```
*********************************************************
   17103 Tue Aug 14 17:24:55 2012
new/./fusion.h
*********************************************************
______unchanged_portion_omitted_
```

****
13466 Tue Aug 14 17:24:55 2012
new/./ld_pd_map.c
****
_____***unchanged_portion_omitted_***

```
 116 /*
 117  * This function will validate Map info data provided by FW
 118  */
 119 U8 MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo)
 120 {
 121         MR_FW_RAID_MAP *pFwRaidMap = &map->raidMap;


 124         if (pFwRaidMap->totalSize !=
 125             (sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP) +
 126             (sizeof (MR_LD_SPAN_MAP) * pFwRaidMap->ldCount))) {

 128                 con_log(CL_ANN1, (CE_NOTE,\
 129                     "map info structure size 0x%x\
 130                     is not matching with ld count\n",\
 131                     (int)((sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP)) +\
 131                     ((sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP)) +\
 132                     (sizeof (MR_LD_SPAN_MAP) * pFwRaidMap->ldCount))));

 134                 con_log(CL_ANN1, (CE_NOTE, "span map 0x%x total size 0x%x\n",\
 135                     (int)sizeof (MR_LD_SPAN_MAP), pFwRaidMap->totalSize));
 135                     sizeof (MR_LD_SPAN_MAP), pFwRaidMap->totalSize));

 137                 return (0);
 138         }

 140         mr_update_load_balance_params(map, lbInfo);

 142         return (1);
 143 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    7031 Tue Aug 14 17:24:55 2012
new/./ld_pd_map.h
**********************************************************
______unchanged_portion_omitted_
```

```
**********************************************************
  208900 Tue Aug 14 17:24:55 2012
new/./mr_sas.c
**********************************************************
______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
 */
/* #if       __SunOS_5_11 */
#if 0
#if       __SunOS_5_11
#define   DDI_PM_RESUME DDI_PM_RESUME_OBSOLETE
#define   DDI_PM_SUSPEND DDI_PM_SUSPEND_OBSOLETE
#endif // __SunOS_5_11
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);
                        }

                        bzero((caddr_t)instance,
                            sizeof (struct mrsas_instance));

                        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 (instance->pci_handle == NULL) {
                                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,
```

```
408                                     "mr_sas: failed to get registers.");

410                             pci_config_teardown(&instance->pci_handle);
411                             ddi_soft_state_free(mrsas_state, instance_no);
412                             return (DDI_FAILURE);
413                     }

415                     vendor_id = pci_config_get16(instance->pci_handle,
416                         PCI_CONF_VENID);
417                     device_id = pci_config_get16(instance->pci_handle,
418                         PCI_CONF_DEVID);

420                     subsysvid = pci_config_get16(instance->pci_handle,
421                         PCI_CONF_SUBVENID);
422                     subsysid = pci_config_get16(instance->pci_handle,
423                         PCI_CONF_SUBSYSID);

425                     pci_config_put16(instance->pci_handle, PCI_CONF_COMM,
426                         (pci_config_get16(instance->pci_handle,
427                         PCI_CONF_COMM) | PCI_COMM_ME));
428                     irq = pci_config_get8(instance->pci_handle,
429                         PCI_CONF_ILINE);

431                     con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
432                         "0x%x:0x%x 0x%x:0x%x, irq:%d drv-ver:%s",
433                         instance_no, vendor_id, device_id, subsysvid,
434                         subsysid, irq, MRSAS_VERSION));

436                     /* enable bus-mastering */
437                     command = pci_config_get16(instance->pci_handle,
438                         PCI_CONF_COMM);

440                     if (!(command & PCI_COMM_ME)) {
441                             command |= PCI_COMM_ME;

443                             pci_config_put16(instance->pci_handle,
444                                 PCI_CONF_COMM, command);

446                             con_log(CL_ANN, (CE_CONT, "mr_sas%d: "
447                                 "enable bus-mastering", instance_no));
448                     } else {
449                             con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
450                             "bus-mastering already set", instance_no));
451                     }

453                     /* initialize function pointers */
454                    switch(device_id) {
455                             case PCI_DEVICE_ID_LSI_TBOLT:
456                             case PCI_DEVICE_ID_LSI_INVADER:
457                                     con_log(CL_ANN, (CE_NOTE,
458                                         "mr_sas: 2208 T.B. device detected")

460                                     instance->func_ptr = &mrsas_function_tem
461                                     instance->tbolt = 1;
462                                     break;

464                             case PCI_DEVICE_ID_LSI_2108VDE:
465                             case PCI_DEVICE_ID_LSI_2108V:
466                                     con_log(CL_ANN, (CE_NOTE,
467                                         "mr_sas: 2108 Liberator device detec

469                                     instance->func_ptr = &mrsas_function_tem
470                                     break;

472                             default:
473                                     cmn_err(CE_WARN,
```



```
474                                         "mr_sas: Invalid device detected");

476                                 pci_config_teardown(&instance->pci_handl
477                                 ddi_soft_state_free(mrsas_state, instanc
478                                 return (DDI_FAILURE);

480                     }

482                     instance->baseaddress = pci_config_get32(
483                         instance->pci_handle, PCI_CONF_BASE0);
484                     instance->baseaddress &= 0x0fffc;

486                     instance->dip           = dip;
487                     instance->vendor_id     = vendor_id;
488                     instance->device_id     = device_id;
489                     instance->subsysvid     = subsysvid;
490                     instance->subsysid      = subsysid;
491                     instance->instance      = instance_no;


494                     /* Setup register map */
495                     if ((ddi_dev_regsize(instance->dip,
496                         REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
497                         reglength < MINIMUM_MFI_MEM_SZ) {
498                             goto fail_attach;
499                     }
500                     if (reglength > DEFAULT_MFI_MEM_SZ) {
501                             reglength = DEFAULT_MFI_MEM_SZ;
502                             con_log(CL_DLEVEL1, (CE_NOTE,
503                                 "mr_sas: register length to map is "
504                                 "0x%lx bytes", reglength));
505                     }
506                     if (ddi_regs_map_setup(instance->dip,
507                         REGISTER_SET_IO_2108, &instance->regmap, 0,
508                         reglength, &endian_attr, &instance->regmap_handle)
509                         != DDI_SUCCESS) {
510                             cmn_err(CE_WARN,
511                                 "mr_sas: couldn't map control registers");
512                             goto fail_attach;
513                     }
514                     if (instance->regmap_handle == NULL) {
515                             cmn_err(CE_WARN,
516                                 "mr_sas: couldn't map control registers");
517                             goto fail_attach;
518                     }

520                     instance->unroll.regs = 1;

522                     /*
523                      * Disable Interrupt Now.
524                      * Setup Software interrupt
525                      */
526                     instance->func_ptr->disable_intr(instance);

528                     if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
529                         "mrsas-enable-msi", &data) == DDI_SUCCESS) {
530                             if (strncmp(data, "no", 3) == 0) {
531                                     msi_enable = 0;
532                                     con_log(CL_ANN1, (CE_WARN,
533                                         "msi_enable = %d disabled",
534                                         msi_enable));
535                             }
536                             ddi_prop_free(data);
537                     }

539                     con_log(CL_DLEVEL1, (CE_NOTE, "msi_enable = %d", msi_ena
```

```
541                     if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
542                         "mrsas-enable-fp", &data) == DDI_SUCCESS) {
543                             if (strncmp(data, "no", 3) == 0) {
544                                     enable_fp = 0;
545                                     cmn_err(CE_NOTE,
546                                         "enable_fp = %d, Fast-Path disabled.
547                                         enable_fp);
548                             }

550                             ddi_prop_free(data);
551                     }

553                     cmn_err(CE_NOTE, "enable_fp = %d\n", enable_fp);

555                     /* Check for all supported interrupt types */
556                     if (ddi_intr_get_supported_types(
557                         dip, &intr_types) != DDI_SUCCESS) {
558                             cmn_err(CE_WARN,
559                                 "ddi_intr_get_supported_types() failed");
560                             goto fail_attach;
561                     }

563                     con_log(CL_DLEVEL1, (CE_NOTE,
564                         "ddi_intr_get_supported_types() ret: 0x%x",
565                         intr_types));

567                     /* Initialize and Setup Interrupt handler */
568                     if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
569                             if (mrsas_add_intrs(instance,
570                                 DDI_INTR_TYPE_MSIX) != DDI_SUCCESS) {
571                                     cmn_err(CE_WARN,
572                                         "MSIX interrupt query failed");
573                                     goto fail_attach;
574                             }
575                             instance->intr_type = DDI_INTR_TYPE_MSIX;
576                     } else if (msi_enable && (intr_types &
577                         DDI_INTR_TYPE_MSI)) {
578                             if (mrsas_add_intrs(instance,
579                                 DDI_INTR_TYPE_MSI) != DDI_SUCCESS) {
580                                     cmn_err(CE_WARN,
581                                         "MSI interrupt query failed");
582                                     goto fail_attach;
583                             }
584                             instance->intr_type = DDI_INTR_TYPE_MSI;
585                     } else if (intr_types & DDI_INTR_TYPE_FIXED) {
586                             msi_enable = 0;
587                             if (mrsas_add_intrs(instance,
588                                 DDI_INTR_TYPE_FIXED) != DDI_SUCCESS) {
589                                     cmn_err(CE_WARN,
590                                         "FIXED interrupt query failed");
591                                     goto fail_attach;
592                             }
593                             instance->intr_type = DDI_INTR_TYPE_FIXED;
594                     } else {
595                             cmn_err(CE_WARN, "Device cannot "
596                                 "suppport either FIXED or MSI/X "
597                                 "interrupts");
598                             goto fail_attach;
599                     }

601                     instance->unroll.intr = 1;


604                     /* setup the mfi based low level driver */
605                     if (mrsas_init_adapter(instance) != DDI_SUCCESS) {
```



```
606                             cmn_err(CE_WARN, "mr_sas: "
607                             "could not initialize the low level driver");

609                             goto fail_attach;
610                     }

612                     /* Initialize all Mutex */
613                     INIT_LIST_HEAD(&instance->completed_pool_list);
614                     mutex_init(&instance->completed_pool_mtx,
615                         "completed_pool_mtx", MUTEX_DRIVER,
616                         DDI_INTR_PRI(instance->intr_pri));
617
618                     mutex_init(&instance->sync_map_mtx,
619                         "sync_map_mtx", MUTEX_DRIVER,
620                         DDI_INTR_PRI(instance->intr_pri));
621
622                     mutex_init(&instance->app_cmd_pool_mtx,
623                         "app_cmd_pool_mtx", MUTEX_DRIVER,
624                         DDI_INTR_PRI(instance->intr_pri));

626                     mutex_init(&instance->config_dev_mtx, "config_dev_mtx",
627                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

629                     mutex_init(&instance->cmd_pend_mtx, "cmd_pend_mtx",
630                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

632                     mutex_init(&instance->ocr_flags_mtx, "ocr_flags_mtx",
633                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

635                     mutex_init(&instance->int_cmd_mtx, "int_cmd_mtx",
636                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
637                     cv_init(&instance->int_cmd_cv, NULL, CV_DRIVER, NULL);

639                     mutex_init(&instance->cmd_pool_mtx, "cmd_pool_mtx",
640                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

642                     mutex_init(&instance->reg_write_mtx,"reg_write_mtx",
643                             MUTEX_DRIVER,DDI_INTR_PRI(instance->intr_pri));

645                     if (instance->tbolt) {
646                             mutex_init(&instance->cmd_app_pool_mtx,
647                                 "cmd_app_pool_mtx", MUTEX_DRIVER,
648                                 DDI_INTR_PRI(instance->intr_pri));

650                             mutex_init(&instance->chip_mtx,
651                                 "chip_mtx", MUTEX_DRIVER,
652                                 DDI_INTR_PRI(instance->intr_pri));

654                     }

656                     instance->unroll.mutexs = 1;

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

660                     /* Register our soft-isr for highlevel interrupts. */
661                     instance->isr_level = instance->intr_pri;
662                     if (!(instance->tbolt)) {
663                             if (instance->isr_level == HIGH_LEVEL_INTR) {
664                                     if (ddi_add_softintr(dip,
665                                         DDI_SOFTINT_HIGH,
666                                         &instance->soft_intr_id,
667                                         NULL, NULL, mrsas_softintr,
668                                         (caddr_t)instance) !=
669                                         DDI_SUCCESS) {
670                                             cmn_err(CE_WARN,
671                                                 "Software ISR "
```

```
672                                         "did not register");

674                                     goto fail_attach;
675                                 }

677                                 instance->unroll.soft_isr = 1;

679                         }
680                 }

682                 instance->softint_running = 0;

684                 /* Allocate a transport structure */
685                 tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

687                 if (tran == NULL) {
688                         cmn_err(CE_WARN,
689                             "scsi_hba_tran_alloc failed");
690                         goto fail_attach;
691                 }

693                 instance->tran = tran;
694                 instance->unroll.tran = 1;

696                 tran->tran_hba_private  = instance;
697                 tran->tran_tgt_init     = mrsas_tran_tgt_init;
698                 tran->tran_tgt_probe    = scsi_hba_probe;
699                 tran->tran_tgt_free     = mrsas_tran_tgt_free;
700                 if (instance->tbolt) {
701                         tran->tran_init_pkt     =
702                             mrsas_tbolt_tran_init_pkt;
703                         tran->tran_start        =
704                             mrsas_tbolt_tran_start;
705                 } else {
706                         tran->tran_init_pkt     = mrsas_tran_init_pkt;
707                         tran->tran_start        = mrsas_tran_start;
708                 }
709                 tran->tran_abort        = mrsas_tran_abort;
710                 tran->tran_reset        = mrsas_tran_reset;
711                 tran->tran_getcap       = mrsas_tran_getcap;
712                 tran->tran_setcap       = mrsas_tran_setcap;
713                 tran->tran_destroy_pkt  = mrsas_tran_destroy_pkt;
714                 tran->tran_dmafree      = mrsas_tran_dmafree;
715                 tran->tran_sync_pkt     = mrsas_tran_sync_pkt;
716                 tran->tran_quiesce      = mrsas_tran_quiesce;
717                 tran->tran_unquiesce    = mrsas_tran_unquiesce;
718                 tran->tran_bus_config   = mrsas_tran_bus_config;

720                 if (mrsas_relaxed_ordering)
721                         mrsas_generic_dma_attr.dma_attr_flags |=
