```
**********************************************************
   17098 Tue Nov  6 14:29:38 2012
new/./fusion.h
**********************************************************
______unchanged_portion_omitted_

 411 typedef struct _MR_SPAN_INFO {
 412         U32             noElements;                     /* 0x00 */
 413         U32             reserved1;                      /* 0x04 */
 414         MR_QUAD_ELEMENT quads[MAX_RAIDMAP_SPAN_DEPTH];  /* 0x08 */
 414         MR_QUAD_ELEMENT quad[MAX_RAIDMAP_SPAN_DEPTH];   /* 0x08 */
 415 } MR_SPAN_INFO;                                         /* 0x108, Total size */
______unchanged_portion_omitted_

 438 typedef struct _MR_LD_RAID {
 439         struct {
 440                 U32     fpCapable       :1;
 441                 U32     reserved5       :3;
 442                 U32     ldPiMode        :4;
 443                 U32     pdPiMode        :4;

 445                 /* FDE or controller encryption (MR_LD_ENCRYPTION_TYPE) */
 446                 U32     encryptionType  :8;

 448                 U32     fpWriteCapable  :1;
 449                 U32     fpReadCapable   :1;
 450                 U32     fpWriteAcrossStripe:1;
 451                 U32     fpReadAcrossStripe:1;
 452                 U32     reserved4       :8;
 453         } capability;                   /* 0x00 */
 454         U32     reserved6;
 455         U64     size;                   /* 0x08, LD size in blocks */
 456         U8      spanDepth;              /* 0x10, Total Number of Spans */
 457         U8      level;                  /* 0x11, RAID level */
 458         /* 0x12, shift-count to get stripe size (0=512, 1=1K, 7=64K, etc.) */
 459         U8      stripeShift;
 460         U8      rowSize;                /* 0x13, number of disks in a row */
 461         /* 0x14, number of data disks in a row */
 462         U8      rowDataSize;
 463         U8      writeMode;              /* 0x15, WRITE_THROUGH or WRITE_BACK */

 465         /* 0x16, To differentiate between RAID1 and RAID1E */
 466         U8      PRL;

 468         U8      SRL;                    /* 0x17 */
 469         U16     targetId;               /* 0x18, ld Target Id. */

 471         /* 0x1a, state of ld, state corresponds to MR_LD_STATE */
 472         U8      ldState;

 474         /* 0x1b, Pre calculate region type requests based on MFC etc.. */
 475         U8      regTypeReqOnWrite;

 477         U8      modFactor;              /* 0x1c, same as rowSize */
 478         /*
 479          * 0x1d, region lock type used for read, valid only if
 480          * regTypeOnReadIsValid=1
 481          */
 482         U8      regTypeReqOnRead;
 478         U8      regTypeReqOnRead;               /* 0x1d, region lock type used f
 483         U16     seqNum;                 /* 0x1e, LD sequence number */

 485         struct {
 486                 /* This LD requires sync command before completing */
 487                 U32     ldSyncRequired:1;
 488                 U32     reserved:31;
```



```
 489         } flags;                        /* 0x20 */

 491         U8      reserved3[0x5C];        /* 0x24 */
 492 } MR_LD_RAID;                           /* 0x80, Total Size */
______unchanged_portion_omitted_

 503 typedef struct _MR_FW_RAID_MAP {
 504         /* total size of this structure, including this field */
 505         U32                     totalSize;
 506         union {
 507                 /* Simple method of version checking variables */
 508                 struct {
 509                         U32     maxLd;
 510                         U32     maxSpanDepth;
 511                         U32     maxRowSize;
 512                         U32     maxPdCount;
 513                         U32     maxArrays;
 514                 } validationInfo;
 515                 U32     version[5];
 516                 U32     reserved1[5];
 517         } u1;

 519         U32                     ldCount;                /* count of lds */
 520         U32                     Reserved1;

 522         /*
 523          * 0x20 This doesn't correspond to
 518         /*   0x20 This doesn't correspond to
 524          * FW Ld Tgt Id to LD, but will purge. For example: if tgt Id is 4
 525          * and FW LD is 2, and there is only one LD, FW will populate the
 526          * array like this. [0xFF, 0xFF, 0xFF, 0xFF, 0x0.....]. This is to
 527          * help reduce the entire structure size if there are few LDs or
 522             * help reduce the entire strcture size if there are few LDs or
 528          * driver is looking info for 1 LD only.
 529          */
 530         U8                      ldTgtIdToLd[MAX_RAIDMAP_LOGICAL_DRIVES+ \
 531                                 MAX_RAIDMAP_VIEWS]; /* 0x20 */
 532         /* timeout value used by driver in FP IOs */
 533         U8                      fpPdIoTimeoutSec;
 534         U8                      reserved2[7];
 535         MR_ARRAY_INFO           arMapInfo[MAX_RAIDMAP_ARRAYS];  /* 0x00a8 */
 536         MR_DEV_HANDLE_INFO      devHndlInfo[MAX_RAIDMAP_PHYSICAL_DEVICES];

 538         /* 0x28a8-[0 -MAX_RAIDMAP_LOGICAL_DRIVES+MAX_RAIDMAP_VIEWS+1]; */
 539         MR_LD_SPAN_MAP          ldSpanMap[1];
 540 }MR_FW_RAID_MAP;                                        /* 0x3288, Total Size */
______unchanged_portion_omitted_
```

```
**********************************************************
   13182 Tue Nov  6 14:29:38 2012
new/./ld_pd_map.c
**********************************************************
```

```
/*
 * ***************************************************************************
 *
 * ld_pd_map.c
 *
 * 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
 *
 *
 * This module contains functions for device drivers
 * to get pd-ld mapping information.
 *
 * ***************************************************************************
 */

#include <sys/scsi/scsi.h>
#include "mr_sas.h"
#include "ld_pd_map.h"
#include "mr_sas.h"

/*
 * This function will check if FAST IO is possible on this logical drive
 * by checking the EVENT information available in the driver
 * by checking the EVENT information availabe in the driver
 */
#define MR_LD_STATE_OPTIMAL 3
#define ABS_DIFF(a, b)   (((a) > (b)) ? ((a) - (b)) : ((b) - (a)))
#define ABS_DIFF(a,b)   ( ((a) > (b)) ? ((a) - (b)) : ((b) - (a)) )

static void mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *,
    PLD_LOAD_BALANCE_INFO);
void
mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbI

U8
MR_BuildRaidContext(struct mrsas_instance *, struct IO_REQUEST_INFO *,
            MPI2_SCSI_IO_VENDOR_UNIQUE *, MR_FW_RAID_MAP_ALL *);

#define FALSE 0
#define TRUE 1

typedef U64     REGION_KEY;
typedef U32     REGION_LEN;
extern int      debug_level_g;


MR_LD_RAID
*MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
{
        return (&map->raidMap.ldSpanMap[ld].ldRaid);
}

U16
```



```
MR_GetLDTgtId(U32 ld, MR_FW_RAID_MAP_ALL *map)
U16 MR_GetLDTgtId
(U32 ld, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].ldRaid.targetId);
}


static MR_SPAN_BLOCK_INFO *
MR_LdSpanInfoGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
static MR_SPAN_BLOCK_INFO *MR_LdSpanInfoGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
{
        return (&map->raidMap.ldSpanMap[ld].spanBlock[0]);
}

static U8
MR_LdDataArmGet(U32 ld, U32 armIdx, MR_FW_RAID_MAP_ALL *map)
static U8 MR_LdDataArmGet(U32 ld, U32 armIdx, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].dataArmMap[armIdx]);
}

static U16
MR_ArPdGet(U32 ar, U32 arm, MR_FW_RAID_MAP_ALL *map)
static U16 MR_ArPdGet(U32 ar, U32 arm, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.arMapInfo[ar].pd[arm]);
}

static U16
MR_LdSpanArrayGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
static U16 MR_LdSpanArrayGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].spanBlock[span].span.arrayRef);
}

static U16
MR_PdDevHandleGet(U32 pd, MR_FW_RAID_MAP_ALL *map)
static U16 MR_PdDevHandleGet(U32 pd, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.devHndlInfo[pd].curDevHdl);
}

U16
MR_TargetIdToLdGet(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
U16 MR_TargetIdToLdGet(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldTgtIdToLd[ldTgtId]);
}

U16
MR_CheckDIF(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
U16 MR_CheckDIF(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
{
        MR_FW_RAID_MAP  *pFwRaidMap = &map->raidMap;
        MR_LD_RAID      *raid;
        U32             ld;

        ld = MR_TargetIdToLdGet(ldTgtId, map);

        if (ld >= MAX_LOGICAL_DRIVES) {
                return (FALSE);
        }

        raid = MR_LdRaidGet(ld, map);
```

```
110         return (raid->capability.ldPiMode == 0x8);
111 }

113 static MR_LD_SPAN *
114 MR_LdSpanPtrGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
111 static MR_LD_SPAN *MR_LdSpanPtrGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
115 {
116         return (&map->raidMap.ldSpanMap[ld].spanBlock[span].span);
117 }

119 /*
120  * This function will validate Map info data provided by FW
121  */
122 U8
123 MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo)
119 U8 MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo)
124 {
125         MR_FW_RAID_MAP *pFwRaidMap = &map->raidMap;
126         U32 fwsize = sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP) +
127             (sizeof (MR_LD_SPAN_MAP) * pFwRaidMap->ldCount);

129         if (pFwRaidMap->totalSize != fwsize) {

131                 con_log(CL_ANN1, (CE_NOTE,
132                     "map info structure size 0x%x is "
133                     "not matching with ld count\n", fwsize));
134                 /* sizeof (foo) returns size_t, which is *LONG*. */
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                     ((sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP)) +\
132                     (sizeof (MR_LD_SPAN_MAP) * pFwRaidMap->ldCount))));

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

138                 return (0);
139         }

141         mr_update_load_balance_params(map, lbInfo);

143         return (1);
144 }

146 U32
147 MR_GetSpanBlock(U32 ld, U64 row, U64 *span_blk, MR_FW_RAID_MAP_ALL *map,
148     int *div_error)
146 MR_GetSpanBlock(U32 ld, U64 row, U64 *span_blk, MR_FW_RAID_MAP_ALL *map, int *di
149 {
150         MR_SPAN_BLOCK_INFO *pSpanBlock = MR_LdSpanInfoGet(ld, map);
151         MR_QUAD_ELEMENT *qe;
149         MR_QUAD_ELEMENT *quad;
152         MR_LD_RAID      *raid = MR_LdRaidGet(ld, map);
153         U32             span, j;

155         for (span = 0; span < raid->spanDepth; span++, pSpanBlock++) {
156                 for (j = 0; j < pSpanBlock->block_span_info.noElements; j++) {
157                         qe = &pSpanBlock->block_span_info.quads[j];
158                         if (qe->diff == 0) {
159                                 *div_error = 1;
155                         quad = &pSpanBlock->block_span_info.quad[j];
```



```
156                         if(quad->diff == 0) {
157                                 *div_error=1;
160                                 return (span);
161                         }
162                         if (qe->logStart <= row && row <= qe->logEnd &&
163                             (((row - qe->logStart) % qe->diff)) == 0) {
160                         if (quad->logStart <= row &&
161                             row <= quad->logEnd &&
162                             (((row-quad->logStart) % quad->diff)) == 0) {
164                                 if (span_blk != NULL) {
165                                         U64     blk;
166                                         blk = ((row - qe->logStart) /
167                                             (qe->diff));
165                                         blk = ((row-quad->logStart) /
166                                             (quad->diff));

169                                         blk = (blk + qe->offsetInSpan) <<
170                                             raid->stripeShift;
168                                         blk = (blk + quad->offsetInSpan) << raid
171                                         *span_blk = blk;
172                                 }
173                                 return (span);
174                         }
175                 }
176         }
177         return (span);
178 }


181 /*
182  * ****************************************************************
183  *
184  * This routine calculates the arm, span and block for
185  * the specified stripe and reference in stripe.
186  *
187  * Inputs :
188  *
189  *    ld   - Logical drive number
190  *    stripRow        - Stripe number
191  *    stripRef    - Reference in stripe
192  *
193  * Outputs :
194  *
195  *    span          - Span number
196  *    block         - Absolute Block number in the physical disk
197  */
198 U8
199 MR_GetPhyParams(struct mrsas_instance *instance, U32 ld, U64 stripRow,
200     U16 stripRef, U64 *pdBlock, U16 *pDevHandle,
201     MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context, MR_FW_RAID_MAP_ALL *map)
197 MR_GetPhyParams(struct mrsas_instance *instance, U32 ld, U64 stripRow, U16 strip
198 U64 *pdBlock, U16 *pDevHandle, MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context,
199 MR_FW_RAID_MAP_ALL *map)
202 {
203         MR_LD_RAID      *raid = MR_LdRaidGet(ld, map);
204         U32             pd, arRef;
205         U8              physArm, span;
206         U64             row;
207         int             error_code = 0;
205         int             error_code=0;
208         U8              retval = TRUE;
209         U32             rowMod;
210         U32             armQ;
211         U32             arm;

213         ASSERT(raid->rowDataSize != 0);
```

```
        row = (stripRow / raid->rowDataSize);

        if (raid->level == 6) {
                U32 logArm =  (stripRow % (raid->rowDataSize));

                if (raid->rowSize == 0) {
                        return (FALSE);
                }
                rowMod = (row % (raid->rowSize));
                armQ = raid->rowSize-1-rowMod;
                arm = armQ + 1 + logArm;
                arm = armQ+1+logArm;
                if (arm >= raid->rowSize)
                        arm -= raid->rowSize;
                physArm = (U8)arm;
        } else {
                if (raid->modFactor == 0)
                        return (FALSE);
                        return FALSE;
                physArm = MR_LdDataArmGet(ld,
                    (stripRow % (raid->modFactor)), map);
        }
        if (raid->spanDepth == 1) {
                span = 0;
                *pdBlock = row << raid->stripeShift;
        } else
                span = (U8)MR_GetSpanBlock(ld, row, pdBlock, map, &error_code);

        if (error_code == 1)
                return (FALSE);
            return FALSE;

        /* Get the array on which this span is present. */
        // Get the array on which this span is present.
        arRef           = MR_LdSpanArrayGet(ld, span, map);
        /* Get the Pd. */
        // Get the Pd.
        pd              = MR_ArPdGet(arRef, physArm, map);
        /* Get dev handle from Pd. */
        // Get dev handle from Pd.
        if (pd != MR_PD_INVALID) {
                *pDevHandle     = MR_PdDevHandleGet(pd, map);
        } else {
                *pDevHandle = MR_PD_INVALID; /* set dev handle as invalid. */
                if ((raid->level >= 5) &&
        }
    else {
            *pDevHandle = MR_PD_INVALID; // set dev handle as invalid.
            if ( (raid->level >= 5) &&
                    ((instance->device_id != PCI_DEVICE_ID_LSI_INVADER) ||
                    (instance->device_id == PCI_DEVICE_ID_LSI_INVADER &&
                    raid->regTypeReqOnRead != REGION_TYPE_UNUSED))) {
                                raid->regTypeReqOnRead != REGION_TYPE_UNUSED)) )
                        pRAID_Context->regLockFlags = REGION_TYPE_EXCLUSIVE;
                } else if (raid->level == 1) {
                        /* Get Alternate Pd. */
                        pd = MR_ArPdGet(arRef, physArm + 1, map);
                        /* Get dev handle from Pd. */
            else if (raid->level == 1) {
                    pd = MR_ArPdGet(arRef, physArm + 1, map); // Get Alternate P
                    if (pd != MR_PD_INVALID)
                            *pDevHandle = MR_PdDevHandleGet(pd, map);
                        *pDevHandle = MR_PdDevHandleGet(pd, map); // Get dev
                }
        }
```



```
        *pdBlock += stripRef + MR_LdSpanPtrGet(ld, span, map)->startBlk;

        pRAID_Context->spanArm = (span << RAID_CTX_SPANARM_SPAN_SHIFT) |
            physArm;
        pRAID_Context->spanArm = (span << RAID_CTX_SPANARM_SPAN_SHIFT) | physArm

        return (retval);
        return retval;
}



/*
 * ****************************************************************************
 *
 * MR_BuildRaidContext function
 *
 * This function will initiate command processing.  The start/end row and strip
 * information is calculated then the lock is acquired.
 * This function will return 0 if region lock
 * was acquired OR return num strips ???
 */

U8
MR_BuildRaidContext(struct mrsas_instance *instance,
    struct IO_REQUEST_INFO *io_info, MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context,
    MR_FW_RAID_MAP_ALL *map)
MR_BuildRaidContext(struct mrsas_instance *instance, struct IO_REQUEST_INFO *io_
MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context, MR_FW_RAID_MAP_ALL *map)
{
        MR_LD_RAID      *raid;
        U32             ld, stripSize, stripe_mask;
        U64             endLba, endStrip, endRow;
        U64             start_row, start_strip;
        REGION_KEY      regStart;
        REGION_LEN      regSize;
        U8              num_strips, numRows;
        U16             ref_in_start_stripe;
        U16             ref_in_end_stripe;

        U64             ldStartBlock;
        U32             numBlocks, ldTgtId;
        U8              isRead;
        U8              retval = 0;

        ldStartBlock = io_info->ldStartBlock;
        numBlocks = io_info->numBlocks;
        ldTgtId = io_info->ldTgtId;
        isRead = io_info->isRead;

        if (map == NULL) {
        if ( map == NULL ) {
                io_info->fpOkForIo = FALSE;
                return (FALSE);
        }

        ld = MR_TargetIdToLdGet(ldTgtId, map);

        if (ld >= MAX_LOGICAL_DRIVES) {
                io_info->fpOkForIo = FALSE;
                return (FALSE);
        }

        raid = MR_LdRaidGet(ld, map);
```

```
327         stripSize = 1 << raid->stripeShift;
328         stripe_mask = stripSize-1;
329         /*
330          * calculate starting row and stripe, and number of strips and rows
331          */
332         start_strip             = ldStartBlock >> raid->stripeShift;
333         ref_in_start_stripe     = (U16)(ldStartBlock & stripe_mask);
334         endLba                  = ldStartBlock + numBlocks - 1;
335         ref_in_end_stripe       = (U16)(endLba & stripe_mask);
336         endStrip                = endLba >> raid->stripeShift;
337         num_strips              = (U8)(endStrip - start_strip + 1);
338         /* Check to make sure is not dividing by zero */
331         /* Check to make sure is not deviding by zero */
339         if (raid->rowDataSize == 0)
340                 return (FALSE);
333                 return FALSE;
341         start_row               =  (start_strip / raid->rowDataSize);
342         endRow                  =  (endStrip  / raid->rowDataSize);
343         /* get the row count */
336         // get the row count
344         numRows                 = (U8)(endRow - start_row + 1);

346         /*
347          * calculate region info.
348          */
349         regStart        = start_row << raid->stripeShift;
350         regSize         = stripSize;

352         /* Check if we can send this I/O via FastPath */
353         if (raid->capability.fpCapable) {
354                 if (isRead) {
347                 if (isRead)
355                         io_info->fpOkForIo = (raid->capability.fpReadCapable &&
356                             ((num_strips == 1) ||
357                             raid->capability.fpReadAcrossStripe));
358                 } else {
359                         io_info->fpOkForIo =
360                             (raid->capability.fpWriteCapable &&
351                 else
352                         io_info->fpOkForIo = (raid->capability.fpWriteCapable &&
361                             ((num_strips == 1) ||
362                             raid->capability.fpWriteAcrossStripe));
363                 }
364         } else
365                 io_info->fpOkForIo = FALSE;


368         /*
369          * Check for DIF support
370          */
371         if (!raid->capability.ldPiMode) {
362         if ( !raid->capability.ldPiMode ) {
372                 io_info->ldPI = FALSE;
373         } else {
374                 io_info->ldPI = TRUE;
375         }

377         if (numRows == 1) {
378                 if (num_strips == 1) {
379                         regStart += ref_in_start_stripe;
380                         regSize = numBlocks;
381                 }
382         } else {
383                 if (start_strip == (start_row + 1) * raid->rowDataSize - 1) {
384                         regStart += ref_in_start_stripe;
385                 regSize = stripSize - ref_in_start_stripe;
```



```
386                 }

388                 if (numRows > 2) {
389                         regSize += (numRows - 2) << raid->stripeShift;
380                         regSize += (numRows-2) << raid->stripeShift;
390                 }

392                 if (endStrip == endRow * raid->rowDataSize) {
393                         regSize += ref_in_end_stripe + 1;
383                 if (endStrip == endRow*raid->rowDataSize) {
384                         regSize += ref_in_end_stripe+1;
394                 } else {
395                         regSize += stripSize;
396                 }
397         }

399         pRAID_Context->timeoutValue = map->raidMap.fpPdIoTimeoutSec;

401         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
402                 pRAID_Context->regLockFlags = (isRead) ?
403                     raid->regTypeReqOnRead : raid->regTypeReqOnWrite;
404         } else {
405                 pRAID_Context->regLockFlags = (isRead) ?
406                     REGION_TYPE_SHARED_READ : raid->regTypeReqOnWrite;
407         }
392         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER)
393                 pRAID_Context->regLockFlags =
394                         (isRead)? raid->regTypeReqOnRead : raid->regTypeReqOnWri
395         else
396                 pRAID_Context->regLockFlags     =
397                         (isRead) ? REGION_TYPE_SHARED_READ : raid->regTypeReqOnW

409         pRAID_Context->ldTargetId = raid->targetId;
410         pRAID_Context->regLockRowLBA = regStart;
411         pRAID_Context->regLockLength = regSize;
412         pRAID_Context->configSeqNum = raid->seqNum;

414         /*
415          * Get Phy Params only if FP capable,
416          * or else leave it to MR firmware to do the calculation.
417          */
418         if (io_info->fpOkForIo) {
419                 /* if fast path possible then get the physical parameters */
420                 retval = MR_GetPhyParams(instance, ld, start_strip,
421                     ref_in_start_stripe, &io_info->pdBlock,
422                     &io_info->devHandle, pRAID_Context, map);
410                 retval = MR_GetPhyParams(instance, ld, start_strip, ref_in_start
411                     &io_info->pdBlock, &io_info->devHandle,
412                     pRAID_Context, map);

424                 /* If IO on an invalid Pd, then FP is not possible. */
425                 if (io_info->devHandle == MR_PD_INVALID)
414                 if (io_info->devHandle == MR_PD_INVALID) // If IO on an invalid
426                         io_info->fpOkForIo = FALSE;

428                 return (retval);
417                 return retval;

430         } else if (isRead) {
431                 uint_t stripIdx;

433                 for (stripIdx = 0; stripIdx < num_strips; stripIdx++) {
434                         if (!MR_GetPhyParams(instance, ld,
435                             start_strip + stripIdx, ref_in_start_stripe,
436                             &io_info->pdBlock, &io_info->devHandle,
437                             pRAID_Context, map)) {
```

```
438                                 return (TRUE);
420                 uint stripIdx;
421                 for (stripIdx=0; stripIdx<num_strips; stripIdx++) {
422                         if (!MR_GetPhyParams(instance, ld, start_strip + stripId
423                             &io_info->pdBlock, &io_info->devHandle, pRAID_Contex
424                         return TRUE;
439                         }
440                 }
441         }
442         return (TRUE);
443 }


446 void
447 mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *map,
448     PLD_LOAD_BALANCE_INFO lbInfo)
432 mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbI
449 {
450         int ldCount;
451         U16 ld;
452         MR_LD_RAID *raid;

454         for (ldCount = 0; ldCount < MAX_LOGICAL_DRIVES; ldCount++) {
438         for (ldCount = 0; ldCount < MAX_LOGICAL_DRIVES; ldCount++)
439         {
455                 ld = MR_TargetIdToLdGet(ldCount, map);

457                 if (ld >= MAX_LOGICAL_DRIVES) {
458                         con_log(CL_ANN1,
459                             (CE_NOTE, "mrsas: ld=%d Invalid ld \n", ld));
443                         con_log(CL_ANN1, (CE_NOTE,
444                                 "mrsas: ld=%d Invalid ld \n", ld));
460                         continue;
461                 }

463                 raid = MR_LdRaidGet(ld, map);

465                 /* Two drive Optimal RAID 1 */
466                 if ((raid->level == 1) && (raid->rowSize == 2) &&
467                     (raid->spanDepth == 1) &&
468                     raid->ldState == MR_LD_STATE_OPTIMAL) {
451                 if ((raid->level == 1)  &&  (raid->rowSize == 2) && (raid->spanD
452                     && raid->ldState == MR_LD_STATE_OPTIMAL) {
469                         U32 pd, arRef;

471                         lbInfo[ldCount].loadBalanceFlag = 1;
456                         //U8 physArm = 0;

473                         /* Get the array on which this span is present. */
474                         arRef = MR_LdSpanArrayGet(ld, 0, map);
458                         arRef = MR_LdSpanArrayGet(ld, 0, map);    // Get the arr

476                         pd = MR_ArPdGet(arRef, 0, map);     /* Get the Pd. */
477                         /* Get dev handle from Pd. */
478                         lbInfo[ldCount].raid1DevHandle[0] =
479                             MR_PdDevHandleGet(pd, map);
460                         pd = MR_ArPdGet(arRef, 0, map);     // Get the Pd.
461                         lbInfo[ldCount].raid1DevHandle[0] = MR_PdDevHandleGet(pd

481                         pd = MR_ArPdGet(arRef, 1, map);     /* Get the Pd. */
482                         /* Get dev handle from Pd. */
483                         lbInfo[ldCount].raid1DevHandle[1] =
484                             MR_PdDevHandleGet(pd, map);
485                         con_log(CL_ANN1, (CE_NOTE,
463                         pd = MR_ArPdGet(arRef, 1, map);     // Get the Pd.
464                         lbInfo[ldCount].raid1DevHandle[1] = MR_PdDevHandleGet(pd
```



```
465                         con_log(CL_ANN1, (CE_NOTE,\
486                             "mrsas: ld=%d load balancing enabled \n", ldCount));
487                 } else {
488                         lbInfo[ldCount].loadBalanceFlag = 0;
489                 }
490         }
491 }


494 U8
495 megasas_get_best_arm(PLD_LOAD_BALANCE_INFO lbInfo, U8 arm, U64 block,
496     U32 count)
474 U8 megasas_get_best_arm(PLD_LOAD_BALANCE_INFO lbInfo, U8 arm, U64 block, U32 cou
497 {
498         U16 pend0, pend1;
499         U64 diff0, diff1;
500         U8 bestArm;

502         /* get the pending cmds for the data and mirror arms */
503         pend0 = lbInfo->scsi_pending_cmds[0];
504         pend1 = lbInfo->scsi_pending_cmds[1];

506         /* Determine the disk whose head is nearer to the req. block */
507         diff0 = ABS_DIFF(block, lbInfo->last_accessed_block[0]);
508         diff1 = ABS_DIFF(block, lbInfo->last_accessed_block[1]);
509         bestArm = (diff0 <= diff1 ? 0 : 1);

511         if ((bestArm == arm && pend0 > pend1 + 16) ||
512             (bestArm != arm && pend1 > pend0 + 16)) {
489     if ((bestArm == arm && pend0 > pend1 + 16)  || (bestArm != arm && pend1 > pe
513                 bestArm ^= 1;
514         }

516         /* Update the last accessed block on the correct pd */
517         lbInfo->last_accessed_block[bestArm] = block + count - 1;
518         return (bestArm);
494     return bestArm;
519 }

521 U16
522 get_updated_dev_handle(PLD_LOAD_BALANCE_INFO lbInfo,
523     struct IO_REQUEST_INFO *io_info)
497 U16 get_updated_dev_handle(PLD_LOAD_BALANCE_INFO lbInfo, struct IO_REQUEST_INFO
524 {
525         U8 arm, old_arm;
526         U16 devHandle;

528         old_arm = lbInfo->raid1DevHandle[0] == io_info->devHandle ? 0 : 1;

530         /* get best new arm */
531         arm  = megasas_get_best_arm(lbInfo, old_arm, io_info->ldStartBlock,
532             io_info->numBlocks);
505         arm  = megasas_get_best_arm(lbInfo, old_arm, io_info->ldStartBlock, io_i

534         devHandle = lbInfo->raid1DevHandle[arm];

536         lbInfo->scsi_pending_cmds[arm]++;

538         return (devHandle);
511         return devHandle;
539 }
______unchanged_portion_omitted_
```

```
**********************************************************
    6952 Tue Nov  6 14:29:38 2012
new/./ld_pd_map.h
**********************************************************
```

```
/*
 * ld_pd_map.h
 *
 * 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
 */

#ifndef _LD_PD_MAP
#define _LD_PD_MAP
#ifndef INCLUDE_LD_PD_MAP
#define INCLUDE_LD_PD_MAP
#include <sys/scsi/scsi.h>
#include "fusion.h"

struct mrsas_instance;  /* This will be defined in mr_sas.h */

/* raid->write_mode; raid->read_ahead; dcmd->state */
/* Write through */
#define WRITE_THROUGH                           0
/* Delayed Write */
#define WRITE_BACK                              1

/* SCSI CDB definitions */
#define READ_6          0x08
#define READ_16         0x88
#define READ_10         0x28
#define READ_12         0xA8
#define WRITE_16        0x8A
#define WRITE_10        0x2A

/* maximum disks per array */
// maximum disks per array
#define MAX_ROW_SIZE                            32
/* maximum spans per logical drive */
// maximum spans per logical drive
#define MAX_SPAN_DEPTH                          8
#define MEGASAS_LOAD_BALANCE_FLAG               0x1
#define MR_DEFAULT_IO_TIMEOUT   20


union desc_value {
        U64 word;
        struct {
                U32 low;
                U32 high;
        } u1;
};
```

*unchanged_portion_omitted_*

```
/*
 * Raid Context structure which describes MegaRAID specific IO Parameters
 * Raid Context structure which describes MegaRAID specific IO Paramenters
 * This resides at offset 0x60 where the SGL normally starts in MPT IO Frames
```



```
 */
typedef struct _MPI2_SCSI_IO_VENDOR_UNIQUE {
        U8 nsegType;            /* 0x00 nseg[7:4], Type[3:0] */
        U8 resvd0;              /* 0x01 */
        U8              nsegType;                       // 0x00 nseg[7:4], Type[
        U8              resvd0;                         // 0x01
        U16 timeoutValue;       /* 0x02 -0x03 */
        U8 regLockFlags;        /* 0x04 */
        U8 reservedForHw1;      /* 0x05 */
        U16 ldTargetId;         /* 0x06 - 0x07 */
        U64 regLockRowLBA;      /* 0x08 - 0x0F */
        U32 regLockLength;      /* 0x10 - 0x13 */
        U16 nextLMId;           /* 0x14 - 0x15 */
        U8 extStatus;           /* 0x16 */
        U8 status;              /* 0x17 status */
        U8 RAIDFlags;           /* 0x18 resvd[7:6], ioSubType[5:4], */
                                /* resvd[3:1], preferredCpu[0] */
        U8 numSGE;              /* 0x19 numSge; not including chain entries */
    U8      RAIDFlags;          /* 0x18 resvd[7:6], ioSubType[5:4], resvd[3:1],
    U8      numSGE;             /* 0x19 numSge; not including chain entries*/
        U16 configSeqNum;       /* 0x1A -0x1B */
        U8 spanArm;             /* 0x1C span[7:5], arm[4:0] */
        U8 resvd2[3];           /* 0x1D-0x1f */
} MPI2_SCSI_IO_VENDOR_UNIQUE, MPI25_SCSI_IO_VENDOR_UNIQUE;
```

*unchanged_portion_omitted_*

```
Mpi2RaidSCSIIORequest_t, MPI2_POINTER pMpi2RaidSCSIIORequest_t;

/*
 * define region lock types
 */
typedef enum    _REGION_TYPE {
        REGION_TYPE_UNUSED      = 0,    /* lock is currently not active */
        REGION_TYPE_SHARED_READ = 1,    /* shared lock (for reads) */
        REGION_TYPE_UNUSED      = 0,    // lock is currently not active
        REGION_TYPE_SHARED_READ = 1,    // shared lock (for reads)
        REGION_TYPE_SHARED_WRITE = 2,
        REGION_TYPE_EXCLUSIVE   = 3     /* exclusive lock (for writes) */
        REGION_TYPE_EXCLUSIVE   = 3,    // exclusive lock (for writes)
} REGION_TYPE;


#define DM_PATH_MAXPATH         2
#define DM_PATH_FIRSTPATH       0
#define DM_PATH_SECONDPATH      1

/* declare valid Region locking values */
// declare valid Region locking values
typedef enum _REGION_LOCK {
        REGION_LOCK_BYPASS              = 0,
        /* for RAID 6 single-drive failure */
        // for RAID 6 single-drive failure
        REGION_LOCK_UNCOND_SHARED_READ  = 1,
        REGION_LOCK_UNCOND_SHARED_WRITE = 2,
        REGION_LOCK_UNCOND_SHARED_OTHER = 3,
        REGION_LOCK_UNCOND_SHARED_EXCLUSIVE = 0xFF
} REGION_LOCK;
```

