```
**********************************************************
  220825 Tue Nov  6 14:28:28 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_


/*
 * ************************************************************************** *
 *                                                                            *
 *               common entry points - for autoconfiguration                  *
 *                                                                            *
 * ************************************************************************** *
 */
/*
 * attach - adds a device to the system as part of initialization
 * @dip:
 * @cmd:
 *
 * The kernel calls a driver's attach() entry point to attach an instance of
 * a device (for MegaRAID, it is instance of a controller) or to resume
 * operation for an instance of a device that has been suspended or has been
 * shut down by the power management framework
 * The attach() entry point typically includes the following types of
 * processing:
 * - allocate a soft-state structure for the device instance (for MegaRAID,
 *   controller instance)
 * - initialize per-instance mutexes
 * - initialize condition variables
 * - register the device's interrupts (for MegaRAID, controller's interrupts)
 * - map the registers and memory of the device instance (for MegaRAID,
 *   controller instance)
 * - create minor device nodes for the device instance (for MegaRAID,
 *   controller instance)
 * - report that the device instance (for MegaRAID, controller instance) has
 *   attached
 */
static int
mrsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int             instance_no;
        int             nregs;
        int             i = 0;
        uint8_t         irq;
        uint16_t        vendor_id;
        uint16_t        device_id;
        uint16_t        subsysvid;
        uint16_t        subsysid;
        uint16_t        command;
        off_t           reglength = 0;
        int             intr_types = 0;
        char            *data;

        scsi_hba_tran_t         *tran;
        ddi_dma_attr_t  tran_dma_attr;
        struct mrsas_instance   *instance;

        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

        /* CONSTCOND */
        ASSERT(NO_COMPETING_THREADS);

        instance_no = ddi_get_instance(dip);

        /*
```



```
         * check to see whether this device is in a DMA-capable slot.
         */
        if (ddi_slaveonly(dip) == DDI_SUCCESS) {
                cmn_err(CE_WARN,
                    "mr_sas%d: Device in slave-only slot, unused",
                    instance_no);
                return (DDI_FAILURE);
        }

        switch (cmd) {
        case DDI_ATTACH:
                /* allocate the soft state for the instance */
                if (ddi_soft_state_zalloc(mrsas_state, instance_no)
                    != DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                            "mr_sas%d: Failed to allocate soft state",
                            instance_no);
                        return (DDI_FAILURE);
                }

                instance = (struct mrsas_instance *)ddi_get_soft_state
                    (mrsas_state, instance_no);

                if (instance == NULL) {
                        cmn_err(CE_WARN,
                            "mr_sas%d: Bad soft state", instance_no);
                        ddi_soft_state_free(mrsas_state, instance_no);
                        return (DDI_FAILURE);
                }

                instance->unroll.softs  = 1;

                /* Setup the PCI configuration space handles */
                if (pci_config_setup(dip, &instance->pci_handle) !=
                    DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                            "mr_sas%d: pci config setup failed ",
                            instance_no);

                        ddi_soft_state_free(mrsas_state, instance_no);
                        return (DDI_FAILURE);
                }

                if (ddi_dev_nregs(dip, &nregs) != DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                            "mr_sas: failed to get registers.");

                        pci_config_teardown(&instance->pci_handle);
                        ddi_soft_state_free(mrsas_state, instance_no);
                        return (DDI_FAILURE);
                }

                vendor_id = pci_config_get16(instance->pci_handle,
                    PCI_CONF_VENID);
                device_id = pci_config_get16(instance->pci_handle,
                    PCI_CONF_DEVID);

                subsysvid = pci_config_get16(instance->pci_handle,
                    PCI_CONF_SUBVENID);
                subsysid = pci_config_get16(instance->pci_handle,
                    PCI_CONF_SUBSYSID);

                pci_config_put16(instance->pci_handle, PCI_CONF_COMM,
                    (pci_config_get16(instance->pci_handle,
                    PCI_CONF_COMM) | PCI_COMM_ME));
                irq = pci_config_get8(instance->pci_handle,
```

```
542                     PCI_CONF_ILINE);

544             con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
545                 "0x%x:0x%x 0x%x:0x%x, irq:%d drv-ver:%s",
546                 instance_no, vendor_id, device_id, subsysvid,
547                 subsysid, irq, MRSAS_VERSION));

549             /* enable bus-mastering */
550             command = pci_config_get16(instance->pci_handle,
551                 PCI_CONF_COMM);

553             if (!(command & PCI_COMM_ME)) {
554                     command |= PCI_COMM_ME;

556                     pci_config_put16(instance->pci_handle,
557                         PCI_CONF_COMM, command);

559                     con_log(CL_ANN, (CE_CONT, "mr_sas%d: "
560                         "enable bus-mastering", instance_no));
561             } else {
562                     con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
563                         "bus-mastering already set", instance_no));
564             }

566             /* initialize function pointers */
567             switch (device_id) {
568             case PCI_DEVICE_ID_LSI_TBOLT:
569             case PCI_DEVICE_ID_LSI_INVADER:
570                     con_log(CL_ANN, (CE_NOTE,
571                         "mr_sas: 2208 T.B. device detected"));

573                     instance->func_ptr =
574                         &mrsas_function_template_fusion;
575                     instance->tbolt = 1;
576                     break;

578             case PCI_DEVICE_ID_LSI_2108VDE:
579             case PCI_DEVICE_ID_LSI_2108V:
580                     con_log(CL_ANN, (CE_NOTE,
581                         "mr_sas: 2108 Liberator device detected"));

583                     instance->func_ptr =
584                         &mrsas_function_template_ppc;
585                     break;

587             default:
588                     cmn_err(CE_WARN,
589                         "mr_sas: Invalid device detected");

591                     pci_config_teardown(&instance->pci_handle);
592                     ddi_soft_state_free(mrsas_state, instance_no);
593                     return (DDI_FAILURE);
594             }

596             instance->baseaddress = pci_config_get32(
597                 instance->pci_handle, PCI_CONF_BASE0);
598             instance->baseaddress &= 0x0fffc;

600             instance->dip           = dip;
601             instance->vendor_id     = vendor_id;
602             instance->device_id     = device_id;
603             instance->subsysvid     = subsysvid;
604             instance->subsysid      = subsysid;
605             instance->instance      = instance_no;

607             /* Initialize FMA */
```



```
608             instance->fm_capabilities = ddi_prop_get_int(
609                 DDI_DEV_T_ANY, instance->dip, DDI_PROP_DONTPASS,
610                 "fm-capable", DDI_FM_EREPORT_CAPABLE |
611                 DDI_FM_ACCCHK_CAPABLE | DDI_FM_DMACHK_CAPABLE
612                 | DDI_FM_ERRCB_CAPABLE);

614             mrsas_fm_init(instance);

616             /* Setup register map */
617             if ((ddi_dev_regsize(instance->dip,
618                 REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
619                 reglength < MINIMUM_MFI_MEM_SZ) {
620                     goto fail_attach;
621             }
622             if (reglength > DEFAULT_MFI_MEM_SZ) {
623                     reglength = DEFAULT_MFI_MEM_SZ;
624                     con_log(CL_DLEVEL1, (CE_NOTE,
625                         "mr_sas: register length to map is 0x%lx bytes",
626                         reglength));
627             }
628             if (ddi_regs_map_setup(instance->dip,
629                 REGISTER_SET_IO_2108, &instance->regmap, 0,
630                 reglength, &endian_attr, &instance->regmap_handle)
631                 != DDI_SUCCESS) {
632                     cmn_err(CE_WARN,
633                         "mr_sas: couldn't map control registers");
634                     goto fail_attach;
635             }

637             instance->unroll.regs = 1;

639             /*
640              * Disable Interrupt Now.
641              * Setup Software interrupt
642              */
643             instance->func_ptr->disable_intr(instance);

645             if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
646                 "mrsas-enable-msi", &data) == DDI_SUCCESS) {
647                     if (strncmp(data, "no", 3) == 0) {
648                             msi_enable = 0;
649                             con_log(CL_ANN1, (CE_WARN,
650                                 "msi_enable = %d disabled", msi_enable));
651                     }
652                     ddi_prop_free(data);
653             }

655             con_log(CL_DLEVEL1, (CE_NOTE, "msi_enable = %d", msi_enable));

657             if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
658                 "mrsas-enable-fp", &data) == DDI_SUCCESS) {
659                     if (strncmp(data, "no", 3) == 0) {
660                             enable_fp = 0;
661                             cmn_err(CE_NOTE,
662                                 "enable_fp = %d, Fast-Path disabled.\n",
663                                 enable_fp);
664                     }

666                     ddi_prop_free(data);
667             }

669             con_log(CL_DLEVEL1, (CE_NOTE, "enable_fp = %d\n", enable_fp));

671             /* Check for all supported interrupt types */
672             if (ddi_intr_get_supported_types(
673                 dip, &intr_types) != DDI_SUCCESS) {
```

```
674                     cmn_err(CE_WARN,
675                         "ddi_intr_get_supported_types() failed");
676                     goto fail_attach;
677             }

679             con_log(CL_DLEVEL1, (CE_NOTE,
680                 "ddi_intr_get_supported_types() ret: 0x%x", intr_types));

682             /* Initialize and Setup Interrupt handler */
683             if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
684                     if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
685                         DDI_SUCCESS) {
686                             cmn_err(CE_WARN,
687                                 "MSIX interrupt query failed");
688                             goto fail_attach;
689                     }
690                     instance->intr_type = DDI_INTR_TYPE_MSIX;
691             } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
692                     if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
693                         DDI_SUCCESS) {
694                             cmn_err(CE_WARN,
695                                 "MSI interrupt query failed");
696                             goto fail_attach;
697                     }
698                     instance->intr_type = DDI_INTR_TYPE_MSI;
699             } else if (intr_types & DDI_INTR_TYPE_FIXED) {
700                     msi_enable = 0;
701                     if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
702                         DDI_SUCCESS) {
703                             cmn_err(CE_WARN,
704                                 "FIXED interrupt query failed");
705                             goto fail_attach;
706                     }
707                     instance->intr_type = DDI_INTR_TYPE_FIXED;
708             } else {
709                     cmn_err(CE_WARN, "Device cannot "
710                         "suppport either FIXED or MSI/X "
711                         "interrupts");
712                     goto fail_attach;
713             }

715             instance->unroll.intr = 1;

717             if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
718                 "mrsas-enable-ctio", &data) == DDI_SUCCESS) {
719                     if (strncmp(data, "no", 3) == 0) {
720                             ctio_enable = 0;
721                             con_log(CL_ANN1, (CE_WARN,
722                                 "ctio_enable = %d disabled", ctio_enable));
723                     }
724                     ddi_prop_free(data);
725             }

727             con_log(CL_DLEVEL1, (CE_WARN, "ctio_enable = %d", ctio_enable));

729             /* setup the mfi based low level driver */
730             if (mrsas_init_adapter(instance) != DDI_SUCCESS) {
731                     cmn_err(CE_WARN, "mr_sas: "
732                         "could not initialize the low level driver");

734                     goto fail_attach;
735             }

737             /* Initialize all Mutex */
738             INIT_LIST_HEAD(&instance->completed_pool_list);
739             mutex_init(&instance->completed_pool_mtx, NULL,
```



```
740                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
739             mutex_init(&instance->completed_pool_mtx,
740                 "completed_pool_mtx", MUTEX_DRIVER,
741                 DDI_INTR_PRI(instance->intr_pri));

742             mutex_init(&instance->sync_map_mtx, NULL,
743                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
743             mutex_init(&instance->sync_map_mtx,
744                 "sync_map_mtx", MUTEX_DRIVER,
745                 DDI_INTR_PRI(instance->intr_pri));

745             mutex_init(&instance->app_cmd_pool_mtx, NULL,
746                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
747             mutex_init(&instance->app_cmd_pool_mtx,
748                 "app_cmd_pool_mtx", MUTEX_DRIVER,
749                 DDI_INTR_PRI(instance->intr_pri));

748             mutex_init(&instance->config_dev_mtx, NULL,
751             mutex_init(&instance->config_dev_mtx, "config_dev_mtx",
749                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

751             mutex_init(&instance->cmd_pend_mtx, NULL,
754             mutex_init(&instance->cmd_pend_mtx, "cmd_pend_mtx",
752                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

754             mutex_init(&instance->ocr_flags_mtx, NULL,
757             mutex_init(&instance->ocr_flags_mtx, "ocr_flags_mtx",
755                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

757             mutex_init(&instance->int_cmd_mtx, NULL,
760             mutex_init(&instance->int_cmd_mtx, "int_cmd_mtx",
758                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
759             cv_init(&instance->int_cmd_cv, NULL, CV_DRIVER, NULL);

761             mutex_init(&instance->cmd_pool_mtx, NULL,
764             mutex_init(&instance->cmd_pool_mtx, "cmd_pool_mtx",
762                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

764             mutex_init(&instance->reg_write_mtx, NULL,
767             mutex_init(&instance->reg_write_mtx, "reg_write_mtx",
765                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

767             if (instance->tbolt) {
768                     mutex_init(&instance->cmd_app_pool_mtx, NULL,
769                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
771                     mutex_init(&instance->cmd_app_pool_mtx,
772                         "cmd_app_pool_mtx", MUTEX_DRIVER,
773                         DDI_INTR_PRI(instance->intr_pri));

771                     mutex_init(&instance->chip_mtx, NULL,
772                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
775                     mutex_init(&instance->chip_mtx,
776                         "chip_mtx", MUTEX_DRIVER,
777                         DDI_INTR_PRI(instance->intr_pri));

774             }

776             instance->unroll.mutexs = 1;

778             instance->timeout_id = (timeout_id_t)-1;

780             /* Register our soft-isr for highlevel interrupts. */
781             instance->isr_level = instance->intr_pri;
782             if (!(instance->tbolt)) {
783                     if (instance->isr_level == HIGH_LEVEL_INTR) {
784                             if (ddi_add_softintr(dip,
```

```
785                                     DDI_SOFTINT_HIGH,
786                                     &instance->soft_intr_id, NULL, NULL,
787                                     mrsas_softintr, (caddr_t)instance) !=
788                                     DDI_SUCCESS) {
789                                         cmn_err(CE_WARN,
790                                             "Software ISR did not register");

792                                         goto fail_attach;
793                                 }

795                                 instance->unroll.soft_isr = 1;

797                         }
798                 }

800                 instance->softint_running = 0;

802                 /* Allocate a transport structure */
803                 tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

805                 if (tran == NULL) {
806                         cmn_err(CE_WARN,
807                             "scsi_hba_tran_alloc failed");
808                         goto fail_attach;
809                 }

811                 instance->tran = tran;
812                 instance->unroll.tran = 1;

814                 tran->tran_hba_private  = instance;
815                 tran->tran_tgt_init     = mrsas_tran_tgt_init;
816                 tran->tran_tgt_probe    = scsi_hba_probe;
817                 tran->tran_tgt_free     = mrsas_tran_tgt_free;
818                 if (instance->tbolt) {
819                         tran->tran_init_pkt     =
820                             mrsas_tbolt_tran_init_pkt;
821                         tran->tran_start        =
822                             mrsas_tbolt_tran_start;
823                 } else {
824                         tran->tran_init_pkt     = mrsas_tran_init_pkt;
825                         tran->tran_start        = mrsas_tran_start;
826                 }
827                 tran->tran_abort        = mrsas_tran_abort;
828                 tran->tran_reset        = mrsas_tran_reset;
829                 tran->tran_getcap       = mrsas_tran_getcap;
830                 tran->tran_setcap       = mrsas_tran_setcap;
831                 tran->tran_destroy_pkt  = mrsas_tran_destroy_pkt;
832                 tran->tran_dmafree      = mrsas_tran_dmafree;
833                 tran->tran_sync_pkt     = mrsas_tran_sync_pkt;
834                 tran->tran_quiesce      = mrsas_tran_quiesce;
835                 tran->tran_unquiesce    = mrsas_tran_unquiesce;
836                 tran->tran_bus_config   = mrsas_tran_bus_config;

838                 if (mrsas_relaxed_ordering)
839                         mrsas_generic_dma_attr.dma_attr_flags |=
840                             DDI_DMA_RELAXED_ORDERING;


843                 tran_dma_attr = mrsas_generic_dma_attr;
844                 tran_dma_attr.dma_attr_sgllen = instance->max_num_sge;

846                 /* Attach this instance of the hba */
847                 if (scsi_hba_attach_setup(dip, &tran_dma_attr, tran, 0)
848                     != DDI_SUCCESS) {
849                         cmn_err(CE_WARN,
850                             "scsi_hba_attach failed");
```



```
852                         goto fail_attach;
853                 }
854                 instance->unroll.tranSetup = 1;
855                 con_log(CL_ANN1,
856                     (CE_CONT, "scsi_hba_attach_setup()  done."));

858                 /* create devctl node for cfgadm command */
859                 if (ddi_create_minor_node(dip, "devctl",
860                     S_IFCHR, INST2DEVCTL(instance_no),
861                     DDI_NT_SCSI_NEXUS, 0) == DDI_FAILURE) {
862                         cmn_err(CE_WARN,
863                             "mr_sas: failed to create devctl node.");

865                         goto fail_attach;
866                 }

868                 instance->unroll.devctl = 1;

870                 /* create scsi node for cfgadm command */
871                 if (ddi_create_minor_node(dip, "scsi", S_IFCHR,
872                     INST2SCSI(instance_no), DDI_NT_SCSI_ATTACHMENT_POINT, 0) ==
873                     DDI_FAILURE) {
874                         cmn_err(CE_WARN,
875                             "mr_sas: failed to create scsi node.");

877                         goto fail_attach;
878                 }

880                 instance->unroll.scsictl = 1;

882                 (void) sprintf(instance->iocnode, "%d:lsirdctl",
883                     instance_no);

885                 /*
886                  * Create a node for applications
887                  * for issuing ioctl to the driver.
888                  */
889                 if (ddi_create_minor_node(dip, instance->iocnode,
890                     S_IFCHR, INST2LSIRDCTL(instance_no), DDI_PSEUDO, 0) ==
891                     DDI_FAILURE) {
892                         cmn_err(CE_WARN,
893                             "mr_sas: failed to create ioctl node.");

895                         goto fail_attach;
896                 }

898                 instance->unroll.ioctl = 1;

900                 /* Create a taskq to handle dr events */
901                 if ((instance->taskq = ddi_taskq_create(dip,
902                     "mrsas_dr_taskq", 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
903                         cmn_err(CE_WARN,
904                             "mr_sas: failed to create taskq ");
905                         instance->taskq = NULL;
906                         goto fail_attach;
907                 }
908                 instance->unroll.taskq = 1;
909                 con_log(CL_ANN1, (CE_CONT, "ddi_taskq_create()  done."));

911                 /* enable interrupt */
912                 instance->func_ptr->enable_intr(instance);

914                 /* initiate AEN */
915                 if (start_mfi_aen(instance)) {
916                         cmn_err(CE_WARN,
```

```
917                             "mr_sas: failed to initiate AEN.");
918                     goto fail_attach;
919             }
920             instance->unroll.aenPend = 1;
921             con_log(CL_ANN1,
922                 (CE_CONT, "AEN started for instance %d.", instance_no));

924             /* Finally! We are on the air.  */
925             ddi_report_dev(dip);

927             /* FMA handle checking. */
928             if (mrsas_check_acc_handle(instance->regmap_handle) !=
929                 DDI_SUCCESS) {
930                     goto fail_attach;
931             }
932             if (mrsas_check_acc_handle(instance->pci_handle) !=
933                 DDI_SUCCESS) {
934                     goto fail_attach;
935             }

937             instance->mr_ld_list =
938                 kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
939                 KM_SLEEP);
945             if (instance->mr_ld_list == NULL) {
946                     cmn_err(CE_WARN, "mr_sas attach(): "
947                         "failed to allocate ld_list array");
948                     goto fail_attach;
949             }
940             instance->unroll.ldlist_buff = 1;

942 #ifdef PDSUPPORT
943             if (instance->tbolt) {
944                     instance->mr_tbolt_pd_max = MRSAS_TBOLT_PD_TGT_MAX;
945                     instance->mr_tbolt_pd_list =
946                         kmem_zalloc(MRSAS_TBOLT_GET_PD_MAX(instance) *
947                         sizeof (struct mrsas_tbolt_pd), KM_SLEEP);
948                     ASSERT(instance->mr_tbolt_pd_list);
949                     for (i = 0; i < instance->mr_tbolt_pd_max; i++) {
950                             instance->mr_tbolt_pd_list[i].lun_type =
951                                 MRSAS_TBOLT_PD_LUN;
952                             instance->mr_tbolt_pd_list[i].dev_id =
953                                 (uint8_t)i;
954                     }

956                     instance->unroll.pdlist_buff = 1;
957             }
958 #endif
959             break;
960     case DDI_PM_RESUME:
961             con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_PM_RESUME"));
962             break;
963     case DDI_RESUME:
964             con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_RESUME"));
965             break;
966     default:
967             con_log(CL_ANN,
968                 (CE_WARN, "mr_sas: invalid attach cmd=%x", cmd));
969             return (DDI_FAILURE);
970     }


973     con_log(CL_DLEVEL1,
974         (CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
975         instance_no));
976     return (DDI_SUCCESS);
```



```
978 fail_attach:

980     mrsas_undo_resources(dip, instance);

982     mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
983     ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

985     mrsas_fm_fini(instance);

987     pci_config_teardown(&instance->pci_handle);
988     ddi_soft_state_free(mrsas_state, instance_no);

990     con_log(CL_ANN, (CE_WARN, "mr_sas: return failure from mrsas_attach"));

992     cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d",
993         instance_no);

995     return (DDI_FAILURE);
996 }
_____unchanged_portion_omitted_

1400 /*
1401  * ioctl - performs a range of I/O commands for character drivers
1402  * @dev:
1403  * @cmd:
1404  * @arg:
1405  * @mode:
1406  * @credp:
1407  * @rvalp:
1408  *
1409  * ioctl() routine must make sure that user data is copied into or out of the
1410  * kernel address space explicitly using copyin(), copyout(), ddi_copyin(),
1411  * and ddi_copyout(), as appropriate.
1412  * This is a wrapper routine to serialize access to the actual ioctl routine.
1413  * ioctl() should return 0 on success, or the appropriate error number. The
1414  * driver may also set the value returned to the calling process through rvalp.