722                             DDI_DMA_RELAXED_ORDERING;


725                 tran_dma_attr = mrsas_generic_dma_attr;
726                 tran_dma_attr.dma_attr_sgllen = instance->max_num_sge;

728                 /* Attach this instance of the hba */
729                 if (scsi_hba_attach_setup(dip, &tran_dma_attr, tran, 0)
730                     != DDI_SUCCESS) {
731                         cmn_err(CE_WARN,
732                             "scsi_hba_attach failed");

734                         goto fail_attach;
735                 }
736                 instance->unroll.tranSetup = 1;
737                 con_log(CL_ANN1, (CE_CONT,
```



```
738                         "scsi_hba_attach_setup()  done."));


741                 /* create devctl node for cfgadm command */
742                 if (ddi_create_minor_node(dip, "devctl",
743                     S_IFCHR, INST2DEVCTL(instance_no),
744                     DDI_NT_SCSI_NEXUS, 0) == DDI_FAILURE) {
745                         cmn_err(CE_WARN,
746                             "mr_sas: failed to create devctl node.");

748                         goto fail_attach;
749                 }

751                 instance->unroll.devctl = 1;

753                 /* create scsi node for cfgadm command */
754                 if (ddi_create_minor_node(dip, "scsi", S_IFCHR,
755                     INST2SCSI(instance_no),
756                     DDI_NT_SCSI_ATTACHMENT_POINT, 0) ==
757                     DDI_FAILURE) {
758                         cmn_err(CE_WARN,
759                             "mr_sas: failed to create scsi node.");

761                         goto fail_attach;
762                 }

764                 instance->unroll.scsictl = 1;

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

769                 /*
770                  * Create a node for applications
771                  * for issuing ioctl to the driver.
772                  */
773                 if (ddi_create_minor_node(dip, instance->iocnode,
774                     S_IFCHR, INST2LSIRDCTL(instance_no),
775                     DDI_PSEUDO, 0) == DDI_FAILURE) {
776                         cmn_err(CE_WARN,
777                             "mr_sas: failed to create ioctl node.");

779                         goto fail_attach;
780                 }

782                 instance->unroll.ioctl = 1;

784                 /* Create a taskq to handle dr events */
785                 if ((instance->taskq = ddi_taskq_create(dip,
786                     "mrsas_dr_taskq", 1,
787                     TASKQ_DEFAULTPRI, 0)) == NULL) {
788                         cmn_err(CE_WARN,
789                             "mr_sas: failed to create taskq ");
790                         instance->taskq = NULL;
791                         goto fail_attach;
792                 }
793                 instance->unroll.taskq = 1;
794                 con_log(CL_ANN1, (CE_CONT,
795                         "ddi_taskq_create()  done."));

797                 /* enable interrupt */
798                 instance->func_ptr->enable_intr(instance);

800                 /* initiate AEN */
801                 if (start_mfi_aen(instance)) {
802                         cmn_err(CE_WARN,
803                             "mr_sas: failed to initiate AEN.");
```

```
804                             goto fail_attach;
805                     }
806                     instance->unroll.aenPend = 1;
807                     con_log(CL_ANN1, (CE_CONT,
808                         "AEN started for instance %d.", instance_no));
809
810                     /* Finally! We are on the air.  */
811                     ddi_report_dev(dip);
812
813                     instance->mr_ld_list =
814                         kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
815                         KM_SLEEP);
816                     if (instance->mr_ld_list == NULL) {
817                             cmn_err(CE_WARN,
818                                 "mr_sas attach(): failed to allocate ld_list
819                             goto fail_attach;
820                     }
821                     instance->unroll.ldlist_buff = 1;
822
823 #ifdef PDSUPPORT
824                     if(instance->tbolt) {
825                             instance->mr_tbolt_pd_max = MRSAS_TBOLT_PD_TGT_M
826                             instance->mr_tbolt_pd_list =
827                                 kmem_zalloc(MRSAS_TBOLT_GET_PD_MAX(instance)
828                                 * sizeof (struct mrsas_tbolt_pd), KM_SLEEP);
829                             ASSERT(instance->mr_tbolt_pd_list);
830                             for (i = 0; i < instance->mr_tbolt_pd_max; i++)
831                                     instance->mr_tbolt_pd_list[i].lun_type =
832                                         MRSAS_TBOLT_PD_LUN;
833                                     instance->mr_tbolt_pd_list[i].dev_id =
834                                         (uint8_t)i;
835                             }
836
837                             instance->unroll.pdlist_buff = 1;
838                     }
839 #endif
840                     break;
841             case DDI_PM_RESUME:
842                     con_log(CL_ANN, (CE_NOTE,
843                         "mr_sas: DDI_PM_RESUME"));
844                     break;
845             case DDI_RESUME:
846                     con_log(CL_ANN, (CE_NOTE,
847                         "mr_sas: DDI_RESUME"));
848                     break;
849             default:
850                     con_log(CL_ANN, (CE_WARN,
851                         "mr_sas: invalid attach cmd=%x", cmd));
852                     return (DDI_FAILURE);
853         }
854
855
856         cmn_err(CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d", instan
857         return (DDI_SUCCESS);
858
859 fail_attach:
860
861         mrsas_undo_resources(dip, instance);
862
863         pci_config_teardown(&instance->pci_handle);
864         ddi_soft_state_free(mrsas_state, instance_no);
865
866         con_log(CL_ANN, (CE_WARN,
867             "mr_sas: return failure from mrsas_attach"));
868
869         cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d", instan
```



```
871         return (DDI_FAILURE);
872 }
_____unchanged_portion_omitted_

2601 void
2602 mrsas_print_cmd_details(struct mrsas_instance *instance,
2603                                         struct mrsas_cmd *cmd, int detail )
2604 {
2605         struct scsi_pkt *pkt = cmd->pkt;
2606         Mpi2RaidSCSIIORequest_t *scsi_io = cmd->scsi_io_request;
2607         MPI2_SCSI_IO_VENDOR_UNIQUE *raidContext;
2608         uint8_t *cdb_p;
2609         char str[100], *strp;
2610         int i, j, len;
2611         int saved_level;
2612
2613
2614         if (detail == 0xDD) {
2615                 saved_level = debug_level_g;
2616                 debug_level_g = CL_ANN1;
2617         }
2618
2619
2620         if (instance->tbolt) {
2621                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p cmd->index
2622                         (void *)cmd, cmd->index, cmd->SMID, cmd->drv_pkt_time))
2623         }
2624         else {
2625                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p cmd->index
2626                         (void *)cmd, cmd->index, cmd->drv_pkt_time));
2627         }
2628
2629         if(pkt) {
2630                 con_log(CL_ANN1, (CE_CONT, "scsi_pkt CDB[0]=0x%x",
2631                       pkt->pkt_cdbp[0]));
2632         }else {
2633                 con_log(CL_ANN1, (CE_CONT, "NO-PKT"));
2634         }
2635
2636         if((detail==0xDD) && instance->tbolt) {
2637                 con_log(CL_ANN1, (CE_CONT, "RAID_SCSI_IO_REQUEST\n"));
2638                 con_log(CL_ANN1, (CE_CONT, "DevHandle=0x%X Function=0x%X IoFlags
2639                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2640                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2641                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2642                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2643                         ddi_get32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2644
2645                 for(i=0; i < 32; i++)
2646                         con_log(CL_ANN1, (CE_CONT, "CDB[%d]=0x%x ",i,
2647                                 ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_h
2648
2649                 con_log(CL_ANN1, (CE_CONT, "RAID-CONTEXT\n"));
2650                 con_log(CL_ANN1, (CE_CONT, "status=0x%X extStatus=0x%X ldTargetI
2651                         "regLockFlags=0x%X RAIDFlags=0x%X regLockRowLBA=0x%" PRI
2650                         "regLockFlags=0x%X RAIDFlags=0x%X regLockRowLBA=0x%lX re
2652                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2653                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2654                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2655                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2656                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2657                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2658                         ddi_get64(instance->mpi2_frame_pool_dma_obj.acc_handle,
2659                         ddi_get32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2660                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
```

```
2663         }

2665         if (detail == 0xDD) {
2666                 debug_level_g = saved_level;
2667         }

2669         return;
2670 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   60029 Tue Aug 14 17:24:55 2012
new/./mr_sas.h
**********************************************************
______unchanged_portion_omitted_

 612 /*
 613  * ### Helper routines ###
 614  */

 616 /*
 617  * con_log() - console log routine
 618  * @param level         : indicates the severity of the message.
 619  * @fparam mt           : format string
 620  *
 621  * con_log displays the error messages on the console based on the current
 622  * debug level. Also it attaches the appropriate kernel severity level with
 623  * the message.
 624  *
 625  *
 626  * console messages debug levels
 627  */
 628 #define CL_NONE         0       /* No debug information */
 629 #define CL_ANN          1       /* print unconditionally, announcements */
 630 #define CL_ANN1         2       /* No o/p  */
 631 #define CL_DLEVEL1      3       /* debug level 1, informative */
 632 #define CL_DLEVEL2      4       /* debug level 2, verbose */
 633 #define CL_DLEVEL3      5       /* debug level 3, very verbose */

 635 #ifdef __SUNPRO_C
 636 #define __func__ ""
 636 #define __func__        __FUNCTION__    //""
 637 #endif

 639 #define con_log(level, fmt) { if (debug_level_g >= level) cmn_err fmt; }

 641 /*
 642  * ### SCSA definitions ###
 643  */
 644 #define PKT2TGT(pkt)    ((pkt)->pkt_address.a_target)
 645 #define PKT2LUN(pkt)    ((pkt)->pkt_address.a_lun)
 646 #define PKT2TRAN(pkt)   ((pkt)->pkt_adress.a_hba_tran)
 647 #define ADDR2TRAN(ap)   ((ap)->a_hba_tran)

 649 #define TRAN2MR(tran)   (struct mrsas_instance *)(tran)->tran_hba_private)
 650 #define ADDR2MR(ap)     (TRAN2MR(ADDR2TRAN(ap))

 652 #define PKT2CMD(pkt)    ((struct scsa_cmd *)(pkt)->pkt_ha_private)
 653 #define CMD2PKT(sp)     ((sp)->cmd_pkt)
 654 #define PKT2REQ(pkt)    (&(PKT2CMD(pkt)->request))

 656 #define CMD2ADDR(cmd)   (&CMD2PKT(cmd)->pkt_address)
 657 #define CMD2TRAN(cmd)   (CMD2PKT(cmd)->pkt_address.a_hba_tran)
 658 #define CMD2MR(cmd)     (TRAN2MR(CMD2TRAN(cmd)))

 660 #define CFLAG_DMAVALID          0x0001  /* requires a dma operation */
 661 #define CFLAG_DMASEND           0x0002  /* Transfer from the device */
 662 #define CFLAG_CONSISTENT        0x0040  /* consistent data transfer */

 664 /*
 665  * ### Data structures for ioctl inteface and internal commands ###
 666  */

 668 /*
 669  * Data direction flags
 670  */
```



```
 671 #define UIOC_RD         0x00001
 672 #define UIOC_WR         0x00002

 674 #define SCP2HOST(scp)           (scp)->device->host     /* to host */
 675 #define SCP2HOSTDATA(scp)       SCP2HOST(scp)->hostdata /* to soft state */
 676 #define SCP2CHANNEL(scp)        (scp)->device->channel  /* to channel */
 677 #define SCP2TARGET(scp)         (scp)->device->id       /* to target */
 678 #define SCP2LUN(scp)            (scp)->device->lun      /* to LUN */

 680 #define SCSIHOST2ADAP(host)     (((caddr_t *)(host->hostdata))[0])
 681 #define SCP2ADAPTER(scp)                                \
 682         (struct mrsas_instance *)SCSIHOST2ADAP(SCP2HOST(scp))

 684 #define MRDRV_IS_LOGICAL_SCSA(instance, acmd)           \
 685         (acmd->device_id < MRDRV_MAX_LD) ? 1 : 0
 686 #define MRDRV_IS_LOGICAL(ap)                            \
 687         ((ap->a_target < MRDRV_MAX_LD) && (ap->a_lun == 0)) ? 1 : 0
 688 #define MAP_DEVICE_ID(instance, ap)                     \
 689         (ap->a_target)

 691 #define HIGH_LEVEL_INTR                 1
 692 #define NORMAL_LEVEL_INTR               0

 694 #define         IO_TIMEOUT_VAL          0
 695 #define         IO_RETRY_COUNT          3
 696 #define         MAX_FW_RESET_COUNT      3
 697 /*
 698  * scsa_cmd  - Per-command mr private data
 699  * @param cmd_dmahandle         :  dma handle
 700  * @param cmd_dmacookies        :  current dma cookies
 701  * @param cmd_pkt               :  scsi_pkt reference
 702  * @param cmd_dmacount          :  dma count
 703  * @param cmd_cookie            :  next cookie
 704  * @param cmd_ncookies          :  cookies per window
 705  * @param cmd_cookiecnt         :  cookies per sub-win
 706  * @param cmd_nwin              :  number of dma windows
 707  * @param cmd_curwin            :  current dma window
 708  * @param cmd_dma_offset        :  current window offset
 709  * @param cmd_dma_len           :  current window length
 710  * @param cmd_flags             :  private flags
 711  * @param cmd_cdblen            :  length of cdb
 712  * @param cmd_scblen            :  length of scb
 713  * @param cmd_buf               :  command buffer
 714  * @param channel               :  channel for scsi sub-system
 715  * @param target                :  target for scsi sub-system
 716  * @param lun                   :  LUN for scsi sub-system
 717  *
 718  * - Allocated at same time as scsi_pkt by scsi_hba_pkt_alloc(9E)
 719  * - Pointed to by pkt_ha_private field in scsi_pkt
 720  */
 721 struct scsa_cmd {
 722         ddi_dma_handle_t        cmd_dmahandle;
 723         ddi_dma_cookie_t        cmd_dmacookies[MRSAS_MAX_SGE_CNT];
 724         struct scsi_pkt         *cmd_pkt;
 725         ulong_t                 cmd_dmacount;
 726         uint_t                  cmd_cookie;
 727         uint_t                  cmd_ncookies;
 728         uint_t                  cmd_cookiecnt;
 729         uint_t                  cmd_nwin;
 730         uint_t                  cmd_curwin;
 731         off_t                   cmd_dma_offset;
 732         ulong_t                 cmd_dma_len;
 733         ulong_t                 cmd_flags;
 734         uint_t                  cmd_cdblen;
 735         uint_t                  cmd_scblen;
 736         struct buf              *cmd_buf;
```

```
 737         ushort_t                device_id;
 738         uchar_t                 islogical;
 739         uchar_t                 lun;
 740         struct mrsas_device     *mrsas_dev;
 741 };
______unchanged_portion_omitted_
```

```
**********************************************************
    4514 Tue Aug 14 17:24:56 2012
new/./mr_sas_list.h
**********************************************************
______unchanged_portion_omitted_
```

```
**********************************************************
  109524 Tue Aug 14 17:24:56 2012
new/./mr_sas_tbolt.c
**********************************************************
```

```
/*
 * mr_sas_tbolt.c: source for mr_sas driver for New Generation.
 * i.e. Thunderbolt and Invader
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 */


#include <stddef.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/atomic.h>
#include <sys/scsi/scsi.h>
#include <sys/byteorder.h>
#include "ld_pd_map.h"
#include "mr_sas.h"
#include "fusion.h"


// Pre-TB command size and TB command size.
#define MR_COMMAND_SIZE (64*20) // 1280 bytes
MR_LD_RAID *MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map);
U16 MR_TargetIdToLdGet(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map);
U16 MR_GetLDTgtId(U32 ld, MR_FW_RAID_MAP_ALL *map);
U16 get_updated_dev_handle(PLD_LOAD_BALANCE_INFO lbInfo, struct IO_REQUEST_INFO
extern ddi_dma_attr_t mrsas_generic_dma_attr;
extern uint32_t mrsas_tbolt_max_cap_maxxfer;
extern struct ddi_device_acc_attr endian_attr;
extern int      debug_level_g;
extern unsigned int     enable_fp;
volatile int dump_io_wait_time = 90;
extern void
io_timeout_checker(void *arg);
extern int
mfi_state_transition_to_ready(struct mrsas_instance *instance);
extern volatile int  debug_timeout_g;
extern int      mrsas_issue_pending_cmds(struct mrsas_instance *);
extern int mrsas_complete_pending_cmds(struct mrsas_instance *instance);
extern void     push_pending_mfi_pkt(struct mrsas_instance *,
                        struct mrsas_cmd *);
extern U8 MR_BuildRaidContext(struct mrsas_instance *, struct IO_REQUEST_INFO *,
            MPI2_SCSI_IO_VENDOR_UNIQUE *, MR_FW_RAID_MAP_ALL *);

static volatile int  debug_tbolt_fw_faults_after_ocr_g  = 0;

/*
 * destroy_mfi_mpi_frame_pool
 */
void
destroy_mfi_mpi_frame_pool(struct mrsas_instance *instance)
{
        int     i;
```



```
        struct mrsas_cmd        *cmd;

        /* return all mfi frames to pool */
        for (i = 0; i < MRSAS_APP_RESERVED_CMDS; i++) {
                cmd = instance->cmd_list[i];
                if (cmd->frame_dma_obj_status == DMA_OBJ_ALLOCATED)
                        (void) mrsas_free_dma_obj(instance,
                            cmd->frame_dma_obj);
                cmd->frame_dma_obj_status  = DMA_OBJ_FREED;
        }
}
```