*unchanged_portion_omitted_*

```
#pragma pack()

enum {
        MRSAS_SCSI_VARIABLE_LENGTH_CMD          = 0x7F,
        MRSAS_SCSI_SERVICE_ACTION_READ32        = 0x9,
        MRSAS_SCSI_SERVICE_ACTION_WRITE32       = 0xB,
        MRSAS_SCSI_ADDL_CDB_LEN                 = 0x18,
        MRSAS_RD_WR_PROTECT                     = 0x20,
```

```
231         MRSAS_EEDPBLOCKSIZE                     = 512
228         MRSAS_EEDPBLOCKSIZE                     = 512,
232 };


235 #define IEEE_SGE_FLAGS_ADDR_MASK        (0x03)
236 #define IEEE_SGE_FLAGS_SYSTEM_ADDR      (0x00)
237 #define IEEE_SGE_FLAGS_IOCDDR_ADDR      (0x01)
238 #define IEEE_SGE_FLAGS_IOCPLB_ADDR      (0x02)
239 #define IEEE_SGE_FLAGS_IOCPLBNTA_ADDR   (0x03)
240 #define IEEE_SGE_FLAGS_CHAIN_ELEMENT    (0x80)
241 #define IEEE_SGE_FLAGS_END_OF_LIST      (0x40)


244 U8 MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo);
245 U16 MR_CheckDIF(U32, MR_FW_RAID_MAP_ALL *);
246 U8 MR_BuildRaidContext(struct mrsas_instance *, struct IO_REQUEST_INFO *,
247     MPI2_SCSI_IO_VENDOR_UNIQUE *, MR_FW_RAID_MAP_ALL *);

249 #endif /* _LD_PD_MAP */
243 #endif // INCLUDE_LD_PD_MAP
```

```
**********************************************************
  220825 Tue Nov  6 14:29:38 2012
new/./mr_sas.c
**********************************************************
```

```
/*
 * mr_sas.c: source for mr_sas driver
 *
 * 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
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the author nor the names of its contributors may be
 *    used to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 Bayard G. Bell. All rights reserved.
 * Copyright 2012 Nexenta System, Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/stat.h>
#include <sys/mkdev.h>
```



```
#include <sys/pci.h>
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/atomic.h>
#include <sys/signal.h>
#include <sys/byteorder.h>
#include <sys/sdt.h>
#include <sys/fs/dv_node.h>     /* devfs_clean */

#include "mr_sas.h"

/*
 * FMA header files
 */
#include <sys/ddifm.h>
#include <sys/fm/protocol.h>
#include <sys/fm/util.h>
#include <sys/fm/io/ddi.h>

/*
 * Local static data
 */
static void     *mrsas_state = NULL;
static volatile boolean_t       mrsas_relaxed_ordering = B_TRUE;
static volatile boolean_t       mrsas_relaxed_ordering = 0;
volatile int    debug_level_g = CL_NONE;

static volatile int     msi_enable = 1;
static volatile int     ctio_enable = 1;

/* Default Timeout value to issue online controller reset */
volatile int  debug_timeout_g  = 0xF0;          /* 0xB4; */
volatile int  debug_timeout_g  = 0xF0;          //0xB4;
/* Simulate consecutive firmware fault */
static volatile int  debug_fw_faults_after_ocr_g  = 0;
#ifdef OCRDEBUG
/* Simulate three consecutive timeout for an IO */
static volatile int  debug_consecutive_timeout_after_ocr_g  = 0;
#endif

/* Enable OCR on firmware fault */
static volatile int  debug_support_ocr_isr_g  = 0;
#pragma weak scsi_hba_open
#pragma weak scsi_hba_close
#pragma weak scsi_hba_ioctl

/* Local static prototypes. */
static int      mrsas_getinfo(dev_info_t *, ddi_info_cmd_t,  void *, void **);
static int      mrsas_attach(dev_info_t *, ddi_attach_cmd_t);
#ifdef __sparc
static int      mrsas_reset(dev_info_t *, ddi_reset_cmd_t);
#else
static int      mrsas_quiesce(dev_info_t *);
#endif
static int      mrsas_detach(dev_info_t *, ddi_detach_cmd_t);
static int      mrsas_open(dev_t *, int, int, cred_t *);
static int      mrsas_close(dev_t, int, int, cred_t *);
static int      mrsas_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

static int      mrsas_tran_tgt_init(dev_info_t *, dev_info_t *,
                    scsi_hba_tran_t *, struct scsi_device *);
static struct scsi_pkt *mrsas_tran_init_pkt(struct scsi_address *, register
                    struct scsi_pkt *, struct buf *, int, int, int, int,
                    int (*)(), caddr_t);
static int      mrsas_tran_start(struct scsi_address *,
```

```
124                     register struct scsi_pkt *);
125 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
126 static int      mrsas_tran_reset(struct scsi_address *, int);
127 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
128 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
129 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
130                     struct scsi_pkt *);
131 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
132 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
133 static int      mrsas_tran_quiesce(dev_info_t *dip);
134 static int      mrsas_tran_unquiesce(dev_info_t *dip);
135 static uint_t   mrsas_isr();
136 static uint_t   mrsas_softintr();
137 static void     mrsas_undo_resources(dev_info_t *, struct mrsas_instance *);
138 static struct mrsas_cmd *get_mfi_pkt(struct mrsas_instance *);
139 static void     return_mfi_pkt(struct mrsas_instance *,
140                     struct mrsas_cmd *);

142 static void     free_space_for_mfi(struct mrsas_instance *);
143 static uint32_t read_fw_status_reg_ppc(struct mrsas_instance *);
144 static void     issue_cmd_ppc(struct mrsas_cmd *, struct mrsas_instance *);
145 static int      issue_cmd_in_poll_mode_ppc(struct mrsas_instance *,
146                     struct mrsas_cmd *);
147 static int      issue_cmd_in_sync_mode_ppc(struct mrsas_instance *,
148                     struct mrsas_cmd *);
149 static void     enable_intr_ppc(struct mrsas_instance *);
150 static void     disable_intr_ppc(struct mrsas_instance *);
151 static int      intr_ack_ppc(struct mrsas_instance *);
152 static void     flush_cache(struct mrsas_instance *instance);
153 void    display_scsi_inquiry(caddr_t);
154 static int      start_mfi_aen(struct mrsas_instance *instance);
155 static int      handle_drv_ioctl(struct mrsas_instance *instance,
156                     struct mrsas_ioctl *ioctl, int mode);
157 static int      handle_mfi_ioctl(struct mrsas_instance *instance,
158                     struct mrsas_ioctl *ioctl, int mode);
159 static int      handle_mfi_aen(struct mrsas_instance *instance,
160                     struct mrsas_aen *aen);
161 static struct mrsas_cmd *build_cmd(struct mrsas_instance *,
162     struct scsi_address *, struct scsi_pkt *, uchar_t *);
163 static int      alloc_additional_dma_buffer(struct mrsas_instance *);
164 static void     complete_cmd_in_sync_mode(struct mrsas_instance *,
165                     struct mrsas_cmd *);
166 static int      mrsas_kill_adapter(struct mrsas_instance *);
167 static int      mrsas_issue_init_mfi(struct mrsas_instance *);
168 static int      mrsas_reset_ppc(struct mrsas_instance *);
169 static uint32_t mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *);
170 static int      wait_for_outstanding(struct mrsas_instance *instance);
171 static int      register_mfi_aen(struct mrsas_instance *instance,
172                     uint32_t seq_num, uint32_t class_locale_word);
173 static int      issue_mfi_pthru(struct mrsas_instance *instance, struct
174                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
175 static int      issue_mfi_dcmd(struct mrsas_instance *instance, struct
176                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
177 static int      issue_mfi_smp(struct mrsas_instance *instance, struct
178                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
179 static int      issue_mfi_stp(struct mrsas_instance *instance, struct
180                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
181 static int      abort_aen_cmd(struct mrsas_instance *instance,
182                     struct mrsas_cmd *cmd_to_abort);

184 static void     mrsas_rem_intrs(struct mrsas_instance *instance);
185 static int      mrsas_add_intrs(struct mrsas_instance *instance, int intr_type);

187 static void     mrsas_tran_tgt_free(dev_info_t *, dev_info_t *,
188                     scsi_hba_tran_t *, struct scsi_device *);
189 static int      mrsas_tran_bus_config(dev_info_t *, uint_t,
```



```
190                     ddi_bus_config_op_t, void *, dev_info_t **);
191 static int      mrsas_parse_devname(char *, int *, int *);
192 static int      mrsas_config_all_devices(struct mrsas_instance *);
193 static int      mrsas_config_ld(struct mrsas_instance *, uint16_t,
194                         uint8_t, dev_info_t **);
195 static int      mrsas_name_node(dev_info_t *, char *, int);
196 static void     mrsas_issue_evt_taskq(struct mrsas_eventinfo *);
197 static void     free_additional_dma_buffer(struct mrsas_instance *);
198 static void io_timeout_checker(void *);
199 static void mrsas_fm_init(struct mrsas_instance *);
200 static void mrsas_fm_fini(struct mrsas_instance *);

202 static struct mrsas_function_template mrsas_function_template_ppc = {
203         .read_fw_status_reg = read_fw_status_reg_ppc,
204         .issue_cmd = issue_cmd_ppc,
205         .issue_cmd_in_sync_mode = issue_cmd_in_sync_mode_ppc,
206         .issue_cmd_in_poll_mode = issue_cmd_in_poll_mode_ppc,
207         .enable_intr = enable_intr_ppc,
208         .disable_intr = disable_intr_ppc,
209         .intr_ack = intr_ack_ppc,
210         .init_adapter = mrsas_init_adapter_ppc
 77 //      .reset_adapter = mrsas_reset_adapter_ppc
211 };


214 static struct mrsas_function_template mrsas_function_template_fusion = {
215         .read_fw_status_reg = tbolt_read_fw_status_reg,
216         .issue_cmd = tbolt_issue_cmd,
217         .issue_cmd_in_sync_mode = tbolt_issue_cmd_in_sync_mode,
218         .issue_cmd_in_poll_mode = tbolt_issue_cmd_in_poll_mode,
219         .enable_intr = tbolt_enable_intr,
220         .disable_intr = tbolt_disable_intr,
221         .intr_ack = tbolt_intr_ack,
222         .init_adapter = mrsas_init_adapter_tbolt
 90 //      .reset_adapter = mrsas_reset_adapter_tbolt
223 };
_____unchanged_portion_omitted_

241 int32_t mrsas_max_cap_maxxfer = 0x1000000;

243 /*
244  * Fix for: Thunderbolt controller IO timeout when IO write size is 1MEG,
245  * Limit size to 256K
246  */
111 //Fix for: Thunderbolt controller IO timeout when IO write size is 1MEG, Limit s
247 uint32_t mrsas_tbolt_max_cap_maxxfer = (512 * 512);

249 /*
250  * cb_ops contains base level routines
251  */
252 static struct cb_ops mrsas_cb_ops = {
253         mrsas_open,             /* open */
254         mrsas_close,            /* close */
255         nodev,                  /* strategy */
256         nodev,                  /* print */
257         nodev,                  /* dump */
258         nodev,                  /* read */
259         nodev,                  /* write */
260         mrsas_ioctl,            /* ioctl */
261         nodev,                  /* devmap */
262         nodev,                  /* mmap */
263         nodev,                  /* segmap */
264         nochpoll,               /* poll */
265         nodev,                  /* cb_prop_op */
266         0,                      /* streamtab  */
267         D_NEW | D_HOTPLUG,      /* cb_flag */
```

```
268         CB_REV,                 /* cb_rev */
269         nodev,                  /* cb_aread */
270         nodev                   /* cb_awrite */
271 };

273 /*
274  * dev_ops contains configuration routines
275  */
276 static struct dev_ops mrsas_ops = {
277         DEVO_REV,               /* rev, */
278         0,                      /* refcnt */
279         mrsas_getinfo,          /* getinfo */
280         nulldev,                /* identify */
281         nulldev,                /* probe */
282         mrsas_attach,           /* attach */
283         mrsas_detach,           /* detach */
284 #ifdef  __sparc
285         mrsas_reset,            /* reset */
286 #else   /* __sparc */
149 #if defined(__SunOS_5_11)
287         nodev,
288 #endif  /* __sparc */
151 #else
152         mrsas_reset,            /* reset */
153 #endif  /* defined(__SunOS_5_11) */
289         &mrsas_cb_ops,          /* char/block ops */
290         NULL,                   /* bus ops */
291         NULL,                   /* power */
292 #ifdef __sparc
293         ddi_quiesce_not_needed
294 #else   /* __sparc */
157 #ifdef __SunOS_5_11
295         mrsas_quiesce   /* quiesce */
296 #endif  /* __sparc */
159 #endif /*__SunOS_5_11 */

297 };

163 char _depends_on[] = "misc/scsi";

299 static struct modldrv modldrv = {
300         &mod_driverops,         /* module type - driver */
301         MRSAS_VERSION,
302         &mrsas_ops,             /* driver ops */
303 };
```

*_____unchanged_portion_omitted_*

```
318 /* Use the LSI Fast Path for the 2208 (tbolt) commands. */

319 unsigned int enable_fp = 1;


322 /*
323  * ************************************************************************** *
324  *                                                                            *
325  *         common entry points - for loadable kernel modules                  *
326  *                                                                            *
327  * ************************************************************************** *
328  */

330 /*
331  * _init - initialize a loadable module
332  * @void
333  *
334  * The driver should perform any one-time resource allocation or data
335  * initialization during driver loading in _init(). For example, the driver
```



```
336  * should initialize any mutexes global to the driver in this routine.
337  * The driver should not, however, use _init() to allocate or initialize
338  * anything that has to do with a particular instance of the device.
339  * Per-instance initialization must be done in attach().
340  */
341 int
342 _init(void)
343 {
344         int ret;

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

348         ret = ddi_soft_state_init(&mrsas_state,
349             sizeof (struct mrsas_instance), 0);

351         if (ret != DDI_SUCCESS) {
352                 cmn_err(CE_WARN, "mr_sas: could not init state");
353                 return (ret);
354         }

356         if ((ret = scsi_hba_init(&modlinkage)) != DDI_SUCCESS) {
357                 cmn_err(CE_WARN, "mr_sas: could not init scsi hba");
358                 ddi_soft_state_fini(&mrsas_state);
359                 return (ret);
360         }

362         ret = mod_install(&modlinkage);

364         if (ret != DDI_SUCCESS) {
365                 cmn_err(CE_WARN, "mr_sas: mod_install failed");
366                 scsi_hba_fini(&modlinkage);
367                 ddi_soft_state_fini(&mrsas_state);
368         }

370         return (ret);
371 }
```

*_____unchanged_portion_omitted_*

```
388 /*
389  * _fini - prepare a loadable module for unloading
390  * @void
391  *
392  * In _fini(), the driver should release any resources that were allocated in
393  * _init(). The driver must remove itself from the system module list.
394  */
395 int
396 _fini(void)
397 {
398         int ret;

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

402         if ((ret = mod_remove(&modlinkage)) != DDI_SUCCESS) {
403                 con_log(CL_ANN1,
404                     (CE_WARN, "_fini: mod_remove() failed, error 0x%X", ret));
268         if ((ret = mod_remove(&modlinkage)) != DDI_SUCCESS)
269         {
270                 con_log(CL_ANN1, (CE_WARN, "_fini: mod_remove() failed, error 0x
405                 return (ret);
406         }

408         scsi_hba_fini(&modlinkage);
409         con_log(CL_DLEVEL1, (CE_NOTE, "_fini: scsi_hba_fini() done."));

411         ddi_soft_state_fini(&mrsas_state);
412         con_log(CL_DLEVEL1, (CE_NOTE, "_fini: ddi_soft_state_fini() done."));
```

```
414         return (ret);
415 }


418 /*
419  * ************************************************************************** *
420  *                                                                            *
421  *               common entry points - for autoconfiguration                  *
422  *                                                                            *
423  * ************************************************************************** *
424  */
425 /*
426  * attach - adds a device to the system as part of initialization
427  * @dip:
428  * @cmd:
429  *
430  * The kernel calls a driver's attach() entry point to attach an instance of
431  * a device (for MegaRAID, it is instance of a controller) or to resume
432  * operation for an instance of a device that has been suspended or has been
433  * shut down by the power management framework
434  * The attach() entry point typically includes the following types of
435  * processing:
436  * - allocate a soft-state structure for the device instance (for MegaRAID,
437  *   controller instance)
438  * - initialize per-instance mutexes
439  * - initialize condition variables
440  * - register the device's interrupts (for MegaRAID, controller's interrupts)
441  * - map the registers and memory of the device instance (for MegaRAID,
442  *   controller instance)
443  * - create minor device nodes for the device instance (for MegaRAID,
444  *   controller instance)
445  * - report that the device instance (for MegaRAID, controller instance) has
446  *   attached
447  */
314 #if       __SunOS_5_11
315 #define   DDI_PM_RESUME DDI_PM_RESUME_OBSOLETE
316 #define   DDI_PM_SUSPEND DDI_PM_SUSPEND_OBSOLETE
317 #endif // __SunOS_5_11
448 static int
449 mrsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
450 {
451         int             instance_no;
452         int             nregs;
453         int             i = 0;
454         uint8_t         irq;
455         uint16_t        vendor_id;
456         uint16_t        device_id;
457         uint16_t        subsysvid;
458         uint16_t        subsysid;
459         uint16_t        command;
460         off_t           reglength = 0;
461         int             intr_types = 0;
462         char            *data;

464         scsi_hba_tran_t         *tran;
465         ddi_dma_attr_t  tran_dma_attr;
466         struct mrsas_instance   *instance;

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

470         /* CONSTCOND */
471         ASSERT(NO_COMPETING_THREADS);

473         instance_no = ddi_get_instance(dip);
```



```
475         /*
476          * check to see whether this device is in a DMA-capable slot.
477          */
478         if (ddi_slaveonly(dip) == DDI_SUCCESS) {
479                 cmn_err(CE_WARN,
480                     "mr_sas%d: Device in slave-only slot, unused",
481                     instance_no);
482                 return (DDI_FAILURE);
483         }

485         switch (cmd) {
486         case DDI_ATTACH:

487                 /* allocate the soft state for the instance */
488                 if (ddi_soft_state_zalloc(mrsas_state, instance_no)
489                     != DDI_SUCCESS) {
490                         cmn_err(CE_WARN,
491                             "mr_sas%d: Failed to allocate soft state",
492                             instance_no);

493                         return (DDI_FAILURE);
494                 }

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

499                 if (instance == NULL) {
500                         cmn_err(CE_WARN,
501                             "mr_sas%d: Bad soft state", instance_no);

502                         ddi_soft_state_free(mrsas_state, instance_no);

503                         return (DDI_FAILURE);
504                 }

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

506                 instance->unroll.softs  = 1;

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

515                         ddi_soft_state_free(mrsas_state, instance_no);
516                         return (DDI_FAILURE);
517                 }
395                         if (instance->pci_handle == NULL) {
396                                 cmn_err(CE_WARN,
397                                     "mr_sas%d: pci config setup failed ",
398                                     instance_no);
399                                 ddi_soft_state_free(mrsas_state, instance_no);
400                                 return (DDI_FAILURE);
401                         }


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

523                         pci_config_teardown(&instance->pci_handle);
524                         ddi_soft_state_free(mrsas_state, instance_no);
```

```
525                     return (DDI_FAILURE);
526             }

528             vendor_id = pci_config_get16(instance->pci_handle,
529                 PCI_CONF_VENID);
530             device_id = pci_config_get16(instance->pci_handle,
531                 PCI_CONF_DEVID);

533             subsysvid = pci_config_get16(instance->pci_handle,
534                 PCI_CONF_SUBVENID);
535             subsysid = pci_config_get16(instance->pci_handle,
536                 PCI_CONF_SUBSYSID);

538             pci_config_put16(instance->pci_handle, PCI_CONF_COMM,
539                 (pci_config_get16(instance->pci_handle,
540                 PCI_CONF_COMM) | PCI_COMM_ME));
541             irq = pci_config_get8(instance->pci_handle,
542                 PCI_CONF_ILINE);

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

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

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

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

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

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

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

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

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

587             default:
```



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

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

594             }

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

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

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

614             mrsas_fm_init(instance);

616             /* Setup register map */
617             if ((ddi_dev_regsize(instance->dip,
618                 REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
619                 reglength < MINIMUM_MFI_MEM_SZ) {
620                     goto fail_attach;
621             }
622             if (reglength > DEFAULT_MFI_MEM_SZ) {
623                     reglength = DEFAULT_MFI_MEM_SZ;
624                     con_log(CL_DLEVEL1, (CE_NOTE,
625                         "mr_sas: register length to map is 0x%lx bytes",
626                         reglength));
502                                 "mr_sas: register length to map is "
503                                 "0x%lx bytes", reglength));
627             }
628             if (ddi_regs_map_setup(instance->dip,
629                 REGISTER_SET_IO_2108, &instance->regmap, 0,
630                 reglength, &endian_attr, &instance->regmap_handle)
631                 != DDI_SUCCESS) {
632                     cmn_err(CE_WARN,
633                         "mr_sas: couldn't map control registers");
634                     goto fail_attach;
635             }
513                     if (instance->regmap_handle == NULL) {
514                             cmn_err(CE_WARN,
515                                 "mr_sas: couldn't map control registers");
516                             goto fail_attach;
517                     }

637             instance->unroll.regs = 1;

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

645             if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
```

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

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

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

666                         ddi_prop_free(data);
667                 }

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

671                 /* Check for all supported interrupt types */
672                 if (ddi_intr_get_supported_types(
673                     dip, &intr_types) != DDI_SUCCESS) {
674                         cmn_err(CE_WARN,
675                             "ddi_intr_get_supported_types() failed");
676                         goto fail_attach;
677                 }

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

682                 /* Initialize and Setup Interrupt handler */
683                 if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
684                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
685                             DDI_SUCCESS) {
568                                 if (mrsas_add_intrs(instance,
569                                     DDI_INTR_TYPE_MSIX) != DDI_SUCCESS) {
686                                 cmn_err(CE_WARN,
687                                     "MSIX interrupt query failed");
688                                 goto fail_attach;
689                         }
690                         instance->intr_type = DDI_INTR_TYPE_MSIX;
691                 } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
692                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
693                             DDI_SUCCESS) {
575                         } else if (msi_enable && (intr_types &
576                             DDI_INTR_TYPE_MSI)) {
577                                 if (mrsas_add_intrs(instance,
578                                     DDI_INTR_TYPE_MSI) != DDI_SUCCESS) {
694                                 cmn_err(CE_WARN,
695                                     "MSI interrupt query failed");
696                                 goto fail_attach;
697                         }
698                         instance->intr_type = DDI_INTR_TYPE_MSI;
699                 } else if (intr_types & DDI_INTR_TYPE_FIXED) {
700                         msi_enable = 0;
```



```
701                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
702                             DDI_SUCCESS) {
586                                 if (mrsas_add_intrs(instance,
587                                     DDI_INTR_TYPE_FIXED) != DDI_SUCCESS) {
703                                 cmn_err(CE_WARN,
704                                     "FIXED interrupt query failed");
705                                 goto fail_attach;
706                         }
707                         instance->intr_type = DDI_INTR_TYPE_FIXED;
708                 } else {
709                         cmn_err(CE_WARN, "Device cannot "
710                             "suppport either FIXED or MSI/X "
711                             "interrupts");
712                         goto fail_attach;
713                 }

715                 instance->unroll.intr = 1;

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

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

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

734                         goto fail_attach;
735                 }

737                 /* Initialize all Mutex */
738                 INIT_LIST_HEAD(&instance->completed_pool_list);
739                 mutex_init(&instance->completed_pool_mtx, NULL,
740                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
613                         mutex_init(&instance->completed_pool_mtx,
614                             "completed_pool_mtx", MUTEX_DRIVER,
615                             DDI_INTR_PRI(instance->intr_pri));

742                 mutex_init(&instance->sync_map_mtx, NULL,
743                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
617                         mutex_init(&instance->sync_map_mtx,
618                             "sync_map_mtx", MUTEX_DRIVER,
619                             DDI_INTR_PRI(instance->intr_pri));

745                 mutex_init(&instance->app_cmd_pool_mtx, NULL,
746                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
621                         mutex_init(&instance->app_cmd_pool_mtx,
622                             "app_cmd_pool_mtx", MUTEX_DRIVER,
623                             DDI_INTR_PRI(instance->intr_pri));

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

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

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

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

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

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

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

771                         mutex_init(&instance->chip_mtx, NULL,
772                             MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
649                             mutex_init(&instance->chip_mtx,
650                                 "chip_mtx", MUTEX_DRIVER,
651                                 DDI_INTR_PRI(instance->intr_pri));

774                 }

776                 instance->unroll.mutexs = 1;

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

780                 /* Register our soft-isr for highlevel interrupts. */
781                 instance->isr_level = instance->intr_pri;
782                 if (!(instance->tbolt)) {
783                         if (instance->isr_level == HIGH_LEVEL_INTR) {
784                                 if (ddi_add_softintr(dip,
785                                     DDI_SOFTINT_HIGH,
786                                     &instance->soft_intr_id, NULL, NULL,
787                                     mrsas_softintr, (caddr_t)instance) !=
665                                             &instance->soft_intr_id,
666                                             NULL, NULL, mrsas_softintr,
667                                             (caddr_t)instance) !=
788                                     DDI_SUCCESS) {
789                                         cmn_err(CE_WARN,
790                                             "Software ISR did not register");
670                                                 "Software ISR "
671                                                 "did not register");

792                                         goto fail_attach;
793                                 }

795                                 instance->unroll.soft_isr = 1;

797                         }
798                 }

800                 instance->softint_running = 0;

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



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

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

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

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


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

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

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


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

865                         goto fail_attach;
866                 }
```

```
868                 instance->unroll.devctl = 1;

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

877                         goto fail_attach;
878                 }

880                 instance->unroll.scsictl = 1;

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

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

895                         goto fail_attach;
896                 }

898                 instance->unroll.ioctl = 1;

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

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

914                 /* initiate AEN */
915                 if (start_mfi_aen(instance)) {
916                         cmn_err(CE_WARN,
917                             "mr_sas: failed to initiate AEN.");
918                         goto fail_attach;
919                 }
920                 instance->unroll.aenPend = 1;
921                 con_log(CL_ANN1,
922                     (CE_CONT, "AEN started for instance %d.", instance_no));
806                         con_log(CL_ANN1, (CE_CONT,
807                             "AEN started for instance %d.", instance_no));
```



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

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

937                 instance->mr_ld_list =
938                     kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
939                     KM_SLEEP);
815                         if (instance->mr_ld_list == NULL) {
816                                 cmn_err(CE_WARN,
817                                     "mr_sas attach(): failed to allocate ld_list
818                                 goto fail_attach;
819                         }
940                 instance->unroll.ldlist_buff = 1;

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

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


973         con_log(CL_DLEVEL1,
974             (CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
```

```
 975             instance_no));
 855         cmn_err(CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d", instan
 976         return (DDI_SUCCESS);

 978 fail_attach:

 980         mrsas_undo_resources(dip, instance);

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

 985         mrsas_fm_fini(instance);

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

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

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

 995         return (DDI_FAILURE);
 996 }
_____unchanged_portion_omitted_

1050 /*
1051  * detach - detaches a device from the system
1052  * @dip: pointer to the device's dev_info structure
1053  * @cmd: type of detach
1054  *
1055  * A driver's detach() entry point is called to detach an instance of a device
1056  * that is bound to the driver. The entry point is called with the instance of
1057  * the device node to be detached and with DDI_DETACH, which is specified as
1058  * the cmd argument to the entry point.
1059  * This routine is called during driver unload. We free all the allocated
1060  * resources and call the corresponding LLD so that it can also release all
1061  * its resources.
1062  */
1063 static int
1064 mrsas_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
1065 {
1066         int     instance_no;

1068         struct mrsas_instance   *instance;

1070         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


1073         /* CONSTCOND */
1074         ASSERT(NO_COMPETING_THREADS);

1076         instance_no = ddi_get_instance(dip);

1078         instance = (struct mrsas_instance *)ddi_get_soft_state(mrsas_state,
1079             instance_no);

1081         if (!instance) {
1082                 cmn_err(CE_WARN,
1083                     "mr_sas:%d could not get instance in detach",
1084                     instance_no);

1086                 return (DDI_FAILURE);
1087         }
```



```
1089         con_log(CL_ANN, (CE_NOTE,
1090             "mr_sas%d: detaching device 0x%4x:0x%4x:0x%4x:0x%4x",
1091             instance_no, instance->vendor_id, instance->device_id,
1092             instance->subsysvid, instance->subsysid));

1094         switch (cmd) {
1095                 case DDI_DETACH:
1096                         con_log(CL_ANN, (CE_NOTE,
1097                             "mrsas_detach: DDI_DETACH"));

1099                         mutex_enter(&instance->config_dev_mtx);
1100                         if (instance->timeout_id != (timeout_id_t)-1) {
1101                                 mutex_exit(&instance->config_dev_mtx);
1102                                 (void) untimeout(instance->timeout_id);
1103                                 instance->timeout_id = (timeout_id_t)-1;
1104                                 mutex_enter(&instance->config_dev_mtx);
1105                                 instance->unroll.timer = 0;
1106                         }
1107                         mutex_exit(&instance->config_dev_mtx);

1109                         if (instance->unroll.tranSetup == 1) {
 984                         if(instance->unroll.tranSetup == 1) {
1110                                 if (scsi_hba_detach(dip) != DDI_SUCCESS) {
1111                                         cmn_err(CE_WARN,
1112                                             "mr_sas2%d: failed to detach",
1113                                             instance_no);
 987                                             "mr_sas2%d: failed to detach", i
1114                                         return (DDI_FAILURE);
1115                                 }
1116                                 instance->unroll.tranSetup = 0;
1117                                 con_log(CL_ANN1,
1118                                     (CE_CONT, "scsi_hba_dettach()  done."));
 991                                 con_log(CL_ANN1, (CE_CONT, "scsi_hba_dettach()
1119                         }

1121                         flush_cache(instance);

1123                         mrsas_undo_resources(dip, instance);

1125                         mrsas_fm_fini(instance);

1127                         pci_config_teardown(&instance->pci_handle);
1128                         ddi_soft_state_free(mrsas_state, instance_no);
1129                         break;

1131                 case DDI_PM_SUSPEND:
1132                         con_log(CL_ANN, (CE_NOTE,
1133                             "mrsas_detach: DDI_PM_SUSPEND"));

1135                         break;
1136                 case DDI_SUSPEND:
1137                         con_log(CL_ANN, (CE_NOTE,
1138                             "mrsas_detach: DDI_SUSPEND"));

1140                         break;
1141                 default:
1142                         con_log(CL_ANN, (CE_WARN,
1143                             "invalid detach command:0x%x", cmd));
1144                         return (DDI_FAILURE);
1145         }

1147         return (DDI_SUCCESS);
1148 }
```

```
1151 static void
1152 mrsas_undo_resources(dev_info_t *dip, struct mrsas_instance *instance)
1022 static int
1023 mrsas_undo_resources (dev_info_t *dip, struct mrsas_instance *instance)
1153 {
1154         int     instance_no;

1156         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


1159         instance_no = ddi_get_instance(dip);


1162         if (instance->unroll.ioctl == 1) {
1033         if(instance->unroll.ioctl == 1)  {
1163                 ddi_remove_minor_node(dip, instance->iocnode);
1164                 instance->unroll.ioctl = 0;
1165         }

1167         if (instance->unroll.scsictl == 1) {
1038         if(instance->unroll.scsictl == 1)  {
1168                 ddi_remove_minor_node(dip, "scsi");
1169                 instance->unroll.scsictl = 0;
1170         }

1172         if (instance->unroll.devctl == 1) {
1043         if(instance->unroll.devctl == 1)  {
1173                 ddi_remove_minor_node(dip, "devctl");
1174                 instance->unroll.devctl = 0;
1175         }

1177         if (instance->unroll.tranSetup == 1) {
1048         if(instance->unroll.tranSetup == 1)  {
1178                 if (scsi_hba_detach(dip) != DDI_SUCCESS) {
1179                         cmn_err(CE_WARN,
1180                             "mr_sas2%d: failed to detach", instance_no);
1181                         return;  /* DDI_FAILURE */
1052                         return (DDI_FAILURE);
1182                 }
1183                 instance->unroll.tranSetup = 0;
1184                 con_log(CL_ANN1, (CE_CONT, "scsi_hba_dettach()  done."));
1185         }

1187         if (instance->unroll.tran == 1)  {
1058         if(instance->unroll.tran == 1)  {
1188                 scsi_hba_tran_free(instance->tran);
1189                 instance->unroll.tran = 0;
1190                 con_log(CL_ANN1, (CE_CONT, "scsi_hba_tran_free()  done."));
1191         }

1193         if (instance->unroll.syncCmd == 1) {
1194                 if (instance->tbolt) {
1195                         if (abort_syncmap_cmd(instance,
1196                             instance->map_update_cmd)) {
1064         if(instance->unroll.syncCmd == 1)  {
1065                 if(instance->tbolt) {
1066                         if (abort_syncmap_cmd(instance, instance->map_update_cmd
1197                                 cmn_err(CE_WARN, "mrsas_detach: "
1198                                     "failed to abort previous syncmap command");
1199                         }

1201                         instance->unroll.syncCmd = 0;
1202                         con_log(CL_ANN1, (CE_CONT, "sync cmd aborted, done."));
1203                 }
1204         }
```



```
1206         if (instance->unroll.aenPend == 1) {
1075         if(instance->unroll.aenPend == 1)  {
1207                 if (abort_aen_cmd(instance, instance->aen_cmd))
1208                         cmn_err(CE_WARN, "mrsas_detach: "
1209                             "failed to abort prevous AEN command");

1211                 instance->unroll.aenPend = 0;
1212                 con_log(CL_ANN1, (CE_CONT, "aen cmd aborted, done."));
1213                 /* This means the controller is fully initialized and running */
1214                 /* Shutdown should be a last command to controller. */
1215                 /* shutdown_controller(); */
1082                  /*This means the controller is fully initialzed and running */
1083                  // shutdown_controller();Shutdown should be a last command to c
1216         }


1219         if (instance->unroll.timer == 1)         {
1087         if(instance->unroll.timer == 1)  {
1220                 if (instance->timeout_id != (timeout_id_t)-1) {
1221                         (void) untimeout(instance->timeout_id);
1222                         instance->timeout_id = (timeout_id_t)-1;

1224                         instance->unroll.timer = 0;
1225                 }
1226         }

1228         instance->func_ptr->disable_intr(instance);


1231         if (instance->unroll.mutexs == 1) {
1099         if(instance->unroll.mutexs == 1)  {
1232                 mutex_destroy(&instance->cmd_pool_mtx);
1233                 mutex_destroy(&instance->app_cmd_pool_mtx);
1234                 mutex_destroy(&instance->cmd_pend_mtx);
1235                 mutex_destroy(&instance->completed_pool_mtx);
1236                 mutex_destroy(&instance->sync_map_mtx);
1237                 mutex_destroy(&instance->int_cmd_mtx);
1238                 cv_destroy(&instance->int_cmd_cv);
1239                 mutex_destroy(&instance->config_dev_mtx);
1240                 mutex_destroy(&instance->ocr_flags_mtx);
1241                 mutex_destroy(&instance->reg_write_mtx);

1243                 if (instance->tbolt) {
1244                         mutex_destroy(&instance->cmd_app_pool_mtx);
1245                         mutex_destroy(&instance->chip_mtx);
1246                 }

1248                 instance->unroll.mutexs = 0;
1249                 con_log(CL_ANN1, (CE_CONT, "Destroy mutex & cv,  done."));
1250         }


1253         if (instance->unroll.soft_isr == 1) {
1254                 ddi_remove_softintr(instance->soft_intr_id);
1255                 instance->unroll.soft_isr = 0;
1256         }

1258         if (instance->unroll.intr == 1) {
1126         if(instance->unroll.intr == 1)  {
1259                 mrsas_rem_intrs(instance);
1260                 instance->unroll.intr = 0;
1261         }


1264         if (instance->unroll.taskq == 1)         {
1132         if(instance->unroll.taskq == 1)  {
```

```
1265                 if (instance->taskq) {
1266                         ddi_taskq_destroy(instance->taskq);
1267                         instance->unroll.taskq = 0;
1268                 }

1270         }

1272         /*
1273          * free dma memory allocated for
1274          * cmds/frames/queues/driver version etc
1275          */
1276         if (instance->unroll.verBuff == 1) {
1277                 (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
1140         /*free dma memory allocated for
1141           cmds/frames/queues/driver version etc */
1142         if(instance->unroll.verBuff == 1) {
1143                 mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
1278                 instance->unroll.verBuff = 0;
1279         }

1281         if (instance->unroll.pdlist_buff == 1)  {
1282                 if (instance->mr_tbolt_pd_list != NULL) {
1147         if(instance->unroll.pdlist_buff == 1)  {
1148                 if (instance->mr_tbolt_pd_list != NULL)
1283                         kmem_free(instance->mr_tbolt_pd_list,
1284                             MRSAS_TBOLT_GET_PD_MAX(instance) *
1285                             sizeof (struct mrsas_tbolt_pd));
1286                 }
1150                             MRSAS_TBOLT_GET_PD_MAX(instance) * sizeof (struc

1288                 instance->mr_tbolt_pd_list = NULL;
1289                 instance->unroll.pdlist_buff = 0;
1290         }

1292         if (instance->unroll.ldlist_buff == 1)  {
1293                 if (instance->mr_ld_list != NULL) {
1156         if(instance->unroll.ldlist_buff == 1)  {
1157                 if (instance->mr_ld_list != NULL)
1294                         kmem_free(instance->mr_ld_list, MRDRV_MAX_LD
1295                             * sizeof (struct mrsas_ld));
1296                 }

1298                 instance->mr_ld_list = NULL;
1299                 instance->unroll.ldlist_buff = 0;
1300         }

1302         if (instance->tbolt) {
1303                 if (instance->unroll.alloc_space_mpi2 == 1) {
1166                 if(instance->unroll.alloc_space_mpi2 == 1) {
1304                         free_space_for_mpi2(instance);
1305                         instance->unroll.alloc_space_mpi2 = 0;
1306                 }
1307         } else {
1308                 if (instance->unroll.alloc_space_mfi == 1) {
1171                 if(instance->unroll.alloc_space_mfi == 1) {
1309                         free_space_for_mfi(instance);
1310                         instance->unroll.alloc_space_mfi = 0;
1311                 }
1312         }

1314         if (instance->unroll.regs == 1)  {
1177         if(instance->unroll.regs == 1)  {
1315                 ddi_regs_map_free(&instance->regmap_handle);
1316                 instance->unroll.regs = 0;
1317                 con_log(CL_ANN1, (CE_CONT, "ddi_regs_map_free()  done."));
1318         }
```



```
1183         return (DDI_SUCCESS);
1319 }
_____unchanged_portion_omitted_

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

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

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

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

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

1436         ioctl = (struct mrsas_ioctl *)kmem_zalloc(sizeof (struct mrsas_ioctl),
1437             KM_SLEEP);
1438         ASSERT(ioctl);
1303         if (ioctl == NULL) {
1304                 /* Failed to allocate memory for ioctl  */
1305                 con_log(CL_ANN, (CE_WARN, "mr_sas_ioctl: failed to allocate memo
1306                 return (ENXIO);
1307         }

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

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

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

1463                        break;
1464                case MRSAS_IOCTL_AEN:
1334                        con_log(CL_ANN, (CE_NOTE,
1335                                    "mrsas_ioctl: IOCTL Register AEN.\n"));

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

1473                        rval = handle_mfi_aen(instance, &aen);

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

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

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

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

1495 /*
1496  * ************************************************************************** *
1497  *                                                                            *
1498  *               common entry points - for block driver types                 *
1499  *                                                                            *
1500  * ************************************************************************** *
1501  */
1502 #ifdef __sparc
1503 /*
1504  * reset - TBD
1505  * @dip:
1506  * @cmd:
1507  *
1508  * TBD
1509  */
1510 /*ARGSUSED*/
1511 static int
1512 mrsas_reset(dev_info_t *dip, ddi_reset_cmd_t cmd)
1513 {
1514        int     instance_no;

1516        struct mrsas_instance   *instance;

1518        instance_no = ddi_get_instance(dip);
```



```
1519        instance = (struct mrsas_instance *)ddi_get_soft_state
1520            (mrsas_state, instance_no);

1522        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1524        if (!instance) {
1525                con_log(CL_ANN, (CE_WARN, "mr_sas:%d could not get adapter "
1526                    "in reset", instance_no));
1527                return (DDI_FAILURE);
1528        }