1415  */

1417 static int
1418 mrsas_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp,
1419     int *rvalp)
1420 {
1421     int     rval = 0;

1423     struct mrsas_instance   *instance;
1424     struct mrsas_ioctl      *ioctl;
1425     struct mrsas_aen        aen;
1426     con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1428     instance = ddi_get_soft_state(mrsas_state, MINOR2INST(getminor(dev)));

1430     if (instance == NULL) {
1431             /* invalid minor number */
1432             con_log(CL_ANN, (CE_WARN, "mr_sas: adapter not found."));
1433             return (ENXIO);
1434     }

1436     ioctl = (struct mrsas_ioctl *)kmem_zalloc(sizeof (struct mrsas_ioctl),
1437         KM_SLEEP);
1438     ASSERT(ioctl);
1448     if (ioctl == NULL) {
1449             /* Failed to allocate memory for ioctl  */
1450             con_log(CL_ANN, (CE_WARN, "mr_sas_ioctl: "
1451                 "failed to allocate memory for ioctl"));
1452             return (ENOMEM);
1453     }
```

```
1440         switch ((uint_t)cmd) {
1441                 case MRSAS_IOCTL_FIRMWARE:
1442                         if (ddi_copyin((void *)arg, ioctl,
1443                             sizeof (struct mrsas_ioctl), mode)) {
1444                                 con_log(CL_ANN, (CE_WARN, "mrsas_ioctl: "
1445                                     "ERROR IOCTL copyin"));
1446                                 kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1447                                 return (EFAULT);
1448                         }

1450                         if (ioctl->control_code == MRSAS_DRIVER_IOCTL_COMMON) {
1451                                 rval = handle_drv_ioctl(instance, ioctl, mode);
1452                         } else {
1453                                 rval = handle_mfi_ioctl(instance, ioctl, mode);
1454                         }

1456                         if (ddi_copyout((void *)ioctl, (void *)arg,
1457                             (sizeof (struct mrsas_ioctl) - 1), mode)) {
1458                                 con_log(CL_ANN, (CE_WARN,
1459                                     "mrsas_ioctl: copy_to_user failed"));
1460                                 rval = 1;
1461                         }

1463                         break;
1464                 case MRSAS_IOCTL_AEN:
1480                         con_log(CL_ANN,
1481                             (CE_NOTE, "mrsas_ioctl: IOCTL Register AEN.\n"));

1465                         if (ddi_copyin((void *) arg, &aen,
1466                             sizeof (struct mrsas_aen), mode)) {
1467                                 con_log(CL_ANN, (CE_WARN,
1468                                     "mrsas_ioctl: ERROR AEN copyin"));
1469                                 kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1470                                 return (EFAULT);
1471                         }

1473                         rval = handle_mfi_aen(instance, &aen);

1475                         if (ddi_copyout((void *) &aen, (void *)arg,
1476                             sizeof (struct mrsas_aen), mode)) {
1477                                 con_log(CL_ANN, (CE_WARN,
1478                                     "mrsas_ioctl: copy_to_user failed"));
1479                                 rval = 1;
1480                         }

1482                         break;
1483                 default:
1484                         rval = scsi_hba_ioctl(dev, cmd, arg,
1485                             mode, credp, rvalp);

1487                         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_ioctl: "
1488                             "scsi_hba_ioctl called, ret = %x.", rval));
1489         }

1491         kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1492         return (rval);
1493 }
```

*unchanged_portion_omitted_*

```
3157 /*
3158  * mrsas_alloc_cmd_pool
3159  */
3160 int
3161 mrsas_alloc_cmd_pool(struct mrsas_instance *instance)
```



```
3162 {
3163         int             i;
3164         int             count;
3165         uint32_t        max_cmd;
3166         uint32_t        reserve_cmd;
3167         size_t          sz;

3169         struct mrsas_cmd        *cmd;

3171         max_cmd = instance->max_fw_cmds;
3172         con_log(CL_ANN1, (CE_NOTE, "mrsas_alloc_cmd_pool: "
3173             "max_cmd %x", max_cmd));


3176         sz = sizeof (struct mrsas_cmd *) * max_cmd;

3178         /*
3179          * instance->cmd_list is an array of struct mrsas_cmd pointers.
3180          * Allocate the dynamic array first and then allocate individual
3181          * commands.
3182          */
3183         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
3184         ASSERT(instance->cmd_list);
3202         if (instance->cmd_list == NULL) {
3203                 con_log(CL_NONE, (CE_WARN,
3204                     "Failed to allocate memory for cmd_list"));
3205                 return (DDI_FAILURE);
3206         }

3186         /* create a frame pool and assign one frame to each cmd */
3187         for (count = 0; count < max_cmd; count++) {
3188                 instance->cmd_list[count] =
3189                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
3190                 ASSERT(instance->cmd_list[count]);
3212                 if (instance->cmd_list[count] == NULL) {
3213                         con_log(CL_NONE, (CE_WARN,
3214                             "Failed to allocate memory for mrsas_cmd"));
3215                         goto mrsas_undo_cmds;
3191         }
3217         }

3193         /* add all the commands to command pool */

3195         INIT_LIST_HEAD(&instance->cmd_pool_list);
3196         INIT_LIST_HEAD(&instance->cmd_pend_list);
3197         INIT_LIST_HEAD(&instance->app_cmd_pool_list);

3199         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

3201         for (i = 0; i < reserve_cmd; i++) {
3202                 cmd = instance->cmd_list[i];
3203                 cmd->index = i;
3204                 mlist_add_tail(&cmd->list, &instance->app_cmd_pool_list);
3205         }


3208         for (i = reserve_cmd; i < max_cmd; i++) {
3209                 cmd = instance->cmd_list[i];
3210                 cmd->index = i;
3211                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
3212         }

3214         return (DDI_SUCCESS);

3216 mrsas_undo_cmds:
3217         if (count > 0) {
```

```
3218                 /* free each cmd */
3219                 for (i = 0; i < count; i++) {
3220                         if (instance->cmd_list[i] != NULL) {
3221                                 kmem_free(instance->cmd_list[i],
3222                                     sizeof (struct mrsas_cmd));
3223                         }
3224                         instance->cmd_list[i] = NULL;
3225                 }
3226         }

3228 mrsas_undo_cmd_list:
3229         if (instance->cmd_list != NULL)
3230                 kmem_free(instance->cmd_list, sz);
3231         instance->cmd_list = NULL;

3233         return (DDI_FAILURE);
3234 }
______unchanged_portion_omitted_

3653 /*
3654  * mrsas_init_adapter - Initialize adapter.
3655  */
3656 int
3657 mrsas_init_adapter(struct mrsas_instance *instance)
3658 {
3659         struct mrsas_ctrl_info          ctrl_info;


3662         /* we expect the FW state to be READY */
3663         if (mfi_state_transition_to_ready(instance)) {
3664                 con_log(CL_ANN, (CE_WARN, "mr_sas: F/W is not ready"));
3665                 return (DDI_FAILURE);
3666         }

3668         /* get various operational parameters from status register */
3669         instance->max_num_sge =
3670             (instance->func_ptr->read_fw_status_reg(instance) &
3671             0xFF0000) >> 0x10;
3672         instance->max_num_sge =
3673             (instance->max_num_sge > MRSAS_MAX_SGE_CNT) ?
3674             MRSAS_MAX_SGE_CNT : instance->max_num_sge;

3676         /*
3677          * Reduce the max supported cmds by 1. This is to ensure that the
3678          * reply_q_sz (1 more than the max cmd that driver may send)
3679          * does not exceed max cmds that the FW can support
3680          */
3681         instance->max_fw_cmds =
3682             instance->func_ptr->read_fw_status_reg(instance) & 0xFFFF;
3683         instance->max_fw_cmds = instance->max_fw_cmds - 1;



3687         /* Initialize adapter */
3688         if (instance->func_ptr->init_adapter(instance) != DDI_SUCCESS) {
3689                 con_log(CL_ANN,
3690                     (CE_WARN, "mr_sas: could not initialize adapter"));
3691                 return (DDI_FAILURE);
3692         }

3694         /* gather misc FW related information */
3695         instance->disable_online_ctrl_reset = 0;

3697         if (!get_ctrl_info(instance, &ctrl_info)) {
3698                 instance->max_sectors_per_req = ctrl_info.max_request_size;
3699                 con_log(CL_ANN1, (CE_NOTE,
```



```
3700                     "product name %s ld present %d",
3701                     ctrl_info.product_name, ctrl_info.ld_present_count));
3702         } else {
3703                 instance->max_sectors_per_req = instance->max_num_sge *
3704                     PAGESIZE / 512;
3705         }

3707         if (ctrl_info.properties.on_off_properties & DISABLE_OCR_PROP_FLAG)
3733         if (ctrl_info.properties.on_off_properties & DISABLE_OCR_PROP_FLAG) {
3708                 instance->disable_online_ctrl_reset = 1;
3735                 con_log(CL_ANN1,
3736                     (CE_NOTE, "Disable online control Flag is set\n"));
3737         } else {
3738                 con_log(CL_ANN1,
3739                     (CE_NOTE, "Disable online control Flag is not set\n"));
3740         }

3710         return (DDI_SUCCESS);

3712 }
______unchanged_portion_omitted_
3807 /*
3808  * mfi_state_transition_to_ready        : Move the FW to READY state
3809  *
3810  * @reg_set                     : MFI register set
3811  */
3812 int
3813 mfi_state_transition_to_ready(struct mrsas_instance *instance)
3814 {
3815         int             i;
3816         uint8_t         max_wait;
3817         uint32_t        fw_ctrl = 0;
3818         uint32_t        fw_state;
3819         uint32_t        cur_state;
3820         uint32_t        cur_abs_reg_val;
3821         uint32_t        prev_abs_reg_val;
3822         uint32_t        status;

3824         cur_abs_reg_val =
3825             instance->func_ptr->read_fw_status_reg(instance);
3826         fw_state =
3827             cur_abs_reg_val & MFI_STATE_MASK;
3828         con_log(CL_ANN1, (CE_CONT,
3829             "mfi_state_transition_to_ready:FW state = 0x%x", fw_state));

3831         while (fw_state != MFI_STATE_READY) {
3832                 con_log(CL_ANN, (CE_CONT,
3833                     "mfi_state_transition_to_ready:FW state%x", fw_state));

3835                 switch (fw_state) {
3836                 case MFI_STATE_FAULT:
3837                         con_log(CL_ANN, (CE_NOTE,
3838                             "mr_sas: FW in FAULT state!!"));

3840                         return (ENODEV);
3841                 case MFI_STATE_WAIT_HANDSHAKE:
3842                         /* set the CLR bit in IMR0 */
3843                         con_log(CL_ANN1, (CE_NOTE,
3844                             "mr_sas: FW waiting for HANDSHAKE"));
3845                         /*
3846                          * PCI_Hot Plug: MFI F/W requires
3847                          * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3848                          * to be set
3849                          */
3850                         /* WR_IB_MSG_0(MFI_INIT_CLEAR_HANDSHAKE, instance); */
3851                         if (!instance->tbolt) {
```

```
3852                                WR_IB_DOORBELL(MFI_INIT_CLEAR_HANDSHAKE |
3853                                    MFI_INIT_HOTPLUG, instance);
3854                        } else {
3855                                WR_RESERVED0_REGISTER(MFI_INIT_CLEAR_HANDSHAKE |
3856                                    MFI_INIT_HOTPLUG, instance);
3857                        }
3858                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3859                        cur_state       = MFI_STATE_WAIT_HANDSHAKE;
3860                        break;
3861                case MFI_STATE_BOOT_MESSAGE_PENDING:
3862                        /* set the CLR bit in IMR0 */
3863                        con_log(CL_ANN1, (CE_NOTE,
3864                            "mr_sas: FW state boot message pending"));
3865                        /*
3866                         * PCI_Hot Plug: MFI F/W requires
3867                         * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3868                         * to be set
3869                         */
3870                        if (!instance->tbolt) {
3871                                WR_IB_DOORBELL(MFI_INIT_HOTPLUG, instance);
3872                        } else {
3873                                WR_RESERVED0_REGISTER(MFI_INIT_HOTPLUG,
3874                                    instance);
3875                        }
3876                        max_wait        = (instance->tbolt == 1) ? 180 : 10;
3877                        cur_state       = MFI_STATE_BOOT_MESSAGE_PENDING;
3878                        break;
3879                case MFI_STATE_OPERATIONAL:
3880                        /* bring it to READY state; assuming max wait 2 secs */
3881                        instance->func_ptr->disable_intr(instance);
3882                        con_log(CL_ANN1, (CE_NOTE,
3883                            "mr_sas: FW in OPERATIONAL state"));
3884                        /*
3885                         * PCI_Hot Plug: MFI F/W requires
3886                         * (MFI_INIT_READY | MFI_INIT_MFIMODE | MFI_INIT_ABORT)
3887                         * to be set
3888                         */
3889                        /* WR_IB_DOORBELL(MFI_INIT_READY, instance); */
3890                        if (!instance->tbolt) {
3891                                WR_IB_DOORBELL(MFI_RESET_FLAGS, instance);
3892                        } else {
3893                                WR_RESERVED0_REGISTER(MFI_RESET_FLAGS,
3894                                    instance);

3896                                for (i = 0; i < (10 * 1000); i++) {
3897                                        status =
3898                                            RD_RESERVED0_REGISTER(instance);
3899                                        if (status & 1) {
3900                                                delay(1 *
3901                                                    drv_usectohz(MILLISEC));
3902                                        } else {
3903                                                break;
3904                                        }
3905                                }

3907                        }
3908                        max_wait        = (instance->tbolt == 1) ? 180 : 10;
3909                        cur_state       = MFI_STATE_OPERATIONAL;
3910                        break;
3911                case MFI_STATE_UNDEFINED:
3912                        /* this state should not last for more than 2 seconds */
3913                        con_log(CL_ANN1, (CE_NOTE, "FW state undefined"));

3915                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3916                        cur_state       = MFI_STATE_UNDEFINED;
3917                        break;
```



```
3918                case MFI_STATE_BB_INIT:
3919                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3920                        cur_state       = MFI_STATE_BB_INIT;
3921                        break;
3922                case MFI_STATE_FW_INIT:
3923                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3924                        cur_state       = MFI_STATE_FW_INIT;
3925                        break;
3926                case MFI_STATE_FW_INIT_2:
3927                        max_wait        = 180;
3928                        cur_state       = MFI_STATE_FW_INIT_2;
3929                        break;
3930                case MFI_STATE_DEVICE_SCAN:
3931                        max_wait        = 180;
3932                        cur_state       = MFI_STATE_DEVICE_SCAN;
3933                        prev_abs_reg_val = cur_abs_reg_val;
3934                        con_log(CL_NONE, (CE_NOTE,
3935                            "Device scan in progress ...\n"));
3936                        break;
3937                case MFI_STATE_FLUSH_CACHE:
3938                        max_wait        = 180;
3939                        cur_state       = MFI_STATE_FLUSH_CACHE;
3940                        break;
3941                default:
3942                        con_log(CL_ANN1, (CE_NOTE,
3943                            "mr_sas: Unknown state 0x%x", fw_state));
3944                        return (ENODEV);
3945                }

3947                /* the cur_state should not last for more than max_wait secs */
3948                for (i = 0; i < (max_wait * MILLISEC); i++) {
3949                        /* fw_state = RD_OB_MSG_0(instance) & MFI_STATE_MASK; */
3950                        cur_abs_reg_val =
3951                            instance->func_ptr->read_fw_status_reg(instance);
3952                        fw_state = cur_abs_reg_val & MFI_STATE_MASK;

3954                        if (fw_state == cur_state) {
3955                                delay(1 * drv_usectohz(MILLISEC));
3956                        } else {
3957                                break;
3958                        }
3959                }
3960                if (fw_state == MFI_STATE_DEVICE_SCAN) {
3961                        if (prev_abs_reg_val != cur_abs_reg_val) {
3962                                continue;
3963                        }
3964                }

3966                /* return error if fw_state hasn't changed after max_wait */
3967                if (fw_state == cur_state) {
3968                        con_log(CL_ANN1, (CE_WARN,
3969                            "FW state hasn't changed in %d secs", max_wait));
3970                        return (ENODEV);
3971                }
3972        };

3974        if (!instance->tbolt) {
3975                fw_ctrl = RD_IB_DOORBELL(instance);
3976                con_log(CL_ANN1, (CE_CONT,
3977                    "mfi_state_transition_to_ready:FW ctrl = 0x%x", fw_ctrl));

3979                /*
3980                 * Write 0xF to the doorbell register to do the following.
3981                 * - Abort all outstanding commands (bit 0).
3982                 * - Transition from OPERATIONAL to READY state (bit 1).
3983                 * - Discard (possible) low MFA posted in 64-bit mode (bit-2).
```

```
3984                     * - Set to release FW to continue running (i.e. BIOS handshake
3985                     *   (bit 3).
3986                     */
3987                    WR_IB_DOORBELL(0xF, instance);
3988            }

3990            if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
3991                    return (EIO);
4023                    return (ENODEV);
3992            }

3994            return (DDI_SUCCESS);
3995    }
______unchanged_portion_omitted_

4843    /*
4844     * mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t)
4845     *
4846     * De-allocate the memory and other resources for an dma object, which must
4847     * have been alloated by a previous call to mrsas_alloc_dma_obj()
4848     */
4881    /* ARGSUSED */
4849    int
4850    mrsas_free_dma_obj(struct mrsas_instance *instance, dma_obj_t obj)
4851    {

4853            if ((obj.dma_handle == NULL) || (obj.acc_handle == NULL)) {
4854                    return (DDI_SUCCESS);
4855            }

4857            /*
4858             * NOTE: These check-handle functions fail if *_handle == NULL, but
4859             * this function succeeds because of the previous check.