_____unchanged_portion_omitted_

```
/*
 * ThunderBolt(TB) Request Message Frame Pool
 */
int
create_mpi2_frame_pool(struct mrsas_instance *instance)
{
        int             i = 0;
        int             cookie_cnt;
        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));

        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) {
```

```
191                 cmn_err(CE_WARN,
192                     "mr_sas: could not alloc mpi2 frame pool");
193                 return (DDI_FAILURE);
194         }

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

199         instance->io_request_frames =
200             (uint8_t *)instance->mpi2_frame_pool_dma_obj.buffer;
201         instance->io_request_frames_phy =
202             (uint32_t)
203             instance->mpi2_frame_pool_dma_obj.dma_cookie[0].dmac_address;

205         con_log(CL_DLEVEL3, (CE_NOTE,
206             "io_request_frames 0x%p",
207             "io_request_frames 0x%x",
207             instance->io_request_frames));

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

213         io_req_base = (uint8_t *)instance->io_request_frames +
214             MRSAS_THUNDERBOLT_MSG_SIZE;
215         io_req_base_phys = instance->io_request_frames_phy +
216             MRSAS_THUNDERBOLT_MSG_SIZE;

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

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

224                 offset = i * MRSAS_THUNDERBOLT_MSG_SIZE;

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

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

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

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

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


249                 cmd->SMID = i+1;

251                 con_log(CL_DLEVEL3, (CE_NOTE,
252                     "Frame Pool Addr [%x]0x%p",
253                     "Frame Pool Addr [%x]0x%x",
253                     cmd->index, cmd->scsi_io_request));
```