1530        instance->func_ptr->disable_intr(instance);

1532        con_log(CL_ANN1, (CE_CONT, "flushing cache for instance %d",
1533            instance_no));

1535        flush_cache(instance);

1537        return (DDI_SUCCESS);
1538 }
1539 #else /* __sparc */


1540 /*ARGSUSED*/
1541 static int
1413 int
1542 mrsas_quiesce(dev_info_t *dip)
1543 {
1544        int     instance_no;

1546        struct mrsas_instance   *instance;

1548        instance_no = ddi_get_instance(dip);
1549        instance = (struct mrsas_instance *)ddi_get_soft_state
1550            (mrsas_state, instance_no);

1552        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1554        if (!instance) {
1555                con_log(CL_ANN1, (CE_WARN, "mr_sas:%d could not get adapter "
1556                    "in quiesce", instance_no));
1557                return (DDI_FAILURE);
1558        }
1559        if (instance->deadadapter || instance->adapterresetinprogress) {
1560                con_log(CL_ANN1, (CE_WARN, "mr_sas:%d adapter is not in "
1561                    "healthy state", instance_no));
1562                return (DDI_FAILURE);
1563        }

1565        if (abort_aen_cmd(instance, instance->aen_cmd)) {
1566                con_log(CL_ANN1, (CE_WARN, "mrsas_quiesce: "
1567                    "failed to abort prevous AEN command QUIESCE"));
1568        }

1570        if (instance->tbolt) {
1571                if (abort_syncmap_cmd(instance,
1572                    instance->map_update_cmd)) {
1573                        cmn_err(CE_WARN,
1574                            "mrsas_detach: failed to abort "
1575                            "previous syncmap command");
1576                        return (DDI_FAILURE);
1577                }
1578        }

1580        instance->func_ptr->disable_intr(instance);
```

```
1582         con_log(CL_ANN1, (CE_CONT, "flushing cache for instance %d",
1583             instance_no));

1585         flush_cache(instance);

1587         if (wait_for_outstanding(instance)) {
1588                 con_log(CL_ANN1,
1589                     (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
1460                 con_log(CL_ANN1, (CE_CONT, "wait_for_outstanding: return FAIL.\n
1590                 return (DDI_FAILURE);
1591         }
1592         return (DDI_SUCCESS);
1593 }
1594 #endif  /* __sparc */

1596 /*
1597  * ************************************************************************ *
1598  *                                                                          *
1599  *                         entry points (SCSI HBA)                          *
1600  *                                                                          *
1601  * ************************************************************************ *
1602  */
1603 /*
1604  * tran_tgt_init - initialize a target device instance
1605  * @hba_dip:
1606  * @tgt_dip:
1607  * @tran:
1608  * @sd:
1609  *
1610  * The tran_tgt_init() entry point enables the HBA to allocate and initialize
1611  * any per-target resources. tran_tgt_init() also enables the HBA to qualify
1612  * the device's address as valid and supportable for that particular HBA.
1613  * By returning DDI_FAILURE, the instance of the target driver for that device
1614  * is not probed or attached.
1615  */
1616 /*ARGSUSED*/
1617 static int
1618 mrsas_tran_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1619                 scsi_hba_tran_t *tran, struct scsi_device *sd)
1620 {
1621         struct mrsas_instance *instance;
1622         uint16_t tgt = sd->sd_address.a_target;
1623         uint8_t lun = sd->sd_address.a_lun;
1624         dev_info_t *child = NULL;

1626         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init target %d lun %d",
1627             tgt, lun));

1629         instance = ADDR2MR(&sd->sd_address);

1631         if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
1632                 /*
1633                  * If no persistent node exists, we don't allow .conf node
1634                  * to be created.
1635                  */
1636                 if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
1637                         con_log(CL_DLEVEL2,
1638                             (CE_NOTE, "mrsas_tgt_init find child ="
1507                         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init find child
1639                             " %p t = %d l = %d", (void *)child, tgt, lun));
1640                         if (ndi_merge_node(tgt_dip, mrsas_name_node) !=
1641                             DDI_SUCCESS)
1642                                 /* Create this .conf node */
1643                                 return (DDI_SUCCESS);
1644                 }
1645                 con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init in ndi_per "
```



```
1646                     "DDI_FAILURE t = %d l = %d", tgt, lun));
1647                 return (DDI_FAILURE);

1649         }

1651         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init dev_dip %p tgt_dip %p",
1652             (void *)instance->mr_ld_list[tgt].dip, (void *)tgt_dip));

1654         if (tgt < MRDRV_MAX_LD && lun == 0) {
1655                 if (instance->mr_ld_list[tgt].dip == NULL &&
1656                     strcmp(ddi_driver_name(sd->sd_dev), "sd") == 0) {
1657                         mutex_enter(&instance->config_dev_mtx);
1658                         instance->mr_ld_list[tgt].dip = tgt_dip;
1659                         instance->mr_ld_list[tgt].lun_type = MRSAS_LD_LUN;
1660                         instance->mr_ld_list[tgt].flag = MRDRV_TGT_VALID;
1661                         mutex_exit(&instance->config_dev_mtx);
1662                 }
1663         }

1665 #ifdef PDSUPPORT
1666         else if (instance->tbolt) {
1535         else if(instance->tbolt) {
1667                 if (instance->mr_tbolt_pd_list[tgt].dip == NULL) {
1668                         mutex_enter(&instance->config_dev_mtx);
1669                         instance->mr_tbolt_pd_list[tgt].dip = tgt_dip;
1670                         instance->mr_tbolt_pd_list[tgt].flag =
1671                             MRDRV_TGT_VALID;
1672                         mutex_exit(&instance->config_dev_mtx);
1673                         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_tgt_init:"
1674                             "t%xl%x", tgt, lun));
1675                 }
1676         }
1677 #endif

1679         return (DDI_SUCCESS);
1680 }

1682 /*ARGSUSED*/
1683 static void
1684 mrsas_tran_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1685     scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1686 {
1687         struct mrsas_instance *instance;
1688         int tgt = sd->sd_address.a_target;
1689         int lun = sd->sd_address.a_lun;

1691         instance = ADDR2MR(&sd->sd_address);

1693         con_log(CL_DLEVEL2, (CE_NOTE, "tgt_free t = %d l = %d", tgt, lun));

1695         if (tgt < MRDRV_MAX_LD && lun == 0) {
1696                 if (instance->mr_ld_list[tgt].dip == tgt_dip) {
1697                         mutex_enter(&instance->config_dev_mtx);
1698                         instance->mr_ld_list[tgt].dip = NULL;
1699                         mutex_exit(&instance->config_dev_mtx);
1700                 }
1701         }

1703 #ifdef PDSUPPORT
1704         else if (instance->tbolt) {
1573         else if(instance->tbolt) {
1705                 mutex_enter(&instance->config_dev_mtx);
1706                 instance->mr_tbolt_pd_list[tgt].dip = NULL;
1707                 mutex_exit(&instance->config_dev_mtx);
1708                 con_log(CL_ANN1, (CE_NOTE, "tgt_free: Setting dip = NULL"
1709                     "for tgt:%x", tgt));
```

```
1710         }
1711 #endif

1713 }
_____unchanged_portion_omitted_

1864 /*
1865  * tran_start - transport a SCSI command to the addressed target
1866  * @ap:
1867  * @pkt:
1868  *
1869  * The tran_start() entry point for a SCSI HBA driver is called to transport a
1870  * SCSI command to the addressed target. The SCSI command is described
1871  * entirely within the scsi_pkt structure, which the target driver allocated
1872  * through the HBA driver's tran_init_pkt() entry point. If the command
1873  * involves a data transfer, DMA resources must also have been allocated for
1874  * the scsi_pkt structure.
1875  *
1876  * Return Values :
1877  *      TRAN_BUSY - request queue is full, no more free scbs
1878  *      TRAN_ACCEPT - pkt has been submitted to the instance
1879  */
1880 static int
1881 mrsas_tran_start(struct scsi_address *ap, register struct scsi_pkt *pkt)
1882 {
1883         uchar_t         cmd_done = 0;

1885         struct mrsas_instance   *instance = ADDR2MR(ap);
1886         struct mrsas_cmd        *cmd;

1888         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1889         if (instance->deadadapter == 1) {
1890                 con_log(CL_ANN1, (CE_WARN,
1891                     "mrsas_tran_start: return TRAN_FATAL_ERROR "
1892                     "for IO, as the HBA doesnt take any more IOs"));
1893                 if (pkt) {
1894                         pkt->pkt_reason         = CMD_DEV_GONE;
1895                         pkt->pkt_statistics     = STAT_DISCON;
1896                 }
1897                 return (TRAN_FATAL_ERROR);
1898         }

1900         if (instance->adapterresetinprogress) {
1901                 con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_start: Reset flag set, "
1902                     "returning mfi_pkt and setting TRAN_BUSY\n"));
1903                 return (TRAN_BUSY);
1904         }

1906         con_log(CL_ANN1, (CE_CONT, "chkpnt:%s:%d:SCSI CDB[0]=0x%x time:%x",
1907             __func__, __LINE__, pkt->pkt_cdbp[0], pkt->pkt_time));

1909         pkt->pkt_reason = CMD_CMPLT;
1910         *pkt->pkt_scbp = STATUS_GOOD; /* clear arq scsi_status */

1912         cmd = build_cmd(instance, ap, pkt, &cmd_done);

1914         /*
1915          * Check if the command is already completed by the mrsas_build_cmd()
1916          * routine. In which case the busy_flag would be clear and scb will be
1917          * NULL and appropriate reason provided in pkt_reason field
1918          */
1919         if (cmd_done) {
1920                 pkt->pkt_reason = CMD_CMPLT;
1921                 pkt->pkt_scbp[0] = STATUS_GOOD;
1922                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1923                     | STATE_SENT_CMD;
```



```
1924                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1925                         (*pkt->pkt_comp)(pkt);
1926                 }

1928                 return (TRAN_ACCEPT);
1929         }

1931         if (cmd == NULL) {
1932                 return (TRAN_BUSY);
1933         }

1935         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1936                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1937                         con_log(CL_ANN, (CE_CONT, "mr_sas:Firmware busy"));
1938                         DTRACE_PROBE2(start_tran_err,
1939                             uint16_t, instance->fw_outstanding,
1940                             uint16_t, instance->max_fw_cmds);
1806                         cmn_err(CE_WARN, "mr_sas:Firmware BUSY, fw_outstanding(0
1807                                         instance->fw_outstanding, instance->max_
1941                         return_mfi_pkt(instance, cmd);
1942                         return (TRAN_BUSY);
1943                 }

1945                 /* Synchronize the Cmd frame for the controller */
1946                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1947                     DDI_DMA_SYNC_FORDEV);
1948                 con_log(CL_ANN, (CE_CONT, "issue_cmd_ppc: SCSI CDB[0]=0x%x"
1949                     "cmd->index:%x\n", pkt->pkt_cdbp[0], cmd->index));
1950                 instance->func_ptr->issue_cmd(cmd, instance);

1952         } else {
1953                 struct mrsas_header *hdr = &cmd->frame->hdr;

1955                 instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);

1823                 instance->func_ptr-> issue_cmd_in_poll_mode(instance, cmd);

1957                 pkt->pkt_reason         = CMD_CMPLT;
1958                 pkt->pkt_statistics     = 0;
1959                 pkt->pkt_state |= STATE_XFERRED_DATA | STATE_GOT_STATUS;

1961                 switch (ddi_get8(cmd->frame_dma_obj.acc_handle,
1962                     &hdr->cmd_status)) {
1963                 case MFI_STAT_OK:
1964                         pkt->pkt_scbp[0] = STATUS_GOOD;
1965                         break;

1967                 case MFI_STAT_SCSI_DONE_WITH_ERROR:
1968                         con_log(CL_ANN, (CE_CONT,
1969                             "mrsas_tran_start: scsi done with error"));
1970                         pkt->pkt_reason = CMD_CMPLT;
1971                         pkt->pkt_statistics = 0;

1973                         ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;
1974                         break;

1976                 case MFI_STAT_DEVICE_NOT_FOUND:
1977                         con_log(CL_ANN, (CE_CONT,
1978                             "mrsas_tran_start: device not found error"));
1979                         pkt->pkt_reason         = CMD_DEV_GONE;
1980                         pkt->pkt_statistics     = STAT_DISCON;
1981                         break;

1983                 default:
1984                         ((struct scsi_status *)pkt->pkt_scbp)->sts_busy = 1;
1985                 }
```

```
                (void) mrsas_common_check(instance, cmd);
                DTRACE_PROBE2(start_nointr_done, uint8_t, hdr->cmd,
                    uint8_t, hdr->cmd_status);
                return_mfi_pkt(instance, cmd);

                if (pkt->pkt_comp) {
                        (*pkt->pkt_comp)(pkt);
                }

        }

        return (TRAN_ACCEPT);
}
```

*unchanged_portion_omitted_*

```
/*
 * tran_reset - reset either the SCSI bus or target
 * @ap:
 * @level:
 *
 * The tran_reset() entry point for a SCSI HBA driver is called to reset either
 * the SCSI bus or a particular SCSI target device. This entry point is called
 * when a target driver calls scsi_reset(). The tran_reset() entry point must
 * reset the SCSI bus if level is RESET_ALL. If level is RESET_TARGET, just the
 * particular target or logical unit must be reset.
 */
/*ARGSUSED*/
static int
mrsas_tran_reset(struct scsi_address *ap, int level)
{
        struct mrsas_instance *instance = ADDR2MR(ap);

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

        if (wait_for_outstanding(instance)) {
                con_log(CL_ANN1,
                    (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
                con_log(CL_ANN1, (CE_CONT, "wait_for_outstanding: return FAIL.\n
                return (DDI_FAILURE);
        } else {
                return (DDI_SUCCESS);
        }
}

#if 0
/*
 * tran_bus_reset - reset the SCSI bus
 * @dip:
 * @level:
 *
 * The tran_bus_reset() vector in the scsi_hba_tran structure should be
 * initialized during the HBA driver's attach(). The vector should point to
 * an HBA entry point that is to be called when a user initiates a bus reset.
 * Implementation is hardware specific. If the HBA driver cannot reset the
 * SCSI bus without affecting the targets, the driver should fail RESET_BUS
 * or not initialize this vector.
 */
/*ARGSUSED*/
static int
mrsas_tran_bus_reset(dev_info_t *dip, int level)
{
        int     instance_no = ddi_get_instance(dip);

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



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

        if (wait_for_outstanding(instance)) {
                con_log(CL_ANN1,
                    (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
                con_log(CL_ANN1, (CE_CONT, "wait_for_outstanding: return FAIL.\n
                return (DDI_FAILURE);
        } else {
                return (DDI_SUCCESS);
        }
}
#endif

/*
 * tran_getcap - get one of a set of SCSA-defined capabilities
 * @ap:
 * @cap:
 * @whom:
 *
 * The target driver can request the current setting of the capability for a
 * particular target by setting the whom parameter to nonzero. A whom value of
 * zero indicates a request for the current setting of the general capability
 * for the SCSI bus or for adapter hardware. The tran_getcap() should return -1
 * for undefined capabilities or the current value of the requested capability.
 */
/*ARGSUSED*/
static int
mrsas_tran_getcap(struct scsi_address *ap, char *cap, int whom)
{
        int     rval = 0;

        struct mrsas_instance   *instance = ADDR2MR(ap);

        con_log(CL_DLEVEL2, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

        /* we do allow inquiring about capabilities for other targets */
        if (cap == NULL) {
                return (-1);
        }

        switch (scsi_hba_lookup_capstr(cap)) {
        case SCSI_CAP_DMA_MAX:
                if (instance->tbolt) {
                        /* Limit to 256k max transfer */
                        rval = mrsas_tbolt_max_cap_maxxfer;
                } else {
                        /* Limit to 16MB max transfer */
                        rval = mrsas_max_cap_maxxfer;
                }
                break;
        case SCSI_CAP_MSG_OUT:
                rval = 1;
                break;
        case SCSI_CAP_DISCONNECT:
                rval = 0;
                break;
        case SCSI_CAP_SYNCHRONOUS:
                rval = 0;
                break;
        case SCSI_CAP_WIDE_XFER:
                rval = 1;
                break;
        case SCSI_CAP_TAGGED_QING:
                rval = 1;
                break;
```

```
2139         case SCSI_CAP_UNTAGGED_QING:
2140                 rval = 1;
2141                 break;
2142         case SCSI_CAP_PARITY:
2143                 rval = 1;
2144                 break;
2145         case SCSI_CAP_INITIATOR_ID:
2146                 rval = instance->init_id;
2147                 break;
2148         case SCSI_CAP_ARQ:
2149                 rval = 1;
2150                 break;
2151         case SCSI_CAP_LINKED_CMDS:
2152                 rval = 0;
2153                 break;
2154         case SCSI_CAP_RESET_NOTIFICATION:
2155                 rval = 1;
2156                 break;
2157         case SCSI_CAP_GEOMETRY:
2158                 rval = -1;

2160                 break;
2161         default:
2162                 con_log(CL_DLEVEL2, (CE_NOTE, "Default cap coming 0x%x",
2163                     scsi_hba_lookup_capstr(cap)));
2164                 rval = -1;
2165                 break;
2166         }

2168         return (rval);
2169 }
______unchanged_portion_omitted_


2347 /*
2348  * mrsas_isr(caddr_t)
2349  *
2350  * The Interrupt Service Routine
2351  *
2352  * Collect status for all completed commands and do callback
2353  *
2354  */
2355 static uint_t
2356 mrsas_isr(struct mrsas_instance *instance)
2357 {
2358         int             need_softintr;
2359         uint32_t        producer;
2360         uint32_t        consumer;
2361         uint32_t        context;
2223         uint32_t        status, value;
2362         int             retval;

2364         struct mrsas_cmd        *cmd;
2365         struct mrsas_header     *hdr;
2366         struct scsi_pkt         *pkt;

2368         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
2369         ASSERT(instance);
2370         if (instance->tbolt) {
2371                 mutex_enter(&instance->chip_mtx);
2372                 if ((instance->intr_type == DDI_INTR_TYPE_FIXED) &&
2373                     !(instance->func_ptr->intr_ack(instance))) {
2374                         mutex_exit(&instance->chip_mtx);
2375                         return (DDI_INTR_UNCLAIMED);
2376                 }
2377                 retval = mr_sas_tbolt_process_outstanding_cmd(instance);
```



```
2378                 mutex_exit(&instance->chip_mtx);
2379                 return (retval);
2380         } else {
2381                 if ((instance->intr_type == DDI_INTR_TYPE_FIXED) &&
2382                     !instance->func_ptr->intr_ack(instance)) {
2383                         return (DDI_INTR_UNCLAIMED);
2384                 }
2385         }

2387         (void) ddi_dma_sync(instance->mfi_internal_dma_obj.dma_handle,
2388             0, 0, DDI_DMA_SYNC_FORCPU);

2390         if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
2391             != DDI_SUCCESS) {
2392                 mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2393                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2394                 con_log(CL_ANN1, (CE_WARN,
2395                     "mr_sas_isr(): FMA check, returning DDI_INTR_UNCLAIMED"));
2396                 return (DDI_INTR_CLAIMED);
2397         }
2398         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2400 #ifdef OCRDEBUG
2401         if (debug_consecutive_timeout_after_ocr_g == 1) {
2402                 con_log(CL_ANN1, (CE_NOTE,
2403                     "simulating consecutive timeout after ocr"));
2404                 return (DDI_INTR_CLAIMED);
2405         }
2406 #endif

2408         mutex_enter(&instance->completed_pool_mtx);
2409         mutex_enter(&instance->cmd_pend_mtx);

2411         producer = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2412             instance->producer);
2413         consumer = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2414             instance->consumer);

2416         con_log(CL_ANN, (CE_CONT, " producer %x consumer %x ",
2417             producer, consumer));
2418         if (producer == consumer) {
2419                 con_log(CL_ANN, (CE_WARN, "producer ==  consumer case"));
2420                 DTRACE_PROBE2(isr_pc_err, uint32_t, producer,
2421                     uint32_t, consumer);
2422                 mutex_exit(&instance->cmd_pend_mtx);
2423                 mutex_exit(&instance->completed_pool_mtx);
2424                 return (DDI_INTR_CLAIMED);
2425         }

2427         while (consumer != producer) {
2428                 context = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2429                     &instance->reply_queue[consumer]);
2430                 cmd = instance->cmd_list[context];

2432                 if (cmd->sync_cmd == MRSAS_TRUE) {
2433                         hdr = (struct mrsas_header *)&cmd->frame->hdr;
2434                         if (hdr) {
2435                                 mlist_del_init(&cmd->list);
2436                         }
2437                 } else {
2438                         pkt = cmd->pkt;
2439                         if (pkt) {
2440                                 mlist_del_init(&cmd->list);
2441                         }
2442                 }
```

```
2444                 mlist_add_tail(&cmd->list, &instance->completed_pool_list);

2446                 consumer++;
2447                 if (consumer == (instance->max_fw_cmds + 1)) {
2448                         consumer = 0;
2449                 }
2450         }
2451         ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
2452             instance->consumer, consumer);
2453         mutex_exit(&instance->cmd_pend_mtx);
2454         mutex_exit(&instance->completed_pool_mtx);

2456         (void) ddi_dma_sync(instance->mfi_internal_dma_obj.dma_handle,
2457             0, 0, DDI_DMA_SYNC_FORDEV);

2459         if (instance->softint_running) {
2460                 need_softintr = 0;
2461         } else {
2462                 need_softintr = 1;
2463         }

2465         if (instance->isr_level == HIGH_LEVEL_INTR) {
2466                 if (need_softintr) {
2467                         ddi_trigger_softintr(instance->soft_intr_id);
2468                 }
2469         } else {
2470                 /*
2471                  * Not a high-level interrupt, therefore call the soft level
2472                  * interrupt explicitly
2473                  */
2474                 (void) mrsas_softintr(instance);
2475         }

2477         return (DDI_INTR_CLAIMED);
2478 }


2481 /*
2482  * ************************************************************************** *
2483  *                                                                            *
2484  *                                   libraries                                *
2485  *                                                                            *
2486  * ************************************************************************** *
2487  */
2488 /*
2489  * get_mfi_pkt : Get a command from the free pool
2490  * After successful allocation, the caller of this routine
2491  * must clear the frame buffer (memset to zero) before
2492  * using the packet further.
2493  *
2494  * ***** Note *****
2495  * After clearing the frame buffer the context id of the
2496  * frame buffer SHOULD be restored back.
2497  */
2498 static struct mrsas_cmd *
2499 get_mfi_pkt(struct mrsas_instance *instance)
2500 {
2501         mlist_t                 *head = &instance->cmd_pool_list;
2502         struct mrsas_cmd        *cmd = NULL;

2504         mutex_enter(&instance->cmd_pool_mtx);
2355         ASSERT(mutex_owned(&instance->cmd_pool_mtx));

2506         if (!mlist_empty(head)) {
2507                 cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2508                 mlist_del_init(head->next);
```



```
2509         }
2510         if (cmd != NULL) {
2511                 cmd->pkt = NULL;
2512                 cmd->retry_count_for_ocr = 0;
2513                 cmd->drv_pkt_time = 0;

2515         }
2516         mutex_exit(&instance->cmd_pool_mtx);

2518         return (cmd);
2519 }

2521 static struct mrsas_cmd *
2522 get_mfi_app_pkt(struct mrsas_instance *instance)
2523 {
2524         mlist_t                 *head = &instance->app_cmd_pool_list;
2525         struct mrsas_cmd        *cmd = NULL;

2527         mutex_enter(&instance->app_cmd_pool_mtx);
2379         ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2529         if (!mlist_empty(head)) {
2530                 cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2531                 mlist_del_init(head->next);
2532         }
2533         if (cmd != NULL) {
2385         if (cmd != NULL){
2534                 cmd->pkt = NULL;
2535                 cmd->retry_count_for_ocr = 0;
2536                 cmd->drv_pkt_time = 0;
2537         }

2539         mutex_exit(&instance->app_cmd_pool_mtx);

2541         return (cmd);
2542 }
2543 /*
2544  * return_mfi_pkt : Return a cmd to free command pool
2545  */
2546 static void
2547 return_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2548 {
2549         mutex_enter(&instance->cmd_pool_mtx);
2550         /* use mlist_add_tail for debug assistance */
2402         ASSERT(mutex_owned(&instance->cmd_pool_mtx));

2551         mlist_add_tail(&cmd->list, &instance->cmd_pool_list);

2553         mutex_exit(&instance->cmd_pool_mtx);
2554 }

2556 static void
2557 return_mfi_app_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2558 {
2559         mutex_enter(&instance->app_cmd_pool_mtx);
2413         ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2561         mlist_add(&cmd->list, &instance->app_cmd_pool_list);

2563         mutex_exit(&instance->app_cmd_pool_mtx);
2564 }
2565 void
2566 push_pending_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2567 {
2568         struct scsi_pkt *pkt;
2569         struct mrsas_header     *hdr;
```

```
2570         con_log(CL_DLEVEL2, (CE_NOTE, "push_pending_pkt(): Called\n"));
2571         mutex_enter(&instance->cmd_pend_mtx);
2426         ASSERT(mutex_owned(&instance->cmd_pend_mtx));
2572         mlist_del_init(&cmd->list);
2573         mlist_add_tail(&cmd->list, &instance->cmd_pend_list);
2574         if (cmd->sync_cmd == MRSAS_TRUE) {
2575                 hdr = (struct mrsas_header *)&cmd->frame->hdr;
2576                 if (hdr) {
2577                         con_log(CL_ANN1, (CE_CONT,
2578                             "push_pending_mfi_pkt: "
2579                             "cmd %p index %x "
2580                             "time %llx",
2581                             (void *)cmd, cmd->index,
2582                             gethrtime()));
2583                         /* Wait for specified interval  */
2584                         cmd->drv_pkt_time = ddi_get16(
2585                             cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2586                         if (cmd->drv_pkt_time < debug_timeout_g)
2587                                 cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2588                                 con_log(CL_ANN1, (CE_CONT,
2589                                     "push_pending_pkt(): "
2590                                     "Called IO Timeout Value %x\n",
2591                                     cmd->drv_pkt_time));
2592                 }
2593                 if (hdr && instance->timeout_id == (timeout_id_t)-1) {
2594                         instance->timeout_id = timeout(io_timeout_checker,
2595                             (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2596                 }
2597         } else {
2598                 pkt = cmd->pkt;
2599                 if (pkt) {
2600                         con_log(CL_ANN1, (CE_CONT,
2601                             "push_pending_mfi_pkt: "
2602                             "cmd %p index %x pkt %p, "
2603                             "time %llx",
2604                             (void *)cmd, cmd->index, (void *)pkt,
2605                             gethrtime()));
2606                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2607                 }
2608                 if (pkt && instance->timeout_id == (timeout_id_t)-1) {
2609                         instance->timeout_id = timeout(io_timeout_checker,
2610                             (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2611                 }
2612         }

2614         mutex_exit(&instance->cmd_pend_mtx);

2616 }

2618 int
2619 mrsas_print_pending_cmds(struct mrsas_instance *instance)
2620 {
2621         mlist_t *head = &instance->cmd_pend_list;
2622         mlist_t *tmp = head;
2623         struct mrsas_cmd *cmd = NULL;
2624         struct mrsas_header     *hdr;
2625         unsigned int            flag = 1;
2626         struct scsi_pkt *pkt;
2627         int saved_level;
2628         int cmd_count = 0;


2630         saved_level = debug_level_g;
2631         debug_level_g = CL_ANN1;

2633         cmn_err(CE_NOTE, "mrsas_print_pending_cmds(): Called\n");
```



```
2635         while (flag) {
2636                 mutex_enter(&instance->cmd_pend_mtx);
2637                 tmp     =       tmp->next;
2638                 if (tmp == head) {
2639                         mutex_exit(&instance->cmd_pend_mtx);
2640                         flag = 0;
2641                         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds():"
2642                             " NO MORE CMDS PENDING....\n"));
2497                         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds():
2643                         break;
2644                 } else {
2645                         cmd = mlist_entry(tmp, struct mrsas_cmd, list);
2646                         mutex_exit(&instance->cmd_pend_mtx);
2647                         if (cmd) {
2648                                 if (cmd->sync_cmd == MRSAS_TRUE) {
2649                                         hdr = (struct mrsas_header *)
2650                                             &cmd->frame->hdr;
2504                                         hdr = (struct mrsas_header *)&cmd->frame->hdr;
2651                                         if (hdr) {
2652                                                 con_log(CL_ANN1, (CE_CONT,
2653                                                     "print: cmd %p index 0x%x "
2654                                                     "drv_pkt_time 0x%x (NO-PKT)"
2655                                                     " hdr %p\n", (void *)cmd,
2656                                                     cmd->index,
2657                                                     cmd->drv_pkt_time,
2658                                                     (void *)hdr));
2507                                                 "print: cmd %p index 0x%x drv_pkt_ti
2508                                                 (void *)cmd, cmd->index, cmd->drv_pk
2659                                         }
2660                                 } else {
2661                                         pkt = cmd->pkt;
2662                                         if (pkt) {
2663                                         con_log(CL_ANN1, (CE_CONT,
2664                                             "print: cmd %p index 0x%x "
2665                                             "drv_pkt_time 0x%x pkt %p \n",
2666                                             (void *)cmd, cmd->index,
2667                                             cmd->drv_pkt_time, (void *)pkt));
2514                                             "print: cmd %p index 0x%x drv_pkt_ti
2515                                             (void *)cmd, cmd->index,  cmd->drv_p
2668                                         }
2669                                 }

2671                                 if (++cmd_count == 1) {
2672                                         mrsas_print_cmd_details(instance, cmd,
2673                                             0xDD);
2674                                 } else {
2675                                         mrsas_print_cmd_details(instance, cmd,
2676                                             1);
2677                                 }
2519                                 if (++cmd_count == 1)
2520                                         mrsas_print_cmd_details(instance, cmd, 0
2521                                 else
2522                                         mrsas_print_cmd_details(instance, cmd, 1

2679                         }
2680                 }
2681         }
2682         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds(): Done\n"));


2685         debug_level_g = saved_level;

2687         return (DDI_SUCCESS);
2688 }
```

_____***unchanged_portion_omitted_***

```
2755 void
2756 mrsas_print_cmd_details(struct mrsas_instance *instance, struct mrsas_cmd *cmd,
2757     int detail)
2601 mrsas_print_cmd_details(struct mrsas_instance *instance,
2602                                     struct mrsas_cmd *cmd, int detail )
2758 {
2759         struct scsi_pkt *pkt = cmd->pkt;
2760         Mpi2RaidSCSIIORequest_t *scsi_io = cmd->scsi_io_request;
2761         int i;
2606         MPI2_SCSI_IO_VENDOR_UNIQUE *raidContext;
2607         uint8_t *cdb_p;
2608         char str[100], *strp;
2609         int i, j, len;
2762         int saved_level;
2763         ddi_acc_handle_t acc_handle =
2764             instance->mpi2_frame_pool_dma_obj.acc_handle;


2766         if (detail == 0xDD) {
2767                 saved_level = debug_level_g;
2768                 debug_level_g = CL_ANN1;
2769         }


2772         if (instance->tbolt) {
2773                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p "
2774                     "cmd->index 0x%x SMID 0x%x timer 0x%x sec\n",
2620                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p cmd->index
2775                     (void *)cmd, cmd->index, cmd->SMID, cmd->drv_pkt_time));
2776         } else {
2777                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p "
2778                     "cmd->index 0x%x timer 0x%x sec\n",
2622         }
2623         else {
2624                 con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p cmd->index
2779                     (void *)cmd, cmd->index, cmd->drv_pkt_time));
2780         }

2782         if (pkt) {
2628         if(pkt) {
2783                 con_log(CL_ANN1, (CE_CONT, "scsi_pkt CDB[0]=0x%x",
2784                     pkt->pkt_cdbp[0]));
2785         } else {
2631         }else {
2786                 con_log(CL_ANN1, (CE_CONT, "NO-PKT"));
2787         }

2789         if ((detail == 0xDD) && instance->tbolt) {
2635         if((detail==0xDD) && instance->tbolt) {
2790                 con_log(CL_ANN1, (CE_CONT, "RAID_SCSI_IO_REQUEST\n"));
2791                 con_log(CL_ANN1, (CE_CONT, "DevHandle=0x%X Function=0x%X "
2792                     "IoFlags=0x%X SGLFlags=0x%X DataLength=0x%X\n",
2793                     ddi_get16(acc_handle, &scsi_io->DevHandle),
2794                     ddi_get8(acc_handle, &scsi_io->Function),
2795                     ddi_get16(acc_handle, &scsi_io->IoFlags),
2796                     ddi_get16(acc_handle, &scsi_io->SGLFlags),
2797                     ddi_get32(acc_handle, &scsi_io->DataLength)));
2637                 con_log(CL_ANN1, (CE_CONT, "DevHandle=0x%X Function=0x%X IoFlags
2638                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2639                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2640                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2641                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2642                         ddi_get32(instance->mpi2_frame_pool_dma_obj.acc_handle,

2799                 for (i = 0; i < 32; i++) {
```



```
2800                         con_log(CL_ANN1, (CE_CONT, "CDB[%d]=0x%x ", i,
2801                             ddi_get8(acc_handle, &scsi_io->CDB.CDB32[i])));
2802                 }
2644                 for(i=0; i < 32; i++)
2645                         con_log(CL_ANN1, (CE_CONT, "CDB[%d]=0x%x ",i,
2646                                 ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_h

2804                 con_log(CL_ANN1, (CE_CONT, "RAID-CONTEXT\n"));
2805                 con_log(CL_ANN1, (CE_CONT, "status=0x%X extStatus=0x%X "
2806                     "ldTargetId=0x%X timeoutValue=0x%X regLockFlags=0x%X "
2807                     "RAIDFlags=0x%X regLockRowLBA=0x%" PRIu64
2808                     " regLockLength=0x%X spanArm=0x%X\n",
2809                     ddi_get8(acc_handle, &scsi_io->RaidContext.status),
2810                     ddi_get8(acc_handle, &scsi_io->RaidContext.extStatus),
2811                     ddi_get16(acc_handle, &scsi_io->RaidContext.ldTargetId),
2812                     ddi_get16(acc_handle, &scsi_io->RaidContext.timeoutValue),
2813                     ddi_get8(acc_handle, &scsi_io->RaidContext.regLockFlags),
2814                     ddi_get8(acc_handle, &scsi_io->RaidContext.RAIDFlags),
2815                     ddi_get64(acc_handle, &scsi_io->RaidContext.regLockRowLBA),
2816                     ddi_get32(acc_handle, &scsi_io->RaidContext.regLockLength),
2817                     ddi_get8(acc_handle, &scsi_io->RaidContext.spanArm)));
2649                 con_log(CL_ANN1, (CE_CONT, "status=0x%X extStatus=0x%X ldTargetI
2650                         "regLockFlags=0x%X RAIDFlags=0x%X regLockRowLBA=0x%lX re
2651                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2652                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2653                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2654                         ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2655                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2656                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &
2657                         ddi_get64(instance->mpi2_frame_pool_dma_obj.acc_handle,
2658                         ddi_get32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2659                         ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle, &


2818         }

2820         if (detail == 0xDD) {
2821                 debug_level_g = saved_level;
2822         }

2668         return;
2823 }


2826 int
2827 mrsas_issue_pending_cmds(struct mrsas_instance *instance)
2828 {
2829         mlist_t *head   =       &instance->cmd_pend_list;
2830         mlist_t *tmp    =       head->next;
2831         struct mrsas_cmd *cmd = NULL;
2832         struct scsi_pkt *pkt;

2834         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_pending_cmds(): Called"));
2835         while (tmp != head) {
2836                 mutex_enter(&instance->cmd_pend_mtx);
2837                 cmd = mlist_entry(tmp, struct mrsas_cmd, list);
2838                 tmp = tmp->next;
2839                 mutex_exit(&instance->cmd_pend_mtx);
2840                 if (cmd) {
2841                         con_log(CL_ANN1, (CE_CONT,
2842                             "mrsas_issue_pending_cmds(): "
2843                             "Got a cmd: cmd %p index 0x%x drv_pkt_time 0x%x ",
2844                             (void *)cmd, cmd->index, cmd->drv_pkt_time));

2846                         /* Reset command timeout value */
2847                         if (cmd->drv_pkt_time < debug_timeout_g)
```

```
2848                                    cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2693                                    cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

2850                            cmd->retry_count_for_ocr++;

2852                            cmn_err(CE_CONT, "cmd retry count = %d\n",
2853                                cmd->retry_count_for_ocr);

