptsas_alloc_reply_frames failed!");
12921         goto fail;
12922     }

12924 mur:
12925     /*
12926      * Re-Initialize ioc to operational state
12927      */
12928     if (mptsas_ioc_init(mpt) == DDI_FAILURE) {
12929         mptsas_log(mpt, CE_WARN, "mptsas_ioc_init failed");
12930         goto fail;
12931     }

12933     mptsas_alloc_reply_args(mpt);

12935     /*
12936      * Initialize reply post index. Reply free index is initialized after
12937      * the next loop.
12938      */
12939     mpt->m_post_index = 0;

12941     /*
12942      * Initialize the Reply Free Queue with the physical addresses of our
12943      * reply frames.
12944      */
12945     cookie.dmac_address = mpt->m_reply_frame_dma_addr & 0xfffffffffu;
12946     for (i = 0; i < mpt->m_max_replies; i++) {
12947         ddi_put32(mpt->m_acc_free_queue_hdl,
12948             &(uint32_t *) (void *) mpt->m_free_queue)[i],
12949             cookie.dmac_address);
12950             cookie.dmac_address += mpt->m_reply_frame_size;
12951     }
12952     (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
12953         DDI_DMA_SYNC_FORDEV);

12955     /*
12956      * Initialize the reply free index to one past the last frame on the
12957      * queue. This will signify that the queue is empty to start with.
12958      */
12959     mpt->m_free_index = i;
12960     ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex, i);

12962     /*
12963      * Initialize the reply post queue to 0xFFFFFFFF,0xFFFFFFFF's.
12964      */
12965     for (i = 0; i < mpt->m_post_queue_depth; i++) {
12966         ddi_put64(mpt->m_acc_post_queue_hdl,
12967             &(uint64_t *) (void *) mpt->m_post_queue)[i],
12968             0xFFFFFFFFFFFFFFFF);
12969     }
12970     (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
12971         DDI_DMA_SYNC_FORDEV);

12973     /*
12974      * Enable ports
12975      */
12976     if (mptsas_ioc_enable_port(mpt) == DDI_FAILURE) {
12977         mptsas_log(mpt, CE_WARN, "mptsas_ioc_enable_port failed");
12978         goto fail;
12979     }

12981     /*
12982      * enable events
12983      */
12984     if (mptsas_ioc_enable_event_notification(mpt)) {

```

```

12985             mptsas_log(mpt, CE_WARN,
12986                 "mptsas_ioc_enable_event_notification failed");
12987             goto fail;
12988         }

12990         /*
12991          * We need checks in attach and these.
12992          * chip_init is called in mult. places
12993          */
12995         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
12996             DDI_SUCCESS) ||
12997             (mptsas_check_dma_handle(mpt->m_dma_req_sense_hdl) !=
12998                 DDI_SUCCESS) ||
12999                 (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl) !=
13000                     DDI_SUCCESS) ||
13001                     (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl) !=
13002                         DDI_SUCCESS) ||
13003                         (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
13004                             DDI_SUCCESS) ||
13005                             (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl) !=
13006                                 DDI_SUCCESS)) {
13007                                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
13008                                 goto fail;
13009     }

13011     /* Check all acc handles */
13012     if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
13013         (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
13014             DDI_SUCCESS) ||
13015                 (mptsas_check_acc_handle(mpt->m_acc_req_sense_hdl) !=
13016                     DDI_SUCCESS) ||
13017                     (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl) !=
13018                         DDI_SUCCESS) ||
13019                         (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl) !=
13020                             DDI_SUCCESS) ||
13021                             (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl) !=
13022                                 DDI_SUCCESS) ||
13023                                 (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl) !=
13024                                     DDI_SUCCESS) ||
13025                                     (mptsas_check_acc_handle(mpt->m_config_handle) !=
13026                                         DDI_SUCCESS)) {
13027                                         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
13028                                         goto fail;
13029     }

13031     return (DDI_SUCCESS);

13033 fail:
13034     return (DDI_FAILURE);
13035 }

13037 static int
13038 mptsas_get_pci_cap(mptsas_t *mpt)
13039 {
13040     ushort_t caps_ptr, cap, cap_count;

13042     if (mpt->m_config_handle == NULL)
13043         return (FALSE);
13044     /*
13045      * Check if capabilities list is supported and if so,
13046      * get initial capabilities pointer and clear bits 0,1.
13047      */
13048     if (pci_config_get16(mpt->m_config_handle, PCI_CONF_STAT) &&
13049         & PCI_STAT_CAP) {
13050         caps_ptr = P2ALIGN(pci_config_get8(mpt->m_config_handle,

```

```

13051         PCI_CONF_CAP_PTR), 4);
13052     } else {
13053         caps_ptr = PCI_CAP_NEXT_PTR_NULL;
13054     }
13055
13056     /*
13057      * Walk capabilities if supported.
13058      */
13059     for (cap_count = 0; caps_ptr != PCI_CAP_NEXT_PTR_NULL; ) {
13060
13061         /*
13062          * Check that we haven't exceeded the maximum number of
13063          * capabilities and that the pointer is in a valid range.
13064          */
13065         if (++cap_count > 48) {
13066             mptsas_log(mpt, CE_WARN,
13067                         "too many device capabilities.\n");
13068             break;
13069         }
13070         if (caps_ptr < 64) {
13071             mptsas_log(mpt, CE_WARN,
13072                         "capabilities pointer 0x%x out of range.\n",
13073                         caps_ptr);
13074             break;
13075         }
13076
13077         /*
13078          * Get next capability and check that it is valid.
13079          * For now, we only support power management.
13080          */
13081         cap = pci_config_get8(mpt->m_config_handle, caps_ptr);
13082         switch (cap) {
13083             case PCI_CAP_ID_PM:
13084                 mptsas_log(mpt, CE_NOTE,
13085                             "?mptsas%d supports power management.\n",
13086                             mpt->m_instance);
13087                 mpt->m_options |= MPTSAS_OPT_PM;
13088
13089                 /* Save PMCSR offset */
13090                 mpt->m_pmcsr_offset = caps_ptr + PCI_PMCSR;
13091                 break;
13092
13093             /*
13094              * The following capabilities are valid. Any others
13095              * will cause a message to be logged.
13096              */
13097             case PCI_CAP_ID_VPD:
13098             case PCI_CAP_ID_MSI:
13099             case PCI_CAP_ID_PCTIX:
13100             case PCI_CAP_ID_PCI_E:
13101             case PCI_CAP_ID_MSI_X:
13102                 break;
13103             default:
13104                 mptsas_log(mpt, CE_NOTE,
13105                             "?mptsas%d unrecognized capability "
13106                             "0x%x.\n", mpt->m_instance, cap);
13107                 break;
13108         }
13109
13110         /*
13111          * Get next capabilities pointer and clear bits 0,1.
13112          */
13113         caps_ptr = P2ALIGN(pci_config_get8(mpt->m_config_handle,
13114                                     (caps_ptr + PCI_CAP_NEXT_PTR)), 4);
13115     }
13116 }
```

```

13118 static int
13119 mptsas_init_pm(mptsas_t *mpt)
13120 {
13121     char pmc_name[16];
13122     char *pmc[] = {
13123         NULL,
13124         "0=Off (PCI D3 State)",
13125         "3=On (PCI D0 State)",
13126         NULL
13127     };
13128     uint16_t pmcsr_stat;
13129
13130     if (mptsas_get_pci_cap(mpt) == FALSE) {
13131         return (DDI_FAILURE);
13132     }
13133     /*
13134      * If PCI's capability does not support PM, then don't need
13135      * to registe the pm-components
13136      */
13137     if (!(mpt->m_options & MPTSAS_OPT_PM))
13138         return (DDI_SUCCESS);
13139
13140     /*
13141      * If power management is supported by this chip, create
13142      * pm-components property for the power management framework
13143      */
13144     (void) sprintf(pmc_name, "NAME=mptsas%d", mpt->m_instance);
13145     pmc[0] = pmc_name;
13146     if (ddi_prop_update_string_array(DDI_DEV_T_NONE, mpt->m_dip,
13147                                     "pm-components", pmc, 3) != DDI_PROP_SUCCESS) {
13148         mpt->m_options &= ~MPTSAS_OPT_PM;
13149         mptsas_log(mpt, CE_WARN,
13150                     "mptsas%d: pm-component property creation failed.",
13151                     mpt->m_instance);
13152         return (DDI_FAILURE);
13153     }
13154
13155     /*
13156      * Power on device.
13157      */
13158     (void) pm_busy_component(mpt->m_dip, 0);
13159     pmcsr_stat = pci_config_get16(mpt->m_config_handle,
13160                                   mpt->m_pmcsr_offset);
13161     if ((pmcsr_stat & PCI_PMCSR_STATE_MASK) != PCI_PMCSR_D0) {
13162         mptsas_log(mpt, CE_WARN, "mptsas%d: Power up the device",
13163                     mpt->m_instance);
13164         pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset,
13165                          PCI_PMCSR_D0);
13166     }
13167     if (pm_power_has_changed(mpt->m_dip, 0, PM_LEVEL_D0) != DDI_SUCCESS) {
13168         mptsas_log(mpt, CE_WARN, "pm_power_has_changed failed");
13169         return (DDI_FAILURE);
13170     }
13171     mpt->m_power_level = PM_LEVEL_D0;
13172
13173     /*
13174      * Set pm idle delay.
13175      */
13176     mpt->m_pm_idle_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
13177                                              mpt->m_dip, 0, "mptsas-pm-idle-delay", MPTSAS_PM_IDLE_TIMEOUT);
13178 }
13179
13180 static int
13181 mptsas_register_intrs(mptsas_t *mpt)
13182 {
```

```

13183     dev_info_t *dip;
13184     int intr_types;
13185
13186     dip = mpt->m_dip;
13187
13188     /* Get supported interrupt types */
13189     if (ddi_intr_get_supported_types(dip, &intr_types) != DDI_SUCCESS) {
13190         mptsas_log(mpt, CE_WARN, "ddi_intr_get_supported_types "
13191                     "failed\n");
13192         return (FALSE);
13193     }
13194
13195     NDBG6(("ddi_intr_get_supported_types() returned: 0x%x", intr_types));
13196
13197     /*
13198      * Try MSI, but fall back to FIXED
13199      */
13200     if (mptsas_enable_msi && (intr_types & DDI_INTR_TYPE_MSI)) {
13201         if (mptsas_add_intrs(mpt, DDI_INTR_TYPE_MSI) == DDI_SUCCESS) {
13202             NDBG0(("Using MSI interrupt type"));
13203             mpt->m_intr_type = DDI_INTR_TYPE_MSI;
13204             return (TRUE);
13205         }
13206
13207         if (intr_types & DDI_INTR_TYPE_FIXED) {
13208             if (mptsas_add_intrs(mpt, DDI_INTR_TYPE_FIXED) == DDI_SUCCESS) {
13209                 NDBG0(("Using FIXED interrupt type"));
13210                 mpt->m_intr_type = DDI_INTR_TYPE_FIXED;
13211                 return (TRUE);
13212             } else {
13213                 NDBG0(("FIXED interrupt registration failed"));
13214                 return (FALSE);
13215             }
13216         }
13217         return (FALSE);
13218     }
13219 }
13220
13221 static void
13222 mptsas_unregister_intrs(mptsas_t *mpt)
13223 {
13224     mptsas_rem_intrs(mpt);
13225 }
13226
13227 */
13228 /* mptsas_add_intrs:
13229  * Register FIXED or MSI interrupts.
13230  */
13231 static int
13232 mptsas_add_intrs(mptsas_t *mpt, int intr_type)
13233 {
13234     dev_info_t      *dip = mpt->m_dip;
13235     int              avail, actual, count = 0;
13236     int              i, flag, ret;
13237
13238     NDBG6(("mptsas_add_intrs:interrupt type 0x%x", intr_type));
13239
13240     /* Get number of interrupts */
13241     ret = ddi_intr_get_nintrs(dip, intr_type, &count);
13242     if ((ret != DDI_SUCCESS) || (count <= 0)) {
13243         mptsas_log(mpt, CE_WARN, "ddi_intr_get_nintrs() failed, "
13244                     "ret %d count %d\n", ret, count);
13245
13246         return (DDI_FAILURE);
13247     }

```

```

13250
13251     /* Get number of available interrupts */
13252     ret = ddi_intr_get_nvavail(dip, intr_type, &avail);
13253     if ((ret != DDI_SUCCESS) || (avail == 0)) {
13254         mptsas_log(mpt, CE_WARN, "ddi_intr_get_nvavail() failed, "
13255                     "ret %d avail %d\n", ret, avail);
13256
13257         return (DDI_FAILURE);
13258     }
13259
13260     if (avail < count) {
13261         mptsas_log(mpt, CE_NOTE, "ddi_intr_get_nvavail returned %d, "
13262                     "nvavail() returned %d", count, avail);
13263     }
13264
13265     /* Mpt only have one interrupt routine */
13266     if ((intr_type == DDI_INTR_TYPE_MSI) && (count > 1)) {
13267         count = 1;
13268     }
13269
13270     /* Allocate an array of interrupt handles */
13271     mpt->m_intr_size = count * sizeof (ddi_intr_handle_t);
13272     mpt->m_htable = kmem_alloc(mpt->m_intr_size, KM_SLEEP);
13273
13274     flag = DDI_INTR_ALLOC_NORMAL;
13275
13276     /* call ddi_intr_alloc() */
13277     ret = ddi_intr_alloc(dip, mpt->m_htable, intr_type, 0,
13278                          count, &actual, flag);
13279
13280     if ((ret != DDI_SUCCESS) || (actual == 0)) {
13281         mptsas_log(mpt, CE_WARN, "ddi_intr_alloc() failed, ret %d\n",
13282                     ret);
13283         kmem_free(mpt->m_htable, mpt->m_intr_size);
13284         return (DDI_FAILURE);
13285     }
13286
13287     /* use interrupt count returned or abort? */
13288     if (actual < count) {
13289         mptsas_log(mpt, CE_NOTE, "Requested: %d, Received: %d\n",
13290                     count, actual);
13291     }
13292
13293     mpt->m_intr_cnt = actual;
13294
13295     /*
13296      * Get priority for first msi, assume remaining are all the same
13297      */
13298     if ((ret = ddi_intr_get_pri(mpt->m_htable[0],
13299                               &mpt->m_intr_pri)) != DDI_SUCCESS) {
13300         mptsas_log(mpt, CE_WARN, "ddi_intr_get_pri() failed %d\n", ret);
13301
13302         /* Free already allocated intr */
13303         for (i = 0; i < actual; i++) {
13304             (void) ddi_intr_free(mpt->m_htable[i]);
13305         }
13306
13307         kmem_free(mpt->m_htable, mpt->m_intr_size);
13308         return (DDI_FAILURE);
13309     }
13310
13311     /* Test for high level mutex */
13312     if (mpt->m_intr_pri >= ddi_intr_get_hilevel_pri()) {
13313         mptsas_log(mpt, CE_WARN, "mptsas_add_intrs: "
13314                     "Hi level interrupt not supported\n");

```

```

13315     /* Free already allocated intr */
13316     for (i = 0; i < actual; i++) {
13317         (void) ddi_intr_free(mpt->m_htable[i]);
13318     }
13319
13320     kmem_free(mpt->m_htable, mpt->m_intr_size);
13321     return (DDI_FAILURE);
13322 }
13323
13324 /* Call ddi_intr_add_handler() */
13325 for (i = 0; i < actual; i++) {
13326     if ((ret = ddi_intr_add_handler(mpt->m_htable[i], mptsas_intr,
13327         (caddr_t)mpt, (caddr_t)(uintptr_t)i)) != DDI_SUCCESS) {
13328         mptsas_log(mpt, CE_WARN, "ddi_intr_add_handler() "
13329                     "failed %d\n", ret);
13330
13331     /* Free already allocated intr */
13332     for (i = 0; i < actual; i++) {
13333         (void) ddi_intr_free(mpt->m_htable[i]);
13334     }
13335
13336     kmem_free(mpt->m_htable, mpt->m_intr_size);
13337     return (DDI_FAILURE);
13338 }
13339
13340 if ((ret = ddi_intr_get_cap(mpt->m_htable[0], &mpt->m_intr_cap))
13341     != DDI_SUCCESS) {
13342     mptsas_log(mpt, CE_WARN, "ddi_intr_get_cap() failed %d\n", ret);
13343
13344     /* Free already allocated intr */
13345     for (i = 0; i < actual; i++) {
13346         (void) ddi_intr_free(mpt->m_htable[i]);
13347     }
13348
13349     kmem_free(mpt->m_htable, mpt->m_intr_size);
13350     return (DDI_FAILURE);
13351 }
13352
13353 /* Enable interrupts
13354 */
13355 if (mpt->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
13356     /* Call ddi_intr_block_enable() for MSI interrupts */
13357     (void) ddi_intr_block_enable(mpt->m_htable, mpt->m_intr_cnt);
13358 } else {
13359     /* Call ddi_intr_enable for MSI or FIXED interrupts */
13360     for (i = 0; i < mpt->m_intr_cnt; i++) {
13361         (void) ddi_intr_enable(mpt->m_htable[i]);
13362     }
13363 }
13364
13365 return (DDI_SUCCESS);
13366
13367 }
13368
13369 */
13370 * mptsas_rem_intrs:
13371 *
13372 * Unregister FIXED or MSI interrupts
13373 */
13374 static void
13375 mptsas_rem_intrs(mptsas_t *mpt)
13376 {
13377     int      i;
13378
13379     NDBG6(("mptsas_rem_intrs"));

```

```

13381     /* Disable all interrupts */
13382     if (mpt->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
13383         /* Call ddi_intr_block_disable() */
13384         (void) ddi_intr_block_disable(mpt->m_htable, mpt->m_intr_cnt);
13385     } else {
13386         for (i = 0; i < mpt->m_intr_cnt; i++) {
13387             (void) ddi_intr_disable(mpt->m_htable[i]);
13388         }
13389     }
13390
13391     /* Call ddi_intr_remove_handler() */
13392     for (i = 0; i < mpt->m_intr_cnt; i++) {
13393         (void) ddi_intr_remove_handler(mpt->m_htable[i]);
13394         (void) ddi_intr_free(mpt->m_htable[i]);
13395     }
13396
13397     kmem_free(mpt->m_htable, mpt->m_intr_size);
13398 }
13399
13400 /*
13401 * The IO fault service error handling callback function
13402 */
13403 /*ARGSUSED*/
13404 static int
13405 mptsas_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
13406 {
13407     /*
13408     * as the driver can always deal with an error in any dma or
13409     * access handle, we can just return the fme_status value.
13410     */
13411     pci_ereport_post(dip, err, NULL);
13412     return (err->fme_status);
13413 }
13414
13415 /*
13416 * mptsas_fm_init - initialize fma capabilities and register with IO
13417 *                      fault services.
13418 */
13419 static void
13420 mptsas_fm_init(mptsas_t *mpt)
13421 {
13422     /*
13423     * Need to change iblock to priority for new MSI intr
13424     */
13425     ddi_iblock_cookie_t    fm_ibc;
13426
13427     /* Only register with IO Fault Services if we have some capability */
13428     if (mpt->m_fm_capabilities) {
13429         /* Adjust access and dma attributes for FMA */
13430         mpt->m_reg_acc_attr.devacc_attr_access = DDI_FLAGERR_ACC;
13431         mpt->m_msg_dma_attr.dma_attr_flags |= DDI_DMA_FLAGERR;
13432         mpt->m_io_dma_attr.dma_attr_flags |= DDI_DMA_FLAGERR;
13433
13434         /*
13435         * Register capabilities with IO Fault Services.
13436         * mpt->m_fm_capabilities will be updated to indicate
13437         * capabilities actually supported (not requested.)
13438         */
13439         ddi_fm_init(mpt->m_dip, &mpt->m_fm_capabilities, &fm_ibc);
13440
13441         /*
13442         * Initialize pci ereport capabilities if ereport
13443         * capable (should always be.)
13444         */
13445         if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities) ||
13446             DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {

```

```

13447             pci_ereport_setup(mpt->m_dip);
13448         }
13449
13450         /*
13451          * Register error callback if error callback capable.
13452          */
13453         if (DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13454             ddi_fm_handler_register(mpt->m_dip,
13455                         mptsas_fm_error_cb, (void *) mpt);
13456         }
13457     }
13458 }
13459
13460 /**
13461  * mptsas_fm_fini - Releases fma capabilities and un-registers with IO
13462  * fault services.
13463  */
13464 */
13465 static void
13466 mptsas_fm_fini(mptsas_t *mpt)
13467 {
13468     /* Only unregister FMA capabilities if registered */
13469     if (mpt->m_fm_capabilities) {
13470
13471         /*
13472          * Un-register error callback if error callback capable.
13473          */
13474
13475         if (DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13476             ddi_fm_handler_unregister(mpt->m_dip);
13477         }
13478
13479         /*
13480          * Release any resources allocated by pci_ereport_setup()
13481          */
13482
13483         if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities) ||
13484             DDI_FM_ERRCB_CAP(mpt->m_fm_capabilities)) {
13485             pci_ereport_teardown(mpt->m_dip);
13486         }
13487
13488         /* Unregister from IO Fault Services */
13489         ddi_fm_fini(mpt->m_dip);
13490
13491         /*Adjust access and dma attributes for FMA */
13492         mpt->m_reg_acc_attr.devacc_attr_access = DDI_DEFAULT_ACC;
13493         mpt->m_msg_dma_attr.dma_attr_flags &= ~DDI_DMA_FLAGERR;
13494         mpt->m_io_dma_attr.dma_attr_flags &= ~DDI_DMA_FLAGERR;
13495     }
13496 }
13497
13498 int
13499 mptsas_check_acc_handle(ddi_acc_handle_t handle)
13500 {
13501     ddi_fm_error_t de;
13502
13503     if (handle == NULL)
13504         return (DDI_FAILURE);
13505     ddi_fm_acc_err_get(handle, &de, DDI_FME_VERO);
13506     return (de.fme_status);
13507 }
13508
13509 int
13510 mptsas_check_dma_handle(ddi_dma_handle_t handle)
13511 {

```

```

13512             ddi_fm_error_t de;
13513
13514     if (handle == NULL)
13515         return (DDI_FAILURE);
13516     ddi_fm_dma_err_get(handle, &de, DDI_FME_VERO);
13517     return (de.fme_status);
13518 }
13519
13520 void
13521 mptsas_fm_ereport(mptsas_t *mpt, char *detail)
13522 {
13523     uint64_t ena;
13524     char buf[FM_MAX_CLASS];
13525
13526     (void) sprintf(buf, FM_MAX_CLASS, "%s.%s", DDI_FM_DEVICE, detail);
13527     ena = fm_ena_generate(0, FM_ENA_FMT1);
13528
13529     if (DDI_FM_EREPORT_CAP(mpt->m_fm_capabilities)) {
13530         ddi_fm_ereport_post(mpt->m_dip, buf, ena, DDI_NOSLEEP,
13531                             FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERS0, NULL);
13532     }
13533 }
13534
13535 static int
13536 mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
13537                                uint16_t *dev_handle, mptsas_target_t **pptgt)
13538 {
13539     int rval;
13540     uint32_t dev_info;
13541     uint64_t sas_wwn;
13542     mptsas_phymask_t phymask;
13543     uint8_t physport, phynum, config, disk;
13544     uint64_t devicename;
13545     uint16_t pdev_hdl;
13546     mptsas_target_t *tmp_tgt = NULL;
13547     uint16_t bay_num, enclosure, io_flags;
13548
13549     ASSERT(*pptgt == NULL);
13550
13551     rval = mptsas_get_sas_device_page0(mpt, page_address, dev_handle,
13552                                         &sas_wwn, &dev_info, &physport, &phynum, &pdev_hdl,
13553                                         &bay_num, &enclosure, &io_flags);
13554     if (rval != DDI_SUCCESS) {
13555         rval = DEV_INFO_FAIL_PAGE0;
13556         return (rval);
13557     }
13558
13559     if ((dev_info & (MPI2_SAS_DEVICE_INFO_SSP_TARGET |
13560                      MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
13561                      MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) == NULL) {
13562         rval = DEV_INFO_WRONG_DEVICE_TYPE;
13563         return (rval);
13564     }
13565
13566     /*
13567      * Check if the dev handle is for a Phys Disk. If so, set return value
13568      * and exit. Don't add Phys Disks to hash.
13569      */
13570     for (config = 0; config < mpt->m_num_raid_configs; config++) {
13571         for (disk = 0; disk < MPTSA_MAX_DISKS_IN_CONFIG; disk++) {
13572             if (*dev_handle == mpt->m_raidconfig[config].m_physdisk_devhdl[disk]) {
13573                 rval = DEV_INFO_PHYS_DISK;
13574                 return (rval);
13575             }
13576         }
13577     }
13578 }

```

```

13580
13581     /*
13582      * Get SATA Device Name from SAS device page0 for
13583      * sata device, if device name doesn't exist, set mta_wwn to
13584      * 0 for direct attached SATA. For the device behind the expander
13585      * we still can use STP address assigned by expander.
13586     */
13587     if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
13588         MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
13589         mutex_exit(&mpt->m_mutex);
13590         /* alloc a tmp_tgt to send the cmd */
13591         tmp_tgt = kmalloc(sizeof (struct mptsas_target),
13592             KM_SLEEP);
13593         tmp_tgt->m_devhdl = *dev_handle;
13594         tmp_tgt->m_deviceinfo = dev_info;
13595         tmp_tgt->m_qfull_retries = QFULL_RETRIES;
13596         tmp_tgt->m_qfull_retry_interval =
13597             drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
13598         tmp_tgt->m_t_throttle = MAX_THROTTLE;
13599         devicename = mptsas_get_sata_guid(mpt, tmp_tgt, 0);
13600         kmem_free(tmp_tgt, sizeof (struct mptsas_target));
13601         mutex_enter(&mpt->m_mutex);
13602         if (devicename != 0 && (((devicename >> 56) & 0xf0) == 0x50)) {
13603             sas_wwn = devicename;
13604         } else if (dev_info & MPI2_SAS_DEVICE_INFO_DIRECT_ATTACH) {
13605             sas_wwn = 0;
13606         }
13607     }

13608     phymask = mptsas_physport_to_phymask(mpt, physport);
13609     *pptgt = mptsas_tgt_alloc(mpt, *dev_handle, sas_wwn,
13610         dev_info, phymask, phynum);
13611     if (*pptgt == NULL) {
13612         mptsas_log(mpt, CE_WARN, "Failed to allocated target"
13613             "structure!");
13614         rval = DEV_INFO_FAIL_ALLOC;
13615         return (rval);
13616     }
13617     (*pptgt)->m_io_flags = io_flags;
13618     (*pptgt)->m_enclosure = enclosure;
13619     (*pptgt)->m_slot_num = bay_num;
13620     return (DEV_INFO_SUCCESS);
13621 }

13622 uint64_t
13623 mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun)
13624 {
13625     uint64_t      sata_guid = 0, *pwnn = NULL;
13626     int          target = ptgt->m_devhdl;
13627     uchar_t      *inq83 = NULL;
13628     uchar_t      inq83_len = 0xFF;
13629     uchar_t      *dblk = NULL;
13630     int          inq83_retry = 3;
13631     int          rval = DDI_FAILURE;
13632
13633     inq83 = kmalloc(inq83_len, KM_SLEEP);
13634
13635     inq83_retry:
13636     rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
13637         inq83_len, NULL, 1);
13638     if (rval != DDI_SUCCESS) {
13639         mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
13640             "0x83 for target:%x, lun:%x failed!", target, lun);
13641         goto out;
13642     }
13643     /* According to SAT2, the first descriptor is logic unit name */

```

```

13645     dblk = &inq83[4];
13646     if ((dblk[1] & 0x30) != 0) {
13647         mptsas_log(mpt, CE_WARN, "!Descriptor is not lun associated.");
13648         goto out;
13649     }
13650     pwnn = (uint64_t *) (void *) (&dblk[4]);
13651     if ((dblk[4] & 0xf0) == 0x50) {
13652         sata_guid = BE_64(*pwnn);
13653         goto out;
13654     } else if (dblk[4] == 'A') {
13655         NDBG20(("SATA drive has no NAA format GUID."));
13656         goto out;
13657     } else {
13658         /* The data is not ready, wait and retry */
13659         inq83_retry--;
13660         if (inq83_retry <= 0) {
13661             goto out;
13662         }
13663         NDBG20(("The GUID is not ready, retry..."));
13664         delay(1 * drv_usectohz(1000000));
13665         goto inq83_retry;
13666     }
13667     out:
13668     kmem_free(inq83, inq83_len);
13669     return (sata_guid);
13670 }

13671 static int
13672 mptsas_inquiry(mptsas_t *mpt, mptsas_target_t *ptgt, int lun, uchar_t page,
13673                 unsigned char *buf, int len, int *reallen, uchar_t evpd)
13674 {
13675     uchar_t          cdb[CDB_GROUP0];
13676     struct scsi_address    ap;
13677     struct buf          *data_bp = NULL;
13678     int                resid = 0;
13679     int                ret = DDI_FAILURE;
13680
13681     ASSERT(len <= 0xffff);
13682
13683     ap.a_target = MPTSAS_INVALID_DEVHDL;
13684     ap.a_lun = (uchar_t)(lun);
13685     ap.a_hba_tran = mpt->m_tran;
13686
13687     data_bp = scsi_alloc_consistent_buf(&ap,
13688                                         (struct buf *)NULL, len, B_READ, NULL_FUNC, NULL);
13689     if (data_bp == NULL) {
13690         return (ret);
13691     }
13692     bzero(cdb, CDB_GROUP0);
13693     cdb[0] = SCMD_INQUIRY;
13694     cdb[1] = evpd;
13695     cdb[2] = page;
13696     cdb[3] = (len & 0xff00) >> 8;
13697     cdb[4] = (len & 0x00ff);
13698     cdb[5] = 0;
13699
13700     ret = mptsas_send_scsi_cmd(mpt, &ap, ptgt, &cdb[0], CDB_GROUP0, data_bp,
13701         &resid);
13702     if (ret == DDI_SUCCESS) {
13703         if (reallen) {
13704             *reallen = len - resid;
13705         }
13706         bcopy((caddr_t) data_bp->b_un.b_addr, buf, len);
13707     }
13708     if (data_bp) {
13709         scsi_free_consistent_buf(data_bp);
13710     }

```

```

13711     }
13712     return (ret);
13713 }

13715 static int
13716 mptsas_send_scsi_cmd(mptsas_t *mpt, struct scsi_address *ap,
13717     mptsas_target_t *ptgt, uchar_t *cdb, int cdblen, struct buf *data_bp,
13718     int *resid)
13719 {
13720     struct scsi_pkt      *pktp = NULL;
13721     scsi_hba_tran_t     *tran_clone = NULL;
13722     mptsas_tgt_private_t *tgt_private = NULL;
13723     int                  ret = DDI_FAILURE;

13725 /*
13726     * scsi_hba_tran_t->tran_tgt_private is used to pass the address
13727     * information to scsi_init_pkt, allocate a scsi_hba_tran structure
13728     * to simulate the cmds from sd
13729 */
13730 tran_clone = kmem_alloc(
13731     sizeof (scsi_hba_tran_t), KM_SLEEP);
13732 if (tran_clone == NULL) {
13733     goto out;
13734 }
13735 bcopy((caddr_t)mpt->m_tran,
13736     (caddr_t)tran_clone, sizeof (scsi_hba_tran_t));
13737 tgt_private = kmem_alloc(
13738     sizeof (mptsas_tgt_private_t), KM_SLEEP);
13739 if (tgt_private == NULL) {
13740     goto out;
13741 }
13742 tgt_private->t_lun = ap->a_lun;
13743 tgt_private->t_private = ptgt;
13744 tran_clone->tran_tgt_private = tgt_private;
13745 ap->a_hba_tran = tran_clone;

13747 pktp = scsi_init_pkt(ap, (struct scsi_pkt *)NULL,
13748     data_bp, cdblen, sizeof (struct scsi_arg_status),
13749     0, PKT_CONSISTENT, NULL, NULL);
13750 if (pktp == NULL) {
13751     goto out;
13752 }
13753 bcopy(cdb, pktp->pkt_cdbp, cdblen);
13754 pktp->pkt_flags = FLAG_NOPARITY;
13755 if (scsi_poll(pktp) < 0) {
13756     goto out;
13757 }
13758 if (((struct scsi_status *)pktp->pkt_scbp)->sts_chk) {
13759     goto out;
13760 }
13761 if (resid != NULL) {
13762     *resid = pktp->pkt_resid;
13763 }

13764 ret = DDI_SUCCESS;
13765 out:
13766     if (pktp) {
13767         scsi_destroy_pkt(pktp);
13768     }
13769     if (tran_clone) {
13770         kmem_free(tran_clone, sizeof (scsi_hba_tran_t));
13771     }
13772     if (tgt_private) {
13773         kmem_free(tgt_private, sizeof (mptsas_tgt_private_t));
13774     }
13775 }
13776 return (ret);

```

```

13777 }
13778 static int
13779 mptsas_parse_address(char *name, uint64_t *wwid, uint8_t *phy, int *lun)
13780 {
13781     char    *cp = NULL;
13782     char    *ptr = NULL;
13783     size_t   s = 0;
13784     char    *wwid_str = NULL;
13785     char    *lun_str = NULL;
13786     long    lunnum;
13787     long    phyid = -1;
13788     int     rc = DDI_FAILURE;

13790     ptr = name;
13791     ASSERT(ptr[0] == 'w' || ptr[0] == 'p');
13792     ptr++;
13793     if ((cp = strchr(ptr, ',')) == NULL) {
13794         return (DDI_FAILURE);
13795     }

13797 wwid_str = kmalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13798 s = (uintptr_t)cp - (uintptr_t)ptr;
13800 bcopy(ptr, wwid_str, s);
13801 wwid_str[s] = '\0';

13803 ptr = ++cp;
13805 if ((cp = strchr(ptr, '\0')) == NULL) {
13806     goto out;
13807 }
13808 lun_str = kmalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13809 s = (uintptr_t)cp - (uintptr_t)ptr;
13811 bcopy(ptr, lun_str, s);
13812 lun_str[s] = '\0';

13814 if (name[0] == 'p') {
13815     rc = ddi strtol(wwid_str, NULL, 0x10, &phyid);
13816 } else {
13817     rc = scsi_wwnstr_to_wwn(wwid_str, wwid);
13818 }
13819 if (rc != DDI_SUCCESS)
13820     goto out;