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

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

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


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

272                 con_log(CL_DLEVEL3, (CE_NOTE,
273                     "Sgl bufffers phys  [%x]0x%x",
274                     cmd->index, cmd->sgl_phys_addr));
275         }

277         return (DDI_SUCCESS);

279 }
_____unchanged_portion_omitted_


414 /*
415  * Allocate Request and Reply  Queue Descriptors.
416  */
417 int
418 alloc_req_rep_desc(struct mrsas_instance *instance)
419 {
420         uint32_t        request_q_sz, reply_q_sz;
421         int             i, max_request_q_sz, max_reply_q_sz;
422         uint64_t        request_desc;
423         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
424         uint64_t        *reply_ptr;

426         /*
427          * ThunderBolt(TB) There's no longer producer consumer mechanism.
428          * Once we have an interrupt we are supposed to scan through the list of
429          * reply descriptors and process them accordingly. We would be needing
430          * to allocate memory for 1024 reply descriptors
431          */

433         /* Allocate Reply Descriptors */
434         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x\n",
435             (uint_t)sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));
436             sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));

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

440         reply_q_sz = 8 * max_reply_q_sz;


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

446         instance->reply_desc_dma_obj.size = reply_q_sz;
447         instance->reply_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
```

```
448         instance->reply_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
449         instance->reply_desc_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
450         instance->reply_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
451         instance->reply_desc_dma_obj.dma_attr.dma_attr_align = 16;

453         if (mrsas_alloc_dma_obj(instance, &instance->reply_desc_dma_obj,
454                         (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
455                 cmn_err(CE_WARN,
456                     "mr_sas: could not alloc reply queue");
457                 return (DDI_FAILURE);
458         }

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

463         // virtual address of  reply queue
464         instance->reply_frame_pool = (MPI2_REPLY_DESCRIPTORS_UNION *)(
465             instance->reply_desc_dma_obj.buffer);

467         instance->reply_q_depth = max_reply_q_sz;

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

472         con_log(CL_ANN1, (CE_NOTE, "[reply queue virt addr]0x%p",
473         con_log(CL_ANN1, (CE_NOTE, "[reply queue virt addr]0x%x",
473             instance->reply_frame_pool));

475         /* initializing reply address to 0xFFFFFFFF */
476         reply_desc = instance->reply_frame_pool;

478         for (i = 0; i < instance->reply_q_depth; i++) {
479                 reply_desc->Words = (uint64_t)~0;
480                 reply_desc++;
481         }


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

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


491         instance->reply_pool_limit_phy = (instance->reply_frame_pool_phy +
492             reply_q_sz);

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


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

501         /* Allocate Request Descriptors */
502         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x\n",
503             (int)sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));
504             sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));

505         request_q_sz = 8 *
506             (instance->max_fw_cmds);

508         instance->request_desc_dma_obj.size = request_q_sz;
509         instance->request_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
510         instance->request_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
```



```
511         instance->request_desc_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU
512         instance->request_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
513         instance->request_desc_dma_obj.dma_attr.dma_attr_align = 16;

515         if (mrsas_alloc_dma_obj(instance, &instance->request_desc_dma_obj,
516             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
517                 cmn_err(CE_WARN,
518                     "mr_sas: could not alloc request queue desc");
519                 goto fail_undo_reply_queue;
520         }

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

525         /* virtual address of  request queue desc */
526         instance->request_message_pool = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
527             (instance->request_desc_dma_obj.buffer);

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

532         max_request_q_sz = instance->max_fw_cmds;

534         return (DDI_SUCCESS);

536 fail_undo_reply_queue:
537         if (instance->reply_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
538                 (void) mrsas_free_dma_obj(instance,
539                     instance->reply_desc_dma_obj);
540                 instance->reply_desc_dma_obj.status = DMA_OBJ_FREED;
541         }

543         return (DDI_FAILURE);
544 }
_____unchanged_portion_omitted_

884 int
885 mrsas_tbolt_ioc_init(struct mrsas_instance *instance, dma_obj_t *mpi2_dma_obj,
886     ddi_acc_handle_t accessp)
887 {
888         int                             numbytes, i;
889         int ret                         = DDI_SUCCESS;
890         uint16_t                        flags;
891         int                             status;
892         timespec_t                      time;
893         uint64_t                        mSec;
894         uint32_t msecs                  = MFI_POLL_TIMEOUT_SECS * MILLISEC;
895         struct mrsas_init_frame2        *mfiFrameInit2;
896         struct mrsas_header             *frame_hr;
897         Mpi2IOCInitRequest_t            *init;
898         struct mrsas_cmd                *cmd = NULL;
899         struct mrsas_drv_ver            drv_ver_info;
900         MRSAS_REQUEST_DESCRIPTOR_UNION  *req_desc;


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


906 #ifdef DEBUG
907         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
908             (int)sizeof (*mfiFrameInit2)));
909         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", (int)sizeof (*init)));
909             sizeof (*mfiFrameInit2)));
910         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", sizeof (*init)));
910         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
911             (int)sizeof (struct mrsas_init_frame2)));
```

```
912             sizeof (struct mrsas_init_frame2)));
912         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n",
913             (int)sizeof (Mpi2IOCInitRequest_t)));
914             sizeof (Mpi2IOCInitRequest_t)));
914 #endif

916         init = (Mpi2IOCInitRequest_t *)mpi2_dma_obj->buffer;
917         numbytes = sizeof (*init);
918         bzero(init, numbytes);

920         ddi_put8(mpi2_dma_obj->acc_handle, &init->Function,
921             MPI2_FUNCTION_IOC_INIT);

923         ddi_put8(mpi2_dma_obj->acc_handle, &init->WhoInit,
924             MPI2_WHOINIT_HOST_DRIVER);

926         /* set MsgVersion and HeaderVersion host driver was built with */
927         ddi_put16(mpi2_dma_obj->acc_handle, &init->MsgVersion,
928             MPI2_VERSION);

930         ddi_put16(mpi2_dma_obj->acc_handle, &init->HeaderVersion,
931             MPI2_HEADER_VERSION);

933         ddi_put16(mpi2_dma_obj->acc_handle, &init->SystemRequestFrameSize,
934             instance->raid_io_msg_size / 4);

936         ddi_put16(mpi2_dma_obj->acc_handle, &init->ReplyFreeQueueDepth,
937             0);

939         ddi_put16(mpi2_dma_obj->acc_handle,
940             &init->ReplyDescriptorPostQueueDepth,
941             instance->reply_q_depth);
942         /*
943          * These addresses are set using the DMA cookie addresses from when the
944          * memory was allocated.  Sense buffer hi address should be 0.
945          * ddi_put32(accessp, &init->SenseBufferAddressHigh, 0);
946          */

948         ddi_put32(mpi2_dma_obj->acc_handle,
949             &init->SenseBufferAddressHigh, 0);

951         ddi_put64(mpi2_dma_obj->acc_handle,
952             (uint64_t *)&init->SystemRequestFrameBaseAddress,
953             instance->io_request_frames_phy);

955         ddi_put64(mpi2_dma_obj->acc_handle,
956             &init->ReplyDescriptorPostQueueAddress,
957             instance->reply_frame_pool_phy);

959         ddi_put64(mpi2_dma_obj->acc_handle,
960             &init->ReplyFreeQueueAddress, 0);

962         cmd = instance->cmd_list[0];
963         if (cmd == NULL) {
964                 return (DDI_FAILURE);
965         }
966         cmd->retry_count_for_ocr = 0;
967         cmd->pkt = NULL;
968         cmd->drv_pkt_time = 0;

970         mfiFrameInit2 = (struct mrsas_init_frame2 *)cmd->scsi_io_request;
971         con_log(CL_ANN1, (CE_CONT, "[mfi vaddr]%p", mfiFrameInit2));
972         con_log(CL_ANN1, (CE_CONT, "[mfi vaddr]%x", mfiFrameInit2));

973         frame_hdr = &cmd->frame->hdr;
```



```
975         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
976             MFI_CMD_STATUS_POLL_MODE);