4860             */
4861            if (mrsas_check_dma_handle(obj.dma_handle) != DDI_SUCCESS) {
4862                    ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
4863                    return (DDI_FAILURE);
4864            }

4866            if (mrsas_check_acc_handle(obj.acc_handle) != DDI_SUCCESS) {
4867                    ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
4868                    return (DDI_FAILURE);
4869            }

4871            (void) ddi_dma_unbind_handle(obj.dma_handle);
4872            ddi_dma_mem_free(&obj.acc_handle);
4873            ddi_dma_free_handle(&obj.dma_handle);
4874            obj.acc_handle = NULL;
4875            return (DDI_SUCCESS);
4876    }
______unchanged_portion_omitted_
5405    #endif  /* __sparc */

5407    /*
5408     * issue_mfi_pthru
5409     */
5410    static int
5411    issue_mfi_pthru(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5412        struct mrsas_cmd *cmd, int mode)
5413    {
5414            void            *ubuf;
5415            uint32_t        kphys_addr = 0;
5416            uint32_t        xferlen = 0;
5417            uint32_t        new_xfer_length = 0;
5418            uint_t          model;
5419            ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
```



```
5420            dma_obj_t                       pthru_dma_obj;
5421            struct mrsas_pthru_frame        *kpthru;
5422            struct mrsas_pthru_frame        *pthru;
5423            int i;
5424            pthru = &cmd->frame->pthru;
5425            kpthru = (struct mrsas_pthru_frame *)&ioctl->frame[0];

5427            if (instance->adapterresetinprogress) {
5428                    con_log(CL_ANN1, (CE_WARN, "issue_mfi_pthru: Reset flag set, "
5429                    "returning mfi_pkt and setting TRAN_BUSY\n"));
5430                    return (DDI_FAILURE);
5431            }
5432            model = ddi_model_convert_from(mode & FMODELS);
5433            if (model == DDI_MODEL_ILP32) {
5434                    con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP32"));

5436                    xferlen = kpthru->sgl.sge32[0].length;

5438                    ubuf    = (void *)(ulong_t)kpthru->sgl.sge32[0].phys_addr;
5439            } else {
5440    #ifdef _ILP32
5441                    con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP32"));
5442                    xferlen = kpthru->sgl.sge32[0].length;
5443                    ubuf    = (void *)(ulong_t)kpthru->sgl.sge32[0].phys_addr;
5444    #else
5445                    con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP64"));
5446                    xferlen = kpthru->sgl.sge64[0].length;
5447                    ubuf    = (void *)(ulong_t)kpthru->sgl.sge64[0].phys_addr;
5448    #endif
5449            }

5451            if (xferlen) {
5452                    /* means IOCTL requires DMA */
5453                    /* allocate the data transfer buffer */
5454                    /* pthru_dma_obj.size = xferlen; */
5455                    MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
5456                        PAGESIZE);
5457                    pthru_dma_obj.size = new_xfer_length;
5458                    pthru_dma_obj.dma_attr = mrsas_generic_dma_attr;
5459                    pthru_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5460                    pthru_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5461                    pthru_dma_obj.dma_attr.dma_attr_sgllen = 1;
5462                    pthru_dma_obj.dma_attr.dma_attr_align = 1;

5464                    /* allocate kernel buffer for DMA */
5465                    if (mrsas_alloc_dma_obj(instance, &pthru_dma_obj,
5466                        (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5467                            con_log(CL_ANN, (CE_WARN, "issue_mfi_pthru: "
5468                                "could not allocate data transfer buffer."));
5469                            return (DDI_FAILURE);
5470                    }
5471                    (void) memset(pthru_dma_obj.buffer, 0, xferlen);

5473                    /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5474                    if (kpthru->flags & MFI_FRAME_DIR_WRITE) {
5475                            for (i = 0; i < xferlen; i++) {
5476                                    if (ddi_copyin((uint8_t *)ubuf+i,
5477                                        (uint8_t *)pthru_dma_obj.buffer+i,
5478                                        1, mode)) {
5479                                            con_log(CL_ANN, (CE_WARN,
5480                                                "issue_mfi_pthru : "
5481                                                "copy from user space failed"));
5482                                            return (DDI_FAILURE);
5483                                    }
5484                            }
5485                    }
```

```
5487                kphys_addr = pthru_dma_obj.dma_cookie[0].dmac_address;
5488        }

5490        ddi_put8(acc_handle, &pthru->cmd, kpthru->cmd);
5491        ddi_put8(acc_handle, &pthru->sense_len, SENSE_LENGTH);
5492        ddi_put8(acc_handle, &pthru->cmd_status, 0);
5493        ddi_put8(acc_handle, &pthru->scsi_status, 0);
5494        ddi_put8(acc_handle, &pthru->target_id, kpthru->target_id);
5495        ddi_put8(acc_handle, &pthru->lun, kpthru->lun);
5496        ddi_put8(acc_handle, &pthru->cdb_len, kpthru->cdb_len);
5497        ddi_put8(acc_handle, &pthru->sge_count, kpthru->sge_count);
5498        ddi_put16(acc_handle, &pthru->timeout, kpthru->timeout);
5499        ddi_put32(acc_handle, &pthru->data_xfer_len, kpthru->data_xfer_len);

5501        ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
5502        pthru->sense_buf_phys_addr_lo = cmd->sense_phys_addr;
5503        /* ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo, 0); */

5505        ddi_rep_put8(acc_handle, (uint8_t *)kpthru->cdb, (uint8_t *)pthru->cdb,
5506            pthru->cdb_len, DDI_DEV_AUTOINCR);

5508        ddi_put16(acc_handle, &pthru->flags, kpthru->flags & ~MFI_FRAME_SGL64);
5509        ddi_put32(acc_handle, &pthru->sgl.sge32[0].length, xferlen);
5510        ddi_put32(acc_handle, &pthru->sgl.sge32[0].phys_addr, kphys_addr);

5512        cmd->sync_cmd = MRSAS_TRUE;
5513        cmd->frame_count = 1;

5515        if (instance->tbolt) {
5516                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5517        }

5519        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5520                con_log(CL_ANN, (CE_WARN,
5521                    "issue_mfi_pthru: fw_ioctl failed"));
5522        } else {
5523                if (xferlen && kpthru->flags & MFI_FRAME_DIR_READ) {
5524                        for (i = 0; i < xferlen; i++) {
5525                                if (ddi_copyout(
5526                                    (uint8_t *)pthru_dma_obj.buffer+i,
5527                                    (uint8_t *)ubuf+i, 1, mode)) {
5528                                        con_log(CL_ANN, (CE_WARN,
5529                                            "issue_mfi_pthru : "
5530                                            "copy to user space failed"));
5531                                        return (DDI_FAILURE);
5532                                }
5533                        }
5534                }
5535        }

5537        kpthru->cmd_status = ddi_get8(acc_handle, &pthru->cmd_status);
5538        kpthru->scsi_status = ddi_get8(acc_handle, &pthru->scsi_status);

5540        con_log(CL_ANN, (CE_CONT, "issue_mfi_pthru: cmd_status %x, "
5541            "scsi_status %x", kpthru->cmd_status, kpthru->scsi_status));
5542        DTRACE_PROBE3(issue_pthru, uint8_t, kpthru->cmd, uint8_t,
5543            kpthru->cmd_status, uint8_t, kpthru->scsi_status);

5545        if (kpthru->sense_len) {
5546                uint_t sense_len = SENSE_LENGTH;
5547                void *sense_ubuf =
5548                    (void *)(ulong_t)kpthru->sense_buf_phys_addr_lo;
5549                if (kpthru->sense_len <= SENSE_LENGTH) {
5550                        sense_len = kpthru->sense_len;
5551                }
```



```
5553                for (i = 0; i < sense_len; i++) {
5554                        if (ddi_copyout(
5555                            (uint8_t *)cmd->sense+i,
5556                            (uint8_t *)sense_ubuf+i, 1, mode)) {
5557                                con_log(CL_ANN, (CE_WARN,
5558                                    "issue_mfi_pthru : "
5559                                    "copy to user space failed"));
5560                        }
5561                        con_log(CL_DLEVEL1, (CE_WARN,
5562                            "Copying Sense info sense_buff[%d] = 0x%X",
5595                            "Copying Sense info sense_buff[%d] = 0x%X\n",
5563                            i, *((uint8_t *)cmd->sense + i)));
5564                }
5565        }
5566        (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
5567            DDI_DMA_SYNC_FORDEV);

5569        if (xferlen) {
5570                /* free kernel buffer */
5571                if (mrsas_free_dma_obj(instance, pthru_dma_obj) != DDI_SUCCESS)
5572                        return (DDI_FAILURE);
5573        }

5575        return (DDI_SUCCESS);
5576 }

5578 /*
5579  * issue_mfi_dcmd
5580  */
5581 static int
5582 issue_mfi_dcmd(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5583     struct mrsas_cmd *cmd, int mode)
5584 {
5585        void            *ubuf;
5586        uint32_t        kphys_addr = 0;
5587        uint32_t        xferlen = 0;
5588        uint32_t        new_xfer_length = 0;
5589        uint32_t        model;
5590        dma_obj_t       dcmd_dma_obj;
5591        struct mrsas_dcmd_frame *kdcmd;
5592        struct mrsas_dcmd_frame *dcmd;
5593        ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5594        int i;
5595        dcmd = &cmd->frame->dcmd;
5596        kdcmd = (struct mrsas_dcmd_frame *)&ioctl->frame[0];

5598        if (instance->adapterresetinprogress) {
5599                con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
5600                "returning mfi_pkt and setting TRAN_BUSY"));
5632                con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
5633                "returning mfi_pkt and setting TRAN_BUSY\n"));
5601                return (DDI_FAILURE);
5602        }
5603        model = ddi_model_convert_from(mode & FMODELS);
5604        if (model == DDI_MODEL_ILP32) {
5605                con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));

5607                xferlen = kdcmd->sgl.sge32[0].length;

5609                ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
5610        } else {
5611 #ifdef _ILP32
5612                con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));
5613                xferlen = kdcmd->sgl.sge32[0].length;
5614                ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
```

```
#else
                con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_LP64"));
                xferlen = kdcmd->sgl.sge64[0].length;
                ubuf    = (void *)(ulong_t)kdcmd->sgl.sge64[0].phys_addr;
#endif
        }
        if (xferlen) {
                /* means IOCTL requires DMA */
                /* allocate the data transfer buffer */
                /* dcmd_dma_obj.size = xferlen; */
                MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
                    PAGESIZE);
                dcmd_dma_obj.size = new_xfer_length;
                dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
                dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
                dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
                dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
                dcmd_dma_obj.dma_attr.dma_attr_align = 1;

                /* allocate kernel buffer for DMA */
                        if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
                            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
                                con_log(CL_ANN,
                                    (CE_WARN, "issue_mfi_dcmd: could not "
                                    "allocate data transfer buffer."));
                                return (DDI_FAILURE);
                        }
                (void) memset(dcmd_dma_obj.buffer, 0, xferlen);

                /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
                if (kdcmd->flags & MFI_FRAME_DIR_WRITE) {
                        for (i = 0; i < xferlen; i++) {
                                if (ddi_copyin((uint8_t *)ubuf + i,
                                    (uint8_t *)dcmd_dma_obj.buffer + i,
                                    1, mode)) {
                                        con_log(CL_ANN, (CE_WARN,
                                            "issue_mfi_dcmd : "
                                            "copy from user space failed"));
                                        return (DDI_FAILURE);
                                }
                        }
                }

                kphys_addr = dcmd_dma_obj.dma_cookie[0].dmac_address;
        }

        ddi_put8(acc_handle, &dcmd->cmd, kdcmd->cmd);
        ddi_put8(acc_handle, &dcmd->cmd_status, 0);
        ddi_put8(acc_handle, &dcmd->sge_count, kdcmd->sge_count);
        ddi_put16(acc_handle, &dcmd->timeout, kdcmd->timeout);
        ddi_put32(acc_handle, &dcmd->data_xfer_len, kdcmd->data_xfer_len);
        ddi_put32(acc_handle, &dcmd->opcode, kdcmd->opcode);

        ddi_rep_put8(acc_handle, (uint8_t *)kdcmd->mbox.b,
            (uint8_t *)dcmd->mbox.b, DCMD_MBOX_SZ, DDI_DEV_AUTOINCR);

        ddi_put16(acc_handle, &dcmd->flags, kdcmd->flags & ~MFI_FRAME_SGL64);
        ddi_put32(acc_handle, &dcmd->sgl.sge32[0].length, xferlen);
        ddi_put32(acc_handle, &dcmd->sgl.sge32[0].phys_addr, kphys_addr);

        cmd->sync_cmd = MRSAS_TRUE;
        cmd->frame_count = 1;

        if (instance->tbolt) {
                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
        }
```



```
        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
                con_log(CL_ANN, (CE_WARN, "issue_mfi_dcmd: fw_ioctl failed"));
        } else {
                if (xferlen && (kdcmd->flags & MFI_FRAME_DIR_READ)) {
                        for (i = 0; i < xferlen; i++) {
                                if (ddi_copyout(
                                    (uint8_t *)dcmd_dma_obj.buffer + i,
                                    (uint8_t *)ubuf + i,
                                    1, mode)) {
                                        con_log(CL_ANN, (CE_WARN,
                                            "issue_mfi_dcmd : "
                                            "copy to user space failed"));
                                        return (DDI_FAILURE);
                                }
                        }
                }
        }

        kdcmd->cmd_status = ddi_get8(acc_handle, &dcmd->cmd_status);
        con_log(CL_ANN,
            (CE_CONT, "issue_mfi_dcmd: cmd_status %x", kdcmd->cmd_status));
        DTRACE_PROBE3(issue_dcmd, uint32_t, kdcmd->opcode, uint8_t,
            kdcmd->cmd, uint8_t, kdcmd->cmd_status);

        if (xferlen) {
                /* free kernel buffer */
                if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
                        return (DDI_FAILURE);
        }

        return (DDI_SUCCESS);
}
______unchanged_portion_omitted_


static int
mrsas_reset_ppc(struct mrsas_instance *instance)
{
        uint32_t status;
        uint32_t retry = 0;
        uint32_t cur_abs_reg_val;
        uint32_t fw_state;

        con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

        if (instance->deadadapter == 1) {
                cmn_err(CE_WARN, "mrsas_reset_ppc: "
                    "no more resets as HBA has been marked dead ");
                return (DDI_FAILURE);
        }
        mutex_enter(&instance->ocr_flags_mtx);
        instance->adapterresetinprogress = 1;
        mutex_exit(&instance->ocr_flags_mtx);
        con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: adpterresetinprogress "
            "flag set, time %llx", gethrtime()));

        instance->func_ptr->disable_intr(instance);
retry_reset:
        WR_IB_WRITE_SEQ(0, instance);
        WR_IB_WRITE_SEQ(4, instance);
        WR_IB_WRITE_SEQ(0xb, instance);
        WR_IB_WRITE_SEQ(2, instance);
        WR_IB_WRITE_SEQ(7, instance);
        WR_IB_WRITE_SEQ(0xd, instance);
        con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: magic number written "
```

```
7002             "to write sequence register\n"));
7003         delay(100 * drv_usectohz(MILLISEC));
7004         status = RD_OB_DRWE(instance);

7006         while (!(status & DIAG_WRITE_ENABLE)) {
7007                 delay(100 * drv_usectohz(MILLISEC));
7008                 status = RD_OB_DRWE(instance);
7009                 if (retry++ == 100) {
7010                         cmn_err(CE_WARN, "mrsas_reset_ppc: DRWE bit "
7011                             "check retry count %d", retry);
7044                             "check retry count %d\n", retry);
7012                         return (DDI_FAILURE);
7013                 }
7014         }
7015         WR_IB_DRWE(status | DIAG_RESET_ADAPTER, instance);
7016         delay(100 * drv_usectohz(MILLISEC));
7017         status = RD_OB_DRWE(instance);
7018         while (status & DIAG_RESET_ADAPTER) {
7019                 delay(100 * drv_usectohz(MILLISEC));
7020                 status = RD_OB_DRWE(instance);
7021                 if (retry++ == 100) {
7022                         cmn_err(CE_WARN, "mrsas_reset_ppc: "
7023                             "RESET FAILED. KILL adapter called.");
7056                             "RESET FAILED. KILL adapter called\n.");

7025                         (void) mrsas_kill_adapter(instance);
7026                         return (DDI_FAILURE);
7027                 }
7028         }
7029         con_log(CL_ANN, (CE_NOTE, "mrsas_reset_ppc: Adapter reset complete"));
7030         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7031             "Calling mfi_state_transition_to_ready"));

7033         /* Mark HBA as bad, if FW is fault after 3 continuous resets */
7034         if (mfi_state_transition_to_ready(instance) ||
7035             debug_fw_faults_after_ocr_g == 1) {
7036                 cur_abs_reg_val =
7037                     instance->func_ptr->read_fw_status_reg(instance);
7038                 fw_state        = cur_abs_reg_val & MFI_STATE_MASK;

7040 #ifdef OCRDEBUG
7041                 con_log(CL_ANN1, (CE_NOTE,
7042                     "mrsas_reset_ppc :before fake: FW is not ready "
7043                     "FW state = 0x%x", fw_state));
7044                 if (debug_fw_faults_after_ocr_g == 1)
7045                         fw_state = MFI_STATE_FAULT;
7046 #endif

7048                 con_log(CL_ANN1, (CE_NOTE,  "mrsas_reset_ppc : FW is not ready "
7049                     "FW state = 0x%x", fw_state));

7051                 if (fw_state == MFI_STATE_FAULT) {
7052                         /* increment the count */
7053                         instance->fw_fault_count_after_ocr++;
7054                         if (instance->fw_fault_count_after_ocr
7055                             < MAX_FW_RESET_COUNT) {
7056                                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
7057                                     "FW is in fault after OCR count %d "
7058                                     "Retry Reset",
7059                                     instance->fw_fault_count_after_ocr);
7060                                 goto retry_reset;

7062                         } else {
7063                                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
7064                                     "Max Reset Count exceeded >%d"
7065                                     "Mark HBA as bad, KILL adapter",
```



```
7066                                     MAX_FW_RESET_COUNT);

7068                                 (void) mrsas_kill_adapter(instance);
7069                                 return (DDI_FAILURE);
7070                         }
7071                 }
7072         }
7073         /* reset the counter as FW is up after OCR */
7074         instance->fw_fault_count_after_ocr = 0;


7077         ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
7078             instance->producer, 0);

7080         ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
7081             instance->consumer, 0);

7083         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7084             " after resetting produconsumer chck indexs:"
7085             "producer %x consumer %x", *instance->producer,
7086             *instance->consumer));

7088         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7089             "Calling mrsas_issue_init_mfi"));
7090         (void) mrsas_issue_init_mfi(instance);
7091         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7092             "mrsas_issue_init_mfi Done"));

7094         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7095             "Calling mrsas_print_pending_cmd\n"));
7096         (void) mrsas_print_pending_cmds(instance);
7097         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7098             "mrsas_print_pending_cmd done\n"));

7100         instance->func_ptr->enable_intr(instance);
7101         instance->fw_outstanding = 0;

7103         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7104             "Calling mrsas_issue_pending_cmds"));
7105         (void) mrsas_issue_pending_cmds(instance);
7106         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7107             "issue_pending_cmds done.\n"));

7109         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7110             "Calling aen registration"));


7113         instance->aen_cmd->retry_count_for_ocr = 0;
7114         instance->aen_cmd->drv_pkt_time = 0;

7116         instance->func_ptr->issue_cmd(instance->aen_cmd, instance);
7117         con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag.\n"));

7119         mutex_enter(&instance->ocr_flags_mtx);
7120         instance->adapterresetinprogress = 0;
7121         mutex_exit(&instance->ocr_flags_mtx);
7122         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7123             "adpterresetinprogress flag unset"));

7125         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc done\n"));
7126         return (DDI_SUCCESS);
7127 }
```

_____***unchanged_portion_omitted_***

```
7306 static int
7307 mrsas_add_intrs(struct mrsas_instance *instance, int intr_type)
```

```
7308 {

7310         dev_info_t *dip = instance->dip;
7311         int     avail, actual, count;
7312         int     i, flag, ret;

7314         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: intr_type = %x",
7315             intr_type));

7317         /* Get number of interrupts */
7318         ret = ddi_intr_get_nintrs(dip, intr_type, &count);
7319         if ((ret != DDI_SUCCESS) || (count == 0)) {
7320                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_nintrs() failed:"
7321                     "ret %d count %d", ret, count));

7323                 return (DDI_FAILURE);
7324         }

7326         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: count = %d ", count));

7328         /* Get number of available interrupts */
7329         ret = ddi_intr_get_navail(dip, intr_type, &avail);
7330         if ((ret != DDI_SUCCESS) || (avail == 0)) {
7331                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_navail() failed:"
7332                     "ret %d avail %d", ret, avail));

7334                 return (DDI_FAILURE);
7335         }
7336         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: avail = %d ", avail));

7338         /* Only one interrupt routine. So limit the count to 1 */
7339         if (count > 1) {
7340                 count = 1;
7341         }

7343         /*
7344          * Allocate an array of interrupt handlers. Currently we support
7345          * only one interrupt. The framework can be extended later.
7346          */
7347         instance->intr_htable_size = count * sizeof (ddi_intr_handle_t);
7348         instance->intr_htable = kmem_zalloc(instance->intr_htable_size,
7349             KM_SLEEP);
7350         ASSERT(instance->intr_htable);
7383         if (instance->intr_htable == NULL) {
7384                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7385                     "failed to allocate memory for intr-handle table"));
7386                 instance->intr_htable_size = 0;
7387                 return (DDI_FAILURE);
7388         }

7352         flag = ((intr_type == DDI_INTR_TYPE_MSI) ||
7353             (intr_type == DDI_INTR_TYPE_MSIX)) ?
7354             DDI_INTR_ALLOC_STRICT : DDI_INTR_ALLOC_NORMAL;

7356         /* Allocate interrupt */
7357         ret = ddi_intr_alloc(dip, instance->intr_htable, intr_type, 0,
7358             count, &actual, flag);

7360         if ((ret != DDI_SUCCESS) || (actual == 0)) {
7361                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7362                     "avail = %d", avail));
7363                 goto mrsas_free_htable;
7364         }

7366         if (actual < count) {
7367                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
```



```
7368                     "Requested = %d  Received = %d", count, actual));
7369         }
7370         instance->intr_cnt = actual;

7372         /*
7373          * Get the priority of the interrupt allocated.
7374          */
7375         if ((ret = ddi_intr_get_pri(instance->intr_htable[0],
7376             &instance->intr_pri)) != DDI_SUCCESS) {
7377                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7378                     "get priority call failed"));
7379                 goto mrsas_free_handles;
7380         }

7382         /*
7383          * Test for high level mutex. we don't support them.
7384          */
7385         if (instance->intr_pri >= ddi_intr_get_hilevel_pri()) {
7386                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7387                     "High level interrupts not supported."));
7388                 goto mrsas_free_handles;
7389         }

7391         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: intr_pri = 0x%x ",
7392             instance->intr_pri));

7394         /* Call ddi_intr_add_handler() */
7395         for (i = 0; i < actual; i++) {
7396                 ret = ddi_intr_add_handler(instance->intr_htable[i],
7397                     (ddi_intr_handler_t *)mrsas_isr, (caddr_t)instance,
7398                     (caddr_t)(uintptr_t)i);

7400                 if (ret != DDI_SUCCESS) {
7401                         con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs:"
7402                             "failed %d", ret));
7403                         goto mrsas_free_handles;
7404                 }

7406         }

7408         con_log(CL_DLEVEL1, (CE_NOTE, " ddi_intr_add_handler done"));

7410         if ((ret = ddi_intr_get_cap(instance->intr_htable[0],
7411             &instance->intr_cap)) != DDI_SUCCESS) {
7412                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_cap() failed %d",
7413                     ret));
7414                 goto mrsas_free_handlers;
7415         }

7417         if (instance->intr_cap &  DDI_INTR_FLAG_BLOCK) {
7418                 con_log(CL_ANN, (CE_WARN, "Calling ddi_intr_block _enable"));

7420                 (void) ddi_intr_block_enable(instance->intr_htable,
7421                     instance->intr_cnt);
7422         } else {
7423                 con_log(CL_ANN, (CE_NOTE, " calling ddi_intr_enable"));

7425                 for (i = 0; i < instance->intr_cnt; i++) {
7426                         (void) ddi_intr_enable(instance->intr_htable[i]);
7427                         con_log(CL_ANN, (CE_NOTE, "ddi intr enable returns "
7428                             "%d", i));
7429                 }
7430         }

7432         return (DDI_SUCCESS);
```

```
7434 mrsas_free_handlers:
7435         for (i = 0; i < actual; i++)
7436                 (void) ddi_intr_remove_handler(instance->intr_htable[i]);

7438 mrsas_free_handles:
7439         for (i = 0; i < actual; i++)
7440                 (void) ddi_intr_free(instance->intr_htable[i]);

7442 mrsas_free_htable:
7443         if (instance->intr_htable != NULL)
7444                 kmem_free(instance->intr_htable, instance->intr_htable_size);

7446         instance->intr_htable = NULL;
7447         instance->intr_htable_size = 0;

7449         return (DDI_FAILURE);

7451 }
______unchanged_portion_omitted_

7622 static int
7623 mrsas_config_ld(struct mrsas_instance *instance, uint16_t tgt,
7624     uint8_t lun, dev_info_t **ldip)
7625 {
7626         struct scsi_device *sd;
7627         dev_info_t *child;
7628         int rval;

7630         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_config_ld: t = %d l = %d",
7631             tgt, lun));

7633         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
7634                 if (ldip) {
7635                         *ldip = child;
7636                 }
7637                 if (instance->mr_ld_list[tgt].flag != MRDRV_TGT_VALID) {
7638                         rval = mrsas_service_evt(instance, tgt, 0,
7639                             MRSAS_EVT_UNCONFIG_TGT, NULL);
7640                         con_log(CL_ANN1, (CE_WARN,
7641                             "mr_sas: DELETING STALE ENTRY rval = %d "
7642                             "tgt id = %d ", rval, tgt));
7643                         return (NDI_FAILURE);
7644                 }
7645                 return (NDI_SUCCESS);
7646         }

7648         sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
7687         if (sd == NULL) {
7688                 con_log(CL_ANN1, (CE_WARN, "mrsas_config_ld: "
7689                     "failed to allocate mem for scsi_device"));
7690                 return (NDI_FAILURE);
7691         }
7649         sd->sd_address.a_hba_tran = instance->tran;
7650         sd->sd_address.a_target = (uint16_t)tgt;
7651         sd->sd_address.a_lun = (uint8_t)lun;

7653         if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS)
7654                 rval = mrsas_config_scsi_device(instance, sd, ldip);
7655         else
7656                 rval = NDI_FAILURE;

7658         /* sd_unprobe is blank now. Free buffer manually */
7659         if (sd->sd_inq) {
7660                 kmem_free(sd->sd_inq, SUN_INQSIZE);
7661                 sd->sd_inq = (struct scsi_inquiry *)NULL;
7662         }
```



```
7664         kmem_free(sd, sizeof (struct scsi_device));
7665         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_config_ld: return rval = %d",
7666             rval));
7667         return (rval);
7668 }
______unchanged_portion_omitted_
```

```
**********************************************************
  106990 Tue Nov  6 14:28:30 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_


/*
 * ThunderBolt(TB) Request Message Frame Pool
 */
int
create_mpi2_frame_pool(struct mrsas_instance *instance)
{
        int             i = 0;
        uint16_t        max_cmd;
        uint32_t        sgl_sz;
        uint32_t        raid_msg_size;
        uint32_t        total_size;
        uint32_t        offset;
        uint32_t        io_req_base_phys;
        uint8_t         *io_req_base;
        struct mrsas_cmd        *cmd;

        max_cmd = instance->max_fw_cmds;

        sgl_sz          = 1024;
        raid_msg_size   = MRSAS_THUNDERBOLT_MSG_SIZE;

        /* Allocating additional 256 bytes to accomodate SMID 0. */
        total_size = MRSAS_THUNDERBOLT_MSG_SIZE + (max_cmd * raid_msg_size) +
            (max_cmd * sgl_sz) + (max_cmd * SENSE_LENGTH);

        con_log(CL_ANN1, (CE_NOTE, "create_mpi2_frame_pool: "
            "max_cmd %x", max_cmd));
            "max_cmd %x ", max_cmd));

        con_log(CL_DLEVEL3, (CE_NOTE, "create_mpi2_frame_pool: "
            "request message frame pool size %x", total_size));

        /*
         * ThunderBolt(TB) We need to create a single chunk of DMA'ble memory
         * and then split the memory to 1024 commands. Each command should be
         * able to contain a RAID MESSAGE FRAME which will embed a MFI_FRAME
         * within it. Further refer the "alloc_req_rep_desc" function where
         * we allocate request/reply descriptors queues for a clue.