13822 if (phyid != -1) {
13823     ASSERT(phyid < MPTAS_MAX_PHYS);
13824     *phy = (uint8_t)phyid;
13825 }
13826 rc = ddi strtol(lun_str, NULL, 0x10, &lunnum);
13827 if (rc != 0)
13828     goto out;

13830 *lun = (int)lunnum;
13831 rc = DDI_SUCCESS;
13832 out:
13833     if (wwid_str)
13834         kmem_free(wwid_str, SCSI_MAXNAMELEN);
13835     if (lun_str)
13836         kmem_free(lun_str, SCSI_MAXNAMELEN);
13838     return (rc);
13839 }

13841 /*
13842 * mptsas_parse_smp_name() is to parse sas wwn string

```

```

13843 * which format is "wWWN"
13844 */
13845 static int
13846 mptsas_parse_smp_name(char *name, uint64_t *wwn)
13847 {
13848     char    *ptr = name;
13849
13850     if (*ptr != 'w') {
13851         return (DDI_FAILURE);
13852     }
13853
13854     ptr++;
13855     if (scsi_wnstr_to_wwn(ptr, wwn)) {
13856         return (DDI_FAILURE);
13857     }
13858     return (DDI_SUCCESS);
13859 }
13860
13861 static int
13862 mptsas_bus_config(dev_info_t *pdip, uint_t flag,
13863     ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
13864 {
13865     int          ret = NDI_FAILURE;
13866     int          circ = 0;
13867     int          circ1 = 0;
13868     mptsas_t    *mpt;
13869     char        *ptr = NULL;
13870     char        *devnm = NULL;
13871     uint64_t    wwid = 0;
13872     uint8_t     phy = 0xFF;
13873     int          lun = 0;
13874     uint_t      mflags = flag;
13875     int          bconfig = TRUE;
13876
13877     if (scsi_hba_iport_unit_address(pdip) == 0) {
13878         return (DDI_FAILURE);
13879     }
13880
13881     mpt = DIP2MPT(pdip);
13882     if (!mpt) {
13883         return (DDI_FAILURE);
13884     }
13885     /*
13886     * Hold the nexus across the bus_config
13887     */
13888     ndi_devi_enter(scsi_vhci_dip, &circ);
13889     ndi_devi_enter(pdip, &circ1);
13890     switch (op) {
13891     case BUS_CONFIG_ONE:
13892         /* parse wwid/target name out of name given */
13893         if ((ptr = strchr((char *)arg, '@')) == NULL) {
13894             ret = NDI_FAILURE;
13895             break;
13896         }
13897         ptr++;
13898         if (strncmp((char *)arg, "smp", 3) == 0) {
13899             /*
13900             * This is a SMP target device
13901             */
13902             ret = mptsas_parse_smp_name(ptr, &wwid);
13903             if (ret != DDI_SUCCESS) {
13904                 ret = NDI_FAILURE;
13905                 break;
13906             }
13907             ret = mptsas_config_smp(pdip, wwid, childp);
13908         } else if ((ptr[0] == 'w') || (ptr[0] == 'p')) {

```

```

13909
13910
13911
13912
13913
13914
13915
13916
13917
13918
13919
13920
13921
13922
13923
13924
13925
13926
13927
13928
13929
13930
13931
13932
13933
13934
13935
13936
13937
13938
13939
13940
13941
13942
13943
13944
13945
13946
13947
13948
13949
13950
13951
13952
13953
13954
13955
13956
13957
13958
13959
13960
13961
13962
13963
13964
13965
13966
13967
13968
13969
13970
13971
13972
13973
13974
13975
13976
13977
13978
13979
13980
13981
13982
13983
13984
13985
13986
13987
13988
13989
13990
13991
13992
13993
13994
13995
13996
13997
13998
13999
14000
14001
14002
14003
14004
14005
14006
14007
14008
14009
14010
14011
14012
14013
14014
14015
14016
14017
14018
14019
14020
14021
14022
14023
14024
14025
14026
14027
14028
14029
14030
14031
14032
14033
14034
14035
14036
14037
14038
14039
14040
14041
14042
14043
14044
14045
14046
14047
14048
14049
14050
14051
14052
14053
14054
14055
14056
14057
14058
14059
14060
14061
14062
14063
14064
14065
14066
14067
14068
14069
14070
14071
14072
14073
14074
14075
14076
14077
14078
14079
14080
14081
14082
14083
14084
14085
14086
14087
14088
14089
14090
14091
14092
14093
14094
14095
14096
14097
14098
14099
14100
14101
14102
14103
14104
14105
14106
14107
14108
14109
14110
14111
14112
14113
14114
14115
14116
14117
14118
14119
14120
14121
14122
14123
14124
14125
14126
14127
14128
14129
14130
14131
14132
14133
14134
14135
14136
14137
14138
14139
14140
14141
14142
14143
14144
14145
14146
14147
14148
14149
14150
14151
14152
14153
14154
14155
14156
14157
14158
14159
14160
14161
14162
14163
14164
14165
14166
14167
14168
14169
14170
14171
14172
14173
14174
14175
14176
14177
14178
14179
14180
14181
14182
14183
14184
14185
14186
14187
14188
14189
14190
14191
14192
14193
14194
14195
14196
14197
14198
14199
14200
14201
14202
14203
14204
14205
14206
14207
14208
14209
14210
14211
14212
14213
14214
14215
14216
14217
14218
14219
14220
14221
14222
14223
14224
14225
14226
14227
14228
14229
14230
14231
14232
14233
14234
14235
14236
14237
14238
14239
14240
14241
14242
14243
14244
14245
14246
14247
14248
14249
14250
14251
14252
14253
14254
14255
14256
14257
14258
14259
14260
14261
14262
14263
14264
14265
14266
14267
14268
14269
14270
14271
14272
14273
14274
14275
14276
14277
14278
14279
14280
14281
14282
14283
14284
14285
14286
14287
14288
14289
14290
14291
14292
14293
14294
14295
14296
14297
14298
14299
14300
14301
14302
14303
14304
14305
14306
14307
14308
14309
14310
14311
14312
14313
14314
14315
14316
14317
14318
14319
14320
14321
14322
14323
14324
14325
14326
14327
14328
14329
14330
14331
14332
14333
14334
14335
14336
14337
14338
14339
14340
14341
14342
14343
14344
14345
14346
14347
14348
14349
14350
14351
14352
14353
14354
14355
14356
14357
14358
14359
14360
14361
14362
14363
14364
14365
14366
14367
14368
14369
14370
14371
14372
14373
14374
14375
14376
14377
14378
14379
14380
14381
14382
14383
14384
14385
14386
14387
14388
14389
14390
14391
14392
14393
14394
14395
14396
14397
14398
14399
14400
14401
14402
14403
14404
14405
14406
14407
14408
14409
14410
14411
14412
14413
14414
14415
14416
14417
14418
14419
14420
14421
14422
14423
14424
14425
14426
14427
14428
14429
14430
14431
14432
14433
14434
14435
14436
14437
14438
14439
14440
14441
14442
14443
14444
14445
14446
14447
14448
14449
14450
14451
14452
14453
14454
14455
14456
14457
14458
14459
14460
14461
14462
14463
14464
14465
14466
14467
14468
14469
14470
14471
14472
14473
14474
14475
14476
14477
14478
14479
14480
14481
14482
14483
14484
14485
14486
14487
14488
14489
14490
14491
14492
14493
14494
14495
14496
14497
14498
14499
14500
14501
14502
14503
14504
14505
14506
14507
14508
14509
14510
14511
14512
14513
14514
14515
14516
14517
14518
14519
14520
14521
14522
14523
14524
14525
14526
14527
14528
14529
14530
14531
14532
14533
14534
14535
14536
14537
14538
14539
14540
14541
14542
14543
14544
14545
14546
14547
14548
14549
14550
14551
14552
14553
14554
14555
14556
14557
14558
14559
14560
14561
14562
14563
14564
14565
14566
14567
14568
14569
14570
14571
14572
14573
14574
14575
14576
14577
14578
14579
14580
14581
14582
14583
14584
14585
14586
14587
14588
14589
14590
14591
14592
14593
14594
14595
14596
14597
14598
14599
14600
14601
14602
14603
14604
14605
14606
14607
14608
14609
14610
14611
14612
14613
14614
14615
14616
14617
14618
14619
14620
14621
14622
14623
14624
14625
14626
14627
14628
14629
14630
14631
14632
14633
14634
14635
14636
14637
14638
14639
14640
14641
14642
14643
14644
14645
14646
14647
14648
14649
14650
14651
14652
14653
14654
14655
14656
14657
14658
14659
14660
14661
14662
14663
14664
14665
14666
14667
14668
14669
14670
14671
14672
14673
14674
14675
14676
14677
14678
14679
14680
14681
14682
14683
14684
14685
14686
14687
14688
14689
14690
14691
14692
14693
14694
14695
14696
14697
14698
14699
14700
14701
14702
14703
14704
14705
14706
14707
14708
14709
14710
14711
14712
14713
14714
14715
14716
14717
14718
14719
14720
14721
14722
14723
14724
14725
14726
14727
14728
14729
14730
14731
14732
14733
14734
14735
14736
14737
14738
14739
14740
14741
14742
14743
14744
14745
14746
14747
14748
14749
14750
14751
14752
14753
14754
14755
14756
14757
14758
14759
14760
14761
14762
14763
14764
14765
14766
14767
14768
14769
14770
14771
14772
14773
14774
14775
14776
14777
14778
14779
14780
14781
14782
14783
14784
14785
14786
14787
14788
14789
14790
14791
14792
14793
14794
14795
14796
14797
14798
14799
14800
14801
14802
14803
14804
14805
14806
14807
14808
14809
14810
14811
14812
14813
14814
14815
14816
14817
14818
14819
14820
14821
14822
14823
14824
14825
14826
14827
14828
14829
14830
14831
14832
14833
14834
14835
14836
14837
14838
14839
14840
14841
14842
14843
14844
14845
14846
14847
14848
14849
14850
14851
14852
14853
14854
14855
14856
14857
14858
14859
14860
14861
14862
14863
14864
14865
14866
14867
14868
14869
14870
14871
14872
14873
14874
14875
14876
14877
14878
14879
14880
14881
14882
14883
14884
14885
14886
14887
14888
14889
14890
14891
14892
14893
14894
14895
14896
14897
14898
14899
14900
14901
14902
14903
14904
14905
14906
14907
14908
14909
14910
14911
14912
14913
14914
14915
14916
14917
14918
14919
14920
14921
14922
14923
14924
14925
14926
14927
14928
14929
14930
14931
14932
14933
14934
14935
14936
14937
14938
14939
14940
14941
14942
14943
14944
14945
14946
14947
14948
14949
14950
14951
14952
14953
14954
14955
14956
14957
14958
14959
14960
14961
14962
14963
14964
14965
14966
14967
14968
14969
14970
14971
14972
14973
14974
14975
14976
14977
14978
14979
14980
14981
14982
14983
14984
14985
14986
14987
14988
14989
14990
14991
14992
14993
14994
14995
14996
14997
14998
14999
14999
15000
15001
15002
15003
15004
15005
15006
15007
15008
15009
150010
150011
150012
150013
150014
150015
150016
150017
150018
150019
150020
150021
150022
150023
150024
150025
150026
150027
150028
150029
150030
150031
150032
150033
150034
150035
150036
150037
150038
150039
150040
150041
150042
150043
150044
150045
150046
150047
150048
150049
150050
150051
150052
150053
150054
150055
150056
150057
150058
150059
150060
150061
150062
150063
150064
150065
150066
150067
150068
150069
150070
150071
150072
150073
150074
150075
150076
150077
150078
150079
150080
150081
150082
150083
150084
150085
150086
150087
150088
150089
150090
150091
150092
150093
150094
150095
150096
150097
150098
150099
1500100
1500101
1500102
1500103
1500104
1500105
1500106
1500107
1500108
1500109
1500110
1500111
1500112
1500113
1500114
1500115
1500116
1500117
1500118
1500119
1500120
1500121
1500122
1500123
1500124
1500125
1500126
1500127
1500128
1500129
1500130
1500131
1500132
1500133
1500134
1500135
1500136
1500137
1500138
1500139
1500140
1500141
1500142
1500143
1500144
1500145
1500146
1500147
1500148
1500149
1500150
1500151
1500152
1500153
1500154
1500155
1500156
1500157
1500158
1500159
1500160
1500161
1500162
1500163
1500164
1500165
1500166
1500167
1500168
1500169
1500170
1500171
1500172
1500173
1500174
1500175
1500176
1500177
1500178
1500179
1500180
1500181
1500182
1500183
1500184
1500185
1500186
1500187
1500188
1500189
1500190
1500191
1500192
1500193
1500194
1500195
1500196
1500197
1500198
1500199
1500200
1500201
1500202
1500203
1500204
1500205
1500206
1500207
1500208
1500209
1500210
1500211
1500212
1500213
1500214
1500215
1500216
1500217
1500218
1500219
1500220
1500221
1500222
1500223
1500224
1500225
1500226
1500227
1500228
1500229
1500230
1500231
1500232
1500233
1500234
1500235
1500236
1500237
1500238
1500239
1500240
1500241
1500242
1500243
1500244
1500245
1500246
1500247
1500248
1500249
1500250
1500251
1500252
1500253
1500254
1500255
1500256
1500257
1500258
1500259
1500260
1500261
1500262
1500263
1500264
1500265
1500266
1500267
1500268
1500269
1500270
1500271
1500272
1500273
1500274
1500275
1500276
1500277
1500278
1500279
1500280
1500281
1500282
1500283
1500284
1500285
1500286
1500287
1500288
1500289
1500290
1500291
1500292
1500293
1500294
1500295
1500296
1500297
1500298
1500299
1500300
1500301
1500302
1500303
1500304
1500305
1500306
1500307
1500308
1500309
1500310
1500311
1500312
1500313
1500314
1500315
1500316
1500317
1500318
1500319
1500320
1500321
1500322
1500323
1500324
1500325
1500326
1500327
1500328
1500329
1500330
1500331
1500332
1500333
1500334
1500335
1500336
1500337
1500338
1500339
1500340
1500341
1500342
1500343
1500344
1500345
1500346
1500347
1500348
1500349
1500350
1500351
1500352
1500353
1500354
1500355
1500356
1500357
1500358
1500359
1500360
1500361
1500362
1500363
1500364
1500365
1500366
1500367
1500368
1500369
1500370
1500371
1500372
1500373
1500374
1500375
1500376
1500377
1500378
1500379
1500380
1500381
1500382
1500383
1500384
1500385
1500386
1500387
1500388
1500389
1500390
1500391
1500392
1500393
1500394
1500395
1500396
1500397
1500398
1500399
1500400
1500401
1500402
1500403
1500404
1500405
1500406
1500407
1500408
1500409
1500410
1500411
1500412
1500413
1500414
1500415
1500416
1500417
1500418
1500419
1500420
1500421
1500422
1500423
1500424
1500425
1500426
1500427
1500428
1500429
1500430
1500431
1500432
1500433
1500434
1500435
1500436
1500437
1500438
1500439
1500440
1500441
1500442
1500443
1500444
1500445
1500446
1500447
1500448
1500449
1500450
1500451
1500452
1500453
1500454
1500455
1500456
1500457
1500458
1500459
1500460
1500461
1500462
1500463
1500464
1500465
1500466
1500467
1500468
1500469
1500470
1500471
1500472
1500473
1500474
1500475
1500476
1500477
1500478
1500479
1500480
1500481
1500482
1500483
15
```

```

13975         kmem_free(devnm, SCSI_MAXNAMELEN);
13976     return (ret);
13977 }

13979 static int
13980 mptsas_probe_lun(dev_info_t *pdip, int lun, dev_info_t **dip,
13981     mptsas_target_t *ptgt)
13982 {
13983     int             rval = DDI_FAILURE;
13984     struct scsi_inquiry      *sd_inq = NULL;
13985     mptsas_t          *mpt = DIP2MPT(pdip);
13986
13987     sd_inq = (struct scsi_inquiry *)kmem_alloc(SUN_INQSIZE, KM_SLEEP);
13988
13989     rval = mptsas_inquiry(mpt, ptgt, lun, 0, (uchar_t *)sd_inq,
13990         SUN_INQSIZE, 0, (uchar_t)0);
13991
13992     if ((rval == DDI_SUCCESS) && MPTSAS_VALID_LUN(sd_inq)) {
13993         rval = mptsas_create_lun(pdip, sd_inq, dip, ptgt, lun);
13994     } else {
13995         rval = DDI_FAILURE;
13996     }
13997
13998     kmem_free(sd_inq, SUN_INQSIZE);
13999     return (rval);
14000 }

14002 static int
14003 mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
14004     dev_info_t **lundip)
14005 {
14006     int             rval;
14007     mptsas_t          *mpt = DIP2MPT(pdip);
14008     int             phymask;
14009     mptsas_target_t *ptgt = NULL;
14010
14011     /*
14012      * Get the physical port associated to the iport
14013      */
14014     phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14015         "phymask", 0);
14016
14017     ptgt = mptsas_wwid_to_ptgt(mpt, phymask, sasaddr);
14018     if (ptgt == NULL) {
14019         /*
14020          * didn't match any device by searching
14021          */
14022         return (DDI_FAILURE);
14023     }
14024
14025     /*
14026      * If the LUN already exists and the status is online,
14027      * we just return the pointer to dev_info_t directly.
14028      * For the mdi_pathinfo node, we'll handle it in
14029      * mptsas_create_virt_lun()
14030      * TODO should be also in mptsas_handle_dr
14031
14032     *lundip = mptsas_find_child_addr(pdip, sasaddr, lun);
14033     if (*lundip != NULL) {
14034         /*
14035          * TODO Another scenario is, we hotplug the same disk
14036          * on the same slot, the devhdl changed, is this
14037          * possible?
14038          * tgt_private->t_private != ptgt
14039          */
14040         if (sasaddr != ptgt->m_addr.mta_wwn) {

```

```

14041         /*
14042          * The device has changed although the devhdl is the
14043          * same (Enclosure mapping mode, change drive on the
14044          * same slot)
14045          */
14046         return (DDI_FAILURE);
14047     }
14048     return (DDI_SUCCESS);
14049 }

14051     if (phymask == 0) {
14052         /*
14053          * Configure IR volume
14054          */
14055         rval = mptsas_config_raid(pdip, ptgt->m_devhdl, lundip);
14056         return (rval);
14057     }
14058     rval = mptsas_probe_lun(pdip, lun, lundip, ptgt);
14059
14060     return (rval);
14061 }

14063 static int
14064 mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
14065     dev_info_t **lundip)
14066 {
14067     int             rval;
14068     mptsas_t          *mpt = DIP2MPT(pdip);
14069     mptsas_phymask_t phymask;
14070     mptsas_target_t *ptgt = NULL;
14071
14072     /*
14073      * Get the physical port associated to the iport
14074      */
14075     phymask = (mptsas_phymask_t)ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14076         "phymask", 0);
14077
14078     ptgt = mptsas_phy_to_tgt(mpt, phymask, phy);
14079     if (ptgt == NULL) {
14080         /*
14081          * didn't match any device by searching
14082          */
14083         return (DDI_FAILURE);
14084     }
14085
14086     /*
14087      * If the LUN already exists and the status is online,
14088      * we just return the pointer to dev_info_t directly.
14089      * For the mdi_pathinfo node, we'll handle it in
14090      * mptsas_create_virt_lun().
14091      */
14092
14093     *lundip = mptsas_find_child_phy(pdip, phy);
14094     if (*lundip != NULL) {
14095         return (DDI_SUCCESS);
14096     }
14097
14098     rval = mptsas_probe_lun(pdip, lun, lundip, ptgt);
14099
14100     return (rval);
14101 }

14103 static int
14104 mptsas_retrieve_lundata(int lun_cnt, uint8_t *buf, uint16_t *lun_num,
14105     uint8_t *lun_addr_type)
14106 {

```

```

14107     uint32_t      lun_idx = 0;
14109     ASSERT(lun_num != NULL);
14110     ASSERT(lun_addr_type != NULL);
14112     lun_idx = (lun_cnt + 1) * MPTSAS_SCSI_REPORTLUNS_ADDRESS_SIZE;
14113     /* determine report luns addressing type */
14114     switch (buf[lun_idx] & MPTSAS_SCSI_REPORTLUNS_ADDRESS_MASK) {
14115         /*
14116             * Vendors in the field have been found to be concatenating
14117             * bus/target/lun to equal the complete lun value instead
14118             * of switching to flat space addressing
14119             */
14120             /* 00b - peripheral device addressing method */
14121             case MPTSAS_SCSI_REPORTLUNS_ADDRESS_PERIPHERAL:
14122                 /* FALLTHRU */
14123             /* 10b - logical unit addressing method */
14124             case MPTSAS_SCSI_REPORTLUNS_ADDRESS_LOGICAL_UNIT:
14125                 /* FALLTHRU */
14126             /* 01b - flat space addressing method */
14127             case MPTSAS_SCSI_REPORTLUNS_ADDRESS_FLAT_SPACE:
14128                 /* byte0 bit0-5=msb lun byte1 bit0-7=lsb lun */
14129                 *lun_addr_type = (buf[lun_idx] &
14130                     MPTSAS_SCSI_REPORTLUNS_ADDRESS_MASK) >> 6;
14131                 *lun_num = (buf[lun_idx] & 0x3F) << 8;
14132                 *lun_num |= buf[lun_idx + 1];
14133                 return (DDI_SUCCESS);
14134             default:
14135                 return (DDI_FAILURE);
14136         }
14137     }

14138 static int
14139 mptsas_config_luns(dev_info_t *pdip, mptsas_target_t *ptgt)
14140 {
14141     struct buf           *repluns_bp = NULL;
14142     struct scsi_address   ap;
14143     uchar_t               cdb[CDB_GROUP5];
14144     int                   ret = DDI_FAILURE;
14145     int                   retry = 0;
14146     int                   lun_list_len = 0;
14147     int                   lun_num = 0;
14148     uint16_t              lun_addr_type = 0;
14149     uint8_t               lun_cnt = 0;
14150     uint32_t              lun_total = 0;
14151     dev_info_t            *cdip = NULL;
14152     uint16_t              *saved_repluns = NULL;
14153     char                  *buffer = NULL;
14154     int                   buf_len = 128;
14155     mptsas_t              *mpt = DIP2MPT(pdip);
14156     uint64_t              sas_wwn = 0;
14157     uint8_t               phy = 0xFF;
14158     uint32_t              dev_info = 0;
14159

14160     mutex_enter(&mpt->m_mutex);
14161     sas_wwn = ptgt->m_addr.mta_wwn;
14162     phy = ptgt->m_phynum;
14163     dev_info = ptgt->m_deviceinfo;
14164     mutex_exit(&mpt->m_mutex);
14165

14166     if (sas_wwn == 0) {
14167         /*
14168             * It's a SATA without Device Name
14169             * So don't try multi-LUNs
14170             */
14171         if (mptsas_find_child_phy(pdip, phy)) {
14172

```

```

14173             return (DDI_SUCCESS);
14174         } else {
14175             /*
14176                 * need configure and create node
14177                 */
14178             return (DDI_FAILURE);
14179         }
14180     }

14181     /*
14182         * WWN (SAS address or Device Name exist)
14183         */
14184     if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14185         MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14186         /*
14187             * SATA device with Device Name
14188             * So don't try multi-LUNs
14189             */
14190         if (mptsas_find_child_addr(pdip, sas_wwn, 0)) {
14191             return (DDI_SUCCESS);
14192         } else {
14193             return (DDI_FAILURE);
14194         }
14195     }

14196     do {
14197         ap.a_target = MPTSAS_INVALID_DEVHDL;
14198         ap.a_lun = 0;
14199         ap.a_hba_tran = mpt->m_tran;
14200         repluns_bp = scsi_alloc_consistent_buf(&ap,
14201             (struct buf *)NULL, buf_len, B_READ, NULL_FUNC, NULL);
14202         if (repluns_bp == NULL) {
14203             retry++;
14204             continue;
14205         }
14206         bzero(cdb, CDB_GROUP5);
14207         cdb[0] = SCMD_REPORT_LUNS;
14208         cdb[6] = (buf_len & 0xffff0000) >> 24;
14209         cdb[7] = (buf_len & 0x0ff0000) >> 16;
14210         cdb[8] = (buf_len & 0x0000ff00) >> 8;
14211         cdb[9] = (buf_len & 0x000000ff);

14212         ret = mptsas_send_scsi_cmd(mpt, &ap, ptgt, &cdb[0], CDB_GROUP5,
14213             repluns_bp, NULL);
14214         if (ret != DDI_SUCCESS) {
14215             scsi_free_consistent_buf(repluns_bp);
14216             retry++;
14217             continue;
14218         }
14219         lun_list_len = BE_32(*((int *)((void *)(
14220             repluns_bp->b_un.b_addr))));
14221         if (buf_len >= lun_list_len + 8) {
14222             ret = DDI_SUCCESS;
14223             break;
14224         }
14225         scsi_free_consistent_buf(repluns_bp);
14226         buf_len = lun_list_len + 8;
14227
14228     } while (retry < 3);

14229     if (ret != DDI_SUCCESS)
14230         return (ret);
14231     buffer = (char *)repluns_bp->b_un.b_addr;
14232     /*
14233         * find out the number of luns returned by the SCSI ReportLun call
14234         * and allocate buffer space
14235
14236
14237
14238
14239
14240
14241
14242
14243
14244
14245
14246
14247
14248
14249
14250
14251
14252
14253
14254
14255
14256
14257
14258
14259
14260
14261
14262
14263
14264
14265
14266
14267
14268
14269
14270
14271
14272
14273
14274
14275
14276
14277
14278
14279
14280
14281
14282
14283
14284
14285
14286
14287
14288
14289
14290
14291
14292
14293
14294
14295
14296
14297
14298
14299
14300
14301
14302
14303
14304
14305
14306
14307
14308
14309
14310
14311
14312
14313
14314
14315
14316
14317
14318
14319
14320
14321
14322
14323
14324
14325
14326
14327
14328
14329
14330
14331
14332
14333
14334
14335
14336
14337
14338
14339
14340
14341
14342
14343
14344
14345
14346
14347
14348
14349
14350
14351
14352
14353
14354
14355
14356
14357
14358
14359
14360
14361
14362
14363
14364
14365
14366
14367
14368
14369
14370
14371
14372
14373
14374
14375
14376
14377
14378
14379
14380
14381
14382
14383
14384
14385
14386
14387
14388
14389
14390
14391
14392
14393
14394
14395
14396
14397
14398
14399
14400
14401
14402
14403
14404
14405
14406
14407
14408
14409
14410
14411
14412
14413
14414
14415
14416
14417
14418
14419
14420
14421
14422
14423
14424
14425
14426
14427
14428
14429
14430
14431
14432
14433
14434
14435
14436
14437
14438
14439
14440
14441
14442
14443
14444
14445
14446
14447
14448
14449
14450
14451
14452
14453
14454
14455
14456
14457
14458
14459
14460
14461
14462
14463
14464
14465
14466
14467
14468
14469
14470
14471
14472
14473
14474
14475
14476
14477
14478
14479
14480
14481
14482
14483
14484
14485
14486
14487
14488
14489
14490
14491
14492
14493
14494
14495
14496
14497
14498
14499
14500
14501
14502
14503
14504
14505
14506
14507
14508
14509
14510
14511
14512
14513
14514
14515
14516
14517
14518
14519
14520
14521
14522
14523
14524
14525
14526
14527
14528
14529
14530
14531
14532
14533
14534
14535
14536
14537
14538
14539
14540
14541
14542
14543
14544
14545
14546
14547
14548
14549
14550
14551
14552
14553
14554
14555
14556
14557
14558
14559
14560
14561
14562
14563
14564
14565
14566
14567
14568
14569
14570
14571
14572
14573
14574
14575
14576
14577
14578
14579
14580
14581
14582
14583
14584
14585
14586
14587
14588
14589
14590
14591
14592
14593
14594
14595
14596
14597
14598
14599
14600
14601
14602
14603
14604
14605
14606
14607
14608
14609
14610
14611
14612
14613
14614
14615
14616
14617
14618
14619
14620
14621
14622
14623
14624
14625
14626
14627
14628
14629
14630
14631
14632
14633
14634
14635
14636
14637
14638
14639
14640
14641
14642
14643
14644
14645
14646
14647
14648
14649
14650
14651
14652
14653
14654
14655
14656
14657
14658
14659
14660
14661
14662
14663
14664
14665
14666
14667
14668
14669
14670
14671
14672
14673
14674
14675
14676
14677
14678
14679
14680
14681
14682
14683
14684
14685
14686
14687
14688
14689
14690
14691
14692
14693
14694
14695
14696
14697
14698
14699
14700
14701
14702
14703
14704
14705
14706
14707
14708
14709
14710
14711
14712
14713
14714
14715
14716
14717
14718
14719
14720
14721
14722
14723
14724
14725
14726
14727
14728
14729
14730
14731
14732
14733
14734
14735
14736
14737
14738
14739
14740
14741
14742
14743
14744
14745
14746
14747
14748
14749
14750
14751
14752
14753
14754
14755
14756
14757
14758
14759
14760
14761
14762
14763
14764
14765
14766
14767
14768
14769
14770
14771
14772
14773
14774
14775
14776
14777
14778
14779
14780
14781
14782
14783
14784
14785
14786
14787
14788
14789
14790
14791
14792
14793
14794
14795
14796
14797
14798
14799
14800
14801
14802
14803
14804
14805
14806
14807
14808
14809
14810
14811
14812
14813
14814
14815
14816
14817
14818
14819
14820
14821
14822
14823
14824
14825
14826
14827
14828
14829
14830
14831
14832
14833
14834
14835
14836
14837
14838
14839
14840
14841
14842
14843
14844
14845
14846
14847
14848
14849
14850
14851
14852
14853
14854
14855
14856
14857
14858
14859
14860
14861
14862
14863
14864
14865
14866
14867
14868
14869
14870
14871
14872
14873
14874
14875
14876
14877
14878
14879
14880
14881
14882
14883
14884
14885
14886
14887
14888
14889
14890
14891
14892
14893
14894
14895
14896
14897
14898
14899
14900
14901
14902
14903
14904
14905
14906
14907
14908
14909
14910
14911
14912
14913
14914
14915
14916
14917
14918
14919
14920
14921
14922
14923
14924
14925
14926
14927
14928
14929
14930
14931
14932
14933
14934
14935
14936
14937
14938
14939
14940
14941
14942
14943
14944
14945
14946
14947
14948
14949
14950
14951
14952
14953
14954
14955
14956
14957
14958
14959
14960
14961
14962
14963
14964
14965
14966
14967
14968
14969
14970
14971
14972
14973
14974
14975
14976
14977
14978
14979
14980
14981
14982
14983
14984
14985
14986
14987
14988
14989
14990
14991
14992
14993
14994
14995
14996
14997
14998
14999
15000
15001
15002
15003
15004
15005
15006
15007
15008
15009
15010
15011
15012
15013
15014
15015
15016
15017
15018
15019
15020
15021
15022
15023
15024
15025
15026
15027
15028
15029
15030
15031
15032
15033
15034
15035
15036
15037
15038
15039
15040
15041
15042
15043
15044
15045
15046
15047
15048
15049
15050
15051
15052
15053
15054
15055
15056
15057
15058
15059
15060
15061
15062
15063
15064
15065
15066
15067
15068
15069
15070
15071
15072
15073
15074
15075
15076
15077
15078
15079
15080
15081
15082
15083
15084
15085
15086
15087
15088
15089
15090
15091
15092
15093
15094
15095
15096
15097
15098
15099
15100
15101
15102
15103
15104
15105
15106
15107
15108
15109
15110
15111
15112
15113
15114
15115
15116
15117
15118
15119
15120
15121
15122
15123
15124
15125
15126
15127
15128
15129
15130
15131
15132
15133
15134
15135
15136
15137
15138
15139
15140
15141
15142
15143
15144
15145
15146
15147
15148
15149
15150
15151
15152
15153
15154
15155
15156
15157
15158
15159
15160
15161
15162
15163
15164
15165
15166
15167
15168
15169
15170
15171
15172
15173
15174
15175
15176
15177
15178
15179
15180
15181
15182
15183
15184
15185
15186
15187
15188
15189
15190
15191
15192
15193
15194
15195
15196
15197
15198
15199
15200
15201
15202
15203
15204
15205
15206
15207
15208
15209
15210
15211
15212
15213
15214
15215
15216
15217
15218
15219
15220
15221
15222
15223
15224
15225
15226
15227
15228
15229
15230
15231
15232
15233
15234
15235
15236
15237
15238
15239
15240
15241
15242
15243
15244
15245
15246
15247
15248
15249
15250
15251
15252
15253
15254
15255
15256
15257
15258
15259
15260
15261
15262
15263
15264
15265
15266
15267
15268
15269
15270
15271
15272
15273
15274
15275
15276
15277
15278
15279
15280
15281
15282
15283
15284
15285
15286
15287
15288
15289
15290
15291
15292
15293
15294
15295
15296
15297
15298
15299
15300
15301
15302
15303
15304
15305
15306
15307
15308
15309
15310
15311
15312
15313
15314
15315
15316
15317
15318
15319
15320
15321
15322
15323
15324
15325
15326
15327
15328
15329
15330
15331
15332
15333
15334
15335
15336
15337
15338
15339
15340
15341
15342
15343
15344
15345
15346
15347
15348
15349
15350
15351
15352
15353
15354
15355
15356
15357
15358
15359
15360
15361
15362
15363
15364
15365
15366
15367
15368
15369
15370
15371
15372
15373
15374
15375
15376
15377
15378
15379
15380
15381
15382
15383
15384
15385
15386
15387
15388
15389
15390
15391
15392
15393
15394
15395
15396
15397
15398
15399
15400
15401
15402
15403
15404
15405
15406
15407
15408
15409
15410
15411
15412
15413
15414
15415
15416
15417
15418
15419
15420
15421
15422
15423
15424
15425
15426
15427
15428
15429
15430
15431
15432
15433
15434
15435
15436
15437
15438
15439
15440
15441
15442
15443
15444
15445
15446
15447
15448
15449
15450
15451
15452
15453
15454
15455
15456
15457
15458
15459
15460
15461
15462
15463
15464
15465
15466
15467
15468
15469
15470
15471
15472
15473
15474
15475
15476
15477
15478
15479
15480
15481
15482
15483
15484
15485
15486
15487
15488
15489
15490
15491
15492
15493
15494
15495
15496
15497
15498
15499
15500
15501
15502
15503
15504
15505
15506
15507
15508
15509
15510
15511
15512
15513
15514
15515
15516
15517
15518
15519
15520
15521
15522
15523
15524
15525
15526
15527
15528
15529
15530
15531
15532
15533
15534
15535
15536
15537
15538
15539
15540
15541
15542
15543
15544
15545
15546
15547
15548
15549
15550
15551
15552
15553
15554
15555
15556
15557
15558
15559
15560
15561
15562
15563
15564
15565
15566
15567
15568
15569
15570
15571
15572
15573
15574
15575
15576
15577
15578
15579
15580
15581
15582
15583
15584
15585
15586
15587
15588
15589
15590
15591
15592
15593
15594
15595
15596
15597
15598
15599
15600
15601
15602
15603
15604
15605
15606
15607
15608
15609
15610
15611
15612
15613
15614
15615
15616
15617
15618
15619
15620
15621
15622
15623
15624
15625
15626
15627
15628
15629
15630
15631
15632
15633
15634
15635
15636
15637
15638
15639
15640
15641
15642
15643
15644
15645
15646
15647
15648
15649
15650
15651
15652
15653
15654
15655
15656
15657
15658
15659
15660
15661
15662
15663
15664
15665
15666
15667
15668
15669
15670
15671
15672
15673
15674
15675
15676
15677
15678
15679
15680
15681
15682
15683
15684
15685
15686
15687
15688
15689
15690
15691
15692
15693
15694
15695
15696
15697
15698
15699
15700
15701
15702
15703
15704
15705
15706
15707
15708
15709
15710
15711
15712
15713
15714
15715
15716
15717
15718
15719
15720
15721
15722
15723
15724
15725
15726
15727
15728
15729
15730
15731
15732
15733
15734
15735
15736
15737
15738
15739
15740
15741
15742
15743
15744
15745
15746
15747
15748
15749
15750
15751
15752
15753
15754
15755
15756
15757
15758
15759
15760
15761
15762
15763
15764
15765
15766
15767
15768
15769
15770
15771
15772

```