978         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);

980         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;

982         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

984         con_log(CL_ANN, (CE_CONT,
985             "mrsas_tbolt_ioc_init: SMID:%x\n", cmd->SMID));

987         // Init the MFI Header
988         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
989             &mfiFrameInit2->cmd, MFI_CMD_OP_INIT);

991         con_log(CL_ANN1, (CE_CONT, "[CMD]%x", mfiFrameInit2->cmd));

993         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
994             &mfiFrameInit2->cmd_status,
995             MFI_STAT_INVALID_STATUS);

997         con_log(CL_ANN1, (CE_CONT, "[Status]%x", mfiFrameInit2->cmd_status));

999         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1000             &mfiFrameInit2->queue_info_new_phys_addr_lo,
1001             mpi2_dma_obj->dma_cookie[0].dmac_address);

1003        ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1004            &mfiFrameInit2->data_xfer_len,
1005            sizeof (Mpi2IOCInitRequest_t));

1007        con_log(CL_ANN1, (CE_CONT, "[reply q desc addr]%x",
1008            (int)init->ReplyDescriptorPostQueueAddress));
1009            init->ReplyDescriptorPostQueueAddress));

1010        /* fill driver version information*/
1011        fill_up_drv_ver(&drv_ver_info);
1012
1013        /* allocate the driver version data transfer buffer */
1014        instance->drv_ver_dma_obj.size = sizeof(drv_ver_info.drv_ver);
1015        instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
1016        instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
1017        instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
1018        instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
1019        instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;

1021        if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
1022            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
1023                cmn_err(CE_WARN,
1024                    "fusion init: Could not allocate driver version buffer.");
1025                return (DDI_FAILURE);
1026        }
1027        /* copy driver version to dma  buffer*/
1028        (void) memset(instance->drv_ver_dma_obj.buffer, 0,sizeof(drv_ver_info.dr
1029        ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
1030                (uint8_t *)drv_ver_info.drv_ver,
1031                (uint8_t *)instance->drv_ver_dma_obj.buffer,
1032                sizeof(drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);

1034        /*send driver version physical address to firmware*/
1035        ddi_put64(cmd->frame_dma_obj.acc_handle,
1036            &mfiFrameInit2->driverversion, instance->drv_ver_dma_obj.dma_cookie[

1038        con_log(CL_ANN1, (CE_CONT, "[MPIINIT2 frame Phys addr ]0x%x len = %x",
1039            mfiFrameInit2->queue_info_new_phys_addr_lo,
```

```
1040             (int)sizeof (Mpi2IOCInitRequest_t)));
1041             sizeof (Mpi2IOCInitRequest_t)));

1042         con_log(CL_ANN1, (CE_CONT, "[Length]%x", mfiFrameInit2->data_xfer_len));

1044         con_log(CL_ANN1, (CE_CONT, "[MFI frame Phys Address]%x len = %x",
1045             cmd->scsi_io_request_phys_addr,
1046             (int) sizeof (struct mrsas_init_frame2)));
1046             cmd->scsi_io_request_phys_addr, sizeof (struct mrsas_init_frame2)));

1048         /* disable interrupts before sending INIT2 frame */
1049         instance->func_ptr->disable_intr(instance);

1051         req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
1052             instance->request_message_pool;
1053         req_desc->Words = cmd->scsi_io_request_phys_addr;
1054         req_desc->MFAIo.RequestFlags =
1055             (MPI2_REQ_DESCRIPT_FLAGS_MFA << MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1057         cmd->request_desc = req_desc;

1059         /* issue the init frame */
1060         instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);

1062         con_log(CL_ANN1, (CE_CONT, "[cmd = %d] ", frame_hdr->cmd));
1063         con_log(CL_ANN1, (CE_CONT, "[cmd  Status= %x] ",
1064             frame_hdr->cmd_status));

1066         if (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1067             &mfiFrameInit2->cmd_status) == 0) {
1068                 con_log(CL_ANN, (CE_NOTE, "INIT2 Success"));
1069                 ret = DDI_SUCCESS;
1070         } else {
1071                 con_log(CL_ANN, (CE_WARN, "INIT2 Fail"));
1072                 mrsas_dump_reply_desc(instance);
1073                 goto fail_ioc_init;
1074         }

1076         mrsas_dump_reply_desc(instance);

1078         instance->unroll.verBuff = 1;

1080         con_log(CL_ANN, (CE_NOTE,
1081                     "mrsas_tbolt_ioc_init: SUCCESSFULL\n"));


1084         return (DDI_SUCCESS);


1087 fail_ioc_init:

1089         mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

1091         return (DDI_FAILURE);
1092 }
_____unchanged_portion_omitted_
1115 /*
1116  * scsi_pkt handling
1117  *
1118  * Visible to the external world via the transport structure.
1119  */

1121 int
1122 mrsas_tbolt_tran_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1123 {
1124         struct mrsas_instance   *instance = ADDR2MR(ap);
```



```
1125         struct scsa_cmd         *acmd = PKT2CMD(pkt);
1126         struct mrsas_cmd        *cmd = NULL;
1127         int                     rval, i;
1128         uchar_t                 cmd_done = 0;
1129         Mpi2RaidSCSIIORequest_t *scsi_raid_io;
1130         uint32_t                msecs = 120 * MILLISEC;

1132         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1133         if (instance->deadadapter == 1) {
1134                 cmn_err(CE_WARN,
1135                     "mrsas_tran_start:TBOLT return TRAN_FATAL_ERROR "
1136                     "for IO, as the HBA doesnt take any more IOs");
1137                 if (pkt) {
1138                         pkt->pkt_reason         = CMD_DEV_GONE;
1139                         pkt->pkt_statistics     = STAT_DISCON;
1140                 }
1141                 return (TRAN_FATAL_ERROR);
1142         }
1143         if (instance->adapterresetinprogress) {
1144                 con_log(CL_ANN, (CE_NOTE, "Reset flag set, "
1145                     "returning mfi_pkt and setting TRAN_BUSY\n"));
1146                 return (TRAN_BUSY);
1147         }
1148         rval = mrsas_tbolt_prepare_pkt(acmd);

1150         cmd = mrsas_tbolt_build_cmd(instance, ap, pkt, &cmd_done);

1152         /*
1153          * Check if the command is already completed by the mrsas_build_cmd()
1154          * routine. In which case the busy_flag would be clear and scb will be
1155          * NULL and appropriate reason provided in pkt_reason field
1156          */
1157         if (cmd_done) {
1158                 pkt->pkt_reason = CMD_CMPLT;
1159                 pkt->pkt_scbp[0] = STATUS_GOOD;
1160                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1161                     | STATE_SENT_CMD;
1162                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1163                         (*pkt->pkt_comp)(pkt);
1164                 }

1166                 return (TRAN_ACCEPT);
1167         }

1169         if (cmd == NULL) {
1170                 return (TRAN_BUSY);
1171         }


1174         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1175                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1176                         cmn_err(CE_WARN,
1177                             "Command Queue Full... Returning BUSY \n");
1178                         return_raid_msg_pkt(instance, cmd);
1179                         return (TRAN_BUSY);
1180                 }

1182                 /* Synchronize the Cmd frame for the controller */
1183                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1184                     DDI_DMA_SYNC_FORDEV);

1186                 con_log(CL_ANN, (CE_CONT, "tbolt_issue_cmd: SCSI CDB[0]=0x%x "
1187                     "cmd->index:0x%x SMID 0x%x\n", pkt->pkt_cdbp[0], cmd->index,
1187                     "cmd->index:0x%x SMID %0x%x\n", pkt->pkt_cdbp[0], cmd->index

1189                 instance->func_ptr->issue_cmd(cmd, instance);
```

```
1191                return (TRAN_ACCEPT);

1193        } else {
1194                instance->func_ptr->issue_cmd(cmd, instance);
1195                (void) wait_for_outstanding_poll_io(instance);
1196                return (TRAN_ACCEPT);
1197        }
1198 }
______unchanged_portion_omitted_


1229 int
1230 mr_sas_tbolt_build_sgl(struct mrsas_instance *instance,
1231     struct scsa_cmd *acmd,
1232     struct mrsas_cmd *cmd,
1233     Mpi2RaidSCSIIORequest_t *scsi_raid_io,
1234     uint32_t *datalen)
1235 {
1236         uint32_t                MaxSGEs;
1237         int                     sg_to_process;
1238         uint32_t                i, j, SGEdwords = 0;
1239         uint32_t                numElements, endElement;
1240         Mpi25IeeeSgeChain64_t   *ieeeChainElement = NULL;
1241         Mpi25IeeeSgeChain64_t   *scsi_raid_io_sgl_ieee = NULL;
1242         uint32_t                SGLFlags = 0;

1244         con_log(CL_ANN1, (CE_NOTE,
1245             "chkpnt: Building Chained SGL :%d", __LINE__));

1247         /* Calulate SGE size in number of Words(32bit) */
1248         /* Clear the datalen before updating it. */
1249         *datalen = 0;

1251         SGEdwords = sizeof (Mpi25IeeeSgeChain64_t) / 4;

1253         MaxSGEs = instance->max_sge_in_main_msg;

1255         ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1256             &scsi_raid_io->SGLFlags,
1257             MPI2_SGE_FLAGS_64_BIT_ADDRESSING);

1259         // set data transfer flag.
1260         if (acmd->cmd_flags & CFLAG_DMASEND) {
1261                 ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1262                     &scsi_raid_io->Control,
1263                     MPI2_SCSIIO_CONTROL_WRITE);
1264         } else {
1265                 ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1266                     &scsi_raid_io->Control, MPI2_SCSIIO_CONTROL_READ);
1267         }

1269
1270         numElements = acmd->cmd_cookiecnt;

1272         con_log(CL_DLEVEL1, (CE_NOTE, "[SGE Count]:%x", numElements));

1274         if (numElements > instance->max_num_sge) {
1275                 con_log(CL_ANN, (CE_NOTE,
1276                     "[Max SGE Count Exceeded]:%x", numElements));
1277                 return (numElements);
1278         }

1280         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1281             &scsi_raid_io->RaidContext.numSGE, (uint8_t)numElements);
```



```
1283         /* set end element in main message frame */
1284         endElement = (numElements <= MaxSGEs) ? numElements : (MaxSGEs - 1);

1286         /* prepare the scatter-gather list for the firmware */
1287         scsi_raid_io_sgl_ieee =
1288             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