         */

        instance->mpi2_frame_pool_dma_obj.size = total_size;
        instance->mpi2_frame_pool_dma_obj.dma_attr = mrsas_generic_dma_attr;
        instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_addr_hi =
            0xFFFFFFFFU;
        instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_count_max =
            0xFFFFFFFFU;
        instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_sgllen = 1;
        instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_align = 256;

        if (mrsas_alloc_dma_obj(instance, &instance->mpi2_frame_pool_dma_obj,
            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
                cmn_err(CE_WARN,
                    "mr_sas: could not alloc mpi2 frame pool");
                return (DDI_FAILURE);
        }

        bzero(instance->mpi2_frame_pool_dma_obj.buffer, total_size);
        instance->mpi2_frame_pool_dma_obj.status |= DMA_OBJ_ALLOCATED;
```



```
        instance->io_request_frames =
            (uint8_t *)instance->mpi2_frame_pool_dma_obj.buffer;
        instance->io_request_frames_phy =
            (uint32_t)
            instance->mpi2_frame_pool_dma_obj.dma_cookie[0].dmac_address;

        con_log(CL_DLEVEL3, (CE_NOTE, "io_request_frames 0x%p",
            (void *)instance->io_request_frames));

        con_log(CL_DLEVEL3, (CE_NOTE, "io_request_frames_phy 0x%x",
            instance->io_request_frames_phy));

        io_req_base = (uint8_t *)instance->io_request_frames +
            MRSAS_THUNDERBOLT_MSG_SIZE;
        io_req_base_phys = instance->io_request_frames_phy +
            MRSAS_THUNDERBOLT_MSG_SIZE;

        con_log(CL_DLEVEL3, (CE_NOTE,
            "io req_base_phys 0x%x", io_req_base_phys));

        for (i = 0; i < max_cmd; i++) {
                cmd = instance->cmd_list[i];

                offset = i * MRSAS_THUNDERBOLT_MSG_SIZE;

                cmd->scsi_io_request = (Mpi2RaidSCSIIORequest_t *)
                    ((uint8_t *)io_req_base + offset);
                cmd->scsi_io_request_phys_addr = io_req_base_phys + offset;

                cmd->sgl = (Mpi2SGEIOUnion_t *)((uint8_t *)io_req_base +
                    (max_cmd * raid_msg_size) + i * sgl_sz);

                cmd->sgl_phys_addr = (io_req_base_phys +
                    (max_cmd * raid_msg_size) + i * sgl_sz);

                cmd->sense1 = (uint8_t *)((uint8_t *)io_req_base +
                    (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
                    (i * SENSE_LENGTH));

                cmd->sense_phys_addr1 = (io_req_base_phys +
                    (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
                    (i * SENSE_LENGTH));


                cmd->SMID = i + 1;

                con_log(CL_DLEVEL3, (CE_NOTE, "Frame Pool Addr [%x]0x%p",
                    cmd->index, (void *)cmd->scsi_io_request));

                con_log(CL_DLEVEL3, (CE_NOTE, "Frame Pool Phys Addr [%x]0x%x",
                    cmd->index, cmd->scsi_io_request_phys_addr));

                con_log(CL_DLEVEL3, (CE_NOTE, "Sense Addr [%x]0x%p",
                    cmd->index, (void *)cmd->sense1));

                con_log(CL_DLEVEL3, (CE_NOTE, "Sense Addr Phys [%x]0x%x",
                    cmd->index, cmd->sense_phys_addr1));

                con_log(CL_DLEVEL3, (CE_NOTE, "Sgl bufffers [%x]0x%p",
                    cmd->index, (void *)cmd->sgl));

                con_log(CL_DLEVEL3, (CE_NOTE, "Sgl bufffers phys [%x]0x%x",
                    cmd->index, cmd->sgl_phys_addr));
        }
```

```
285         return (DDI_SUCCESS);

287 }


290 /*
291  * alloc_additional_dma_buffer for AEN
292  */
293 int
294 mrsas_tbolt_alloc_additional_dma_buffer(struct mrsas_instance *instance)
295 {
296         uint32_t        internal_buf_size = PAGESIZE*2;
297         int i;

299         /* Initialize buffer status as free */
300         instance->mfi_internal_dma_obj.status = DMA_OBJ_FREED;
301         instance->mfi_evt_detail_obj.status = DMA_OBJ_FREED;
302         instance->ld_map_obj[0].status = DMA_OBJ_FREED;
303         instance->ld_map_obj[1].status = DMA_OBJ_FREED;


306         instance->mfi_internal_dma_obj.size = internal_buf_size;
307         instance->mfi_internal_dma_obj.dma_attr = mrsas_generic_dma_attr;
308         instance->mfi_internal_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
309         instance->mfi_internal_dma_obj.dma_attr.dma_attr_count_max =
310             0xFFFFFFFFU;
311         instance->mfi_internal_dma_obj.dma_attr.dma_attr_sgllen = 1;

313         if (mrsas_alloc_dma_obj(instance, &instance->mfi_internal_dma_obj,
314             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
315                 cmn_err(CE_WARN,
316                     "mr_sas: could not alloc reply queue");
317                 return (DDI_FAILURE);
318         }

320         bzero(instance->mfi_internal_dma_obj.buffer, internal_buf_size);

322         instance->mfi_internal_dma_obj.status |= DMA_OBJ_ALLOCATED;
323         instance->internal_buf =
324             (caddr_t)(((unsigned long)instance->mfi_internal_dma_obj.buffer));
325         instance->internal_buf_dmac_add =
326             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address;
327         instance->internal_buf_size = internal_buf_size;

329         /* allocate evt_detail */
330         instance->mfi_evt_detail_obj.size = sizeof (struct mrsas_evt_detail);
331         instance->mfi_evt_detail_obj.dma_attr = mrsas_generic_dma_attr;
332         instance->mfi_evt_detail_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
333         instance->mfi_evt_detail_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
334         instance->mfi_evt_detail_obj.dma_attr.dma_attr_sgllen = 1;
335         instance->mfi_evt_detail_obj.dma_attr.dma_attr_align = 8;

337         if (mrsas_alloc_dma_obj(instance, &instance->mfi_evt_detail_obj,
338             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
339                 cmn_err(CE_WARN, "mrsas_tbolt_alloc_additional_dma_buffer: "
340                     "could not allocate data transfer buffer.");
341                 goto fail_tbolt_additional_buff;
342         }

344         bzero(instance->mfi_evt_detail_obj.buffer,
345             sizeof (struct mrsas_evt_detail));

347         instance->mfi_evt_detail_obj.status |= DMA_OBJ_ALLOCATED;

349         instance->size_map_info = sizeof (MR_FW_RAID_MAP) +
350             (sizeof (MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));
```



```
352         for (i = 0; i < 2; i++) {
353                 /* allocate the data transfer buffer */
354                 instance->ld_map_obj[i].size = instance->size_map_info;
355                 instance->ld_map_obj[i].dma_attr = mrsas_generic_dma_attr;
356                 instance->ld_map_obj[i].dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
357                 instance->ld_map_obj[i].dma_attr.dma_attr_count_max =
358                     0xFFFFFFFFU;
359                 instance->ld_map_obj[i].dma_attr.dma_attr_sgllen = 1;
360                 instance->ld_map_obj[i].dma_attr.dma_attr_align = 1;

362                 if (mrsas_alloc_dma_obj(instance, &instance->ld_map_obj[i],
363                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
364                         cmn_err(CE_WARN,
365                             "could not allocate data transfer buffer.");
366                         goto fail_tbolt_additional_buff;
367                 }

369                 instance->ld_map_obj[i].status |= DMA_OBJ_ALLOCATED;

371                 bzero(instance->ld_map_obj[i].buffer, instance->size_map_info);
371                 (void) memset(instance->ld_map_obj[i].buffer, 0,
372                     instance->size_map_info);

373                 instance->ld_map[i] =
374                     (MR_FW_RAID_MAP_ALL *)instance->ld_map_obj[i].buffer;
375                 instance->ld_map_phy[i] = (uint32_t)instance->
376                     ld_map_obj[i].dma_cookie[0].dmac_address;

378                 con_log(CL_DLEVEL3, (CE_NOTE,
379                     "ld_map Addr Phys 0x%x", instance->ld_map_phy[i]));

381                 con_log(CL_DLEVEL3, (CE_NOTE,
382                     "size_map_info 0x%x", instance->size_map_info));
383         }

385         return (DDI_SUCCESS);

387 fail_tbolt_additional_buff:
388         mrsas_tbolt_free_additional_dma_buffer(instance);

390         return (DDI_FAILURE);
391 }

393 MRSAS_REQUEST_DESCRIPTOR_UNION *
394 mr_sas_get_request_descriptor(struct mrsas_instance *instance, uint16_t index)
395 {
396         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc;

398         if (index > instance->max_fw_cmds) {
399                 con_log(CL_ANN1, (CE_NOTE,
400                     "Invalid SMID 0x%x request for descriptor", index));
401                 con_log(CL_ANN1, (CE_NOTE,
402                     "max_fw_cmds : 0x%x", instance->max_fw_cmds));
403                     "max_fw_cmds : 0x%x\n", instance->max_fw_cmds));
403                 return (NULL);
404         }

406         req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
407             ((char *)instance->request_message_pool +
408             (sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION) * index));

410         con_log(CL_ANN1, (CE_NOTE,
411             "request descriptor : 0x%08lx", (unsigned long)req_desc));
412             "request descriptor : 0x%08lx\n", (unsigned long)req_desc));
```

```
413         con_log(CL_ANN1, (CE_NOTE,
414             "request descriptor base phy : 0x%08lx",
415             "request descriptor base phy : 0x%08lx\n",
415             (unsigned long)instance->request_message_pool_phy));

417         return ((MRSAS_REQUEST_DESCRIPTOR_UNION *)req_desc);
418 }


421 /*
422  * Allocate Request and Reply  Queue Descriptors.
423  */
424 int
425 alloc_req_rep_desc(struct mrsas_instance *instance)
426 {
427         uint32_t        request_q_sz, reply_q_sz;
428         int             i, max_reply_q_sz;
429         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;

431         /*
432          * ThunderBolt(TB) There's no longer producer consumer mechanism.
433          * Once we have an interrupt we are supposed to scan through the list of
434          * reply descriptors and process them accordingly. We would be needing
435          * to allocate memory for 1024 reply descriptors
436          */

438         /* Allocate Reply Descriptors */
439         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x",
440         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x\n",
440             (uint_t)sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));

442         /* reply queue size should be multiple of 16 */
443         max_reply_q_sz = ((instance->max_fw_cmds + 1 + 15)/16)*16;

445         reply_q_sz = 8 * max_reply_q_sz;


448         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x",
449         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x\n",
449             (uint_t)sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));

451         instance->reply_desc_dma_obj.size = reply_q_sz;
452         instance->reply_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
453         instance->reply_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
454         instance->reply_desc_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
455         instance->reply_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
456         instance->reply_desc_dma_obj.dma_attr.dma_attr_align = 16;

458         if (mrsas_alloc_dma_obj(instance, &instance->reply_desc_dma_obj,
459             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
460                 cmn_err(CE_WARN,
461                     "mr_sas: could not alloc reply queue");
462                 return (DDI_FAILURE);
463         }

465         bzero(instance->reply_desc_dma_obj.buffer, reply_q_sz);
466         instance->reply_desc_dma_obj.status |= DMA_OBJ_ALLOCATED;

468         /* virtual address of  reply queue */
469         instance->reply_frame_pool = (MPI2_REPLY_DESCRIPTORS_UNION *)(
470             instance->reply_desc_dma_obj.buffer);

472         instance->reply_q_depth = max_reply_q_sz;

474         con_log(CL_ANN1, (CE_NOTE, "[reply queue depth]0x%x",
475             instance->reply_q_depth));
```



```
477         con_log(CL_ANN1, (CE_NOTE, "[reply queue virt addr]0x%p",
478             (void *)instance->reply_frame_pool));

480         /* initializing reply address to 0xFFFFFFFF */
481         reply_desc = instance->reply_frame_pool;

483         for (i = 0; i < instance->reply_q_depth; i++) {
484                 reply_desc->Words = (uint64_t)~0;
485                 reply_desc++;
486         }


489         instance->reply_frame_pool_phy =
490             (uint32_t)instance->reply_desc_dma_obj.dma_cookie[0].dmac_address;

492         con_log(CL_ANN1, (CE_NOTE,
493             "[reply queue phys addr]0x%x", instance->reply_frame_pool_phy));


496         instance->reply_pool_limit_phy = (instance->reply_frame_pool_phy +
497             reply_q_sz);

499         con_log(CL_ANN1, (CE_NOTE, "[reply pool limit phys addr]0x%x",
500             instance->reply_pool_limit_phy));


503         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x",
504         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x\n",
504             (int)sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));

506         /* Allocate Request Descriptors */
507         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x",
508         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x\n",
508             (int)sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));

510         request_q_sz = 8 *
511             (instance->max_fw_cmds);

513         instance->request_desc_dma_obj.size = request_q_sz;
514         instance->request_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
515         instance->request_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
516         instance->request_desc_dma_obj.dma_attr.dma_attr_count_max =
517             0xFFFFFFFFU;
518         instance->request_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
519         instance->request_desc_dma_obj.dma_attr.dma_attr_align = 16;

521         if (mrsas_alloc_dma_obj(instance, &instance->request_desc_dma_obj,
522             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
523                 cmn_err(CE_WARN,
524                     "mr_sas: could not alloc request queue desc");
525                 goto fail_undo_reply_queue;
526         }

528         bzero(instance->request_desc_dma_obj.buffer, request_q_sz);
529         instance->request_desc_dma_obj.status |= DMA_OBJ_ALLOCATED;

531         /* virtual address of  request queue desc */
532         instance->request_message_pool = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
533             (instance->request_desc_dma_obj.buffer);

535         instance->request_message_pool_phy =
536             (uint32_t)instance->request_desc_dma_obj.dma_cookie[0].dmac_address;

538         return (DDI_SUCCESS);
```

```
fail_undo_reply_queue:
        if (instance->reply_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->reply_desc_dma_obj);
                instance->reply_desc_dma_obj.status = DMA_OBJ_FREED;
        }

        return (DDI_FAILURE);
}

/*
 * mrsas_alloc_cmd_pool_tbolt
 *
 * TODO: merge tbolt-specific code into mrsas_alloc_cmd_pool() to have single
 * routine
 */
int
mrsas_alloc_cmd_pool_tbolt(struct mrsas_instance *instance)
{
        int             i;
        int             count;
        uint32_t        max_cmd;
        uint32_t        reserve_cmd;
        size_t          sz;

        struct mrsas_cmd        *cmd;

        max_cmd = instance->max_fw_cmds;
        con_log(CL_ANN1, (CE_NOTE, "mrsas_alloc_cmd_pool: "
            "max_cmd %x", max_cmd));


        sz = sizeof (struct mrsas_cmd *) * max_cmd;

        /*
         * instance->cmd_list is an array of struct mrsas_cmd pointers.
         * Allocate the dynamic array first and then allocate individual
         * commands.
         */
        instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
        if (instance->cmd_list == NULL) {
                con_log(CL_NONE, (CE_WARN,
                    "Failed to allocate memory for cmd_list"));
                return (DDI_FAILURE);
        }

        /* create a frame pool and assign one frame to each cmd */
        for (count = 0; count < max_cmd; count++) {
                instance->cmd_list[count] =
                    kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
                if (instance->cmd_list[count] == NULL) {
                        con_log(CL_NONE, (CE_WARN,
                            "Failed to allocate memory for mrsas_cmd"));
                        goto mrsas_undo_cmds;
        }
        }

        /* add all the commands to command pool */

        INIT_LIST_HEAD(&instance->cmd_pool_list);
        INIT_LIST_HEAD(&instance->cmd_pend_list);
        INIT_LIST_HEAD(&instance->cmd_app_pool_list);

        reserve_cmd = MRSAS_APP_RESERVED_CMDS;

        /* cmd index 0 reservered for IOC INIT */
```



```
        for (i = 1; i < reserve_cmd; i++) {
                cmd             = instance->cmd_list[i];
                cmd->index      = i;
                mlist_add_tail(&cmd->list, &instance->cmd_app_pool_list);
        }


        for (i = reserve_cmd; i < max_cmd; i++) {
                cmd             = instance->cmd_list[i];
                cmd->index      = i;
                mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
        }

        return (DDI_SUCCESS);

mrsas_undo_cmds:
        if (count > 0) {
                /* free each cmd */
                for (i = 0; i < count; i++) {
                        if (instance->cmd_list[i] != NULL) {
                                kmem_free(instance->cmd_list[i],
                                    sizeof (struct mrsas_cmd));
                        }
                        instance->cmd_list[i] = NULL;
                }
        }

mrsas_undo_cmd_list:
        if (instance->cmd_list != NULL)
                kmem_free(instance->cmd_list, sz);
        instance->cmd_list = NULL;

        return (DDI_FAILURE);
}
_____unchanged_portion_omitted_


/*
 * mrsas_init_adapter_tbolt - Initialize fusion interface adapter.
 */
int
mrsas_init_adapter_tbolt(struct mrsas_instance *instance)
{

        /*
         * Reduce the max supported cmds by 1. This is to ensure that the
         * reply_q_sz (1 more than the max cmd that driver may send)
         * does not exceed max cmds that the FW can support
         */

        if (instance->max_fw_cmds > 1008) {
                instance->max_fw_cmds = 1008;
                instance->max_fw_cmds = instance->max_fw_cmds-1;
        }

        con_log(CL_ANN, (CE_NOTE, "mrsas_init_adapter_tbolt: "
            " instance->max_fw_cmds 0x%X.", instance->max_fw_cmds));


        /* create a pool of commands */
        if (alloc_space_for_mpi2(instance) != DDI_SUCCESS) {
                cmn_err(CE_WARN,
                    " alloc_space_for_mpi2() failed.");

                return (DDI_FAILURE);
        }
```

```
793         /* Send ioc init message */
794         /* NOTE: the issue_init call does FMA checking already. */
795         if (mrsas_issue_init_mpi2(instance) != DDI_SUCCESS) {
796                 cmn_err(CE_WARN,
797                     " mrsas_issue_init_mpi2() failed.");
798
799                 goto fail_init_fusion;
800         }
801
802         instance->unroll.alloc_space_mpi2 = 1;
803
804         con_log(CL_ANN, (CE_NOTE,
805             "mrsas_init_adapter_tbolt: SUCCESSFUL"));
806             "mrsas_init_adapter_tbolt: SUCCESSFULL\n"));
807         return (DDI_SUCCESS);
808
809 fail_init_fusion:
810         free_space_for_mpi2(instance);
811
812         return (DDI_FAILURE);
813 }
814
815
816
817 /*
818  * init_mpi2
819  */
820 int
821 mrsas_issue_init_mpi2(struct mrsas_instance *instance)
822 {
823         dma_obj_t init2_dma_obj;
824         int ret_val = DDI_SUCCESS;
825
826         /* allocate DMA buffer for IOC INIT message */
827         init2_dma_obj.size = sizeof (Mpi2IOCInitRequest_t);
828         init2_dma_obj.dma_attr = mrsas_generic_dma_attr;
829         init2_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
830         init2_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
831         init2_dma_obj.dma_attr.dma_attr_sgllen = 1;
832         init2_dma_obj.dma_attr.dma_attr_align = 256;
833
834         if (mrsas_alloc_dma_obj(instance, &init2_dma_obj,
835             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
836                 cmn_err(CE_WARN, "mr_sas_issue_init_mpi2 "
837                     "could not allocate data transfer buffer.");
838                 return (DDI_FAILURE);
839         }
840         (void) memset(init2_dma_obj.buffer, 2, sizeof (Mpi2IOCInitRequest_t));
841         (void) memset(init2_dma_obj.buffer, 2,
842             sizeof (Mpi2IOCInitRequest_t));
843         con_log(CL_ANN1, (CE_NOTE,
844             "mrsas_issue_init_mpi2 _phys adr: %x",
845             "mrsas_issue_init_mpi2 _phys adr: %x \n",
846             init2_dma_obj.dma_cookie[0].dmac_address));
847
848         /* Initialize and send ioc init message */
849         ret_val = mrsas_tbolt_ioc_init(instance, &init2_dma_obj);
850         if (ret_val == DDI_FAILURE) {
851                 con_log(CL_ANN1, (CE_WARN,
852                     "mrsas_issue_init_mpi2: Failed"));
863                     "mrsas_issue_init_mpi2: Failed\n"));
852                 goto fail_init_mpi2;
```



```
853         }
854
855         /* free IOC init DMA buffer */
856         if (mrsas_free_dma_obj(instance, init2_dma_obj)
857             != DDI_SUCCESS) {
858                 con_log(CL_ANN1, (CE_WARN,
859                     "mrsas_issue_init_mpi2: Free Failed"));
871                     "mrsas_issue_init_mpi2: Free Failed\n"));
860                 return (DDI_FAILURE);
861         }
862
863         /* Get/Check and sync ld_map info */
864         instance->map_id = 0;
865         if (mrsas_tbolt_check_map_info(instance) == DDI_SUCCESS)
866                 (void) mrsas_tbolt_sync_map_info(instance);
867
868
869         /* No mrsas_cmd to send, so send NULL. */
870         if (mrsas_common_check(instance, NULL) != DDI_SUCCESS)
871                 goto fail_init_mpi2;
872
873         con_log(CL_ANN, (CE_NOTE,
874             "mrsas_issue_init_mpi2: SUCCESSFUL"));
886             "mrsas_issue_init_mpi2: SUCCESSFULL\n"));
875
876         return (DDI_SUCCESS);
877
878 fail_init_mpi2:
879         (void) mrsas_free_dma_obj(instance, init2_dma_obj);
880
881         return (DDI_FAILURE);
882 }
883
884 static int
885 mrsas_tbolt_ioc_init(struct mrsas_instance *instance, dma_obj_t *mpi2_dma_obj)
886 {
887         int                             numbytes;
888         uint16_t                        flags;
889         struct mrsas_init_frame2        *mfiFrameInit2;
890         struct mrsas_header             *frame_hdr;
891         Mpi2IOCInitRequest_t            *init;
892         struct mrsas_cmd                *cmd = NULL;
893         struct mrsas_drv_ver            drv_ver_info;
894         MRSAS_REQUEST_DESCRIPTOR_UNION  *req_desc;
895
896         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
897
898
899 #ifdef DEBUG
900         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
901             (int)sizeof (*mfiFrameInit2)));
902         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", (int)sizeof (*init)));
903         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
904             (int)sizeof (struct mrsas_init_frame2)));
905         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n",
906             (int)sizeof (Mpi2IOCInitRequest_t)));
907 #endif
908
909         init = (Mpi2IOCInitRequest_t *)mpi2_dma_obj->buffer;
910         numbytes = sizeof (*init);
911         bzero(init, numbytes);
912
913         ddi_put8(mpi2_dma_obj->acc_handle, &init->Function,
914             MPI2_FUNCTION_IOC_INIT);
915
916         ddi_put8(mpi2_dma_obj->acc_handle, &init->WhoInit,
```

```
917             MPI2_WHOINIT_HOST_DRIVER);
919         /* set MsgVersion and HeaderVersion host driver was built with */
920         ddi_put16(mpi2_dma_obj->acc_handle, &init->MsgVersion,
921             MPI2_VERSION);
923         ddi_put16(mpi2_dma_obj->acc_handle, &init->HeaderVersion,
924             MPI2_HEADER_VERSION);
926         ddi_put16(mpi2_dma_obj->acc_handle, &init->SystemRequestFrameSize,
927             instance->raid_io_msg_size / 4);
929         ddi_put16(mpi2_dma_obj->acc_handle, &init->ReplyFreeQueueDepth,
930             0);
932         ddi_put16(mpi2_dma_obj->acc_handle,
933             &init->ReplyDescriptorPostQueueDepth,
934             instance->reply_q_depth);
935         /*
936          * These addresses are set using the DMA cookie addresses from when the
937          * memory was allocated.  Sense buffer hi address should be 0.