```

14239 /*
14240 lun_total = lun_list_len / MPTSSAS_SCSI_REPORTLUNS_ADDRESS_SIZE;
14241 saved_repluns = kmem_zalloc(sizeof (uint16_t) * lun_total, KM_SLEEP);
14242 if (saved_repluns == NULL) {
14243     scsi_free_consistent_buf(repluns_bp);
14244     return (DDI_FAILURE);
14245 }
14246 for (lun_cnt = 0; lun_cnt < lun_total; lun_cnt++) {
14247     if (mptsas_retrieve_lundata(lun_cnt, (uint8_t *) (buffer),
14248         &lun_num, &lun_addr_type) != DDI_SUCCESS) {
14249         continue;
14250     }
14251     saved_repluns[lun_cnt] = lun_num;
14252     if (cdip = mptsas_find_child_addr(pdip, sas_wwn, lun_num)) {
14253         ret = DDI_SUCCESS;
14254     } else
14255         ret = mptsas_probe_lun(pdip, lun_num, &cdip,
14256             ptgt);
14257     if ((ret == DDI_SUCCESS) && (cdip != NULL)) {
14258         (void) ndi_prop_remove(DDI_DEV_T_NONE, cdip,
14259             MPTSSAS_DEV_GONE);
14260     }
14261 }
14262 mptsas_offline_missed_luns(pdip, saved_repluns, lun_total, ptgt);
14263 kmem_free(saved_repluns, sizeof (uint16_t) * lun_total);
14264 scsi_free_consistent_buf(repluns_bp);
14265 return (DDI_SUCCESS);
14266 }

14268 static int
14269 mptsas_config_raid(dev_info_t *pdip, uint16_t target, dev_info_t **dip)
14270 {
14271     int rval = DDI_FAILURE;
14272     struct scsi_inquiry *sd_inq = NULL;
14273     mptsas_t *mpt = DIP2MPT(pdip);
14274     mptsas_target_t *ptgt = NULL;

14276     mutex_enter(&mpt->m_mutex);
14277     ptgt = refhash_linear_search(mpt->m_targets,
14278         mptsas_target_eval_devhdl, &target);
14279     mutex_exit(&mpt->m_mutex);
14280     if (ptgt == NULL) {
14281         mptsas_log(mpt, CE_WARN, "Volume with VolDevHandle of 0x%llx",
14282                     "not found.", target);
14283         return (rval);
14284     }

14286     sd_inq = (struct scsi_inquiry *)kmem_alloc(SUN_INQSIZE, KM_SLEEP);
14287     rval = mptsas_inquiry(mpt, ptgt, 0, 0, (uchar_t *)sd_inq,
14288         SUN_INQSIZE, 0, (uchar_t)0);

14290     if ((rval == DDI_SUCCESS) && MPTSSAS_VALID_LUN(sd_inq)) {
14291         rval = mptsas_create_phys_lun(pdip, sd_inq, NULL, dip, ptgt,
14292             0);
14293     } else {
14294         rval = DDI_FAILURE;
14295     }

14297     kmem_free(sd_inq, SUN_INQSIZE);
14298     return (rval);
14299 }

14301 /*
14302  * configure all RAID volumes for virtual iport
14303  */
14304 static void

```

```

14305 mptsas_config_all_viport(dev_info_t *pdip)
14306 {
14307     mptsas_t          *mpt = DIP2MPT(pdip);
14308     int                config, vol;
14309     int                target;
14310     dev_info_t         *lunidp = NULL;
14311
14312     /*
14313      * Get latest RAID info and search for any Volume DevHandles. If any
14314      * are found, configure the volume.
14315      */
14316     mutex_enter(&mpt->m_mutex);
14317     for (config = 0; config < mpt->m_num_raid_configs; config++) {
14318         for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
14319             if (mpt->m_raidconfig[config].m_raidvol[vol].m_israids
14320                 == 1) {
14321                 target = mpt->m_raidconfig[config];
14322                 m_raidvol[vol].m_raidhandle;
14323                 mutex_exit(&mpt->m_mutex);
14324                 (void) mptsas_config_raid(pdip, target,
14325                  &lunidp);
14326                 mutex_enter(&mpt->m_mutex);
14327             }
14328         }
14329     }
14330     mutex_exit(&mpt->m_mutex);
14331 }
14332
14333 static void
14334 mptsas_offline_missed_luns(dev_info_t *pdip, uint16_t *repluns,
14335     int lun_cnt, mptsas_target_t *ptgt)
14336 {
14337     dev_info_t          *child = NULL, *savechild = NULL;
14338     mdi_pathinfo_t      *pip = NULL, *savepip = NULL;
14339     uint64_t             sas_wwn, wwid;
14340     uint8_t              phy;
14341     int                  lun;
14342     int                  i;
14343     int                  find;
14344     char                *addr;
14345     char                *nodename;
14346     mptsas_t             *mpt = DIP2MPT(pdip);
14347
14348     mutex_enter(&mpt->m_mutex);
14349     wwid = ptgt->m_addr.mta_wwn;
14350     mutex_exit(&mpt->m_mutex);
14351
14352     child = ddi_get_child(pdip);
14353     while (child) {
14354         find = 0;
14355         savechild = child;
14356         child = ddi_get_next_sibling(child);
14357
14358         nodename = ddi_node_name(savechild);
14359         if (strcmp(nodename, "smp") == 0) {
14360             continue;
14361         }
14362
14363         addr = ddi_get_name_addr(savechild);
14364         if (addr == NULL) {
14365             continue;
14366         }
14367
14368         if (mptsas_parse_address(addr, &sas_wwn, &phy, &lun) !=
14369             DDI_SUCCESS) {
14370             continue;

```

```

14371         }
14372
14373     if (wwid == sas_wwn) {
14374         for (i = 0; i < lun_cnt; i++) {
14375             if (repluns[i] == lun) {
14376                 find = 1;
14377                 break;
14378             }
14379         } else {
14380             continue;
14381         }
14382     if (find == 0) {
14383         /*
14384             * The lun has not been there already
14385         */
14386         (void) mptsas_offline_lun(pdip, savechild, NULL,
14387             NDI_DEVI_REMOVE);
14388     }
14389 }
14390
14391 pip = mdi_get_next_client_path(pdip, NULL);
14392 while (pip) {
14393     find = 0;
14394     savepip = pip;
14395     addr = MDI_PI(pip)->pi_addr;
14396
14397     pip = mdi_get_next_client_path(pdip, pip);
14398
14399     if (addr == NULL) {
14400         continue;
14401     }
14402
14403     if (mptsas_parse_address(addr, &sas_wwn, &phy,
14404         &lun) != DDI_SUCCESS) {
14405         continue;
14406     }
14407
14408     if (sas_wwn == wwid) {
14409         for (i = 0; i < lun_cnt; i++) {
14410             if (repluns[i] == lun) {
14411                 find = 1;
14412                 break;
14413             }
14414         }
14415     } else {
14416         continue;
14417     }
14418
14419     if (find == 0) {
14420         /*
14421             * The lun has not been there already
14422         */
14423         (void) mptsas_offline_lun(pdip, NULL, savepip,
14424             NDI_DEVI_REMOVE);
14425     }
14426
14427 }
14428 }
14429
14430 void
14431 mptsas_update_hashtab(struct mptsas *mpt)
14432 {
14433     uint32_t      page_address;
14434     int          rval = 0;
14435     uint16_t      dev_handle;
14436     mptsas_target_t *ptgt = NULL;

```

```

14437     mptsas_smp_t    smp_node;
14438
14439     /*
14440         * Get latest RAID info.
14441     */
14442     (void) mptsas_get_raid_info(mpt);
14443
14444     dev_handle = mpt->m_smp_devhdl;
14445     for ( ; mpt->m_done_traverse_smp == 0; ) {
14446         page_address = (MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL &
14447                         MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)dev_handle;
14448         if (mptsas_get_sas_expander_page0(mpt, page_address, &smp_node)
14449             != DDI_SUCCESS) {
14450             break;
14451         }
14452         mpt->m_smp_devhdl = dev_handle = smp_node.m_devhdl;
14453         (void) mptsas_smp_alloc(mpt, &smp_node);
14454     }
14455
14456     /*
14457         * Config target devices
14458     */
14459     dev_handle = mpt->m_dev_handle;
14460
14461     /*
14462         * Do loop to get sas device page 0 by GetNextHandle till the
14463         * the last handle. If the sas device is a SATA/SSP target,
14464         * we try to config it.
14465     */
14466     for ( ; mpt->m_done_traverse_dev == 0; ) {
14467         ptgt = NULL;
14468         page_address =
14469             (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
14470                 MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14471                 (uint32_t)dev_handle;
14472         rval = mptsas_get_target_device_info(mpt, page_address,
14473             &dev_handle, &ptgt);
14474         if ((rval == DEV_INFO_FAIL_PAGE0) ||
14475             (rval == DEV_INFO_FAIL_ALLOC)) {
14476             break;
14477         }
14478         mpt->m_dev_handle = dev_handle;
14479     }
14480
14481
14482 }
14483
14484 void
14485 mptsas_update_driver_data(struct mptsas *mpt)
14486 {
14487     mptsas_target_t *tp;
14488     mptsas_smp_t *sp;
14489
14490     ASSERT(MUTEX_HELD(&mpt->m_mutex));
14491
14492     /*
14493         * TODO after hard reset, update the driver data structures
14494         * 1. update port/phymask mapping table mpt->m_phy_info
14495         * 2. invalid all the entries in hash table
14496         *      m_devhdl = 0xffff and m_deviceinfo = 0
14497         * 3. call sas_device_page/expander_page to update hash table
14498     */
14499     mptsas_update_phymask(mpt);
14500
14501     /*
14502         * Remove all the devhdl's for existing entries but leave their

```

```

14503     * addresses alone. In update_hashtab() below, we'll find all
14504     * targets that are still present and reassociate them with
14505     * their potentially new devhds. Leaving the targets around in
14506     * this fashion allows them to be used on the tx waitq even
14507     * while IOC reset is occurring.
14508     */
14509     for (tp = refhash_first(mpt->m_targets); tp != NULL;
14510         tp = refhash_next(mpt->m_targets, tp)) {
14511         tp->m_devhdl = MPTSAS_INVALID_DEVHDL;
14512         tp->m_deviceinfo = 0;
14513         tp->m_dr_flag = MPTSAS_DR_INACTIVE;
14514     }
14515     for (sp = refhash_first(mpt->m_smp_targets); sp != NULL;
14516         sp = refhash_next(mpt->m_smp_targets, sp)) {
14517         sp->m_devhdl = MPTSAS_INVALID_DEVHDL;
14518         sp->m_deviceinfo = 0;
14519     }
14520     mpt->m_done_traverse_dev = 0;
14521     mpt->m_done_traverse_smp = 0;
14522     mpt->m_dev_handle = mpt->m_smp_devhdl = MPTSAS_INVALID_DEVHDL;
14523     mptsas_update_hashtab(mpt);
14524 }

14525 static void
14526 mptsas_config_all(dev_info_t *pdip)
14527 {
14528     dev_info_t      *smpdip = NULL;
14529     mptsas_t        *mpt = DIP2MPT(pdip);
14530     int             phymask = 0;
14531     mptsas_phymask_t phy_mask;
14532     mptsas_target_t *ptgt = NULL;
14533     mptsas_smp_t    *psmp;
14534

14535     /*
14536     * Get the phymask associated to the iport
14537     */
14538     phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
14539                               "phymask", 0);
14540

14541     /*
14542     * Enumerate RAID volumes here (phymask == 0).
14543     */
14544     if (phymask == 0) {
14545         mptsas_config_all_viport(pdip);
14546         return;
14547     }
14548

14549     mutex_enter(&mpt->m_mutex);

14550     if (!mpt->m_done_traverse_dev || !mpt->m_done_traverse_smp) {
14551         mptsas_update_hashtab(mpt);
14552     }

14553     for (psmp = refhash_first(mpt->m_smp_targets); psmp != NULL;
14554         psmp = refhash_next(mpt->m_smp_targets, psmp)) {
14555         phy_mask = psmp->m_addr.mta_phymask;
14556         if (phy_mask == phymask) {
14557             smpdip = NULL;
14558             mutex_exit(&mpt->m_mutex);
14559             (void) mptsas_online_smp(pdip, psmp, &smpdip);
14560             mutex_enter(&mpt->m_mutex);
14561         }
14562     }

14563     for (ptgt = refhash_first(mpt->m_targets); ptgt != NULL;
14564         ptgt = refhash_next(mpt->m_targets, ptgt)) {

```

```

14565         phy_mask = ptgt->m_addr.mta_phymask;
14566         if (phy_mask == phymask) {
14567             mutex_exit(&mpt->m_mutex);
14568             (void) mptsas_config_target(pdip, ptgt);
14569             mutex_enter(&mpt->m_mutex);
14570         }
14571     }
14572     mutex_exit(&mpt->m_mutex);
14573 }

14574 static int
14575 mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt)
14576 {
14577     int          rval = DDI_FAILURE;
14578     dev_info_t   *tdip;
14579
14580     rval = mptsas_config_luns(pdip, ptgt);
14581     if (rval != DDI_SUCCESS) {
14582         /*
14583         * The return value means the SCMD_REPORT_LUNS
14584         * did not execute successfully. The target maybe
14585         * doesn't support such command.
14586         */
14587         rval = mptsas_probe_lun(pdip, 0, &tdip, ptgt);
14588     }
14589     return (rval);
14590 }

14591 /*
14592 * Return fail if not all the childs/paths are freed.
14593 * if there is any path under the HBA, the return value will be always fail
14594 * because we didn't call mdi_pi_free for path
14595 */
14596 static int
14597 mptsas_offline_target(dev_info_t *pdip, char *name)
14598 {
14599     dev_info_t      *child = NULL, *prechild = NULL;
14600     mdi_pathinfo_t  *pip = NULL, *savepip = NULL;
14601     int             tmp_rval, rval = DDI_SUCCESS;
14602     char            *addr, *cp;
14603     size_t          s;
14604     mptsas_t        *mpt = DIP2MPT(pdip);
14605
14606     child = ddi_get_child(pdip);
14607     while (child) {
14608         addr = ddi_get_name_addr(child);
14609         prechild = child;
14610         child = ddi_get_next_sibling(child);
14611
14612         if (addr == NULL) {
14613             continue;
14614         }
14615         if ((cp = strchr(addr, ',')) == NULL) {
14616             continue;
14617         }
14618         s = (uintptr_t)cp - (uintptr_t)addr;
14619
14620         if (strncmp(addr, name, s) != 0) {
14621             continue;
14622         }
14623
14624         tmp_rval = mptsas_offline_lun(pdip, prechild, NULL,
14625                                       NDI_DEVI_REMOVE);
14626         if (tmp_rval != DDI_SUCCESS) {
14627             rval = DDI_FAILURE;
14628         }
14629
14630     }
14631
14632     if (tmp_rval != DDI_SUCCESS) {
14633         rval = DDI_FAILURE;
14634     }

```

```

14635         if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
14636             prechild, MPTSAS_DEV_GONE) !=  

14637             DDI_PROP_SUCCESS) {  

14638                 mptsas_log(mpt, CE_WARN, "mptsas driver "  

14639                     "unable to create property for "  

14640                     "SAS %s (MPTSAS_DEV_GONE)", addr);  

14641             }  

14642         }  

14643     }  

14645     pip = mdi_get_next_client_path(pdip, NULL);  

14646     while (pip) {  

14647         addr = MDI_PI(pip)->pi_addr;  

14648         savepip = pip;  

14649         pip = mdi_get_next_client_path(pdip, pip);  

14650         if (addr == NULL) {  

14651             continue;  

14652         }  

14654         if ((cp = strchr(addr, ',')) == NULL) {  

14655             continue;  

14656         }  

14658         s = (uintptr_t)cp - (uintptr_t)addr;  

14660         if (strncmp(addr, name, s) != 0) {  

14661             continue;  

14662         }  

14664         (void) mptsas_offline_lun(pdip, NULL, savepip,  

14665             NDI_DEVI_REMOVE);  

14666         /*  

14667             * driver will not invoke mdi_pi_free, so path will not  

14668             * be freed forever, return DDI_FAILURE.  

14669             */  

14670         rval = DDI_FAILURE;  

14671     }  

14672     return (rval);  

14673 }  

14675 static int  

14676 mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,  

14677     mdi_pathinfo_t *rpipe, uint_t flags)  

14678 {  

14679     int          rval = DDI_FAILURE;  

14680     char          *devname;  

14681     dev_info_t    *cdip, *parent;  

14683     if (rpipe != NULL) {  

14684         parent = scsi_vhci_dip;  

14685         cdip = mdi_pi_get_client(rpipe);  

14686     } else if (rdip != NULL) {  

14687         parent = pdip;  

14688         cdip = rdip;  

14689     } else {  

14690         return (DDI_FAILURE);  

14691     }  

14693     /*  

14694         * Make sure node is attached otherwise  

14695         * it won't have related cache nodes to  

14696         * clean up. i_ddi_dev_attached is  

14697         * similar to i_ddi_node_state(cdip) >=  

14698         * DS_ATTACHED.  

14699         */  

14700     if (i_ddi_dev_attached(cdip)) {  


```

```

14702         /* Get full devname */  

14703         devname = kmem_alloc(MAXNAMELEN + 1, KM_SLEEP);  

14704         (void) ddi_devname(cdip, devname);  

14705         /* Clean cache */  

14706         (void) devfs_clean(parent, devname + 1,  

14707             DV_CLEAN_FORCE);  

14708         kmem_free(devname, MAXNAMELEN + 1);  

14709     }  

14710     if (rpipe != NULL) {  

14711         if (MDI_PI_IS_OFFLINE(rpipe)) {  

14712             rval = DDI_SUCCESS;  

14713         } else {  

14714             rval = mdi_pi_offline(rpipe, 0);  

14715         }  

14716     } else {  

14717         rval = ndi_devi_offline(cdip, flags);  

14718     }  

14720     return (rval);  

14721 }  

14723 static dev_info_t *  

14724 mptsas_find_smp_child(dev_info_t *parent, char *str_wwn)  

14725 {  

14726     dev_info_t    *child = NULL;  

14727     char          *smp_wwn = NULL;  

14729     child = ddi_get_child(parent);  

14730     while (child) {  

14731         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, child,  

14732             DDI_PROP_DONTPASS, SMP_WWN, &smp_wwn)  

14733             != DDI_SUCCESS) {  

14734             child = ddi_get_next_sibling(child);  

14735             continue;  

14736         }  

14738         if (strcmp(smp_wwn, str_wwn) == 0) {  

14739             ddi_prop_free(smp_wwn);  

14740             break;  

14741         }  

14742         child = ddi_get_next_sibling(child);  

14743         ddi_prop_free(smp_wwn);  

14744     }  

14745     return (child);  

14746 }  

14748 static int  

14749 mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node, uint_t flags)  

14750 {  

14751     int          rval = DDI_FAILURE;  

14752     char          *devname;  

14753     char          wwn_str[MPTSAS_WWN_STRLEN];  

14754     dev_info_t    *cdip;  

14756     (void) sprintf(wwn_str, "%PRIx64", smp_node->m_addr.mta_wwn);  

14758     cdip = mptsas_find_smp_child(pdip, wwn_str);  

14760     if (cdip == NULL)  

14761         return (DDI_SUCCESS);  

14763     /*  

14764         * Make sure node is attached otherwise  

14765         * it won't have related cache nodes to  

14766         * clean up. i_ddi_dev_attached is

```

```

14767     * similiar to i_ddi_node_state(cdip) >=
14768     * DS_ATTACHED.
14769     */
14770 if (i_ddi_devi_attached(cdip)) {
14771
14772     /* Get full devname */
14773     devname = kmem_alloc(MAXNAMELEN + 1, KM_SLEEP);
14774     (void) ddi_devname(cdip, devname);
14775     /* Clean cache */
14776     (void) devfs_clean(pdip, devname + 1,
14777         DV_CLEAN_FORCE);
14778     kmem_free(devname, MAXNAMELEN + 1);
14779 }
14780
14781 rval = ndi_devi_offline(cdip, flags);
14782
14783 return (rval);
14784 }
14785
14786 static dev_info_t *
14787 mptsas_find_child(dev_info_t *pdip, char *name)
14788 {
14789     dev_info_t      *child = NULL;
14790     char            *rname = NULL;
14791     int              rval = DDI_FAILURE;
14792
14793     rname = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14794
14795     child = ddi_get_child(pdip);
14796     while (child) {
14797         rval = mptsas_name_child(child, rname, SCSI_MAXNAMELEN);
14798         if (rval != DDI_SUCCESS) {
14799             child = ddi_get_next_sibling(child);
14800             bzero(rname, SCSI_MAXNAMELEN);
14801             continue;
14802         }
14803
14804         if (strcmp(rname, name) == 0) {
14805             break;
14806         }
14807         child = ddi_get_next_sibling(child);
14808         bzero(rname, SCSI_MAXNAMELEN);
14809     }
14810
14811     kmem_free(rname, SCSI_MAXNAMELEN);
14812
14813     return (child);
14814 }
14815
14816 static dev_info_t *
14817 mptsas_find_child_addr(dev_info_t *pdip, uint64_t sasaddr, int lun)
14818 {
14819     dev_info_t      *child = NULL;
14820     char            *name = NULL;
14821     char            *addr = NULL;
14822
14823     name = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14824     addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14825     (void) sprintf(name, "%016"PRIx64, sasaddr);
14826     (void) sprintf(addr, "w%z.%x", name, lun);
14827     child = mptsas_find_child(pdip, addr);
14828     kmem_free(name, SCSI_MAXNAMELEN);
14829     kmem_free(addr, SCSI_MAXNAMELEN);
14830     return (child);
14831 }
14832 }
```

```

14834 static dev_info_t *
14835 mptsas_find_child_phys(dev_info_t *pdip, uint8_t phy)
14836 {
14837     dev_info_t      *child;
14838     char            *addr;
14839
14840     addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14841     (void) sprintf(addr, "p%z.0", phy);
14842     child = mptsas_find_child(pdip, addr);
14843     kmem_free(addr, SCSI_MAXNAMELEN);
14844     return (child);
14845 }
14846
14847 static mdi_pathinfo_t *
14848 mptsas_find_path_phys(dev_info_t *pdip, uint8_t phy)
14849 {
14850     mdi_pathinfo_t  *path;
14851     char            *addr = NULL;
14852
14853     addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14854     (void) sprintf(addr, "p%z.0", phy);
14855     path = mdi_pi_find(pdip, NULL, addr);
14856     kmem_free(addr, SCSI_MAXNAMELEN);
14857     return (path);
14858 }
14859
14860 static mdi_pathinfo_t *
14861 mptsas_find_path_addr(dev_info_t *parent, uint64_t sasaddr, int lun)
14862 {
14863     mdi_pathinfo_t  *path;
14864     char            *name = NULL;
14865     char            *addr = NULL;
14866
14867     name = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14868     addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
14869     (void) sprintf(name, "%016"PRIx64, sasaddr);
14870     (void) sprintf(addr, "w%z.%x", name, lun);
14871     path = mdi_pi_find(parent, NULL, addr);
14872     kmem_free(name, SCSI_MAXNAMELEN);
14873     kmem_free(addr, SCSI_MAXNAMELEN);
14874
14875     return (path);
14876 }
14877
14878 static int
14879 mptsas_create_lun(dev_info_t *pdip, struct scsi_inquiry *sd_inq,
14880                     dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14881 {
14882     int              i = 0;
14883     uchar_t          *inq83 = NULL;
14884     int              inq83_len1 = 0xFF;
14885     int              inq83_len = 0;
14886     int              rval = DDI_FAILURE;
14887     ddi_devid_t      devid;
14888     char            *guid = NULL;
14889     int              target = ptgt->m_devhdl;
14890     mdi_pathinfo_t   *pip = NULL;
14891     mptsas_t          *mpt = DIP2MPT(pdip);
14892
14893     /*
14894      * For DVD/CD ROM and tape devices and optical
14895      * devices, we won't try to enumerate them under
14896      * scsi_vhci, so no need to try page83
14897      */
14898     if (sd_inq && (sd_inq->inq_dtype == DTTYPE_RODIRECT ||
```

```

14899     sd_inq->inq_dtype == DTYPE_OPTICAL ||
14900     sd_inq->inq_dtype == DTYPE_ESI))
14901         goto create_lun;
14902
14903 /* The LCA returns good SCSI status, but corrupt page 83 data the first
14904 * time it is queried. The solution is to keep trying to request page83
14905 * and verify the GUID is not (DDI_NOT_WELL_FORMED) in
14906 * mptsas_inq83_retry_timeout seconds. If the timeout expires, driver
14907 * give up to get VPD page at this stage and fail the enumeration.
14908 */
14909
14911     inq83 = kmem_zalloc(inq83_len1, KM_SLEEP);
14912
14913     for (i = 0; i < mptsas_inq83_retry_timeout; i++) {
14914         rval = mptsas_inquiry(mpt, ptgt, lun, 0x83, inq83,
14915             inq83_len1, &inq83_len1, 1);
14916         if (rval != 0) {
14917             mptsas_log(mpt, CE_WARN, "!mptsas request inquiry page "
14918             "0x83 for target:%x, lun:%x failed!", target, lun);
14919             if (mptsas_physical_bind_failed_page_83 != B_FALSE)
14920                 goto create_lun;
14921             goto out;
14922         }
14923     }
14924     /*
14925      * create DEVID from inquiry data
14926     */
14927     if ((rval = ddi_devid_scsi_encode(
14928         DEVID_SCSI_ENCODE_VERSION_LATEST, NULL, (uchar_t *)sd_inq,
14929         sizeof (struct scsi_inquiry), NULL, 0, inq83,
14930         (size_t)inq83_len, &devid)) == DDI_SUCCESS) {
14931         /*
14932          * extract GUID from DEVID
14933         */
14934         guid = ddi_devid_to_guid(devid);
14935
14936         /*
14937          * Do not enable MPXIO if the strlen(guid) is greater
14938          * than MPTSAS_MAX_GUID_LEN, this constrain would be
14939          * handled by framework later.
14940         */
14941         if (guid && (strlen(guid) > MPTSAS_MAX_GUID_LEN)) {
14942             ddi_devid_free_guid(guid);
14943             guid = NULL;
14944             if (mpt->m_mpxio_enable == TRUE) {
14945                 mptsas_log(mpt, CE_NOTE, "!Target:%x, "
14946                 "lun:%x doesn't have a valid GUID, "
14947                 "multipathing for this drive is "
14948                 "not enabled", target, lun);
14949             }
14950
14951         /*
14952          * devid no longer needed
14953         */
14954         ddi_devid_free(devid);
14955         break;
14956     } else if (rval == DDI_NOT_WELL_FORMED) {
14957         /*
14958          * return value of ddi_devid_scsi_encode equal to
14959          * DDI_NOT_WELL_FORMED means DEVID_RETRY, it worth
14960          * to retry inquiry page 0x83 and get GUID.
14961         */
14962         NDBG20(("Not well formed devid, retry..."));
14963         delay(1 * drv_usectohz(1000000));
14964         continue;

```

```

14965     } else {
14966         mptsas_log(mpt, CE_WARN, "!Encode devid failed for "
14967             "path target:%x, lun:%x", target, lun);
14968         rval = DDI_FAILURE;
14969         goto create_lun;
14970     }
14971 }
14973     if (i == mptsas_inq83_retry_timeout) {
14974         mptsas_log(mpt, CE_WARN, "!Repeated page83 requests timeout "
14975             "for path target:%x, lun:%x", target, lun);
14976     }
14978     rval = DDI_FAILURE;
14980
14981     create_lun:
14982         if ((guid != NULL) && (mpt->m_mpxio_enable == TRUE)) {
14983             rval = mptsas_create_virt_lun(pdip, sd_inq, guid, lun_dip, &pip,
14984                 ptgt, lun);
14985         }
14986         if (rval != DDI_SUCCESS) {
14987             rval = mptsas_create_phys_lun(pdip, sd_inq, guid, lun_dip,
14988                 ptgt, lun);
14989         }
14990     }
14991     out:
14992         if (guid != NULL) {
14993             /*
14994              * guid no longer needed
14995             */
14996             ddi_devid_free_guid(guid);
14997         }
14998         if (inq83 != NULL)
14999             kmem_free(inq83, inq83_len1);
15000     }
15002 static int
15003 mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *inq, char *guid,
15004     dev_info_t **lun_dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt, int lun)
15005 {
15006     int target;
15007     char *nodename = NULL;
15008     char **compatible = NULL;
15009     int ncompatible = 0;
15010     int mdi_rtn = MDI_FAILURE;
15011     int old_guid = NULL;
15012     char *mpt = DIP2MPT(pdip);
15013     mptsas_t
15014     char *lun_addr = NULL;
15015     char *wwn_str = NULL;
15016     char *attached_wwn_str = NULL;
15017     char *component = NULL;
15018     uint8_t phy = 0xFF;
15019     uint64_t sas_wwn;
15020     int64_t lun64 = 0;
15021     uint32_t devinfo;
15022     uint16_t dev_hdl;
15023     pdev_hdl;
15024     dev_sas_wwn;
15025     pdev_sas_wwn;
15026     uint32_t pdev_info;
15027     physport;
15028     phy_id;
15029     page_address;
15030     uint16_t bay_num, enclosure, io_flags;
```

```

15031     char pdev_wwn_str[MPTSAS_WWN_STRLEN];
15032     uint32_t dev_info;

15034     mutex_enter(&mpt->m_mutex);
15035     target = ptgt->m_devhdl;
15036     sas_wwn = ptgt->m_addr.mta_wwn;
15037     devinfo = ptgt->m_deviceinfo;
15038     phy = ptgt->m_phynum;
15039     mutex_exit(&mpt->m_mutex);

15041     if (sas_wwn) {
15042         *pip = mptsas_find_path_addr(pdip, sas_wwn, lun);
15043     } else {
15044         *pip = mptsas_find_path_phy(pdip, phy);
15045     }

15047     if (*pip != NULL) {
15048         *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
15049         ASSERT(*lun_dip != NULL);
15050         if (ddi_prop_lookup_string(DDI_DEV_T_ANY, *lun_dip,
15051             (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM),
15052             MDI_CLIENT_GUID_PROP, &old_guid) == DDI_SUCCESS) {
15053             if (strncmp(guid, old_guid, strlen(guid)) == 0) {
15054                 /*
15055                  * Same path back online again.
15056                  */
15057                 (void) ddi_prop_free(old_guid);
15058                 if ((!MDI_PI_IS_ONLINE(*pip)) &&
15059                     (!MDI_PI_IS_STANDBY(*pip)) &&
15060                     (ptgt->m_tgt_unconfigured == 0)) {
15061                     rval = mdi_pi_online(*pip, 0);
15062                     mutex_enter(&mpt->m_mutex);
15063                     ptgt->m_led_status = 0;
15064                     (void) mptsas_flush_led_status(mpt,
15065                         ptgt);
15066                     mutex_exit(&mpt->m_mutex);
15067                 } else {
15068                     rval = DDI_SUCCESS;
15069                 }
15070                 if (rval != DDI_SUCCESS) {
15071                     mptsas_log(mpt, CE_WARN, "path:target: %x, lun:%x online failed!", target,
15072                                         lun);
15073                     *pip = NULL;
15074                     *lun_dip = NULL;
15075                 }
15076             }
15077             return (rval);
15078         } else {
15079             /*
15080              * The GUID of the LUN has changed which maybe
15081              * because customer mapped another volume to the
15082              * same LUN.
15083              */
15084             mptsas_log(mpt, CE_WARN, "The GUID of the "
15085                 "target:%x, lun:%x was changed, maybe "
15086                 "because someone mapped another volume "
15087                 "to the same LUN", target, lun);
15088             (void) ddi_prop_free(old_guid);
15089             if (!MDI_PI_IS_OFFLINE(*pip)) {
15090                 rval = mdi_pi_offline(*pip, 0);
15091                 if (rval != MDI_SUCCESS) {
15092                     mptsas_log(mpt, CE_WARN, "path: %x, lun:%x offline "
15093                                         "failed!", target, lun);
15094                     *pip = NULL;
15095                     *lun_dip = NULL;
15096                 }
15097             }
15098         }
15099     }
15100 }
```

```

15097 } return (DDI_FAILURE);
15098
15099 }
15100 if (mdi_pi_free(*pip, 0) != MDI_SUCCESS) {
15101     mptsas_log(mpt, CE_WARN, "path:target:%x",
15102                 "lun:%x free failed!", target,
15103                 lun);
15104     *pip = NULL;
15105     *lun_dip = NULL;
15106     return (DDI_FAILURE);
15107 }
15108 }
15109 } else {
15110     mptsas_log(mpt, CE_WARN, "Can't get client-guid "
15111                 "property for path:target:%x, lun:%x", target, lun);
15112     *pip = NULL;
15113     *lun_dip = NULL;
15114     return (DDI_FAILURE);
15115 }
15116 }
15117 scsi_hba_nodename_compatible_get(inq, NULL,
15118         inq->inq_dtype, NULL, &nodename, &compatible, &ncompatible);