2855                            if (cmd->retry_count_for_ocr > IO_RETRY_COUNT) {
2856                                    cmn_err(CE_WARN, "mrsas_issue_pending_cmds(): "
2857                                        "cmd->retry_count exceeded limit >%d\n",
2701                                    cmn_err(CE_WARN,
2702                                            "mrsas_issue_pending_cmds(): cmd->retry_
2858                                        IO_RETRY_COUNT);
2859                                    mrsas_print_cmd_details(instance, cmd, 0xDD);

2861                                    cmn_err(CE_WARN,
2862                                        "mrsas_issue_pending_cmds():"
2863                                        "Calling KILL Adapter\n");
2864                                    if (instance->tbolt)
2865                                            mrsas_tbolt_kill_adapter(instance);
2710                                            (void) mrsas_tbolt_kill_adapter(instance
2866                                    else
2867                                            (void) mrsas_kill_adapter(instance);
2868                                    return (DDI_FAILURE);
2869                            }

2871                            pkt = cmd->pkt;
2872                            if (pkt) {
2873                                    con_log(CL_ANN1, (CE_CONT,
2874                                        "PENDING PKT-CMD ISSUE: cmd %p index %x "
2875                                        "pkt %p time %llx",
2876                                        (void *)cmd, cmd->index,
2877                                        (void *)pkt,
2878                                        gethrtime()));

2880                            } else {
2881                                    cmn_err(CE_CONT,
2882                                        "mrsas_issue_pending_cmds(): NO-PKT, "
2883                                        "cmd %p index 0x%x drv_pkt_time 0x%x ",
2727                                            "mrsas_issue_pending_cmds(): "
2728                                            "NO-PKT, cmd %p index 0x%x drv_pkt_time
2884                                        (void *)cmd, cmd->index, cmd->drv_pkt_time);
2885                            }


2888                            if (cmd->sync_cmd == MRSAS_TRUE) {
2889                                    cmn_err(CE_CONT, "mrsas_issue_pending_cmds(): "
2890                                        "SYNC_CMD == TRUE \n");
2891                                    instance->func_ptr->issue_cmd_in_sync_mode(
2892                                        instance, cmd);
2734                                    cmn_err(CE_CONT, "mrsas_issue_pending_cmds(): SY

2736                                    instance->func_ptr->issue_cmd_in_sync_mode(insta
2893                            } else {
2894                                    instance->func_ptr->issue_cmd(cmd, instance);
2895                            }
2896                    } else {
2897                            con_log(CL_ANN1, (CE_CONT,
2898                                "mrsas_issue_pending_cmds: NULL command\n"));
2899                    }
2900                    con_log(CL_ANN1, (CE_CONT,
2901                        "mrsas_issue_pending_cmds:"
2902                        "looping for more commands"));
2903            }
2904            con_log(CL_ANN1, (CE_CONT, "mrsas_issue_pending_cmds(): DONE\n"));
```



```
2905            return (DDI_SUCCESS);
2906    }
_______unchanged_portion_omitted_


3113    void
3114    mrsas_free_cmd_pool(struct mrsas_instance *instance)
3115    {
3116            int             i;
3117            uint32_t        max_cmd;
3118            size_t          sz;

3120            /* already freed */
3121            if (instance->cmd_list == NULL) {
3122                    return;
3123            }

3125            max_cmd = instance->max_fw_cmds;

3127            /* size of cmd_list array */
3128            sz = sizeof (struct mrsas_cmd *) * max_cmd;

3130            /* First free each cmd */
3131            for (i = 0; i < max_cmd; i++) {
3132                    if (instance->cmd_list[i] != NULL) {
3133                            kmem_free(instance->cmd_list[i],
3134                                sizeof (struct mrsas_cmd));
3135                    }
2976                    if (instance->cmd_list[i] != NULL)
2977                            kmem_free(instance->cmd_list[i],sizeof (struct mrsas_cmd

3137                    instance->cmd_list[i] = NULL;
3138            }

3140            /* Now, free cmd_list array */
3141            if (instance->cmd_list != NULL)
3142                    kmem_free(instance->cmd_list, sz);
2984                    kmem_free(instance->cmd_list,sz);

3144            instance->cmd_list = NULL;

3146            INIT_LIST_HEAD(&instance->cmd_pool_list);
3147            INIT_LIST_HEAD(&instance->cmd_pend_list);
3148            if (instance->tbolt) {
3149                    INIT_LIST_HEAD(&instance->cmd_app_pool_list);
3150            } else {
2992            }
2993            else {
3151                    INIT_LIST_HEAD(&instance->app_cmd_pool_list);
3152            }

3154    }


3157    /*
3158     * mrsas_alloc_cmd_pool
3159     */
3160    int
3161    mrsas_alloc_cmd_pool(struct mrsas_instance *instance)
3162    {
3163            int             i;
3164            int             count;
3165            uint32_t        max_cmd;
3166            uint32_t        reserve_cmd;
3167            size_t          sz;
```

```
3169         struct mrsas_cmd        *cmd;

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


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

3178         /*
3179          * instance->cmd_list is an array of struct mrsas_cmd pointers.
3180          * Allocate the dynamic array first and then allocate individual
3181          * commands.
3182          */
3183         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
3184         ASSERT(instance->cmd_list);
3027         if (instance->cmd_list == NULL) {
3028                 con_log(CL_NONE, (CE_WARN,
3029                     "Failed to allocate memory for cmd_list"));
3030                 return (DDI_FAILURE);
3031         }

3186         /* create a frame pool and assign one frame to each cmd */
3187         for (count = 0; count < max_cmd; count++) {
3188                 instance->cmd_list[count] =
3189                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
3190                 ASSERT(instance->cmd_list[count]);
3035                 instance->cmd_list[count] = kmem_zalloc(sizeof (struct mrsas_cmd
3036                     KM_SLEEP);
3037                 if (instance->cmd_list[count] == NULL) {
3038                         con_log(CL_NONE, (CE_WARN,
3039                                 "Failed to allocate memory for mrsas_cmd"));
3040                         goto mrsas_undo_cmds;
3191         }
3042         }

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

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

3199         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

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


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

3214         return (DDI_SUCCESS);

3216 mrsas_undo_cmds:
3217         if (count > 0) {
3218                 /* free each cmd */
3219                 for (i = 0; i < count; i++) {
3220                         if (instance->cmd_list[i] != NULL) {
3221                                 kmem_free(instance->cmd_list[i],
3222                                     sizeof (struct mrsas_cmd));
```



```
3223                         }
3071                         if (instance->cmd_list[i] != NULL)
3072                                 kmem_free(instance->cmd_list[i],sizeof (struct m
3224                         instance->cmd_list[i] = NULL;
3225                 }
3226         }

3228 mrsas_undo_cmd_list:
3229         if (instance->cmd_list != NULL)
3230                 kmem_free(instance->cmd_list, sz);
3079                 kmem_free(instance->cmd_list,sz);
3231         instance->cmd_list = NULL;

3233         return (DDI_FAILURE);
3234 }
______unchanged_portion_omitted_

3260 /*
3261  * alloc_space_for_mfi
3262  */
3263 static int
3264 alloc_space_for_mfi(struct mrsas_instance *instance)
3265 {
3266         /* Allocate command pool (memory for cmd_list & individual commands) */
3115         /* Allocate command pool ( memory for cmd_list & individual commands )*/
3267         if (mrsas_alloc_cmd_pool(instance)) {
3268                 cmn_err(CE_WARN, "error creating cmd pool");
3269                 return (DDI_FAILURE);
3270         }

3272         /* Allocate MFI Frame pool */
3273         if (create_mfi_frame_pool(instance)) {
3274                 cmn_err(CE_WARN, "error creating frame DMA pool");
3275                 goto mfi_undo_cmd_pool;
3276         }

3278         /* Allocate additional DMA buffer */
3279         if (alloc_additional_dma_buffer(instance)) {
3280                 cmn_err(CE_WARN, "error creating frame DMA pool");
3281                 goto mfi_undo_frame_pool;
3282         }

3284         return (DDI_SUCCESS);

3286 mfi_undo_frame_pool:
3287         destroy_mfi_frame_pool(instance);

3289 mfi_undo_cmd_pool:
3290         mrsas_free_cmd_pool(instance);

3292         return (DDI_FAILURE);
3293 }



3297 /*
3298  * get_ctrl_info
3299  */
3300 static int
3301 get_ctrl_info(struct mrsas_instance *instance,
3302     struct mrsas_ctrl_info *ctrl_info)
3303 {
3304         int     ret = 0;

3306         struct mrsas_cmd                *cmd;
3307         struct mrsas_dcmd_frame *dcmd;
```

```
3308         struct mrsas_ctrl_info  *ci;

3310         if (instance->tbolt) {
3159         if(instance->tbolt) {
3311                 cmd = get_raid_msg_mfi_pkt(instance);
3312         } else {
3161         }
3162         else {
3313                 cmd = get_mfi_pkt(instance);
3314         }

3316         if (!cmd) {
3317                 con_log(CL_ANN, (CE_WARN,
3318                     "Failed to get a cmd for ctrl info"));
3319                 DTRACE_PROBE2(info_mfi_err, uint16_t, instance->fw_outstanding,
3320                     uint16_t, instance->max_fw_cmds);
3167                 cmn_err(CE_WARN,
3168                     "Failed to get a cmd from free-pool in get_ctrl_info(). fw_o
3169                                 instance->fw_outstanding, instance->max_fw_cmds)
3321                 return (DDI_FAILURE);
3322         }

3324         /* Clear the frame buffer and assign back the context id */
3325         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3326         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3327             cmd->index);

3329         dcmd = &cmd->frame->dcmd;

3331         ci = (struct mrsas_ctrl_info *)instance->internal_buf;

3333         if (!ci) {
3334                 cmn_err(CE_WARN,
3335                     "Failed to alloc mem for ctrl info");
3336                 return_mfi_pkt(instance, cmd);
3337                 return (DDI_FAILURE);
3338         }

3340         (void) memset(ci, 0, sizeof (struct mrsas_ctrl_info));

3342         /* for( i = 0; i < DCMD_MBOX_SZ; i++ ) dcmd->mbox.b[i] = 0; */
3343         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

3345         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3346         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status,
3347             MFI_CMD_STATUS_POLL_MODE);
3348         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3349         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3350             MFI_FRAME_DIR_READ);
3351         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3352         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3353             sizeof (struct mrsas_ctrl_info));
3354         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3355             MR_DCMD_CTRL_GET_INFO);
3356         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3357             instance->internal_buf_dmac_add);
3358         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3359             sizeof (struct mrsas_ctrl_info));

3361         cmd->frame_count = 1;

3363         if (instance->tbolt) {
3364                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3365         }

3367         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
```



```
3368                 ret = 0;

3370                 ctrl_info->max_request_size = ddi_get32(
3371                     cmd->frame_dma_obj.acc_handle, &ci->max_request_size);

3373                 ctrl_info->ld_present_count = ddi_get16(
3374                     cmd->frame_dma_obj.acc_handle, &ci->ld_present_count);

3376                 ctrl_info->properties.on_off_properties = ddi_get32(
3377                     cmd->frame_dma_obj.acc_handle,
3225         ctrl_info->properties.on_off_properties =
3226             ddi_get32(cmd->frame_dma_obj.acc_handle,
3378                     &ci->properties.on_off_properties);
3379                 ddi_rep_get8(cmd->frame_dma_obj.acc_handle,
3380                     (uint8_t *)(ctrl_info->product_name),
3381                     (uint8_t *)(ci->product_name), 80 * sizeof (char),
3382                     DDI_DEV_AUTOINCR);
3383                 /* should get more members of ci with ddi_get when needed */
3384         } else {
3385                 cmn_err(CE_WARN, "get_ctrl_info: Ctrl info failed");
3386                 ret = -1;
3387         }

3389         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3390                 ret = -1;
3391         }
3392         if (instance->tbolt) {
3238         if(instance->tbolt) {
3393                 return_raid_msg_mfi_pkt(instance, cmd);
3394         } else {
3240         }
3241         else {
3395                 return_mfi_pkt(instance, cmd);
3396         }

3398         return (ret);
3399 }

3401 /*
3402  * abort_aen_cmd
3403  */
3404 static int
3405 abort_aen_cmd(struct mrsas_instance *instance,
3406     struct mrsas_cmd *cmd_to_abort)
3407 {
3408         int     ret = 0;

3410         struct mrsas_cmd                *cmd;
3411         struct mrsas_abort_frame        *abort_fr;

3413         con_log(CL_ANN1, (CE_NOTE, "chkpnt: abort_aen:%d", __LINE__));

3415         if (instance->tbolt) {
3416                 cmd = get_raid_msg_mfi_pkt(instance);
3417         } else {
3418                 cmd = get_mfi_pkt(instance);
3419         }

3421         if (!cmd) {
3422                 con_log(CL_ANN1, (CE_WARN,
3423                     "abort_aen_cmd():Failed to get a cmd for abort_aen_cmd"));
3424                 DTRACE_PROBE2(abort_mfi_err, uint16_t, instance->fw_outstanding,
3425                     uint16_t, instance->max_fw_cmds);
3269                 cmn_err(CE_WARN,
3270                     "Failed to get a cmd from free-pool in abort_aen_cmd(). fw_o
3271                                 instance->fw_outstanding, instance->max_fw_cmds)
```

```
3426                 return (DDI_FAILURE);
3427         }

3429         /* Clear the frame buffer and assign back the context id */
3430         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3431         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3432             cmd->index);

3434         abort_fr = &cmd->frame->abort;

3436         /* prepare and issue the abort frame */
3437         ddi_put8(cmd->frame_dma_obj.acc_handle,
3438             &abort_fr->cmd, MFI_CMD_OP_ABORT);
3439         ddi_put8(cmd->frame_dma_obj.acc_handle, &abort_fr->cmd_status,
3440             MFI_CMD_STATUS_SYNC_MODE);
3441         ddi_put16(cmd->frame_dma_obj.acc_handle, &abort_fr->flags, 0);
3442         ddi_put32(cmd->frame_dma_obj.acc_handle, &abort_fr->abort_context,
3443             cmd_to_abort->index);
3444         ddi_put32(cmd->frame_dma_obj.acc_handle,
3445             &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3446         ddi_put32(cmd->frame_dma_obj.acc_handle,
3447             &abort_fr->abort_mfi_phys_addr_hi, 0);

3449         instance->aen_cmd->abort_aen = 1;

3451         cmd->frame_count = 1;

3453         if (instance->tbolt) {
3454                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3455         }

3457         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3458                 con_log(CL_ANN1, (CE_WARN,
3459                     "abort_aen_cmd: issue_cmd_in_poll_mode failed"));
3460                 ret = -1;
3461         } else {
3462                 ret = 0;
3463         }

3465         instance->aen_cmd->abort_aen = 1;
3466         instance->aen_cmd = 0;

3468         if (instance->tbolt) {
3469                 return_raid_msg_mfi_pkt(instance, cmd);
3470         } else {
3471                 return_mfi_pkt(instance, cmd);
3472         }

3474         atomic_add_16(&instance->fw_outstanding, (-1));

3476         return (ret);
3477 }


3480 static int
3481 mrsas_build_init_cmd(struct mrsas_instance *instance,
3482     struct mrsas_cmd **cmd_ptr)
3326 mrsas_build_init_cmd(struct mrsas_instance *instance, struct mrsas_cmd **cmd_ptr
3483 {
3484         struct mrsas_cmd                *cmd;
3485         struct mrsas_init_frame         *init_frame;
3486         struct mrsas_init_queue_info    *initq_info;
3487         struct mrsas_drv_ver            drv_ver_info;


3490         /*
```



```
3491          * Prepare a init frame. Note the init frame points to queue info
3492          * structure. Each frame has SGL allocated after first 64 bytes. For
3493          * this frame - since we don't need any SGL - we use SGL's space as
3494          * queue info structure
3495          */
3496         cmd = *cmd_ptr;


3499         /* Clear the frame buffer and assign back the context id */
3500         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3501         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3502             cmd->index);

3504         init_frame = (struct mrsas_init_frame *)cmd->frame;
3505         initq_info = (struct mrsas_init_queue_info *)
3506             ((unsigned long)init_frame + 64);

3508         (void) memset(init_frame, 0, MRMFI_FRAME_SIZE);
3509         (void) memset(initq_info, 0, sizeof (struct mrsas_init_queue_info));

3511         ddi_put32(cmd->frame_dma_obj.acc_handle, &initq_info->init_flags, 0);

3513         ddi_put32(cmd->frame_dma_obj.acc_handle,
3514             &initq_info->reply_queue_entries, instance->max_fw_cmds + 1);

3516         ddi_put32(cmd->frame_dma_obj.acc_handle,
3517             &initq_info->producer_index_phys_addr_hi, 0);
3518         ddi_put32(cmd->frame_dma_obj.acc_handle,
3519             &initq_info->producer_index_phys_addr_lo,
3520             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address);

3522         ddi_put32(cmd->frame_dma_obj.acc_handle,
3523             &initq_info->consumer_index_phys_addr_hi, 0);
3524         ddi_put32(cmd->frame_dma_obj.acc_handle,
3525             &initq_info->consumer_index_phys_addr_lo,
3526             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 4);

3528         ddi_put32(cmd->frame_dma_obj.acc_handle,
3529             &initq_info->reply_queue_start_phys_addr_hi, 0);
3530         ddi_put32(cmd->frame_dma_obj.acc_handle,
3531             &initq_info->reply_queue_start_phys_addr_lo,
3532             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 8);

3534         ddi_put8(cmd->frame_dma_obj.acc_handle,
3535             &init_frame->cmd, MFI_CMD_OP_INIT);
3536         ddi_put8(cmd->frame_dma_obj.acc_handle, &init_frame->cmd_status,
3537             MFI_CMD_STATUS_POLL_MODE);
3538         ddi_put16(cmd->frame_dma_obj.acc_handle, &init_frame->flags, 0);
3539         ddi_put32(cmd->frame_dma_obj.acc_handle,
3540             &init_frame->queue_info_new_phys_addr_lo,
3541             cmd->frame_phys_addr + 64);
3542         ddi_put32(cmd->frame_dma_obj.acc_handle,
3543             &init_frame->queue_info_new_phys_addr_hi, 0);


3546         /* fill driver version information */
3390         /* fill driver version information*/
3547         fill_up_drv_ver(&drv_ver_info);

3549         /* allocate the driver version data transfer buffer */
3550         instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
3394         instance->drv_ver_dma_obj.size = sizeof(drv_ver_info.drv_ver);
3551         instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
3552         instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
3553         instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
3554         instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
```

```
3555         instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;

3557         if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
3558             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
3559                 con_log(CL_ANN, (CE_WARN,
3560                     "init_mfi : Could not allocate driver version buffer."));
3561                 return (DDI_FAILURE);
3562         }
3563         /* copy driver version to dma buffer */
3564         (void) memset(instance->drv_ver_dma_obj.buffer, 0,
3565             sizeof (drv_ver_info.drv_ver));
3407         /* copy driver version to dma  buffer*/
3408         (void) memset(instance->drv_ver_dma_obj.buffer, 0,sizeof(drv_ver_info.dr
3566         ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
3567             (uint8_t *)drv_ver_info.drv_ver,
3568             (uint8_t *)instance->drv_ver_dma_obj.buffer,
3569             sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);
3412             sizeof(drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);


3572         /* copy driver version physical address to init frame */
3573         ddi_put64(cmd->frame_dma_obj.acc_handle, &init_frame->driverversion,
3574             instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);
3415         /*copy driver version physical address to init frame*/
3416         ddi_put64(cmd->frame_dma_obj.acc_handle,
3417             &init_frame->driverversion, instance->drv_ver_dma_obj.dma_cookie[0].

3576         ddi_put32(cmd->frame_dma_obj.acc_handle, &init_frame->data_xfer_len,
3577             sizeof (struct mrsas_init_queue_info));

3579         cmd->frame_count = 1;

3581         *cmd_ptr = cmd;

3583         return (DDI_SUCCESS);
3584 }


3587 /*
3588  * mrsas_init_adapter_ppc - Initialize MFI interface adapter.
3589  */
3590 int
3591 mrsas_init_adapter_ppc(struct mrsas_instance *instance)
3434 mrsas_init_adapter_ppc (struct mrsas_instance *instance)
3592 {
3593         struct mrsas_cmd                *cmd;

3595         /*
3596          * allocate memory for mfi adapter(cmd pool, individual commands, mfi
3597          * frames etc
3598          */
3599         if (alloc_space_for_mfi(instance) != DDI_SUCCESS) {
3438         /* allocate memory for mfi adapter(cmd pool, individual commands, mfi fr
3439         if (alloc_space_for_mfi(instance) != DDI_SUCCESS){
3600                 con_log(CL_ANN, (CE_NOTE,
3601                     "Error, failed to allocate memory for MFI adapter"));
3602                 return (DDI_FAILURE);
3603         }

3605         /* Build INIT command */
3606         cmd = get_mfi_pkt(instance);

3608         if (mrsas_build_init_cmd(instance, &cmd) != DDI_SUCCESS) {
3609                 con_log(CL_ANN,
3610                     (CE_NOTE, "Error, failed to build INIT command"));
3448         if (mrsas_build_init_cmd(instance, &cmd) != DDI_SUCCESS){
```



```
3449                 con_log(CL_ANN, (CE_NOTE,
3450                         "Error, failed to build INIT command"));

3612                 goto fail_undo_alloc_mfi_space;
3613         }

3615         /*
3616          * Disable interrupt before sending init frame ( see linux driver code)
3617          * send INIT MFI frame in polled mode
3618          */
3455         //Disalbe interrupt before sending init frame ( see linux driver code)
3456         /* send INIT MFI frame in polled mode */
3619         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3620                 con_log(CL_ANN, (CE_WARN, "failed to init firmware"));
3621                 goto fail_fw_init;
3622         }

3624         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
3625                 goto fail_fw_init;
3626         return_mfi_pkt(instance, cmd);

3628         if (ctio_enable &&
3629             (instance->func_ptr->read_fw_status_reg(instance) & 0x04000000)) {
3462         if (instance->func_ptr->read_fw_status_reg(instance) & 0x04000000) {
3630                 con_log(CL_ANN, (CE_NOTE, "mr_sas: IEEE SGL's supported"));
3631                 instance->flag_ieee = 1;
3632         } else {
3633                 instance->flag_ieee = 0;
3634         }

3636         instance->unroll.alloc_space_mfi = 1;
3637         instance->unroll.verBuff = 1;

3639         return (DDI_SUCCESS);


3642 fail_fw_init:
3643         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
3474         mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

3645 fail_undo_alloc_mfi_space:
3646         return_mfi_pkt(instance, cmd);
3647         free_space_for_mfi(instance);

3649         return (DDI_FAILURE);

3651 }

3653 /*
3654  * mrsas_init_adapter - Initialize adapter.
3655  */
3656 int
3657 mrsas_init_adapter(struct mrsas_instance *instance)
3488 mrsas_init_adapter (struct mrsas_instance *instance)
3658 {
3659         struct mrsas_ctrl_info          ctrl_info;


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

3668         /* get various operational parameters from status register */
3669         instance->max_num_sge =
```

```
3670             (instance->func_ptr->read_fw_status_reg(instance) &
3671             0xFF0000) >> 0x10;
3672         instance->max_num_sge =
3673             (instance->max_num_sge > MRSAS_MAX_SGE_CNT) ?
3674             MRSAS_MAX_SGE_CNT : instance->max_num_sge;

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



3687         /* Initialize adapter */
3688         if (instance->func_ptr->init_adapter(instance) != DDI_SUCCESS) {
3689                 con_log(CL_ANN,
3690                     (CE_WARN, "mr_sas: could not initialize adapter"));
3520                 con_log(CL_ANN, (CE_WARN, "mr_sas: "
3521                         "could not initialize adapter"));
3691                 return (DDI_FAILURE);
3692         }

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

3697         if (!get_ctrl_info(instance, &ctrl_info)) {
3698                 instance->max_sectors_per_req = ctrl_info.max_request_size;
3699                 con_log(CL_ANN1, (CE_NOTE,
3700                     "product name %s ld present %d",
3701                     ctrl_info.product_name, ctrl_info.ld_present_count));
3702         } else {
3703                 instance->max_sectors_per_req = instance->max_num_sge *
3704                     PAGESIZE / 512;
3705         }

3707         if (ctrl_info.properties.on_off_properties & DISABLE_OCR_PROP_FLAG)
3538         if (ctrl_info.properties.on_off_properties & DISABLE_OCR_PROP_FLAG) {
3708                 instance->disable_online_ctrl_reset = 1;
3540                 con_log(CL_ANN1, (CE_NOTE,
3541                         "Disable online control Flag is set\n"));
3542         }
3543         else {
3544                 con_log(CL_ANN1, (CE_NOTE,
3545                         "Disable online control Flag is not set\n"));
3546         }

3710         return (DDI_SUCCESS);

3712 }



3716 static int
3717 mrsas_issue_init_mfi(struct mrsas_instance *instance)
3718 {
3719         struct mrsas_cmd                *cmd;
3720         struct mrsas_init_frame         *init_frame;
3721         struct mrsas_init_queue_info    *initq_info;

3723 /*
3724  * Prepare a init frame. Note the init frame points to queue info
3725  * structure. Each frame has SGL allocated after first 64 bytes. For
```



```
3726  * this frame - since we don't need any SGL - we use SGL's space as
3727  * queue info structure
3728  */
3729         con_log(CL_ANN1, (CE_NOTE,
3730             "mrsas_issue_init_mfi: entry\n"));
3731         cmd = get_mfi_app_pkt(instance);

3733         if (!cmd) {
3734                 con_log(CL_ANN1, (CE_WARN,
3735                     "mrsas_issue_init_mfi: get_pkt failed\n"));
3736                 return (DDI_FAILURE);
3737         }

3739         /* Clear the frame buffer and assign back the context id */
3740         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3741         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3742             cmd->index);

3744         init_frame = (struct mrsas_init_frame *)cmd->frame;
3745         initq_info = (struct mrsas_init_queue_info *)
3746             ((unsigned long)init_frame + 64);

3748         (void) memset(init_frame, 0, MRMFI_FRAME_SIZE);
3749         (void) memset(initq_info, 0, sizeof (struct mrsas_init_queue_info));

3751         ddi_put32(cmd->frame_dma_obj.acc_handle, &initq_info->init_flags, 0);

3753         ddi_put32(cmd->frame_dma_obj.acc_handle,
3754             &initq_info->reply_queue_entries, instance->max_fw_cmds + 1);
3755         ddi_put32(cmd->frame_dma_obj.acc_handle,
3756             &initq_info->producer_index_phys_addr_hi, 0);
3757         ddi_put32(cmd->frame_dma_obj.acc_handle,
3758             &initq_info->producer_index_phys_addr_lo,
3759             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address);
3760         ddi_put32(cmd->frame_dma_obj.acc_handle,
3761             &initq_info->consumer_index_phys_addr_hi, 0);
3762         ddi_put32(cmd->frame_dma_obj.acc_handle,
3763             &initq_info->consumer_index_phys_addr_lo,
3764             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 4);

3766         ddi_put32(cmd->frame_dma_obj.acc_handle,
3767             &initq_info->reply_queue_start_phys_addr_hi, 0);
3768         ddi_put32(cmd->frame_dma_obj.acc_handle,
3769             &initq_info->reply_queue_start_phys_addr_lo,
3770             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 8);

3772         ddi_put8(cmd->frame_dma_obj.acc_handle,
3773             &init_frame->cmd, MFI_CMD_OP_INIT);
3774         ddi_put8(cmd->frame_dma_obj.acc_handle, &init_frame->cmd_status,
3775             MFI_CMD_STATUS_POLL_MODE);
3776         ddi_put16(cmd->frame_dma_obj.acc_handle, &init_frame->flags, 0);
3777         ddi_put32(cmd->frame_dma_obj.acc_handle,
3778             &init_frame->queue_info_new_phys_addr_lo,
3779             cmd->frame_phys_addr + 64);
3780         ddi_put32(cmd->frame_dma_obj.acc_handle,
3781             &init_frame->queue_info_new_phys_addr_hi, 0);

3783         ddi_put32(cmd->frame_dma_obj.acc_handle, &init_frame->data_xfer_len,
3784             sizeof (struct mrsas_init_queue_info));

3786         cmd->frame_count = 1;

3788         /* issue the init frame in polled mode */
3789         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3790                 con_log(CL_ANN1, (CE_WARN,
3791                     "mrsas_issue_init_mfi():failed to "
```

```
3792                    "init firmware"));
3793                return_mfi_app_pkt(instance, cmd);
3794                return (DDI_FAILURE);
3795        }

3797        if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3798                return_mfi_pkt(instance, cmd);
3799                return (DDI_FAILURE);
3800        }

3802        return_mfi_app_pkt(instance, cmd);
3803        con_log(CL_ANN1, (CE_CONT, "mrsas_issue_init_mfi: Done"));

3805        return (DDI_SUCCESS);
3806 }
3807 /*
3808  * mfi_state_transition_to_ready        : Move the FW to READY state
3809  *
3810  * @reg_set                    : MFI register set
3811  */
3812 int
3813 mfi_state_transition_to_ready(struct mrsas_instance *instance)
3814 {
3815        int             i;
3816        uint8_t         max_wait;
3817        uint32_t        fw_ctrl = 0;
3818        uint32_t        fw_state;
3819        uint32_t        cur_state;
3820        uint32_t        cur_abs_reg_val;
3821        uint32_t        prev_abs_reg_val;
3822        uint32_t        status;

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

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

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

3840                        return (ENODEV);
3841                case MFI_STATE_WAIT_HANDSHAKE:
3842                        /* set the CLR bit in IMR0 */
3843                        con_log(CL_ANN1, (CE_NOTE,
3844                            "mr_sas: FW waiting for HANDSHAKE"));
3845                        /*
3846                         * PCI_Hot Plug: MFI F/W requires
3847                         * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3848                         * to be set
3849                         */
3850                        /* WR_IB_MSG_0(MFI_INIT_CLEAR_HANDSHAKE, instance); */
3851                        if (!instance->tbolt) {
3852                                WR_IB_DOORBELL(MFI_INIT_CLEAR_HANDSHAKE |
3853                                    MFI_INIT_HOTPLUG, instance);
3854                        } else {
3855                                WR_RESERVED0_REGISTER(MFI_INIT_CLEAR_HANDSHAKE |
3856                                    MFI_INIT_HOTPLUG, instance);
3857                        }
```



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

3896                                for (i = 0; i < (10 * 1000); i++) {
3897                                        status =
3898                                            RD_RESERVED0_REGISTER(instance);
3899                                        if (status & 1) {
3732                                        if (status & 1)
3900                                                delay(1 *
3901                                                    drv_usectohz(MILLISEC));
3902                                        } else {
3735                                        else
3903                                                break;
3904                                        }
3905                                }

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

3915                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3916                        cur_state       = MFI_STATE_UNDEFINED;
3917                        break;
3918                case MFI_STATE_BB_INIT:
3919                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
3920                        cur_state       = MFI_STATE_BB_INIT;
3921                        break;
```

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

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

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

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

3974         if (!instance->tbolt) {
3975                 fw_ctrl = RD_IB_DOORBELL(instance);
3976                 con_log(CL_ANN1, (CE_CONT,
3977                     "mfi_state_transition_to_ready:FW ctrl = 0x%x", fw_ctrl));
3810         }

3812 #if 0
3979                 /*
3980                  * Write 0xF to the doorbell register to do the following.
3981                  * - Abort all outstanding commands (bit 0).
3982                  * - Transition from OPERATIONAL to READY state (bit 1).
3983                  * - Discard (possible) low MFA posted in 64-bit mode (bit-2).
3984                  * - Set to release FW to continue running (i.e. BIOS handshake
3985                  *   (bit 3).
```



```
3986                  */
3821         if (!instance->tbolt) {
3987                 WR_IB_DOORBELL(0xF, instance);
3988         }
3824 #endif

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

3994         return (DDI_SUCCESS);
3995 }

3997 /*
3998  * get_seq_num
3999  */
4000 static int
4001 get_seq_num(struct mrsas_instance *instance,
4002     struct mrsas_evt_log_info *eli)
4003 {
4004         int     ret = DDI_SUCCESS;

4006         dma_obj_t                       dcmd_dma_obj;
4007         struct mrsas_cmd                *cmd;
4008         struct mrsas_dcmd_frame         *dcmd;
4009         struct mrsas_evt_log_info *eli_tmp;
4010         if (instance->tbolt) {
4011                 cmd = get_raid_msg_mfi_pkt(instance);
4012         } else {
4013                 cmd = get_mfi_pkt(instance);
4014         }

4016         if (!cmd) {
4017                 cmn_err(CE_WARN, "mr_sas: failed to get a cmd");
4018                 DTRACE_PROBE2(seq_num_mfi_err, uint16_t,
4019                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
3849                 cmn_err(CE_WARN,
3850                     "Failed to get a cmd from free-pool in get_seq_num(). fw_out
3851                         instance->fw_outstanding, instance->max_fw_cmds);
4020                 return (ENOMEM);
4021         }

4023         /* Clear the frame buffer and assign back the context id */
4024         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4025         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4026             cmd->index);

4028         dcmd = &cmd->frame->dcmd;

4030         /* allocate the data transfer buffer */
4031         dcmd_dma_obj.size = sizeof (struct mrsas_evt_log_info);
4032         dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
4033         dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
4034         dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
4035         dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
4036         dcmd_dma_obj.dma_attr.dma_attr_align = 1;

4038         if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
4039             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
4040                 cmn_err(CE_WARN,
4041                     "get_seq_num: could not allocate data transfer buffer.");
4042                 return (DDI_FAILURE);
4043         }

4045         (void) memset(dcmd_dma_obj.buffer, 0,
4046             sizeof (struct mrsas_evt_log_info));
```

```
4048         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4050         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
4051         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
4052         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
4053         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4054             MFI_FRAME_DIR_READ);
4055         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4056         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
4057             sizeof (struct mrsas_evt_log_info));
4058         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
4059             MR_DCMD_CTRL_EVENT_GET_INFO);
4060         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
4061             sizeof (struct mrsas_evt_log_info));
4062         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
4063             dcmd_dma_obj.dma_cookie[0].dmac_address);

4065         cmd->sync_cmd = MRSAS_TRUE;
4066         cmd->frame_count = 1;

4068         if (instance->tbolt) {
4069                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4070         }

4072         if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
4073                 cmn_err(CE_WARN, "get_seq_num: "
4074                     "failed to issue MRSAS_DCMD_CTRL_EVENT_GET_INFO");
4075                 ret = DDI_FAILURE;
4076         } else {
4077                 eli_tmp = (struct mrsas_evt_log_info *)dcmd_dma_obj.buffer;
4078                 eli->newest_seq_num = ddi_get32(cmd->frame_dma_obj.acc_handle,
4079                     &eli_tmp->newest_seq_num);
4080                 ret = DDI_SUCCESS;
4081         }

4083         if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
4084                 ret = DDI_FAILURE;

4086         if (instance->tbolt) {
4087                 return_raid_msg_mfi_pkt(instance, cmd);
4088         } else {
4089                 return_mfi_pkt(instance, cmd);
4090         }

4092         return (ret);
4093 }
```

_____***unchanged_portion_omitted_***

```
4131 /*
4132  * flush_cache
4133  */
4134 static void
4135 flush_cache(struct mrsas_instance *instance)
4136 {
4137         struct mrsas_cmd                *cmd = NULL;
4138         struct mrsas_dcmd_frame         *dcmd;
3971         uint32_t        max_cmd = instance->max_fw_cmds;
4139         if (instance->tbolt) {
4140                 cmd = get_raid_msg_mfi_pkt(instance);
4141         } else {
4142                 cmd = get_mfi_pkt(instance);
4143         }

4145         if (!cmd) {
4146                 con_log(CL_ANN1, (CE_WARN,
```



```
4147                     "flush_cache():Failed to get a cmd for flush_cache"));
4148                 DTRACE_PROBE2(flush_cache_err, uint16_t,
4149                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
3979                 cmn_err(CE_WARN,
3980                     "Failed to get a cmd from free-pool in flush_cache(). fw_out
3981                         instance->fw_outstanding, instance->max_fw_cmds)
4150                 return;
4151         }

4153         /* Clear the frame buffer and assign back the context id */
4154         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4155         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4156             cmd->index);

4158         dcmd = &cmd->frame->dcmd;

4160         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4162         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
4163         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
4164         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 0);
4165         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4166             MFI_FRAME_DIR_NONE);
4167         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4168         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len, 0);
4169         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
4170             MR_DCMD_CTRL_CACHE_FLUSH);
4171         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.b[0],
4172             MR_FLUSH_CTRL_CACHE | MR_FLUSH_DISK_CACHE);

4174         cmd->frame_count = 1;

4176         if (instance->tbolt) {
4177                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4178         }