938          * ddi_put32(accessp, &init->SenseBufferAddressHigh, 0);
939          */
941         ddi_put32(mpi2_dma_obj->acc_handle,
942             &init->SenseBufferAddressHigh, 0);
944         ddi_put64(mpi2_dma_obj->acc_handle,
945             (uint64_t *)&init->SystemRequestFrameBaseAddress,
946             instance->io_request_frames_phy);
948         ddi_put64(mpi2_dma_obj->acc_handle,
949             &init->ReplyDescriptorPostQueueAddress,
950             instance->reply_frame_pool_phy);
952         ddi_put64(mpi2_dma_obj->acc_handle,
953             &init->ReplyFreeQueueAddress, 0);
955         cmd = instance->cmd_list[0];
956         if (cmd == NULL) {
957                 return (DDI_FAILURE);
958         }
959         cmd->retry_count_for_ocr = 0;
960         cmd->pkt = NULL;
961         cmd->drv_pkt_time = 0;
963         mfiFrameInit2 = (struct mrsas_init_frame2 *)cmd->scsi_io_request;
964         con_log(CL_ANN1, (CE_CONT, "[mfi vaddr]%p", (void *)mfiFrameInit2));
966         frame_hdr = &cmd->frame->hdr;
968         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
969             MFI_CMD_STATUS_POLL_MODE);
971         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
973         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;
975         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);
977         con_log(CL_ANN, (CE_CONT,
978             "mrsas_tbolt_ioc_init: SMID:%x\n", cmd->SMID));
980         /* Init the MFI Header */
981         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
982             &mfiFrameInit2->cmd, MFI_CMD_OP_INIT);
```



```
984         con_log(CL_ANN1, (CE_CONT, "[CMD]%x", mfiFrameInit2->cmd));
986         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
987             &mfiFrameInit2->cmd_status,
988             MFI_STAT_INVALID_STATUS);
990         con_log(CL_ANN1, (CE_CONT, "[Status]%x", mfiFrameInit2->cmd_status));
992         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
993             &mfiFrameInit2->queue_info_new_phys_addr_lo,
994             mpi2_dma_obj->dma_cookie[0].dmac_address);
996         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
997             &mfiFrameInit2->data_xfer_len,
998             sizeof (Mpi2IOCInitRequest_t));
1000        con_log(CL_ANN1, (CE_CONT, "[reply q desc addr]%x",
1001            (int)init->ReplyDescriptorPostQueueAddress));
1003        /* fill driver version information */
1004        fill_up_drv_ver(&drv_ver_info);
1006        /* allocate the driver version data transfer buffer */
1007        instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
1008        instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
1009        instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
1010        instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
1011        instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
1012        instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;
1014        if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
1015            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
1016                cmn_err(CE_WARN,
1017                    "fusion init: Could not allocate driver version buffer.");
1018                return (DDI_FAILURE);
1019        }
1020        /* copy driver version to dma buffer */
1021        bzero(instance->drv_ver_dma_obj.buffer, sizeof (drv_ver_info.drv_ver));
1033        (void) memset(instance->drv_ver_dma_obj.buffer, 0,
1034            sizeof (drv_ver_info.drv_ver));
1022        ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
1023            (uint8_t *)drv_ver_info.drv_ver,
1024            (uint8_t *)instance->drv_ver_dma_obj.buffer,
1025            sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);
1027        /* send driver version physical address to firmware */
1028        ddi_put64(cmd->frame_dma_obj.acc_handle, &mfiFrameInit2->driverversion,
1029            instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);
1031        con_log(CL_ANN1, (CE_CONT, "[MPIINIT2 frame Phys addr ]0x%x len = %x",
1032            mfiFrameInit2->queue_info_new_phys_addr_lo,
1033            (int)sizeof (Mpi2IOCInitRequest_t)));
1035        con_log(CL_ANN1, (CE_CONT, "[Length]%x", mfiFrameInit2->data_xfer_len));
1037        con_log(CL_ANN1, (CE_CONT, "[MFI frame Phys Address]%x len = %x",
1038            cmd->scsi_io_request_phys_addr,
1039            (int)sizeof (struct mrsas_init_frame2)));
1041        /* disable interrupts before sending INIT2 frame */
1042        instance->func_ptr->disable_intr(instance);
1044        req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
1045            instance->request_message_pool;
1046        req_desc->Words = cmd->scsi_io_request_phys_addr;
```

```
1047         req_desc->MFAIo.RequestFlags =
1048             (MPI2_REQ_DESCRIPT_FLAGS_MFA << MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1050         cmd->request_desc = req_desc;

1052         /* issue the init frame */
1053         instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);

1055         con_log(CL_ANN1, (CE_CONT, "[cmd = %d] ", frame_hdr->cmd));
1056         con_log(CL_ANN1, (CE_CONT, "[cmd  Status= %x] ",
1057             frame_hdr->cmd_status));

1059         if (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1060             &mfiFrameInit2->cmd_status) == 0) {
1061                 con_log(CL_ANN, (CE_NOTE, "INIT2 Success"));
1062         } else {
1063                 con_log(CL_ANN, (CE_WARN, "INIT2 Fail"));
1064                 mrsas_dump_reply_desc(instance);
1065                 goto fail_ioc_init;
1066         }

1068         mrsas_dump_reply_desc(instance);

1070         instance->unroll.verBuff = 1;

1072         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_ioc_init: SUCCESSFUL"));
1085         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_ioc_init: SUCCESSFULL\n"));

1074         return (DDI_SUCCESS);


1077 fail_ioc_init:

1079         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

1081         return (DDI_FAILURE);
1082 }
_____unchanged_portion_omitted_
1106 /*
1107  * scsi_pkt handling
1108  *
1109  * Visible to the external world via the transport structure.
1110  */

1112 int
1113 mrsas_tbolt_tran_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1114 {
1115         struct mrsas_instance   *instance = ADDR2MR(ap);
1116         struct scsa_cmd         *acmd = PKT2CMD(pkt);
1117         struct mrsas_cmd        *cmd = NULL;
1118         uchar_t                 cmd_done = 0;

1120         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1121         if (instance->deadadapter == 1) {
1122                 cmn_err(CE_WARN,
1123                     "mrsas_tran_start:TBOLT return TRAN_FATAL_ERROR "
1124                     "for IO, as the HBA doesnt take any more IOs");
1125                 if (pkt) {
1126                         pkt->pkt_reason         = CMD_DEV_GONE;
1127                         pkt->pkt_statistics     = STAT_DISCON;
1128                 }
1129                 return (TRAN_FATAL_ERROR);
1130         }
1131         if (instance->adapterresetinprogress) {
1132                 con_log(CL_ANN, (CE_NOTE, "Reset flag set, "
1133                     "returning mfi_pkt and setting TRAN_BUSY\n"));
```



```
1134                 return (TRAN_BUSY);
1135         }
1136         (void) mrsas_tbolt_prepare_pkt(acmd);

1138         cmd = mrsas_tbolt_build_cmd(instance, ap, pkt, &cmd_done);

1140         /*
1141          * Check if the command is already completed by the mrsas_build_cmd()
1142          * routine. In which case the busy_flag would be clear and scb will be
1143          * NULL and appropriate reason provided in pkt_reason field
1144          */
1145         if (cmd_done) {
1146                 pkt->pkt_reason = CMD_CMPLT;
1147                 pkt->pkt_scbp[0] = STATUS_GOOD;
1148                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1149                     | STATE_SENT_CMD;
1150                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1151                         (*pkt->pkt_comp)(pkt);
1152                 }

1154                 return (TRAN_ACCEPT);
1155         }

1157         if (cmd == NULL) {
1158                 return (TRAN_BUSY);
1159         }


1162         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1163                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1164                         cmn_err(CE_WARN,
1165                             "Command Queue Full... Returning BUSY");
1178                             "Command Queue Full... Returning BUSY \n");
1166                         return_raid_msg_pkt(instance, cmd);
1167                         return (TRAN_BUSY);
1168                 }

1170                 /* Synchronize the Cmd frame for the controller */
1171                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1172                     DDI_DMA_SYNC_FORDEV);

1174                 con_log(CL_ANN, (CE_CONT, "tbolt_issue_cmd: SCSI CDB[0]=0x%x "
1175                     "cmd->index:0x%x SMID 0x%x\n", pkt->pkt_cdbp[0],
1176                     cmd->index, cmd->SMID));

1178                 instance->func_ptr->issue_cmd(cmd, instance);
1179         } else {
1180                 instance->func_ptr->issue_cmd(cmd, instance);
1181                 (void) wait_for_outstanding_poll_io(instance);
1182                 (void) mrsas_common_check(instance, cmd);
1183         }

1185         return (TRAN_ACCEPT);
1186 }
_____unchanged_portion_omitted_


1405 /*
1406  * build_cmd
1407  */
1408 static struct mrsas_cmd *
1409 mrsas_tbolt_build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
1410     struct scsi_pkt *pkt, uchar_t *cmd_done)
1411 {
1412         uint8_t         fp_possible = 0;
1413         uint32_t        index;
```

```
1414        uint32_t        lba_count = 0;
1415        uint32_t        start_lba_hi = 0;
1416        uint32_t        start_lba_lo = 0;
1417        ddi_acc_handle_t acc_handle =
1418            instance->mpi2_frame_pool_dma_obj.acc_handle;
1419        struct mrsas_cmd                *cmd = NULL;
1420        struct scsa_cmd                 *acmd = PKT2CMD(pkt);
1421        MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
1422        Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
1423        uint32_t                        datalen;
1424        struct IO_REQUEST_INFO io_info;
1425        MR_FW_RAID_MAP_ALL *local_map_ptr;
1426        uint16_t pd_cmd_cdblen;

1428        con_log(CL_DLEVEL1, (CE_NOTE,
1429            "chkpnt: Entered mrsas_tbolt_build_cmd:%d", __LINE__));

1431        /* find out if this is logical or physical drive command.  */
1432        acmd->islogical = MRDRV_IS_LOGICAL(ap);
1433        acmd->device_id = MAP_DEVICE_ID(instance, ap);

1435        *cmd_done = 0;

1437        /* get the command packet */
1438        if (!(cmd = get_raid_msg_pkt(instance))) {
1439                return (NULL);
1440        }

1442        index = cmd->index;
1443        ReqDescUnion =  mr_sas_get_request_descriptor(instance, index);
1444        ReqDescUnion->Words = 0;
1445        ReqDescUnion->SCSIIO.SMID = cmd->SMID;
1446        ReqDescUnion->SCSIIO.RequestFlags =
1447            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1448            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);


1451        cmd->request_desc = ReqDescUnion;
1452        cmd->pkt = pkt;
1453        cmd->cmd = acmd;

1455        /* lets get the command directions */
1456        if (acmd->cmd_flags & CFLAG_DMASEND) {
1457                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1458                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1459                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1460                            DDI_DMA_SYNC_FORDEV);
1461                }
1462        } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
1463                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1464                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1465                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1466                            DDI_DMA_SYNC_FORCPU);
1467                }
1468        } else {
1469                con_log(CL_ANN, (CE_NOTE, "NO DMA"));
1482                con_log(CL_ANN, (CE_NOTE, "NO DMA\n"));
1470        }


1473        /* get SCSI_IO raid message frame pointer */
1474        scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

1476        /* zero out SCSI_IO raid message frame */
1477        bzero(scsi_raid_io, sizeof (Mpi2RaidSCSIIORequest_t));
1490        (void) memset(scsi_raid_io, 0, sizeof (Mpi2RaidSCSIIORequest_t));
```



```
1479        /* Set the ldTargetId set by BuildRaidContext() */
1480        ddi_put16(acc_handle, &scsi_raid_io->RaidContext.ldTargetId,
1481            acmd->device_id);

1483        /*  Copy CDB to scsi_io_request message frame */
1484        ddi_rep_put8(acc_handle,
1485            (uint8_t *)pkt->pkt_cdbp, (uint8_t *)scsi_raid_io->CDB.CDB32,
1486            acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

1488        /*
1489         * Just the CDB length, rest of the Flags are zero
1490         * This will be modified later.
1491         */
1492        ddi_put16(acc_handle, &scsi_raid_io->IoFlags, acmd->cmd_cdblen);

1494        pd_cmd_cdblen = acmd->cmd_cdblen;

1496        switch (pkt->pkt_cdbp[0]) {
1497        case SCMD_READ:
1498        case SCMD_WRITE:
1499        case SCMD_READ_G1:
1500        case SCMD_WRITE_G1:
1501        case SCMD_READ_G4:
1502        case SCMD_WRITE_G4:
1503        case SCMD_READ_G5:
1504        case SCMD_WRITE_G5:

1506                if (acmd->islogical) {
1507                        /* Initialize sense Information */
1508                        if (cmd->sense1 == NULL) {
1509                                con_log(CL_ANN, (CE_NOTE, "tbolt_build_cmd: "
1510                                    "Sense buffer ptr NULL "));
1523                                    "Sense buffer ptr NULL \n"));
1511                        }
1512                        bzero(cmd->sense1, SENSE_LENGTH);
1513                        con_log(CL_DLEVEL2, (CE_NOTE, "tbolt_build_cmd "
1514                            "CDB[0] = %x\n", pkt->pkt_cdbp[0]));

1516                        if (acmd->cmd_cdblen == CDB_GROUP0) {
1517                                /* 6-byte cdb */
1518                                lba_count = (uint16_t)(pkt->pkt_cdbp[4]);
1519                                start_lba_lo = ((uint32_t)(pkt->pkt_cdbp[3]) |
1520                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
1521                                    ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
1522                                    << 16));
1523                        } else if (acmd->cmd_cdblen == CDB_GROUP1) {
1524                                /* 10-byte cdb */
1525                                lba_count =
1526                                    (((uint16_t)(pkt->pkt_cdbp[8])) |
1527                                    ((uint16_t)(pkt->pkt_cdbp[7]) << 8));

1529                                start_lba_lo =
1530                                    (((uint32_t)(pkt->pkt_cdbp[5])) |
1531                                    ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1532                                    ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1533                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1535                        } else if (acmd->cmd_cdblen == CDB_GROUP5) {
1536                                /* 12-byte cdb */
1537                                lba_count = (
1538                                    ((uint32_t)(pkt->pkt_cdbp[9])) |
1539                                    ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1540                                    ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1541                                    ((uint32_t)(pkt->pkt_cdbp[6]) << 24));
```

```
1543                                    start_lba_lo =
1544                                        (((uint32_t)(pkt->pkt_cdbp[5])) |
1545                                        ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1546                                        ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1547                                        ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1549                            } else if (acmd->cmd_cdblen == CDB_GROUP4) {
1550                                    /* 16-byte cdb */
1551                                    lba_count = (
1552                                        ((uint32_t)(pkt->pkt_cdbp[13])) |
1553                                        ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
1554                                        ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
1555                                        ((uint32_t)(pkt->pkt_cdbp[10]) << 24));

1557                                    start_lba_lo = (
1558                                        ((uint32_t)(pkt->pkt_cdbp[9])) |
1559                                        ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1560                                        ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1561                                        ((uint32_t)(pkt->pkt_cdbp[6]) << 24));

1563                                    start_lba_hi = (
1564                                        ((uint32_t)(pkt->pkt_cdbp[5])) |
1565                                        ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1566                                        ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1567                                        ((uint32_t)(pkt->pkt_cdbp[2]) << 24));
1568                            }

1570                            if (instance->tbolt &&
1571                                ((lba_count * 512) > mrsas_tbolt_max_cap_maxxfer)) {
1572                                    cmn_err(CE_WARN, " IO SECTOR COUNT exceeds "
1573                                        "controller limit 0x%x sectors",
1586                                        "controller limit 0x%x sectors\n",
1574                                        lba_count);
1575                            }

1577                            bzero(&io_info, sizeof (struct IO_REQUEST_INFO));
1590                            (void) memset(&io_info, 0,
1591                                sizeof (struct IO_REQUEST_INFO));
1578                            io_info.ldStartBlock = ((uint64_t)start_lba_hi << 32) |
1579                                start_lba_lo;
1580                            io_info.numBlocks = lba_count;
1581                            io_info.ldTgtId = acmd->device_id;

1583                            if (acmd->cmd_flags & CFLAG_DMASEND)
1584                                    io_info.isRead = 0;
1585                            else
1586                                    io_info.isRead = 1;


1589                            /* Acquire SYNC MAP UPDATE lock */
1590                            mutex_enter(&instance->sync_map_mtx);

1592                            local_map_ptr =
1593                                instance->ld_map[(instance->map_id & 1)];

1595                            if ((MR_TargetIdToLdGet(
1596                                acmd->device_id, local_map_ptr) >=
1597                                MAX_LOGICAL_DRIVES) || !instance->fast_path_io) {
1598                                    cmn_err(CE_NOTE, "Fast Path NOT Possible, "
1599                                        "targetId >= MAX_LOGICAL_DRIVES || "
1600                                        "!instance->fast_path_io");
1614                                        "!instance->fast_path_io\n");
1601                                    fp_possible = 0;
1602                                    /* Set Regionlock flags to BYPASS */
1603                                    /* io_request->RaidContext.regLockFlags  = 0; */
1604                                    ddi_put8(acc_handle,
```



```
1605                                        &scsi_raid_io->RaidContext.regLockFlags, 0);
1606                                } else {
1607                                        if (MR_BuildRaidContext(instance, &io_info,
1608                                            &scsi_raid_io->RaidContext, local_map_ptr))
1609                                                fp_possible = io_info.fpOkForIo;
1610                                }

1612                                if (!enable_fp)
1613                                        fp_possible = 0;

1615                                con_log(CL_ANN1, (CE_NOTE, "enable_fp %d  "
1616                                    "instance->fast_path_io %d fp_possible %d",
1630                                    "instance->fast_path_io %d fp_possible %d \n",
1617                                    enable_fp, instance->fast_path_io, fp_possible));

1619                        if (fp_possible) {

1621                                /* Check for DIF enabled LD */
1622                                if (MR_CheckDIF(acmd->device_id, local_map_ptr)) {
1623                                        /* Prepare 32 Byte CDB for DIF capable Disk */
1624                                        mrsas_tbolt_prepare_cdb(instance,
1625                                            scsi_raid_io->CDB.CDB32,
1626                                            &io_info, scsi_raid_io, start_lba_lo);
1627                                } else {
1628                                        mrsas_tbolt_set_pd_lba(scsi_raid_io->CDB.CDB32,
1629                                            (uint8_t *)&pd_cmd_cdblen,
1630                                            io_info.pdBlock, io_info.numBlocks);
1631                                        ddi_put16(acc_handle,
1632                                            &scsi_raid_io->IoFlags, pd_cmd_cdblen);
1633                                }

1635                                ddi_put8(acc_handle, &scsi_raid_io->Function,
1636                                    MPI2_FUNCTION_SCSI_IO_REQUEST);

1638                                ReqDescUnion->SCSIIO.RequestFlags =
1639                                    (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
1640                                    MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1642                                if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1643                                        uint8_t regLockFlags = ddi_get8(acc_handle,
1644                                            &scsi_raid_io->RaidContext.regLockFlags);
1645                                        uint16_t IoFlags = ddi_get16(acc_handle,
1646                                            &scsi_raid_io->IoFlags);

1648                                        if (regLockFlags == REGION_TYPE_UNUSED)
1649                                                ReqDescUnion->SCSIIO.RequestFlags =
1650                                                    (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1651                                                    MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1653                                        IoFlags |=
1654                                            MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH;
1655                                        regLockFlags |=
1656                                            (MR_RL_FLAGS_GRANT_DESTINATION_CUDA |
1657                                            MR_RL_FLAGS_SEQ_NUM_ENABLE);

1659                                        ddi_put8(acc_handle,
1660                                            &scsi_raid_io->ChainOffset, 0);
1661                                        ddi_put8(acc_handle,
1662                                            &scsi_raid_io->RaidContext.nsegType,
1663                                            ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1664                                            MPI2_TYPE_CUDA));
1665                                        ddi_put8(acc_handle,
1666                                            &scsi_raid_io->RaidContext.regLockFlags,
1667                                            regLockFlags);
1668                                        ddi_put16(acc_handle,
1669                                            &scsi_raid_io->IoFlags, IoFlags);
```

```
1670                        }

1672                        if ((instance->load_balance_info[
1673                            acmd->device_id].loadBalanceFlag) &&
1674                            (io_info.isRead)) {
1675                                io_info.devHandle =
1676                                    get_updated_dev_handle(&instance->
1677                                    load_balance_info[acmd->device_id],
1678                                    &io_info);
1679                                cmd->load_balance_flag |=
1680                                    MEGASAS_LOAD_BALANCE_FLAG;
1681                        } else {
1682                                cmd->load_balance_flag &=
1683                                    ~MEGASAS_LOAD_BALANCE_FLAG;
1684                        }

1686                        ReqDescUnion->SCSIIO.DevHandle = io_info.devHandle;
1687                        ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1688                            io_info.devHandle);

1690                } else {
1691                        ddi_put8(acc_handle, &scsi_raid_io->Function,
1692                            MPI2_FUNCTION_LD_IO_REQUEST);

1694                        ddi_put16(acc_handle,
1695                            &scsi_raid_io->DevHandle, acmd->device_id);

1697                        ReqDescUnion->SCSIIO.RequestFlags =
1698                            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1699                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1701                        ddi_put16(acc_handle,
1702                            &scsi_raid_io->RaidContext.timeoutValue,
1703                            local_map_ptr->raidMap.fpPdIoTimeoutSec);

1705                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1706                                uint8_t regLockFlags = ddi_get8(acc_handle,
1707                                    &scsi_raid_io->RaidContext.regLockFlags);

1709                                if (regLockFlags == REGION_TYPE_UNUSED) {
1710                                     ReqDescUnion->SCSIIO.RequestFlags =
1711                                         (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1712                                         MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1713                                }

1715                                regLockFlags |=
1716                                    (MR_RL_FLAGS_GRANT_DESTINATION_CPU0 |
1717                                    MR_RL_FLAGS_SEQ_NUM_ENABLE);

1719                                ddi_put8(acc_handle,
1720                                    &scsi_raid_io->RaidContext.nsegType,
1721                                    ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1722                                    MPI2_TYPE_CUDA));
1723                                ddi_put8(acc_handle,
1724                                    &scsi_raid_io->RaidContext.regLockFlags,
1725                                    regLockFlags);
1726                        }
1727                } /* Not FP */

1729                /* Release SYNC MAP UPDATE lock */
1730                mutex_exit(&instance->sync_map_mtx);


1733                /*
1734                 * Set sense buffer physical address/length in scsi_io_request.