1290         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1291                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = scsi_raid_io_sgl_ieee;
1292                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;
1293
1294                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1295                     &sgl_ptr_end->Flags, 0);
1296         }

1298         for (i = 0; i < endElement; i++, scsi_raid_io_sgl_ieee++) {
1299                 ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
1300                     &scsi_raid_io_sgl_ieee->Address,
1301                     acmd->cmd_dmacookies[i].dmac_laddress);

1303                 ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1304                     &scsi_raid_io_sgl_ieee->Length,
1305                     acmd->cmd_dmacookies[i].dmac_size);

1307                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1308                     &scsi_raid_io_sgl_ieee->Flags, 0);
1309
1310                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1311                         if (i == (numElements - 1))
1312                                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1313                                         &scsi_raid_io_sgl_ieee->Flags, IEEE_SGE_
1314                 }

1316                 *datalen += acmd->cmd_dmacookies[i].dmac_size;

1318 #ifdef DEBUG
1319                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]: %" PRIx64,
1319                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]:%llx",
1320                     scsi_raid_io_sgl_ieee->Address));
1321                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Length]:%x",
1322                     scsi_raid_io_sgl_ieee->Length));
1323                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Flags]:%x",
1324                     scsi_raid_io_sgl_ieee->Flags));
1325 #endif

1327         }

1329         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1330             &scsi_raid_io->ChainOffset, 0);

1332         /* check if chained SGL required */
1333         if (i < numElements) {

1335                 con_log(CL_ANN1, (CE_NOTE, "[Chain Element index]:%x", i));
1336
1337                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1338                         uint16_t        ioFlags = ddi_get16(instance->mpi2_frame
1339                                                 &scsi_raid_io->IoFlags);

1341                         if ((ioFlags & MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH
1342                                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1343                                         &scsi_raid_io->ChainOffset, (U8)instance
1344                         else
1345                                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1346                                         &scsi_raid_io->ChainOffset, 0);
1347                 }
```

```
1348                 else {
1349                         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1350                                 &scsi_raid_io->ChainOffset, (U8)instance->chain_
1351                 }

1353                 /* prepare physical chain element */
1354                 ieeeChainElement  = scsi_raid_io_sgl_ieee;

1356                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1357                     &ieeeChainElement->NextChainOffset, 0);

1359                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER)
1360                         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1361                                 &ieeeChainElement->Flags, IEEE_SGE_FLAGS_CHAIN_E
1362                 else
1363                         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1364                                 &ieeeChainElement->Flags,
1365                                 (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
1366                    MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));

1368                 ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1369                     &ieeeChainElement->Length,
1370                     (sizeof (MPI2_SGE_IO_UNION) * (numElements - i)));

1372                 ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
1373                     &ieeeChainElement->Address,
1374                     (U64)cmd->sgl_phys_addr);

1376                 sg_to_process = numElements - i;

1378                 con_log(CL_ANN1, (CE_NOTE,
1379                     "[Additional SGE Count]:%x", endElement));

1381                 /* point to the chained SGL buffer */
1382                 scsi_raid_io_sgl_ieee = (Mpi25IeeeSgeChain64_t *)cmd->sgl;

1384                 /* build rest of the SGL in chained buffer */
1385                 for (j = 0; j < sg_to_process; j++, scsi_raid_io_sgl_ieee++) {
1386                         con_log(CL_DLEVEL3, (CE_NOTE, "[remaining SGL]:%x", i));

1388                         ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
1389                             &scsi_raid_io_sgl_ieee->Address,
1390                             acmd->cmd_dmacookies[i].dmac_laddress);

1392                         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1393                             &scsi_raid_io_sgl_ieee->Length,
1394                             acmd->cmd_dmacookies[i].dmac_size);

1396                         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1397                             &scsi_raid_io_sgl_ieee->Flags, 0);

1399                         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1400                                 if (i == (numElements - 1))
1401                                         ddi_put8(instance->mpi2_frame_pool_dma_o
1402                                                 &scsi_raid_io_sgl_ieee->Flags, I
1403                         }

1405                         *datalen += acmd->cmd_dmacookies[i].dmac_size;

1407 #if DEBUG
1408                         con_log(CL_DLEVEL1, (CE_NOTE,
1409                             "[SGL Address]: %" PRIx64,
1409                             "[SGL Address]:%llx",
1410                             scsi_raid_io_sgl_ieee->Address));
1411                         con_log(CL_DLEVEL1, (CE_NOTE,
1412                             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
```



```
1413                         con_log(CL_DLEVEL1, (CE_NOTE,
1414                             "[SGL Flags]:%x", scsi_raid_io_sgl_ieee->Flags));
1415 #endif

1417                         i++;
1418                 }
1419         }

1421         return (0);
1422 } /*end of BuildScatterGather */
_____unchanged_portion_omitted_

1978 void
1979 tbolt_issue_cmd(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
1980 {
1981         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;
1982         atomic_add_16(&instance->fw_outstanding, 1);

1984         struct scsi_pkt *pkt;

1986         con_log(CL_ANN1, (CE_NOTE, "tbolt_issue_cmd: cmd->[SMID]=0x%X", cmd->SMI

1988         con_log(CL_DLEVEL1, (CE_CONT,
1989             " [req desc Words] %" PRIx64 " \n", req_desc->Words));
1989             " [req desc Words] %llx \n", req_desc->Words));
1990         con_log(CL_DLEVEL1, (CE_CONT,
1991             " [req desc low part] %x \n",
1992             (uint_t)(req_desc->Words & 0xffffffffff)));
1991             " [req desc low part] %x \n", req_desc->Words));
1993         con_log(CL_DLEVEL1, (CE_CONT,
1994             " [req desc high part] %x \n", (uint_t)(req_desc->Words >> 32)));
1993             " [req desc high part] %x \n", (req_desc->Words >> 32)));
1995         pkt = cmd->pkt;
1996
1997         if (pkt) {
1998                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
1999                     "ISSUED CMD TO FW : called : cmd:"
2000                     ": %p instance : %p pkt : %p pkt_time : %x\n",
2001                     gethrtime(), (void *)cmd, (void *)instance,
2002                     (void *)pkt, cmd->drv_pkt_time));
2003                 if (instance->adapterresetinprogress) {
2004                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2005                         con_log(CL_ANN, (CE_NOTE,
2006                             "TBOLT Reset the scsi_pkt timer"));
2007                 } else {
2008                         push_pending_mfi_pkt(instance, cmd);
2009                 }

2011         } else {
2012                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
2013                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
2014                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
2015         }

2017         /* Issue the command to the FW */
2018         mutex_enter(&instance->reg_write_mtx);
2019         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2020         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2021         mutex_exit(&instance->reg_write_mtx);
2022 }

2024 /*
2025  * issue_cmd_in_sync_mode
2026  */
2027 int
2028 tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *instance,
```

```
2029     struct mrsas_cmd *cmd)
2030 {
2031         int             i;
2032         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2033         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

2035         struct mrsas_header     *hdr;
2036         hdr = (struct mrsas_header *)&cmd->frame->hdr;

2038         con_log(CL_ANN, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode: cmd->[SMID]=0x%


2041         if (instance->adapterresetinprogress) {
2042                 cmd->drv_pkt_time = ddi_get16
2043                     (cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2044                 if (cmd->drv_pkt_time < debug_timeout_g)
2045                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2046                 con_log(CL_ANN, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode:"
2047                     "RESET-IN-PROGRESS, issue cmd & return.\n"));

2049                 mutex_enter(&instance->reg_write_mtx);
2050                 WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2051                 WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2052                 mutex_exit(&instance->reg_write_mtx);

2054                 return (DDI_SUCCESS);
2055         } else {
2056                 con_log(CL_ANN1, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode: pushin
2057                 push_pending_mfi_pkt(instance, cmd);
2058         }

2060         con_log(CL_DLEVEL2, (CE_NOTE,
2061             "HighQport offset :%p",
2060             "HighQport offset :%lx",
2062             (uint32_t *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT)));
2063         con_log(CL_DLEVEL2, (CE_NOTE,
2064             "LowQport offset :%p",
2063             "LowQport offset :%lx",
2065             (uint32_t *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT)));

2067         cmd->sync_cmd = MRSAS_TRUE;
2068         cmd->cmd_status =  ENODATA;


2071         mutex_enter(&instance->reg_write_mtx);
2072         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2073         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2074         mutex_exit(&instance->reg_write_mtx);

2076         con_log(CL_ANN1, (CE_NOTE,
2077             " req desc high part %x \n", (uint_t)(req_desc->Words >> 32)));
2076             " req desc high part %x \n", (req_desc->Words >> 32)));
2078         con_log(CL_ANN1, (CE_NOTE,
2079             " req desc low part %x \n", (uint_t)(req_desc->Words & 0xffffffff)))
2078             " req desc low part %x \n", req_desc->Words));

2081         mutex_enter(&instance->int_cmd_mtx);
2082         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
2083                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
2084         }
2085         mutex_exit(&instance->int_cmd_mtx);


2088         if (i < (msecs -1)) {
2089                 return (DDI_SUCCESS);
2090         } else {
```



```
2091                 return (DDI_FAILURE);
2092         }
2093 }

2095 /*
2096  * issue_cmd_in_poll_mode
2097  */
2098 int
2099 tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *instance,
2100     struct mrsas_cmd *cmd)
2101 {
2102         int             i;
2103         uint16_t        flags;
2104         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2105         struct mrsas_header *frame_hdr;

2107         con_log(CL_ANN, (CE_NOTE, "tbolt_issue_cmd_in_poll_mode: cmd->[SMID]=0x%
2108
2109         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

2111         frame_hdr = (struct mrsas_header *)&cmd->frame->hdr;
2112         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
2113             MFI_CMD_STATUS_POLL_MODE);
2114         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
2115         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;
2116         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

2118         con_log(CL_ANN1, (CE_NOTE,
2119             " req desc low part %x \n", (uint_t)(req_desc->Words & 0xffffffff)))
2118             " req desc low part %x \n", req_desc->Words));
2120         con_log(CL_ANN1, (CE_NOTE,
2121             " req desc high part %x \n", (uint_t)(req_desc->Words >> 32)));
2120             " req desc high part %x \n", (req_desc->Words >> 32)));

2123         /* issue the frame using inbound queue port */
2124         mutex_enter(&instance->reg_write_mtx);
2125         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2126         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2127         mutex_exit(&instance->reg_write_mtx);