15120 /*
15121 * if nodename can't be determined then print a message and skip it
15122 */
15123 if (nodename == NULL) {
15124     mptsas_log(mpt, CE_WARN, "mptsas driver found no compatible "
15125                 "driver for target%d lun %d dtype:0x%02x", target, lun,
15126                 inq->inq_dtype);
15127     return (DDI_FAILURE);
15128 }

15130 wwn_str = kmem_zalloc(MPTSAWS_WWN_STRLEN, KM_SLEEP);
15131 /* The property is needed by MPAPI */
15132 (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);

15134 lun_addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
15135 if (guid) {
15136     (void) sprintf(lun_addr, "w%8x", wwn_str, lun);
15137     (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15138 } else {
15139     (void) sprintf(lun_addr, "p%8x,%x", phy, lun);
15140     (void) sprintf(wwn_str, "p%8x", phy);
15141 }

15143 mdi_rtn = mdi_pi_alloc_compatible(pdip, nodename,
15144         guid, lun_addr, compatible, ncompatible,
15145         0, pip);
15146 if (mdi_rtn == MDI_SUCCESS) {

15148     if (mdi_prop_update_string(*pip, MDI_GUID,
15149                     guid) != DDI_SUCCESS) {
15150         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15151                     "create prop for target %d lun %d (MDI_GUID)",
15152                     target, lun);
15153         mdi_rtn = MDI_FAILURE;
15154         goto virt_create_done;
15155     }

15157     if (mdi_prop_update_int(*pip, LUN_PROP,
15158                     lun) != DDI_SUCCESS) {
15159         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15160                     "create prop for target %d lun %d (LUN_PROP)",
15161                     target, lun);
15162         mdi_rtn = MDI_FAILURE;

```

```

15163         goto virt_create_done;
15164     }
15165     lun64 = (int64_t)lun;
15166     if (mdi_prop_update_int64(*pip, LUN64_PROP,
15167         lun64) != DDI_SUCCESS) {
15168         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15169             "create prop for target %d (LUN64_PROP)",
15170             target);
15171         mdi_rtn = MDI_FAILURE;
15172         goto virt_create_done;
15173     }
15174     if (mdi_prop_update_string_array(*pip, "compatible",
15175         compatible, ncompatible) !=
15176         DDI_PROP_SUCCESS) {
15177         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15178             "create prop for target %d lun %d (COMPATIBLE)",
15179             target, lun);
15180         mdi_rtn = MDI_FAILURE;
15181         goto virt_create_done;
15182     }
15183     if (sas_wwn && (mdi_prop_update_string(*pip,
15184         SCSI_ADDR_PROP_TARGET_PORT, wnn_str) != DDI_PROP_SUCCESS)) {
15185         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15186             "create prop for target %d lun %d "
15187             "(target-port)", target, lun);
15188         mdi_rtn = MDI_FAILURE;
15189         goto virt_create_done;
15190     } else if ((sas_wwn == 0) && (mdi_prop_update_int(*pip,
15191         "sata-ph", phy) != DDI_PROP_SUCCESS)) {
15192         /*
15193             * Direct attached SATA device without DeviceName
15194         */
15195         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15196             "create prop for SAS target %d lun %d "
15197             "(sata-ph)", target, lun);
15198         mdi_rtn = MDI_FAILURE;
15199         goto virt_create_done;
15200     }
15201     mutex_enter(&mpt->m_mutex);

page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15202     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | 
15203     (uint32_t)ptgt->m_devhdl;
15204     rval = mptsas_get_sas_device_page0(mpt, page_address,
15205         &dev_hdl, &dev_sas_wnn, &dev_info, &physport,
15206         &phy_id, &pdev_hdl, &bay_num, &enclosure, &ioc_flags);
15207     if (rval != DDI_SUCCESS) {
15208         mutex_exit(&mpt->m_mutex);
15209         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15210             "parent device for handle %d", page_address);
15211         mdi_rtn = MDI_FAILURE;
15212         goto virt_create_done;
15213     }

page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15214     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
15215     rval = mptsas_get_sas_device_page0(mpt, page_address,
15216         &dev_hdl, &pdev_sas_wnn, &pdev_info, &physport,
15217         &phy_id, &pdev_hdl, &bay_num, &enclosure, &ioc_flags);
15218     if (rval != DDI_SUCCESS) {
15219         mutex_exit(&mpt->m_mutex);
15220         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
15221             "device info for handle %d", page_address);
15222         mdi_rtn = MDI_FAILURE;
15223         goto virt_create_done;
15224     }

```

```

15230         mutex_exit(&mpt->m_mutex);

15231         /*
15232             * If this device direct attached to the controller
15233             * set the attached-port to the base wwid
15234         */
15235         if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15236             != DEVINFO_DIRECT_ATTACHED) {
15237             (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15238                           pdev_sas_wwn);
15239         } else {
15240             /*
15241                 * Update the iport's attached-port to guid
15242             */
15243             if (sas_wwn == 0) {
15244                 (void) sprintf(wwn_str, "p%x", phy);
15245             } else {
15246                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15247             }
15248         }
15249         if (ddi_prop_update_string(DDI_DEV_T_NONE,
15250             pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wnn_str) !=
15251             DDI_PROP_SUCCESS) {
15252             mptsas_log(mpt, CE_WARN,
15253                 "mptsas unable to create "
15254                 "property for iport target-port"
15255                 " %s (sas_wwn)",
15256                 wnn_str);
15257             mdi_rtn = MDI_FAILURE;
15258             goto virt_create_done;
15259         }
15260         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15261                         mpt->un.m_base_wwid);
15262     }

15263     if (mdi_prop_update_string(*pip,
15264         SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
15265             DDI_PROP_SUCCESS) {
15266         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15267             "property for iport attached-port %s (sas_wwn)",
15268             attached_wwn_str);
15269         mdi_rtn = MDI_FAILURE;
15270         goto virt_create_done;
15271     }

15272     if (inq->inq_dtype == 0) {
15273         component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
15274         /*
15275             * set obp path for pathinfo
15276         */
15277         (void) snprintf(component, MAXPATHLEN,
15278                         "disk@%s", lun_addr);
15279         if (mdi_pi_pathname_obp_set(*pip, component) !=
15280             DDI_SUCCESS) {
15281             mptsas_log(mpt, CE_WARN, "mpt_sas driver "
15282                 "unable to set obp-path for object %s",
15283                 component);
15284             mdi_rtn = MDI_FAILURE;
15285             goto virt_create_done;
15286         }
15287     }

15288     *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;

```

```

15295     if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
15296         MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
15297         if ((ndi_prop_update_int(DDI_DEV_T_NONE, *lun_dip,
15298             "pm-capable", 1)) != DDI_PROP_SUCCESS) {
15299             mptsas_log(mpt, CE_WARN, "mptsas driver"
15300                 "failed to create pm-capable "
15301                     "property, target %d", target);
15302             mdi_rtn = MDI_FAILURE;
15303             goto virt_create_done;
15304         }
15305     }
15306     /*
15307      * Create the phy-num property
15308      */
15309     if (mdi_prop_update_int(*pip, "phy-num",
15310         ptgt->m_phynum) != DDI_SUCCESS) {
15311         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
15312             "create phy-num property for target %d lun %d",
15313                 target, lun);
15314         mdi_rtn = MDI_FAILURE;
15315         goto virt_create_done;
15316     }
15317 NDBG20(("new path:%s onlining,", MDI_PI(*pip)->pi_addr));
15318     mdi_rtn = mdi_pi_online(*pip, 0);
15319     if (mdi_rtn == MDI_SUCCESS) {
15320         mutex_enter(&mpt->m_mutex);
15321         ptgt->m_led_status = 0;
15322         (void) mptsas_flush_led_status(mpt, ptgt);
15323         mutex_exit(&mpt->m_mutex);
15324     }
15325     if (mdi_rtn == MDI_NOT_SUPPORTED) {
15326         mdi_rtn = MDI_FAILURE;
15327     }
15328 virt_create_done:
15329     if (*pip && mdi_rtn != MDI_SUCCESS) {
15330         (void) mdi_pi_free(*pip, 0);
15331         *pip = NULL;
15332         *lun_dip = NULL;
15333     }
15334 }
15335
15336 sscsi_hba_nodename_compatible_free(nodename, compatible);
15337 if (lun_addr != NULL) {
15338     kmem_free(lun_addr, SCSI_MAXNAMELEN);
15339 }
15340 if (wwn_str != NULL) {
15341     kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
15342 }
15343 if (component != NULL) {
15344     kmem_free(component, MAXPATHLEN);
15345 }
15346
15347 return ((mdi_rtn == MDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
15348
15349 }

15350 static int
15351 mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *inq,
15352     char *guid, dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
15353 {
15354     int target;
15355     int rval;
15356     int ndi_rtn = NDI_FAILURE;
15357     uint64_t be_sas_wwn;
15358     char *nodename = NULL;
15359     **compatible = NULL;
15360

```

```

15361     int ncompatible = 0;
15362     int instance = 0;
15363     mptsas_t *mpt = DIP2MPT(pdip);
15364     char *wwn_str = NULL;
15365     *component = NULL;
15366     *attached_wwn_str = NULL;
15367     phy = 0xFF;
15368     sas_wwn;
15369     devinfo;
15370     dev_hdl;
15371     pdev_hdl;
15372     pdev_sas_wwn;
15373     dev_sas_wwn;
15374     pdev_info;
15375     physport;
15376     phy_id;
15377     page_address;
15378     bay_num, enclosure, io_flags;
15379     char pdev_wwn_str[MPTSAS_WWN_STRLEN];
15380     dev_info;
15381     lun64 = 0;
15382
15383 mutex_enter(&mpt->m_mutex);
15384 target = ptgt->m_devhdl;
15385 sas_wwn = ptgt->m_addr.mta_wwn;
15386 devinfo = ptgt->m_deviceinfo;
15387 phy = ptgt->m_phynum;
15388 mutex_exit(&mpt->m_mutex);
15389
15390 /*
15391  * generate compatible property with binding-set "mpt"
15392  */
15393 sscsi_hba_nodename_compatible_get(inq, NULL, inq->inq_dtype, NULL,
15394     &nodename, &compatible, &ncompatible);
15395
15396 /*
15397  * if nodename can't be determined then print a message and skip it
15398  */
15399 if (nodename == NULL) {
15400     mptsas_log(mpt, CE_WARN, "mptsas found no compatible driver "
15401         "for target %d lun %d", target, lun);
15402     return (DDI_FAILURE);
15403 }
15404
15405 ndi_rtn = ndi_devi_alloc(pdip, nodename,
15406     DEVI_SID_NODEID, lun_dip);
15407
15408 /*
15409  * if lun alloc success, set props
15410  */
15411 if (ndi_rtn == NDI_SUCCESS) {
15412     if (ndi_prop_update_int(DDI_DEV_T_NONE,
15413         *lun_dip, LUN_PROP, lun) != DDI_PROP_SUCCESS) {
15414         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15415             "property for target %d lun %d (LUN_PROP)",
15416                 target, lun);
15417         ndi_rtn = NDI_FAILURE;
15418         goto phys_create_done;
15419     }
15420
15421     lun64 = (int64_t)lun;
15422     if (ndi_prop_update_int64(DDI_DEV_T_NONE,
15423         *lun_dip, LUN64_PROP, lun64) != DDI_PROP_SUCCESS) {
15424

```

```

15427             mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15428                     "property for target %d lun64 %d (LUN64_PROP)",
15429                     target, lun);
15430             ndi_rtn = NDI_FAILURE;
15431             goto phys_create_done;
15432     }
15433     if (ndi_prop_update_string_array(DDI_DEV_T_NONE,
15434             *lun_dip, "compatible", compatible, ncompatible)
15435             != DDI_PROP_SUCCESS) {
15436         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15437                     "property for target %d lun %d (COMPATIBLE)",
15438                     target, lun);
15439         ndi_rtn = NDI_FAILURE;
15440         goto phys_create_done;
15441     }
15442
15443     /*
15444      * We need the SAS WWN for non-multipath devices, so
15445      * we'll use the same property as that multipathing
15446      * devices need to present for MPAPI. If we don't have
15447      * a WWN (e.g. parallel SCSI), don't create the prop.
15448      */
15449     wwn_str = kmalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
15450     (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15451     if (sas_wwn && ndi_prop_update_string(DDI_DEV_T_NONE,
15452             *lun_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str)
15453             != DDI_PROP_SUCCESS) {
15454         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15455                     "create property for SAS target %d lun %d "
15456                     "(target-port)", target, lun);
15457         ndi_rtn = NDI_FAILURE;
15458         goto phys_create_done;
15459     }
15460
15461     be_sas_wwn = BE_64(sas_wwn);
15462     if (sas_wwn && ndi_prop_update_byte_array(
15463             DDI_DEV_T_NONE, *lun_dip, "port-wwn",
15464             (uchar_t *)&be_sas_wwn, 8) != DDI_PROP_SUCCESS) {
15465         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15466                     "create property for SAS target %d lun %d "
15467                     "(port-wwn)", target, lun);
15468         ndi_rtn = NDI_FAILURE;
15469         goto phys_create_done;
15470     } else if ((sas_wwn == 0) && (ndi_prop_update_int(
15471             DDI_DEV_T_NONE, *lun_dip, "sata-phy", phy) !=
15472             DDI_PROP_SUCCESS)) {
15473         /*
15474          * Direct attached SATA device without DeviceName
15475          */
15476         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15477                     "create property for SAS target %d lun %d "
15478                     "(sata-phy)", target, lun);
15479         ndi_rtn = NDI_FAILURE;
15480         goto phys_create_done;
15481     }
15482
15483     if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
15484             *lun_dip, SAS_PROP) != DDI_PROP_SUCCESS) {
15485         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15486                     "create property for SAS target %d lun %d"
15487                     " (%SAS_PROP)", target, lun);
15488         ndi_rtn = NDI_FAILURE;
15489         goto phys_create_done;
15490     }
15491     if (guid && (ndi_prop_update_string(DDI_DEV_T_NONE,
15492             *lun_dip, NDI_GUID, guid) != DDI_SUCCESS)) {

```

```

15493             mptsas_log(mpt, CE_WARN, "mptsas unable to "
15494                     "create guid property for target %d "
15495                     "lun %d", target, lun);
15496             ndi_rtn = NDI_FAILURE;
15497             goto phys_create_done;
15498     }
15499
15500     /*
15501      * The following code is to set properties for SM-HBA support,
15502      * it doesn't apply to RAID volumes
15503      */
15504     if (ptgt->m_addr.mta.phymask == 0)
15505         goto phys_raid_lun;
15506
15507     mutex_enter(&mpt->m_mutex);
15508
15509     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15510                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)ptgt->m_devhdl;
15511     rval = mptsas_get_sas_device_page0(mpt, page_address,
15512             &dev_hdl, &dev_sas_wwn, &dev_info,
15513             &physport, &phy_id, &pdev_hdl,
15514             &bay_num, &enclosure, &io_flags);
15515     if (rval != DDI_SUCCESS) {
15516         mutex_exit(&mpt->m_mutex);
15517         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
15518                     "parent device for handle %d.", page_address);
15519         ndi_rtn = NDI_FAILURE;
15520         goto phys_create_done;
15521     }
15522
15523     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15524                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
15525     rval = mptsas_get_sas_device_page0(mpt, page_address,
15526             &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
15527             &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15528     if (rval != DDI_SUCCESS) {
15529         mutex_exit(&mpt->m_mutex);
15530         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15531                     "device for handle %d.", page_address);
15532         ndi_rtn = NDI_FAILURE;
15533         goto phys_create_done;
15534     }
15535
15536     mutex_exit(&mpt->m_mutex);
15537
15538     /*
15539      * If this device direct attached to the controller
15540      * set the attached-port to the base wwid
15541      */
15542     if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED) ==
15543             !DEVINFO_DIRECT_ATTACHED) {
15544         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15545                     pdev_sas_wwn);
15546     } else {
15547         /*
15548          * Update the iport's attached-port to guid
15549          */
15550         if (sas_wwn == 0) {
15551             (void) sprintf(wwn_str, "P%"X, phy);
15552         } else {
15553             (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
15554         }
15555         if (ddi_prop_update_string(DDI_DEV_T_NONE,
15556             pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
15557             DDI_PROP_SUCCESS) {
15558

```

```

15559             mptsas_log(mpt, CE_WARN,
15560                     "mptsas unable to create "
15561                     "property for iport target-port"
15562                     "%s (sas_wwn)",
15563                     wwn_str);
15564             ndi_rtn = NDI_FAILURE;
15565             goto phys_create_done;
15566         }
15567
15568         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
15569                         mpt->un.m_base_wwid);
15570     }
15571
15572     if (ndi_prop_update_string(DDI_DEV_T_NONE,
15573                               *lun_dip, SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
15574                               DDI_PROP_SUCCESS) {
15575         mptsas_log(mpt, CE_WARN,
15576                     "mptsas unable to create "
15577                     "property for iport attached-port %s (sas_wwn)",
15578                     attached_wwn_str);
15579             ndi_rtn = NDI_FAILURE;
15580             goto phys_create_done;
15581     }
15582
15583     if (IS_SATA_DEVICE(dev_info)) {
15584         if (ndi_prop_update_string(DDI_DEV_T_NONE,
15585                               *lun_dip, MPTSAS_VARIANT, "sata") !=
15586                               DDI_PROP_SUCCESS) {
15587             mptsas_log(mpt, CE_WARN,
15588                         "mptsas unable to create "
15589                         "property for device variant ");
15590             ndi_rtn = NDI_FAILURE;
15591             goto phys_create_done;
15592     }
15593
15594     if (IS_ATAPI_DEVICE(dev_info)) {
15595         if (ndi_prop_update_string(DDI_DEV_T_NONE,
15596                               *lun_dip, MPTSAS_VARIANT, "atapi") !=
15597                               DDI_PROP_SUCCESS) {
15598             mptsas_log(mpt, CE_WARN,
15599                         "mptsas unable to create "
15600                         "property for device variant ");
15601             ndi_rtn = NDI_FAILURE;
15602             goto phys_create_done;
15603     }
15604
15605 }
15606
15607 phys_raid_lun:
15608 /*
15609 * if this is a SAS controller, and the target is a SATA
15610 * drive, set the 'pm-capable' property for sd and if on
15611 * an OPL platform, also check if this is an ATAPI
15612 * device.
15613 */
15614 instance = ddi_get_instance(mpt->m_dip);
15615 if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
15616 MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
15617     NDBG2(("mptsas%d: creating pm-capable property, "
15618           "target %d", instance, target));
15619
15620     if ((ndi_prop_update_int(DDI_DEV_T_NONE,
15621                               *lun_dip, "pm-capable", 1)) !=
15622         DDI_PROP_SUCCESS) {
15623         mptsas_log(mpt, CE_WARN, "mptsas "
15624             "failed to create pm-capable "

```

```

15625                     "property, target %d", target);
15626             ndi_rtn = NDI_FAILURE;
15627             goto phys_create_done;
15628         }
15629     }
15630
15631     if ((inq->inq_dtype == 0) || (inq->inq_dtype == 5)) {
15632         /*
15633          * add 'obp-path' properties for devinfo
15634          */
15635         bzero(wwn_str, sizeof(wwn_str));
15636         (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
15637         component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
15638         if (guid) {
15639             (void) snprintf(component, MAXPATHLEN,
15640                           "disk@w%*s,%x", wwn_str, lun);
15641         } else {
15642             (void) snprintf(component, MAXPATHLEN,
15643                           "disk@p%*x,%x", phy, lun);
15644         }
15645         if (ddi_pathname_obp_set(*lun_dip, component)
15646             != DDI_SUCCESS) {
15647             mptsas_log(mpt, CE_WARN, "mpt_sas driver "
15648                         "unable to set obp-path for SAS "
15649                         "object %s", component);
15650             ndi_rtn = NDI_FAILURE;
15651             goto phys_create_done;
15652         }
15653     }
15654     /*
15655      * Create the phy-num property for non-raid disk
15656      */
15657     if (ptgt->m_addr.mta_phymask != 0) {
15658         if (ndi_prop_update_int(DDI_DEV_T_NONE,
15659                               *lun_dip, "phy-num", ptgt->m_phynum) !=
15660                               DDI_PROP_SUCCESS) {
15661             mptsas_log(mpt, CE_WARN, "mptsas driver "
15662                         "failed to create phy-num property for "
15663                         "target %d", target);
15664             ndi_rtn = NDI_FAILURE;
15665             goto phys_create_done;
15666         }
15667     }
15668 phys_create_done:
15669 /*
15670  * If props were setup ok, online the lun
15671 */
15672 if (ndi_rtn == NDI_SUCCESS) {
15673     /*
15674      * Try to online the new node
15675      */
15676     ndi_rtn = ndi_devi_online(*lun_dip, NDI_ONLINE_ATTACH);
15677 }
15678 if (ndi_rtn == NDI_SUCCESS) {
15679     mutex_enter(&mpt->m_mutex);
15680     ptgt->m_led_status = 0;
15681     (void) mptsas_flush_led_status(mpt, ptgt);
15682     mutex_exit(&mpt->m_mutex);
15683 }
15684
15685 /*
15686  * If success set rtn flag, else unwire alloc'd lun
15687 */
15688 if (ndi_rtn != NDI_SUCCESS) {
15689     NDBG12(("mptsas driver unable to online "
15690             "
```

```

15691             "target %d lun %d", target, lun));
15692         ndi_prop_remove_all(*lun_dip);
15693         (void) ndi_devi_free(*lun_dip);
15694         *lun_dip = NULL;
15695     }
15696 }

15698     scsi_hba_nodename_compatible_free(nodename, compatible);

15700     if (wwn_str != NULL) {
15701         kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
15702     }
15703     if (component != NULL) {
15704         kmem_free(component, MAXPATHLEN);
15705     }

15708     return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
15709 }

15711 static int
15712 mptsas_probe_smp(dev_info_t *pdip, uint64_t wwn)
15713 {
15714     mptsas_t          *mpt = DIP2MPT(pdip);
15715     struct smp_device smp_sd;

15717     /* XXX An HBA driver should not be allocating an smp_device. */
15718     bzero(&smp_sd, sizeof (struct smp_device));
15719     smp_sd.smp_sd_address.smp_a_hba_tran = mpt->m_smptran;
15720     bcopy(&wwn, smp_sd.smp_sd_address.smp_a_wwn, SAS_WWN_BYTE_SIZE);

15722     if (smp_probe(&smp_sd) != DDI_PROBE_SUCCESS)
15723         return (NDI_FAILURE);
15724     return (NDI_SUCCESS);
15725 }

15727 static int
15728 mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn, dev_info_t **smp_dip)
15729 {
15730     mptsas_t          *mpt = DIP2MPT(pdip);
15731     mptsas_smp_t      *psmp = NULL;
15732     int                rval;
15733     int                phymask;

15735     /*
15736     * Get the physical port associated to the iport
15737     * PHYMASK TODO
15738     */
15739     phymask = ddi_prop_get_int(DDI_DEV_T_ANY, pdip, 0,
15740                               "phymask", 0);
15741     /*
15742     * Find the smp node in hash table with specified sas address and
15743     * physical port
15744     */
15745     psmp = mptsas_wwid_to_psmp(mpt, phymask, sas_wwn);
15746     if (psmp == NULL) {
15747         return (DDI_FAILURE);
15748     }

15750     rval = mptsas_online_smp(pdip, psmp, smp_dip);

15752     return (rval);
15753 }

15755 static int
15756 mptsas_online_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,

```

```

15757     dev_info_t **smp_dip)
15758 {
15759     char          wwn_str[MPTSAS_WWN_STRLEN];
15760     char          attached_wwn_str[MPTSAS_WWN_STRLEN];
15761     int           ndi_rtn = NDI_FAILURE;
15762     int           rval = 0;
15763     mptsas_smp_t dev_info;
15764     uint32_t      page_address;
15765     mptsas_t      *mpt = DIP2MPT(pdip);
15766     dev_hdl_t    dev_hdl;
15767     uint64_t      sas_wwn;
15768     uint64_t      smp_sas_wwn;
15769     uint8_t       physport;
15770     uint8_t       phy_id;
15771     uint16_t      pdev_hdl;
15772     uint8_t       numphys = 0;
15773     uint16_t      i = 0;
15774     char          phymask[MPTSAS_MAX_PHYS];
15775     char          *iport = NULL;
15776     mptsas_phymask_t phy_mask = 0;
15777     uint16_t      attached_devhdl;
15778     uint16_t      bay_num, enclosure, io_flags;

15780     (void) sprintf(wwn_str, "%PRIx64, smp_node->m_addr.mta_wwn");

15782     /*
15783     * Probe smp device, prevent the node of removed device from being
15784     * configured successfully
15785     */
15786     if (mptsas_probe_smp(pdip, smp_node->m_addr.mta_wwn) != NDI_SUCCESS) {
15787         return (DDI_FAILURE);
15788     }

15790     if ((*smp_dip = mptsas_find_smp_child(pdip, wwn_str)) != NULL) {
15791         return (DDI_SUCCESS);
15792     }

15794     ndi_rtn = ndi_devi_alloc(pdip, "smp", DEVI_SID_NODEID, smp_dip);

15796     /*
15797     * if lun alloc success, set props
15798     */
15799     if (ndi_rtn == NDI_SUCCESS) {
15800         /*
15801         * Set the flavor of the child to be SMP flavored
15802         */
15803     ndi_flavor_set(*smp_dip, SCSA_FLAVOR_SMP);

15805     if (ndi_prop_update_string(DDI_DEV_T_NONE,
15806                               *smp_dip, SMP_WWN, wwn_str) != DDI_PROP_SUCCESS) {
15807         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15808                     "property for smp device %s (sas_wwn)",
15809                     wwn_str);
15810         ndi_rtn = NDI_FAILURE;
15811         goto smp_create_done;
15812     }
15813     (void) sprintf(wwn_str, "w%PRIx64, smp_node->m_addr.mta_wwn");
15814     if (ndi_prop_update_string(DDI_DEV_T_NONE,
15815                               *smp_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str) != DDI_PROP_SUCCESS) {
15816         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15817                     "property for iport target-port %s (sas_wwn)",
15818                     wwn_str);
15819         ndi_rtn = NDI_FAILURE;
15820         goto smp_create_done;
15821     }
15822

```

```

15823     }
15825     mutex_enter(&mpt->m_mutex);
15826
15827     page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HANDLE &
15828         MPI2_SAS_EXPAND_PGAD_FORM_MASK) | smp_node->m_devhdl;
15829     rval = mptsas_get_sas_expander_page0(mpt, page_address,
15830         &dev_info);
15831     if (rval != DDI_SUCCESS) {
15832         mutex_exit(&mpt->m_mutex);
15833         mptsas_log(mpt, CE_WARN,
15834             "mptsas unable to get expander "
15835             "parent device info for %x", page_address);
15836         ndi_rtn = NDI_FAILURE;
15837         goto smp_create_done;
15838     }
15839
15840     smp_node->m_pdevhdl = dev_info.m_pdevhdl;
15841     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15842         MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15843         (uint32_t)dev_info.m_pdevhdl;
15844     rval = mptsas_get_sas_device_page0(mpt, page_address,
15845         &dev_hdl, &smp_sas_wnn, &smp_node->m_pdevinfo, &physport,
15846         &phy_id, &pdev_hdl, &bay_num, &enclosure, &io_flags);
15847     if (rval != DDI_SUCCESS) {
15848         mutex_exit(&mpt->m_mutex);
15849         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15850             "device info for %x", page_address);
15851         ndi_rtn = NDI_FAILURE;
15852         goto smp_create_done;
15853     }
15854
15855     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
15856         MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
15857         (uint32_t)dev_info.m_devhdl;
15858     rval = mptsas_get_sas_device_page0(mpt, page_address,
15859         &dev_hdl, &smp_sas_wnn, &smp_node->m_deviceinfo,
15860         &physport, &phy_id, &pdev_hdl, &bay_num, &enclosure,
15861         &io_flags);
15862     if (rval != DDI_SUCCESS) {
15863         mutex_exit(&mpt->m_mutex);
15864         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
15865             "device info for %x", page_address);
15866         ndi_rtn = NDI_FAILURE;
15867         goto smp_create_done;
15868     }
15869     mutex_exit(&mpt->m_mutex);
15870
15871     /*
15872      * If this smp direct attached to the controller
15873      * set the attached-port to the base wwid
15874      */
15875     if ((smp_node->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15876         != DEVINFO_DIRECT_ATTACHED) {
15877         (void) sprintf(attached_wwn_str, "w%016"PRIx64,
15878                         sas_wnn);
15879     } else {
15880         (void) sprintf(attached_wwn_str, "w%016"PRIx64,
15881                         mpt->un.m_base_wwid);
15882     }
15883
15884     if (ndi_prop_update_string(DDI_DEV_T_NONE,
15885         *smp_dip, SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwn_str) !=
15886         DDI_PROP_SUCCESS) {
15887         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
15888             "property for smp attached-port %s (sas_wnn)",
```

```

15889         attached_wwn_str);
15890         ndi_rtn = NDI_FAILURE;
15891         goto smp_create_done;
15892     }
15893
15894     if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
15895         *smp_dip, SMP_PROP) != DDI_PROP_SUCCESS) {
15896         mptsas_log(mpt, CE_WARN, "mptsas unable to "
15897             "create property for SMP %s (SMP_PROP) ",
15898             wnn_str);
15899         ndi_rtn = NDI_FAILURE;
15900         goto smp_create_done;
15901     }
15902
15903     /*
15904      * check the smp to see whether it direct
15905      * attached to the controller
15906      */
15907     if ((smp_node->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
15908         != DEVINFO_DIRECT_ATTACHED) {
15909         goto smp_create_done;
15910     }
15911     numphys = ddi_prop_get_int(DDI_DEV_T_ANY, pdip,
15912         DDI_PROP_DONTPASS, MPTSAS_NUM_PHYS, -1);
15913     if (numphys > 0) {
15914         goto smp_create_done;
15915     }
15916
15917     /*
15918      * this iport is an old iport, we need to
15919      * reconfig the props for it.
15920      */
15921     if (ddi_prop_update_int(DDI_DEV_T_NONE, pdip,
15922         MPTSAS_VIRTUAL_PORT, 0) != DDI_PROP_SUCCESS) {
15923         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15924             MPTSAS_VIRTUAL_PORT);
15925         mptsas_log(mpt, CE_WARN, "mptsas virtual port "
15926             "prop update failed");
15927         goto smp_create_done;
15928     }
15929
15930     mutex_enter(&mpt->m_mutex);
15931     numphys = 0;
15932     iport = ddi_get_name_addr(pdip);
15933     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
15934         bzero(phymask, sizeof(phymask));
15935         (void) sprintf(phymask,
15936             "%x", mpt->m_phy_info[i].phy_mask);
15937         if (strcmp(phymask, iport) == 0) {
15938             phy_mask = mpt->m_phy_info[i].phy_mask;
15939             break;
15940         }
15941     }
15942
15943     for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
15944         if ((phy_mask >> i) & 0x01) {
15945             numphys++;
15946         }
15947     }
15948
15949     /*
15950      * Update PHY info for smhba
15951      */
15952     if (mptsas_smhba_phy_init(mpt)) {
15953         mutex_exit(&mpt->m_mutex);
15954         mptsas_log(mpt, CE_WARN, "mptsas phy update "
15955             "failed");
```

```

15955         goto smp_create_done;
15956     }
15957     mutex_exit(&mpt->m_mutex);
15958
15959     mptsas_smhba_set_all_phys_props(mpt, pdip, numphys, phy_mask,
15960                                     &attached_devhdl);
15961
15962     if (ddi_prop_update_int(DDI_DEV_T_NONE, pdip,
15963                             MPTAS_NUM_PHYS, numphys) != DDI_PROP_SUCCESS) {
15964         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15965                               MPTAS_NUM_PHYS);
15966         mptsas_log(mpt, CE_WARN, "mptsas update "
15967                    "num phys props failed");
15968         goto smp_create_done;
15969     }
15970     /*
15971      * Add parent's props for SMHBA support
15972     */
15973     if (ddi_prop_update_string(DDI_DEV_T_NONE, pdip,
15974                               SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) != DDI_PROP_SUCCESS) {
15975         (void) ddi_prop_remove(DDI_DEV_T_NONE, pdip,
15976                               SCSI_ADDR_PROP_ATTACHED_PORT);
15977         mptsas_log(mpt, CE_WARN, "mptsas update iport "
15978                    "attached-port failed");
15979         goto smp_create_done;
15980     }
15981 }
15982
15983 smp_create_done:
15984     /*
15985      * If props were setup ok, online the lun
15986     */
15987     if (ndi_rtn == NDI_SUCCESS) {
15988         /*
15989          * Try to online the new node
15990        */
15991         ndi_rtn = ndi_devi_online(*smp_dip, NDI_ONLINE_ATTACH);
15992     }
15993
15994     /*
15995      * If success set rtn flag, else unwire alloc'd lun
15996     */
15997     if (ndi_rtn != NDI_SUCCESS) {
15998         NDBG12(("mptsas unable to online "
15999                  "SMP target %s", wwn_str));
16000         ndi_prop_remove_all(*smp_dip);
16001         (void) ndi_devi_free(*smp_dip);
16002     }
16003 }
16004
16005 return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
16006
16007 }
16008
16009 /* smp transport routine */
16010 static int mptsas_smp_start(struct smp_pkt *smp_pkt)
16011 {
16012     uint64_t           wwn;
16013     Mpi2SmpPassthroughRequest_t    req;
16014     Mpi2SmpPassthroughReply_t     rep;
16015     uint32_t            direction = 0;
16016     mptsas_t             *mpt;
16017     int                  ret;
16018     uint64_t            tmp64;
16019
16020     mpt = (mptsas_t *)smp_pkt->smp_pkt_address->

```