1735                 */
```



```
1736                ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
1737                    cmd->sense_phys_addr1);
1738                ddi_put8(acc_handle, &scsi_raid_io->SenseBufferLength,
1739                    SENSE_LENGTH);

1741                /* Construct SGL */
1742                ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1743                    offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1745                (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1746                    scsi_raid_io, &datalen);

1748                ddi_put32(acc_handle, &scsi_raid_io->DataLength, datalen);

1750                break;
1751 #ifndef PDSUPPORT       /* if PDSUPPORT, skip break and fall through */
1752        } else {
1753                break;
1754 #endif
1755        }
1756        /* fall through For all non-rd/wr cmds */
1757        default:
1758                switch (pkt->pkt_cdbp[0]) {
1759                case 0x35: { /* SCMD_SYNCHRONIZE_CACHE */
1760                        return_raid_msg_pkt(instance, cmd);
1761                        *cmd_done = 1;
1762                        return (NULL);
1763                }

1765                case SCMD_MODE_SENSE:
1766                case SCMD_MODE_SENSE_G1: {
1767                        union scsi_cdb  *cdbp;
1768                        uint16_t        page_code;

1770                        cdbp = (void *)pkt->pkt_cdbp;
1771                        page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
1772                        switch (page_code) {
1773                        case 0x3:
1774                        case 0x4:
1775                                (void) mrsas_mode_sense_build(pkt);
1776                                return_raid_msg_pkt(instance, cmd);
1777                                *cmd_done = 1;
1778                                return (NULL);
1779                        }
1780                        break;
1781                }

1783                default: {
1784                        /*
1785                         * Here we need to handle PASSTHRU for
1786                         * Logical Devices. Like Inquiry etc.
1787                         */

1789                        if (!(acmd->islogical)) {

1791                                /* Acquire SYNC MAP UPDATE lock */
1792                                mutex_enter(&instance->sync_map_mtx);

1794                                local_map_ptr =
1795                                    instance->ld_map[(instance->map_id & 1)];

1797                                ddi_put8(acc_handle, &scsi_raid_io->Function,
1798                                    MPI2_FUNCTION_SCSI_IO_REQUEST);

1800                                ReqDescUnion->SCSIIO.RequestFlags =
1801                                    (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
```

```
1802                                    MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1804                                ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1805                                    local_map_ptr->raidMap.
1806                                    devHndlInfo[acmd->device_id].curDevHdl);


1809                                /* Set regLockFlasgs to REGION_TYPE_BYPASS */
1810                                ddi_put8(acc_handle,
1811                                    &scsi_raid_io->RaidContext.regLockFlags, 0);
1812                                ddi_put64(acc_handle,
1813                                    &scsi_raid_io->RaidContext.regLockRowLBA,
1814                                    0);
1815                                ddi_put32(acc_handle,
1816                                    &scsi_raid_io->RaidContext.regLockLength,
1817                                    0);
1818                                ddi_put8(acc_handle,
1819                                    &scsi_raid_io->RaidContext.RAIDFlags,
1820                                    MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD <<
1821                                    MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT);
1822                                ddi_put16(acc_handle,
1823                                    &scsi_raid_io->RaidContext.timeoutValue,
1824                                    local_map_ptr->raidMap.fpPdIoTimeoutSec);
1825                                ddi_put16(acc_handle,
1826                                    &scsi_raid_io->RaidContext.ldTargetId,
1827                                    acmd->device_id);
1828                                ddi_put8(acc_handle,
1829                                    &scsi_raid_io->LUN[1], acmd->lun);

1831                                /* Release SYNC MAP UPDATE lock */
1832                                mutex_exit(&instance->sync_map_mtx);

1834                            } else {
1835                                    ddi_put8(acc_handle, &scsi_raid_io->Function,
1836                                        MPI2_FUNCTION_LD_IO_REQUEST);
1837                                    ddi_put8(acc_handle,
1838                                        &scsi_raid_io->LUN[1], acmd->lun);
1839                                    ddi_put16(acc_handle,
1840                                        &scsi_raid_io->DevHandle, acmd->device_id);
1841                                    ReqDescUnion->SCSIIO.RequestFlags =
1842                                        (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
1843                                        MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1844                            }

1846                            /*
1847                             * Set sense buffer physical address/length in
1848                             * scsi_io_request.
1849                             */
1850                            ddi_put32(acc_handle,
1851                                &scsi_raid_io->SenseBufferLowAddress,
1852                                cmd->sense_phys_addr1);
1853                            ddi_put8(acc_handle,
1854                                &scsi_raid_io->SenseBufferLength, SENSE_LENGTH);

1856                            /* Construct SGL */
1857                            ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1858                                offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1860                            (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1861                                scsi_raid_io, &datalen);

1863                            ddi_put32(acc_handle,
1864                                &scsi_raid_io->DataLength, datalen);


1867                            con_log(CL_ANN, (CE_CONT,
```



```
1868                                    "tbolt_build_cmd CDB[0] =%x, TargetID =%x\n",
1869                                    pkt->pkt_cdbp[0], acmd->device_id));
1870                                con_log(CL_DLEVEL1, (CE_CONT,
1871                                    "data length = %x\n",
1872                                    scsi_raid_io->DataLength));
1873                                con_log(CL_DLEVEL1, (CE_CONT,
1874                                    "cdb length = %x\n",
1875                                    acmd->cmd_cdblen));
1876                        }
1877                                break;
1878                        }

1880                }

1882                return (cmd);
1883        }
______unchanged_portion_omitted_

2028 /*
2029  * issue_cmd_in_sync_mode
2030  */
2031 int
2032 tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *instance,
2033     struct mrsas_cmd *cmd)
2034 {
2035         int             i;
2036         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2037         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

2039         struct mrsas_header     *hdr;
2040         hdr = (struct mrsas_header *)&cmd->frame->hdr;

2042         con_log(CL_ANN,
2043             (CE_NOTE, "tbolt_issue_cmd_in_sync_mode: cmd->[SMID]=0x%X",
2044             cmd->SMID));


2047         if (instance->adapterresetinprogress) {
2048                 cmd->drv_pkt_time = ddi_get16
2049                     (cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2050                 if (cmd->drv_pkt_time < debug_timeout_g)
2051                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2052                 con_log(CL_ANN, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode:"
2053                     "RESET-IN-PROGRESS, issue cmd & return."));
2067                     "RESET-IN-PROGRESS, issue cmd & return.\n"));

2055                 mutex_enter(&instance->reg_write_mtx);
2056                 WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2057                 WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2058                 mutex_exit(&instance->reg_write_mtx);

2060                 return (DDI_SUCCESS);
2061         } else {
2062                 con_log(CL_ANN1, (CE_NOTE,
2063                     "tbolt_issue_cmd_in_sync_mode: pushing the pkt"));
2077                     "tbolt_issue_cmd_in_sync_mode: pushing the pkt\n"));
2064                 push_pending_mfi_pkt(instance, cmd);
2065         }

2067         con_log(CL_DLEVEL2, (CE_NOTE,
2068             "HighQport offset :%p",
2069             (void *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT)));
2070         con_log(CL_DLEVEL2, (CE_NOTE,
2071             "LowQport offset :%p",
2072             (void *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT)));
```

```
2074         cmd->sync_cmd = MRSAS_TRUE;
2075         cmd->cmd_status =  ENODATA;


2078         mutex_enter(&instance->reg_write_mtx);
2079         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2080         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2081         mutex_exit(&instance->reg_write_mtx);

2083         con_log(CL_ANN1, (CE_NOTE,
2084             " req desc high part %x", (uint_t)(req_desc->Words >> 32)));
2085         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x",
2098             " req desc high part %x \n", (uint_t)(req_desc->Words >> 32)));
2099         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x \n",
2086             (uint_t)(req_desc->Words & 0xffffffff)));

2088         mutex_enter(&instance->int_cmd_mtx);
2089         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
2090                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
2091         }
2092         mutex_exit(&instance->int_cmd_mtx);


2095         if (i < (msecs -1)) {
2096                 return (DDI_SUCCESS);
2097         } else {
2098                 return (DDI_FAILURE);
2099         }
2100 }

2102 /*
2103  * issue_cmd_in_poll_mode
2104  */
2105 int
2106 tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *instance,
2107     struct mrsas_cmd *cmd)
2108 {
2109         int             i;
2110         uint16_t        flags;
2111         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2112         struct mrsas_header *frame_hdr;

2114         con_log(CL_ANN,
2115             (CE_NOTE, "tbolt_issue_cmd_in_poll_mode: cmd->[SMID]=0x%X",
2116             cmd->SMID));

2118         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

2120         frame_hdr = (struct mrsas_header *)&cmd->frame->hdr;
2121         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
2122             MFI_CMD_STATUS_POLL_MODE);
2123         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
2124         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;
2125         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

2127         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x",
2141         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x \n",
2128             (uint_t)(req_desc->Words & 0xffffffff)));
2129         con_log(CL_ANN1, (CE_NOTE,
2130             " req desc high part %x", (uint_t)(req_desc->Words >> 32)));
2144             " req desc high part %x \n", (uint_t)(req_desc->Words >> 32)));

2132         /* issue the frame using inbound queue port */
2133         mutex_enter(&instance->reg_write_mtx);
2134         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2135         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
```



```
2136         mutex_exit(&instance->reg_write_mtx);

2138         for (i = 0; i < msecs && (
2139             ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
2140             == MFI_CMD_STATUS_POLL_MODE); i++) {
2141                 /* wait for cmd_status to change from 0xFF */
2142                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
2143         }

2145         if (ddi_get8(cmd->frame_dma_obj.acc_handle,
2146             &frame_hdr->cmd_status) == MFI_CMD_STATUS_POLL_MODE) {
2147                 con_log(CL_ANN1, (CE_NOTE,
2148                     " cmd failed %" PRIx64, (req_desc->Words)));
2162                     " cmd failed %" PRIx64 " \n", (req_desc->Words)));
2149                 return (DDI_FAILURE);
2150         }

2152         return (DDI_SUCCESS);
2153 }
______unchanged_portion_omitted_


2182 int
2183 tbolt_intr_ack(struct mrsas_instance *instance)
2184 {
2185         uint32_t        status;

2187         /* check if it is our interrupt */
2188         status = RD_OB_INTR_STATUS(instance);
2189         con_log(CL_ANN1, (CE_NOTE,
2190             "chkpnt: Entered tbolt_intr_ack status = %d", status));
2204             "chkpnt: Entered tbolt_intr_ack status = %d \n", status));

2192         if (!(status & MFI_FUSION_ENABLE_INTERRUPT_MASK)) {
2193                 return (DDI_INTR_UNCLAIMED);
2194         }

2196         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
2197                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2198                 return (DDI_INTR_UNCLAIMED);
2199         }

2201         if ((status & 1) || (status & MFI_FUSION_ENABLE_INTERRUPT_MASK)) {
2202                 /* clear the interrupt by writing back the same value */
2203                 WR_OB_INTR_STATUS(status, instance);
2204                 /* dummy READ */
2205                 (void) RD_OB_INTR_STATUS(instance);
2206         }
2207         return (DDI_INTR_CLAIMED);
2208 }
______unchanged_portion_omitted_


2306 void
2307 mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *instance,
2308     struct mrsas_cmd *cmd)
2309 {
2310         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2311         Mpi25IeeeSgeChain64_t           *scsi_raid_io_sgl_ieee;
2312         MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
2313         uint32_t                        index;
2314         ddi_acc_handle_t acc_handle =
2315             instance->mpi2_frame_pool_dma_obj.acc_handle;

2317         if (!instance->tbolt) {
2318                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled."));
```

```
2332                con_log(CL_ANN, (CE_NOTE, "Not MFA enabled.\n"));
2319                return;
2320        }

2322        index = cmd->index;

2324        ReqDescUnion = mr_sas_get_request_descriptor(instance, index);

2326        if (!ReqDescUnion) {
2327                con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]"));
2328                return;
2329        }

2331        con_log(CL_ANN1, (CE_NOTE, "[SMID]%x", cmd->SMID));

2333        ReqDescUnion->Words = 0;

2335        ReqDescUnion->SCSIIO.RequestFlags =
2336            (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
2337            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

2339        ReqDescUnion->SCSIIO.SMID = cmd->SMID;

2341        cmd->request_desc = ReqDescUnion;

2343        /* get raid message frame pointer */
2344        scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2346        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
2347                Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)
2348                    &scsi_raid_io->SGL.IeeeChain;
2349                sgl_ptr_end += instance->max_sge_in_main_msg - 1;
2350                ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
2351        }

2353        ddi_put8(acc_handle, &scsi_raid_io->Function,
2354            MPI2_FUNCTION_PASSTHRU_IO_REQUEST);

2356        ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
2357            offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

2359        ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
2360            (U8)offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16);

2362        ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
2363            cmd->sense_phys_addr1);


2366        scsi_raid_io_sgl_ieee =
2367            (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

2369        ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
2370            (U64)cmd->frame_phys_addr);

2372        ddi_put8(acc_handle,
2373            &scsi_raid_io_sgl_ieee->Flags, (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2374            MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
2375        /* LSI put hardcoded 1024 instead of MEGASAS_MAX_SZ_CHAIN_FRAME. */
2376        ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length, 1024);

2378        con_log(CL_ANN1, (CE_NOTE,
2379            "[MFI CMD PHY ADDRESS]:%" PRIx64,
2380            scsi_raid_io_sgl_ieee->Address));
2381        con_log(CL_ANN1, (CE_NOTE,
2382            "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
2383        con_log(CL_ANN1, (CE_NOTE, "[SGL Flags]:%x",
```



```
2384            scsi_raid_io_sgl_ieee->Flags));
2385 }


2388 void
2389 tbolt_complete_cmd(struct mrsas_instance *instance,
2390     struct mrsas_cmd *cmd)
2391 {
2392         uint8_t                         status;
2393         uint8_t                         extStatus;
2394         uint8_t                         arm;
2395         struct scsa_cmd                 *acmd;
2396         struct scsi_pkt                 *pkt;
2397         struct scsi_arq_status          *arqstat;
2398         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2399         LD_LOAD_BALANCE_INFO            *lbinfo;
2400         ddi_acc_handle_t acc_handle =
2401             instance->mpi2_frame_pool_dma_obj.acc_handle;

2403         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2405         status = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.status);
2406         extStatus = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.extStatus);

2408         con_log(CL_DLEVEL3, (CE_NOTE, "status %x", status));
2409         con_log(CL_DLEVEL3, (CE_NOTE, "extStatus %x", extStatus));

2411         if (status != MFI_STAT_OK) {
2412                 con_log(CL_ANN, (CE_WARN,
2413                     "IO Cmd Failed SMID %x", cmd->SMID));
2414         } else {
2415                 con_log(CL_ANN, (CE_NOTE,
2416                     "IO Cmd Success  SMID %x", cmd->SMID));
2417         }

2419         /* regular commands */

2421         switch (ddi_get8(acc_handle, &scsi_raid_io->Function)) {

2423         case MPI2_FUNCTION_SCSI_IO_REQUEST :  /* Fast Path IO. */
2424                 acmd =  (struct scsa_cmd *)cmd->cmd;
2425                 lbinfo = &instance->load_balance_info[acmd->device_id];

2427                 if (cmd->load_balance_flag & MEGASAS_LOAD_BALANCE_FLAG) {
2428                         arm = lbinfo->raid1DevHandle[0] ==
2429                             scsi_raid_io->DevHandle ? 0 : 1;

2431                         lbinfo->scsi_pending_cmds[arm]--;
2432                         cmd->load_balance_flag &= ~MEGASAS_LOAD_BALANCE_FLAG;
2433                 }
2434                 con_log(CL_DLEVEL3, (CE_NOTE,
2435                     "FastPath IO Completion Success "));
2436                 /* FALLTHRU */

2438         case MPI2_FUNCTION_LD_IO_REQUEST :   { /* Regular Path IO. */
2439                 acmd =  (struct scsa_cmd *)cmd->cmd;
2440                 pkt =   (struct scsi_pkt *)CMD2PKT(acmd);

2442                 if (acmd->cmd_flags & CFLAG_DMAVALID) {
2443                         if (acmd->cmd_flags & CFLAG_CONSISTENT) {
2444                                 (void) ddi_dma_sync(acmd->cmd_dmahandle,
2445                                     acmd->cmd_dma_offset, acmd->cmd_dma_len,
2446                                     DDI_DMA_SYNC_FORCPU);
2447                         }
2448                 }
```

```
2450                 pkt->pkt_reason         = CMD_CMPLT;
2451                 pkt->pkt_statistics     = 0;
2452                 pkt->pkt_state = STATE_GOT_BUS | STATE_GOT_TARGET |
2453                     STATE_SENT_CMD | STATE_XFERRED_DATA | STATE_GOT_STATUS;

2455                 con_log(CL_ANN, (CE_CONT, " CDB[0] = %x completed for %s: "
2456                     "size %lx SMID %x cmd_status %x", pkt->pkt_cdbp[0],
2457                     ((acmd->islogical) ? "LD" : "PD"),
2458                     acmd->cmd_dmacount, cmd->SMID, status));

2460                 if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
2461                         struct scsi_inquiry     *inq;

2463                         if (acmd->cmd_dmacount != 0) {
2464                                 bp_mapin(acmd->cmd_buf);
2465                                 inq = (struct scsi_inquiry *)
2466                                     acmd->cmd_buf->b_un.b_addr;

2468                                 /* don't expose physical drives to OS */
2469                                 if (acmd->islogical &&
2470                                     (status == MFI_STAT_OK)) {
2471                                         display_scsi_inquiry((caddr_t)inq);
2472 #ifdef PDSUPPORT
2473                                 } else if ((status == MFI_STAT_OK) &&
2474                                     inq->inq_dtype == DTYPE_DIRECT) {
2475                                         display_scsi_inquiry((caddr_t)inq);
2476 #endif
2477                                 } else {
2478                                         /* for physical disk */
2479                                         status = MFI_STAT_DEVICE_NOT_FOUND;
2480                                 }
2481                         }
2482                 }

2484                 switch (status) {
2485                 case MFI_STAT_OK:
2486                         pkt->pkt_scbp[0] = STATUS_GOOD;
2487                         break;
2488                 case MFI_STAT_LD_CC_IN_PROGRESS:
2489                 case MFI_STAT_LD_RECON_IN_PROGRESS:
2490                         pkt->pkt_scbp[0] = STATUS_GOOD;
2491                         break;
2492                 case MFI_STAT_LD_INIT_IN_PROGRESS:
2493                         pkt->pkt_reason = CMD_TRAN_ERR;
2494                         break;
2495                 case MFI_STAT_SCSI_IO_FAILED:
2496                         cmn_err(CE_WARN, "tbolt_complete_cmd: scsi_io failed");
2497                         pkt->pkt_reason = CMD_TRAN_ERR;
2498                         break;
2499                 case MFI_STAT_SCSI_DONE_WITH_ERROR:
2500                         con_log(CL_ANN, (CE_WARN,
2501                             "tbolt_complete_cmd: scsi_done with error"));

2503                         pkt->pkt_reason = CMD_CMPLT;
2504                         ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;

2506                         if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
2507                                 con_log(CL_ANN,
2508                                     (CE_WARN, "TEST_UNIT_READY fail"));
2509                         } else {
2510                                 pkt->pkt_state |= STATE_ARQ_DONE;
2511                                 arqstat = (void *)(pkt->pkt_scbp);
2512                                 arqstat->sts_rqpkt_reason = CMD_CMPLT;
2513                                 arqstat->sts_rqpkt_resid = 0;
2514                                 arqstat->sts_rqpkt_state |=
2515                                     STATE_GOT_BUS | STATE_GOT_TARGET
```



```
2516                                     | STATE_SENT_CMD
2517                                     | STATE_XFERRED_DATA;
2518                                 *(uint8_t *)&arqstat->sts_rqpkt_status =
2519                                     STATUS_GOOD;
2520                                 con_log(CL_ANN1,
2521                                     (CE_NOTE, "Copying Sense data %x",
2522                                     cmd->SMID));

2524                                 ddi_rep_get8(acc_handle,
2525                                     (uint8_t *)&(arqstat->sts_sensedata),
2526                                     cmd->sense1,
2527                                     sizeof (struct scsi_extended_sense),
2528                                     DDI_DEV_AUTOINCR);

2530                         }
2531                         break;
2532                 case MFI_STAT_LD_OFFLINE:
2533                         cmn_err(CE_WARN,
2534                             "tbolt_complete_cmd: ld offline "
2535                             "CDB[0]=0x%x targetId=0x%x devhandle=0x%x",
2549                             "CDB[0]=0x%x targetId=0x%x devhandle=0x%x\n",
2536                             /* UNDO: */
2537                             ddi_get8(acc_handle, &scsi_raid_io->CDB.CDB32[0]),

2539                             ddi_get16(acc_handle,
2540                             &scsi_raid_io->RaidContext.ldTargetId),

2542                             ddi_get16(acc_handle, &scsi_raid_io->DevHandle));

2544                         pkt->pkt_reason = CMD_DEV_GONE;
2545                         pkt->pkt_statistics  = STAT_DISCON;
2546                         break;
2547                 case MFI_STAT_DEVICE_NOT_FOUND:
2548                         con_log(CL_ANN, (CE_CONT,
2549                             "tbolt_complete_cmd: device not found error"));
2550                         pkt->pkt_reason = CMD_DEV_GONE;
2551                         pkt->pkt_statistics  = STAT_DISCON;
2552                         break;

2554                 case MFI_STAT_LD_LBA_OUT_OF_RANGE:
2555                         pkt->pkt_state |= STATE_ARQ_DONE;
2556                         pkt->pkt_reason = CMD_CMPLT;
2557                         ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;

2559                         arqstat = (void *)(pkt->pkt_scbp);
2560                         arqstat->sts_rqpkt_reason = CMD_CMPLT;
2561                         arqstat->sts_rqpkt_resid = 0;
2562                         arqstat->sts_rqpkt_state |= STATE_GOT_BUS
2563                             | STATE_GOT_TARGET | STATE_SENT_CMD
2564                             | STATE_XFERRED_DATA;
2565                         *(uint8_t *)&arqstat->sts_rqpkt_status = STATUS_GOOD;

2567                         arqstat->sts_sensedata.es_valid = 1;
2568                         arqstat->sts_sensedata.es_key = KEY_ILLEGAL_REQUEST;
2569                         arqstat->sts_sensedata.es_class = CLASS_EXTENDED_SENSE;

2571                         /*
2572                          * LOGICAL BLOCK ADDRESS OUT OF RANGE:
2573                          * ASC: 0x21h; ASCQ: 0x00h;
2574                          */
2575                         arqstat->sts_sensedata.es_add_code = 0x21;
2576                         arqstat->sts_sensedata.es_qual_code = 0x00;