4180         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
4181                 con_log(CL_ANN1, (CE_WARN,
4182             "flush_cache: failed to issue MFI_DCMD_CTRL_CACHE_FLUSH"));
4183         }
4184         con_log(CL_ANN1, (CE_CONT, "flush_cache done"));
4185         if (instance->tbolt) {
4186                 return_raid_msg_mfi_pkt(instance, cmd);
4187         } else {
4188                 return_mfi_pkt(instance, cmd);
4189         }

4191 }

4193 /*
4194  * service_mfi_aen-     Completes an AEN command
4195  * @instance:                  Adapter soft state
4196  * @cmd:                       Command to be completed
4197  *
4198  */
4199 void
4200 service_mfi_aen(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
4201 {
4202         uint32_t        seq_num;
4035         uint32_t        i;
4203         struct mrsas_evt_detail *evt_detail =
4204             (struct mrsas_evt_detail *)instance->mfi_evt_detail_obj.buffer;
4205         int             rval = 0;
4206         int             tgt = 0;
4207         uint8_t         dtype;
4208 #ifdef PDSUPPORT
```

```
4209         mrsas_pd_address_t      *pd_addr;
4210 #endif
4211         ddi_acc_handle_t                acc_handle;

4213         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

4215         acc_handle = cmd->frame_dma_obj.acc_handle;
4216         cmd->cmd_status = ddi_get8(acc_handle, &cmd->frame->io.cmd_status);
4217         if (cmd->cmd_status == ENODATA) {
4218                 cmd->cmd_status = 0;
4219         }

4221         /*
4222          * log the MFI AEN event to the sysevent queue so that
4223          * application will get noticed
4224          */
4225         if (ddi_log_sysevent(instance->dip, DDI_VENDOR_LSI, "LSIMEGA", "SAS",
4226             NULL, NULL, DDI_NOSLEEP) != DDI_SUCCESS) {
4227                 int     instance_no = ddi_get_instance(instance->dip);
4228                 con_log(CL_ANN, (CE_WARN,
4229                     "mr_sas%d: Failed to log AEN event", instance_no));
4230         }
4231         /*
4232          * Check for any ld devices that has changed state. i.e. online
4233          * or offline.
4234          */
4235         con_log(CL_ANN1, (CE_CONT,
4236             "AEN: code = %x class = %x locale = %x args = %x",
4237             ddi_get32(acc_handle, &evt_detail->code),
4238             evt_detail->cl.members.class,
4239             ddi_get16(acc_handle, &evt_detail->cl.members.locale),
4240             ddi_get8(acc_handle, &evt_detail->arg_type)));

4242         switch (ddi_get32(acc_handle, &evt_detail->code)) {
4243         case MR_EVT_CFG_CLEARED: {
4244                 for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
4245                         if (instance->mr_ld_list[tgt].dip != NULL) {
4246                                 mutex_enter(&instance->config_dev_mtx);
4247                                 instance->mr_ld_list[tgt].flag =
4248                                     (uint8_t)~MRDRV_TGT_VALID;
4081                                     ~MRDRV_TGT_VALID;
4249                                 mutex_exit(&instance->config_dev_mtx);
4250                                 rval = mrsas_service_evt(instance, tgt, 0,
4251                                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4252                                 con_log(CL_ANN1, (CE_WARN,
4253                                     "mr_sas: CFG CLEARED AEN rval = %d "
4254                                     "tgt id = %d", rval, tgt));
4255                         }
4256                 }
4257                 break;
4258         }

4260         case MR_EVT_LD_DELETED: {
4261                 tgt = ddi_get16(acc_handle, &evt_detail->args.ld.target_id);
4262                 mutex_enter(&instance->config_dev_mtx);
4263                 instance->mr_ld_list[tgt].flag = (uint8_t)~MRDRV_TGT_VALID;
4096                 instance->mr_ld_list[tgt].flag = ~MRDRV_TGT_VALID;
4264                 mutex_exit(&instance->config_dev_mtx);
4265                 rval = mrsas_service_evt(instance,
4266                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4267                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4268                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD DELETED AEN rval = %d "
4269                     "tgt id = %d index = %d", rval,
4270                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4271                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4272                 break;
```



```
4273         } /* End of MR_EVT_LD_DELETED */

4275         case MR_EVT_LD_CREATED: {
4276                 rval = mrsas_service_evt(instance,
4277                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4278                     MRSAS_EVT_CONFIG_TGT, NULL);
4279                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD CREATED AEN rval = %d "
4280                     "tgt id = %d index = %d", rval,
4281                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4282                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4283                 break;
4284         } /* End of MR_EVT_LD_CREATED */

4286 #ifdef PDSUPPORT
4287         case MR_EVT_PD_REMOVED_EXT: {
4288                 if (instance->tbolt) {
4289                         pd_addr = &evt_detail->args.pd_addr;
4290                         dtype = pd_addr->scsi_dev_type;
4291                         con_log(CL_DLEVEL1, (CE_NOTE,
4292                             " MR_EVT_PD_REMOVED_EXT: dtype = %x,"
4293                             " arg_type = %d ", dtype, evt_detail->arg_type));
4294                         tgt = ddi_get16(acc_handle,
4295                             &evt_detail->args.pd.device_id);
4127                             tgt = ddi_get16(acc_handle, &evt_detail->args.pd.dev
4296                         mutex_enter(&instance->config_dev_mtx);
4297                         instance->mr_tbolt_pd_list[tgt].flag =
4298                             (uint8_t)~MRDRV_TGT_VALID;
4129                         instance->mr_tbolt_pd_list[tgt].flag = (uint8_t)~MRDRV_T
4299                         mutex_exit(&instance->config_dev_mtx);
4300                         rval = mrsas_service_evt(instance, ddi_get16(
4301                             acc_handle, &evt_detail->args.pd.device_id),
4131                         rval = mrsas_service_evt(instance,
4132                         ddi_get16(acc_handle, &evt_detail->args.pd.device_id),
4302                             1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4303                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4304                             "rval = %d tgt id = %d ", rval,
4305                             ddi_get16(acc_handle,
4306                             &evt_detail->args.pd.device_id)));
4307                 }
4136                         ddi_get16(acc_handle, &evt_detail->args.pd.device_id)));
4308                 break;
4309         } /* End of MR_EVT_PD_REMOVED_EXT */
4138                 }
4139         }/* End of MR_EVT_PD_REMOVED_EXT */

4311         case MR_EVT_PD_INSERTED_EXT: {
4312                 if (instance->tbolt) {
4313                         rval = mrsas_service_evt(instance,
4314                             ddi_get16(acc_handle,
4315                             &evt_detail->args.pd.device_id),
4144                         ddi_get16(acc_handle, &evt_detail->args.pd.device_id),
4316                             1, MRSAS_EVT_CONFIG_TGT, NULL);
4317                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_INSERTEDi_EXT:"
4318                             "rval = %d tgt id = %d ", rval,
4319                             ddi_get16(acc_handle,
4320                             &evt_detail->args.pd.device_id)));
4321                 }
4148                         ddi_get16(acc_handle, &evt_detail->args.pd.device_id
4322                 break;
4323         } /* End of MR_EVT_PD_INSERTED_EXT */
4150                 }
4151         }/* End of MR_EVT_PD_INSERTED_EXT */

4325         case MR_EVT_PD_STATE_CHANGE: {
4326                 if (instance->tbolt) {
4327                         tgt = ddi_get16(acc_handle,
```

```
4328                            &evt_detail->args.pd.device_id);
4329                        if ((evt_detail->args.pd_state.prevState ==
4330                            PD_SYSTEM) &&
4155                        tgt = ddi_get16(acc_handle, &evt_detail->args.pd.device_
4156                        if ((evt_detail->args.pd_state.prevState == PD_SYSTEM) &
4331                            (evt_detail->args.pd_state.newState != PD_SYSTEM)) {
4332                            mutex_enter(&instance->config_dev_mtx);
4333                            instance->mr_tbolt_pd_list[tgt].flag =
4334                                (uint8_t)~MRDRV_TGT_VALID;
4335                            mutex_exit(&instance->config_dev_mtx);
4336                            rval = mrsas_service_evt(instance,
4337                                ddi_get16(acc_handle,
4338                                &evt_detail->args.pd.device_id),
4339                                1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4340                            con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4341                                "rval = %d tgt id = %d ", rval,
4342                                ddi_get16(acc_handle,
4343                                &evt_detail->args.pd.device_id)));
4344                            break;
4345                        }
4346                        if ((evt_detail->args.pd_state.prevState
4347                            == UNCONFIGURED_GOOD) &&
4348                            (evt_detail->args.pd_state.newState == PD_SYSTEM)) {
4349                            rval = mrsas_service_evt(instance,
4350                                ddi_get16(acc_handle,
4351                                &evt_detail->args.pd.device_id),
4352                                1, MRSAS_EVT_CONFIG_TGT, NULL);
4353                            con_log(CL_ANN1, (CE_WARN,
4354                                "mr_sas: PD_INSERTED: rval = %d "
4355                                " tgt id = %d ", rval,
4356                                ddi_get16(acc_handle,
4357                                &evt_detail->args.pd.device_id)));
4358                            break;
4359                        }
4360                    }
4361                    break;
4362            }
4363 #endif

4365            } /* End of Main Switch */

4367            /* get copy of seq_num and class/locale for re-registration */
4368            seq_num = ddi_get32(acc_handle, &evt_detail->seq_num);
4369            seq_num++;
4370            (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
4371                sizeof (struct mrsas_evt_detail));

4373            ddi_put8(acc_handle, &cmd->frame->dcmd.cmd_status, 0x0);
4374            ddi_put32(acc_handle, &cmd->frame->dcmd.mbox.w[0], seq_num);

4376            instance->aen_seq_num = seq_num;

4378            cmd->frame_count = 1;

4380            cmd->retry_count_for_ocr = 0;
4381            cmd->drv_pkt_time = 0;

4383            /* Issue the aen registration frame */
4384            instance->func_ptr->issue_cmd(cmd, instance);
4385 }
```

*_____unchanged_portion_omitted_*

```
4417 /*
4418  * Call this function inside mrsas_softintr.
4419  * mrsas_initiate_ocr_if_fw_is_faulty  - Initiates OCR if FW status is faulty
4420  * @instance:                  Adapter soft state
```



```
4421  */

4423 static uint32_t
4424 mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *instance)
4425 {
4426            uint32_t            cur_abs_reg_val;
4427            uint32_t            fw_state;

4429            cur_abs_reg_val =  instance->func_ptr->read_fw_status_reg(instance);
4430            fw_state = cur_abs_reg_val & MFI_STATE_MASK;
4431            if (fw_state == MFI_STATE_FAULT) {
4432                    if (instance->disable_online_ctrl_reset == 1) {
4433                            cmn_err(CE_WARN,
4434                                "mrsas_initiate_ocr_if_fw_is_faulty: "
4435                                "FW in Fault state, detected in ISR: "
4436                                "FW doesn't support ocr ");

4438                            return (ADAPTER_RESET_NOT_REQUIRED);
4439                    } else {
4440                            con_log(CL_ANN, (CE_NOTE,
4441                                "mrsas_initiate_ocr_if_fw_is_faulty: FW in Fault "
4442                                "state, detected in ISR: FW supports ocr "));
4266                                "mrsas_initiate_ocr_if_fw_is_faulty: "
4267                                "FW in Fault state, detected in ISR: FW supports

4444                            return (ADAPTER_RESET_REQUIRED);
4445                    }
4446            }

4448            return (ADAPTER_RESET_NOT_REQUIRED);
4449 }

4451 /*
4452  * mrsas_softintr - The Software ISR
4453  * @param arg   : HBA soft state
4454  *
4455  * called from high-level interrupt if hi-level interrupt are not there,
4456  * otherwise triggered as a soft interrupt
4457  */
4458 static uint_t
4459 mrsas_softintr(struct mrsas_instance *instance)
4460 {
4461            struct scsi_pkt         *pkt;
4462            struct scsa_cmd         *acmd;
4463            struct mrsas_cmd        *cmd;
4464            struct mlist_head       *pos, *next;
4465            mlist_t                 process_list;
4466            struct mrsas_header     *hdr;
4467            struct scsi_arq_status  *arqstat;

4469            con_log(CL_ANN1, (CE_NOTE, "mrsas_softintr() called."));

4471            ASSERT(instance);

4473            mutex_enter(&instance->completed_pool_mtx);

4475            if (mlist_empty(&instance->completed_pool_list)) {
4476                    mutex_exit(&instance->completed_pool_mtx);
4477                    return (DDI_INTR_CLAIMED);
4478            }

4480            instance->softint_running = 1;

4482            INIT_LIST_HEAD(&process_list);
4483            mlist_splice(&instance->completed_pool_list, &process_list);
4484            INIT_LIST_HEAD(&instance->completed_pool_list);
```

```
4486         mutex_exit(&instance->completed_pool_mtx);

4488         /* perform all callbacks first, before releasing the SCBs */
4489         mlist_for_each_safe(pos, next, &process_list) {
4490                 cmd = mlist_entry(pos, struct mrsas_cmd, list);

4492                 /* syncronize the Cmd frame for the controller */
4493                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle,
4494                     0, 0, DDI_DMA_SYNC_FORCPU);

4496                 if (mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
4497                     DDI_SUCCESS) {
4498                         mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
4499                         ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4500                         con_log(CL_ANN1, (CE_WARN,
4501                             "mrsas_softintr: "
4502                             "FMA check reports DMA handle failure"));
4503                         return (DDI_INTR_CLAIMED);
4504                 }

4506                 hdr = &cmd->frame->hdr;

4508                 /* remove the internal command from the process list */
4509                 mlist_del_init(&cmd->list);

4511                 switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
4512                 case MFI_CMD_OP_PD_SCSI:
4513                 case MFI_CMD_OP_LD_SCSI:
4514                 case MFI_CMD_OP_LD_READ:
4515                 case MFI_CMD_OP_LD_WRITE:
4516                         /*
4517                          * MFI_CMD_OP_PD_SCSI and MFI_CMD_OP_LD_SCSI
4518                          * could have been issued either through an
4519                          * IO path or an IOCTL path. If it was via IOCTL,
4520                          * we will send it to internal completion.
4521                          */
4522                         if (cmd->sync_cmd == MRSAS_TRUE) {
4523                                 complete_cmd_in_sync_mode(instance, cmd);
4524                                 break;
4525                         }

4527                         /* regular commands */
4528                         acmd =  cmd->cmd;
4529                         pkt =   CMD2PKT(acmd);

4531                         if (acmd->cmd_flags & CFLAG_DMAVALID) {
4532                                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
4533                                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
4534                                             acmd->cmd_dma_offset,
4535                                             acmd->cmd_dma_len,
4536                                             DDI_DMA_SYNC_FORCPU);
4537                                 }
4538                         }

4540                         pkt->pkt_reason         = CMD_CMPLT;
4541                         pkt->pkt_statistics     = 0;
4542                         pkt->pkt_state = STATE_GOT_BUS
4543                             | STATE_GOT_TARGET | STATE_SENT_CMD
4544                             | STATE_XFERRED_DATA | STATE_GOT_STATUS;

4546                         con_log(CL_ANN, (CE_CONT,
4547                             "CDB[0] = %x completed for %s: size %lx context %x",
4548                             pkt->pkt_cdbp[0], ((acmd->islogical) ? "LD" : "PD"),
4549                             acmd->cmd_dmacount, hdr->context));
4550                         DTRACE_PROBE3(softintr_cdb, uint8_t, pkt->pkt_cdbp[0],
```



```
4551                             uint_t, acmd->cmd_cdblen, ulong_t,
4552                             acmd->cmd_dmacount);

4554                         if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
4555                                 struct scsi_inquiry     *inq;

4557                                 if (acmd->cmd_dmacount != 0) {
4558                                         bp_mapin(acmd->cmd_buf);
4559                                         inq = (struct scsi_inquiry *)
4560                                             acmd->cmd_buf->b_un.b_addr;

4562                                         /* don't expose physical drives to OS */
4563                                         if (acmd->islogical &&
4564                                             (hdr->cmd_status == MFI_STAT_OK)) {
4565                                                 display_scsi_inquiry(
4566                                                     (caddr_t)inq);
4567                                         } else if ((hdr->cmd_status ==
4568                                             MFI_STAT_OK) && inq->inq_dtype ==
4569                                             DTYPE_DIRECT) {

4571                                                 display_scsi_inquiry(
4572                                                     (caddr_t)inq);

4574                                                 /* for physical disk */
4575                                                 hdr->cmd_status =
4576                                                     MFI_STAT_DEVICE_NOT_FOUND;
4577                                         }
4578                                 }
4579                         }

4581                         DTRACE_PROBE2(softintr_done, uint8_t, hdr->cmd,
4582                             uint8_t, hdr->cmd_status);

4584                         switch (hdr->cmd_status) {
4585                         case MFI_STAT_OK:
4586                                 pkt->pkt_scbp[0] = STATUS_GOOD;
4587                                 break;
4588                         case MFI_STAT_LD_CC_IN_PROGRESS:
4589                         case MFI_STAT_LD_RECON_IN_PROGRESS:
4590                                 pkt->pkt_scbp[0] = STATUS_GOOD;
4591                                 break;
4592                         case MFI_STAT_LD_INIT_IN_PROGRESS:
4593                                 con_log(CL_ANN,
4594                                     (CE_WARN, "Initialization in Progress"));
4402                                 con_log(CL_ANN, (CE_WARN, "Initialization in Pro
4595                                 pkt->pkt_reason = CMD_TRAN_ERR;

4597                                 break;
4598                         case MFI_STAT_SCSI_DONE_WITH_ERROR:
4599                                 con_log(CL_ANN, (CE_CONT, "scsi_done error"));

4601                                 pkt->pkt_reason = CMD_CMPLT;
4602                                 ((struct scsi_status *)
4603                                     pkt->pkt_scbp)->sts_chk = 1;

4605                                 if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
4606                                         con_log(CL_ANN,
4607                                             (CE_WARN, "TEST_UNIT_READY fail"));

4415                                         con_log(CL_ANN, (CE_WARN, "TEST_UNIT_REA

4608                                 } else {
4609                                         pkt->pkt_state |= STATE_ARQ_DONE;
4610                                         arqstat = (void *)(pkt->pkt_scbp);
4611                                         arqstat->sts_rqpkt_reason = CMD_CMPLT;
4612                                         arqstat->sts_rqpkt_resid = 0;
```

```
4613                                    arqstat->sts_rqpkt_state |=
4614                                        STATE_GOT_BUS | STATE_GOT_TARGET
4615                                        | STATE_SENT_CMD
4616                                        | STATE_XFERRED_DATA;
4617                                    *(uint8_t *)&arqstat->sts_rqpkt_status =
4618                                        STATUS_GOOD;
4619                                    ddi_rep_get8(
4620                                        cmd->frame_dma_obj.acc_handle,
4621                                        (uint8_t *)
4622                                        &(arqstat->sts_sensedata),
4623                                        cmd->sense,
4624                                        sizeof (struct scsi_extended_sense),
4625                                        DDI_DEV_AUTOINCR);
4626                            }
4627                            break;
4628                    case MFI_STAT_LD_OFFLINE:
4629                    case MFI_STAT_DEVICE_NOT_FOUND:
4630                            con_log(CL_ANN, (CE_CONT,
4631                            "mrsas_softintr:device not found error"));
4632                            pkt->pkt_reason = CMD_DEV_GONE;
4633                            pkt->pkt_statistics  = STAT_DISCON;
4634                            break;
4635                    case MFI_STAT_LD_LBA_OUT_OF_RANGE:
4636                            pkt->pkt_state |= STATE_ARQ_DONE;
4637                            pkt->pkt_reason = CMD_CMPLT;
4638                            ((struct scsi_status *)
4639                                pkt->pkt_scbp)->sts_chk = 1;

4641                            arqstat = (void *)(pkt->pkt_scbp);
4642                            arqstat->sts_rqpkt_reason = CMD_CMPLT;
4643                            arqstat->sts_rqpkt_resid = 0;
4644                            arqstat->sts_rqpkt_state |= STATE_GOT_BUS
4645                                | STATE_GOT_TARGET | STATE_SENT_CMD
4646                                | STATE_XFERRED_DATA;
4647                            *(uint8_t *)&arqstat->sts_rqpkt_status =
4648                                STATUS_GOOD;

4650                            arqstat->sts_sensedata.es_valid = 1;
4651                            arqstat->sts_sensedata.es_key =
4652                                KEY_ILLEGAL_REQUEST;
4653                            arqstat->sts_sensedata.es_class =
4654                                CLASS_EXTENDED_SENSE;

4656                            /*
4657                             * LOGICAL BLOCK ADDRESS OUT OF RANGE:
4658                             * ASC: 0x21h; ASCQ: 0x00h;
4659                             */
4660                            arqstat->sts_sensedata.es_add_code = 0x21;
4661                            arqstat->sts_sensedata.es_qual_code = 0x00;

4663                            break;

4665                    default:
4666                            con_log(CL_ANN, (CE_CONT, "Unknown status!"));
4667                            pkt->pkt_reason = CMD_TRAN_ERR;

4669                            break;
4670                    }

4672                    atomic_add_16(&instance->fw_outstanding, (-1));

4674                    (void) mrsas_common_check(instance, cmd);

4676                    if (acmd->cmd_dmahandle) {
4677                            if (mrsas_check_dma_handle(
4678                                acmd->cmd_dmahandle) != DDI_SUCCESS) {
```



```
4679                                    ddi_fm_service_impact(instance->dip,
4680                                        DDI_SERVICE_UNAFFECTED);
4681                                    pkt->pkt_reason = CMD_TRAN_ERR;
4682                                    pkt->pkt_statistics = 0;
4683                            }
4684                    }

4686                    /* Call the callback routine */
4687                    if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4688                        pkt->pkt_comp) {

4690                            con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_softintr: "
4691                                "posting to scsa cmd %p index %x pkt %p "
4692                                "time %llx", (void *)cmd, cmd->index,
4693                                (void *)pkt, gethrtime()));
4694                            (*pkt->pkt_comp)(pkt);

4696                    }

4698                    return_mfi_pkt(instance, cmd);
4699                    break;

4701            case MFI_CMD_OP_SMP:
4702            case MFI_CMD_OP_STP:
4703                    complete_cmd_in_sync_mode(instance, cmd);
4704                    break;

4706            case MFI_CMD_OP_DCMD:
4707                    /* see if got an event notification */
4708                    if (ddi_get32(cmd->frame_dma_obj.acc_handle,
4709                        &cmd->frame->dcmd.opcode) ==
4710                        MR_DCMD_CTRL_EVENT_WAIT) {
4711                            if ((instance->aen_cmd == cmd) &&
4712                                (instance->aen_cmd->abort_aen)) {
4713                                    con_log(CL_ANN, (CE_WARN,
4714                                        "mrsas_softintr: "
4715                                        "aborted_aen returned"));
4716                            } else {
4717                                    atomic_add_16(&instance->fw_outstanding,
4718                                        (-1));
4719                                    service_mfi_aen(instance, cmd);
4720                            }
4721                    } else {
4722                            complete_cmd_in_sync_mode(instance, cmd);
4723                    }

4725                    break;

4727            case MFI_CMD_OP_ABORT:
4728                    con_log(CL_ANN, (CE_NOTE, "MFI_CMD_OP_ABORT complete"));
4729                    /*
4730                     * MFI_CMD_OP_ABORT successfully completed
4731                     * in the synchronous mode
4732                     */
4733                    complete_cmd_in_sync_mode(instance, cmd);
4734                    break;

4736            default:
4737                    mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
4738                    ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

4740                    if (cmd->pkt != NULL) {
4741                            pkt = cmd->pkt;
4742                            if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4743                                pkt->pkt_comp) {
```

```
4745                                    con_log(CL_ANN1, (CE_CONT, "posting to "
4746                                        "scsa cmd %p index %x pkt %p"
4747                                        "time %llx, default ", (void *)cmd,
4748                                        cmd->index, (void *)pkt,
4749                                        gethrtime()));

4751                                    (*pkt->pkt_comp)(pkt);

4753                                }
4754                        }
4755                        con_log(CL_ANN, (CE_WARN, "Cmd type unknown !"));
4756                        break;
4757                }
4758        }

4760        instance->softint_running = 0;

4762        return (DDI_INTR_CLAIMED);
4763 }

4765 /*
4766  * mrsas_alloc_dma_obj
4767  *
4768  * Allocate the memory and other resources for an dma object.
4769  */
4770 int
4771 mrsas_alloc_dma_obj(struct mrsas_instance *instance, dma_obj_t *obj,
4772     uchar_t endian_flags)
4773 {
4774        int     i;
4775        size_t  alen = 0;
4776        uint_t  cookie_cnt;
4777        struct ddi_device_acc_attr tmp_endian_attr;

4779        tmp_endian_attr = endian_attr;
4780        tmp_endian_attr.devacc_attr_endian_flags = endian_flags;
4781        tmp_endian_attr.devacc_attr_access = DDI_DEFAULT_ACC;

4783        i = ddi_dma_alloc_handle(instance->dip, &obj->dma_attr,
4784            DDI_DMA_SLEEP, NULL, &obj->dma_handle);
4785        if (i != DDI_SUCCESS) {

4787                switch (i) {
4788                        case DDI_DMA_BADATTR :
4789                                con_log(CL_ANN, (CE_WARN,
4790                                "Failed ddi_dma_alloc_handle- Bad attribute"));
4791                                break;
4792                        case DDI_DMA_NORESOURCES :
4793                                con_log(CL_ANN, (CE_WARN,
4794                                "Failed ddi_dma_alloc_handle- No Resources"));
4795                                break;
4796                        default :
4797                                con_log(CL_ANN, (CE_WARN,
4798                                "Failed ddi_dma_alloc_handle: "
4799                                "unknown status %d", i));
4800                                break;
4801                }

4803                return (-1);
4804        }

4806        if ((ddi_dma_mem_alloc(obj->dma_handle, obj->size, &tmp_endian_attr,
4807            DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL,
4808            &obj->buffer, &alen, &obj->acc_handle) != DDI_SUCCESS) ||
4809            alen < obj->size) {
```



```
4811                ddi_dma_free_handle(&obj->dma_handle);

4813                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_mem_alloc"));

4815                return (-1);
4816        }
4611        if (obj->dma_handle == NULL) {
4612                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_mem_alloc"));
4613                return (-1);
4614        }

4818        if (ddi_dma_addr_bind_handle(obj->dma_handle, NULL, obj->buffer,
4819            obj->size, DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP,
4820            NULL, &obj->dma_cookie[0], &cookie_cnt) != DDI_SUCCESS) {

4822                ddi_dma_mem_free(&obj->acc_handle);
4823                ddi_dma_free_handle(&obj->dma_handle);

4825                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_addr_bind_handle"));

4827                return (-1);
4828        }
4627        if (obj->acc_handle == NULL) {
4628                ddi_dma_mem_free(&obj->acc_handle);
4629                ddi_dma_free_handle(&obj->dma_handle);

4830        if (mrsas_check_dma_handle(obj->dma_handle) != DDI_SUCCESS) {
4831                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4631                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_addr_bind_handle"));
4832                return (-1);
4833        }

4835        if (mrsas_check_acc_handle(obj->acc_handle) != DDI_SUCCESS) {
4836                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4837                return (-1);
4838        }

4840        return (cookie_cnt);
4841 }

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

4853        if ((obj.dma_handle == NULL) || (obj.acc_handle == NULL)) {
4649        if ( (obj.dma_handle == NULL) || (obj.acc_handle == NULL) ) {
4854                return (DDI_SUCCESS);
4855        }

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

4866        if (mrsas_check_acc_handle(obj.acc_handle) != DDI_SUCCESS) {
4867                ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
```

```
4868                    return (DDI_FAILURE);
4869            }

4871            (void) ddi_dma_unbind_handle(obj.dma_handle);
4872            ddi_dma_mem_free(&obj.acc_handle);
4873            ddi_dma_free_handle(&obj.dma_handle);
4874            obj.acc_handle = NULL;
4875            return (DDI_SUCCESS);
4876    }

4878    /*
4879     * mrsas_dma_alloc(instance_t *, struct scsi_pkt *, struct buf *,
4880     * int, int (*)())
4881     *
4882     * Allocate dma resources for a new scsi command
4883     */
4884    int
4885    mrsas_dma_alloc(struct mrsas_instance *instance, struct scsi_pkt *pkt,
4886        struct buf *bp, int flags, int (*callback)())
4887    {
4888            int     dma_flags;
4889            int     (*cb)(caddr_t);
4890            int     i;

4892            ddi_dma_attr_t  tmp_dma_attr = mrsas_generic_dma_attr;
4893            struct scsa_cmd *acmd = PKT2CMD(pkt);

4895            acmd->cmd_buf = bp;

4897            if (bp->b_flags & B_READ) {
4898                    acmd->cmd_flags &= ~CFLAG_DMASEND;
4899                    dma_flags = DDI_DMA_READ;
4900            } else {
4901                    acmd->cmd_flags |= CFLAG_DMASEND;
4902                    dma_flags = DDI_DMA_WRITE;
4903            }

4905            if (flags & PKT_CONSISTENT) {
4906                    acmd->cmd_flags |= CFLAG_CONSISTENT;
4907                    dma_flags |= DDI_DMA_CONSISTENT;
4908            }

4910            if (flags & PKT_DMA_PARTIAL) {
4911                    dma_flags |= DDI_DMA_PARTIAL;
4912            }

4914            dma_flags |= DDI_DMA_REDZONE;

4916            cb = (callback == NULL_FUNC) ? DDI_DMA_DONTWAIT : DDI_DMA_SLEEP;

4918            tmp_dma_attr.dma_attr_sgllen = instance->max_num_sge;
4919            tmp_dma_attr.dma_attr_addr_hi = 0xffffffffffffffffull;
4920            if (instance->tbolt) {
4921                    /* OCR-RESET FIX */
4922                    tmp_dma_attr.dma_attr_count_max =
4923                        (U64)mrsas_tbolt_max_cap_maxxfer;  /* limit to 256K */
4924                    tmp_dma_attr.dma_attr_maxxfer =
4925                        (U64)mrsas_tbolt_max_cap_maxxfer;  /* limit to 256K */
4703                    //OCR-RESET FIX
4704                    tmp_dma_attr.dma_attr_count_max = (U64)mrsas_tbolt_max_cap_maxxf
4705                    tmp_dma_attr.dma_attr_maxxfer = (U64)mrsas_tbolt_max_cap_maxxfer
4926            }


4928            if ((i = ddi_dma_alloc_handle(instance->dip, &tmp_dma_attr,
4929                cb, 0, &acmd->cmd_dmahandle)) != DDI_SUCCESS) {
```



```
4930                    switch (i) {
4931                    case DDI_DMA_BADATTR:
4932                            bioerror(bp, EFAULT);
4933                            return (DDI_FAILURE);

4935                    case DDI_DMA_NORESOURCES:
4936                            bioerror(bp, 0);
4937                            return (DDI_FAILURE);

4939                    default:
4940                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_alloc_handle: "
4941                                "impossible result (0x%x)", i));
4942                            bioerror(bp, EFAULT);
4943                            return (DDI_FAILURE);
4944                    }
4945            }

4947            i = ddi_dma_buf_bind_handle(acmd->cmd_dmahandle, bp, dma_flags,
4948                cb, 0, &acmd->cmd_dmacookies[0], &acmd->cmd_ncookies);

4950            switch (i) {
4951            case DDI_DMA_PARTIAL_MAP:
4952                    if ((dma_flags & DDI_DMA_PARTIAL) == 0) {
4953                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle: "
4954                                "DDI_DMA_PARTIAL_MAP impossible"));
4955                            goto no_dma_cookies;
4956                    }

4958                    if (ddi_dma_numwin(acmd->cmd_dmahandle, &acmd->cmd_nwin) ==
4959                        DDI_FAILURE) {
4960                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_numwin failed"));
4961                            goto no_dma_cookies;
4962                    }

4964                    if (ddi_dma_getwin(acmd->cmd_dmahandle, acmd->cmd_curwin,
4965                        &acmd->cmd_dma_offset, &acmd->cmd_dma_len,
4966                        &acmd->cmd_dmacookies[0], &acmd->cmd_ncookies) ==
4967                        DDI_FAILURE) {

4969                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_getwin failed"));
4970                            goto no_dma_cookies;
4971                    }

4973                    goto get_dma_cookies;
4974            case DDI_DMA_MAPPED:
4975                    acmd->cmd_nwin = 1;
4976                    acmd->cmd_dma_len = 0;
4977                    acmd->cmd_dma_offset = 0;

4979    get_dma_cookies:
4980                    i = 0;
4981                    acmd->cmd_dmacount = 0;
4982                    for (;;) {
4983                            acmd->cmd_dmacount +=
4984                                acmd->cmd_dmacookies[i++].dmac_size;

4986                            if (i == instance->max_num_sge ||
4987                                i == acmd->cmd_ncookies)
4988                                    break;

4990                            ddi_dma_nextcookie(acmd->cmd_dmahandle,
4991                                &acmd->cmd_dmacookies[i]);
4992                    }

4994                    acmd->cmd_cookie = i;
4995                    acmd->cmd_cookiecnt = i;
```

```
4997                 acmd->cmd_flags |= CFLAG_DMAVALID;

4999                 if (bp->b_bcount >= acmd->cmd_dmacount) {
5000                         pkt->pkt_resid = bp->b_bcount - acmd->cmd_dmacount;
5001                 } else {
5002                         pkt->pkt_resid = 0;
5003                 }

5005                 return (DDI_SUCCESS);
5006         case DDI_DMA_NORESOURCES:
5007                 bioerror(bp, 0);
5008                 break;
5009         case DDI_DMA_NOMAPPING:
5010                 bioerror(bp, EFAULT);
5011                 break;
5012         case DDI_DMA_TOOBIG:
5013                 bioerror(bp, EINVAL);
5014                 break;
5015         case DDI_DMA_INUSE:
5016                 con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle:"
5017                     " DDI_DMA_INUSE impossible"));
5018                 break;
5019         default:
5020                 con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle: "
5021                     "impossible result (0x%x)", i));
5022                 break;
5023         }

5025 no_dma_cookies:
5026         ddi_dma_free_handle(&acmd->cmd_dmahandle);
5027         acmd->cmd_dmahandle = NULL;
5028         acmd->cmd_flags &= ~CFLAG_DMAVALID;
5029         return (DDI_FAILURE);
5030 }
```

*unchanged_portion_omitted_*

```
5099 /*
5100  * build_cmd
5101  */
5102 static struct mrsas_cmd *
5103 build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
5104     struct scsi_pkt *pkt, uchar_t *cmd_done)
5105 {
5106         uint16_t        flags = 0;
5107         uint32_t        i;
5108         uint32_t        context;
5109         uint32_t        sge_bytes;
5110         uint32_t        tmp_data_xfer_len;
5111         ddi_acc_handle_t acc_handle;
5112         struct mrsas_cmd                *cmd;
5113         struct mrsas_sge64              *mfi_sgl;
5114         struct mrsas_sge_ieee           *mfi_sgl_ieee;
5115         struct scsa_cmd                 *acmd = PKT2CMD(pkt);
5116         struct mrsas_pthru_frame        *pthru;
5117         struct mrsas_io_frame           *ldio;

5119         /* find out if this is logical or physical drive command.  */
5120         acmd->islogical = MRDRV_IS_LOGICAL(ap);
5121         acmd->device_id = MAP_DEVICE_ID(instance, ap);
5122         *cmd_done = 0;

5124         /* get the command packet */
5125         if (!(cmd = get_mfi_pkt(instance))) {
5126                 DTRACE_PROBE2(build_cmd_mfi_err, uint16_t,
5127                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
```



```
4907                 cmn_err(CE_WARN,
4908                     "Failed to get a cmd from free-pool in build_cmd(). fw_outst
4909                         instance->fw_outstanding, instance->max_fw_cmds)
5128                 return (NULL);
5129         }

5131         acc_handle = cmd->frame_dma_obj.acc_handle;

5133         /* Clear the frame buffer and assign back the context id */
5134         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
5135         ddi_put32(acc_handle, &cmd->frame->hdr.context, cmd->index);

5137         cmd->pkt = pkt;
5138         cmd->cmd = acmd;
5139         DTRACE_PROBE3(build_cmds, uint8_t, pkt->pkt_cdbp[0],
5140             ulong_t, acmd->cmd_dmacount, ulong_t, acmd->cmd_dma_len);

5142         /* lets get the command directions */
5143         if (acmd->cmd_flags & CFLAG_DMASEND) {
5144                 flags = MFI_FRAME_DIR_WRITE;

5146                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5147                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5148                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5149                             DDI_DMA_SYNC_FORDEV);
5150                 }
5151         } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
5152                 flags = MFI_FRAME_DIR_READ;

5154                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5155                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5156                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5157                             DDI_DMA_SYNC_FORCPU);
5158                 }
5159         } else {
5160                 flags = MFI_FRAME_DIR_NONE;
5161         }