2129         for (i = 0; i < msecs && (
2130             ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
2131             == MFI_CMD_STATUS_POLL_MODE); i++) {
2132                 /* wait for cmd_status to change from 0xFF */
2133                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
2134         }

2136         if (ddi_get8(cmd->frame_dma_obj.acc_handle,
2137             &frame_hdr->cmd_status) == MFI_CMD_STATUS_POLL_MODE) {
2138                 con_log(CL_ANN1, (CE_NOTE,
2139                     " cmd failed %" PRIx64 " \n", (req_desc->Words)));
2138                     " cmd failed %x \n", (req_desc->Words)));
2140                 return (DDI_FAILURE);
2141         }

2143         return (DDI_SUCCESS);
2144 }
_____unchanged_portion_omitted_


2296 void
2297 mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *instance,
2298     struct mrsas_cmd *cmd)
2299 {
2300         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2301         Mpi25IeeeSgeChain64_t           *scsi_raid_io_sgl_ieee;
```

```
2302         MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
2303         uint32_t                        index;

2305         if (!instance->tbolt) {
2306                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled.\n"));
2307                 return;
2308         }

2310         index = cmd->index;

2312         ReqDescUnion =
2313             mr_sas_get_request_descriptor(instance, index, cmd);

2315         if (!ReqDescUnion) {
2316                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]"));
2315                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]%x"));
2317                 return;
2318         }

2320         con_log(CL_ANN1, (CE_NOTE, "[SMID]%x", cmd->SMID));

2322         ReqDescUnion->Words = 0;

2324         ReqDescUnion->SCSIIO.RequestFlags =
2325             (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
2326             MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

2328         ReqDescUnion->SCSIIO.SMID = cmd->SMID;

2330         cmd->request_desc = ReqDescUnion;

2332         // get raid message frame pointer
2333         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2335         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
2336                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)&s
2337                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;
2338                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2339                     &sgl_ptr_end->Flags, 0);
2340         }

2342         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2343             &scsi_raid_io->Function,
2344             MPI2_FUNCTION_PASSTHRU_IO_REQUEST);

2346         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2347             &scsi_raid_io->SGLOffset0,
2348             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

2350         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2351             &scsi_raid_io->ChainOffset,
2352             (U8)offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16);

2354         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2355             &scsi_raid_io->SenseBufferLowAddress,
2356             cmd->sense_phys_addr1);


2359         scsi_raid_io_sgl_ieee =
2360             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

2362         ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
2363             &scsi_raid_io_sgl_ieee->Address,
2364             (U64)cmd->frame_phys_addr);

2366         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
```



```
2367             &scsi_raid_io_sgl_ieee->Flags,
2368             (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2369                     MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
2370         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2371             &scsi_raid_io_sgl_ieee->Length, 1024); //MEGASAS_MAX_SZ_CHAIN_FRAME

2373         con_log(CL_ANN1, (CE_NOTE,
2374             "[MFI CMD PHY ADDRESS]:%" PRIx64,
2373             "[MFI CMD PHY ADDRESS]:%x",
2375             scsi_raid_io_sgl_ieee->Address));
2376         con_log(CL_ANN1, (CE_NOTE,
2377             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
2378         con_log(CL_ANN1, (CE_NOTE, "[SGL Flags]:%x",
2379             scsi_raid_io_sgl_ieee->Flags));
2380 }


2383 void
2384 tbolt_complete_cmd(struct mrsas_instance *instance,
2385     struct mrsas_cmd *cmd)
2386 {
2387         uint8_t                         status;
2388         uint8_t                         extStatus;
2389         uint8_t                         arm;
2390         struct scsa_cmd                 *acmd;
2391         struct scsi_pkt                 *pkt;
2392         struct scsi_arq_status          *arqstat;
2393         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2394         LD_LOAD_BALANCE_INFO            *lbinfo;
2395         int i;

2397         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2399         status = ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2400             &scsi_raid_io->RaidContext.status);
2401         extStatus = ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2402             &scsi_raid_io->RaidContext.extStatus);

2404         con_log(CL_DLEVEL3, (CE_NOTE, "status %x", status));
2405         con_log(CL_DLEVEL3, (CE_NOTE, "extStatus %x", extStatus));

2407         if (status != MFI_STAT_OK) {
2408                 con_log(CL_ANN, (CE_WARN,
2409                     "IO Cmd Failed SMID %x", cmd->SMID));
2410         } else {
2411                 con_log(CL_ANN, (CE_NOTE,
2412                     "IO Cmd Success  SMID %x", cmd->SMID));
2413         }

2415         /* regular commands */

2417         switch (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2418             &scsi_raid_io->Function)) {

2420                 case MPI2_FUNCTION_SCSI_IO_REQUEST :  /* Fast Path IO. */
2421                         acmd =  (struct scsa_cmd *)cmd->cmd;
2422                         lbinfo = &instance->load_balance_info[acmd->device_id];

2424                         if (cmd->load_balance_flag & MEGASAS_LOAD_BALANCE_FLAG)
2425                                 arm = lbinfo->raid1DevHandle[0] == scsi_raid_io-

2427                                 lbinfo->scsi_pending_cmds[arm]--;
2428                                 cmd->load_balance_flag &= ~MEGASAS_LOAD_BALANCE_
2429                         }
2430                         con_log(CL_DLEVEL3, (CE_NOTE,
2431                             "FastPath IO Completion Success "));
```

```
2433                 case MPI2_FUNCTION_LD_IO_REQUEST :   {// Regular Path IO.
2434                         acmd =  (struct scsa_cmd *)cmd->cmd;
2435                         pkt =   (struct scsi_pkt *)CMD2PKT(acmd);

2437                         if (acmd->cmd_flags & CFLAG_DMAVALID) {
2438                                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
2439                                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
2440                                             acmd->cmd_dma_offset,
2441                                             acmd->cmd_dma_len,
2442                                             DDI_DMA_SYNC_FORCPU);
2443                                 }
2444                         }

2446                         pkt->pkt_reason         = CMD_CMPLT;
2447                         pkt->pkt_statistics     = 0;
2448                         pkt->pkt_state = STATE_GOT_BUS
2449                             | STATE_GOT_TARGET | STATE_SENT_CMD
2450                             | STATE_XFERRED_DATA | STATE_GOT_STATUS;

2452                         con_log(CL_ANN, (CE_CONT,
2453                             " CDB[0] = %x completed for %s: size %lx SMID %x cmd
2454                             pkt->pkt_cdbp[0],
2455                             ((acmd->islogical) ? "LD" : "PD"),
2456                             acmd->cmd_dmacount, cmd->SMID, status));

2458                         if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
2459                                 struct scsi_inquiry     *inq;

2461                                 if (acmd->cmd_dmacount != 0) {
2462                                         bp_mapin(acmd->cmd_buf);
2463                                         inq = (struct scsi_inquiry *)
2464                                             acmd->cmd_buf->b_un.b_addr;

2466                                         /* don't expose physical drives to OS */
2467                                         if (acmd->islogical &&
2468                                             (status == MFI_STAT_OK)) {
2469                                                 display_scsi_inquiry(
2470                                                     (caddr_t)inq);
2471                                         }
2472 #ifdef PDSUPPORT
2473                                         else if ((status ==
2474                                             MFI_STAT_OK) && inq->inq_dtype ==
2475                                             DTYPE_DIRECT) {

2477                                                 display_scsi_inquiry(
2478                                                     (caddr_t)inq);
2479                                         }
2480 #endif
2481                                         else {
2482                                                 /* for physical disk */
2483                                                 status =
2484                                                     MFI_STAT_DEVICE_NOT_FOUND;
2485                                         }
2486                                 }
2487                         }

2489                         switch (status) {
2490                         case MFI_STAT_OK:
2491                                 pkt->pkt_scbp[0] = STATUS_GOOD;
2492                                 break;
2493                         case MFI_STAT_LD_CC_IN_PROGRESS:
2494                         case MFI_STAT_LD_RECON_IN_PROGRESS:
2495                                 pkt->pkt_scbp[0] = STATUS_GOOD;
2496                                 break;
2497                         case MFI_STAT_LD_INIT_IN_PROGRESS:
```



```
2498                                 pkt->pkt_reason = CMD_TRAN_ERR;
2499                                 break;
2500                         case MFI_STAT_SCSI_IO_FAILED:
2501                                 cmn_err(CE_WARN, "tbolt_complete_cmd: scsi_io fa
2502                                 pkt->pkt_reason = CMD_TRAN_ERR;
2503                                 break;
2504                         case MFI_STAT_SCSI_DONE_WITH_ERROR:
2505                                 con_log(CL_ANN, (CE_WARN,
2506                                         "tbolt_complete_cmd: scsi_done with erro

2508                                 pkt->pkt_reason = CMD_CMPLT;
2509                                 ((struct scsi_status *)
2510                                     pkt->pkt_scbp)->sts_chk = 1;

2512                                 if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
2513                                         con_log(CL_ANN, (CE_WARN, "TEST_UNIT_REA
2514                                 } else {
2515                                         pkt->pkt_state |= STATE_ARQ_DONE;
2516                                         arqstat = (void *)(pkt->pkt_scbp);
2517                                         arqstat->sts_rqpkt_reason = CMD_CMPLT;
2518                                         arqstat->sts_rqpkt_resid = 0;
2519                                         arqstat->sts_rqpkt_state |=
2520                                             STATE_GOT_BUS | STATE_GOT_TARGET
2521                                             | STATE_SENT_CMD
2522                                             | STATE_XFERRED_DATA;
2523                                         *(uint8_t *)&arqstat->sts_rqpkt_status =
2524                                             STATUS_GOOD;
2525                                         con_log(CL_ANN1, (CE_NOTE,
2526                                             "Copying Sense data %x",
2527                                             cmd->SMID));