2577                         break;
2578                 case MFI_STAT_INVALID_CMD:
2579                 case MFI_STAT_INVALID_DCMD:
2580                 case MFI_STAT_INVALID_PARAMETER:
```

```
2581                case MFI_STAT_INVALID_SEQUENCE_NUMBER:
2582                default:
2583                        cmn_err(CE_WARN, "tbolt_complete_cmd: Unknown status!");
2584                        pkt->pkt_reason = CMD_TRAN_ERR;

2586                        break;
2587                }

2589                atomic_add_16(&instance->fw_outstanding, (-1));

2591                (void) mrsas_common_check(instance, cmd);
2592                if (acmd->cmd_dmahandle) {
2593                        if (mrsas_check_dma_handle(acmd->cmd_dmahandle) !=
2594                            DDI_SUCCESS) {
2595                                ddi_fm_service_impact(instance->dip,
2596                                    DDI_SERVICE_UNAFFECTED);
2597                                pkt->pkt_reason = CMD_TRAN_ERR;
2598                                pkt->pkt_statistics = 0;
2599                        }
2600                }

2602                /* Call the callback routine */
2603                if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp)
2604                        (*pkt->pkt_comp)(pkt);

2606                con_log(CL_ANN1, (CE_NOTE, "Free smid %x", cmd->SMID));

2608                ddi_put8(acc_handle, &scsi_raid_io->RaidContext.status, 0);

2610                ddi_put8(acc_handle, &scsi_raid_io->RaidContext.extStatus, 0);

2612                return_raid_msg_pkt(instance, cmd);
2613                break;
2614        }
2615        case MPI2_FUNCTION_PASSTHRU_IO_REQUEST:  /* MFA command. */

2617                if (cmd->frame->dcmd.opcode == MR_DCMD_LD_MAP_GET_INFO &&
2618                    cmd->frame->dcmd.mbox.b[1] == 1) {

2620                        mutex_enter(&instance->sync_map_mtx);

2622                        con_log(CL_ANN, (CE_NOTE,
2623                            "LDMAP sync command SMID RECEIVED 0x%X",
2624                            cmd->SMID));
2625                        if (cmd->frame->hdr.cmd_status != 0) {
2626                                cmn_err(CE_WARN,
2627                                    "map sync failed, status = 0x%x.",
2641                                    "map sync failed, status = 0x%x.\n",
2628                                    cmd->frame->hdr.cmd_status);
2629                        } else {
2630                                instance->map_id++;
2631                                cmn_err(CE_NOTE,
2632                                    "map sync received, switched map_id to %"
2633                                    PRIu64 " \n", instance->map_id);
2634                        }

2636                        if (MR_ValidateMapInfo(instance->ld_map[
2637                            (instance->map_id & 1)],
2638                            instance->load_balance_info)) {
2639                                instance->fast_path_io = 1;
2640                        } else {
2641                                instance->fast_path_io = 0;
2642                        }

2644                        con_log(CL_ANN, (CE_NOTE,
2645                            "instance->fast_path_io %d",
```



```
2659                            "instance->fast_path_io %d \n",
2646                            instance->fast_path_io));

2648                        instance->unroll.syncCmd = 0;

2650                        if (instance->map_update_cmd == cmd) {
2651                                return_raid_msg_pkt(instance, cmd);
2652                                atomic_add_16(&instance->fw_outstanding, (-1));
2653                                (void) mrsas_tbolt_sync_map_info(instance);
2654                        }

2656                        cmn_err(CE_NOTE, "LDMAP sync completed.");
2670                        cmn_err(CE_NOTE, "LDMAP sync completed.\n");
2657                        mutex_exit(&instance->sync_map_mtx);
2658                        break;
2659                }

2661                if (cmd->frame->dcmd.opcode == MR_DCMD_CTRL_EVENT_WAIT) {
2662                        con_log(CL_ANN1, (CE_CONT,
2663                            "AEN command SMID RECEIVED 0x%X",
2664                            cmd->SMID));
2665                        if ((instance->aen_cmd == cmd) &&
2666                            (instance->aen_cmd->abort_aen)) {
2667                                con_log(CL_ANN, (CE_WARN, "mrsas_softintr: "
2668                                    "aborted_aen returned"));
2669                        } else {
2670                                atomic_add_16(&instance->fw_outstanding, (-1));
2671                                service_mfi_aen(instance, cmd);
2672                        }
2673                }

2675                if (cmd->sync_cmd == MRSAS_TRUE) {
2676                        con_log(CL_ANN1, (CE_CONT,
2677                            "Sync-mode Command Response SMID RECEIVED 0x%X",
2678                            cmd->SMID));

2680                        tbolt_complete_cmd_in_sync_mode(instance, cmd);
2681                } else {
2682                        con_log(CL_ANN, (CE_CONT,
2683                            "tbolt_complete_cmd: Wrong SMID RECEIVED 0x%X",
2684                            cmd->SMID));
2685                }
2686                break;
2687        default:
2688                mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2689                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

2691                /* free message */
2692                con_log(CL_ANN,
2693                    (CE_NOTE, "tbolt_complete_cmd: Unknown Type!!!!!!!!"));
2694                break;
2695        }
2696 }

2698 uint_t
2699 mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *instance)
2700 {
2701        uint8_t                         replyType;
2702        Mpi2SCSIIOSuccessReplyDescriptor_t *replyDesc;
2703        Mpi2ReplyDescriptorsUnion_t     *desc;
2704        uint16_t                        smid;
2705        union desc_value                d_val;
2706        struct mrsas_cmd                *cmd;

2708        struct mrsas_header     *hdr;
2709        struct scsi_pkt         *pkt;
```

```
2711        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2712            0, 0, DDI_DMA_SYNC_FORDEV);

2714        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2715            0, 0, DDI_DMA_SYNC_FORCPU);

2717        desc = instance->reply_frame_pool;
2718        desc += instance->reply_read_index;

2720        replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;
2721        replyType = replyDesc->ReplyFlags &
2722            MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2724        if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2725                return (DDI_INTR_UNCLAIMED);

2727        if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
2728            != DDI_SUCCESS) {
2729                mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2730                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2731                con_log(CL_ANN1,
2732                    (CE_WARN, "mr_sas_tbolt_process_outstanding_cmd(): "
2733                    "FMA check, returning DDI_INTR_UNCLAIMED"));
2734                return (DDI_INTR_CLAIMED);
2735        }

2737        con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %p  Words = %" PRIx64,
2751        con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %p  Words = %" PRIx64 " \n",
2738            (void *)desc, desc->Words));

2740        d_val.word = desc->Words;


2743        /* Read Reply descriptor */
2744        while ((d_val.u1.low != 0xffffffff) &&
2745            (d_val.u1.high != 0xffffffff)) {

2747                (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2748                    0, 0, DDI_DMA_SYNC_FORCPU);

2750                smid = replyDesc->SMID;

2752                if (!smid || smid > instance->max_fw_cmds + 1) {
2753                        con_log(CL_ANN1, (CE_NOTE,
2754                            "Reply Desc at Break  = %p  Words = %" PRIx64,
2768                            "Reply Desc at Break  = %p  Words = %" PRIx64 " \n",
2755                            (void *)desc, desc->Words));
2756                        break;
2757                }

2759                cmd     = instance->cmd_list[smid - 1];
2760                if (!cmd) {
2761                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_tbolt_process_"
2762                            "outstanding_cmd: Invalid command "
2763                            " or Poll commad Received in completion path"));
2777                            " or Poll commad Received in completion path\n"));
2764                } else {
2765                        mutex_enter(&instance->cmd_pend_mtx);
2766                        if (cmd->sync_cmd == MRSAS_TRUE) {
2767                                hdr = (struct mrsas_header *)&cmd->frame->hdr;
2768                                if (hdr) {
2769                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2770                                            "tbolt_process_outstanding_cmd:"
2771                                            " mlist_del_init(&cmd->list)."));
2785                                            " mlist_del_init(&cmd->list).\n"));
```



```
2772                                        mlist_del_init(&cmd->list);
2773                                }
2774                        } else {
2775                                pkt = cmd->pkt;
2776                                if (pkt) {
2777                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2778                                            "tbolt_process_outstanding_cmd:"
2779                                            "mlist_del_init(&cmd->list).));
2793                                            "mlist_del_init(&cmd->list).\n"));
2780                                        mlist_del_init(&cmd->list);
2781                                }
2782                        }

2784                        mutex_exit(&instance->cmd_pend_mtx);

2786                        tbolt_complete_cmd(instance, cmd);
2787                }
2788                /* set it back to all 1s. */
2789                desc->Words = -1LL;

2791                instance->reply_read_index++;

2793                if (instance->reply_read_index >= (instance->reply_q_depth)) {
2794                        con_log(CL_ANN1, (CE_NOTE, "wrap around"));
2795                        instance->reply_read_index = 0;
2796                }

2798                /* Get the next reply descriptor */
2799                if (!instance->reply_read_index)
2800                        desc = instance->reply_frame_pool;
2801                else
2802                        desc++;

2804                replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;

2806                d_val.word = desc->Words;

2808                con_log(CL_ANN1, (CE_NOTE,
2809                    "Next Reply Desc  = %p Words = %" PRIx64,
2823                    "Next Reply Desc  = %p Words = %" PRIx64 "\n",
2810                    (void *)desc, desc->Words));

2812                replyType = replyDesc->ReplyFlags &
2813                    MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2815                if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2816                        break;

2818        } /* End of while loop. */

2820        /* update replyIndex to FW */
2821        WR_MPI2_REPLY_POST_INDEX(instance->reply_read_index, instance);


2824        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2825            0, 0, DDI_DMA_SYNC_FORDEV);

2827        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2828            0, 0, DDI_DMA_SYNC_FORCPU);
2829        return (DDI_INTR_CLAIMED);
2830 }
```

_____***unchanged_portion_omitted_***

```
2863 /*
2864  * mrsas_tbolt_get_ld_map_info -        Returns  ld_map structure
2865  * instance:                            Adapter soft state
```

```
2866  *
2867  * Issues an internal command (DCMD) to get the FW's controller PD
2868  * list structure.  This information is mainly used to find out SYSTEM
2869  * supported by the FW.
2870  */
2871 int
2872 mrsas_tbolt_get_ld_map_info(struct mrsas_instance *instance)
2873 {
2874         int ret = 0;
2875         struct mrsas_cmd        *cmd = NULL;
2876         struct mrsas_dcmd_frame *dcmd;
2877         MR_FW_RAID_MAP_ALL *ci;
2878         uint32_t ci_h = 0;
2879         U32 size_map_info;

2881         cmd = get_raid_msg_pkt(instance);

2883         if (cmd == NULL) {
2884                 cmn_err(CE_WARN,
2885                     "Failed to get a cmd from free-pool in get_ld_map_info()");
2886                 return (DDI_FAILURE);
2887         }

2889         dcmd = &cmd->frame->dcmd;

2891         size_map_info = sizeof (MR_FW_RAID_MAP) +
2892             (sizeof (MR_LD_SPAN_MAP) *
2893             (MAX_LOGICAL_DRIVES - 1));

2895         con_log(CL_ANN, (CE_NOTE,
2896             "size_map_info : 0x%x", size_map_info));

2898         ci = instance->ld_map[(instance->map_id & 1)];
2899         ci_h = instance->ld_map_phy[(instance->map_id & 1)];

2901         if (!ci) {
2902                 cmn_err(CE_WARN, "Failed to alloc mem for ld_map_info");
2903                 return_raid_msg_pkt(instance, cmd);
2904                 return (-1);
2905         }

2907         bzero(ci, sizeof (*ci));
2908         bzero(dcmd->mbox.b, DCMD_MBOX_SZ);
2921         (void) memset(ci, 0, sizeof (*ci));
2922         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

2910         dcmd->cmd = MFI_CMD_OP_DCMD;
2911         dcmd->cmd_status = 0xFF;
2912         dcmd->sge_count = 1;
2913         dcmd->flags = MFI_FRAME_DIR_READ;
2914         dcmd->timeout = 0;
2915         dcmd->pad_0 = 0;
2916         dcmd->data_xfer_len = size_map_info;
2917         dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
2918         dcmd->sgl.sge32[0].phys_addr = ci_h;
2919         dcmd->sgl.sge32[0].length = size_map_info;


2922         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

2924         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
2925                 ret = 0;
2926                 con_log(CL_ANN1, (CE_NOTE, "Get LD Map Info success"));
2940                 con_log(CL_ANN1, (CE_NOTE,
2941                     "Get LD Map Info success\n"));
2927         } else {
```



```
2928                 cmn_err(CE_WARN, "Get LD Map Info failed");
2943                 cmn_err(CE_WARN,
2944                     "Get LD Map Info failed\n");
2929                 ret = -1;
2930         }

2932         return_raid_msg_pkt(instance, cmd);

2934         return (ret);
2935 }
_____unchanged_portion_omitted_

2954 /*
2955  * mrsas_tbolt_command_create - Create command for fast path.
2956  * @io_info:    MegaRAID IO request packet pointer.
2957  * @ref_tag:    Reference tag for RD/WRPROTECT
2958  *
2959  * Create the command for fast path.
2960  */
2961 void
2962 mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2963     struct IO_REQUEST_INFO *io_info, Mpi2RaidSCSIIORequest_t *scsi_io_request,
2964     U32 ref_tag)
2965 {
2966         uint16_t                EEDPFlags;
2967         uint32_t                Control;
2968         ddi_acc_handle_t acc_handle =
2969             instance->mpi2_frame_pool_dma_obj.acc_handle;

2971         /* Prepare 32-byte CDB if DIF is supported on this device */
2972         con_log(CL_ANN, (CE_NOTE, "Prepare DIF CDB"));
2988         con_log(CL_ANN, (CE_NOTE, "Prepare DIF CDB\n"));

2974         bzero(cdb, 32);
2990         (void) memset(cdb, 0, 32);

2976         cdb[0] =  MRSAS_SCSI_VARIABLE_LENGTH_CMD;


2979         cdb[7] =  MRSAS_SCSI_ADDL_CDB_LEN;

2981         if (io_info->isRead)
2982                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_READ32;
2983         else
2984                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_WRITE32;

2986         /* Verify within linux driver, set to MEGASAS_RD_WR_PROTECT_CHECK_ALL */
2987         cdb[10] = MRSAS_RD_WR_PROTECT;

2989         /* LOGICAL BLOCK ADDRESS */
2990         cdb[12] = (U8)(((io_info->pdBlock) >> 56) & 0xff);
2991         cdb[13] = (U8)(((io_info->pdBlock) >> 48) & 0xff);
2992         cdb[14] = (U8)(((io_info->pdBlock) >> 40) & 0xff);
2993         cdb[15] = (U8)(((io_info->pdBlock) >> 32) & 0xff);
2994         cdb[16] = (U8)(((io_info->pdBlock) >> 24) & 0xff);
2995         cdb[17] = (U8)(((io_info->pdBlock) >> 16) & 0xff);
2996         cdb[18] = (U8)(((io_info->pdBlock) >> 8) & 0xff);
2997         cdb[19] = (U8)((io_info->pdBlock) & 0xff);

2999         /* Logical block reference tag */
3000         ddi_put32(acc_handle, &scsi_io_request->CDB.EEDP32.PrimaryReferenceTag,
3001             BE_32(ref_tag));

3003         ddi_put16(acc_handle,
3004             &scsi_io_request->CDB.EEDP32.PrimaryApplicationTagMask, 0xffff);
```

```
3006         ddi_put32(acc_handle, &scsi_io_request->DataLength,
3007             ((io_info->numBlocks)*512));
3008         /* Specify 32-byte cdb */
3009         ddi_put16(acc_handle, &scsi_io_request->IoFlags, 32);

3011         /* Transfer length */
3012         cdb[28] = (U8)(((io_info->numBlocks) >> 24) & 0xff);
3013         cdb[29] = (U8)(((io_info->numBlocks) >> 16) & 0xff);
3014         cdb[30] = (U8)(((io_info->numBlocks) >> 8) & 0xff);
3015         cdb[31] = (U8)((io_info->numBlocks) & 0xff);

3017         /* set SCSI IO EEDPFlags */
3018         EEDPFlags = ddi_get16(acc_handle, &scsi_io_request->EEDPFlags);
3019         Control = ddi_get32(acc_handle, &scsi_io_request->Control);

3021         /* set SCSI IO EEDPFlags bits */
3022         if (io_info->isRead) {
3023                 /*
3024                  * For READ commands, the EEDPFlags shall be set to specify to
3025                  * Increment the Primary Reference Tag, to Check the Reference
3026                  * Tag, and to Check and Remove the Protection Information
3027                  * fields.
3028                  */
3029                 EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3030                     MPI2_SCSIIO_EEDPFLAGS_CHECK_REFTAG  |
3031                     MPI2_SCSIIO_EEDPFLAGS_CHECK_REMOVE_OP       |
3032                     MPI2_SCSIIO_EEDPFLAGS_CHECK_APPTAG  |
3033                     MPI2_SCSIIO_EEDPFLAGS_CHECK_GUARD;
3034         } else {
3035                 /*
3036                  * For WRITE commands, the EEDPFlags shall be set to specify to
3037                  * Increment the Primary Reference Tag, and to Insert
3038                  * Protection Information fields.
3039                  */
3040                 EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3041                     MPI2_SCSIIO_EEDPFLAGS_INSERT_OP;
3042         }
3043         Control |= (0x4 << 26);

3045         ddi_put16(acc_handle, &scsi_io_request->EEDPFlags, EEDPFlags);
3046         ddi_put32(acc_handle, &scsi_io_request->Control, Control);
3047         ddi_put32(acc_handle,
3048             &scsi_io_request->EEDPBlockSize, MRSAS_EEDPBLOCKSIZE);
3049 }


3052 /*
3053  * mrsas_tbolt_set_pd_lba -     Sets PD LBA
3054  * @cdb:               CDB
3055  * @cdb_len:           cdb length
3056  * @start_blk:         Start block of IO
3057  *
3058  * Used to set the PD LBA in CDB for FP IOs
3059  */
3060 static void
3061 mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len_ptr, U64 start_blk,
3062     U32 num_blocks)
3063 {
3064         U8 cdb_len = *cdb_len_ptr;
3065         U8 flagvals = 0, opcode = 0, groupnum = 0, control = 0;

3067         /* Some drives don't support 16/12 byte CDB's, convert to 10 */
3068         if (((cdb_len == 12) || (cdb_len == 16)) &&
3069             (start_blk <= 0xffffffff)) {
3070                 if (cdb_len == 16) {
3071                         con_log(CL_ANN,
```



```
3072                             (CE_NOTE, "Converting READ/WRITE(16) to READ10"));
3088                             (CE_NOTE, "Converting READ/WRITE(16) to READ10\n"));
3073                         opcode = cdb[0] == READ_16 ? READ_10 : WRITE_10;
3074                         flagvals = cdb[1];
3075                         groupnum = cdb[14];
3076                         control = cdb[15];
3077                 } else {
3078                         con_log(CL_ANN,
3079                             (CE_NOTE, "Converting READ/WRITE(12) to READ10"));
3095                             (CE_NOTE, "Converting READ/WRITE(12) to READ10\n"));
3080                         opcode = cdb[0] == READ_12 ? READ_10 : WRITE_10;
3081                         flagvals = cdb[1];
3082                         groupnum = cdb[10];
3083                         control = cdb[11];
3084                 }

3086                 bzero(cdb, sizeof (cdb));
3102                 (void) memset(cdb, 0, sizeof (cdb));

3088                 cdb[0] = opcode;
3089                 cdb[1] = flagvals;
3090                 cdb[6] = groupnum;
3091                 cdb[9] = control;
3092                 /* Set transfer length */
3093                 cdb[8] = (U8)(num_blocks & 0xff);
3094                 cdb[7] = (U8)((num_blocks >> 8) & 0xff);
3095                 cdb_len = 10;
3096         } else if ((cdb_len < 16) && (start_blk > 0xffffffff)) {
3097                 /* Convert to 16 byte CDB for large LBA's */
3098                 con_log(CL_ANN,
3099                     (CE_NOTE, "Converting 6/10/12 CDB to 16 byte CDB"));
3115                     (CE_NOTE, "Converting 6/10/12 CDB to 16 byte CDB\n"));
3100                 switch (cdb_len) {
3101                 case 6:
3102                         opcode = cdb[0] == READ_6 ? READ_16 : WRITE_16;
3103                         control = cdb[5];
3104                         break;
3105                 case 10:
3106                         opcode = cdb[0] == READ_10 ? READ_16 : WRITE_16;
3107                         flagvals = cdb[1];
3108                         groupnum = cdb[6];
3109                         control = cdb[9];
3110                         break;
3111                 case 12:
3112                         opcode = cdb[0] == READ_12 ? READ_16 : WRITE_16;
3113                         flagvals = cdb[1];
3114                         groupnum = cdb[10];
3115                         control = cdb[11];
3116                         break;
3117                 }

3119                 bzero(cdb, sizeof (cdb));
3135                 (void) memset(cdb, 0, sizeof (cdb));

3121                 cdb[0] = opcode;
3122                 cdb[1] = flagvals;
3123                 cdb[14] = groupnum;
3124                 cdb[15] = control;

3126                 /* Transfer length */
3127                 cdb[13] = (U8)(num_blocks & 0xff);
3128                 cdb[12] = (U8)((num_blocks >> 8) & 0xff);
3129                 cdb[11] = (U8)((num_blocks >> 16) & 0xff);
3130                 cdb[10] = (U8)((num_blocks >> 24) & 0xff);

3132                 /* Specify 16-byte cdb */
```

```
3133                    cdb_len = 16;
3134            } else if ((cdb_len == 6) && (start_blk > 0x1fffff)) {
3135                    /* convert to 10 byte CDB */
3136                    opcode = cdb[0] == READ_6 ? READ_10 : WRITE_10;
3137                    control = cdb[5];

3139                    bzero(cdb, sizeof (cdb));
3155                    (void) memset(cdb, 0, sizeof (cdb));
3140                    cdb[0] = opcode;
3141                    cdb[9] = control;

3143                    /* Set transfer length */
3144                    cdb[8] = (U8)(num_blocks & 0xff);
3145                    cdb[7] = (U8)((num_blocks >> 8) & 0xff);

3147                    /* Specify 10-byte cdb */
3148                    cdb_len = 10;
3149            }


3152            /* Fall through Normal case, just load LBA here */
3153            switch (cdb_len) {
3154            case 6:
3155            {
3156                    U8 val = cdb[1] & 0xE0;
3157                    cdb[3] = (U8)(start_blk & 0xff);
3158                    cdb[2] = (U8)((start_blk >> 8) & 0xff);
3159                    cdb[1] = val | ((U8)(start_blk >> 16) & 0x1f);
3160                    break;
3161            }
3162            case 10:
3163                    cdb[5] = (U8)(start_blk & 0xff);
3164                    cdb[4] = (U8)((start_blk >> 8) & 0xff);
3165                    cdb[3] = (U8)((start_blk >> 16) & 0xff);
3166                    cdb[2] = (U8)((start_blk >> 24) & 0xff);
3167                    break;
3168            case 12:
3169                    cdb[5]    = (U8)(start_blk & 0xff);
3170                    cdb[4]    = (U8)((start_blk >> 8) & 0xff);
3171                    cdb[3]    = (U8)((start_blk >> 16) & 0xff);
3172                    cdb[2]    = (U8)((start_blk >> 24) & 0xff);
3173                    break;

3175            case 16:
3176                    cdb[9]  = (U8)(start_blk & 0xff);
3177                    cdb[8]  = (U8)((start_blk >> 8) & 0xff);
3178                    cdb[7]  = (U8)((start_blk >> 16) & 0xff);
3179                    cdb[6]  = (U8)((start_blk >> 24) & 0xff);
3180                    cdb[5]  = (U8)((start_blk >> 32) & 0xff);
3181                    cdb[4]  = (U8)((start_blk >> 40) & 0xff);
3182                    cdb[3]  = (U8)((start_blk >> 48) & 0xff);
3183                    cdb[2]  = (U8)((start_blk >> 56) & 0xff);
3184                    break;
3185            }

3187            *cdb_len_ptr = cdb_len;
3188    }


3191    static int
3192    mrsas_tbolt_check_map_info(struct mrsas_instance *instance)
3193    {
3194            MR_FW_RAID_MAP_ALL *ld_map;

3196            if (!mrsas_tbolt_get_ld_map_info(instance)) {
```



```
3198                    ld_map = instance->ld_map[(instance->map_id & 1)];

3200                    con_log(CL_ANN1, (CE_NOTE, "ldCount=%d, map size=%d",
3216                    con_log(CL_ANN1, (CE_NOTE, "ldCount=%d, map size=%d\n",
3201                        ld_map->raidMap.ldCount, ld_map->raidMap.totalSize));

3203                    if (MR_ValidateMapInfo(instance->ld_map[
3204                        (instance->map_id & 1)], instance->load_balance_info)) {
3205                            con_log(CL_ANN,
3206                                (CE_CONT, "MR_ValidateMapInfo success"));

3208                            instance->fast_path_io = 1;
3209                            con_log(CL_ANN,
3210                                (CE_NOTE, "instance->fast_path_io %d",
3226                                (CE_NOTE, "instance->fast_path_io %d \n",
3211                                instance->fast_path_io));

3213                            return (DDI_SUCCESS);
3214                    }

3216            }

3218            instance->fast_path_io = 0;
3219            cmn_err(CE_WARN, "MR_ValidateMapInfo failed");
3220            con_log(CL_ANN, (CE_NOTE,
3221                "instance->fast_path_io %d", instance->fast_path_io));
3237                "instance->fast_path_io %d \n", instance->fast_path_io));

3223            return (DDI_FAILURE);
3224    }

3226    /*
3227     * Marks HBA as bad. This will be called either when an
3228     * IO packet times out even after 3 FW resets
3229     * or FW is found to be fault even after 3 continuous resets.