```

16021         smp_a_hba_tran->smp_tran_hba_private;
16022
16023     bcopy(smp_pkt->smp_pkt_address->smp_a_wwn, &wwn, SAS_WWN_BYTE_SIZE);
16024     /*
16025      * Need to compose a SMP request message
16026      * and call mptsas_do_passthru() function
16027     */
16028     bzero(&req, sizeof (req));
16029     bzero(&rep, sizeof (rep));
16030     req.PassthroughFlags = 0;
16031     req.PhysicalPort = 0xff;
16032     req.ChainOffset = 0;
16033     req.Function = MPI2_FUNCTION_SMP_PASSTHROUGH;
16034
16035     if ((smp_pkt->smp_pkt_reqsize & 0xfffff0000ul) != 0) {
16036         smp_pkt->smp_pkt_reason = ERANGE;
16037         return (DDI_FAILURE);
16038     }
16039     req.RequestDataLength = LE_16((uint16_t)(smp_pkt->smp_pkt_reqsize - 4));
16040
16041     req.MsgFlags = 0;
16042     tmp64 = LE_64(wwn);
16043     bcopy(&tmp64, &req.SASAddress, SAS_WWN_BYTE_SIZE);
16044     if (smp_pkt->smp_pkt_rpsize > 0) {
16045         direction |= MPTAS_PASS_THRU_DIRECTION_READ;
16046     }
16047     if (smp_pkt->smp_pkt_reqsize > 0) {
16048         direction |= MPTAS_PASS_THRU_DIRECTION_WRITE;
16049     }
16050
16051     mutex_enter(&mpt->m_mutex);
16052     ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep,
16053                           (uint8_t *)smp_pkt->smp_pkt_rsp,
16054                           offsetof(Mpi2SmpPassthroughRequest_t, SGL), sizeof (rep),
16055                           smp_pkt->smp_pkt_rpsize - 4, direction,
16056                           (uint8_t *)smp_pkt->smp_pkt_req, smp_pkt->smp_pkt_reqsize - 4,
16057                           smp_pkt->smp_pkt_timeout, FKIOCTL);
16058     mutex_exit(&mpt->m_mutex);
16059     if (ret != 0) {
16060         cmn_err(CE_WARN, "smp_start do passthru error %d", ret);
16061         smp_pkt->smp_pkt_reason = (uchar_t)(ret);
16062         return (DDI_FAILURE);
16063     }
16064     /*
16065      * do passthrough success, check the smp status */
16066     if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
16067         switch (LE_16(rep.IOCStatus)) {
16068             case MPI2_IOCSTATUS_SCSI_DEVICE_NOT THERE:
16069                 smp_pkt->smp_pkt_reason = ENODEV;
16070                 break;
16071             case MPI2_IOCSTATUS_SAS_SMP_DATA_OVERRUN:
16072                 smp_pkt->smp_pkt_reason = EOVERRFLOW;
16073                 break;
16074             case MPI2_IOCSTATUS_SAS_SMP_REQUEST_FAILED:
16075                 smp_pkt->smp_pkt_reason = EIO;
16076                 break;
16077             default:
16078                 mptsas_log(mpt, CE_NOTE, "smp_start: get unknown ioc"
16079                            "status:%x", LE_16(rep.IOCStatus));
16080                 smp_pkt->smp_pkt_reason = EIO;
16081                 break;
16082         }
16083         return (DDI_FAILURE);
16084     }
16085     if (rep.SASStatus != MPI2_SASSTATUS_SUCCESS) {
16086         mptsas_log(mpt, CE_NOTE, "smp_start: get error SAS status:%x",
16087                    rep.SASStatus);
16088
16089 }

```

```

16087             smp_pkt->smp_pkt_reason = EIO;
16088         }
16089     }
16090     return (DDI_SUCCESS);
16091 }
16092 }

16094 /*
16095 * If we didn't get a match, we need to get sas page0 for each device, and
16096 * until we get a match. If failed, return NULL
16097 */
16098 static mptsas_target_t *
16099 mptsas_phy_to_tgt(mptsas_t *mpt, mptsas_phymask_t phymask, uint8_t phy)
16100 {
16101     int           i, j = 0;
16102     int           rval = 0;
16103     uint16_t      cur_handle;
16104     uint32_t      page_address;
16105     mptsas_target_t *ptgt = NULL;

16106 /*
16107 * PHY named device must be direct attached and attaches to
16108 * narrow port, if the iport is not parent of the device which
16109 * we are looking for.
16110 */
16111 for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
16112     if ((1 << i) & phymask)
16113         j++;
16114 }

16115 if (j > 1)
16116     return (NULL);

16117 /*
16118 * Must be a narrow port and single device attached to the narrow port
16119 * So the physical port num of device which is equal to the iport's
16120 * port num is the device what we are looking for.
16121 */
16122 if (mpt->m_phy_info[phy].phy_mask != phymask)
16123     return (NULL);

16124 mutex_enter(&mpt->m_mutex);

16125 ptgt = rehash_linear_search(mpt->m_targets, mptsas_target_eval_nowwn,
16126                             &phy);
16127 if (ptgt != NULL) {
16128     mutex_exit(&mpt->m_mutex);
16129     return (ptgt);
16130 }

16131 if (mpt->m_done_traverse_dev) {
16132     mutex_exit(&mpt->m_mutex);
16133     return (NULL);
16134 }

16135 /* If didn't get a match, come here */
16136 cur_handle = mpt->m_dev_handle;
16137 for (; ; ) {
16138     ptgt = NULL;
16139     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
16140                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)cur_handle;
16141     rval = mptsas_get_target_device_info(mpt, page_address,
16142                                         &cur_handle, &ptgt);
16143     if ((rval == DEV_INFO_FAIL_PAGE0) ||
16144         (rval == DEV_INFO_FAIL_ALLOC)) {

```

```

16145             break;
16146         }
16147         if ((rval == DEV_INFO_WRONG_DEVICE_TYPE) ||
16148             (rval == DEV_INFO_PHYS_DISK)) {
16149             continue;
16150         }
16151         mpt->m_dev_handle = cur_handle;
16152
16153         if ((ptgt->m_addr.mta_wwn == 0) && (ptgt->m_phynum == phy)) {
16154             break;
16155         }
16156     }
16157 }
16158 mutex_exit(&mpt->m_mutex);
16159 return (ptgt);
16160 }

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

16176     addr.mta_wwn = wwid;
16177     addr.mta_phymask = phymask;
16178     mutex_enter(&mpt->m_mutex);
16179     tmp_tgt = rehash_lookup(mpt->m_targets, &addr);
16180     if (tmp_tgt != NULL) {
16181         mutex_exit(&mpt->m_mutex);
16182         return (tmp_tgt);
16183     }
16184 }

16185 if (phymask == 0) {
16186     /*
16187     * It's IR volume
16188     */
16189     rval = mptsas_get_raid_info(mpt);
16190     if (rval) {
16191         tmp_tgt = rehash_lookup(mpt->m_targets, &addr);
16192     }
16193     mutex_exit(&mpt->m_mutex);
16194     return (tmp_tgt);
16195 }

16196 if (mpt->m_done_traverse_dev) {
16197     mutex_exit(&mpt->m_mutex);
16198     return (NULL);
16199 }

16200 if (mpt->m_done_traverse_dev) {
16201     mutex_exit(&mpt->m_mutex);
16202     return (NULL);
16203 }

16204 if (mpt->m_done_traverse_dev) {
16205     mutex_exit(&mpt->m_mutex);
16206     return (NULL);
16207 }

16208 if (mpt->m_done_traverse_dev) {
16209     mutex_exit(&mpt->m_mutex);
16210     return (NULL);
16211 }

16212 /* If didn't get a match, come here */
16213 cur_handle = mpt->m_dev_handle;
16214 for (;;) {
16215     tmp_tgt = NULL;
16216     page_address = (MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE &
16217                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | cur_handle;
16218     rval = mptsas_get_target_device_info(mpt, page_address,

```

```

16219             &cur_handle, &tmp_tgt);
16220     if ((rval == DEV_INFO_FAIL_PAGE0) ||
16221         (rval == DEV_INFO_FAIL_ALLOC)) {
16222         tmp_tgt = NULL;
16223         break;
16224     }
16225     if ((rval == DEV_INFO_WRONG_DEVICE_TYPE) ||
16226         (rval == DEV_INFO_PHYS_DISK)) {
16227         continue;
16228     }
16229     mpt->m_dev_handle = cur_handle;
16230     if ((tmp_tgt->m_addr.mta_wwn) &&
16231         (tmp_tgt->m_addr.mta_wwn == wwid) &&
16232         (tmp_tgt->m_addr.mta_phymask == phymask)) {
16233         break;
16234     }
16235 }
16236 mutex_exit(&mpt->m_mutex);
16237 return (tmp_tgt);
16238 }
16239 }

16240 static mptsas_smp_t *
16241 mptsas_wwid_to_psmp(mptsas_t *mpt, mptsas_phymask_t phymask, uint64_t wwid)
16242 {
16243     int          rval = 0;
16244     uint16_t      cur_handle;
16245     uint32_t      page_address;
16246     mptsas_smp_t  smp_node, *psmp = NULL;
16247     mptsas_target_addr_t addr;
16248
16249     addr.mta_wwn = wwid;
16250     addr.mta_phymask = phymask;
16251     mutex_enter(&mpt->m_mutex);
16252     psmp = refhash_lookup(mpt->m_smp_targets, &addr);
16253     if (psmp != NULL) {
16254         mutex_exit(&mpt->m_mutex);
16255         return (psmp);
16256     }
16257
16258     if (mpt->m_done_traverse_smp) {
16259         mutex_exit(&mpt->m_mutex);
16260         return (NULL);
16261     }
16262
16263 /* If didn't get a match, come here */
16264 cur_handle = mpt->m_smp_devhdl;
16265 for (;;) {
16266     psmp = NULL;
16267     page_address = (MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL &
16268                     MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)cur_handle;
16269     rval = mptsas_get_sas_expander_page0(mpt, page_address,
16270                                         &smp_node);
16271     if (rval != DDI_SUCCESS) {
16272         break;
16273     }
16274     mpt->m_smp_devhdl = cur_handle = smp_node.m_devhdl;
16275     psmp = mptsas_smp_alloc(mpt, &smp_node);
16276     ASSERT(psmp);
16277     if ((psmp->m_addr.mta_wwn) && (psmp->m_addr.mta_wwn == wwid) &&
16278         (psmp->m_addr.mta_phymask == phymask)) {
16279         break;
16280     }
16281 }
16282
16283 mutex_exit(&mpt->m_mutex);

```

```

16284
16285         return (psmp);
16286     }
16287
16288     mptsas_target_t *
16289     mptsas_tgt_alloc(mptsas_t *mpt, uint16_t devhdl, uint64_t wwid,
16290                      uint32_t devinfo, mptsas_phymask_t phymask, uint8_t phynum)
16291 {
16292     mptsas_target_t *tmp_tgt = NULL;
16293     mptsas_target_addr_t addr;
16294
16295     addr.mta_wwn = wwid;
16296     addr.mta_phymask = phymask;
16297     tmp_tgt = refhash_lookup(mpt->m_targets, &addr);
16298     if (tmp_tgt != NULL) {
16299         NDBG20(("Hash item already exist"));
16300         tmp_tgt->m_deviceinfo = devinfo;
16301         tmp_tgt->m_devhdl = devhdl; /* XXX - duplicate? */
16302         return (tmp_tgt);
16303     }
16304     tmp_tgt = kmalloc(sizeof (struct mptsas_target), KM_SLEEP);
16305     if (tmp_tgt == NULL) {
16306         cmn_err(CE_WARN, "Fatal, allocated tgt failed");
16307         return (NULL);
16308     }
16309     tmp_tgt->m_devhdl = devhdl;
16310     tmp_tgt->m_addr.mta_wwn = wwid;
16311     tmp_tgt->m_deviceinfo = devinfo;
16312     tmp_tgt->m_addr.mta_phymask = phymask;
16313     tmp_tgt->m_phynum = phynum;
16314     /* Initialized the tgt structure */
16315     tmp_tgt->m_qfull_retries = QFULL_RETRIES;
16316     tmp_tgt->m_qfull_retry_interval =
16317         drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
16318     tmp_tgt->m_t_throttle = MAX_THROTTLE;
16319     TAILQ_INIT(&tmp_tgt->m_active_cmdq);
16320
16321     refhash_insert(mpt->m_targets, tmp_tgt);
16322
16323     return (tmp_tgt);
16324 }
16325
16326 static void
16327 mptsas_smp_target_copy(mptsas_smp_t *src, mptsas_smp_t *dst)
16328 {
16329     dst->m_devhdl = src->m_devhdl;
16330     dst->m_deviceinfo = src->m_deviceinfo;
16331     dst->m_pdevhdl = src->m_pdevhdl;
16332     dst->m_pdevinfo = src->m_pdevinfo;
16333 }
16334
16335 static mptsas_smp_t *
16336 mptsas_smp_alloc(mptsas_t *mpt, mptsas_smp_t *data)
16337 {
16338     mptsas_target_addr_t addr;
16339     mptsas_smp_t *ret_data;
16340
16341     addr.mta_wwn = data->m_addr.mta_wwn;
16342     addr.mta_phymask = data->m_addr.mta_phymask;
16343     ret_data = refhash_lookup(mpt->m_smp_targets, &addr);
16344     /*
16345      * If there's already a matching SMP target, update its fields
16346      * in place. Since the address is not changing, it's safe to do
16347      * this. We cannot just bcopy() here because the structure we've
16348      * been given has invalid hash links.
16349     */
16350     if (ret_data != NULL) {

```

```

16351         mptsas_smp_target_copy(data, ret_data);
16352         return (ret_data);
16353     }
16354
16355     ret_data = kmem_alloc(sizeof (mptsas_smp_t), KM_SLEEP);
16356     bcopy(data, ret_data, sizeof (mptsas_smp_t));
16357     rehash_insert(mpt->m_smp_targets, ret_data);
16358     return (ret_data);
16359 }
16360
16361 /* Functions for SGPIO LED support
16362 */
16363 static dev_info_t *
16364 mptsas_get_dip_from_dev(dev_t dev, mptsas_phymask_t *phymask)
16365 {
16366     dev_info_t      *dip;
16367     int             prop;
16368     dip = e_ddi_hold_devi_by_dev(dev, 0);
16369     if (dip == NULL)
16370         return (dip);
16371     prop = ddi_prop_get_int(DDI_DEV_T_ANY, dip, 0,
16372                            "phymask", 0);
16373     *phymask = (mptsas_phymask_t)prop;
16374     ddi_release_devi(dip);
16375     return (dip);
16376 }
16377 static mptsas_target_t *
16378 mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr, mptsas_phymask_t phymask)
16379 {
16380     uint8_t          phynum;
16381     uint64_t          wwn;
16382     int              lun;
16383     mptsas_target_t  *ptgt = NULL;
16384
16385     if (mptsas_parse_address(addr, &wwn, &phynum, &lun) != DDI_SUCCESS) {
16386         return (NULL);
16387     }
16388     if (addr[0] == 'w') {
16389         ptgt = mptsas_wwid_to_ptgt(mpt, (int)phymask, wwn);
16390     } else {
16391         ptgt = mptsas_phy_to_tgt(mpt, (int)phymask, phynum);
16392     }
16393     return (ptgt);
16394 }
16395
16396 static int
16397 mptsas_flush_led_status(mptsas_t *mpt, mptsas_target_t *ptgt)
16398 {
16399     uint32_t slotstatus = 0;
16400
16401     /* Build an MPI2 Slot Status based on our view of the world */
16402     if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_IDENT - 1)))
16403         slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_IDENTIFY_REQUEST;
16404     if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_FAIL - 1)))
16405         slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_PREDICTED_FAULT;
16406     if (ptgt->m_led_status & (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1)))
16407         slotstatus |= MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;
16408
16409     /* Write it to the controller */
16410     NDBG14(("mptsas_ioctl: set LED status %x for slot %x",
16411            slotstatus, ptgt->m_slot_num));
16412     return (mptsas_send_sep(mpt, ptgt, &slotstatus,
16413                           MPI2_SEP_ACTION_WRITE_STATUS));
16414 }
16415

```

```

16417 /*
16418  * send sep request, use enclosure/slot addressing
16419 */
16420 static int
16421 mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
16422                  uint32_t *status, uint8_t act)
16423 {
16424     Mpi2SepRequest_t    req;
16425     Mpi2SepReply_t     rep;
16426     int                 ret;
16427
16428     ASSERT(mutex_owned(&mpt->m_mutex));
16429
16430     /*
16431      * We only support SEP control of directly-attached targets, in which
16432      * case the "SEP" we're talking to is a virtual one contained within
16433      * the HBA itself. This is necessary because DA targets typically have
16434      * no other mechanism for LED control. Targets for which a separate
16435      * enclosure service processor exists should be controlled via ses(7d)
16436      * or sgen(7d). Furthermore, since such requests can time out, they
16437      * should be made in user context rather than in response to
16438      * asynchronous fabric changes.
16439
16440      * In addition, we do not support this operation for RAID volumes,
16441      * since there is no slot associated with them.
16442
16443     if (!(ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED) ||
16444         ptgt->m_addr.mta_phymask == 0) {
16445         return (ENOTTY);
16446     }
16447
16448     bzero(&req, sizeof (req));
16449     bzero(&rep, sizeof (rep));
16450
16451     req.Function = MPI2_FUNCTION_SCSI_ENCLOSURE_PROCESSOR;
16452     req.Action = act;
16453     req.Flags = MPI2_SEP_REQ_FLAGS_ENCLOSURE_SLOT_ADDRESS;
16454     req.EnclosureHandle = LE_16(ptgt->m_enclosure);
16455     req.Slot = LE_16(ptgt->m_slot_num);
16456     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16457         req.SlotStatus = LE_32(*status);
16458     }
16459     ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
16460                             sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
16461     if (ret != 0) {
16462         mptsas_log(mpt, CE_NOTE, "mptsas_send_sep: passthru SEP "
16463                    "Processor Request message error %d", ret);
16464     }
16465     /* do passthrough success, check the ioc status */
16466     if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
16467         mptsas_log(mpt, CE_NOTE, "send_sep act %x: ioc "
16468                     "status:%x loginfo %x", act, LE_16(rep.IOCStatus),
16469                     LE_32(rep.IOCLoginInfo));
16470         switch (LE_16(rep.IOCStatus) & MPI2_IOCSTATUS_MASK) {
16471             case MPI2_IOCSTATUS_INVALID_FUNCTION:
16472             case MPI2_IOCSTATUS_INVALID_VPID:
16473             case MPI2_IOCSTATUS_INVALID_FIELD:
16474             case MPI2_IOCSTATUS_INVALID_STATE:
16475             case MPI2_IOCSTATUS_OP_STATE_NOT_SUPPORTED:
16476             case MPI2_IOCSTATUS_CONFIG_INVALID_ACTION:
16477             case MPI2_IOCSTATUS_CONFIG_INVALID_TYPE:
16478             case MPI2_IOCSTATUS_CONFIG_INVALID_PAGE:
16479             case MPI2_IOCSTATUS_CONFIG_INVALID_DATA:
16480             case MPI2_IOCSTATUS_CONFIG_NO_DEFAULTS:
16481                 return (EINVAL);
16482         }
16483     }
16484
16485     /* Set the LED status */
16486     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16487         if (status != 0)
16488             ptgt->m_led_status |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16489         if (status != 0)
16490             ptgt->m_led_status |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16491         if (status != 0)
16492             ptgt->m_led_status |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16493     }
16494
16495     /* Set the slot status */
16496     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16497         if (status != 0)
16498             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16499         if (status != 0)
16500             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16501         if (status != 0)
16502             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16503     }
16504
16505     /* Set the target status */
16506     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16507         if (status != 0)
16508             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16509         if (status != 0)
16510             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16511         if (status != 0)
16512             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16513     }
16514
16515     /* Set the enclosure status */
16516     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16517         if (status != 0)
16518             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16519         if (status != 0)
16520             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16521         if (status != 0)
16522             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16523     }
16524
16525     /* Set the global status */
16526     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16527         if (status != 0)
16528             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16529         if (status != 0)
16530             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16531         if (status != 0)
16532             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16533     }
16534
16535     /* Set the port status */
16536     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16537         if (status != 0)
16538             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16539         if (status != 0)
16540             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16541         if (status != 0)
16542             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16543     }
16544
16545     /* Set the host status */
16546     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16547         if (status != 0)
16548             ptgt->m_hoststatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16549         if (status != 0)
16550             ptgt->m_hoststatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16551         if (status != 0)
16552             ptgt->m_hoststatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16553     }
16554
16555     /* Set the device status */
16556     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16557         if (status != 0)
16558             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16559         if (status != 0)
16560             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16561         if (status != 0)
16562             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16563     }
16564
16565     /* Set the slot status */
16566     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16567         if (status != 0)
16568             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16569         if (status != 0)
16570             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16571         if (status != 0)
16572             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16573     }
16574
16575     /* Set the target status */
16576     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16577         if (status != 0)
16578             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16579         if (status != 0)
16580             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16581         if (status != 0)
16582             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16583     }
16584
16585     /* Set the enclosure status */
16586     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16587         if (status != 0)
16588             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16589         if (status != 0)
16590             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16591         if (status != 0)
16592             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16593     }
16594
16595     /* Set the global status */
16596     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16597         if (status != 0)
16598             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16599         if (status != 0)
16600             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16601         if (status != 0)
16602             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16603     }
16604
16605     /* Set the port status */
16606     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16607         if (status != 0)
16608             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16609         if (status != 0)
16610             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16611         if (status != 0)
16612             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16613     }
16614
16615     /* Set the device status */
16616     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16617         if (status != 0)
16618             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16619         if (status != 0)
16620             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16621         if (status != 0)
16622             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16623     }
16624
16625     /* Set the slot status */
16626     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16627         if (status != 0)
16628             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16629         if (status != 0)
16630             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16631         if (status != 0)
16632             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16633     }
16634
16635     /* Set the target status */
16636     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16637         if (status != 0)
16638             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16639         if (status != 0)
16640             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16641         if (status != 0)
16642             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16643     }
16644
16645     /* Set the enclosure status */
16646     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16647         if (status != 0)
16648             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16649         if (status != 0)
16650             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16651         if (status != 0)
16652             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16653     }
16654
16655     /* Set the global status */
16656     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16657         if (status != 0)
16658             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16659         if (status != 0)
16660             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16661         if (status != 0)
16662             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16663     }
16664
16665     /* Set the port status */
16666     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16667         if (status != 0)
16668             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16669         if (status != 0)
16670             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16671         if (status != 0)
16672             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16673     }
16674
16675     /* Set the device status */
16676     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16677         if (status != 0)
16678             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16679         if (status != 0)
16680             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16681         if (status != 0)
16682             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16683     }
16684
16685     /* Set the slot status */
16686     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16687         if (status != 0)
16688             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16689         if (status != 0)
16690             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16691         if (status != 0)
16692             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16693     }
16694
16695     /* Set the target status */
16696     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16697         if (status != 0)
16698             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16699         if (status != 0)
16700             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16701         if (status != 0)
16702             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16703     }
16704
16705     /* Set the enclosure status */
16706     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16707         if (status != 0)
16708             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16709         if (status != 0)
16710             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16711         if (status != 0)
16712             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16713     }
16714
16715     /* Set the global status */
16716     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16717         if (status != 0)
16718             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16719         if (status != 0)
16720             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16721         if (status != 0)
16722             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16723     }
16724
16725     /* Set the port status */
16726     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16727         if (status != 0)
16728             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16729         if (status != 0)
16730             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16731         if (status != 0)
16732             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16733     }
16734
16735     /* Set the device status */
16736     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16737         if (status != 0)
16738             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16739         if (status != 0)
16740             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16741         if (status != 0)
16742             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16743     }
16744
16745     /* Set the slot status */
16746     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16747         if (status != 0)
16748             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16749         if (status != 0)
16750             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16751         if (status != 0)
16752             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16753     }
16754
16755     /* Set the target status */
16756     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16757         if (status != 0)
16758             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16759         if (status != 0)
16760             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16761         if (status != 0)
16762             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16763     }
16764
16765     /* Set the enclosure status */
16766     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16767         if (status != 0)
16768             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16769         if (status != 0)
16770             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16771         if (status != 0)
16772             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16773     }
16774
16775     /* Set the global status */
16776     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16777         if (status != 0)
16778             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16779         if (status != 0)
16780             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16781         if (status != 0)
16782             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16783     }
16784
16785     /* Set the port status */
16786     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16787         if (status != 0)
16788             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16789         if (status != 0)
16790             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16791         if (status != 0)
16792             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16793     }
16794
16795     /* Set the device status */
16796     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16797         if (status != 0)
16798             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16799         if (status != 0)
16800             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16801         if (status != 0)
16802             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16803     }
16804
16805     /* Set the slot status */
16806     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16807         if (status != 0)
16808             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16809         if (status != 0)
16810             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16811         if (status != 0)
16812             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16813     }
16814
16815     /* Set the target status */
16816     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16817         if (status != 0)
16818             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16819         if (status != 0)
16820             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16821         if (status != 0)
16822             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16823     }
16824
16825     /* Set the enclosure status */
16826     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16827         if (status != 0)
16828             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16829         if (status != 0)
16830             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16831         if (status != 0)
16832             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16833     }
16834
16835     /* Set the global status */
16836     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16837         if (status != 0)
16838             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16839         if (status != 0)
16840             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16841         if (status != 0)
16842             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16843     }
16844
16845     /* Set the port status */
16846     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16847         if (status != 0)
16848             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16849         if (status != 0)
16850             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16851         if (status != 0)
16852             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16853     }
16854
16855     /* Set the device status */
16856     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16857         if (status != 0)
16858             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16859         if (status != 0)
16860             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16861         if (status != 0)
16862             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16863     }
16864
16865     /* Set the slot status */
16866     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16867         if (status != 0)
16868             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16869         if (status != 0)
16870             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16871         if (status != 0)
16872             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16873     }
16874
16875     /* Set the target status */
16876     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16877         if (status != 0)
16878             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16879         if (status != 0)
16880             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16881         if (status != 0)
16882             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16883     }
16884
16885     /* Set the enclosure status */
16886     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16887         if (status != 0)
16888             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16889         if (status != 0)
16890             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16891         if (status != 0)
16892             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16893     }
16894
16895     /* Set the global status */
16896     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16897         if (status != 0)
16898             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16899         if (status != 0)
16900             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16901         if (status != 0)
16902             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16903     }
16904
16905     /* Set the port status */
16906     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16907         if (status != 0)
16908             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16909         if (status != 0)
16910             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16911         if (status != 0)
16912             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16913     }
16914
16915     /* Set the device status */
16916     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16917         if (status != 0)
16918             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16919         if (status != 0)
16920             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16921         if (status != 0)
16922             ptgt->m_devicestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16923     }
16924
16925     /* Set the slot status */
16926     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16927         if (status != 0)
16928             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16929         if (status != 0)
16930             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16931         if (status != 0)
16932             ptgt->m_slotstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16933     }
16934
16935     /* Set the target status */
16936     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16937         if (status != 0)
16938             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16939         if (status != 0)
16940             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16941         if (status != 0)
16942             ptgt->m_targetstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16943     }
16944
16945     /* Set the enclosure status */
16946     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16947         if (status != 0)
16948             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16949         if (status != 0)
16950             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16951         if (status != 0)
16952             ptgt->m_enclosurestatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16953     }
16954
16955     /* Set the global status */
16956     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16957         if (status != 0)
16958             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 1));
16959         if (status != 0)
16960             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_FAIL - 1));
16961         if (status != 0)
16962             ptgt->m_globalstatus |= (1 << (MPTSAS_LEDCTL_LED_OK2RM - 1));
16963     }
16964
16965     /* Set the port status */
16966     if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16967         if (status != 0)
16968             ptgt->m_portstatus |= (1 << (MPTSAS_LEDCTL_LED_IDENT - 
```

```
16483         case MPI2_IOCSTATUS_BUSY:
16484             return (EBUSY);
16485         case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
16486             return (EAGAIN);
16487         case MPI2_IOCSTATUS_INVALID_SGL:
16488         case MPI2_IOCSTATUS_INTERNAL_ERROR:
16489         case MPI2_IOCSTATUS_CONFIG_CANT_COMMIT:
16490         default:
16491             return (EIO);
16492     }
16493 }
16494 if (act != MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16495     *status = LE_32(rep.SlotStatus);
16496 }
16497
16498 return (0);
16499 }
16500 int
16501 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
16502     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
16503     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
16504 {
16505     ddi_dma_cookie_t      new_cookie;
16506     size_t                alloc_len;
16507     uint_t                ncookie;
16508
16509     if (cookiep == NULL)
16510         cookiep = &new_cookie;
16511
16512     if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
16513         NULL, dma_hdp) != DDI_SUCCESS) {
16514         return (FALSE);
16515     }
16516
16517     if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
16518         DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
16519         acc_hdp) != DDI_SUCCESS) {
16520         ddi_dma_free_handle(dma_hdp);
16521         *dma_hdp = NULL;
16522         return (FALSE);
16523     }
16524
16525     if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
16526         (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
16527         cookiep, &ncookie) != DDI_DMA_MAPPED) {
16528         (void) ddi_dma_mem_free(acc_hdp);
16529         ddi_dma_free_handle(dma_hdp);
16530         *dma_hdp = NULL;
16531         return (FALSE);
16532     }
16533 }
16534
16535     return (TRUE);
16536 }
16537
16538 void
16539 mptsas_dma_addr_destroy(ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp)
16540 {
16541     if (*dma_hdp == NULL)
16542         return;
16543
16544     (void) ddi_dma_unbind_handle(*dma_hdp);
16545     (void) ddi_dma_mem_free(acc_hdp);
16546     ddi_dma_free_handle(dma_hdp);
16547     *dma_hdp = NULL;
16548 }
```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
*****
83729 Fri Dec 19 13:49:24 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
First pass at 4310
*****
unchanged_portion_omitted

1080 /*
1081 * NOTE: We should be able to queue TM requests in the controller to make this
1082 * a lot faster. If resetting all targets, for example, we can load the hi
1083 * priority queue with its limit and the controller will reply as they are
1084 * completed. This way, we don't have to poll for one reply at a time.
1085 * Think about enhancing this later.
1086 */
1087 int
1088 mptsas_ioc_task_management(mptsas_t *mpt, int task_type, uint16_t dev_handle,
1089     int lun, uint8_t *reply, uint32_t reply_size, int mode)
1090 {
1091     /*
1092     * In order to avoid allocating variables on the stack,
1093     * we make use of the pre-existing mptsas_cmd_t and
1094     * scsi_pkt which are included in the mptsas_t which
1095     * is passed to this routine.
1096     */
1097
1098     pMpI2SCSITaskManagementRequest_t          task;
1099     int                                     rval = FALSE;
1100     mptsas_cmd_t                            *cmd;
1101     struct scsi_pkt                         *pkt;
1102     mptsas_slots_t                          *slots = mpt->m_active;
1103     uint64_t                                request_desc, i;
1104     pMpI2DefaultReply_t                     reply_msg;
1105
1106     /*
1107     * Can't start another task management routine.
1108     */
1109     if (slots->m_slot[MPTSAS_TM_SLOT(mpt)] != NULL) {
1110         mptsas_log(mpt, CE_WARN, "Can only start 1 task management"
1111                     " command at a time\n");
1112         return (FALSE);
1113     }
1114
1115     cmd = &(mpt->m_event_task_mgmt.m_event_cmd);
1116     pkt = &(mpt->m_event_task_mgmt.m_event_pkt);
1117
1118     bzero((caddr_t)cmd, sizeof (*cmd));
1119     bzero((caddr_t)pkt, scsi_pkt_size());
1120
1121     pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
1122     pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
1123     pkt->pkt_ha_private     = (opaque_t)cmd;
1124     pkt->pkt_flags          = (FLAG_NOINTR | FLAG_HEAD);
1125     pkt->pkt_time            = 60;
1126     pkt->pkt_address.a_target = dev_handle;
1127     pkt->pkt_address.a_lun   = (uchar_t)lun;
1128     cmd->cmd_pkt             = pkt;
1129     cmd->cmd_scrlen          = 1;
1130     cmd->cmd_flags            = CFLAG_TM_CMD;
1131     cmd->cmd_slot              = MPTSAS_TM_SLOT(mpt);
1132
1133     slots->m_slot[MPTSAS_TM_SLOT(mpt)] = cmd;
1134
1135     /*
1136     * Store the TM message in memory location corresponding to the TM slot
1137     * number.
1138     */

```

```
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c

1139         task = (pMpI2SCSITaskManagementRequest_t)(mpt->m_req_frame +
1140             (mpt->m_req_frame_size * cmd->cmd_slot));
1141             bzero(task, mpt->m_req_frame_size);

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

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

1154         /*
1155             * Send TM request using High Priority Queue.
1156             */
1157         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1158             DDI_DMA_SYNC_FORDEV);
1159         request_desc = (cmd->cmd_slot << 16) +
1160             MPI2_REQ_DESCRIPTOR_FLAGS_HIGH_PRIORITY;
1161         MPTSAS_START_CMD(mpt, request_desc);
1162         rval = mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);

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

1167     /*
1168         * If a reply frame was used and there is a reply buffer to copy the
1169         * reply data into, copy it. If this fails, log a message, but don't
1170         * fail the TM request.
1171         */
1172     if (cmd->cmd_rfm && reply) {
1173         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
1174             DDI_DMA_SYNC_FORCPU);
1175         reply_msg = (pMpI2DefaultReply_t)
1176             (mpt->m_reply_frame + (cmd->cmd_rfm -
1177                 (mpt->m_reply_frame_dma_addr & 0xfffffffffu)));
1178         if (reply_size > sizeof(MPI2_SCSI_TASK_MANAGE_REPLY)) {
1179             reply_size = sizeof(MPI2_SCSI_TASK_MANAGE_REPLY);
1180         }
1181         mutex_exit(&mpt->m_mutex);
1182         for (i = 0; i < reply_size; i++) {
1183             if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
1184                 mode)) {
1185                 mptsas_log(mpt, CE_WARN, "failed to copy out "
1186                     "reply data for TM request");
1187                 break;
1188             }
1189         }
1190         mutex_enter(&mpt->m_mutex);
1191     }

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

1198     /*
1199         * If we lost our task management command
1200         * we need to reset the ioc
1201         */
1202     if (rval == FALSE) {
1203         mptsas_log(mpt, CE_WARN, "mptsas_ioc_task_management failed "
1204             "try to reset ioc to recovery!");
1205     }
1206 }
```

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c

1205     mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1206     /* Let's try this instead of the old codepath commented below...
1207     ddi_taskq_dispatch(mpt->m_reset_taskq, mptsas_handle_restart_ioc
1208     mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc dispatch attempted");
1209     rval = FAILED;
1210    */
1211    if (mptsas_restart_ioc(mpt)) {
1212        if (mptsas_restart_ioc(mpt)) {
1213            mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1214            rval = FAILED;
1215        }
1216    }
1217
1218    return (rval);
1219 }