5163         if (instance->flag_ieee) {
5164                 flags |= MFI_FRAME_IEEE;
5165         }
5166         flags |= MFI_FRAME_SGL64;

5168         switch (pkt->pkt_cdbp[0]) {

5170         /*
5171          * case SCMD_SYNCHRONIZE_CACHE:
5172          *      flush_cache(instance);
5173          *      return_mfi_pkt(instance, cmd);
5174          *      *cmd_done = 1;
5175          *
5176          *      return (NULL);
5177          */

5179         case SCMD_READ:
5180         case SCMD_WRITE:
5181         case SCMD_READ_G1:
5182         case SCMD_WRITE_G1:
5183         case SCMD_READ_G4:
5184         case SCMD_WRITE_G4:
5185         case SCMD_READ_G5:
5186         case SCMD_WRITE_G5:
5187                 if (acmd->islogical) {
5188                         ldio = (struct mrsas_io_frame *)cmd->frame;

5190                         /*
```

```
5191                     * preare the Logical IO frame:
5192                     * 2nd bit is zero for all read cmds
5193                     */
5194                    ddi_put8(acc_handle, &ldio->cmd,
5195                        (pkt->pkt_cdbp[0] & 0x02) ? MFI_CMD_OP_LD_WRITE
5196                        : MFI_CMD_OP_LD_READ);
5197                    ddi_put8(acc_handle, &ldio->cmd_status, 0x0);
5198                    ddi_put8(acc_handle, &ldio->scsi_status, 0x0);
5199                    ddi_put8(acc_handle, &ldio->target_id, acmd->device_id);
5200                    ddi_put16(acc_handle, &ldio->timeout, 0);
5201                    ddi_put8(acc_handle, &ldio->reserved_0, 0);
5202                    ddi_put16(acc_handle, &ldio->pad_0, 0);
5203                    ddi_put16(acc_handle, &ldio->flags, flags);

5205                    /* Initialize sense Information */
5206                    bzero(cmd->sense, SENSE_LENGTH);
5207                    ddi_put8(acc_handle, &ldio->sense_len, SENSE_LENGTH);
5208                    ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_hi, 0);
5209                    ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_lo,
5210                        cmd->sense_phys_addr);
5211                    ddi_put32(acc_handle, &ldio->start_lba_hi, 0);
5212                    ddi_put8(acc_handle, &ldio->access_byte,
5213                        (acmd->cmd_cdblen != 6) ? pkt->pkt_cdbp[1] : 0);
5214                    ddi_put8(acc_handle, &ldio->sge_count,
5215                        acmd->cmd_cookiecnt);
5216                    if (instance->flag_ieee) {
5217                            mfi_sgl_ieee =
5218                                (struct mrsas_sge_ieee *)&ldio->sgl;
5219                    } else {
5220                            mfi_sgl = (struct mrsas_sge64   *)&ldio->sgl;
5221                    }

5223                    context = ddi_get32(acc_handle, &ldio->context);

5225                    if (acmd->cmd_cdblen == CDB_GROUP0) {
5226                            /* 6-byte cdb */
5005                    if (acmd->cmd_cdblen == CDB_GROUP0) {
5227                            ddi_put32(acc_handle, &ldio->lba_count, (
5228                                (uint16_t)(pkt->pkt_cdbp[4])));

5230                            ddi_put32(acc_handle, &ldio->start_lba_lo, (
5231                                ((uint32_t)(pkt->pkt_cdbp[3])) |
5232                                ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
5233                                ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
5234                                << 16)));
5235                    } else if (acmd->cmd_cdblen == CDB_GROUP1) {
5236                            /* 10-byte cdb */
5014                    } else if (acmd->cmd_cdblen == CDB_GROUP1) {
5237                            ddi_put32(acc_handle, &ldio->lba_count, (
5238                                ((uint16_t)(pkt->pkt_cdbp[8])) |
5239                                ((uint16_t)(pkt->pkt_cdbp[7]) << 8)));

5241                            ddi_put32(acc_handle, &ldio->start_lba_lo, (
5242                                ((uint32_t)(pkt->pkt_cdbp[5])) |
5243                                ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5244                                ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5245                                ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5246                    } else if (acmd->cmd_cdblen == CDB_GROUP5) {
5247                            /* 12-byte cdb */
5024                    } else if (acmd->cmd_cdblen == CDB_GROUP5) {
5248                            ddi_put32(acc_handle, &ldio->lba_count, (
5249                                ((uint32_t)(pkt->pkt_cdbp[9])) |
5250                                ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5251                                ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5252                                ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));
```



```
5254                                    ddi_put32(acc_handle, &ldio->start_lba_lo, (
5255                                        ((uint32_t)(pkt->pkt_cdbp[5])) |
5256                                        ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5257                                        ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5258                                        ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5259                            } else if (acmd->cmd_cdblen == CDB_GROUP4) {
5260                                    /* 16-byte cdb */
5036                            } else if (acmd->cmd_cdblen == CDB_GROUP4) {
5261                                    ddi_put32(acc_handle, &ldio->lba_count, (
5262                                        ((uint32_t)(pkt->pkt_cdbp[13])) |
5263                                        ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
5264                                        ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
5265                                        ((uint32_t)(pkt->pkt_cdbp[10]) << 24)));

5267                                    ddi_put32(acc_handle, &ldio->start_lba_lo, (
5268                                        ((uint32_t)(pkt->pkt_cdbp[9])) |
5269                                        ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5270                                        ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5271                                        ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));

5273                                    ddi_put32(acc_handle, &ldio->start_lba_hi, (
5274                                        ((uint32_t)(pkt->pkt_cdbp[5])) |
5275                                        ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5276                                        ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5277                                        ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5278                            }

5280                            break;
5281                    }
5282                    /* fall through For all non-rd/wr cmds */
5283            default:

5285                    switch (pkt->pkt_cdbp[0]) {
5286                    case SCMD_MODE_SENSE:
5287                    case SCMD_MODE_SENSE_G1: {
5288                            union scsi_cdb  *cdbp;
5289                            uint16_t        page_code;

5291                            cdbp = (void *)pkt->pkt_cdbp;
5292                            page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
5293                            switch (page_code) {
5294                            case 0x3:
5295                            case 0x4:
5296                                    (void) mrsas_mode_sense_build(pkt);
5297                                    return_mfi_pkt(instance, cmd);
5298                                    *cmd_done = 1;
5299                                    return (NULL);
5300                            }
5301                            break;
5302                    }
5303                    default:
5304                            break;
5305                    }

5307                    pthru   = (struct mrsas_pthru_frame *)cmd->frame;

5309                    /* prepare the DCDB frame */
5310                    ddi_put8(acc_handle, &pthru->cmd, (acmd->islogical) ?
5311                        MFI_CMD_OP_LD_SCSI : MFI_CMD_OP_PD_SCSI);
5312                    ddi_put8(acc_handle, &pthru->cmd_status, 0x0);
5313                    ddi_put8(acc_handle, &pthru->scsi_status, 0x0);
5314                    ddi_put8(acc_handle, &pthru->target_id, acmd->device_id);
5315                    ddi_put8(acc_handle, &pthru->lun, 0);
5316                    ddi_put8(acc_handle, &pthru->cdb_len, acmd->cmd_cdblen);
5317                    ddi_put16(acc_handle, &pthru->timeout, 0);
5318                    ddi_put16(acc_handle, &pthru->flags, flags);
```

```
5319                 tmp_data_xfer_len = 0;
5320                 for (i = 0; i < acmd->cmd_cookiecnt; i++) {
5321                         tmp_data_xfer_len += acmd->cmd_dmacookies[i].dmac_size;
5322                 }
5323                 ddi_put32(acc_handle, &pthru->data_xfer_len,
5324                     tmp_data_xfer_len);
5325                 ddi_put8(acc_handle, &pthru->sge_count, acmd->cmd_cookiecnt);
5326                 if (instance->flag_ieee) {
5327                         mfi_sgl_ieee = (struct mrsas_sge_ieee *)&pthru->sgl;
5328                 } else {
5329                         mfi_sgl = (struct mrsas_sge64 *)&pthru->sgl;
5330                 }

5332                 bzero(cmd->sense, SENSE_LENGTH);
5333                 ddi_put8(acc_handle, &pthru->sense_len, SENSE_LENGTH);
5334                 ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
5335                 ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo,
5336                     cmd->sense_phys_addr);

5338                 context = ddi_get32(acc_handle, &pthru->context);
5339                 ddi_rep_put8(acc_handle, (uint8_t *)pkt->pkt_cdbp,
5340                     (uint8_t *)pthru->cdb, acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

5342                 break;
5343         }
5344 #ifdef lint
5345         context = context;
5346 #endif
5347         /* prepare the scatter-gather list for the firmware */
5348         if (instance->flag_ieee) {
5349                 for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl_ieee++) {
5350                         ddi_put64(acc_handle, &mfi_sgl_ieee->phys_addr,
5351                             acmd->cmd_dmacookies[i].dmac_laddress);
5352                         ddi_put32(acc_handle, &mfi_sgl_ieee->length,
5353                             acmd->cmd_dmacookies[i].dmac_size);
5354                 }
5355                 sge_bytes = sizeof (struct mrsas_sge_ieee)*acmd->cmd_cookiecnt;
5356         } else {
5357                 for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl++) {
5358                         ddi_put64(acc_handle, &mfi_sgl->phys_addr,
5359                             acmd->cmd_dmacookies[i].dmac_laddress);
5360                         ddi_put32(acc_handle, &mfi_sgl->length,
5361                             acmd->cmd_dmacookies[i].dmac_size);
5362                 }
5363                 sge_bytes = sizeof (struct mrsas_sge64)*acmd->cmd_cookiecnt;
5364         }

5366         cmd->frame_count = (sge_bytes / MRMFI_FRAME_SIZE) +
5367             ((sge_bytes % MRMFI_FRAME_SIZE) ? 1 : 0) + 1;

5369         if (cmd->frame_count >= 8) {
5370                 cmd->frame_count = 8;
5371         }

5373         return (cmd);
5374 }

5376 #ifndef __sparc
5377 /*
5378  * wait_for_outstanding -       Wait for all outstanding cmds
5379  * @instance:                          Adapter soft state
5380  *
5381  * This function waits for upto MRDRV_RESET_WAIT_TIME seconds for FW to
5382  * complete all its outstanding commands. Returns error if one or more IOs
5383  * are pending after this time period.
5384  */
```



```
5385 static int
5386 wait_for_outstanding(struct mrsas_instance *instance)
5387 {
5388         int             i;
5389         uint32_t        wait_time = 90;

5391         for (i = 0; i < wait_time; i++) {
5392                 if (!instance->fw_outstanding) {
5393                         break;
5394                 }

5396                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */;
5397         }

5399         if (instance->fw_outstanding) {
5400                 return (1);
5401         }

5403         return (0);
5404 }
5405 #endif  /* __sparc */


5407 /*
5408  * issue_mfi_pthru
5409  */
5410 static int
5411 issue_mfi_pthru(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5412     struct mrsas_cmd *cmd, int mode)
5413 {
5414         void            *ubuf;
5415         uint32_t        kphys_addr = 0;
5416         uint32_t        xferlen = 0;
5417         uint32_t        new_xfer_length = 0;
5192         uint32_t new_xfer_length =0;
5418         uint_t          model;
5419         ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5420         dma_obj_t                       pthru_dma_obj;
5421         struct mrsas_pthru_frame        *kpthru;
5422         struct mrsas_pthru_frame        *pthru;
5423         int i;
5424         pthru = &cmd->frame->pthru;
5425         kpthru = (struct mrsas_pthru_frame *)&ioctl->frame[0];

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

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

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

```
5449         }

5451         if (xferlen) {
5452                 /* means IOCTL requires DMA */
5453                 /* allocate the data transfer buffer */
5454                 /* pthru_dma_obj.size = xferlen; */
5455                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
5456                     PAGESIZE);
5229                 //pthru_dma_obj.size = xferlen;
5230                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen,new_xfer_length,PAGESIZE)
5457                 pthru_dma_obj.size = new_xfer_length;
5458                 pthru_dma_obj.dma_attr = mrsas_generic_dma_attr;
5459                 pthru_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5460                 pthru_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5461                 pthru_dma_obj.dma_attr.dma_attr_sgllen = 1;
5462                 pthru_dma_obj.dma_attr.dma_attr_align = 1;

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

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

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

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

5501         ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
5502         pthru->sense_buf_phys_addr_lo = cmd->sense_phys_addr;
5503         /* ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo, 0); */
5277         /*ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo, 0); */

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

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



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

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

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

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

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

5545         if (kpthru->sense_len) {
5546                 uint_t sense_len = SENSE_LENGTH;
5547                 void *sense_ubuf =
5548                     (void *)(ulong_t)kpthru->sense_buf_phys_addr_lo;
5318                 uint sense_len = SENSE_LENGTH;
5319                 void *sense_ubuf = (void *)(ulong_t)kpthru->sense_buf_phys_addr_
5549                 if (kpthru->sense_len <= SENSE_LENGTH) {
5550                         sense_len = kpthru->sense_len;
5551                 }

5553                 for (i = 0; i < sense_len; i++) {
5554                         if (ddi_copyout(
5555                             (uint8_t *)cmd->sense+i,
5556                             (uint8_t *)sense_ubuf+i, 1, mode)) {
5557                                 con_log(CL_ANN, (CE_WARN,
5558                                     "issue_mfi_pthru : "
5559                                     "copy to user space failed"));
5560                         }
5561                         con_log(CL_DLEVEL1, (CE_WARN,
5562                             "Copying Sense info sense_buff[%d] = 0x%X",
5563                             i, *((uint8_t *)cmd->sense + i)));
5333                             "Copying Sense info sense_buff[%d] = 0x%X\n",i,*((ui
5564                 }
5565         }
5566         (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
5567             DDI_DMA_SYNC_FORDEV);

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

```
5575         return (DDI_SUCCESS);
5576 }

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

5598         if (instance->adapterresetinprogress) {
5599                 con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
5600                 "returning mfi_pkt and setting TRAN_BUSY"));
5369                 con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
5370                 "returning mfi_pkt and setting TRAN_BUSY\n"));
5601                 return (DDI_FAILURE);
5602         }
5603         model = ddi_model_convert_from(mode & FMODELS);
5604         if (model == DDI_MODEL_ILP32) {
5605                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));

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

5609                 ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
5610         } else {
5611 #ifdef _ILP32
5612                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));
5613                 xferlen = kdcmd->sgl.sge32[0].length;
5614                 ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
5615 #else
5616                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_LP64"));
5617                 xferlen = kdcmd->sgl.sge64[0].length;
5618                 ubuf    = (void *)(ulong_t)kdcmd->sgl.sge64[0].phys_addr;
5619 #endif
5620         }
5621         if (xferlen) {
5622                 /* means IOCTL requires DMA */
5623                 /* allocate the data transfer buffer */
5624                 /* dcmd_dma_obj.size = xferlen; */
5625                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
5626                     PAGESIZE);
5394                 //dcmd_dma_obj.size = xferlen;
5395                  MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen,new_xfer_length,PAGESIZE
5627                 dcmd_dma_obj.size = new_xfer_length;
5628                 dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
5629                 dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5630                 dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5631                 dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
5632                 dcmd_dma_obj.dma_attr.dma_attr_align = 1;

5634                 /* allocate kernel buffer for DMA */
5635                         if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
5636                             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
```



```
5637                                 con_log(CL_ANN,
5638                                     (CE_WARN, "issue_mfi_dcmd: could not "
5639                                     "allocate data transfer buffer."));
5406                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_dcmd: "
5407                                     "could not allocate data transfer buffer."))
5640                                 return (DDI_FAILURE);
5641                         }
5642                 (void) memset(dcmd_dma_obj.buffer, 0, xferlen);

5644                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5645                 if (kdcmd->flags & MFI_FRAME_DIR_WRITE) {
5646                         for (i = 0; i < xferlen; i++) {
5647                                 if (ddi_copyin((uint8_t *)ubuf + i,
5648                                     (uint8_t *)dcmd_dma_obj.buffer + i,
5649                                     1, mode)) {
5650                                         con_log(CL_ANN, (CE_WARN,
5651                                             "issue_mfi_dcmd : "
5652                                             "copy from user space failed"));
5653                                         return (DDI_FAILURE);
5654                                 }
5655                         }
5656                 }

5658                 kphys_addr = dcmd_dma_obj.dma_cookie[0].dmac_address;
5659         }

5661         ddi_put8(acc_handle, &dcmd->cmd, kdcmd->cmd);
5662         ddi_put8(acc_handle, &dcmd->cmd_status, 0);
5663         ddi_put8(acc_handle, &dcmd->sge_count, kdcmd->sge_count);
5664         ddi_put16(acc_handle, &dcmd->timeout, kdcmd->timeout);
5665         ddi_put32(acc_handle, &dcmd->data_xfer_len, kdcmd->data_xfer_len);
5666         ddi_put32(acc_handle, &dcmd->opcode, kdcmd->opcode);

5668         ddi_rep_put8(acc_handle, (uint8_t *)kdcmd->mbox.b,
5669             (uint8_t *)dcmd->mbox.b, DCMD_MBOX_SZ, DDI_DEV_AUTOINCR);

5671         ddi_put16(acc_handle, &dcmd->flags, kdcmd->flags & ~MFI_FRAME_SGL64);
5672         ddi_put32(acc_handle, &dcmd->sgl.sge32[0].length, xferlen);
5673         ddi_put32(acc_handle, &dcmd->sgl.sge32[0].phys_addr, kphys_addr);

5675         cmd->sync_cmd = MRSAS_TRUE;
5676         cmd->frame_count = 1;

5678         if (instance->tbolt) {
5679                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5680         }

5682         if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5683                 con_log(CL_ANN, (CE_WARN, "issue_mfi_dcmd: fw_ioctl failed"));
5684         } else {
5685                 if (xferlen && (kdcmd->flags & MFI_FRAME_DIR_READ)) {
5686                         for (i = 0; i < xferlen; i++) {
5687                                 if (ddi_copyout(
5688                                     (uint8_t *)dcmd_dma_obj.buffer + i,
5689                                     (uint8_t *)ubuf + i,
5690                                     1, mode)) {
5691                                         con_log(CL_ANN, (CE_WARN,
5692                                             "issue_mfi_dcmd : "
5693                                             "copy to user space failed"));
5694                                         return (DDI_FAILURE);
5695                                 }
5696                         }
5697                 }
5698         }

5700         kdcmd->cmd_status = ddi_get8(acc_handle, &dcmd->cmd_status);
```

```
5701         con_log(CL_ANN,
5702             (CE_CONT, "issue_mfi_dcmd: cmd_status %x", kdcmd->cmd_status));
5703         DTRACE_PROBE3(issue_dcmd, uint32_t, kdcmd->opcode, uint8_t,
5704             kdcmd->cmd, uint8_t, kdcmd->cmd_status);
5469         con_log(CL_ANN, (CE_CONT, "issue_mfi_dcmd: cmd_status %x", kdcmd->cmd_st

5706         if (xferlen) {
5707                 /* free kernel buffer */
5708                 if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
5709                         return (DDI_FAILURE);
5710         }

5712         return (DDI_SUCCESS);
5713 }

5715 /*
5716  * issue_mfi_smp
5717  */
5718 static int
5719 issue_mfi_smp(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5720     struct mrsas_cmd *cmd, int mode)
5721 {
5722         void            *request_ubuf;
5723         void            *response_ubuf;
5724         uint32_t        request_xferlen = 0;
5725         uint32_t        response_xferlen = 0;
5726         uint32_t        new_xfer_length1 = 0;
5727         uint32_t        new_xfer_length2 = 0;
5728         uint_t          model;
5729         dma_obj_t                       request_dma_obj;
5730         dma_obj_t                       response_dma_obj;
5731         ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5732         struct mrsas_smp_frame          *ksmp;
5733         struct mrsas_smp_frame          *smp;
5734         struct mrsas_sge32              *sge32;
5735 #ifndef _ILP32
5736         struct mrsas_sge64              *sge64;
5737 #endif
5738         int i;
5739         uint64_t                        tmp_sas_addr;

5741         smp = &cmd->frame->smp;
5742         ksmp = (struct mrsas_smp_frame *)&ioctl->frame[0];

5744         if (instance->adapterresetinprogress) {
5745                 con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
5746                 "returning mfi_pkt and setting TRAN_BUSY\n"));
5747                 return (DDI_FAILURE);
5748         }
5749         model = ddi_model_convert_from(mode & FMODELS);
5750         if (model == DDI_MODEL_ILP32) {
5751                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_ILP32"));

5753                 sge32                   = &ksmp->sgl[0].sge32[0];
5754                 response_xferlen        = sge32[0].length;
5755                 request_xferlen         = sge32[1].length;
5756                 con_log(CL_ANN, (CE_CONT, "issue_mfi_smp: "
5757                     "response_xferlen = %x, request_xferlen = %x",
5758                     response_xferlen, request_xferlen));

5760                 response_ubuf   = (void *)(ulong_t)sge32[0].phys_addr;
5761                 request_ubuf    = (void *)(ulong_t)sge32[1].phys_addr;
5762                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: "
5763                     "response_ubuf = %p, request_ubuf = %p",
5764                     response_ubuf, request_ubuf));
5765         } else {
```



```
5766 #ifdef _ILP32
5767                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_ILP32"));

5769                 sge32                   = &ksmp->sgl[0].sge32[0];
5770                 response_xferlen        = sge32[0].length;
5771                 request_xferlen         = sge32[1].length;
5772                 con_log(CL_ANN, (CE_CONT, "issue_mfi_smp: "
5773                     "response_xferlen = %x, request_xferlen = %x",
5774                     response_xferlen, request_xferlen));

5776                 response_ubuf   = (void *)(ulong_t)sge32[0].phys_addr;
5777                 request_ubuf    = (void *)(ulong_t)sge32[1].phys_addr;
5778                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: "
5779                     "response_ubuf = %p, request_ubuf = %p",
5780                     response_ubuf, request_ubuf));
5781 #else
5782                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_LP64"));

5784                 sge64                   = &ksmp->sgl[0].sge64[0];
5785                 response_xferlen        = sge64[0].length;
5786                 request_xferlen         = sge64[1].length;

5788                 response_ubuf   = (void *)(ulong_t)sge64[0].phys_addr;
5789                 request_ubuf    = (void *)(ulong_t)sge64[1].phys_addr;
5790 #endif
5791         }
5792         if (request_xferlen) {
5793                 /* means IOCTL requires DMA */
5794                 /* allocate the data transfer buffer */
5795                 /* request_dma_obj.size = request_xferlen; */
5796                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(request_xferlen,
5797                     new_xfer_length1, PAGESIZE);
5560                 //request_dma_obj.size = request_xferlen;
5561                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(request_xferlen,new_xfer_length1,
5798                 request_dma_obj.size = new_xfer_length1;
5799                 request_dma_obj.dma_attr = mrsas_generic_dma_attr;
5800                 request_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5801                 request_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5802                 request_dma_obj.dma_attr.dma_attr_sgllen = 1;
5803                 request_dma_obj.dma_attr.dma_attr_align = 1;

5805                 /* allocate kernel buffer for DMA */
5806                 if (mrsas_alloc_dma_obj(instance, &request_dma_obj,
5807                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5808                         con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5809                             "could not allocate data transfer buffer."));
5810                         return (DDI_FAILURE);
5811                 }
5812                 (void) memset(request_dma_obj.buffer, 0, request_xferlen);

5814                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5815                 for (i = 0; i < request_xferlen; i++) {
5816                         if (ddi_copyin((uint8_t *)request_ubuf + i,
5817                             (uint8_t *)request_dma_obj.buffer + i,
5818                             1, mode)) {
5819                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5820                                     "copy from user space failed"));
5821                                 return (DDI_FAILURE);
5822                         }
5823                 }
5824         }

5826         if (response_xferlen) {
5827                 /* means IOCTL requires DMA */
5828                 /* allocate the data transfer buffer */
5829                 /* response_dma_obj.size = response_xferlen; */
```

```
5830                MRSAS_GET_BOUNDARY_ALIGNED_LEN(response_xferlen,
5831                    new_xfer_length2, PAGESIZE);
5593                //response_dma_obj.size = response_xferlen;
5594                MRSAS_GET_BOUNDARY_ALIGNED_LEN(response_xferlen,new_xfer_length2
5832                response_dma_obj.size = new_xfer_length2;
5833                response_dma_obj.dma_attr = mrsas_generic_dma_attr;
5834                response_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5835                response_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5836                response_dma_obj.dma_attr.dma_attr_sgllen = 1;
5837                response_dma_obj.dma_attr.dma_attr_align = 1;

5839                /* allocate kernel buffer for DMA */
5840                if (mrsas_alloc_dma_obj(instance, &response_dma_obj,
5841                    (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5842                        con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5843                            "could not allocate data transfer buffer."));
5844                        return (DDI_FAILURE);
5845                }
5846                (void) memset(response_dma_obj.buffer, 0, response_xferlen);

5848                /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5849                for (i = 0; i < response_xferlen; i++) {
5850                        if (ddi_copyin((uint8_t *)response_ubuf + i,
5851                            (uint8_t *)response_dma_obj.buffer + i,
5852                            1, mode)) {
5853                                con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5854                                    "copy from user space failed"));
5855                                return (DDI_FAILURE);
5856                        }
5857                }
5858        }

5860        ddi_put8(acc_handle, &smp->cmd, ksmp->cmd);
5861        ddi_put8(acc_handle, &smp->cmd_status, 0);
5862        ddi_put8(acc_handle, &smp->connection_status, 0);
5863        ddi_put8(acc_handle, &smp->sge_count, ksmp->sge_count);
5864        /* smp->context         = ksmp->context; */
5865        ddi_put16(acc_handle, &smp->timeout, ksmp->timeout);
5866        ddi_put32(acc_handle, &smp->data_xfer_len, ksmp->data_xfer_len);

5868        bcopy((void *)&ksmp->sas_addr, (void *)&tmp_sas_addr,
5869            sizeof (uint64_t));
5870        ddi_put64(acc_handle, &smp->sas_addr, tmp_sas_addr);

5872        ddi_put16(acc_handle, &smp->flags, ksmp->flags & ~MFI_FRAME_SGL64);

5874        model = ddi_model_convert_from(mode & FMODELS);
5875        if (model == DDI_MODEL_ILP32) {
5876                con_log(CL_ANN1, (CE_CONT,
5877                    "issue_mfi_smp: DDI_MODEL_ILP32"));

5879                sge32 = &smp->sgl[0].sge32[0];
5880                ddi_put32(acc_handle, &sge32[0].length, response_xferlen);
5881                ddi_put32(acc_handle, &sge32[0].phys_addr,
5882                    response_dma_obj.dma_cookie[0].dmac_address);
5883                ddi_put32(acc_handle, &sge32[1].length, request_xferlen);
5884                ddi_put32(acc_handle, &sge32[1].phys_addr,
5885                    request_dma_obj.dma_cookie[0].dmac_address);
5886        } else {
5887 #ifdef _ILP32
5888                con_log(CL_ANN1, (CE_CONT,
5889                    "issue_mfi_smp: DDI_MODEL_ILP32"));
5890                sge32 = &smp->sgl[0].sge32[0];
5891                ddi_put32(acc_handle, &sge32[0].length, response_xferlen);
5892                ddi_put32(acc_handle, &sge32[0].phys_addr,
5893                    response_dma_obj.dma_cookie[0].dmac_address);
```



```
5894                ddi_put32(acc_handle, &sge32[1].length, request_xferlen);
5895                ddi_put32(acc_handle, &sge32[1].phys_addr,
5896                    request_dma_obj.dma_cookie[0].dmac_address);
5897 #else
5898                con_log(CL_ANN1, (CE_CONT,
5899                    "issue_mfi_smp: DDI_MODEL_LP64"));
5900                sge64 = &smp->sgl[0].sge64[0];
5901                ddi_put32(acc_handle, &sge64[0].length, response_xferlen);
5902                ddi_put64(acc_handle, &sge64[0].phys_addr,
5903                    response_dma_obj.dma_cookie[0].dmac_address);
5904                ddi_put32(acc_handle, &sge64[1].length, request_xferlen);
5905                ddi_put64(acc_handle, &sge64[1].phys_addr,
5906                    request_dma_obj.dma_cookie[0].dmac_address);
5907 #endif
5908        }
5909        con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp : "
5910            "smp->response_xferlen = %d, smp->request_xferlen = %d "
5911            "smp->data_xfer_len = %d", ddi_get32(acc_handle, &sge32[0].length),
5912            ddi_get32(acc_handle, &sge32[1].length),
5913            ddi_get32(acc_handle, &smp->data_xfer_len)));

5915        cmd->sync_cmd = MRSAS_TRUE;
5916        cmd->frame_count = 1;

5918        if (instance->tbolt) {
5919                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5920        }

5922        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5923                con_log(CL_ANN, (CE_WARN,
5924                    "issue_mfi_smp: fw_ioctl failed"));
5925        } else {
5926                con_log(CL_ANN1, (CE_CONT,
5927                    "issue_mfi_smp: copy to user space"));

5929                if (request_xferlen) {
5930                        for (i = 0; i < request_xferlen; i++) {
5931                                if (ddi_copyout(
5932                                    (uint8_t *)request_dma_obj.buffer +
5933                                    i, (uint8_t *)request_ubuf + i,
5934                                    1, mode)) {
5935                                        con_log(CL_ANN, (CE_WARN,
5936                                            "issue_mfi_smp : copy to user space"
5937                                            " failed"));
5938                                        return (DDI_FAILURE);
5939                                }
5940                        }
5941                }

5943                if (response_xferlen) {
5944                        for (i = 0; i < response_xferlen; i++) {
5945                                if (ddi_copyout(
5946                                    (uint8_t *)response_dma_obj.buffer
5947                                    + i, (uint8_t *)response_ubuf
5948                                    + i, 1, mode)) {
5949                                        con_log(CL_ANN, (CE_WARN,
5950                                            "issue_mfi_smp : copy to "
5951                                            "user space failed"));
5952                                        return (DDI_FAILURE);
5953                                }
5954                        }
5955                }
5956        }

5958        ksmp->cmd_status = ddi_get8(acc_handle, &smp->cmd_status);
5959        con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: smp->cmd_status = %d",
```

```
5960             ksmp->cmd_status));
5961         DTRACE_PROBE2(issue_smp, uint8_t, ksmp->cmd, uint8_t, ksmp->cmd_status);

5963         if (request_xferlen) {
5964                 /* free kernel buffer */
5965                 if (mrsas_free_dma_obj(instance, request_dma_obj) !=
5966                     DDI_SUCCESS)
5967                         return (DDI_FAILURE);
5968         }

5970         if (response_xferlen) {
5971                 /* free kernel buffer */
5972                 if (mrsas_free_dma_obj(instance, response_dma_obj) !=
5973                     DDI_SUCCESS)
5974                         return (DDI_FAILURE);
5975         }

5977         return (DDI_SUCCESS);
5978 }

5980 /*
5981  * issue_mfi_stp
5982  */
5983 static int
5984 issue_mfi_stp(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5985     struct mrsas_cmd *cmd, int mode)
5986 {
5987         void            *fis_ubuf;
5988         void            *data_ubuf;
5989         uint32_t        fis_xferlen = 0;
5990         uint32_t   new_xfer_length1 = 0;
5991         uint32_t   new_xfer_length2 = 0;
5992         uint32_t        data_xferlen = 0;
5993         uint_t          model;
5994         dma_obj_t       fis_dma_obj;
5995         dma_obj_t       data_dma_obj;
5996         struct mrsas_stp_frame  *kstp;
5997         struct mrsas_stp_frame  *stp;
5998         ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5999         int i;

6001         stp = &cmd->frame->stp;
6002         kstp = (struct mrsas_stp_frame *)&ioctl->frame[0];

6004         if (instance->adapterresetinprogress) {
6005                 con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
6006                 "returning mfi_pkt and setting TRAN_BUSY\n"));
6007                 return (DDI_FAILURE);
6008         }
6009         model = ddi_model_convert_from(mode & FMODELS);
6010         if (model == DDI_MODEL_ILP32) {
6011                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_ILP32"));

6013                 fis_xferlen     = kstp->sgl.sge32[0].length;
6014                 data_xferlen    = kstp->sgl.sge32[1].length;

6016                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge32[0].phys_addr;
6017                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge32[1].phys_addr;
6018         } else {
5780         }
5781         else
5782         {
6019 #ifdef _ILP32
6020                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_ILP32"));

6022                 fis_xferlen     = kstp->sgl.sge32[0].length;
```



```
6023                 data_xferlen    = kstp->sgl.sge32[1].length;

6025                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge32[0].phys_addr;
6026                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge32[1].phys_addr;
6027 #else
6028                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_LP64"));

6030                 fis_xferlen     = kstp->sgl.sge64[0].length;
6031                 data_xferlen    = kstp->sgl.sge64[1].length;

6033                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge64[0].phys_addr;
6034                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge64[1].phys_addr;
6035 #endif
6036         }


6039         if (fis_xferlen) {
6040                 con_log(CL_ANN, (CE_CONT, "issue_mfi_stp: "
6041                     "fis_ubuf = %p fis_xferlen = %x", fis_ubuf, fis_xferlen));

6043                 /* means IOCTL requires DMA */
6044                 /* allocate the data transfer buffer */
6045                 /* fis_dma_obj.size = fis_xferlen; */
6046                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(fis_xferlen,
6047                     new_xfer_length1, PAGESIZE);
5809                 //fis_dma_obj.size = fis_xferlen;
5810                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(fis_xferlen,new_xfer_length1,PAGE
6048                 fis_dma_obj.size = new_xfer_length1;
6049                 fis_dma_obj.dma_attr = mrsas_generic_dma_attr;
6050                 fis_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
6051                 fis_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
6052                 fis_dma_obj.dma_attr.dma_attr_sgllen = 1;
6053                 fis_dma_obj.dma_attr.dma_attr_align = 1;

6055                 /* allocate kernel buffer for DMA */
6056                 if (mrsas_alloc_dma_obj(instance, &fis_dma_obj,
6057                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
6058                         con_log(CL_ANN, (CE_WARN, "issue_mfi_stp : "
6059                             "could not allocate data transfer buffer."));
6060                         return (DDI_FAILURE);
6061                 }
6062                 (void) memset(fis_dma_obj.buffer, 0, fis_xferlen);

6064                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
6065                 for (i = 0; i < fis_xferlen; i++) {
6066                         if (ddi_copyin((uint8_t *)fis_ubuf + i,
6067                             (uint8_t *)fis_dma_obj.buffer + i, 1, mode)) {
6068                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6069                                     "copy from user space failed"));
6070                                 return (DDI_FAILURE);
6071                         }
6072                 }
6073         }

6075         if (data_xferlen) {
6076                 con_log(CL_ANN, (CE_CONT, "issue_mfi_stp: data_ubuf = %p "
6077                     "data_xferlen = %x", data_ubuf, data_xferlen));

6079                 /* means IOCTL requires DMA */
6080                 /* allocate the data transfer buffer */
6081                 /* data_dma_obj.size = data_xferlen; */
6082                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(data_xferlen, new_xfer_length2,
6083                     PAGESIZE);
5844                 //data_dma_obj.size = data_xferlen;
5845                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(data_xferlen,new_xfer_length2,PAG
6084                 data_dma_obj.size = new_xfer_length2;
```

```
6085                data_dma_obj.dma_attr = mrsas_generic_dma_attr;
6086                data_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
6087                data_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
6088                data_dma_obj.dma_attr.dma_attr_sgllen = 1;
6089                data_dma_obj.dma_attr.dma_attr_align = 1;

6091                /* allocate kernel buffer for DMA */
5853 /* allocate kernel buffer for DMA */
6092                if (mrsas_alloc_dma_obj(instance, &data_dma_obj,
6093                    (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
6094                        con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6095                            "could not allocate data transfer buffer."));
6096                        return (DDI_FAILURE);
6097                }
6098                (void) memset(data_dma_obj.buffer, 0, data_xferlen);

6100                /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
6101                for (i = 0; i < data_xferlen; i++) {
6102                        if (ddi_copyin((uint8_t *)data_ubuf + i,
6103                            (uint8_t *)data_dma_obj.buffer + i, 1, mode)) {
6104                                con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6105                                    "copy from user space failed"));
6106                                return (DDI_FAILURE);
6107                        }
6108                }
6109        }

6111        ddi_put8(acc_handle, &stp->cmd, kstp->cmd);
6112        ddi_put8(acc_handle, &stp->cmd_status, 0);
6113        ddi_put8(acc_handle, &stp->connection_status, 0);
6114        ddi_put8(acc_handle, &stp->target_id, kstp->target_id);
6115        ddi_put8(acc_handle, &stp->sge_count, kstp->sge_count);

6117        ddi_put16(acc_handle, &stp->timeout, kstp->timeout);
6118        ddi_put32(acc_handle, &stp->data_xfer_len, kstp->data_xfer_len);