2529                                         ddi_rep_get8(
2530                                             instance->
2531                                             mpi2_frame_pool_dma_obj.acc_handle,
2532                                             (uint8_t *)
2533                                             &(arqstat->sts_sensedata),
2534                                             cmd->sense1,
2535                                             sizeof (struct scsi_extended_sense),
2536                                             DDI_DEV_AUTOINCR);
2537
2538                                 }
2539                                 break;
2540                         case MFI_STAT_LD_OFFLINE:
2541                                 cmn_err(CE_WARN,
2542                                         "tbolt_complete_cmd: ld offline "
2543                                         "CDB[0]=0x%x targetId=0x%x devhandle=0x%
2544                                         ddi_get8(instance->mpi2_frame_pool_dma_o
2545                                         ddi_get16(instance->mpi2_frame_pool_dma_
2546                                         ddi_get16(instance->mpi2_frame_pool_dma_
2547                                 pkt->pkt_reason = CMD_DEV_GONE;
2548                                 pkt->pkt_statistics  = STAT_DISCON;
2549                                 break;
2550                         case MFI_STAT_DEVICE_NOT_FOUND:
2551                         con_log(CL_ANN, (CE_CONT,
2552                                 "tbolt_complete_cmd: device not found error"));
2553                                 pkt->pkt_reason = CMD_DEV_GONE;
2554                                 pkt->pkt_statistics  = STAT_DISCON;
2555                                 break;

2557                         case MFI_STAT_LD_LBA_OUT_OF_RANGE:
2558                                 pkt->pkt_state |= STATE_ARQ_DONE;
2559                                 pkt->pkt_reason = CMD_CMPLT;
2560                                 ((struct scsi_status *)
2561                                     pkt->pkt_scbp)->sts_chk = 1;

2563                                 arqstat = (void *)(pkt->pkt_scbp);
```

```
2564                                arqstat->sts_rqpkt_reason = CMD_CMPLT;
2565                                arqstat->sts_rqpkt_resid = 0;
2566                                arqstat->sts_rqpkt_state |= STATE_GOT_BUS
2567                                    | STATE_GOT_TARGET | STATE_SENT_CMD
2568                                    | STATE_XFERRED_DATA;
2569                                *(uint8_t *)&arqstat->sts_rqpkt_status =
2570                                    STATUS_GOOD;

2572                                arqstat->sts_sensedata.es_valid = 1;
2573                                arqstat->sts_sensedata.es_key =
2574                                    KEY_ILLEGAL_REQUEST;
2575                                arqstat->sts_sensedata.es_class =
2576                                    CLASS_EXTENDED_SENSE;

2578                                /*
2579                                 * LOGICAL BLOCK ADDRESS OUT OF RANGE:
2580                                 * ASC: 0x21h; ASCQ: 0x00h;
2581                                 */
2582                                arqstat->sts_sensedata.es_add_code = 0x21;
2583                                arqstat->sts_sensedata.es_qual_code = 0x00;
2584                                break;
2585                        case MFI_STAT_INVALID_CMD:
2586                        case MFI_STAT_INVALID_DCMD:
2587                        case MFI_STAT_INVALID_PARAMETER:
2588                        case MFI_STAT_INVALID_SEQUENCE_NUMBER:
2589                        default:
2590                                cmn_err(CE_WARN, "tbolt_complete_cmd: Unknown st
2591                                pkt->pkt_reason = CMD_TRAN_ERR;

2593                                break;
2594                        }

2596                        atomic_add_16(&instance->fw_outstanding, (-1));

2598                        /* Call the callback routine */
2599                        if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
2600                            pkt->pkt_comp) {
2601                                (*pkt->pkt_comp)(pkt);
2602                        }

2604                        con_log(CL_ANN1, (CE_NOTE, "Free smid %x", cmd->SMID));

2606                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2607                            &scsi_raid_io->RaidContext.status, 0);

2609                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2610                            &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
2618                         == MR_DCMD_LD_MAP_GET_INFO &&
2619                         cmd->frame->dcmd.mbox.b[1]
2620                         == 1) {
2621
2622                        mutex_enter(&instance->sync_map_mtx);

2624                        con_log(CL_ANN, (CE_NOTE,
2625                           "LDMAP sync command  SMID RECEIVED 0x%X",
2626                           cmd->SMID));
2627                        if (cmd->frame->hdr.cmd_status != 0) {
2628                                cmn_err(CE_WARN,
2629                                        "map sync failed, status = 0x%x.\n",cmd-
```



```
2630                        }
2631                        else {
2632                                instance->map_id++;
2633                                cmn_err(CE_NOTE,
2634                                        "map sync received, switched map_id to %
2633                                        "map sync received, switched map_id to %
2635                        }

2637                        if (MR_ValidateMapInfo(instance->ld_map[(instance->map_i
2638                                instance->fast_path_io = 1;
2639                        else
2640                                instance->fast_path_io = 0;
2641
2642                        con_log(CL_ANN, (CE_NOTE,
2643                                "instance->fast_path_io %d \n",instance->fast_pa

2645                        instance->unroll.syncCmd = 0;

2647                        if(instance->map_update_cmd == cmd) {
2648                                return_raid_msg_pkt(instance, cmd);
2649                                atomic_add_16(&instance->fw_outstanding, (-1));
2650                                mrsas_tbolt_sync_map_info(instance);
2651                        }
2652
2653                        cmn_err(CE_NOTE, "LDMAP sync completed.\n");
2654                        mutex_exit(&instance->sync_map_mtx);
2655                        break;
2656                }

2658                if (cmd->frame->dcmd.opcode == MR_DCMD_CTRL_EVENT_WAIT) {
2659                        con_log(CL_ANN1, (CE_CONT,
2660                           "AEN command SMID RECEIVED 0x%X",
2661                           cmd->SMID));
2662                         if ((instance->aen_cmd == cmd) &&
2663                                 (instance->aen_cmd->abort_aen)) {
2664                                     con_log(CL_ANN, (CE_WARN,
2665                                         "mrsas_softintr: "
2666                                         "aborted_aen returned"));
2667                        }
2668                        else
2669                        {
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                }
2682                else
2683                {
2684                        con_log(CL_ANN, (CE_CONT,
2685                                "tbolt_complete_cmd: Wrong SMID RECEIVED 0x%X",
2686                                cmd->SMID));
2687                }
2688                break;
2689        default:
2690                /* free message */
2691                con_log(CL_ANN, (CE_NOTE, "tbolt_complete_cmd: Unknown Type!!!!!
2692                break;
2693        }
2694 }
```

```
2696 uint_t
2697 mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *instance)
2698 {
2699         uint8_t                         replyType;
2700         Mpi2SCSIIOSuccessReplyDescriptor_t *replyDesc;
2701         Mpi2ReplyDescriptorsUnion_t     *desc;
2702         uint16_t                        smid;
2703         union desc_value                d_val;
2704         struct mrsas_cmd                *cmd;
2705         uint32_t                        i;
2706         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2707         uint8_t                         status;

2709         struct mrsas_header     *hdr;
2710         struct scsi_pkt         *pkt;

2712         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2713             0, 0, DDI_DMA_SYNC_FORDEV);

2715         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2716             0, 0, DDI_DMA_SYNC_FORCPU);

2718         desc = instance->reply_frame_pool;
2719         desc += instance->reply_read_index;

2721         replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;
2722         replyType = replyDesc->ReplyFlags &
2723             MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2725         if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2726                 return (DDI_INTR_UNCLAIMED);

2728         con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %p  Words = %" PRIx64 " \n",
2727         con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %llx  Words = %llx \n",
2729                                 desc, desc->Words));

2731         d_val.word = desc->Words;
2732

2734         /* Read Reply descriptor */
2735         while ((d_val.u1.low != 0xffffffff) &&
2736             (d_val.u1.high != 0xffffffff)) {

2738                 (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2739                 0, 0, DDI_DMA_SYNC_FORCPU);

2741                 smid = replyDesc->SMID;

2743                 if (!smid || smid > instance->max_fw_cmds + 1) {
2744                         con_log(CL_ANN1, (CE_NOTE,
2745                             "Reply Desc at Break  = %p  Words = %" PRIx64 " \n",
2744                             "Reply Desc at Break  = %llx  Words = %llx \n",
2746                             desc, desc->Words));
2747                         break;
2748                 }

2750                 cmd     = instance->cmd_list[smid - 1];
2751                 if(!cmd ) {
2752                         con_log(CL_ANN1, (CE_NOTE,
2753                                 "mr_sas_tbolt_process_outstanding_cmd: Invalid c
2754                                 " or Poll commad Received in completion path\n")
2755                 }
2756                 else {
2757                         mutex_enter(&instance->cmd_pend_mtx);
2758                         if (cmd->sync_cmd == MRSAS_TRUE) {
```



```
2759                                 hdr = (struct mrsas_header *)&cmd->frame->hdr;
2760                                 if (hdr) {
2761                                         con_log(CL_ANN1, (CE_NOTE,
2762                                                 "mr_sas_tbolt_process_outstandin
2763                                                 " mlist_del_init(&cmd->list).\n"
2764                                         mlist_del_init(&cmd->list);
2765                                 }
2766                                 } else {
2767                                         pkt = cmd->pkt;
2768                                         if (pkt) {
2769                                                 con_log(CL_ANN1, (CE_NOTE,
2770                                                         "mr_sas_tbolt_process_ou
2771                                                         "mlist_del_init(&cmd->li
2772                                         mlist_del_init(&cmd->list);
2773                                 }
2774                         }

2776                         mutex_exit(&instance->cmd_pend_mtx);
2777
2778                         tbolt_complete_cmd(instance, cmd);
2779                 }
2780                 // set it back to all 0xfffffffff.
2781                 desc->Words = (uint64_t)~0;

2783                 instance->reply_read_index++;

2785                 if (instance->reply_read_index >= (instance->reply_q_depth)) {
2786                         con_log(CL_ANN1, (CE_NOTE, "wrap around"));
2787                         instance->reply_read_index = 0;
2788                 }

2790                 /* Get the next reply descriptor */
2791                 if (!instance->reply_read_index)
2792                         desc = instance->reply_frame_pool;
2793                 else
2794                         desc++;

2796                 replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;

2798                 d_val.word = desc->Words;

2800                 con_log(CL_ANN1, (CE_NOTE,
2801                     "Next Reply Desc  = %p Words = %" PRIx64 "\n",
2800                     "Next Reply Desc  = %llx Words = %llx\n",
2802                     desc, desc->Words));

2804                 replyType = replyDesc->ReplyFlags &
2805                     MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2807                 if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2808                         break;

2810         } /* End of while loop. */

2812         /* update replyIndex to FW */
2813         WR_MPI2_REPLY_POST_INDEX(instance->reply_read_index, instance);


2816         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2817             0, 0, DDI_DMA_SYNC_FORDEV);

2819         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2820             0, 0, DDI_DMA_SYNC_FORCPU);
2821         return (DDI_INTR_CLAIMED);
2822 }
```

_____***unchanged_portion_omitted_***