1220 /**
1221  * Complete firmware download frame for v2.0 cards.
1222  */
1223 static void
1224 mptsas_uflash2(pMpi2FWDownloadRequest fwdownload,
1225                 ddi_acc_handle_t acc_hdl, uint32_t size, uint8_t type,
1226                 ddi_dma_cookie_t flsh_cookie)
1227 {
1228     pMpi2FWDownloadTCSGE_t tcsge;
1229     pMpi2SGESimple64_t sge;
1230     uint32_t flagslength;
1231
1232     ddi_put8(acc_hdl, &fwdownload->Function,
1233             MPI2_FUNCTION_FW_DOWNLOAD);
1234     ddi_put8(acc_hdl, &fwdownload->ImageType, type);
1235     ddi_put8(acc_hdl, &fwdownload->MsgFlags,
1236             MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1237     ddi_put32(acc_hdl, &fwdownload->TotalImageSize, size);
1238
1239     tcsge = (pMpi2FWDownloadTCSGE_t)&fwdownload->SGL;
1240     ddi_put8(acc_hdl, &tcsge->ContextSize, 0);
1241     ddi_put8(acc_hdl, &tcsge->DetailsLength, 12);
1242     ddi_put8(acc_hdl, &tcsge->Flags, 0);
1243     ddi_put32(acc_hdl, &tcsge->ImageOffset, 0);
1244     ddi_put32(acc_hdl, &tcsge->ImageSize, size);
1245
1246     sge = (pMpi2SGESimple64_t)(tcsge + 1);
1247     flagslength = size;
1248     flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
1249                               MPI2_SGE_FLAGS_END_OF_BUFFER |
1250                               MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
1251                               MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
1252                               MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
1253                               MPI2_SGE_FLAGS_HOST_TO_IOC |
1254                               MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
1255     ddi_put32(acc_hdl, &sge->FlagsLength, flagslength);
1256     ddi_put32(acc_hdl, &sge->Address.Low,
1257                flsh_cookie.dmac_address);
1258     ddi_put32(acc_hdl, &sge->Address.High,
1259                (uint32_t)(flsh_cookie.dmac_laddress >> 32));
1260 }
1261
1262 /**
1263  * Complete firmware download frame for v2.5 cards.
1264  */
1265 static void
1266 mptsas_uflash25(pMpi25FWDownloadRequest fwdownload,
1267                  ddi_acc_handle_t acc_hdl, uint32_t size, uint8_t type,
1268                  ddi_dma_cookie_t flsh_cookie)
1269

```

3

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c

1270 {
1271     pMpi2IeeeSgeSimple64_t sge;
1272     uint8_t flags;
1273
1274     ddi_put8(acc_hdl, &fwdownload->Function,
1275             MPI2_FUNCTION_FW_DOWNLOAD);
1276     ddi_put8(acc_hdl, &fwdownload->ImageType, type);
1277     ddi_put8(acc_hdl, &fwdownload->MsgFlags,
1278             MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1279     ddi_put32(acc_hdl, &fwdownload->TotalImageSize, size);
1280
1281     ddi_put32(acc_hdl, &fwdownload->ImageOffset, 0);
1282     ddi_put32(acc_hdl, &fwdownload->ImageSize, size);
1283
1284     sge = (pMpi2IeeeSgeSimple64_t)&fwdownload->SGL;
1285     flags = MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
1286             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR |
1287             MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
1288     ddi_put8(acc_hdl, &sge->Flags, flags);
1289     ddi_put32(acc_hdl, &sge->Length, size);
1290     ddi_put32(acc_hdl, &sge->Address.Low,
1291               flsh_cookie.dmac_address);
1292     ddi_put32(acc_hdl, &sge->Address.High,
1293               (uint32_t)(flsh_cookie.dmac_laddress >> 32));
1294 }
1295
1296 static int mptsas_enable_mpi25_flashupdate = 0;
1297
1298 int
1299 mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
1300                      uint8_t type, int mode)
1301 {
1302
1303     /*
1304      * In order to avoid allocating variables on the stack,
1305      * we make use of the pre-existing mptsas_cmd_t and
1306      * scsi_pkt which are included in the mptsas_t which
1307      * is passed to this routine.
1308      */
1309
1310     ddi_dma_attr_t flsh_dmaAttrs;
1311     ddi_dma_cookie_t flsh_cookie;
1312     ddi_dma_handle_t flsh_dmaHandle;
1313     ddi_acc_handle_t flsh_accessSp;
1314     caddr_t memp, flsh_memp;
1315     mptsas_cmd_t *cmd;
1316     struct scsi_pkt *pkt;
1317     int i;
1318     int rvalue = 0;
1319     uint64_t request_desc;
1320
1321     if (mpt->m_MPI25 && !mptsas_enable_mpi25_flashupdate) {
1322         /*
1323          * The code is there but not tested yet.
1324          * User has to know there are risks here.
1325          */
1326         mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): "
1327                    "Updating firmware through MPI 2.5 has not been "
1328                    "tested yet!\n"
1329                    "To enable set mptsas_enable_mpi25_flashupdate to 1\n");
1330         return (-1);
1331     } /* Otherwise, you pay your money and you take your chances. */
1332
1333     if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt)) == -1) {
1334         mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): allocation "
1335                    "failed. event ack command pool is full\n");
1336

```

4

```

1336         return (rvalue);
1337     }
1338
1339     bzero((caddr_t)cmd, sizeof (*cmd));
1340     bzero((caddr_t)pkt, scsi_pkt_size());
1341     cmd->ioc_cmd_slot = (uint32_t)rvalue;
1342
1343     /*
1344      * dynamically create a customized dma attribute structure
1345      * that describes the flash file.
1346      */
1347     flsh_dma_attrs = mpt->m_msg_dma_attr;
1348     flsh_dma_attrs.dma_attr_sglen = 1;
1349
1350     if (mptsas_dma_addr_create(mpt, flsh_dma_attrs, &flsh_dma_handle,
1351                               &flsh_accesssp, &flsh_memp, size, &flsh_cookie) == FALSE) {
1352         mptsas_log(mpt, CE_WARN,
1353                    "(unable to allocate dma resource.");
1354         mptsas_return_to_pool(mpt, cmd);
1355         return (-1);
1356     }
1357
1358     bzero(flsh_memp, size);
1359
1360     for (i = 0; i < size; i++) {
1361         (void) ddi_copyin(ptrbuffer + i, flsh_memp + i, 1, mode);
1362     }
1363     (void) ddi_dma_sync(flsh_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);
1364
1365     /*
1366      * form a cmd/pkt to store the fw download message
1367      */
1368     pkt->pkt_cdbp          = (opaque_t)&cmd->cmd_cdb[0];
1369     pkt->pkt_scbp          = (opaque_t)&cmd->cmd_scb;
1370     pkt->pkt_ha_private    = (opaque_t)cmd;
1371     pkt->pkt_flags         = FLAG_HEAD;
1372     pkt->pkt_time          = 60;
1373     cmd->cmd_pkt            = pkt;
1374     cmd->cmd_scrlen         = 1;
1375     cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_CMD;
1376
1377     /*
1378      * Save the command in a slot
1379      */
1380     if (mptsas_save_cmd(mpt, cmd) == FALSE) {
1381         mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accesssp);
1382         mptsas_return_to_pool(mpt, cmd);
1383         return (-1);
1384     }
1385
1386     /*
1387      * Fill in fw download message
1388      */
1389     ASSERT(cmd->cmd_slot != 0);
1390     memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
1391     bzero(memp, mpt->m_req_frame_size);
1392
1393     if (mpt->m_MPI25)
1394         mptsas_uflash25((pMpi25FWDownloadRequest)memp,
1395                         mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);
1396     else
1397         mptsas_uflash2((pMpi2FWDownloadRequest)memp,
1398                         mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);
1399
1400     /*
1401      * Start command

```

```

1402         */
1403         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1404                             DDI_DMA_SYNC_FORDEV);
1405         request_desc = (cmd->cmd_slot << 16) +
1406                         MPI2_REQ_DESCRIPTOR_FLAGS_DEFAULT_TYPE;
1407         cmd->cmd_rfm = NULL;
1408         MPTSAS_START_CMD(mpt, request_desc);
1409
1410         rvalue = 0;
1411         (void) cv_reltimedwait(&mpt->m_fw_cv, &mpt->m_mutex,
1412                               drv_usectohz(60 * MICROSEC), TR_CLOCK_TICK);
1413         if (!(cmd->cmd_flags & CFLAG_FINISHED)) {
1414             mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1415             if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
1416                 mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1417             }
1418             rvalue = -1;
1419         }
1420         mptsas_remove_cmd(mpt, cmd);
1421         mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accesssp);
1422
1423     }
1424
1425     static int
1426     mptsas_sasdevpage_0_cb(mptsas_t *mpt, caddr_t page_memp,
1427                           ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1428                           va_list ap)
1429     {
1430 #ifndef _lock_lint
1431     __NOTE(ARGUNUSED(ap))
1432 #endif
1433     pMpi2SasDevicePage0_t    sasdevpage;
1434     int                      rval = DDI_SUCCESS, i;
1435     uint8_t                  *sas_addr = NULL;
1436     uint8_t                  tmp_sas_wwn[SAS_WWN_BYTE_SIZE];
1437     uint16_t                 *devhdl, *bay_num, *enclosure;
1438     uint64_t                 *sas_wwn;
1439     uint32_t                 *dev_info;
1440     uint8_t                  *physport, *phynum;
1441     uint16_t                 *pdevhdl, *io_flags;
1442     uint32_t                 page_address;
1443
1444     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
1445         (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
1446         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_device_page0 "
1447                    "header: IOCStatus=0x%x, IOCLogInfo=0x%x",
1448                    iocstatus, iocloginfo);
1449         rval = DDI_FAILURE;
1450         return (rval);
1451     }
1452     page_address = va_arg(ap, uint32_t);
1453
1454     /*
1455      * The INVALID_PAGE status is normal if using GET_NEXT_HANDLE and there
1456      * are no more pages. If everything is OK up to this point but the
1457      * status is INVALID_PAGE, change rval to FAILURE and quit. Also,
1458      * signal that device traversal is complete.
1459      */
1460     if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
1461         if ((page_address & MPI2_SAS_DEVICE_PGAD_FORM_MASK) ==
1462             MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE) {
1463             mpt->m_done_traverse_dev = 1;
1464         }
1465         rval = DDI_FAILURE;
1466     }
1467 }
```

```

1468     devhdl = va_arg(ap, uint16_t *);
1469     sas_wwn = va_arg(ap, uint64_t *);
1470     dev_info = va_arg(ap, uint32_t *);
1471     physport = va_arg(ap, uint8_t *);
1472     phynum = va_arg(ap, uint8_t *);
1473     pdevhdl = va_arg(ap, uint16_t *);
1474     bay_num = va_arg(ap, uint16_t *);
1475     enclosure = va_arg(ap, uint16_t *);
1476     io_flags = va_arg(ap, uint16_t *);

1478     sasdevpage = (pMpI2SasDevicePage0_t)page_memp;

1480     *dev_info = ddi_get32(accesssp, &sasdevpage->DeviceInfo);
1481     *devhdl = ddi_get16(accesssp, &sasdevpage->DevHandle);
1482     sas_addr = (uint8_t *)(&sasdevpage->SASAddress);
1483     for (i = 0; i < SAS_WWN_BYTE_SIZE; i++) {
1484         tmp_sas_wwn[i] = ddi_get8(accesssp, sas_addr + i);
1485     }
1486     bcopy(tmp_sas_wwn, sas_wwn, SAS_WWN_BYTE_SIZE);
1487     *sas_wwn = LE_64(*sas_wwn);
1488     *physport = ddi_get8(accesssp, &sasdevpage->PhysicalPort);
1489     *phynum = ddi_get8(accesssp, &sasdevpage->PhyNum);
1490     *pdevhdl = ddi_get16(accesssp, &sasdevpage->ParentDevHandle);
1491     *bay_num = ddi_get16(accesssp, &sasdevpage->Slot);
1492     *enclosure = ddi_get16(accesssp, &sasdevpage->EnclosureHandle);
1493     *io_flags = ddi_get16(accesssp, &sasdevpage->Flags);

1495     if (*io_flags & MPI25_SAS_DEVICE0_FLAGS_FAST_PATH_CAPABLE) {
1496         /*
1497          * Leave a messages about FP capability in the log.
1498          */
1499         mptsas_log(mpt, CE_CONT,
1500                    "%!w%16PRIx64" FastPath Capable%s", *sas_wwn,
1501                    (*io_flags &
1502                     MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH)?
1503                    " and Enabled":" but Disabled");
1504     }

1506     return (rval);
1507 }

1509 /*
1510  * Request MPI configuration page SAS device page 0 to get DevHandle, device
1511  * info and SAS address.
1512 */
1513 int
1514 mptsas_get_sas_device_page0(mptsas_t *mpt, uint32_t page_address,
1515     uint16_t *dev_handle, uint64_t *sas_wwn, uint32_t *dev_info,
1516     uint8_t *physport, uint8_t *phynum, uint16_t *pdev_handle,
1517     uint16_t *bay_num, uint16_t *enclosure, uint16_t *io_flags)
1518 {
1519     int rval = DDI_SUCCESS;
1520
1521     ASSERT(mutex_owned(&mpt->m_mutex));
1522
1523     /*
1524      * Get the header and config page. reply contains the reply frame,
1525      * which holds status info for the request.
1526      */
1527     rval = mptsas_access_config_page(mpt,
1528         MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
1529         MPI2_CONFIG_EXTPAGETYPE_SAS_DEVICE, 0, page_address,
1530         mptsas_sasdevpage_0_cb, page_address, dev_handle, sas_wwn,
1531         dev_info, physport, phynum, pdev_handle,
1532         bay_num, enclosure, io_flags);

```

```

1534         return (rval);
1535     }

1537     static int
1538     mptsas_sasexpdpage_0_cb(mptsas_t *mpt, caddr_t page_memp,
1539         ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1540         va_list ap)
1541     {
1542 #ifndef __lock_lint
1543     _NOTE(ARGUNUSED(ap))
1544 #endif
1545     pMpI2ExpanderPage0_t expddevpage;
1546     int rval = DDI_SUCCESS, i;
1547     uint8_t *sas_addr = NULL;
1548     uint8_t *tmp_sas_wwn[SAS_WWN_BYTE_SIZE];
1549     uint16_t *devhdl;
1550     uint64_t *sas_wwn;
1551     uint8_t *physport;
1552     mptsas_phymask_t *phymask;
1553     uint16_t *pdevhdl;
1554     uint32_t *page_address;

1556     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
1557         (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
1558         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_expander_page0 "
1559                    "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
1560                    iocstatus, iocloginfo);
1561         rval = DDI_FAILURE;
1562     }
1563     return (rval);
1564 }
1565 page_address = va_arg(ap, uint32_t_t);
1566 /*
1567  * The INVALID_PAGE status is normal if using GET_NEXT_HANDLE and there
1568  * are no more pages. If everything is OK up to this point but the
1569  * status is INVALID_PAGE, change rval to FAILURE and quit. Also,
1570  * signal that device traversal is complete.
1571 */
1572 if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
1573     if ((page_address & MPI2_SAS_EXPAND_PGAD_FORM_MASK) ==
1574         MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL) {
1575         mpt->m_done_traverse_smp = 1;
1576     }
1577     rval = DDI_FAILURE;
1578 }
1579 devhdl = va_arg(ap, uint16_t *);
1580 sas_wwn = va_arg(ap, uint64_t *);
1581 phymask = va_arg(ap, mptsas_phymask_t *);
1582 pdevhdl = va_arg(ap, uint16_t *);

1584 expddevpage = (pMpI2ExpanderPage0_t)page_memp;

1585 *devhdl = ddi_get16(accesssp, &expddevpage->DevHandle);
1586 physport = ddi_get8(accesssp, &expddevpage->PhysicalPort);
1587 *phymask = mptsas_physport_to_phymask(mpt, physport);
1588 *pdevhdl = ddi_get16(accesssp, &expddevpage->ParentDevHandle);
1589 sas_addr = (uint8_t *)(&expddevpage->SASAddress);
1590 for (i = 0; i < SAS_WWN_BYTE_SIZE; i++) {
1591     tmp_sas_wwn[i] = ddi_get8(accesssp, sas_addr + i);
1592 }
1593 bcopy(tmp_sas_wwn, sas_wwn, SAS_WWN_BYTE_SIZE);
1594 *sas_wwn = LE_64(*sas_wwn);

1597 return (rval);
1598 }
```

```

1600 /*
1601  * Request MPI configuration page SAS device page 0 to get DevHandle, phymask
1602  * and SAS address.
1603  */
1604 int
1605 mptsas_get_sas_expander_page0(mptsas_t *mpt, uint32_t page_address,
1606     mptsas_smp_t *info)
1607 {
1608     int                     rval = DDI_SUCCESS;
1609
1610     ASSERT(mutex_owned(&mpt->m_mutex));
1611
1612     /*
1613      * Get the header and config page. reply contains the reply frame,
1614      * which holds status info for the request.
1615      */
1616     rval = mptsas_access_config_page(mpt,
1617         MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
1618         MPI2_CONFIG_EXTPAGETYPE_SAS_EXPANDER, 0, page_address,
1619         mptsas_sasexpdpage_0_cb, page_address, &info->m_devhdl,
1620         &info->m_addr.mta_wwn, &info->m_addr.mta_phymask, &info->m_pdevhdl);
1621
1622     return (rval);
1623 }
1624
1625 static int
1626 mptsas_sasportpage_0_cb(mptsas_t *mpt, caddr_t page_memp,
1627     ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1628     va_list ap)
1629 {
1630 #ifndef __lock_lint
1631     _NOTE(ARGUNUSED(ap))
1632 #endif
1633     int     rval = DDI_SUCCESS, i;
1634     uint8_t *sas_addr = NULL;
1635     uint64_t *sas_wwn;
1636     uint8_t tmp_sas_wwn[SAS_WWN_BYTE_SIZE];
1637     uint8_t *portwidth;
1638     pMpI2SasPortPage0_t sasportpage;
1639
1640     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
1641         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_port_page0 "
1642             "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
1643             iocstatus, iocloginfo);
1644     rval = DDI_FAILURE;
1645     return (rval);
1646 }
1647     sas_wwn = va_arg(ap, uint64_t *);
1648     portwidth = va_arg(ap, uint8_t *);
1649
1650     sasportpage = (pMpI2SasPortPage0_t)page_memp;
1651     sas_addr = (uint8_t *)(&sasportpage->SASAddress);
1652     for (i = 0; i < SAS_WWN_BYTE_SIZE; i++) {
1653         tmp_sas_wwn[i] = ddi_get8(accesssp, sas_addr + i);
1654     }
1655     bcopy(tmp_sas_wwn, sas_wwn, SAS_WWN_BYTE_SIZE);
1656     *sas_wwn = LE_64(*sas_wwn);
1657     *portwidth = ddi_get8(accesssp, &sasportpage->PortWidth);
1658     return (rval);
1659 }
1660
1661 /*
1662  * Request MPI configuration page SAS port page 0 to get initiator SAS address
1663  * and port width.
1664 */
1665 int

```

```

1666 mptsas_get_sas_port_page0(mptsas_t *mpt, uint32_t page_address,
1667     uint64_t *sas_wwn, uint8_t *portwidth)
1668 {
1669     int rval = DDI_SUCCESS;
1670
1671     ASSERT(mutex_owned(&mpt->m_mutex));
1672
1673     /*
1674      * Get the header and config page. reply contains the reply frame,
1675      * which holds status info for the request.
1676      */
1677     rval = mptsas_access_config_page(mpt,
1678         MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
1679         MPI2_CONFIG_EXTPAGETYPE_SAS_PORT, 0, page_address,
1680         mptsas_sasportpage_0_cb, sas_wwn, portwidth);
1681
1682     return (rval);
1683 }
1684
1685 static int
1686 mptsas_sasiou_page_0_cb(mptsas_t *mpt, caddr_t page_memp,
1687     ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1688     va_list ap)
1689 {
1690 #ifndef __lock_lint
1691     _NOTE(ARGUNUSED(ap))
1692 #endif
1693     int rval = DDI_SUCCESS;
1694     pMpI2SasIOUnitPage0_t sasioupage0;
1695     int i, num_phys;
1696     uint32_t cpdi[MPTSAS_MAX_PHYS], *retrypage0, *readpage1;
1697     uint8_t port_flags;
1698
1699     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
1700         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_io_unit_page0 "
1701             "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
1702             iocstatus, iocloginfo);
1703     rval = DDI_FAILURE;
1704     return (rval);
1705 }
1706     readpage1 = va_arg(ap, uint32_t *);
1707     retrypage0 = va_arg(ap, uint32_t *);
1708
1709     sasioupage0 = (pMpI2SasIOUnitPage0_t)page_memp;
1710
1711     num_phys = ddi_get8(accesssp, &sasioupage0->NumPhys);
1712
1713     /*
1714      * ASSERT that the num_phys value in SAS IO Unit Page 0 is the same as
1715      * was initially set. This should never change throughout the life of
1716      * the driver.
1717      */
1718     ASSERT(num_phys == mpt->m_num_phys);
1719     for (i = 0; i < num_phys; i++) {
1720         cpdi[i] = ddi_get32(accesssp,
1721             &sasioupage0->PhyData[i].ControllerPhyDeviceInfo);
1722         port_flags = ddi_get8(accesssp,
1723             &sasioupage0->PhyData[i].PortFlags);
1724         mpt->m_phys_info[i].port_num =
1725             ddi_get8(accesssp,
1726             &sasioupage0->PhyData[i].Port);
1727         mpt->m_phys_info[i].ctrlr_devhdl =
1728             ddi_get16(accesssp, &sasioupage0->
1729             PhyData[i].ControllerDevHandle);
1730         mpt->m_phys_info[i].attached_devhdl =
1731             ddi_get16(accesssp, &sasioupage0->

```

```

1732             PhyData[i].AttachedDevHandle);
1733     mpt->m_phy_info[i].phy_device_type = cpdi[i];
1734     mpt->m_phy_info[i].port_flags = port_flags;
1735
1736     if (port_flags & DISCOVERY_IN_PROGRESS) {
1737         *retrypage0 = *retrypage0 + 1;
1738         break;
1739     } else {
1740         *retrypage0 = 0;
1741     }
1742     if (!(port_flags & AUTO_PORT_CONFIGURATION)) {
1743         /*
1744          * some PHY configuration described in
1745          * SAS IO Unit Page1
1746          */
1747         *readpage1 = 1;
1748     }
1749 }
1750
1751     return (rval);
1752 }
1753
1754 static int
1755 mptsas_sasiou_page_1_cb(mptsas_t *mpt, caddr_t page_memp,
1756 ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1757 va_list ap)
1758 {
1759 #ifndef _lock_lint
1760     _NOTE(ARGUNUSED(ap))
1761 #endif
1762     int rval = DDI_SUCCESS;
1763     PMpi2SasiouUnitPage1_t sasioupage1;
1764     int i, num_phys;
1765     uint32_t cpdi[MPTSAS_MAX_PHYS];
1766     uint8_t port_flags;
1767
1768     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
1769         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_io_unit_page1 "
1770                    "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
1771                    iocstatus, iocloginfo);
1772         rval = DDI_FAILURE;
1773         return (rval);
1774     }
1775
1776     sasioupage1 = (PMpi2SasiouUnitPage1_t)page_memp;
1777     num_phys = ddi_get8(accesssp, &sasioupage1->NumPhys);
1778     /*
1779      * ASSERT that the num_phys value in SAS IO Unit Page 1 is the same as
1780      * was initially set. This should never change throughout the life of
1781      * the driver.
1782      */
1783     ASSERT(num_phys == mpt->m_num_phys);
1784     for (i = 0; i < num_phys; i++) {
1785         cpdi[i] = ddi_get32(accesssp, &sasioupage1->PhyData[i].
1786             ControllerPhyDeviceInfo);
1787         port_flags = ddi_get8(accesssp,
1788             &sasioupage1->PhyData[i].PortFlags);
1789         mpt->m_phy_info[i].port_num =
1790             ddi_get8(accesssp,
1791                 &sasioupage1->PhyData[i].Port);
1792         mpt->m_phy_info[i].port_flags = port_flags;
1793         mpt->m_phy_info[i].phy_device_type = cpdi[i];
1794     }
1795
1796     return (rval);
1797 }
```

```

1798 /*
1799  * Read IO unit page 0 to get information for each PHY. If needed, Read IO Unit
1800  * page1 to update the PHY information. This is the message passing method of
1801  * this function which should be called except during initialization.
1802 */
1803 int
1804 mptsas_get_sas_io_unit_page(mptsas_t *mpt)
1805 {
1806     int rval = DDI_SUCCESS, state;
1807     uint32_t readpage1 = 0, retrypage0 = 0;
1808
1809     ASSERT(mutex_owned(&mpt->m_mutex));
1810
1811     /*
1812      * Now we cycle through the state machine. Here's what happens:
1813      * 1. Read IO unit page 0 and set phy information
1814      * 2. See if Read IO unit page1 is needed because of port configuration
1815      * 3. Read IO unit page 1 and update phy information.
1816      */
1817     state = IOUC_READ_PAGE0;
1818     while (state != IOUC_DONE) {
1819         if (state == IOUC_READ_PAGE0) {
1820             rval = mptsas_access_config_page(mpt,
1821                 MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
1822                 MPI2_CONFIG_EXTPAGETYPE_SAS_IO_UNIT, 0, 0,
1823                 mptsas_sasiou_page_0_cb, &readpage1,
1824                 &retrypage0);
1825         } else if (state == IOUC_READ_PAGE1) {
1826             rval = mptsas_access_config_page(mpt,
1827                 MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
1828                 MPI2_CONFIG_EXTPAGETYPE_SAS_IO_UNIT, 1, 0,
1829                 mptsas_sasiou_page_1_cb);
1830         }
1831
1832         if (rval == DDI_SUCCESS) {
1833             switch (state) {
1834                 case IOUC_READ_PAGE0:
1835                     /*
1836                      * retry 30 times if discovery is in process
1837                      */
1838                     if (retrypage0 && (retrypage0 < 30)) {
1839                         drv_usecwait(1000 * 100);
1840                         state = IOUC_READ_PAGE0;
1841                         break;
1842                     } else if (retrypage0 == 30) {
1843                         mptsas_log(mpt, CE_WARN,
1844                                     "!Discovery in progress, can't "
1845                                     "verify IO unit config, then "
1846                                     "after 30 times retry, give "
1847                                     "up!");
1848                     state = IOUC_DONE;
1849                     rval = DDI_FAILURE;
1850                     break;
1851             }
1852
1853             if (readpage1 == 0) {
1854                 state = IOUC_DONE;
1855                 rval = DDI_SUCCESS;
1856                 break;
1857             }
1858
1859             state = IOUC_READ_PAGE1;
1860             break;
1861
1862         case IOUC_READ_PAGE1:
1863             state = IOUC_DONE;
1864         }
1865     }
1866 }
```

```

1864             rval = DDI_SUCCESS;
1865         }
1866     } else {
1867         return (rval);
1868     }
1869 }
1870 }
1871
1872 return (rval);
1873 }

1875 static int
1876 mptsas_biospage_3_cb(mptsas_t *mpt, caddr_t page_memp,
1877 ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
1878 va_list ap)
1879 {
1880 #ifndef _lock_lint
1881     _NOTE(ARGUNUSED(ap))
1882 #endif
1883     pMpi2BiosPage3_t sasbiospage;
1884     int rval = DDI_SUCCESS;
1885     uint32_t *bios_version;
1886
1887     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
1888         (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
1889         mptsas_log(mpt, CE_WARN, "mptsas_get_bios_page3 header: "
1890                    "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus, iocloginfo);
1891         rval = DDI_FAILURE;
1892     }
1893     return (rval);
1894 }
1895 bios_version = va_arg(ap, uint32_t *);
1896 sasbiospage = (pMpi2BiosPage3_t)page_memp;
1897 *bios_version = ddi_get32(accesssp, &sasbiospage->BiosVersion);
1898
1899 }

1901 /*
1902  * Request MPI configuration page BIOS page 3 to get BIOS version. Since all
1903  * other information in this page is not needed, just ignore it.
1904 */
1905 int
1906 mptsas_get_bios_page3(mptsas_t *mpt, uint32_t *bios_version)
1907 {
1908     int rval = DDI_SUCCESS;
1909
1910     ASSERT(mutex_owned(&mpt->m_mutex));
1911
1912     /*
1913      * Get the header and config page. reply contains the reply frame,
1914      * which holds status info for the request.
1915     */
1916     rval = mptsas_access_config_page(mpt,
1917                                     MPI2_CONFIG_ACTION_PAGE_READ_CURRENT, MPI2_CONFIG_PAGETYPE_BIOS, 3,
1918                                     0, mptsas_biospage_3_cb, bios_version);
1919
1920     return (rval);
1921 }

1923 /*
1924  * Read IO unit page 0 to get information for each PHY. If needed, Read IO Unit
1925  * page1 to update the PHY information. This is the handshaking version of
1926  * this function, which should be called during initialization only.
1927 */
1928 int
1929 mptsas_get_sas_io_unit_page_hndshk(mptsas_t *mpt)

```

```

1930 {
1931     ddi_dma_attr_t recv_dma_attrs, page_dma_attrs;
1932     ddi_dma_cookie_t page_cookie;
1933     ddi_dma_handle_t recv_dma_handle, page_dma_handle;
1934     ddi_acc_handle_t recv_accesssp, page_accesssp;
1935     pMpi2ConfigReply_t configreply;
1936     pMpi2SASIOUnitPage0_t sasioupage0;
1937     pMpi2SASIOUnitPage1_t sasioupage1;
1938     int recv_numbytes;
1939     caddr_t recv_memp, page_memp;
1940     int i, num_phys, start_phys = 0;
1941     int page0_size =
1942         sizeof (MPI2_CONFIG_PAGE_SASIOUNIT_0) +
1943         (sizeof (MPI2_SAS_IO_UNIT0_PHY_DATA) * (MPTSAS_MAX_PHYS - 1));
1944     int page1_size =
1945         sizeof (MPI2_CONFIG_PAGE_SASIOUNIT_1) +
1946         (sizeof (MPI2_SAS_IO_UNIT1_PHY_DATA) * (MPTSAS_MAX_PHYS - 1));
1947     uint32_t flags_length;
1948     uint32_t cpdi[MPTSAS_MAX_PHYS];
1949     uint32_t readpage1 = 0, retrypage0 = 0;
1950     uint16_t iocstatus;
1951     uint8_t port_flags, page_number, action;
1952     uint32_t reply_size = 256; /* Big enough for any page */
1953     uint_t state;
1954     int rval = DDI_FAILURE;
1955     boolean_t free_recv = B_FALSE, free_page = B_FALSE;

1956     /*
1957      * Initialize our "state machine". This is a bit convoluted,
1958      * but it keeps us from having to do the ddi allocations numerous
1959      * times.
1960     */
1961
1962     NDBG20(("mptsas_get_sas_io_unit_page_hndshk enter"));
1963     ASSERT(mutex_owned(&mpt->m_mutex));
1964     state = IOUC_READ_PAGE0;

1965     /*
1966      * dynamically create a customized dma attribute structure
1967      * that describes mpt's config reply page request structure.
1968     */
1969     recv_dma_attrs = mpt->m_msg_dma_attr;
1970     recv_dma_attrs.dma_attr_sgllen = 1;
1971     recv_dma_attrs.dma_attr_granular = (sizeof (MPI2_CONFIG_REPLY));

1972     if (mptsas_dma_addr_create(mpt, recv_dma_attrs,
1973                                &recv_dma_handle, &recv_accesssp, &recv_memp,
1974                                (sizeof (MPI2_CONFIG_REPLY)), NULL) == FALSE) {
1975         mptsas_log(mpt, CE_WARN,
1976                     "mptsas_get_sas_io_unit_page_hndshk: recv dma failed");
1977         goto cleanup;
1978     }
1979     /* Now safe to call mptsas_dma_addr_destroy(recv_dma_handle). */
1980     free_recv = B_TRUE;

1981     page_dma_attrs = mpt->m_msg_dma_attr;
1982     page_dma_attrs.dma_attr_sgllen = 1;
1983     page_dma_attrs.dma_attr_granular = reply_size;

1984     if (mptsas_dma_addr_create(mpt, page_dma_attrs,
1985                                &page_dma_handle, &page_accesssp, &page_memp,
1986                                reply_size, &page_cookie) == FALSE) {
1987         mptsas_log(mpt, CE_WARN,
1988                     "mptsas_get_sas_io_unit_page_hndshk: page dma failed");
1989         goto cleanup;
1990     }

```

```
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsa5_impl.c          15

1996     /* Now safe to call mptsa5_dma_addr_destroy(page_dma_handle). */
1997     free_page = B_TRUE;
1998
1999     /*
2000      * Now we cycle through the state machine. Here's what happens:
2001      * 1. Read IO unit page 0 and set phy information
2002      * 2. See if Read IO unit page1 is needed because of port configuration
2003      * 3. Read IO unit page 1 and update phy information.
2004     */
2005
2006     sasioupage0 = (pMpi2SasIOUnitPage0_t)page_memp;
2007     sasioupage1 = (pMpi2SasIOUnitPage1_t)page_memp;
2008
2009     while (state != IOUC_DONE) {
2010         switch (state) {
2011             case IOUC_READ_PAGE0:
2012                 page_number = 0;
2013                 action = MPI2_CONFIG_ACTION_PAGE_READ_CURRENT;
2014                 flags_length = (uint32_t)page0_size;
2015                 flags_length |= ((uint32_t)(
2016                     MPI2_SGE_FLAGS_LAST_ELEMENT |
2017                     MPI2_SGE_FLAGS_END_OF_BUFFER |
2018                     MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
2019                     MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
2020                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
2021                     MPI2_SGE_FLAGS_IOC_TO_HOST |
2022                     MPI2_SGE_FLAGS_END_OF_LIST) <<
2023                     MPI2_SGE_FLAGS_SHIFT);
2024
2025                 break;
2026
2027             case IOUC_READ_PAGE1:
2028                 page_number = 1;
2029                 action = MPI2_CONFIG_ACTION_PAGE_READ_CURRENT;
2030                 flags_length = (uint32_t)page1_size;
2031                 flags_length |= ((uint32_t)(
2032                     MPI2_SGE_FLAGS_LAST_ELEMENT |
2033                     MPI2_SGE_FLAGS_END_OF_BUFFER |
2034                     MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
2035                     MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
2036                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
2037                     MPI2_SGE_FLAGS_IOC_TO_HOST |
2038                     MPI2_SGE_FLAGS_END_OF_LIST) <<
2039                     MPI2_SGE_FLAGS_SHIFT);
2040
2041                 break;
2042             default:
2043                 break;
2044         }
2045
2046         bzero(recv_memp, sizeof (MPI2_CONFIG_REPLY));
2047         configreply = (pMpi2ConfigReply_t)recv_memp;
2048         recv_numbytes = sizeof (MPI2_CONFIG_REPLY);
2049
2050         if (mptsa5_send_extended_config_request_msg(mpt,
2051             MPI2_CONFIG_ACTION_PAGE_HEADER,
2052             MPI2_CONFIG_EXTPAGETYPE_SAS_IO_UNIT,
2053             0, page_number, 0, 0, 0, 0)) {
2054             goto cleanup;
2055         }
2056
2057         if (mptsa5_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2058             recv_accesssp)) {
2059             goto cleanup;
2060         }