6120        ddi_rep_put8(acc_handle, (uint8_t *)kstp->fis, (uint8_t *)stp->fis, 10,
6121            DDI_DEV_AUTOINCR);

6123        ddi_put16(acc_handle, &stp->flags, kstp->flags & ~MFI_FRAME_SGL64);
6124        ddi_put32(acc_handle, &stp->stp_flags, kstp->stp_flags);
6125        ddi_put32(acc_handle, &stp->sgl.sge32[0].length, fis_xferlen);
6126        ddi_put32(acc_handle, &stp->sgl.sge32[0].phys_addr,
6127            fis_dma_obj.dma_cookie[0].dmac_address);
6128        ddi_put32(acc_handle, &stp->sgl.sge32[1].length, data_xferlen);
6129        ddi_put32(acc_handle, &stp->sgl.sge32[1].phys_addr,
6130            data_dma_obj.dma_cookie[0].dmac_address);

6132        cmd->sync_cmd = MRSAS_TRUE;
6133        cmd->frame_count = 1;

6135        if (instance->tbolt) {
6136                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
6137        }

6139        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
6140                con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: fw_ioctl failed"));
6141        } else {

6143                if (fis_xferlen) {
6144                        for (i = 0; i < fis_xferlen; i++) {
6145                                if (ddi_copyout(
6146                                    (uint8_t *)fis_dma_obj.buffer + i,
6147                                    (uint8_t *)fis_ubuf + i, 1, mode)) {
6148                                        con_log(CL_ANN, (CE_WARN,
6149                                            "issue_mfi_stp : copy to "
```



```
6150                                            "user space failed"));
6151                                        return (DDI_FAILURE);
6152                                }
6153                        }
6154                }
6155        }
6156        if (data_xferlen) {
6157                for (i = 0; i < data_xferlen; i++) {
6158                        if (ddi_copyout(
6159                            (uint8_t *)data_dma_obj.buffer + i,
6160                            (uint8_t *)data_ubuf + i, 1, mode)) {
6161                                con_log(CL_ANN, (CE_WARN,
6162                                    "issue_mfi_stp : copy to"
6163                                    " user space failed"));
6164                                return (DDI_FAILURE);
6165                        }
6166                }
6167        }

6169        kstp->cmd_status = ddi_get8(acc_handle, &stp->cmd_status);
6170        con_log(CL_ANN1, (CE_NOTE, "issue_mfi_stp: stp->cmd_status = %d",
6171            kstp->cmd_status));
6172        DTRACE_PROBE2(issue_stp, uint8_t, kstp->cmd, uint8_t, kstp->cmd_status);

6174        if (fis_xferlen) {
6175                /* free kernel buffer */
6176                if (mrsas_free_dma_obj(instance, fis_dma_obj) != DDI_SUCCESS)
6177                        return (DDI_FAILURE);
6178        }

6180        if (data_xferlen) {
6181                /* free kernel buffer */
6182                if (mrsas_free_dma_obj(instance, data_dma_obj) != DDI_SUCCESS)
6183                        return (DDI_FAILURE);
6184        }

6186        return (DDI_SUCCESS);
6187 }
_____unchanged_portion_omitted_

6318 /*
6319  * handle_mfi_ioctl
6320  */
6321 static int
6322 handle_mfi_ioctl(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
6323     int mode)
6324 {
6325        int     rval = DDI_SUCCESS;

6327        struct mrsas_header     *hdr;
6328        struct mrsas_cmd        *cmd;

6330        if (instance->tbolt) {
6331                cmd = get_raid_msg_mfi_pkt(instance);
6332        } else {
6333                cmd = get_mfi_pkt(instance);
6334        }
6335        if (!cmd) {
6336                con_log(CL_ANN, (CE_WARN, "mr_sas: "
6337                    "failed to get a cmd packet"));
6338                DTRACE_PROBE2(mfi_ioctl_err, uint16_t,
6339                    instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
6097                cmn_err(CE_WARN,
6098                    "Failed to get a cmd from free-pool in handle_mfi_ioctl(). "
6099                        "fw_outstanding=0x%X max_fw_cmds=0x%X",
6100                        instance->fw_outstanding, instance->max_fw_cmds);
```

```
6340                    return (DDI_FAILURE);
6341            }

6343            /* Clear the frame buffer and assign back the context id */
6344            (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6345            ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6346                cmd->index);

6348            hdr = (struct mrsas_header *)&ioctl->frame[0];

6350            switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
6351            case MFI_CMD_OP_DCMD:
6352                    rval = issue_mfi_dcmd(instance, ioctl, cmd, mode);
6353                    break;
6354            case MFI_CMD_OP_SMP:
6355                    rval = issue_mfi_smp(instance, ioctl, cmd, mode);
6356                    break;
6357            case MFI_CMD_OP_STP:
6358                    rval = issue_mfi_stp(instance, ioctl, cmd, mode);
6359                    break;
6360            case MFI_CMD_OP_LD_SCSI:
6361            case MFI_CMD_OP_PD_SCSI:
6362                    rval = issue_mfi_pthru(instance, ioctl, cmd, mode);
6363                    break;
6364            default:
6365                    con_log(CL_ANN, (CE_WARN, "handle_mfi_ioctl: "
6366                        "invalid mfi ioctl hdr->cmd = %d", hdr->cmd));
6367                    rval = DDI_FAILURE;
6368                    break;
6369            }

6371            if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
6372                    rval = DDI_FAILURE;

6374            if (instance->tbolt) {
6375                    return_raid_msg_mfi_pkt(instance, cmd);
6376            } else {
6377                    return_mfi_pkt(instance, cmd);
6378            }

6380            return (rval);
6381    }
_____unchanged_portion_omitted_

6399    static int
6400    register_mfi_aen(struct mrsas_instance *instance, uint32_t seq_num,
6401        uint32_t class_locale_word)
6402    {
6403            int     ret_val;

6405            struct mrsas_cmd        *cmd, *aen_cmd;
6406            struct mrsas_dcmd_frame *dcmd;
6407            union mrsas_evt_class_locale    curr_aen;
6408            union mrsas_evt_class_locale    prev_aen;

6410            con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
6411            /*
6412             * If there an AEN pending already (aen_cmd), check if the
6413             * class_locale of that pending AEN is inclusive of the new
6414             * AEN request we currently have. If it is, then we don't have
6415             * to do anything. In other words, whichever events the current
6416             * AEN request is subscribing to, have already been subscribed
6417             * to.
6418             *
6419             * If the old_cmd is _not_ inclusive, then we have to abort
6420             * that command, form a class_locale that is superset of both
```



```
6421             * old and current and re-issue to the FW
6422             */

6424            curr_aen.word = LE_32(class_locale_word);
6425            curr_aen.members.locale = LE_16(curr_aen.members.locale);
6426            aen_cmd = instance->aen_cmd;
6427            if (aen_cmd) {
6428                    prev_aen.word = ddi_get32(aen_cmd->frame_dma_obj.acc_handle,
6429                        &aen_cmd->frame->dcmd.mbox.w[1]);
6430                    prev_aen.word = LE_32(prev_aen.word);
6431                    prev_aen.members.locale = LE_16(prev_aen.members.locale);
6432                    /*
6433                     * A class whose enum value is smaller is inclusive of all
6434                     * higher values. If a PROGRESS (= -1) was previously
6435                     * registered, then a new registration requests for higher
6436                     * classes need not be sent to FW. They are automatically
6437                     * included.
6438                     *
6439                     * Locale numbers don't have such hierarchy. They are bitmap
6440                     * values
6441                     */
6442                    if ((prev_aen.members.class <= curr_aen.members.class) &&
6443                        !((prev_aen.members.locale & curr_aen.members.locale) ^
6444                        curr_aen.members.locale)) {
6445                            /*
6446                             * Previously issued event registration includes
6447                             * current request. Nothing to do.
6448                             */

6450                            return (0);
6451                    } else {
6452                            curr_aen.members.locale |= prev_aen.members.locale;

6454                            if (prev_aen.members.class < curr_aen.members.class)
6455                                    curr_aen.members.class = prev_aen.members.class;

6457                            ret_val = abort_aen_cmd(instance, aen_cmd);

6459                            if (ret_val) {
6460                                    con_log(CL_ANN, (CE_WARN, "register_mfi_aen: "
6461                                        "failed to abort prevous AEN command"));

6463                                    return (ret_val);
6464                            }
6465                    }
6466            } else {
6467                    curr_aen.word = LE_32(class_locale_word);
6468                    curr_aen.members.locale = LE_16(curr_aen.members.locale);
6469            }

6471            if (instance->tbolt) {
6472                    cmd = get_raid_msg_mfi_pkt(instance);
6473            } else {
6474                    cmd = get_mfi_pkt(instance);
6475            }

6477            if (!cmd) {
6478                    DTRACE_PROBE2(mfi_aen_err, uint16_t, instance->fw_outstanding,
6479                        uint16_t, instance->max_fw_cmds);
6236                    cmn_err(CE_WARN,
6237                        "Failed to get a cmd from free-pool in register_mfi_aen(). "
6238                            "fw_outstanding=0x%X max_fw_cmds=0x%X",
6239                            instance->fw_outstanding, instance->max_fw_cmds);
6480                    return (ENOMEM);
6481            }
```

```
6483         /* Clear the frame buffer and assign back the context id */
6484         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6485         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6486             cmd->index);

6488         dcmd = &cmd->frame->dcmd;

6490         /* for(i = 0; i < DCMD_MBOX_SZ; i++) dcmd->mbox.b[i] = 0; */
6491         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

6493         (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
6494             sizeof (struct mrsas_evt_detail));

6496         /* Prepare DCMD for aen registration */
6497         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
6498         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
6499         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
6500         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
6501             MFI_FRAME_DIR_READ);
6502         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
6503         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
6504             sizeof (struct mrsas_evt_detail));
6505         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
6506             MR_DCMD_CTRL_EVENT_WAIT);
6507         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], seq_num);
6508         curr_aen.members.locale = LE_16(curr_aen.members.locale);
6509         curr_aen.word = LE_32(curr_aen.word);
6510         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[1],
6511             curr_aen.word);
6512         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
6513             instance->mfi_evt_detail_obj.dma_cookie[0].dmac_address);
6514         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
6515             sizeof (struct mrsas_evt_detail));

6517         instance->aen_seq_num = seq_num;


6520         /*
6521          * Store reference to the cmd used to register for AEN. When an
6522          * application wants us to register for AEN, we have to abort this
6523          * cmd and re-register with a new EVENT LOCALE supplied by that app
6524          */
6525         instance->aen_cmd = cmd;

6527         cmd->frame_count = 1;

6529         /* Issue the aen registration frame */
6530         /* atomic_add_16 (&instance->fw_outstanding, 1); */
6531         if (instance->tbolt) {
6532                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
6533         }
6534         instance->func_ptr->issue_cmd(cmd, instance);

6536         return (0);
6537 }
```

*_____unchanged_portion_omitted_*

```
6608 static void
6369 void
6609 io_timeout_checker(void *arg)
6610 {
6611         struct scsi_pkt *pkt;
6612         struct mrsas_instance *instance = arg;
6613         struct mrsas_cmd        *cmd = NULL;
6614         struct mrsas_header     *hdr;
```



```
6615         int time = 0;
6616         int counter = 0;
6617         struct mlist_head       *pos, *next;
6618         mlist_t                 process_list;

6620         if (instance->adapterresetinprogress == 1) {
6621                 con_log(CL_ANN, (CE_NOTE, "io_timeout_checker:"
6622                     " reset in progress"));

6624                 instance->timeout_id = timeout(io_timeout_checker,
6625                     (void *) instance, drv_usectohz(MRSAS_1_SECOND));
6626                 return;
6627         }

6629         /* See if this check needs to be in the beginning or last in ISR */
6630         if (mrsas_initiate_ocr_if_fw_is_faulty(instance) ==  1) {
6631                 cmn_err(CE_WARN, "io_timeout_checker: "
6392                 cmn_err(CE_WARN, "io_timeout_checker:"
6632                     "FW Fault, calling reset adapter");
6633                 cmn_err(CE_CONT, "io_timeout_checker: "
6634                     "fw_outstanding 0x%X max_fw_cmds 0x%X",
6635                     instance->fw_outstanding, instance->max_fw_cmds);
6394                 cmn_err(CE_CONT, "io_timeout_checker: fw_outstanding 0x%X max_fw
6395                         instance->fw_outstanding, instance->max_fw_cmds );
6636                 if (instance->adapterresetinprogress == 0) {
6637                         instance->adapterresetinprogress = 1;
6638                         if (instance->tbolt)
6639                                 (void) mrsas_tbolt_reset_ppc(instance);
6399                                 mrsas_tbolt_reset_ppc(instance);
6640                         else
6641                                 (void) mrsas_reset_ppc(instance);
6401                                 mrsas_reset_ppc(instance);
6642                         instance->adapterresetinprogress = 0;
6643                 }
6644                 instance->timeout_id = timeout(io_timeout_checker,
6645                     (void *) instance, drv_usectohz(MRSAS_1_SECOND));
6646                 return;
6647         }

6649         INIT_LIST_HEAD(&process_list);

6651         mutex_enter(&instance->cmd_pend_mtx);
6652         mlist_for_each_safe(pos, next, &instance->cmd_pend_list) {
6653                 cmd = mlist_entry(pos, struct mrsas_cmd, list);

6655                 if (cmd == NULL) {
6656                         continue;
6657                 }

6659                 if (cmd->sync_cmd == MRSAS_TRUE) {
6660                         hdr = (struct mrsas_header *)&cmd->frame->hdr;
6661                         if (hdr == NULL) {
6662                                 continue;
6663                         }
6664                         time = --cmd->drv_pkt_time;
6665                 } else {
6666                         pkt = cmd->pkt;
6667                         if (pkt == NULL) {
6668                                 continue;
6669                         }
6670                         time = --cmd->drv_pkt_time;
6671                 }
6672                 if (time <= 0) {
6673                         cmn_err(CE_WARN, "%llx: "
6674                             "io_timeout_checker: TIMING OUT: pkt: %p, "
6675                             "cmd %p fw_outstanding 0x%X max_fw_cmds 0x%X\n",
```

```
                            gethrtime(), (void *)pkt, (void *)cmd,
                            instance->fw_outstanding, instance->max_fw_cmds);
                            "io_timeout_checker: TIMING OUT: pkt "
                            ": %p, cmd %p fw_outstanding 0x%X max_fw_cmds 0x%X\n
                                gethrtime(), (void *)pkt, (void *)cmd, instance-

                        counter++;
                        break;
                }
        }
        mutex_exit(&instance->cmd_pend_mtx);

        if (counter) {
                if (instance->disable_online_ctrl_reset == 1) {
                        cmn_err(CE_WARN, "mr_sas %d: %s(): OCR is NOT "
                            "supported by Firmware, KILL adapter!!!",
                        cmn_err(CE_WARN, "mr_sas %d: %s(): OCR is NOT supported
                            instance->instance, __func__);

                        if (instance->tbolt)
                                mrsas_tbolt_kill_adapter(instance);
                                (void) mrsas_tbolt_kill_adapter(instance);
                        else
                                (void) mrsas_kill_adapter(instance);

                        return;
                } else {
                        if (cmd->retry_count_for_ocr <=  IO_RETRY_COUNT) {
                                if (instance->adapterresetinprogress == 0) {
                                        if (instance->tbolt) {
                                                (void) mrsas_tbolt_reset_ppc(
                                                    instance);
                                        } else {
                                                (void) mrsas_reset_ppc(
                                                    instance);
                                        if (instance->tbolt)
                                                mrsas_tbolt_reset_ppc(instance);
                                        else
                                                mrsas_reset_ppc(instance);
                                        }
                                }
                        } else {
                                cmn_err(CE_WARN,
                                    "io_timeout_checker: "
                                    "cmd %p cmd->index %d "
                                    "timed out even after 3 resets: "
                                    "so KILL adapter", (void *)cmd, cmd->index);

                                mrsas_print_cmd_details(instance, cmd, 0xDD);

                                if (instance->tbolt)
                                        mrsas_tbolt_kill_adapter(instance);
                                        (void) mrsas_tbolt_kill_adapter(instance
                                else
                                        (void) mrsas_kill_adapter(instance);
                                return;
                        }
                }
        }
        con_log(CL_ANN, (CE_NOTE, "mrsas: "
            "schedule next timeout check: "
            "do timeout \n"));
        instance->timeout_id =
            timeout(io_timeout_checker, (void *)instance,
            drv_usectohz(MRSAS_1_SECOND));
}
```

_____unchanged_portion_omitted_


```
static void
issue_cmd_ppc(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
{
        struct scsi_pkt *pkt;
        atomic_add_16(&instance->fw_outstanding, 1);

        pkt = cmd->pkt;
        if (pkt) {
                con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
                    "ISSUED CMD TO FW : called : cmd:"
                    ": %p instance : %p pkt : %p pkt_time : %x\n",
                    gethrtime(), (void *)cmd, (void *)instance,
                    (void *)pkt, cmd->drv_pkt_time));
                if (instance->adapterresetinprogress) {
                        cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
                        con_log(CL_ANN1, (CE_NOTE, "Reset the scsi_pkt timer"));
                } else {
                        push_pending_mfi_pkt(instance, cmd);
                }

        } else {
                con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
                    "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
                    "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
        }

        mutex_enter(&instance->reg_write_mtx);
        ASSERT(mutex_owned(&instance->reg_write_mtx));
        /* Issue the command to the FW */
        WR_IB_QPORT((cmd->frame_phys_addr) |
            (((cmd->frame_count - 1) << 1) | 1), instance);
        mutex_exit(&instance->reg_write_mtx);

}

/*
 * issue_cmd_in_sync_mode
 */
static int
issue_cmd_in_sync_mode_ppc(struct mrsas_instance *instance,
struct mrsas_cmd *cmd)
{
        int     i;
        uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * (10 * MILLISEC);
        struct mrsas_header *hdr = &cmd->frame->hdr;

        con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: called"));

        if (instance->adapterresetinprogress) {
                cmd->drv_pkt_time = ddi_get16(
                    cmd->frame_dma_obj.acc_handle, &hdr->timeout);
                if (cmd->drv_pkt_time < debug_timeout_g)
                        cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

                con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: "
                    "issue and return in reset case\n"));
                WR_IB_QPORT((cmd->frame_phys_addr) |
                    (((cmd->frame_count - 1) << 1) | 1), instance);

                return (DDI_SUCCESS);
        } else {
                con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: pushing the pkt\n"));
                push_pending_mfi_pkt(instance, cmd);
        }
```

```
6803         cmd->cmd_status = ENODATA;

6805         mutex_enter(&instance->reg_write_mtx);
6562         ASSERT(mutex_owned(&instance->reg_write_mtx));
6806         /* Issue the command to the FW */
6807         WR_IB_QPORT((cmd->frame_phys_addr) |
6808             (((cmd->frame_count - 1) << 1) | 1), instance);
6809         mutex_exit(&instance->reg_write_mtx);

6811         mutex_enter(&instance->int_cmd_mtx);
6812         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
6813                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
6814         }
6815         mutex_exit(&instance->int_cmd_mtx);

6817         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: done"));

6819         if (i < (msecs -1)) {
6820                 return (DDI_SUCCESS);
6821         } else {
6822                 return (DDI_FAILURE);
6823         }
6824 }
```

*unchanged_portion_omitted_*

```
6912 static int
6913 intr_ack_ppc(struct mrsas_instance *instance)
6914 {
6915         uint32_t        status;
6916         int ret = DDI_INTR_CLAIMED;

6918         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: called"));

6920         /* check if it is our interrupt */
6921         status = RD_OB_INTR_STATUS(instance);

6923         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: status = 0x%x", status));

6925         if (!(status & MFI_REPLY_2108_MESSAGE_INTR)) {
6926                 ret = DDI_INTR_UNCLAIMED;
6927         }

6929         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
6930                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
6931                 ret = DDI_INTR_UNCLAIMED;
6932         }

6934         if (ret == DDI_INTR_UNCLAIMED) {
6935                 return (ret);
6936         }
6937         /* clear the interrupt by writing back the same value */
6938         WR_OB_DOORBELL_CLEAR(status, instance);

6940         /* dummy READ */
6941         status = RD_OB_INTR_STATUS(instance);

6943         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: interrupt cleared"));

6945         return (ret);
6946 }
```

*unchanged_portion_omitted_*

```
6972 static int
6973 mrsas_reset_ppc(struct mrsas_instance *instance)
6974 {
```



```
6975         uint32_t status;
6976         uint32_t retry = 0;
6977         uint32_t cur_abs_reg_val;
6978         uint32_t fw_state;

6980         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

6982         if (instance->deadadapter == 1) {
6983                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
6984                     "no more resets as HBA has been marked dead ");
6985                 return (DDI_FAILURE);
6986         }
6987         mutex_enter(&instance->ocr_flags_mtx);
6988         instance->adapterresetinprogress = 1;
6989         mutex_exit(&instance->ocr_flags_mtx);
6990         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: adpterresetinprogress "
6991             "flag set, time %llx", gethrtime()));

6993         instance->func_ptr->disable_intr(instance);
6994 retry_reset:
6995         WR_IB_WRITE_SEQ(0, instance);
6996         WR_IB_WRITE_SEQ(4, instance);
6997         WR_IB_WRITE_SEQ(0xb, instance);
6998         WR_IB_WRITE_SEQ(2, instance);
6999         WR_IB_WRITE_SEQ(7, instance);
7000         WR_IB_WRITE_SEQ(0xd, instance);
7001         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: magic number written "
7002             "to write sequence register\n"));
7003         delay(100 * drv_usectohz(MILLISEC));
7004         status = RD_OB_DRWE(instance);

7006         while (!(status & DIAG_WRITE_ENABLE)) {
7007                 delay(100 * drv_usectohz(MILLISEC));
7008                 status = RD_OB_DRWE(instance);
7009                 if (retry++ == 100) {
7010                         cmn_err(CE_WARN, "mrsas_reset_ppc: DRWE bit "
7011                             "check retry count %d", retry);
6763                         "check retry count %d\n", retry);
7012                         return (DDI_FAILURE);
7013                 }
7014         }
7015         WR_IB_DRWE(status | DIAG_RESET_ADAPTER, instance);
7016         delay(100 * drv_usectohz(MILLISEC));
7017         status = RD_OB_DRWE(instance);
7018         while (status & DIAG_RESET_ADAPTER) {
7019                 delay(100 * drv_usectohz(MILLISEC));
7020                 status = RD_OB_DRWE(instance);
7021                 if (retry++ == 100) {
7022                         cmn_err(CE_WARN, "mrsas_reset_ppc: "
7023                             "RESET FAILED. KILL adapter called.");
6774                         cmn_err(CE_WARN,
6775                             "mrsas_reset_ppc: RESET FAILED. KILL adapter cal

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

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

```
7038                 fw_state        = cur_abs_reg_val & MFI_STATE_MASK;

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

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

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

7062                         } else {
7063                                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
7064                                     "Max Reset Count exceeded >%d"
7065                                     "Mark HBA as bad, KILL adapter",
7066                                     MAX_FW_RESET_COUNT);
6817                                 "Mark HBA as bad, KILL adapter", MAX_FW_RESE

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


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

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

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

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

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

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



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

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


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

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

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

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

7129 /*
7130  * FMA functions.
7131  */
7132 int
7133 mrsas_common_check(struct mrsas_instance *instance, struct  mrsas_cmd *cmd)
7134 {
7135         int ret = DDI_SUCCESS;

7137         if (cmd != NULL &&
7138             mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
7139             DDI_SUCCESS) {
7140                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7141                 if (cmd->pkt != NULL) {
7142                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7143                         cmd->pkt->pkt_statistics = 0;
7144                 }
7145                 ret = DDI_FAILURE;
7146         }
7147         if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
7148             != DDI_SUCCESS) {
7149                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7150                 if (cmd != NULL && cmd->pkt != NULL) {
7151                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7152                         cmd->pkt->pkt_statistics = 0;
7153                 }
7154                 ret = DDI_FAILURE;
7155         }
7156         if (mrsas_check_dma_handle(instance->mfi_evt_detail_obj.dma_handle) !=
7157             DDI_SUCCESS) {
7158                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7159                 if (cmd != NULL && cmd->pkt != NULL) {
7160                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7161                         cmd->pkt->pkt_statistics = 0;
7162                 }
7163                 ret = DDI_FAILURE;
7164         }
7165         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
7166                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);

7168                 ddi_fm_acc_err_clear(instance->regmap_handle, DDI_FME_VER0);
```

```
7170                 if (cmd != NULL && cmd->pkt != NULL) {
7171                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7172                         cmd->pkt->pkt_statistics = 0;
7173                 }
7174                 ret = DDI_FAILURE;
7175         }

7177         return (ret);
7178 }

7180 /*ARGSUSED*/
7181 static int
7182 mrsas_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
7183 {
7184         /*
7185          * as the driver can always deal with an error in any dma or
7186          * access handle, we can just return the fme_status value.
7187          */
7188         pci_ereport_post(dip, err, NULL);
7189         return (err->fme_status);
7190 }

7192 static void
7193 mrsas_fm_init(struct mrsas_instance *instance)
7194 {
7195         /* Need to change iblock to priority for new MSI intr */
7196         ddi_iblock_cookie_t fm_ibc;

7198         /* Only register with IO Fault Services if we have some capability */
7199         if (instance->fm_capabilities) {
7200                 /* Adjust access and dma attributes for FMA */
7201                 endian_attr.devacc_attr_access = DDI_FLAGERR_ACC;
7202                 mrsas_generic_dma_attr.dma_attr_flags = DDI_DMA_FLAGERR;

7204                 /*
7205                  * Register capabilities with IO Fault Services.
7206                  * fm_capabilities will be updated to indicate
7207                  * capabilities actually supported (not requested.)
7208                  */

7210                 ddi_fm_init(instance->dip, &instance->fm_capabilities, &fm_ibc);

7212                 /*
7213                  * Initialize pci ereport capabilities if ereport
7214                  * capable (should always be.)
7215                  */

7217                 if (DDI_FM_EREPORT_CAP(instance->fm_capabilities) ||
7218                     DDI_FM_ERRCB_CAP(instance->fm_capabilities)) {
7219                         pci_ereport_setup(instance->dip);
7220                 }

7222                 /*
7223                  * Register error callback if error callback capable.
7224                  */
7225                 if (DDI_FM_ERRCB_CAP(instance->fm_capabilities)) {
7226                         ddi_fm_handler_register(instance->dip,
7227                             mrsas_fm_error_cb, (void*) instance);
7228                 }
7229         } else {
7230                 endian_attr.devacc_attr_access = DDI_DEFAULT_ACC;
7231                 mrsas_generic_dma_attr.dma_attr_flags = 0;
7232         }
7233 }
```



```
7235 static void
7236 mrsas_fm_fini(struct mrsas_instance *instance)
7237 {
7238         /* Only unregister FMA capabilities if registered */
7239         if (instance->fm_capabilities) {
7240                 /*
7241                  * Un-register error callback if error callback capable.
7242                  */
7243                 if (DDI_FM_ERRCB_CAP(instance->fm_capabilities)) {
7244                         ddi_fm_handler_unregister(instance->dip);
7245                 }

7247                 /*
7248                  * Release any resources allocated by pci_ereport_setup()
7249                  */
7250                 if (DDI_FM_EREPORT_CAP(instance->fm_capabilities) ||
7251                     DDI_FM_ERRCB_CAP(instance->fm_capabilities)) {
7252                         pci_ereport_teardown(instance->dip);
7253                 }

7255                 /* Unregister from IO Fault Services */
7256                 ddi_fm_fini(instance->dip);

7258                 /* Adjust access and dma attributes for FMA */
7259                 endian_attr.devacc_attr_access = DDI_DEFAULT_ACC;
7260                 mrsas_generic_dma_attr.dma_attr_flags = 0;
7261         }
7262 }

7264 int
7265 mrsas_check_acc_handle(ddi_acc_handle_t handle)
7266 {
7267         ddi_fm_error_t de;

7269         if (handle == NULL) {
7270                 return (DDI_FAILURE);
7271         }

7273         ddi_fm_acc_err_get(handle, &de, DDI_FME_VERSION);

7275         return (de.fme_status);
7276 }

7278 int
7279 mrsas_check_dma_handle(ddi_dma_handle_t handle)
7280 {
7281         ddi_fm_error_t de;

7283         if (handle == NULL) {
7284                 return (DDI_FAILURE);
7285         }

7287         ddi_fm_dma_err_get(handle, &de, DDI_FME_VERSION);

7289         return (de.fme_status);
7290 }

7292 void
7293 mrsas_fm_ereport(struct mrsas_instance *instance, char *detail)
7294 {
7295         uint64_t ena;
7296         char buf[FM_MAX_CLASS];

7298         (void) snprintf(buf, FM_MAX_CLASS, "%s.%s", DDI_FM_DEVICE, detail);
7299         ena = fm_ena_generate(0, FM_ENA_FMT1);
7300         if (DDI_FM_EREPORT_CAP(instance->fm_capabilities)) {
```

```
7301                 ddi_fm_ereport_post(instance->dip, buf, ena, DDI_NOSLEEP,
7302                     FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERSION, NULL);
7303         }
7304 }

7306 static int
7307 mrsas_add_intrs(struct mrsas_instance *instance, int intr_type)
7308 {

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

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

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

7323                 return (DDI_FAILURE);
7324         }

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

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

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

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

7343         /*
7344          * Allocate an array of interrupt handlers. Currently we support
7345          * only one interrupt. The framework can be extended later.
7346          */
7347         instance->intr_htable_size = count * sizeof (ddi_intr_handle_t);
7348         instance->intr_htable = kmem_zalloc(instance->intr_htable_size,
7349             KM_SLEEP);
7350         ASSERT(instance->intr_htable);
6923         instance->intr_htable = kmem_zalloc(instance->intr_htable_size, KM_SLEEP
6924         if (instance->intr_htable == NULL) {
6925                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
6926                     "failed to allocate memory for intr-handle table"));
6927                 instance->intr_htable_size = 0;
6928                 return (DDI_FAILURE);
6929         }

7352         flag = ((intr_type == DDI_INTR_TYPE_MSI) ||
7353             (intr_type == DDI_INTR_TYPE_MSIX)) ?
7354             DDI_INTR_ALLOC_STRICT : DDI_INTR_ALLOC_NORMAL;
6931         flag = ((intr_type == DDI_INTR_TYPE_MSI) || (intr_type ==
6932                 DDI_INTR_TYPE_MSIX)) ? DDI_INTR_ALLOC_STRICT:DDI_INTR_ALLOC_NORM

7356         /* Allocate interrupt */
7357         ret = ddi_intr_alloc(dip, instance->intr_htable, intr_type, 0,
```



```
7358             count, &actual, flag);

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

7366         if (actual < count) {
7367                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7368                     "Requested = %d  Received = %d", count, actual));
7369         }
7370         instance->intr_cnt = actual;

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

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

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

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

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

7406         }

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

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

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

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

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

7432         return (DDI_SUCCESS);

7434 mrsas_free_handlers:
7435         for (i = 0; i < actual; i++)
7014         {
7436                 (void) ddi_intr_remove_handler(instance->intr_htable[i]);
7016         }

7438 mrsas_free_handles:
7439         for (i = 0; i < actual; i++)
7020         {
7440                 (void) ddi_intr_free(instance->intr_htable[i]);
7022         }

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

7446         instance->intr_htable = NULL;
7028         instance->intr_htable =NULL;
7447         instance->intr_htable_size = 0;

7449         return (DDI_FAILURE);

7451 }


7454 static void
7455 mrsas_rem_intrs(struct mrsas_instance *instance)
7456 {
7457         int i;

7459         con_log(CL_ANN, (CE_NOTE, "mrsas_rem_intrs called"));

7461         /* Disable all interrupts first */
7462         if (instance->intr_cap & DDI_INTR_FLAG_BLOCK) {
7463                 (void) ddi_intr_block_disable(instance->intr_htable,
7464                     instance->intr_cnt);
7465         } else {
7466                 for (i = 0; i < instance->intr_cnt; i++) {
7467                         (void) ddi_intr_disable(instance->intr_htable[i]);
7468                 }
7469         }

7471         /* Remove all the handlers */

7473         for (i = 0; i < instance->intr_cnt; i++) {
7474                 (void) ddi_intr_remove_handler(instance->intr_htable[i]);
7475                 (void) ddi_intr_free(instance->intr_htable[i]);
7476         }

7478         if (instance->intr_htable != NULL)
7479                 kmem_free(instance->intr_htable, instance->intr_htable_size);

7481         instance->intr_htable = NULL;
7063         instance->intr_htable =NULL;
7482         instance->intr_htable_size = 0;
```



```
7484 }

7486 static int
7487 mrsas_tran_bus_config(dev_info_t *parent, uint_t flags,
7488     ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
7489 {
7490         struct mrsas_instance *instance;
7491         int config;
7492         int rval  = NDI_SUCCESS;

7494         char *ptr = NULL;
7495         int tgt, lun;

7497         con_log(CL_ANN1, (CE_NOTE, "Bus config called for op = %x", op));

7499         if ((instance = ddi_get_soft_state(mrsas_state,
7500             ddi_get_instance(parent))) == NULL) {
7501                 return (NDI_FAILURE);
7502         }

7504         /* Hold nexus during bus_config */
7505         ndi_devi_enter(parent, &config);
7506         switch (op) {
7507         case BUS_CONFIG_ONE: {

7509                 /* parse wwid/target name out of name given */
7510                 if ((ptr = strchr((char *)arg, '@')) == NULL) {
7511                         rval = NDI_FAILURE;
7512                         break;
7513                 }
7514                 ptr++;

7516                 if (mrsas_parse_devname(arg, &tgt, &lun) != 0) {
7517                         rval = NDI_FAILURE;
7518                         break;
7519                 }

7521                 if (lun == 0) {
7522                         rval = mrsas_config_ld(instance, tgt, lun, childp);
7105                 }
7523 #ifdef PDSUPPORT
7524                 } else if (instance->tbolt == 1 && lun != 0) {
7107                 else if ( instance->tbolt == 1 && lun != 0) {
7525                         rval = mrsas_tbolt_config_pd(instance,
7526                             tgt, lun, childp);
7110                 }
7527 #endif
7528                 } else {
7112                 else {
7529                         rval = NDI_FAILURE;
7530                 }

7532                 break;
7533         }
7534         case BUS_CONFIG_DRIVER:
7535         case BUS_CONFIG_ALL: {

7537                 rval = mrsas_config_all_devices(instance);

7539                 rval = NDI_SUCCESS;
7540                 break;
7541         }
7542         }

7544         if (rval == NDI_SUCCESS) {
7545                 rval = ndi_busop_bus_config(parent, flags, op, arg, childp, 0);
```

```
7547         }
7548         ndi_devi_exit(parent, config);

7550         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_bus_config: rval = %x",
7551             rval));
7552         return (rval);
7553 }

7555 static int
7556 mrsas_config_all_devices(struct mrsas_instance *instance)
7557 {
7558         int rval, tgt;

7560         for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
7561                 (void) mrsas_config_ld(instance, tgt, 0, NULL);

7563         }

7565 #ifdef PDSUPPORT
7566         /* Config PD devices connected to the card */
7567         if (instance->tbolt) {
7151         if(instance->tbolt) {
7568                 for (tgt = 0; tgt < instance->mr_tbolt_pd_max; tgt++) {
7569                         (void) mrsas_tbolt_config_pd(instance, tgt, 1, NULL);
7570                 }
7571         }
7572 #endif

7574         rval = NDI_SUCCESS;
7575         return (rval);
7576 }
```

*unchanged_portion_omitted_*

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

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

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

7648         sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
7233         if (sd == NULL) {
7234                 con_log(CL_ANN1, (CE_WARN,
7235                         "mrsas_config_ld: failed to allocate mem for scsi_de
7236                 return (NDI_FAILURE);
7237         }
```



```
7649         sd->sd_address.a_hba_tran = instance->tran;
7650         sd->sd_address.a_target = (uint16_t)tgt;
7651         sd->sd_address.a_lun = (uint8_t)lun;