3230     */

3232    void
3233    mrsas_tbolt_kill_adapter(struct mrsas_instance *instance)
3234    {
3235            cmn_err(CE_NOTE, "TBOLT Kill adapter called");
3251            cmn_err(CE_WARN, "TBOLT Kill adapter called\n");

3237            if (instance->deadadapter == 1)
3238                    return;

3240            con_log(CL_ANN1, (CE_NOTE, "tbolt_kill_adapter: "
3241                "Writing to doorbell with MFI_STOP_ADP "));
3242            mutex_enter(&instance->ocr_flags_mtx);
3243            instance->deadadapter = 1;
3244            mutex_exit(&instance->ocr_flags_mtx);
3245            instance->func_ptr->disable_intr(instance);
3246            WR_RESERVED0_REGISTER(MFI_STOP_ADP, instance);
3247            /* Flush */
3248            (void) RD_RESERVED0_REGISTER(instance);

3250            (void) mrsas_print_pending_cmds(instance);
3251            (void) mrsas_complete_pending_cmds(instance);
3252    }
______unchanged_portion_omitted_

3270    int
3271    mrsas_tbolt_reset_ppc(struct mrsas_instance *instance)
3272    {
3273            uint32_t status = 0x00;
3274            uint32_t retry = 0;
```

```
3275         uint32_t cur_abs_reg_val;
3276         uint32_t fw_state;
3277         uint32_t abs_state;
3278         uint32_t i;

3280         con_log(CL_ANN, (CE_NOTE,
3281             "mrsas_tbolt_reset_ppc entered"));
3297             "mrsas_tbolt_reset_ppc entered\n "));

3283         if (instance->deadadapter == 1) {
3284                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3285                     "no more resets as HBA has been marked dead ");
3286                 return (DDI_FAILURE);
3287         }

3289         mutex_enter(&instance->ocr_flags_mtx);
3290         instance->adapterresetinprogress = 1;
3291         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc:"
3292             "adpterresetinprogress flag set, time %llx", gethrtime()));
3293         mutex_exit(&instance->ocr_flags_mtx);

3295         instance->func_ptr->disable_intr(instance);

3297         /* Add delay inorder to complete the ioctl & io cmds in-flight */
3298         for (i = 0; i < 3000; i++) {
3299                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
3300         }

3302         instance->reply_read_index = 0;

3304 retry_reset:
3305         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3306             ":Resetting TBOLT "));

3308         WR_TBOLT_IB_WRITE_SEQ(0xF, instance);
3309         WR_TBOLT_IB_WRITE_SEQ(4, instance);
3310         WR_TBOLT_IB_WRITE_SEQ(0xb, instance);
3311         WR_TBOLT_IB_WRITE_SEQ(2, instance);
3312         WR_TBOLT_IB_WRITE_SEQ(7, instance);
3313         WR_TBOLT_IB_WRITE_SEQ(0xd, instance);
3314         con_log(CL_ANN1, (CE_NOTE,
3315             "mrsas_tbolt_reset_ppc: magic number written "
3316             "to write sequence register"));
3332             "to write sequence register\n"));
3317         delay(100 * drv_usectohz(MILLISEC));
3318         status = RD_TBOLT_HOST_DIAG(instance);
3319         con_log(CL_ANN1, (CE_NOTE,
3320             "mrsas_tbolt_reset_ppc: READ HOSTDIAG SUCCESS "
3321             "to write sequence register"));
3337             "to write sequence register\n"));

3323         while (status & DIAG_TBOLT_RESET_ADAPTER) {
3324                 delay(100 * drv_usectohz(MILLISEC));
3325                 status = RD_TBOLT_HOST_DIAG(instance);
3326                 if (retry++ == 100) {
3327                         cmn_err(CE_WARN,
3328                             "mrsas_tbolt_reset_ppc:"
3329                             "resetadapter bit is set already "
3330                             "check retry count %d", retry);
3346                             "check retry count %d\n", retry);
3331                         return (DDI_FAILURE);
3332                 }
3333         }

3335         WR_TBOLT_HOST_DIAG(status | DIAG_TBOLT_RESET_ADAPTER, instance);
3336         delay(100 * drv_usectohz(MILLISEC));
```



```
3338         ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3339             (uint8_t *)((uintptr_t)(instance)->regmap +
3340             RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);

3342         while ((status & DIAG_TBOLT_RESET_ADAPTER)) {
3343                 delay(100 * drv_usectohz(MILLISEC));
3344                 ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3345                     (uint8_t *)((uintptr_t)(instance)->regmap +
3346                     RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);
3347                 if (retry++ == 100) {
3348                         /* Dont call kill adapter here */
3349                         /* RESET BIT ADAPTER is cleared by firmare */
3350                         /* mrsas_tbolt_kill_adapter(instance); */
3351                         cmn_err(CE_WARN,
3352                             "mr_sas %d: %s(): RESET FAILED; return failure!!!",
3353                             instance->instance, __func__);
3354                         return (DDI_FAILURE);
3355                 }
3356         }

3358         con_log(CL_ANN,
3359             (CE_NOTE, "mrsas_tbolt_reset_ppc: Adapter reset complete"));
3360         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3361             "Calling mfi_state_transition_to_ready"));

3363         abs_state = instance->func_ptr->read_fw_status_reg(instance);
3364         retry = 0;
3365         while ((abs_state <= MFI_STATE_FW_INIT) && (retry++ < 1000)) {
3366                 delay(100 * drv_usectohz(MILLISEC));
3367                 abs_state = instance->func_ptr->read_fw_status_reg(instance);
3368         }
3369         if (abs_state <= MFI_STATE_FW_INIT) {
3370                 cmn_err(CE_WARN,
3371                     "mrsas_tbolt_reset_ppc: firmware state < MFI_STATE_FW_INIT"
3372                     "state = 0x%x, RETRY RESET.", abs_state);
3388                     "state = 0x%x, RETRY RESET.\n", abs_state);
3373                 goto retry_reset;
3374         }

3376         /* Mark HBA as bad, if FW is fault after 3 continuous resets */
3377         if (mfi_state_transition_to_ready(instance) ||
3378             debug_tbolt_fw_faults_after_ocr_g == 1) {
3379                 cur_abs_reg_val =
3380                     instance->func_ptr->read_fw_status_reg(instance);
3381                 fw_state        = cur_abs_reg_val & MFI_STATE_MASK;

3383                 con_log(CL_ANN1, (CE_NOTE,
3384                     "mrsas_tbolt_reset_ppc :before fake: FW is not ready "
3385                     "FW state = 0x%x", fw_state));
3386                 if (debug_tbolt_fw_faults_after_ocr_g == 1)
3387                         fw_state = MFI_STATE_FAULT;

3389                 con_log(CL_ANN,
3390                     (CE_NOTE,  "mrsas_tbolt_reset_ppc : FW is not ready "
3391                     "FW state = 0x%x", fw_state));

3393                 if (fw_state == MFI_STATE_FAULT) {
3394                         /* increment the count */
3395                         instance->fw_fault_count_after_ocr++;
3396                         if (instance->fw_fault_count_after_ocr
3397                             < MAX_FW_RESET_COUNT) {
3398                                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3399                                     "FW is in fault after OCR count %d "
3400                                     "Retry Reset",
3401                                     instance->fw_fault_count_after_ocr);
```

```
3402                                goto retry_reset;

3404                        } else {
3405                                cmn_err(CE_WARN, "mrsas %d: %s:"
3406                                    "Max Reset Count exceeded >%d"
3407                                    "Mark HBA as bad, KILL adapter",
3408                                    instance->instance, __func__,
3409                                    MAX_FW_RESET_COUNT);

3411                                mrsas_tbolt_kill_adapter(instance);
3412                                return (DDI_FAILURE);
3413                        }
3414                }
3415        }

3417        /* reset the counter as FW is up after OCR */
3418        instance->fw_fault_count_after_ocr = 0;

3420        mrsas_reset_reply_desc(instance);


3423        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3424            "Calling mrsas_issue_init_mpi2"));
3425        abs_state = mrsas_issue_init_mpi2(instance);
3426        if (abs_state == (uint32_t)DDI_FAILURE) {
3427                cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3428                    "INIT failed Retrying Reset");
3429                goto retry_reset;
3430        }
3431        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3432            "mrsas_issue_init_mpi2 Done"));

3434        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3435            "Calling mrsas_print_pending_cmd"));
3451            "Calling mrsas_print_pending_cmd\n"));
3436        (void) mrsas_print_pending_cmds(instance);
3437        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3438            "mrsas_print_pending_cmd done"));
3454            "mrsas_print_pending_cmd done\n"));

3440        instance->func_ptr->enable_intr(instance);
3441        instance->fw_outstanding = 0;

3443        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3444            "Calling mrsas_issue_pending_cmds"));
3445        (void) mrsas_issue_pending_cmds(instance);
3446        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3447        "issue_pending_cmds done."));
3463        "issue_pending_cmds done.\n"));

3449        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3450            "Calling aen registration"));

3452        instance->aen_cmd->retry_count_for_ocr = 0;
3453        instance->aen_cmd->drv_pkt_time = 0;

3455        instance->func_ptr->issue_cmd(instance->aen_cmd, instance);

3457        con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag."));
3473        con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag.\n"));
3458        mutex_enter(&instance->ocr_flags_mtx);
3459        instance->adapterresetinprogress = 0;
3460        mutex_exit(&instance->ocr_flags_mtx);
3461        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3462            "adpterresetinprogress flag unset"));
```



```
3464        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc done"));
3480        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc done\n"));
3465        return (DDI_SUCCESS);

3467 }


3470 /*
3471  * mrsas_sync_map_info -        Returns FW's ld_map structure
3472  * @instance:                          Adapter soft state
3473  *
3474  * Issues an internal command (DCMD) to get the FW's controller PD
3475  * list structure.  This information is mainly used to find out SYSTEM
3476  * supported by the FW.
3477  */

3479 static int
3480 mrsas_tbolt_sync_map_info(struct mrsas_instance *instance)
3481 {
3482        int                     ret = 0, i;
3483        struct mrsas_cmd        *cmd = NULL;
3484        struct mrsas_dcmd_frame *dcmd;
3485        uint32_t size_sync_info, num_lds;
3486        LD_TARGET_SYNC *ci = NULL;
3487        MR_FW_RAID_MAP_ALL *map;
3488        MR_LD_RAID  *raid;
3489        LD_TARGET_SYNC *ld_sync;
3490        uint32_t ci_h = 0;
3491        uint32_t size_map_info;

3493        cmd = get_raid_msg_pkt(instance);

3495        if (cmd == NULL) {
3496                cmn_err(CE_WARN, "Failed to get a cmd from free-pool in "
3497                    "mrsas_tbolt_sync_map_info(). ");
3498                return (DDI_FAILURE);
3499        }

3501        /* Clear the frame buffer and assign back the context id */
3502        bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3518        (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3503        ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3504            cmd->index);
3505        bzero(cmd->scsi_io_request, sizeof (Mpi2RaidSCSIIORequest_t));


3508        map = instance->ld_map[instance->map_id & 1];

3510        num_lds = map->raidMap.ldCount;

3512        dcmd = &cmd->frame->dcmd;

3514        size_sync_info = sizeof (LD_TARGET_SYNC) * num_lds;

3516        con_log(CL_ANN, (CE_NOTE, "size_sync_info =0x%x ; ld count = 0x%x",
3532        con_log(CL_ANN, (CE_NOTE, "size_sync_info =0x%x ; ld count = 0x%x \n ",
3517            size_sync_info, num_lds));

3519        ci = (LD_TARGET_SYNC *)instance->ld_map[(instance->map_id - 1) & 1];

3521        bzero(ci, sizeof (MR_FW_RAID_MAP_ALL));
3537        (void) memset(ci, 0, sizeof (MR_FW_RAID_MAP_ALL));
3522        ci_h = instance->ld_map_phy[(instance->map_id - 1) & 1];

3524        bzero(dcmd->mbox.b, DCMD_MBOX_SZ);
3540        (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);
```

```
3526         ld_sync = (LD_TARGET_SYNC *)ci;

3528         for (i = 0; i < num_lds; i++, ld_sync++) {
3529                 raid = MR_LdRaidGet(i, map);

3531                 con_log(CL_ANN1,
3532                     (CE_NOTE, "i : 0x%x, Seq Num : 0x%x, Sync Reqd : 0x%x",
3548                     (CE_NOTE, "i : 0x%x, Seq Num : 0x%x, Sync Reqd : 0x%x\n",
3533                     i, raid->seqNum, raid->flags.ldSyncRequired));

3535                 ld_sync->ldTargetId = MR_GetLDTgtId(i, map);

3537                 con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, tgt : 0x%x",
3553                 con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, tgt : 0x%x \n",
3538                     i, ld_sync->ldTargetId));

3540                 ld_sync->seqNum = raid->seqNum;
3541         }


3544         size_map_info = sizeof (MR_FW_RAID_MAP) +
3545             (sizeof (MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));

3547         dcmd->cmd = MFI_CMD_OP_DCMD;
3548         dcmd->cmd_status = 0xFF;
3549         dcmd->sge_count = 1;
3550         dcmd->flags = MFI_FRAME_DIR_WRITE;
3551         dcmd->timeout = 0;
3552         dcmd->pad_0 = 0;
3553         dcmd->data_xfer_len = size_map_info;
3554         ASSERT(num_lds <= 255);
3555         dcmd->mbox.b[0] = (U8)num_lds;
3556         dcmd->mbox.b[1] = 1; /* Pend */
3557         dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
3558         dcmd->sgl.sge32[0].phys_addr = ci_h;
3559         dcmd->sgl.sge32[0].length = size_map_info;


3562         instance->map_update_cmd = cmd;
3563         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

3565         instance->func_ptr->issue_cmd(cmd, instance);

3567         instance->unroll.syncCmd = 1;
3568         con_log(CL_ANN1, (CE_NOTE, "sync cmd issued. [SMID]:%x", cmd->SMID));

3570         return (ret);
3571 }

3573 /*
3574  * abort_syncmap_cmd
3575  */
3576 int
3577 abort_syncmap_cmd(struct mrsas_instance *instance,
3578     struct mrsas_cmd *cmd_to_abort)
3579 {
3580         int     ret = 0;

3582         struct mrsas_cmd                *cmd;
3583         struct mrsas_abort_frame        *abort_fr;

3585         con_log(CL_ANN1, (CE_NOTE, "chkpnt: abort_ldsync:%d", __LINE__));

3587         cmd = get_raid_msg_mfi_pkt(instance);
```



```
3589         if (!cmd) {
3590                 cmn_err(CE_WARN,
3591                     "Failed to get a cmd from free-pool abort_syncmap_cmd().");
3592                 return (DDI_FAILURE);
3593         }
3594         /* Clear the frame buffer and assign back the context id */
3595         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3611         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3596         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3597             cmd->index);

3599         abort_fr = &cmd->frame->abort;

3601         /* prepare and issue the abort frame */
3602         ddi_put8(cmd->frame_dma_obj.acc_handle,
3603             &abort_fr->cmd, MFI_CMD_OP_ABORT);
3604         ddi_put8(cmd->frame_dma_obj.acc_handle, &abort_fr->cmd_status,
3605             MFI_CMD_STATUS_SYNC_MODE);
3606         ddi_put16(cmd->frame_dma_obj.acc_handle, &abort_fr->flags, 0);
3607         ddi_put32(cmd->frame_dma_obj.acc_handle, &abort_fr->abort_context,
3608             cmd_to_abort->index);
3609         ddi_put32(cmd->frame_dma_obj.acc_handle,
3610             &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3611         ddi_put32(cmd->frame_dma_obj.acc_handle,
3612             &abort_fr->abort_mfi_phys_addr_hi, 0);

3614         cmd->frame_count = 1;

3616         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

3618         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3619                 con_log(CL_ANN1, (CE_WARN,
3620                     "abort_ldsync_cmd: issue_cmd_in_poll_mode failed"));
3621                 ret = -1;
3622         } else {
3623                 ret = 0;
3624         }

3626         return_raid_msg_mfi_pkt(instance, cmd);

3628         atomic_add_16(&instance->fw_outstanding, (-1));

3630         return (ret);
3631 }


3634 #ifdef PDSUPPORT
3635 int
3636 mrsas_tbolt_config_pd(struct mrsas_instance *instance, uint16_t tgt,
3637     uint8_t lun, dev_info_t **ldip)
3638 {
3639         struct scsi_device *sd;
3640         dev_info_t *child;
3641         int rval, dtype;
3642         struct mrsas_tbolt_pd_info *pds = NULL;

3644         con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_config_pd: t = %d l = %d",
3645             tgt, lun));

3647         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
3648                 if (ldip) {
3649                         *ldip = child;
3650                 }
3651                 if (instance->mr_tbolt_pd_list[tgt].flag != MRDRV_TGT_VALID) {
3652                         rval = mrsas_service_evt(instance, tgt, 1,
3653                             MRSAS_EVT_UNCONFIG_TGT, NULL);
```

```
3654                        con_log(CL_ANN1, (CE_WARN,
3655                            "mr_sas:DELETING STALE ENTRY  rval = %d "
3656                            "tgt id = %d", rval, tgt));
3672                            "tgt id = %d ", rval, tgt));
3657                        return (NDI_FAILURE);
3658                }
3659                return (NDI_SUCCESS);
3660        }

3662        pds = (struct mrsas_tbolt_pd_info *)
3663            kmem_zalloc(sizeof (struct mrsas_tbolt_pd_info), KM_SLEEP);
3664        mrsas_tbolt_get_pd_info(instance, pds, tgt);
3665        dtype = pds->scsiDevType;

3667        /* Check for Disk */
3668        if ((dtype == DTYPE_DIRECT)) {
3669                if ((dtype == DTYPE_DIRECT) &&
3670                    (LE_16(pds->fwState) != PD_SYSTEM)) {
3671                        kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3672                        return (NDI_FAILURE);
3673                }
3674                sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
3675                sd->sd_address.a_hba_tran = instance->tran;
3676                sd->sd_address.a_target = (uint16_t)tgt;
3677                sd->sd_address.a_lun = (uint8_t)lun;

3679                if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS) {
3680                        rval = mrsas_config_scsi_device(instance, sd, ldip);
3681                        con_log(CL_DLEVEL1, (CE_NOTE,
3682                            "Phys. device found: tgt %d dtype %d: %s",
3683                            tgt, dtype, sd->sd_inq->inq_vid));
3684                } else {
3685                        rval = NDI_FAILURE;
3686                        con_log(CL_DLEVEL1, (CE_NOTE, "Phys. device Not found "
3687                            "scsi_hba_probe Failed: tgt %d dtype %d: %s",
3688                            tgt, dtype, sd->sd_inq->inq_vid));
3689                }

3691                /* sd_unprobe is blank now. Free buffer manually */
3692                if (sd->sd_inq) {
3693                        kmem_free(sd->sd_inq, SUN_INQSIZE);
3694                        sd->sd_inq = (struct scsi_inquiry *)NULL;
3695                }
3696                kmem_free(sd, sizeof (struct scsi_device));
3697                rval = NDI_SUCCESS;
3698        } else {
3699                con_log(CL_ANN1, (CE_NOTE,
3700                    "Device not supported: tgt %d lun %d dtype %d",
3701                    tgt, lun, dtype));
3702                rval = NDI_FAILURE;
3703        }

3705        kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3706        con_log(CL_ANN1, (CE_NOTE, "mrsas_config_pd: return rval = %d",
3707            rval));
3708        return (rval);
3709 }

3711 static void
3712 mrsas_tbolt_get_pd_info(struct mrsas_instance *instance,
3713     struct mrsas_tbolt_pd_info *pds, int tgt)
3714 {
3715        struct mrsas_cmd        *cmd;
3716        struct mrsas_dcmd_frame *dcmd;
3717        dma_obj_t               dcmd_dma_obj;
```



```
3719        cmd = get_raid_msg_pkt(instance);

3721        if (!cmd) {
3722                con_log(CL_ANN1,
3723                    (CE_WARN, "Failed to get a cmd for get pd info"));
3724                return;
3725        }

3727        /* Clear the frame buffer and assign back the context id */
3728        bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3744        (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3729        ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3730            cmd->index);


3733        dcmd = &cmd->frame->dcmd;
3734        dcmd_dma_obj.size = sizeof (struct mrsas_tbolt_pd_info);
3735        dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
3736        dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xffffffff;
3737        dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xffffffff;
3738        dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
3739        dcmd_dma_obj.dma_attr.dma_attr_align = 1;

3741        (void) mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
3742            DDI_STRUCTURE_LE_ACC);
3743        bzero(dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info));
3744        bzero(dcmd->mbox.b, 12);
3759        (void) memset(dcmd_dma_obj.buffer, 0,
3760            sizeof (struct mrsas_tbolt_pd_info));
3761        (void) memset(dcmd->mbox.b, 0, 12);
3745        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3746        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
3747        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3748        ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3749            MFI_FRAME_DIR_READ);
3750        ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3751        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3752            sizeof (struct mrsas_tbolt_pd_info));
3753        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3754            MR_DCMD_PD_GET_INFO);
3755        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], tgt);
3756        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3757            sizeof (struct mrsas_tbolt_pd_info));
3758        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3759            dcmd_dma_obj.dma_cookie[0].dmac_address);

3761        cmd->sync_cmd = MRSAS_TRUE;
3762        cmd->frame_count = 1;

3764        if (instance->tbolt) {
3765                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3766        }

3768        instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd);

3770        ddi_rep_get8(cmd->frame_dma_obj.acc_handle, (uint8_t *)pds,
3771            (uint8_t *)dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info),
3772            DDI_DEV_AUTOINCR);
3773        (void) mrsas_free_dma_obj(instance, dcmd_dma_obj);
3774        return_raid_msg_pkt(instance, cmd);
3775 }
```

***unchanged_portion_omitted_***