```

```
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
16

2062     iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus);
2063     iocstatus = MPTSAS_IOCSTATUS(iocstatus);
2064
2065     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
2066         mptsas_log(mpt, CE_WARN,
2067                     "mptsas_get_sas_io_unit_page_hndshk: read page "
2068                     "header iocstatus = 0x%#x", iocstatus);
2069         goto cleanup;
2070     }
2071
2072     if (action != MPI2_CONFIG_ACTION_PAGE_WRITE_NVRAM) {
2073         bzero(page_memp, reply_size);
2074     }
2075
2076     if (mptsas_send_extended_config_request_msg(mpt, action,
2077                                                 MPI2_CONFIG_EXTPAGETYPE_SAS_IO_UNIT, 0, page_number,
2078                                                 ddi_get8(recv_accesssp, &configreply->Header.PageVersion),
2079                                                 ddi_get16(recv_accesssp, &configreply->ExtPageLength),
2080                                                 flags_length, page_cookie.dmac_laddress)) {
2081         goto cleanup;
2082     }
2083
2084     if (mptsas_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2085                                   recv_accesssp)) {
2086         goto cleanup;
2087     }
2088
2089     iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus);
2090     iocstatus = MPTSAS_IOCSTATUS(iocstatus);
2091
2092     if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
2093         mptsas_log(mpt, CE_WARN,
2094                     "mptsas_get_sas_io_unit_page_hndshk: IO unit "
2095                     "config failed for action %d, iocstatus = 0x%#x",
2096                     action, iocstatus);
2097         goto cleanup;
2098     }
2099
2100     switch (state) {
2101     case IOUC_READ_PAGE0:
2102         if ((ddi_dma_sync(page_dma_handle, 0, 0,
2103                           DDI_DMA_SYNC_FORCPU)) != DDI_SUCCESS) {
2104             goto cleanup;
2105         }
2106
2107         num_phys = ddi_get8(page_accesssp,
2108                             &sasioupage0->NumPhys);
2109         ASSERT(num_phys == mpt->m_num_phys);
2110         if (num_phys > MPTSAS_MAX_PHYS) {
2111             mptsas_log(mpt, CE_WARN, "Number of phys "
2112                         "supported by HBA (%d) is more than max "
2113                         "supported by driver (%d). Driver will "
2114                         "not attach.", num_phys,
2115                         MPTSAS_MAX_PHYS);
2116         rval = DDI_FAILURE;
2117         goto cleanup;
2118     }
2119     for (i = start_phy; i < num_phys; i++, start_phy = i) {
2120         cpdi[i] = ddi_get32(page_accesssp,
2121                             &sasioupage0->PhyData[i].ControllerPhyDeviceInfo);
2122         port_flags = ddi_get8(page_accesssp,
2123                               &sasioupage0->PhyData[i].PortFlags);
2124
2125         mpt->m_phy_info[i].port_num =
2126             ddi_get8(page_accesssp,
```

`new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.`

17

```

2128     &sasioupage0->PhyData[i].Port);
2129     mpt->m_phy_info[i].ctrl_devhdl =
2130         ddi_get16(page_accesssp, &sasioupage0->
2131             PhyData[i].ControllerDevHandle);
2132     mpt->m_phy_info[i].attached_devhdl =
2133         ddi_get16(page_accesssp, &sasioupage0->
2134             PhyData[i].AttachedDevHandle);
2135     mpt->m_phy_info[i].phy_device_type = cpdi[i];
2136     mpt->m_phy_info[i].port_flags = port_flags;

2138     if (port_flags & DISCOVERY_IN_PROGRESS) {
2139         retrypage0++;
2140         NDBG20(("Discovery in progress, can't
2141                 "verify IO unit config, then NO.%d
2142                 " times retry", retrypage0));
2143         break;
2144     } else {
2145         retrypage0 = 0;
2146     }
2147     if (!(port_flags & AUTO_PORT_CONFIGURATION)) {
2148         /*
2149             * some PHY configuration described in
2150             * SAS IO Unit Page1
2151             */
2152         readpage1 = 1;
2153     }
2154 }

2156 /*
2157     * retry 30 times if discovery is in process
2158 */
2159 if (retrypage0 && (retrypage0 < 30)) {
2160     drv_usecwait(1000 * 100);
2161     state = IOUC_READ_PAGE0;
2162     break;
2163 } else if (retrypage0 == 30) {
2164     mptsas_log(mpt, CE_WARN,
2165                 "Discovery in progress, can't "
2166                 "verify IO unit config, then after"
2167                 " 30 times retry, give up!");
2168     state = IOUC_DONE;
2169     rval = DDI_FAILURE;
2170     break;
2171 }

2173 if (readpage1 == 0) {
2174     state = IOUC_DONE;
2175     rval = DDI_SUCCESS;
2176     break;
2177 }

2179 state = IOUC_READ_PAGE1;
2180 break;

2182 case IOUC_READ_PAGE1:
2183     if ((ddi_dma_sync(page_dma_handle, 0, 0,
2184                         DDI_DMA_SYNC_FORCPU)) != DDI_SUCCESS) {
2185         goto cleanup;
2186     }

2188 num_phys = ddi_get8(page_accesssp,
2189                     &sasioupage1->NumPhys);
2190 ASSERT(num_phys == mpt->m_num_phys);
2191 if (num_phys > MPTAS_MAX_PHYS) {
2192     mptsas_log(mpt, CE_WARN, "Number of phys "
2193                 "supported by HBA (%d) is more than max "

```

new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c

18

```

194         "supported by driver (%d).  Driver will "
195         "not attach.", num_phys,
196         MPTSAS_MAX_PHYS);
197     rval = DDI_FAILURE;
198     goto cleanup;
199 }
200 for (i = 0; i < num_phys; i++) {
201     cpdi[i] = ddi_get32(page_accesssp,
202         &sasioupage1->PhyData[i].ControllerPhyDeviceInfo);
203     port_flags = ddi_get8(page_accesssp,
204         &sasioupage1->PhyData[i].PortFlags);
205     mpt->m_phys_info[i].port_num =
206         ddi_get8(page_accesssp,
207             &sasioupage1->PhyData[i].Port);
208     mpt->m_phys_info[i].port_flags = port_flags;
209     mpt->m_phys_info[i].phy_device_type = cpdi[i];
210 }
211 state = IOUC_DONE;
212 rval = DDI_SUCCESS;
213 break;
214 }
215 if ((mptsas_check_dma_handle(recv_dma_handle) != DDI_SUCCESS) ||
216     (mptsas_check_dma_handle(page_dma_handle) != DDI_SUCCESS)) {
217     ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
218     rval = DDI_FAILURE;
219     goto cleanup;
220 }
221 if ((mptsas_check_acc_handle(recv_accesssp) != DDI_SUCCESS) ||
222     (mptsas_check_acc_handle(page_accesssp) != DDI_SUCCESS)) {
223     ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
224     rval = DDI_FAILURE;
225     goto cleanup;
226 }
227 cleanup:
228     if (free_recv)
229         mptsas_dma_addr_destroy(&recv_dma_handle, &recv_accesssp);
230     if (free_page)
231         mptsas_dma_addr_destroy(&page_dma_handle, &page_accesssp);
232     if (rval != DDI_SUCCESS) {
233         mptsas_fm_xreport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
234         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
235     }
236     return (rval);
237 }

238 /*
239 * mptsas_get_manufacture_page5
240 *
241 * This function will retrieve the base WWID from the adapter.  Since this
242 * function is only called during the initialization process, use handshaking.
243 */
244 int
245 mptsas_get_manufacture_page5(mptsas_t *mpt)
246 {
247     ddi_dma_attr_t          recv_dmaAttrs, page_dmaAttrs;
248     ddi_dma_cookie_t         pageCookie;
249     ddi_dma_handle_t         recv_dmaHandle, page_dmaHandle;
250     ddi_acc_handle_t         recvAccesssp, page_accesssp;
251     pMpI2ConfigReply_t      configReply;
252     caddr_t                  recvMemp, page_memp;
253     int                      recvNumbytes;

```

```

2260     pMpi2ManufacturingPage5_t      m5;
2261     uint32_t                      flagslength;
2262     int                           rval = DDI_SUCCESS;
2263     uint_t                        iocstatus;
2264     boolean_t                     free_recv = B_FALSE, free_page = B_FALSE;
2265
2266     MPTSAS_DISABLE_INTR(mpt);
2267
2268     if (mptsas_send_config_request_msg(mpt, MPI2_CONFIG_ACTION_PAGE_HEADER,
2269                                         MPI2_CONFIG_PAGETYPE_MANUFACTURING, 0, 5, 0, 0, 0, 0)) {
2270         rval = DDI_FAILURE;
2271         goto done;
2272     }
2273
2274     /*
2275      * dynamically create a customized dma attribute structure
2276      * that describes the MPT's config reply page request structure.
2277      */
2278     recv_dma_attrs = mpt->m_msg_dma_attr;
2279     recv_dma_attrs.dma_attr_sglen = 1;
2280     recv_dma_attrs.dma_attr_granular = (sizeof (MPI2_CONFIG_REPLY));
2281
2282     if (mptsas_dma_addr_create(mpt, recv_dma_attrs,
2283                               &recv_dma_handle, &recv_accesssp, &recv_memp,
2284                               (sizeof (MPI2_CONFIG_REPLY)), NULL) == FALSE) {
2285         rval = DDI_FAILURE;
2286         goto done;
2287     }
2288     /* Now safe to call mptsas_dma_addr_destroy(recv_dma_handle). */
2289     free_recv = B_TRUE;
2290
2291     bzero(recv_memp, sizeof (MPI2_CONFIG_REPLY));
2292     configreply = (pMpi2ConfigReply_t)recv_memp;
2293     recv_numbytes = sizeof (MPI2_CONFIG_REPLY);
2294
2295     /*
2296      * get config reply message
2297      */
2298     if (mptsas_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2299                                 recv_accesssp)) {
2300         rval = DDI_FAILURE;
2301         goto done;
2302     }
2303
2304     if (iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus)) {
2305         mptsas_log(mpt, CE_WARN, "mptsas_get_manufacture_page5 update: "
2306                    "IOCStatus=0x%lx, IOCLogInfo=0x%lx", iocstatus,
2307                    ddi_get32(recv_accesssp, &configreply->IOCLogInfo));
2308         goto done;
2309     }
2310
2311     /*
2312      * dynamically create a customized dma attribute structure
2313      * that describes the MPT's config page structure.
2314      */
2315     page_dma_attrs = mpt->m_msg_dma_attr;
2316     page_dma_attrs.dma_attr_sglen = 1;
2317     page_dma_attrs.dma_attr_granular = (sizeof (MPI2_CONFIG_PAGE_MAN_5));
2318
2319     if (mptsas_dma_addr_create(mpt, page_dma_attrs, &page_dma_handle,
2320                               &page_accesssp, &page_memp, (sizeof (MPI2_CONFIG_PAGE_MAN_5)),
2321                               &page_cookie) == FALSE) {
2322         rval = DDI_FAILURE;
2323         goto done;
2324     }
2325     /* Now safe to call mptsas_dma_addr_destroy(page_dma_handle). */

```

```

2326     free_page = B_TRUE;
2327
2328     bzero(page_memp, sizeof (MPI2_CONFIG_PAGE_MAN_5));
2329     m5 = (pMpi2ManufacturingPage5_t)page_memp;
2330     NDBG20(("mptsas_get_manufacture_page5: paddr 0x%p",
2331             (void *)(uintptr_t)page_cookie.dmac_laddress));
2332
2333     /*
2334      * Give reply address to IOC to store config page in and send
2335      * config request out.
2336      */
2337
2338     flagslength = sizeof (MPI2_CONFIG_PAGE_MAN_5);
2339     flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
2340                               MPI2_SGE_FLAGS_END_OF_BUFFER | MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
2341                               MPI2_SGE_FLAGS_SYSTEM_ADDRESS | MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
2342                               MPI2_SGE_FLAGS_IOC_TO_HOST |
2343                               MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
2344
2345     if (mptsas_send_config_request_msg(mpt,
2346                                         MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
2347                                         MPI2_CONFIG_PAGETYPE_MANUFACTURING, 0, 5,
2348                                         ddi_get8(recv_accesssp, &configreply->Header.PageVersion),
2349                                         ddi_get8(recv_accesssp, &configreply->Header.PageLength),
2350                                         flagslength, page_cookie.dmac_laddress)) {
2351         rval = DDI_FAILURE;
2352         goto done;
2353     }
2354
2355     /*
2356      * get reply view handshake
2357      */
2358     if (mptsas_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2359                                 recv_accesssp)) {
2360         rval = DDI_FAILURE;
2361         goto done;
2362     }
2363
2364     if (iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus)) {
2365         mptsas_log(mpt, CE_WARN, "mptsas_get_manufacture_page5 config: "
2366                    "IOCStatus=0x%lx, IOCLogInfo=0x%lx", iocstatus,
2367                    ddi_get32(recv_accesssp, &configreply->IOCLogInfo));
2368         goto done;
2369     }
2370
2371     (void) ddi_dma_sync(page_dma_handle, 0, 0, DDI_DMA_SYNC_FORCPU);
2372
2373     /*
2374      * Fusion-MPT stores fields in little-endian format. This is
2375      * why the low-order 32 bits are stored first.
2376      */
2377     mpt->un.sasaddr.m_base_wwid_lo =
2378         ddi_get32(page_accesssp, (uint32_t *)(&m5->Phy[0].WWID));
2379     mpt->un.sasaddr.m_base_wwid_hi =
2380         ddi_get32(page_accesssp, (uint32_t *)(&m5->Phy[0].WWID + 1));
2381
2382     if (ddi_prop_update_int64(DDI_DEV_T_NONE, mpt->m_dip,
2383                               "base-wwid", mpt->un.m_base_wwid) != DDI_PROP_SUCCESS) {
2384         NDBG2((("%s%d: failed to create base-wwid property",
2385                 ddi_driver_name(mpt->m_dip), ddi_get_instance(mpt->m_dip)));
2386     }
2387
2388     /*
2389      * Set the number of PHYS present.
2390      */
2391     mpt->m_num_phys = ddi_get8(page_accesssp, (uint8_t *)(&m5->NumPhys));

```

```

2393     if (ddi_prop_update_int(DDI_DEV_T_NONE, mpt->m_dip,
2394         "num-phys", mpt->m_num_phys) != DDI_PROP_SUCCESS) {
2395         NDBG2(("'%s%d: failed to create num-phys property",
2396             ddi_driver_name(mpt->m_dip), ddi_get_instance(mpt->m_dip)));
2397     }
2398
2399     mptsas_log(mpt, CE_NOTE, "!mpt%d: Initiator WWNs: 0x%016llx-0x%016llx",
2400         mpt->m_instance, (unsigned long long)mpt->un.m_base_wwid,
2401         (unsigned long long)mpt->un.m_base_wwid + mpt->m_num_phys - 1);
2402
2403     if ((mptsas_check_dma_handle(recv_dma_handle) != DDI_SUCCESS) ||
2404         (mptsas_check_dma_handle(page_dma_handle) != DDI_SUCCESS)) {
2405         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2406         rval = DDI_FAILURE;
2407         goto done;
2408     }
2409     if ((mptsas_check_acc_handle(recv_accesssp) != DDI_SUCCESS) ||
2410         (mptsas_check_acc_handle(page_accesssp) != DDI_SUCCESS)) {
2411         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2412         rval = DDI_FAILURE;
2413     }
2414 done:
2415     /*
2416      * free up memory
2417      */
2418     if (free_recv)
2419         mptsas_dma_addr_destroy(&recv_dma_handle, &recv_accesssp);
2420     if (free_page)
2421         mptsas_dma_addr_destroy(&page_dma_handle, &page_accesssp);
2422     MPTAS_ENABLE_INTR(mpt);
2423
2424     return (rval);
2425 }
2426
2427 static int
2428 mptsas_sasphypage_0_cb(mptsas_t *mpt, caddr_t page_memp,
2429     ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
2430     va_list ap)
2431 {
2432 #ifndef __lock_lint
2433     _NOTE(ARGUNUSED(ap))
2434 #endif
2435     pMpi2SasPhyPage0_t sasphypage;
2436     int rval = DDI_SUCCESS;
2437     uint16_t *owner_devhdl, *attached_devhdl;
2438     uint8_t *attached_phy_identify;
2439     uint32_t *attached_phy_info;
2440     uint8_t *programmed_link_rate;
2441     uint8_t *hw_link_rate;
2442     uint8_t *change_count;
2443     uint32_t *phy_info;
2444     uint8_t *negotiated_link_rate;
2445     uint32_t page_address;
2446
2447     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
2448         (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
2449         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_expander_page0 "
2450             "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
2451             iocstatus, iocloginfo);
2452         rval = DDI_FAILURE;
2453         return (rval);
2454     }
2455     page_address = va_arg(ap, uint32_t);
2456     /*
2457      * The INVALID_PAGE status is normal if using GET_NEXT_HANDLE and there

```

```

2458     * are no more pages. If everything is OK up to this point but the
2459     * status is INVALID_PAGE, change rval to FAILURE and quit. Also,
2460     * signal that device traversal is complete.
2461     */
2462     if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
2463         if ((page_address & MPI2_SAS_EXPAND_PGAD_FORM_MASK) ==
2464             MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL) {
2465             mpt->m_done_traverse_smp = 1;
2466         }
2467         rval = DDI_FAILURE;
2468         return (rval);
2469     }
2470     owner_devhdl = va_arg(ap, uint16_t *);
2471     attached_devhdl = va_arg(ap, uint16_t *);
2472     attached_phy_identify = va_arg(ap, uint8_t *);
2473     attached_phy_info = va_arg(ap, uint32_t *);
2474     programmed_link_rate = va_arg(ap, uint8_t *);
2475     hw_link_rate = va_arg(ap, uint8_t *);
2476     change_count = va_arg(ap, uint8_t *);
2477     phy_info = va_arg(ap, uint32_t *);
2478     negotiated_link_rate = va_arg(ap, uint8_t *);

2479     sasphypage = (pMpi2SasPhyPage0_t)page_memp;

2480     *owner_devhdl =
2481         ddi_get16(accesssp, &sasphypage->OwnerDevHandle);
2482     *attached_devhdl =
2483         ddi_get16(accesssp, &sasphypage->AttachedDevHandle);
2484     *attached_phy_identify =
2485         ddi_get8(accesssp, &sasphypage->AttachedPhyIdentifier);
2486     *attached_phy_info =
2487         ddi_get32(accesssp, &sasphypage->AttachedPhyInfo);
2488     *programmed_link_rate =
2489         ddi_get8(accesssp, &sasphypage->ProgrammedLinkRate);
2490     *hw_link_rate =
2491         ddi_get8(accesssp, &sasphypage->HwLinkRate);
2492     *change_count =
2493         ddi_get8(accesssp, &sasphypage->ChangeCount);
2494     *phy_info =
2495         ddi_get32(accesssp, &sasphypage->PhyInfo);
2496     *negotiated_link_rate =
2497         ddi_get8(accesssp, &sasphypage->NegotiatedLinkRate);

2498     2501     return (rval);
2499
2500 /* Request MPI configuration page SAS phy page 0 to get DevHandle, phymask
2501 * and SAS address.
2502 */
2503 int
2504 mptsas_get_sas_phy_page0(mptsas_t *mpt, uint32_t page_address,
2505     smhba_info_t *info)
2506 {
2507     int rval = DDI_SUCCESS;
2508
2509     ASSERT(mutex_owned(&mpt->m_mutex));
2510
2511     /*
2512      * Get the header and config page. reply contains the reply frame,
2513      * which holds status info for the request.
2514      */
2515     rval = mptsas_access_config_page(mpt,
2516         MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
2517         MPI2_CONFIG_EXTPAGETYPE_SAS_PHY, 0, page_address,
2518         mptsas_sasphypage_0_cb, page_address, &info->owner_devhdl,
2519

```

```

2524         &info->attached_devhdl, &info->attached_phy_identify,
2525         &info->attached_phy_info, &info->programmed_link_rate,
2526         &info->hw_link_rate, &info->change_count,
2527         &info->phy_info, &info->negotiated_link_rate);
2528
2529     return (rval);
2530 }
2531
2532 static int
2533 mptsas_sasphypage_1_cb(mptsas_t *mpt, caddr_t page_memp,
2534                         ddi_acc_handle_t accesssp, uint16_t iocstatus, uint32_t iocloginfo,
2535                         va_list ap)
2536 {
2537 #ifndef __lock_lint
2538     _NOTE(ARGUNUSED(ap))
2539 #endif
2540     pMpi2SasPhyPage1_t    sasphypage;
2541     int                   rval = DDI_SUCCESS;
2542
2543     uint32_t               *invalid_dword_count;
2544     uint32_t               *running_disparity_error_count;
2545     uint32_t               *loss_of_dword_sync_count;
2546     uint32_t               *phy_reset_problem_count;
2547     uint32_t               page_address;
2548
2549     if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
2550         (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
2551         mptsas_log(mpt, CE_WARN, "mptsas_get_sas_expander_page1 "
2552                    "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
2553                    iocstatus, iocloginfo);
2554         rval = DDI_FAILURE;
2555         return (rval);
2556     }
2557     page_address = va_arg(ap, uint32_t);
2558 /*
2559  * The INVALID_PAGE status is normal if using GET_NEXT_HANDLE and there
2560  * are no more pages. If everything is OK up to this point but the
2561  * status is INVALID_PAGE, change rval to FAILURE and quit. Also,
2562  * signal that device traversal is complete.
2563 */
2564     if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
2565         if ((page_address & MPI2_SAS_EXPAND_PGAD_FORM_MASK) ==
2566             MPI2_SAS_EXPAND_PGAD_FORM_GET_NEXT_HNDL) {
2567             mpt->m_done_traverse_smp = 1;
2568         }
2569         rval = DDI_FAILURE;
2570         return (rval);
2571     }
2572
2573     invalid_dword_count = va_arg(ap, uint32_t *);
2574     running_disparity_error_count = va_arg(ap, uint32_t *);
2575     loss_of_dword_sync_count = va_arg(ap, uint32_t *);
2576     phy_reset_problem_count = va_arg(ap, uint32_t *);
2577
2578     sasphypage = (pMpi2SasPhyPage1_t)page_memp;
2579
2580     *invalid_dword_count =
2581         ddi_get32(accesssp, &sasphypage->InvalidDwordCount);
2582     *running_disparity_error_count =
2583         ddi_get32(accesssp, &sasphypage->RunningDisparityErrorCount);
2584     *loss_of_dword_sync_count =
2585         ddi_get32(accesssp, &sasphypage->LossDwordSyncCount);
2586     *phy_reset_problem_count =
2587         ddi_get32(accesssp, &sasphypage->PhyResetProblemCount);
2588
2589     return (rval);

```

```

2590 }
2591 /*
2592  * Request MPI configuration page SAS phy page 0 to get DevHandle, phymask
2593  * and SAS address.
2594 */
2595 int
2596 mptsas_get_sas_phy_page1(mptsas_t *mpt, uint32_t page_address,
2597                           smhba_info_t *info)
2598 {
2599     int                                rval = DDI_SUCCESS;
2600
2601     ASSERT(mutex_owned(&mpt->m_mutex));
2602
2603 /*
2604  * Get the header and config page. reply contains the reply frame,
2605  * which holds status info for the request.
2606 */
2607     rval = mptsas_access_config_page(mpt,
2608                                     MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
2609                                     MPI2_CONFIG_EXTPAGETYPE_SAS_PHY, 1, page_address,
2610                                     mptsas_sasphypage_1_cb, page_address,
2611                                     &info->invalid_dword_count,
2612                                     &info->running_disparity_error_count,
2613                                     &info->loss_of_dword_sync_count,
2614                                     &info->phy_reset_problem_count);
2615
2616     return (rval);
2617 }
2618 /*
2619  * mptsas_get_manufacture_page0
2620 */
2621 /*
2622  * This function will retrieve the base
2623  * Chip name, Board Name,Board Trace number from the adapter.
2624  * Since this function is only called during the
2625  * initialization process, use handshaking.
2626 */
2627 int
2628 mptsas_get_manufacture_page0(mptsas_t *mpt)
2629 {
2630     ddi_dma_attr_t          recv_dmaAttrs, page_dmaAttrs;
2631     ddi_dma_cookie_t        page_cookie;
2632     ddi_dma_handle_t        recv_dmaHandle, page_dmaHandle;
2633     ddi_acc_handle_t        recv_accesssp, page_accesssp;
2634     pMpi2ConfigReply_t      configReply;
2635     caddr_t                 recv_memp, page_memp;
2636     int                      recv_numbytes;
2637     pMpi2ManufacturingPage0_t m0;
2638     uint32_t                flagsLength;
2639     int                      rval = DDI_SUCCESS;
2640     uint_t                  iocstatus;
2641     uint8_t                 i = 0;
2642     boolean_t                free_recv = B_FALSE, free_page = B_FALSE;
2643
2644     MPTSAES_DISABLE_INTR(mpt);
2645
2646     if (mptsas_send_config_request_msg(mpt, MPI2_CONFIG_ACTION_PAGE_HEADER,
2647                                         MPI2_CONFIG_PAGETYPE_MANUFACTURING, 0, 0, 0, 0, 0, 0)) {
2648         rval = DDI_FAILURE;
2649         goto done;
2650     }
2651
2652 /*
2653  * dynamically create a customized dma attribute structure
2654  * that describes the MPT's config reply page request structure.
2655 */

```

```

2656     recv_dma_attrs = mpt->m_msg_dma_attr;
2657     recv_dma_attrs.dma_attr_sgllen = 1;
2658     recv_dma_attrs.dma_attr_granular = (sizeof (MPI2_CONFIG_REPLY));
2659
2660     if (mptsas_dma_addr_create(mpt, recv_dma_attrs, &recv_dma_handle,
2661         &recv_accesssp, &recv_memp, (sizeof (MPI2_CONFIG_REPLY)),
2662         NULL) == FALSE) {
2663         rval = DDI_FAILURE;
2664         goto done;
2665     }
2666     /* Now safe to call mptsas_dma_addr_destroy(recv_dma_handle). */
2667     free_recv = B_TRUE;
2668
2669     bzero(recv_memp, sizeof (MPI2_CONFIG_REPLY));
2670     configreply = (pMpI2ConfigReply_t)recv_memp;
2671     recv_numbytes = sizeof (MPI2_CONFIG_REPLY);
2672
2673     /*
2674      * get config reply message
2675      */
2676     if (mptsas_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2677         recv_accesssp)) {
2678         rval = DDI_FAILURE;
2679         goto done;
2680     }
2681
2682     if (iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus)) {
2683         mptsas_log(mpt, CE_WARN, "mptsas_get_manufacture_page5 update: "
2684             "IOCStatus=0x%lx, IOCLogInfo=0x%lx", iocstatus,
2685             ddi_get32(recv_accesssp, &configreply->IOCLogInfo));
2686         goto done;
2687     }
2688
2689     /*
2690      * dynamically create a customized dma attribute structure
2691      * that describes the MPT's config page structure.
2692      */
2693     page_dma_attrs = mpt->m_msg_dma_attr;
2694     page_dma_attrs.dma_attr_sgllen = 1;
2695     page_dma_attrs.dma_attr_granular = (sizeof (MPI2_CONFIG_PAGE_MAN_0));
2696
2697     if (mptsas_dma_addr_create(mpt, page_dma_attrs, &page_dma_handle,
2698         &page_accesssp, &page_memp, (sizeof (MPI2_CONFIG_PAGE_MAN_0)),
2699         &page_cookie) == FALSE) {
2700         rval = DDI_FAILURE;
2701         goto done;
2702     }
2703     /* Now safe to call mptsas_dma_addr_destroy(page_dma_handle). */
2704     free_page = B_TRUE;
2705
2706     bzero(page_memp, sizeof (MPI2_CONFIG_PAGE_MAN_0));
2707     m0 = (pMpI2ManufacturingPage0_t)page_memp;
2708
2709     /*
2710      * Give reply address to IOC to store config page in and send
2711      * config request out.
2712      */
2713
2714     flagslength = sizeof (MPI2_CONFIG_PAGE_MAN_0);
2715     flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_END_OF_BUFFER | MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
2716         MPI2_SGE_FLAGS_SYSTEM_ADDRESS | MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
2717         MPI2_SGE_FLAGS_IOC_TO_HOST |
2718         MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
2719
2720     if (mptsas_send_config_request_msg(mpt,

```

```

2722     MPI2_CONFIG_ACTION_PAGE_READ_CURRENT,
2723     MPI2_CONFIG_PAGETYPE_MANUFACTURING, 0, 0,
2724     ddi_get8(recv_accesssp, &configreply->Header.PageVersion),
2725     ddi_get8(recv_accesssp, &configreply->Header.PageLength),
2726     flagslength, page_cookie.dmac_laddress)) {
2727         rval = DDI_FAILURE;
2728         goto done;
2729     }
2730
2731     /*
2732      * get reply view handshake
2733      */
2734     if (mptsas_get_handshake_msg(mpt, recv_memp, recv_numbytes,
2735         recv_accesssp)) {
2736         rval = DDI_FAILURE;
2737         goto done;
2738     }
2739
2740     if (iocstatus = ddi_get16(recv_accesssp, &configreply->IOCStatus)) {
2741         mptsas_log(mpt, CE_WARN, "mptsas_get_manufacture_page0 config: "
2742             "IOCStatus=0x%lx, IOCLogInfo=0x%lx", iocstatus,
2743             ddi_get32(recv_accesssp, &configreply->IOCLogInfo));
2744         goto done;
2745     }
2746
2747     (void) ddi_dma_sync(page_dma_handle, 0, 0, DDI_DMA_SYNC_FORCPU);
2748
2749     /*
2750      * Fusion-MPT stores fields in little-endian format. This is
2751      * why the low-order 32 bits are stored first.
2752      */
2753
2754     for (i = 0; i < 16; i++) {
2755         mpt->m_MANU_page0.ChipName[i] =
2756             ddi_get8(page_accesssp,
2757             (uint8_t *) (void *) &m0->ChipName[i]);
2758     }
2759
2760     for (i = 0; i < 8; i++) {
2761         mpt->m_MANU_page0.ChipRevision[i] =
2762             ddi_get8(page_accesssp,
2763             (uint8_t *) (void *) &m0->ChipRevision[i]);
2764     }
2765
2766     for (i = 0; i < 16; i++) {
2767         mpt->m_MANU_page0.BoardName[i] =
2768             ddi_get8(page_accesssp,
2769             (uint8_t *) (void *) &m0->BoardName[i]);
2770     }
2771
2772     for (i = 0; i < 16; i++) {
2773         mpt->m_MANU_page0.BoardAssembly[i] =
2774             ddi_get8(page_accesssp,
2775             (uint8_t *) (void *) &m0->BoardAssembly[i]);
2776     }
2777
2778     for (i = 0; i < 16; i++) {
2779         mpt->m_MANU_page0.BoardTracerNumber[i] =
2780             ddi_get8(page_accesssp,
2781             (uint8_t *) (void *) &m0->BoardTracerNumber[i]);
2782     }
2783
2784     if ((mptsas_check_dma_handle(recv_dma_handle) != DDI_SUCCESS) ||
2785         (mptsas_check_dma_handle(page_dma_handle) != DDI_SUCCESS)) {
2786         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2787         rval = DDI_FAILURE;
2788     }

```

```
2788         goto done;
2789     }
2790     if ((mptsas_check_acc_handle(recv_accesssp) != DDI_SUCCESS) ||
2791         (mptsas_check_acc_handle(page_accesssp) != DDI_SUCCESS)) {
2792         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
2793         rval = DDI_FAILURE;
2794     }
2795 done:
2796 /*
2797  * free up memory
2798  */
2799 if (free_recv)
2800     mptsas_dma_addr_destroy(&recv_dma_handle, &recv_accesssp);
2801 if (free_page)
2802     mptsas_dma_addr_destroy(&page_dma_handle, &page_accesssp);
2803 MPTSAS_ENABLE_INTR(mpt);
2804
2805 return (rval);
2806 }
```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      1
*****
45248 Fri Dec 19 13:49:25 2014
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
First pass at 4310
*****
_____ unchanged_portion_omitted_


679 typedef struct mptsas {
680     int             m_instance;
682     struct mptsas *m_next;
684     scsi_hba_tran_t    *m_tran;
685     smp_hba_tran_t   *m_smpttran;
686     kmutex_t          m_mutex;
687     kmutex_t          m_passthru_mutex;
688     kcondvar_t        m_cv;
689     kcondvar_t        m_passthru_cv;
690     kcondvar_t        m_fw_cv;
691     kcondvar_t        m_config_cv;
692     kcondvar_t        m_fw_diag_cv;
693     dev_info_t        *m_dip;
695     /*
696      * soft state flags
697      */
698     uint_t            m_softstate;
700     refhash_t         *m_targets;
701     refhash_t         *m_smp_targets;
703     m_raidconfig_t   m_raidconfig[MPTSAS_MAX_RAIDCONFIGS];
704     uint8_t           m_num_raid_configs;
706     struct mptsas_slots *m_active; /* outstanding cmdbs */
708     mptsas_cmd_t     *m_waitq;    /* cmd queue for active request */
709     mptsas_cmd_t     **m_waitqtail; /* wait queue tail ptr */
711     kmutex_t          m_tx_waitq_mutex;
712     mptsas_cmd_t     *m_tx_waitq; /* TX cmd queue for active request */
713     mptsas_cmd_t     **m_tx_waitqtail; /* tx_wait queue tail ptr */
714     int               m_tx_draining; /* TX queue draining flag */
716     mptsas_cmd_t     *m_doneq;    /* queue of completed commands */
717     mptsas_cmd_t     **m_donetail; /* queue tail ptr */
719     /*
720      * variables for helper threads (fan-out interrupts)
721      */
722     mptsas_doneq_thread_list_t   *m_doneq_thread_id;
723     uint32_t            m_doneq_thread_n;
724     uint32_t            m_doneq_thread_threshold;
725     uint32_t            m_doneq_length_threshold;
726     uint32_t            m_doneq_len;
727     kcondvar_t        m_doneq_cv;
728     kmutex_t          m_doneq_mutex;

730     int               m_ncmds;    /* number of outstanding commands */
731     m_event_struct_t  *m_ioc_event_cmdq; /* cmd queue for ioc event */
732     m_event_struct_t  **m_ioc_event_cmdtail; /* ioc cmd queue tail */
734     ddi_acc_handle_t  m_datap;    /* operating regs data access handle */
736     struct _MPI2_SYSTEM_INTERFACE_REGS   *m_reg;

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      2
*****
738     ushort_t          m_devid;        /* device id of chip. */
739     uchar_t            m_revid;        /* revision of chip. */
740     uint16_t           m_svid;         /* subsystem Vendor ID of chip */
741     uint16_t           m_ssid;         /* subsystem Device ID of chip */
743     uchar_t            m_sync_offset; /* default offset for this chip. */
745     timeout_id_t       m_quiesce_timeid;
747     ddi_dma_handle_t   m_dma_req_frame_hdl;
748     ddi_acc_handle_t   m_acc_req_frame_hdl;
749     ddi_dma_handle_t   m_dma_req_sense_hdl;
750     ddi_acc_handle_t   m_acc_req_sense_hdl;
751     ddi_dma_handle_t   m_dma_reply_frame_hdl;
752     ddi_acc_handle_t   m_acc_reply_frame_hdl;
753     ddi_dma_handle_t   m_dma_free_queue_hdl;
754     ddi_acc_handle_t   m_acc_free_queue_hdl;
755     ddi_dma_handle_t   m_dma_post_queue_hdl;
756     ddi_acc_handle_t   m_acc_post_queue_hdl;
758     /*
759      * list of reset notification requests
760      */
761     struct scsi_reset_notify_entry *m_reset_notify_listf;
763     /*
764      * qfull handling
765      */
766     timeout_id_t       m_restart_cmd_timeid;
768     /*
769      * scsi reset delay per bus
770      */
771     uint_t              m_scsi_reset_delay;
773     int                m_pm_idle_delay;
775     uchar_t            m_polled_intr; /* intr was polled. */
776     uchar_t            m_suspended;   /* true if driver is suspended */
778     struct kmem_cache *m_kmem_cache;
779     struct kmem_cache *m_cache_frames;
781     /*
782      * hba options.
783      */
784     uint_t              m_options;
786     int                m_in_callback;
788     int                m_power_level; /* current power level */
790     int                m_busy;        /* power management busy state */
792     off_t              m_pmcsr_offset; /* PMCSR offset */
794     ddi_acc_handle_t   m_config_handle;
796     ddi_dma_attr_t     m_io_dma_attr; /* Used for data I/O */
797     ddi_dma_attr_t     m_msg_dma_attr; /* Used for message frames */
798     ddi_device_acc_attr_t m_dev_acc_attr;
799     ddi_device_acc_attr_t m_reg_acc_attr;
801     /*
802      * request/reply variables
803      */

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      3
804     caddr_t          m_req_frame;
805     uint64_t         m_req_frame_dma_addr;
806     caddr_t          m_req_sense;
807     caddr_t          m_extreq_sense;
808     uint64_t         m_req_sense_dma_addr;
809     caddr_t          m_reply_frame;
810     uint64_t         m_reply_frame_dma_addr;
811     caddr_t          m_free_queue;
812     uint64_t         m_free_queue_dma_addr;
813     caddr_t          m_post_queue;
814     uint64_t         m_post_queue_dma_addr;
815     struct map        *m_ergsense_map;

817     m_replyh_arg_t *m_replyh_args;

819     uint16_t          m_max_requests;
820     uint16_t          m_req_frame_size;
821     uint16_t          m_req_sense_size;

823     /*
824     * Max frames per request reported in IOC Facts
825     */
826     uint8_t           m_max_chain_depth;
827     /*
828     * Max frames per request which is used in reality. It's adjusted
829     * according DMA SG length attribute, and shall not exceed the
830     * m_max_chain_depth.
831     */
832     uint8_t           m_max_request_frames;

834     uint16_t          m_free_queue_depth;
835     uint16_t          m_post_queue_depth;
836     uint16_t          m_max_replies;
837     uint32_t          m_free_index;
838     uint32_t          m_post_index;
839     uint8_t           m_reply_frame_size;
840     uint32_t          m_ioc_capabilities;

842     /*
843     * indicates if the firmware was upload by the driver
844     * at boot time
845     */
846     ushort_t          m_fwupload;

848     uint16_t          m_productid;

850     /*
851     * per instance data structures for dma memory resources for
852     * MPI handshake protocol. only one handshake cmd can run at a time.
853     */
854     ddi_dma_handle_t   m_hshk_dma_hdl;
855     ddi_acc_handle_t   m_hshk_acc_hdl;
856     caddr_t           m_hshk_memp;
857     size_t             m_hshk_dma_size;

859     /* Firmware version on the card at boot time */
860     uint32_t          m_fvversion;

862     /* MSI specific fields */
863     ddi_intr_handle_t *m_htable;        /* For array of interrupts */
864     int                m_intr_type;      /* What type of interrupt */
865     int                m_intr_cnt;       /* # of intrs count returned */
866     size_t             m_intr_size;      /* Size of intr array */
867     uint_t              m_intr_pri;       /* Interrupt priority */
868     int                m_intr_cap;       /* Interrupt capabilities */
869     ddi_taskq_t        *m_event_taskq;

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      4
871     /* SAS specific information */

873     union {
874         uint64_t         m_base_wwid;      /* Base WWID */
875         struct {
876             #ifdef _BIG_ENDIAN
877                 uint32_t      m_base_wwid_hi;
878                 uint32_t      m_base_wwid_lo;
879             #else
880                 uint32_t      m_base_wwid_lo;
881                 uint32_t      m_base_wwid_hi;
882             #endif
883         } sasaddr;
884     } un;

886     uint8_t           m_num_phys;        /* # of PHYS */
887     mptsas_phy_info_t m_phys_info[MPTSAS_MAX_PHYS];
888     uint8_t           m_port_chng;       /* initiator port changes */
889     MPI2_CONFIG_PAGE_MAN_0    m_MANU_page0; /* Manufacturer page 0 info */
890     MPI2_CONFIG_PAGE_MAN_1    m_MANU_page1; /* Manufacturer page 1 info */

892     /* FMA Capabilities */
893     int                m_fm_capabilities;
894     ddi_taskq_t        *m_dr_taskq;
895     int                m_mprio_enable;
896     uint8_t           m_done_traverse_dev;
897     uint8_t           m_done_traverse_smp;
898     int                m_diag_action_in_progress;
899     uint16_t          m_dev_handle;
900     uint16_t          m_smp_devhdl;

902     /* In case of reset */
903     ddi_taskq_t        *m_reset_taskq;

905     #endif /* ! codereview */
906     /*
907     * Event recording
908     */
909     uint8_t           m_event_index;
910     uint32_t          m_event_number;
911     uint32_t          m_event_mask[4];
912     mptsas_event_entry_t m_events[MPTSAS_EVENT_QUEUE_SIZE];

914     /*
915     * FW diag Buffer List
916     */
917     mptsas_fw_diagnostic_buffer_t
918         m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_COUNT];

920     /* GEN3 support */
921     uint8_t           m_MPI25;

923     /*
924     * Event Replay flag (MUR support)
925     */
926     uint8_t           m_event_replay;

928     /*
929     * IR Capable flag
930     */
931     uint8_t           m_ir_capable;

933     /*
934     * Is HBA processing a diag reset?
935     */

```

```

936     uint8_t          m_in_reset;
938
939     /* per instance cmd data structures for task management cmd
940     */
941     m_event_struct_t    m_event_task_mgmt;      /* must be last */
942     /* ... scsi_pkt_size */
943 } mptsas_t;
944 #define MPTSAS_SIZE      (sizeof (struct mptsas) - \
945                         sizeof (struct scsi_pkt) + scsi_pkt_size())
946 /*
947 * Only one of below two conditions is satisfied, we
948 * think the target is associated to the iport and
949 * allow call into mptsas_probe_lun().
950 * 1. physports == physport
951 * 2. (phymask & (1 << physport)) == 0
952 * The condition #2 is because LSI uses lowest PHY
953 * number as the value of physical port when auto port
954 * configuration.
955 */
956 #define IS_SAME_PORT(physicalport, physport, phymask, dynamicport) \
957     ((physicalport == physport) || (dynamicport && (phymask & \
958         (1 << physport))))
959
960 _NOTE(MUTEX_PROTECTS_DATA(mptsas::m_mutex, mptsas))
961 _NOTE(SCHEME_PROTECTS_DATA("safe sharing", mptsas::m_next))
962 _NOTE(SCHEME_PROTECTS_DATA("stable data", mptsas::m_dip mptsas::m_tran))
963 _NOTE(SCHEME_PROTECTS_DATA("stable data", mptsas::m_kmem_cache))
964 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_ic_dma_attr.dma_attr_sgllen))
965 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_devid))
966 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_productid))
967 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_mprio_enable))
968 _NOTE(DATA_READABLE_WITHOUT_LOCK(mptsas::m_instance))
969
970 /*
971 * These should eventually migrate into the mpt header files
972 * that may become the /kernel/misc/mpt module...
973 */
974 #define mptsas_init_std_hdr(hdl, mp, DevHandle, Lun, ChainOffset, Function) \
975     mptsas_put_msg_DevHandle(hdl, mp, DevHandle); \
976     mptsas_put_msg_ChainOffset(hdl, mp, ChainOffset); \
977     mptsas_put_msg_Function(hdl, mp, Function); \
978     mptsas_put_msg_Lun(hdl, mp, Lun)
979
980 #define mptsas_put_msg_DevHandle(hdl, mp, val) \
981     ddi_put16(hdl, &(mp)->DevHandle, (val))
982 #define mptsas_put_msg_ChainOffset(hdl, mp, val) \
983     ddi_put8(hdl, &(mp)->ChainOffset, (val))
984 #define mptsas_put_msg_Function(hdl, mp, val) \
985     ddi_put8(hdl, &(mp)->Function, (val))
986 #define mptsas_put_msg_Lun(hdl, mp, val) \
987     ddi_put8(hdl, &(mp)->LUN[1], (val))
988
989 #define mptsas_get_msg_Function(hdl, mp) \
990     ddi_get8(hdl, &(mp)->Function)
991
992 #define mptsas_get_msg_MsgFlags(hdl, mp) \
993     ddi_get8(hdl, &(mp)->MsgFlags)
994
995 #define MPTSAS_ENABLE_DRWE(hdl) \
996     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
997                 MPI2_WRSEQ_FLUSH_KEY_VALUE); \
998     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
999                 MPI2_WRSEQ_1ST_KEY_VALUE); \
1000    ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1001                 MPI2_WRSEQ_2ND_KEY_VALUE); \

```

```

1002     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1003                 MPI2_WRSEQ_3RD_KEY_VALUE); \
1004     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1005                 MPI2_WRSEQ_4TH_KEY_VALUE); \
1006     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1007                 MPI2_WRSEQ_5TH_KEY_VALUE); \
1008     ddi_put32(hdl->m_datap, &hdl->m_reg->WriteSequence, \
1009                 MPI2_WRSEQ_6TH_KEY_VALUE);
1010
1011 /*
1012 * m_options flags
1013 */
1014 #define MPTSAS_OPT_PM          0x01    /* Power Management */
1015
1016 /*
1017 * m_softstate flags
1018 */
1019 #define MPTSAS_SS_DRAINING      0x02
1020 #define MPTSAS_SS QUIESCED      0x04
1021 #define MPTSAS_SS_MSG_UNIT_RESET 0x08
1022 #define MPTSAS_DID_MSG_UNIT_RESET 0x10
1023
1024 /*
1025 * regspec defines.
1026 */
1027 #define CONFIG_SPACE      0      /* regset[0] - configuration space */
1028 #define IO_SPACE          1      /* regset[1] - used for i/o mapped device */
1029 #define MEM_SPACE          2      /* regset[2] - used for memory mapped device */
1030 #define BASE_REG2        3      /* regset[3] - used for 875 scripts ram */
1031
1032 /*
1033 * Handy constants
1034 */
1035 #define FALSE            0
1036 #define TRUE             1
1037 #define UNDEFINED        -1
1038 #define FAILED           -2
1039
1040 /*
1041 * power management.
1042 */
1043 #define MPTSAS_POWER_ON(mp) { \
1044     pci_config_put16(mp->m_config_handle, mp->m_pmcsr_offset, \
1045                     PCI_PMCSR_D0); \
1046     delay(drv_usectohz(10000)); \
1047     (void) pci_restore_config_regs(mp->m_dip); \
1048     mptsas_setup_cmd_reg(mp); \
1049 }
1050
1051 #define MPTSAS_POWER_OFF(mp) { \
1052     (void) pci_save_config_regs(mp->m_dip); \
1053     pci_config_put16(mp->m_config_handle, mp->m_pmcsr_offset, \
1054                     PCI_PMCSR_D3HOT); \
1055     mp->m_power_level = PM_LEVEL_D3; \
1056 }
1057
1058 /*
1059 * inq_dtype:
1060 * Bits 5 through 7 are the Peripheral Device Qualifier
1061 * 001b: device not connected to the LUN
1062 * Bits 0 through 4 are the Peripheral Device Type
1063 * 1fh: Unknown or no device type
1064 *
1065 * Although the inquiry may return success, the following value
1066 * means no valid LUN connected.
1067 */

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      7

1068 #define MPTAS_VALID_LUN(sd_inq) \
1069     (((sd_inq->inq_dtype & 0xe0) != 0x20) && \
1070     ((sd_inq->inq_dtype & 0x1f) != 0x1f))

1072 /*
1073  * Default is to have 10 retries on receiving QFULL status and
1074  * each retry to be after 100 ms.
1075 */
1076 #define QFULL_RETRIES          10
1077 #define QFULL_RETRY_INTERVAL    100

1079 /*
1080  * Handy macros
1081 */
1082 #define Tgt(sp)   ((sp)->cmd_pkt->pkt_address.a_target)
1083 #define Lun(sp)   ((sp)->cmd_pkt->pkt_address.a_lun)

1085 #define IS_HEX_DIGIT(n) (((n) >= '0' && (n) <= '9') || \
1086     ((n) >= 'a' && (n) <= 'f') || ((n) >= 'A' && (n) <= 'F'))

1088 /*
1089  * poll time for mptsas_pollret() and mptsas_wait_intr()
1090 */
1091 #define MPTAS_POLL_TIME        30000 /* 30 seconds */

1093 /*
1094  * default time for mptsas_do_passthru
1095 */
1096 #define MPTAS_PASS_THRU_TIME_DEFAULT 60 /* 60 seconds */

1098 /*
1099  * macro to return the effective address of a given per-target field
1100 */
1101 #define EFF_ADDR(start, offset) ((start) + (offset))

1103 #define SDEV2ADDR(devp)         (&((devp)->sd_address))
1104 #define SDEV2TRAN(devp)        ((devp)->sd_address.a_hba_tran)
1105 #define PKT2TRAN(pkt)         ((pkt)->pkt_address.a_hba_tran)
1106 #define ADDR2TRAN(ap)         ((ap)->a_hba_tran)
1107 #define DIP2TRAN(dip)         (ddi_get_driver_private(dip))

1110 #define TRAN2MPT(hba)          ((mptsas_t *) (hba)->tran_hba_private)
1111 #define DIP2MPT(dip)           (TRAN2MPT((scsi_hba_trant *)DIP2TRAN(dip)))
1112 #define SDEV2MPT(sd)           (TRAN2MPT(SDEV2TRAN(sd)))
1113 #define PKT2MPT(pkt)           (TRAN2MPT(PKT2TRAN(pkt)))

1115 #define ADDR2MPT(ap)           (TRAN2MPT(ADDR2TRAN(ap)))

1117 #define POLL_TIMEOUT           (2 * SCSI_POLL_TIMEOUT * 1000000)
1118 #define SHORT_POLL_TIMEOUT     (1000000) /* in usec, about 1 secs */
1119 #define MPTAS QUIESCE TIMEOUT 1 /* 1 sec */
1120 #define MPTAS_PM_IDLE_TIMEOUT  60 /* 60 seconds */

1122 #define MPTAS_GET_ISTAT(mpt)   (ddi_get32((mpt)->m_datap, \
1123                                         &(mpt)->m_reg->HostInterruptStatus))

1125 #define MPTAS_SET_SIGP(P) \
1126     ClrSetBits(mpt->m_devaddr + NREG_ISTAT, 0, NB_ISTAT_SIGP)

1128 #define MPTAS_RESET_SIGP(P) (void) ddi_get8(mpt->m_datap, \
1129                                         (uint8_t *) (mpt->m_devaddr + NREG_CTEST2))

1131 #define MPTAS_GET_INTCODE(P) (ddi_get32(mpt->m_datap, \
1132                                         (uint32_t *) (mpt->m_devaddr + NREG_DSPS)))

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      8

1135 #define MPTAS_START_CMD(mpt, req_desc) \
1136     ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostLow, \
1137     req_desc & 0xfffffffffu);
1138     ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostHigh, \
1139     (req_desc >> 32) & 0xfffffffffu);

1141 #define INTPENDING(mpt) \
1142     (MPTAS_GET_ISTAT(mpt) & MPI2_HIS_REPLY_DESCRIPTOR_INTERRUPT)

1144 /*
1145  * Mask all interrupts to disable
1146 */
1147 #define MPTAS_DISABLE_INTR(mpt) \
1148     ddi_put32((mpt)->m_datap, &(mpt)->m_reg->HostInterruptMask, \
1149     (MPI2_HIM_RIM | MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

1151 /*
1152  * Mask Doorbell and Reset interrupts to enable reply desc int.
1153 */
1154 #define MPTAS_ENABLE_INTR(mpt) \
1155     ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, \
1156     (MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

1158 #define MPTAS_GET_NEXT_REPLY(mpt, index) \
1159     &((uint64_t *) (void *) (mpt->m_post_queue))[index]

1161 #define MPTAS_GET_NEXT_FRAME(mpt, SMID) \
1162     (mpt->m_req_frame + (mpt->m_req_frame_size * SMID))

1164 #define ClrSetBits32(hdl, reg, clr, set) \
1165     ddi_put32(hdl, (reg), \
1166     ((ddi_get32(mpt->m_datap, (reg)) & ~(clr)) | (set)))

1168 #define ClrSetBits(reg, clr, set) \
1169     ddi_put8(mpt->m_datap, (uint8_t *) (reg), \
1170     ((ddi_get8(mpt->m_datap, (uint8_t *) (reg)) & ~(clr)) | (set)))

1172 #define MPTAS_WAITQ_RM(mpt, cmdp) \
1173     if ((cmdp = mpt->m_waitq) != NULL) { \
1174         /* If the queue is now empty fix the tail pointer */ \
1175         if ((mpt->m_waitq = cmdp->cmd_linkp) == NULL) \
1176             mpt->m_waitqtail = &mpt->m_waitq; \
1177         cmdp->cmd_linkp = NULL; \
1178         cmdp->cmd_queued = FALSE; \
1179     }

1181 #define MPTAS_TX_WAITQ_RM(mpt, cmdp) \
1182     if ((cmdp = mpt->m_tx_waitq) != NULL) { \
1183         /* If the queue is now empty fix the tail pointer */ \
1184         if ((mpt->m_tx_waitq = cmdp->cmd_linkp) == NULL) \
1185             mpt->m_tx_waitqtail = &mpt->m_tx_waitq; \
1186         cmdp->cmd_linkp = NULL; \
1187         cmdp->cmd_queued = FALSE; \
1188     }

1190 /*
1191  * defaults for the global properties
1192 */
1193 #define DEFAULT_SCSI_OPTIONS    SCSI_OPTIONS_DR
1194 #define DEFAULT_TAG_AGE_LIMIT   2
1195 #define DEFAULT_WD_TICK         1

1197 /*
1198  * invalid hostid.
1199 */

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      9
1200 #define MPTSAUTH_INVALID_HOSTID -1
1202 /*
1203  * Get/Set hostid from SCSI port configuration page
1204 */
1205 #define MPTSAUTH_GET_HOST_ID(configuration) (configuration & 0xFF)
1206 #define MPTSAUTH_SET_HOST_ID(hostid) (hostid | ((1 << hostid) << 16))
1208 /*
1209  * Config space.
1210 */
1211 #define MPTSAUTH_LATENCY_TIMER 0x40
1213 /*
1214  * Offset to firmware version
1215 */
1216 #define MPTSAUTH_FW_VERSION_OFFSET 9
1218 /*
1219  * Offset and masks to get at the ProductId field
1220 */
1221 #define MPTSAUTH_FW_PRODUCTID_OFFSET 8
1222 #define MPTSAUTH_FW_PRODUCTID_MASK 0xFFFF0000
1223 #define MPTSAUTH_FW_PRODUCTID_SHIFT 16
1225 /*
1226  * Subsystem ID for HBAs.
1227 */
1228 #define MPTSAUTH_HBA_SUBSYSTEM_ID 0x10C0
1229 #define MPTSAUTH_RHEA_SUBSYSTEM_ID 0x10B0
1231 /*
1232  * reset delay tick
1233 */
1234 #define MPTSAUTH_WATCH_RESET_DELAY_TICK 50 /* specified in milli seconds */
1236 /*
1237  * Ioc reset return values
1238 */
1239 #define MPTSAUTH_RESET_FAIL -1
1240 #define MPTSAUTH_NO_RESET 0
1241 #define MPTSAUTH_SUCCESS_HARDRESET 1
1242 #define MPTSAUTH_SUCCESS_MUR 2
1244 /*
1245  * throttle support.
1246 */
1247 #define MAX_THROTTLE 32
1248 #define HOLD_THROTTLE 0
1249 #define DRAIN_THROTTLE -1
1250 #define QFULL_THROTTLE -2
1252 /*
1253  * Passthrough/config request flags
1254 */
1255 #define MPTSAUTH_DATA_ALLOCATED 0x0001
1256 #define MPTSAUTH_DATAOUT_ALLOCATED 0x0002
1257 #define MPTSAUTH_REQUEST_POOL_CMD 0x0004
1258 #define MPTSAUTH_ADDRESS_REPLY 0x0008
1259 #define MPTSAUTH_CMD_TIMEOUT 0x0010
1261 /*
1262  * response code tlr flag
1263 */
1264 #define MPTSAUTH_SCSI_RESPONSE_CODE_TLR_OFF 0x02

```

```

new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h      10
1266 /*
1267  * System Events
1268 */
1269 #ifndef DDI_VENDOR_LSI
1270 #define DDI_VENDOR_LSI "LSI"
1271 #endif /* DDI_VENDOR_LSI */
1273 /*
1274  * Shared functions
1275 */
1276 int mptsas_save_cmd(struct mptsas *mpt, struct mptsas_cmd *cmd);
1277 void mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);
1278 void mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
1279 void mptsas_log(struct mptsas *mpt, int level, char *fmt, ...);
1280 int mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime);
1281 int mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)());
1282 int mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
1283 	uint8_t type, int mode);
1284 int mptsas_check_flash(mptsas_t *mpt, caddr_t origfile, uint32_t size,
1285 	uint8_t type, int mode);
1286 int mptsas_download_firmware();
1287 int mptsas_can_download_firmware();
1288 int mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_stattep);
1289 void mptsas_dma_free(mptsas_dma_alloc_state_t *dma_stattep);
1290 mptsas_phymask_t mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport);
1291 void mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd);
1292 int mptsas_check_acc_handle(ddi_acc_handle_t handle);
1293 int mptsas_check_dma_handle(ddi_dma_handle_t handle);
1294 void mptsas_fn_eREPORT(mptsas_t *mpt, char *detail);
1295 int mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
1296 	ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
1297 	uint32_t alloc_size, ddi_dma_cookie_t *cookiep);
1298 void mptsas_dma_addr_destroy(ddi_dma_handle_t *, ddi_acc_handle_t *);
1300 /*
1301  * impl functions
1302 */
1303 int mptsas_ioc_wait_for_response(mptsas_t *mpt);
1304 int mptsas_ioc_wait_for_doorbell(mptsas_t *mpt);
1305 int mptsas_ioc_reset(mptsas_t *mpt, int);
1306 int mptsas_send_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
1307 	ddi_acc_handle_t accesssp);
1308 int mptsas_get_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
1309 	ddi_acc_handle_t accesssp);
1310 int mptsas_send_config_request_msg(mptsas_t *mpt, uint8_t action,
1311 	uint8_t pagetype, uint32_t pageaddress, uint8_t pagenumber,
1312 	uint8_t pageversion, uint8_t pagelength, uint32_t SGEflagslength,
1313 	uint64_t SGEaddress);
1314 int mptsas_send_extended_config_request_msg(mptsas_t *mpt, uint8_t action,
1315 	uint8_t extpagetype, uint32_t pageaddress, uint8_t pagenumber,
1316 	uint8_t pageversion, uint16_t extpagelength,
1317 	uint32_t SGEflagslength, uint64_t SGEaddress);
1319 int mptsas_request_from_pool(mptsas_t *mpt, mptsas_cmd_t **cmd,
1320 	struct scsi_pkt **pkt);
1321 void mptsas_return_to_pool(mptsas_t *mpt, mptsas_cmd_t *cmd);
1322 void mptsas_destroy_ioc_event_cmd(mptsas_t *mpt);
1323 void mptsas_start_config_page_access(mptsas_t *mpt, mptsas_cmd_t *cmd);
1324 int mptsas_access_config_page(mptsas_t *mpt, uint8_t action, uint8_t page_type,
1325 	uint8_t page_number, uint32_t page_address, int (*callback) (mptsas_t *,
1326 	caddr_t, ddi_acc_handle_t, uint16_t, uint32_t, va_list), ...);
1328 int mptsas_ioc_task_management(mptsas_t *mpt, int task_type,
1329 	uint16_t dev_handle, int lun, uint8_t *reply, uint32_t reply_size,
1330 	int mode);
1331 int mptsas_send_event_ack(mptsas_t *mpt, uint32_t event, uint32_t eventctx);

```

```

1332 void mptsas_send_pending_event_ack(mptsas_t *mpt);
1333 void mptsas_set_throttle(struct mptsas *mpt, mptsas_target_t *ptgt, int what);
1334 void mptsas_handle_restart_ioc(void *mpt);
1335 #endif /* ! codereview */
1336 int mptsas_restart_ioc(mptsas_t *mpt);
1337 void mptsas_update_driver_data(struct mptsas *mpt);
1338 uint64_t mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun);

1340 /*
1341  * init functions
1342 */
1343 int mptsas_ioc_get_facts(mptsas_t *mpt);
1344 int mptsas_ioc_get_port_facts(mptsas_t *mpt, int port);
1345 int mptsas_ioc_enable_port(mptsas_t *mpt);
1346 int mptsas_ioc_enable_event_notification(mptsas_t *mpt);
1347 int mptsas_ioc_init(mptsas_t *mpt);

1349 /*
1350  * configuration pages operation
1351 */
1352 int mptsas_get_sas_device_page0(mptsas_t *mpt, uint32_t page_address,
1353     uint16_t *dev_handle, uint64_t *sas_wnn, uint32_t *dev_info,
1354     uint8_t *physport, uint8_t *phnum, uint16_t *pdevhandle,
1355     uint16_t *slot_num, uint16_t *enclosure, uint16_t *io_flags);
1356 int mptsas_get_sas_io_unit_page(mptsas_t *mpt);
1357 int mptsas_get_sas_io_unit_page_hndshk(mptsas_t *mpt);
1358 int mptsas_get_sas_expander_page0(mptsas_t *mpt, uint32_t page_address,
1359     mptsas_smp_t *info);
1360 int mptsas_set_ioc_params(mptsas_t *mpt);
1361 int mptsas_get_manufacture_page5(mptsas_t *mpt);
1362 int mptsas_get_sas_port_page0(mptsas_t *mpt, uint32_t page_address,
1363     uint64_t *sas_wnn, uint8_t *portwidth);
1364 int mptsas_get_bios_page3(mptsas_t *mpt, uint32_t *bios_version);
1365 int
1366 mptsas_get_sas_phy_page0(mptsas_t *mpt, uint32_t page_address,
1367     smhba_info_t *info);
1368 int
1369 mptsas_get_sas_phy_page1(mptsas_t *mpt, uint32_t page_address,
1370     smhba_info_t *info);
1371 int
1372 mptsas_get_manufacture_page0(mptsas_t *mpt);
1373 void
1374 mptsas_create_phy_stats(mptsas_t *mpt, char *iport, dev_info_t *dip);
1375 void mptsas_destroy_phy_stats(mptsas_t *mpt);
1376 int mptsas_smhba_phy_init(mptsas_t *mpt);
1377 /*
1378  * RAID functions
1379 */
1380 int mptsas_get_raid_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol);
1381 int mptsas_get_raid_info(mptsas_t *mpt);
1382 int mptsas_get_physdisk_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol,
1383     uint8_t physdisknum);
1384 int mptsas_delete_volume(mptsas_t *mpt, uint16_t valid);
1385 void mptsas_raid_action_system_shutdown(mptsas_t *mpt);

1387 #define MPTSAS_IOCSTATUS(status) (status & MPI2_IOCSTATUS_MASK)
1388 /*
1389  * debugging.
1390  * MPTSAS_DBGLOG_LINECNT must be a power of 2.
1391 */
1392 #define MPTSAS_DBGLOG_LINECNT 128
1393 #define MPTSAS_DBGLOG_LINELEN 256
1394 #define MPTSAS_DBGLOG_BUFSIZE (MPTSAS_DBGLOG_LINECNT * MPTSAS_DBGLOG_LINELEN)

1396 #if defined(MPTSAS_DEBUG)

```

```

1398 extern uint32_t mptsas_debugprt_flags;
1399 extern uint32_t mptsas_debuglog_flags;

1401 void mptsas_printf(char *fmt, ...);
1402 void mptsas_debug_log(char *fmt, ...);

1404 #define MPTSAS_DBGPR(m, args) \
1405     if (mptsas_debugprt_flags & (m)) \
1406         mptsas_printf args; \
1407     if (mptsas_debuglog_flags & (m)) \
1408         mptsas_debug_log args
1409 #else /* ! defined(MPTSAS_DEBUG) */
1410 #define MPTSAS_DBGPR(m, args)
1411 #endif /* defined(MPTSAS_DEBUG) */

1413 #define NDBG0(args) MPTSAS_DBGPR(0x01, args) /* init */
1414 #define NDBG1(args) MPTSAS_DBGPR(0x02, args) /* normal running */
1415 #define NDBG2(args) MPTSAS_DBGPR(0x04, args) /* property handling */
1416 #define NDBG3(args) MPTSAS_DBGPR(0x08, args) /* pkt handling */
1418 #define NDBG4(args) MPTSAS_DBGPR(0x10, args) /* kmem alloc/free */
1419 #define NDBG5(args) MPTSAS_DBGPR(0x20, args) /* polled cmd */
1420 #define NDBG6(args) MPTSAS_DBGPR(0x40, args) /* interrupts */
1421 #define NDBG7(args) MPTSAS_DBGPR(0x80, args) /* queue handling */
1423 #define NDBG8(args) MPTSAS_DBGPR(0x100, args) /* arq */
1424 #define NDBG9(args) MPTSAS_DBGPR(0x200, args) /* Tagged Q'ing */
1425 #define NDBG10(args) MPTSAS_DBGPR(0x400, args) /* halting chip */
1426 #define NDBG11(args) MPTSAS_DBGPR(0x800, args) /* power management */
1428 #define NDBG12(args) MPTSAS_DBGPR(0x1000, args) /* enumeration */
1429 #define NDBG13(args) MPTSAS_DBGPR(0x2000, args) /* configuration page */
1430 #define NDBG14(args) MPTSAS_DBGPR(0x4000, args) /* LED control */
1431 #define NDBG15(args) MPTSAS_DBGPR(0x8000, args) /* Passthrough */
1433 #define NDBG16(args) MPTSAS_DBGPR(0x10000, args) /* SAS Broadcasts */
1434 #define NDBG17(args) MPTSAS_DBGPR(0x20000, args) /* scatter/gather */
1435 #define NDBG18(args) MPTSAS_DBGPR(0x40000, args) /* */
1436 #define NDBG19(args) MPTSAS_DBGPR(0x80000, args) /* handshaking */
1438 #define NDBG20(args) MPTSAS_DBGPR(0x100000, args) /* events */
1439 #define NDBG21(args) MPTSAS_DBGPR(0x200000, args) /* dma */
1440 #define NDBG22(args) MPTSAS_DBGPR(0x400000, args) /* reset */
1441 #define NDBG23(args) MPTSAS_DBGPR(0x800000, args) /* abort */
1443 #define NDBG24(args) MPTSAS_DBGPR(0x1000000, args) /* capabilities */
1444 #define NDBG25(args) MPTSAS_DBGPR(0x2000000, args) /* flushing */
1445 #define NDBG26(args) MPTSAS_DBGPR(0x4000000, args) /* */
1446 #define NDBG27(args) MPTSAS_DBGPR(0x8000000, args) /* passthrough */
1448 #define NDBG28(args) MPTSAS_DBGPR(0x10000000, args) /* hotplug */
1449 #define NDBG29(args) MPTSAS_DBGPR(0x20000000, args) /* timeouts */
1450 #define NDBG30(args) MPTSAS_DBGPR(0x40000000, args) /* mptsas_watch */
1451 #define NDBG31(args) MPTSAS_DBGPR(0x80000000, args) /* negotiations */

1453 /*
1454  * auto request sense
1455 */
1456 #define RQ_MAKECOM_COMMON(pkt, flag, cmd) \
1457     (pkt)->pkt_flags = (flag), \
1458     ((union scsi_cdb *) (pkt)->pkt_cdbp)->scc_cmd = (cmd), \
1459     ((union scsi_cdb *) (pkt)->pkt_cdbp)->scc_lun = \
1460         (pkt)->pkt_address.a_lun

1462 #define RQ_MAKECOM_G0(pkt, flag, cmd, addr, cnt) \
1463     RQ_MAKECOM_COMMON((pkt), (flag), (cmd)), \

```

```
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h          13
1464     FORMG0ADDR(((union scsi_cdb *)(pkt)->pkt_cdbp), (addr)), \
1465     FORMG0COUNT(((union scsi_cdb *)(pkt)->pkt_cdbp), (cnt))
1468 #ifdef __cplusplus
1469 }
1470 #endif
1472 #endif /* _SYS_SCSI_ADAPTERS_MPTVAR_H */
```