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

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

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

*unchanged_portion_omitted_*

```
7752 /*ARGSUSED*/
7753 int
7754 mrsas_service_evt(struct mrsas_instance *instance, int tgt, int lun, int event,
7755     uint64_t wwn)
7756 {
7757         struct mrsas_eventinfo *mrevt = NULL;

7759         con_log(CL_ANN1, (CE_NOTE,
7760             "mrsas_service_evt called for t%dl%d event = %d",
7761             tgt, lun, event));

7763         if ((instance->taskq == NULL) || (mrevt =
7764             kmem_zalloc(sizeof (struct mrsas_eventinfo), KM_NOSLEEP)) == NULL) {
7765                 return (ENOMEM);
7766         }

7768         mrevt->instance = instance;
7769         mrevt->tgt = tgt;
7770         mrevt->lun = lun;
7771         mrevt->event = event;
7772         mrevt->wwn = wwn;

7774         if ((ddi_taskq_dispatch(instance->taskq,
7775             (void (*)(void *))mrsas_issue_evt_taskq, mrevt, DDI_NOSLEEP)) !=
7776             DDI_SUCCESS) {
7777                 con_log(CL_ANN1, (CE_NOTE,
7778                     "mr_sas: Event task failed for t%dl%d event = %d",
7779                     tgt, lun, event));
7780                 kmem_free(mrevt, sizeof (struct mrsas_eventinfo));
7781                 return (DDI_FAILURE);
7782         }
7783         DTRACE_PROBE3(service_evt, int, tgt, int, lun, int, event);

7784         return (DDI_SUCCESS);
7785 }

7787 static void
7788 mrsas_issue_evt_taskq(struct mrsas_eventinfo *mrevt)
7789 {
7790         struct mrsas_instance *instance = mrevt->instance;
7791         dev_info_t *dip, *pdip;
7792         int circ1 = 0;
7793         char *devname;
```

```
7795         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_evt_taskq: called for"
7796             " tgt %d lun %d event %d",
7797             mrevt->tgt, mrevt->lun, mrevt->event));

7799         if (mrevt->tgt < MRDRV_MAX_LD && mrevt->lun == 0) {
7800                 mutex_enter(&instance->config_dev_mtx);
7801                 dip = instance->mr_ld_list[mrevt->tgt].dip;
7802                 mutex_exit(&instance->config_dev_mtx);
7392         }

7803 #ifdef PDSUPPORT
7804         } else {
7395         else {
7805                 mutex_enter(&instance->config_dev_mtx);
7806                 dip = instance->mr_tbolt_pd_list[mrevt->tgt].dip;
7807                 mutex_exit(&instance->config_dev_mtx);
7808 #endif
7809         }
7400 #endif


7812         ndi_devi_enter(instance->dip, &circ1);
7813         switch (mrevt->event) {
7814         case MRSAS_EVT_CONFIG_TGT:
7815                 if (dip == NULL) {

7817                         if (mrevt->lun == 0) {
7818                                 (void) mrsas_config_ld(instance, mrevt->tgt,
7819                                     0, NULL);
7410                         }
7820 #ifdef PDSUPPORT
7821                         } else if (instance->tbolt) {
7412                         else if (instance->tbolt) {
7822                                 (void) mrsas_tbolt_config_pd(instance,
7823                                     mrevt->tgt,
7824                                     1, NULL);
7825 #endif
7826                         }
7417 #endif
7827                         con_log(CL_ANN1, (CE_NOTE,
7828                             "mr_sas: EVT_CONFIG_TGT called:"
7829                             " for tgt %d lun %d event %d",
7830                             mrevt->tgt, mrevt->lun, mrevt->event));

7832                 } else {
7833                         con_log(CL_ANN1, (CE_NOTE,
7834                             "mr_sas: EVT_CONFIG_TGT dip != NULL:"
7835                             " for tgt %d lun %d event %d",
7836                             mrevt->tgt, mrevt->lun, mrevt->event));
7837                 }
7838                 break;
7839         case MRSAS_EVT_UNCONFIG_TGT:
7840                 if (dip) {
7841                         if (i_ddi_devi_attached(dip)) {

7843                                 pdip = ddi_get_parent(dip);

7845                                 devname = kmem_zalloc(MAXNAMELEN + 1, KM_SLEEP);
7846                                 (void) ddi_deviname(dip, devname);

7848                                 (void) devfs_clean(pdip, devname + 1,
7849                                     DV_CLEAN_FORCE);
7850                                 kmem_free(devname, MAXNAMELEN + 1);
7851                         }
7852                         (void) ndi_devi_offline(dip, NDI_DEVI_REMOVE);
7853                         con_log(CL_ANN1, (CE_NOTE,
```



```
7854                             "mr_sas: EVT_UNCONFIG_TGT called:"
7855                             " for tgt %d lun %d event %d",
7856                             mrevt->tgt, mrevt->lun, mrevt->event));
7857                 } else {
7858                         con_log(CL_ANN1, (CE_NOTE,
7859                             "mr_sas: EVT_UNCONFIG_TGT dip == NULL:"
7860                             " for tgt %d lun %d event %d",
7861                             mrevt->tgt, mrevt->lun, mrevt->event));
7862                 }
7863                 break;
7864         }
7865         kmem_free(mrevt, sizeof (struct mrsas_eventinfo));
7866         ndi_devi_exit(instance->dip, circ1);
7867 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   55863 Tue Nov  6 14:29:40 2012
new/./mr_sas.h
**********************************************************
```

```
/*
 * mr_sas.h: header for mr_sas
 *
 * Solaris MegaRAID 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
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the author nor the names of its contributors may be
 *    used to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _MR_SAS_H_
#define _MR_SAS_H_

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/scsi/scsi.h>
#include "mr_sas_list.h"
#include "ld_pd_map.h"

/*
 * MegaRAID SAS2.0 Driver meta data
 */
#define MRSAS_VERSION                           "6.503.00.00ILLUMOS"
```



```
#define MRSAS_VERSION                           "6.503.00.00JOYENT"
#define MRSAS_RELDATE                           "July 30, 2012"

#define MRSAS_TRUE                              1
#define MRSAS_FALSE                             0

#define ADAPTER_RESET_NOT_REQUIRED              0
#define ADAPTER_RESET_REQUIRED                  1

#define PDSUPPORT       1

#define SWAP_BYTES(w)   ((((w)>>8)&0xFF) | (((w)&0xFF)<<8))
#define BIG_ENDIAN(d)   (SWAP_BYTES((d) >> 16) | (SWAP_BYTES(d) << 16))
/*
 * MegaRAID SAS2.0 device id conversion definitions.
 */
#define INST2LSIRDCTL(x)                ((x) << INST_MINOR_SHIFT)
#define MRSAS_GET_BOUNDARY_ALIGNED_LEN(len, new_len, boundary_len)  { \
        int rem; \
        rem = (len / boundary_len); \
        if ((rem * boundary_len) != len) { \
                new_len = len + ((rem + 1) * boundary_len - len); \
        } else { \
                new_len = len; \
        } \
}
} \


/*
 * MegaRAID SAS2.0 supported controllers
 */
#define PCI_DEVICE_ID_LSI_2108VDE               0x0078
#define PCI_DEVICE_ID_LSI_2108V                 0x0079
#define PCI_DEVICE_ID_LSI_TBOLT                 0x005b
#define PCI_DEVICE_ID_LSI_INVADER               0x005d

/*
 * Register Index for 2108 Controllers.
 */
#define REGISTER_SET_IO_2108                    (2)

#define MRSAS_MAX_SGE_CNT                       0x50
#define MRSAS_APP_RESERVED_CMDS                 32

#define MRSAS_IOCTL_DRIVER                      0x12341234
#define MRSAS_IOCTL_FIRMWARE                    0x12345678
#define MRSAS_IOCTL_AEN                         0x87654321

#define MRSAS_1_SECOND                          1000000

#ifdef PDSUPPORT

#define UNCONFIGURED_GOOD                       0x0
#define PD_SYSTEM                               0x40
#define MR_EVT_PD_STATE_CHANGE                  0x0072
#define MR_EVT_PD_REMOVED_EXT           0x00f8
#define MR_EVT_PD_INSERTED_EXT          0x00f7
#define MR_DCMD_PD_GET_INFO                     0x02020000
#define MRSAS_TBOLT_PD_LUN              1
#define MRSAS_TBOLT_PD_TGT_MAX  255
#define MRSAS_TBOLT_GET_PD_MAX(s)       ((s)->mr_tbolt_pd_max)

#endif

/* Raid Context Flags */
```

```
125 #define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT 0x4
126 #define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_MASK 0x30
127 typedef enum MR_RAID_FLAGS_IO_SUB_TYPE {
128         MR_RAID_FLAGS_IO_SUB_TYPE_NONE = 0,
129         MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD = 1
130 } MR_RAID_FLAGS_IO_SUB_TYPE;
      unchanged_portion_omitted_
453 #endif

455 typedef struct mrsas_instance {
456         uint32_t        *producer;
457         uint32_t        *consumer;

459         uint32_t        *reply_queue;
460         dma_obj_t       mfi_internal_dma_obj;
461         uint16_t        adapterresetinprogress;
462         uint16_t        deadadapter;
463         /* ThunderBolt (TB) specific */
464         dma_obj_t       mpi2_frame_pool_dma_obj;
465         dma_obj_t       request_desc_dma_obj;
466         dma_obj_t       reply_desc_dma_obj;
467         dma_obj_t       ld_map_obj[2];

469         uint8_t         init_id;
470         uint8_t         flag_ieee;
471         uint8_t         disable_online_ctrl_reset;
472         uint8_t         fw_fault_count_after_ocr;

474         uint16_t        max_num_sge;
475         uint16_t        max_fw_cmds;
476         uint32_t        max_sectors_per_req;

478         struct mrsas_cmd **cmd_list;

480         mlist_t         cmd_pool_list;
481         kmutex_t        cmd_pool_mtx;
482         kmutex_t        sync_map_mtx;

484         mlist_t         app_cmd_pool_list;
485         kmutex_t        app_cmd_pool_mtx;
486         mlist_t         cmd_app_pool_list;
487         kmutex_t        cmd_app_pool_mtx;


490         mlist_t         cmd_pend_list;
491         kmutex_t        cmd_pend_mtx;

493         dma_obj_t       mfi_evt_detail_obj;
494         struct mrsas_cmd *aen_cmd;

496         uint32_t        aen_seq_num;
497         uint32_t        aen_class_locale_word;

499         scsi_hba_tran_t         *tran;

501         kcondvar_t      int_cmd_cv;
502         kmutex_t        int_cmd_mtx;

504         kcondvar_t      aen_cmd_cv;
505         kmutex_t        aen_cmd_mtx;

507         kcondvar_t      abort_cmd_cv;
508         kmutex_t        abort_cmd_mtx;

510         kmutex_t        reg_write_mtx;
511         kmutex_t        chip_mtx;
```



```
513         dev_info_t              *dip;
514         ddi_acc_handle_t        pci_handle;

516         timeout_id_t    timeout_id;
517         uint32_t        unique_id;
518         uint16_t        fw_outstanding;
519         caddr_t         regmap;
520         ddi_acc_handle_t        regmap_handle;
521         uint8_t         isr_level;
522         ddi_iblock_cookie_t     iblock_cookie;
523         ddi_iblock_cookie_t     soft_iblock_cookie;
524         ddi_softintr_t          soft_intr_id;
525         uint8_t         softint_running;
526         uint8_t         tbolt_softint_running;
527         kmutex_t        completed_pool_mtx;
528         mlist_t         completed_pool_list;

530         caddr_t         internal_buf;
531         uint32_t        internal_buf_dmac_add;
532         uint32_t        internal_buf_size;

534         uint16_t        vendor_id;
535         uint16_t        device_id;
536         uint16_t        subsysvid;
537         uint16_t        subsysid;
538         int             instance;
539         int             baseaddress;
540         char            iocnode[16];

542         int             fm_capabilities;
543         /*
544          * Driver resources unroll flags.  The flag is set for resources that
545          * are needed to be free'd at detach() time.
546          */
513         /*Driver resources unroll flags.
514           The flag is set for resources that are needed to be free'd at detach()

547         struct _unroll {
548                 uint8_t softs;          /* The software state was allocated. */
549                 uint8_t regs;           /* Controller registers mapped. */
550                 uint8_t intr;           /* Interrupt handler added. */
551                 uint8_t reqs;           /* Request structs allocated. */
552                 uint8_t mutexs;         /* Mutex's allocated. */
553                 uint8_t taskq;          /* Task q's created. */
554                 uint8_t tran;           /* Tran struct allocated */
555                 uint8_t tranSetup;      /* Tran attached to the ddi. */
556                 uint8_t devctl;         /* Device nodes for cfgadm created. */
557                 uint8_t scsictl;        /* Device nodes for cfgadm created. */
558                 uint8_t ioctl;          /* Device nodes for ioctl's created. */
559                 uint8_t timer;          /* Timer started. */
560                 uint8_t aenPend;        /* AEN cmd pending f/w. */
561                 uint8_t mapUpdate_pend; /* LD MAP update cmd pending f/w. */
562                 uint8_t soft_isr;       /* Soft interrupt handler allocated. */
563                 uint8_t ldlist_buff;    /* Logical disk list allocated. */
564                 uint8_t pdlist_buff;    /* Physical disk list allocated. */
565                 uint8_t syncCmd;        /* Sync map command allocated. */
566                 uint8_t verBuff;        /* 2108 MFI buffer allocated. */
567                 uint8_t alloc_space_mfi;  /* Allocated space for 2108 MFI. */
568                 uint8_t alloc_space_mpi2; /* Allocated space for 2208 MPI2. */
517                 uint8_t softs;          // The software state was allocated.
518                 uint8_t regs;           // Controller registers mapped.
519                 uint8_t intr;           // Interrupt handler added.
520                 uint8_t reqs;           // Request structs allocated.
521                 uint8_t mutexs;         // Mutex's allocated.
522                 uint8_t taskq;          // Task q's created.
```

```
523                 uint8_t tran;           // Tran struct allocated
524                 uint8_t tranSetup;      // Tran attached to the ddi.
525                 uint8_t devctl;         // Device nodes for cfgadm created.
526                 uint8_t scsictl;        // Device nodes for cfgadm created.
527                 uint8_t ioctl;          // Device nodes for ioctl's created.
528                 uint8_t timer;          // Timer started.
529                 uint8_t aenPend;        // AEN cmd pending f/w.
530                 uint8_t mapUpdate_pend; // LD MAP update cmd pending f/w.
531                 uint8_t soft_isr;
532                 uint8_t ldlist_buff;
533                 uint8_t pdlist_buff;
534                 uint8_t syncCmd;
535                 uint8_t verBuff;
536                 uint8_t alloc_space_mfi;
537                 uint8_t alloc_space_mpi2;
569         } unroll;


572         /* function template pointer */
573         struct mrsas_function_template *func_ptr;


576         /* MSI interrupts specific */
577         ddi_intr_handle_t *intr_htable;         /* Interrupt handle array */
578         size_t          intr_htable_size;       /* Int. handle array size */
546         ddi_intr_handle_t *intr_htable;       //Interrupt handle array
547         size_t          intr_htable_size;     //Interrupt handle array size
579         int             intr_type;
580         int             intr_cnt;
581         uint_t          intr_pri;
582         int             intr_cap;

584         ddi_taskq_t     *taskq;
585         struct mrsas_ld *mr_ld_list;
586         kmutex_t        config_dev_mtx;
587         /* ThunderBolt (TB) specific */
588         ddi_softintr_t  tbolt_soft_intr_id;

590 #ifdef PDSUPPORT
591         uint32_t        mr_tbolt_pd_max;
592         struct mrsas_tbolt_pd *mr_tbolt_pd_list;
593 #endif

595         uint8_t         fast_path_io;

597         uint16_t        tbolt;
598         uint16_t        reply_read_index;
599         uint16_t        reply_size;             /* Single Reply struct size */
600         uint16_t        raid_io_msg_size;       /* Single message size */
568         uint16_t        reply_size;             // Single Reply structure size
569         uint16_t        raid_io_msg_size;       // Single message size
601         uint32_t        io_request_frames_phy;
602         uint8_t         *io_request_frames;
603         /* Virtual address of request desc frame pool */
604         MRSAS_REQUEST_DESCRIPTOR_UNION  *request_message_pool;
605         /* Physical address of request desc frame pool */
606         uint32_t        request_message_pool_phy;
607         /* Virtual address of reply Frame */
608         MPI2_REPLY_DESCRIPTORS_UNION    *reply_frame_pool;
609         /* Physical address of reply Frame */
610         uint32_t        reply_frame_pool_phy;
611         uint8_t         *reply_pool_limit;      /* Last reply frame address */
612         /* Physical address of Last reply frame */
613         uint32_t        reply_pool_limit_phy;
614         uint32_t        reply_q_depth;          /* Reply Queue Depth */
572         MRSAS_REQUEST_DESCRIPTOR_UNION  *request_message_pool;   // Virtual addr
```



```
573         uint32_t        request_message_pool_phy;               // Physical add
574         MPI2_REPLY_DESCRIPTORS_UNION    *reply_frame_pool;      // Virtual addr
575         uint32_t        reply_frame_pool_phy;                   // Physical add
576         uint8_t         *reply_pool_limit;                      // Last reply f
577         uint32_t        reply_pool_limit_phy;                   // Physical add
578         uint32_t        reply_q_depth;                          // Reply Queue
615         uint8_t         max_sge_in_main_msg;
616         uint8_t         max_sge_in_chain;
617         uint8_t         chain_offset_io_req;
618         uint8_t         chain_offset_mpt_msg;
619         MR_FW_RAID_MAP_ALL *ld_map[2];
620         uint32_t        ld_map_phy[2];
621         uint32_t        size_map_info;
622         uint64_t        map_id;
623         LD_LOAD_BALANCE_INFO load_balance_info[MAX_LOGICAL_DRIVES];
624         struct mrsas_cmd *map_update_cmd;
625         uint32_t        SyncRequired;
626         kmutex_t        ocr_flags_mtx;
627         dma_obj_t       drv_ver_dma_obj;
628 } mrsas_t;


631 /*
632  * Function templates for various controller specific functions
633  */
634 struct mrsas_function_template {
635         uint32_t (*read_fw_status_reg)(struct mrsas_instance *);
636         void (*issue_cmd)(struct mrsas_cmd *, struct mrsas_instance *);
637         int (*issue_cmd_in_sync_mode)(struct mrsas_instance *,
638             struct mrsas_cmd *);
639         int (*issue_cmd_in_poll_mode)(struct mrsas_instance *,
640             struct mrsas_cmd *);
641         void (*enable_intr)(struct mrsas_instance *);
642         void (*disable_intr)(struct mrsas_instance *);
643         int (*intr_ack)(struct mrsas_instance *);
644         int (*init_adapter)(struct mrsas_instance *);
645 /*      int (*reset_adapter)(struct mrsas_instance *); */
609 //      int (*reset_adapter)(struct mrsas_instance *);
646 };

648 /*
649  * ### Helper routines ###
650  */

652 /*
653  * con_log() - console log routine
654  * @param level         : indicates the severity of the message.
655  * @fparam mt           : format string
656  *
657  * con_log displays the error messages on the console based on the current
658  * debug level. Also it attaches the appropriate kernel severity level with
659  * the message.
660  *
661  *
662  * console messages debug levels
663  */
664 #define CL_NONE         0       /* No debug information */
665 #define CL_ANN          1       /* print unconditionally, announcements */
666 #define CL_ANN1         2       /* No-op  */
630 #define CL_ANN1         2       /* No o/p  */
667 #define CL_DLEVEL1      3       /* debug level 1, informative */
668 #define CL_DLEVEL2      4       /* debug level 2, verbose */
669 #define CL_DLEVEL3      5       /* debug level 3, very verbose */

671 #ifdef __SUNPRO_C
672 #define __func__ ""
```

```
 636 #define __func__        __FUNCTION__    //""
 673 #endif

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

 677 /*
 678  * ### SCSA definitions ###
 679  */
 680 #define PKT2TGT(pkt)    ((pkt)->pkt_address.a_target)
 681 #define PKT2LUN(pkt)    ((pkt)->pkt_address.a_lun)
 682 #define PKT2TRAN(pkt)   ((pkt)->pkt_adress.a_hba_tran)
 683 #define ADDR2TRAN(ap)   ((ap)->a_hba_tran)

 685 #define TRAN2MR(tran)   (struct mrsas_instance *)(tran)->tran_hba_private)
 686 #define ADDR2MR(ap)     (TRAN2MR(ADDR2TRAN(ap))

 688 #define PKT2CMD(pkt)    ((struct scsa_cmd *)(pkt)->pkt_ha_private)
 689 #define CMD2PKT(sp)     ((sp)->cmd_pkt)
 690 #define PKT2REQ(pkt)    (&(PKT2CMD(pkt)->request))

 692 #define CMD2ADDR(cmd)   (&CMD2PKT(cmd)->pkt_address)
 693 #define CMD2TRAN(cmd)   (CMD2PKT(cmd)->pkt_address.a_hba_tran)
 694 #define CMD2MR(cmd)     (TRAN2MR(CMD2TRAN(cmd)))

 696 #define CFLAG_DMAVALID          0x0001  /* requires a dma operation */
 697 #define CFLAG_DMASEND           0x0002  /* Transfer from the device */
 698 #define CFLAG_CONSISTENT        0x0040  /* consistent data transfer */

 700 /*
 701  * ### Data structures for ioctl inteface and internal commands ###
 702  */

 704 /*
 705  * Data direction flags
 706  */
 707 #define UIOC_RD         0x00001
 708 #define UIOC_WR         0x00002

 710 #define SCP2HOST(scp)           (scp)->device->host     /* to host */
 711 #define SCP2HOSTDATA(scp)       SCP2HOST(scp)->hostdata /* to soft state */
 712 #define SCP2CHANNEL(scp)        (scp)->device->channel  /* to channel */
 713 #define SCP2TARGET(scp)         (scp)->device->id       /* to target */
 714 #define SCP2LUN(scp)            (scp)->device->lun      /* to LUN */

 716 #define SCSIHOST2ADAP(host)     (((caddr_t *)(host->hostdata))[0])
 717 #define SCP2ADAPTER(scp)                                \
 718         (struct mrsas_instance *)SCSIHOST2ADAP(SCP2HOST(scp))

 720 #define MRDRV_IS_LOGICAL_SCSA(instance, acmd)           \
 721         (acmd->device_id < MRDRV_MAX_LD) ? 1 : 0
 722 #define MRDRV_IS_LOGICAL(ap)                            \
 723         ((ap->a_target < MRDRV_MAX_LD) && (ap->a_lun == 0)) ? 1 : 0
 724 #define MAP_DEVICE_ID(instance, ap)                     \
 725         (ap->a_target)

 727 #define HIGH_LEVEL_INTR                 1
 728 #define NORMAL_LEVEL_INTR               0

 730 #define         IO_TIMEOUT_VAL          0
 731 #define         IO_RETRY_COUNT          3
 732 #define         MAX_FW_RESET_COUNT      3
 733 /*
 734  * scsa_cmd  - Per-command mr private data
 735  * @param cmd_dmahandle         :  dma handle
 736  * @param cmd_dmacookies        :  current dma cookies
 737  * @param cmd_pkt               :  scsi_pkt reference
```



```
 738  * @param cmd_dmacount          :  dma count
 739  * @param cmd_cookie            :  next cookie
 740  * @param cmd_ncookies          :  cookies per window
 741  * @param cmd_cookiecnt         :  cookies per sub-win
 742  * @param cmd_nwin              :  number of dma windows
 743  * @param cmd_curwin            :  current dma window
 744  * @param cmd_dma_offset        :  current window offset
 745  * @param cmd_dma_len           :  current window length
 746  * @param cmd_flags             :  private flags
 747  * @param cmd_cdblen            :  length of cdb
 748  * @param cmd_scblen            :  length of scb
 749  * @param cmd_buf               :  command buffer
 750  * @param channel               :  channel for scsi sub-system
 751  * @param target                :  target for scsi sub-system
 752  * @param lun                   :  LUN for scsi sub-system
 753  *
 754  * - Allocated at same time as scsi_pkt by scsi_hba_pkt_alloc(9E)
 755  * - Pointed to by pkt_ha_private field in scsi_pkt
 756  */
 757 struct scsa_cmd {
 758         ddi_dma_handle_t        cmd_dmahandle;
 759         ddi_dma_cookie_t        cmd_dmacookies[MRSAS_MAX_SGE_CNT];
 760         struct scsi_pkt         *cmd_pkt;
 761         ulong_t                 cmd_dmacount;
 762         uint_t                  cmd_cookie;
 763         uint_t                  cmd_ncookies;
 764         uint_t                  cmd_cookiecnt;
 765         uint_t                  cmd_nwin;
 766         uint_t                  cmd_curwin;
 767         off_t                   cmd_dma_offset;
 768         ulong_t                 cmd_dma_len;
 769         ulong_t                 cmd_flags;
 770         uint_t                  cmd_cdblen;
 771         uint_t                  cmd_scblen;
 772         struct buf              *cmd_buf;
 773         ushort_t                device_id;
 774         uchar_t                 islogical;
 775         uchar_t                 lun;
 776         struct mrsas_device     *mrsas_dev;
 777 };
______unchanged_portion_omitted_

1950 #pragma pack()

1952 #ifndef DDI_VENDOR_LSI
1953 #define DDI_VENDOR_LSI          "LSI"
1954 #endif /* DDI_VENDOR_LSI */

1920 static int      mrsas_getinfo(dev_info_t *, ddi_info_cmd_t,  void *, void **);
1921 static int      mrsas_attach(dev_info_t *, ddi_attach_cmd_t);
1922 static int      mrsas_reset(dev_info_t *, ddi_reset_cmd_t);
1923 int     mrsas_quiesce(dev_info_t *);
1924 static int      mrsas_detach(dev_info_t *, ddi_detach_cmd_t);
1925 static int      mrsas_open(dev_t *, int, int, cred_t *);
1926 static int      mrsas_close(dev_t, int, int, cred_t *);
1927 static int      mrsas_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

1929 static int      mrsas_tran_tgt_init(dev_info_t *, dev_info_t *,
1930                     scsi_hba_tran_t *, struct scsi_device *);
1931 static struct scsi_pkt *mrsas_tran_init_pkt(struct scsi_address *, register
1932                     struct scsi_pkt *, struct buf *, int, int, int, int,
1933                     int (*)(), caddr_t);
1934 static int      mrsas_tran_start(struct scsi_address *,
1935                     register struct scsi_pkt *);
1936 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
1937 static int      mrsas_tran_reset(struct scsi_address *, int);
```

```
1938 static int      mrsas_tran_bus_reset(dev_info_t *, int);
1939 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
1940 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
1941 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
1942                     struct scsi_pkt *);
1943 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
1944 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
1945 static int      mrsas_tran_quiesce(dev_info_t *dip);
1946 static int      mrsas_tran_unquiesce(dev_info_t *dip);
1947 static uint_t   mrsas_isr();
1948 static uint_t   mrsas_softintr();

1950 static struct mrsas_cmd *get_mfi_pkt(struct mrsas_instance *);
1951 static void     return_mfi_pkt(struct mrsas_instance *,
1952                     struct mrsas_cmd *);

1954 static void     free_space_for_mfi(struct mrsas_instance *);
1955 static int      mrsas_tbolt_alloc_additional_dma_buffer
1956 (struct mrsas_instance *);
1957 int             mrsas_tbolt_sync_map_info(struct mrsas_instance *instance);
1958 static int      alloc_additional_dma_buffer(struct mrsas_instance *);
1959 static uint32_t read_fw_status_reg_ppc(struct mrsas_instance *);
1960 static void     issue_cmd_ppc(struct mrsas_cmd *, struct mrsas_instance *);
1961 static int      issue_cmd_in_poll_mode_ppc(struct mrsas_instance *,
1962                     struct mrsas_cmd *);
1963 static int      issue_cmd_in_sync_mode_ppc(struct mrsas_instance *,
1964                     struct mrsas_cmd *);
1965 static void     enable_intr_ppc(struct mrsas_instance *);
1966 static void     disable_intr_ppc(struct mrsas_instance *);
1967 static int      intr_ack_ppc(struct mrsas_instance *);
1968 int     mfi_state_transition_to_ready(struct mrsas_instance *);
1969 static void     flush_cache(struct mrsas_instance *instance);
1970 void    display_scsi_inquiry(caddr_t);
1971 static int      start_mfi_aen(struct mrsas_instance *instance);
1972 static int      handle_drv_ioctl(struct mrsas_instance *instance,
1973                     struct mrsas_ioctl *ioctl, int mode);
1974 static int      handle_mfi_ioctl(struct mrsas_instance *instance,
1975                     struct mrsas_ioctl *ioctl, int mode);
1976 static int      handle_mfi_aen(struct mrsas_instance *instance,
1977                     struct mrsas_aen *aen);
1978 void    fill_up_drv_ver(struct mrsas_drv_ver *dv);
1979 static struct mrsas_cmd *build_cmd(struct mrsas_instance *instance,
1980                     struct scsi_address *ap, struct scsi_pkt *pkt,
1981                     uchar_t *cmd_done);
1982 static struct mrsas_cmd *mrsas_tbolt_build_cmd(struct mrsas_instance *instance,
1983                     struct scsi_address *ap, struct scsi_pkt *pkt,
1984                     uchar_t *cmd_done);
1985 static int      wait_for_outstanding(struct mrsas_instance *instance);
1986 static int      register_mfi_aen(struct mrsas_instance *instance,
1987                     uint32_t seq_num, uint32_t class_locale_word);
1988 static int      issue_mfi_pthru(struct mrsas_instance *instance, struct
1989                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1990 static int      issue_mfi_dcmd(struct mrsas_instance *instance, struct
1991                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1992 static int      issue_mfi_smp(struct mrsas_instance *instance, struct
1993                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1994 static int      issue_mfi_stp(struct mrsas_instance *instance, struct
1995                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1996 static int      abort_aen_cmd(struct mrsas_instance *instance,
1997                     struct mrsas_cmd *cmd_to_abort);


2000 static void     mrsas_rem_intrs(struct mrsas_instance *instance);
2001 static int      mrsas_add_intrs(struct mrsas_instance *instance, int intr_type);

2003 static void     mrsas_tran_tgt_free(dev_info_t *, dev_info_t *,
```



```
2004                    scsi_hba_tran_t *, struct scsi_device *);
2005 static int      mrsas_tran_bus_config(dev_info_t *, uint_t,
2006                     ddi_bus_config_op_t, void *, dev_info_t **);
2007 static int      mrsas_parse_devname(char *, int *, int *);
2008 static int      mrsas_config_all_devices(struct mrsas_instance *);
1956 int mrsas_config_scsi_device(struct mrsas_instance *,
1957     struct scsi_device *, dev_info_t **);
2011 static int      mrsas_config_ld(struct mrsas_instance *, uint16_t,
2012                     uint8_t, dev_info_t **);

1959 #ifdef PDSUPPORT
1960 int mrsas_tbolt_config_pd(struct mrsas_instance *, uint16_t,
1961     uint8_t, dev_info_t **);
2017 static void     mrsas_tbolt_get_pd_info(struct mrsas_instance *,
2018                     struct mrsas_tbolt_pd_info *, int);
1962 #endif

1964 dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t, uint8_t);
1965 int mrsas_service_evt(struct mrsas_instance *, int, int, int, uint64_t);
2021 dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t,
2022                     uint8_t);
2023 static int      mrsas_name_node(dev_info_t *, char *, int);
2024 static void     mrsas_issue_evt_taskq(struct mrsas_eventinfo *);
2025 int     mrsas_service_evt(struct mrsas_instance *, int, int, int,
2026                     uint64_t);
2027 static void     free_additional_dma_buffer(struct mrsas_instance *);

2029 struct mrsas_cmd *get_raid_msg_pkt(struct mrsas_instance *);
1966 void return_raid_msg_pkt(struct mrsas_instance *, struct mrsas_cmd *);
1967 struct mrsas_cmd *get_raid_msg_mfi_pkt(struct mrsas_instance *);
1968 void return_raid_msg_mfi_pkt(struct mrsas_instance *, struct mrsas_cmd *);

1970 int     alloc_space_for_mpi2(struct mrsas_instance *);
1971 void    fill_up_drv_ver(struct mrsas_drv_ver *dv);
2035 int     alloc_additional_dma_buffer(struct mrsas_instance *);

1973 int     mrsas_issue_init_mpi2(struct mrsas_instance *);
1974 struct scsi_pkt *mrsas_tbolt_tran_init_pkt(struct scsi_address *, register
1975                     struct scsi_pkt *, struct buf *, int, int, int, int,
1976                     int (*)(), caddr_t);
1977 int     mrsas_tbolt_tran_start(struct scsi_address *,
1978                     register struct scsi_pkt *);
1979 uint32_t tbolt_read_fw_status_reg(struct mrsas_instance *);
1980 void    tbolt_issue_cmd(struct mrsas_cmd *, struct mrsas_instance *);
1981 int     tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *,
1982                     struct mrsas_cmd *);
1983 int     tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *,
1984                     struct mrsas_cmd *);
1985 void    tbolt_enable_intr(struct mrsas_instance *);
1986 void    tbolt_disable_intr(struct mrsas_instance *);
1987 int     tbolt_intr_ack(struct mrsas_instance *);
1988 uint_t  mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *);
1989     uint_t tbolt_softintr();
1990 int     mrsas_tbolt_dma(struct mrsas_instance *, uint32_t, int, int (*)());
1991 int     mrsas_check_dma_handle(ddi_dma_handle_t handle);
1992 int     mrsas_check_acc_handle(ddi_acc_handle_t handle);
1993 int     mrsas_dma_alloc(struct mrsas_instance *, struct scsi_pkt *,
1994                     struct buf *, int, int (*)());
1995 int     mrsas_dma_move(struct mrsas_instance *,
1996                     struct scsi_pkt *, struct buf *);
1997 int     mrsas_alloc_dma_obj(struct mrsas_instance *, dma_obj_t *,
1998                     uchar_t);
2063 int     mrsas_tbolt_ioc_init(struct mrsas_instance *, dma_obj_t *,
2064         ddi_acc_handle_t);
2065 int     mrsas_tbolt_prepare_pkt(struct scsa_cmd *);
1999 void    mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *, struct mrsas_cmd *);
```

```
2000 int     mrsas_dma_alloc_dmd(struct mrsas_instance *, dma_obj_t *);
2068 int     mr_sas_tbolt_build_sgl(struct mrsas_instance *,
2069         struct scsa_cmd *,
2070         struct mrsas_cmd *,
2071         Mpi2RaidSCSIIORequest_t *,
2072         uint32_t *);
2001 void    tbolt_complete_cmd_in_sync_mode(struct mrsas_instance *,
2002         struct mrsas_cmd *);
2003 int     alloc_req_rep_desc(struct mrsas_instance *);
2076 static void     complete_cmd_in_sync_mode(struct mrsas_instance *,
2077                 struct mrsas_cmd *);
2078 static void     io_timeout_checker(void *instance);
2079 static int      mrsas_kill_adapter(struct mrsas_instance *);
2004 int             mrsas_mode_sense_build(struct scsi_pkt *);
2005 void            push_pending_mfi_pkt(struct mrsas_instance *,
2006                         struct mrsas_cmd *);
2083 static int      mrsas_issue_init_mfi(struct mrsas_instance *);
2007 int     mrsas_issue_pending_cmds(struct mrsas_instance *);
2008 int     mrsas_print_pending_cmds(struct mrsas_instance *);
2009 int     mrsas_complete_pending_cmds(struct mrsas_instance *);
2087 static int      mrsas_reset_ppc(struct mrsas_instance *);
2088 static uint32_t mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *);

2090 MRSAS_REQUEST_DESCRIPTOR_UNION *\
2091         mr_sas_get_request_descriptor(struct mrsas_instance *,
2092         uint16_t, struct mrsas_cmd *);

2011 int     create_mfi_frame_pool(struct mrsas_instance *);
2012 void    destroy_mfi_frame_pool(struct mrsas_instance *);
2013 int     create_mfi_mpi_frame_pool(struct mrsas_instance *);
2014 void    destroy_mfi_mpi_frame_pool(struct mrsas_instance *);
2015 int     create_mpi2_frame_pool(struct mrsas_instance *);
2016 void    destroy_mpi2_frame_pool(struct mrsas_instance *);
2017 int     mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t);
2018 void    mrsas_tbolt_free_additional_dma_buffer(struct mrsas_instance *);
2019 void    free_req_desc_pool(struct mrsas_instance *);
2020 void    free_space_for_mpi2(struct mrsas_instance *);
2021 void    mrsas_dump_reply_desc(struct mrsas_instance *);
2022 void    tbolt_complete_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2106 void    io_timeout_checker(void *);
2023 void    display_scsi_inquiry(caddr_t);
2024 void    service_mfi_aen(struct mrsas_instance *, struct mrsas_cmd *);
2025 int     mrsas_mode_sense_build(struct scsi_pkt *);
2026 int     mrsas_tbolt_get_ld_map_info(struct mrsas_instance *);
2111 void    mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len, U64 start_blk, U32 nu
2112 U8      mrsas_tbolt_check_map_info(struct mrsas_instance *);
2027 struct mrsas_cmd *mrsas_tbolt_build_poll_cmd(struct mrsas_instance *,
2028         struct scsi_address *, struct scsi_pkt *, uchar_t *);
2029 int     mrsas_tbolt_reset_ppc(struct mrsas_instance *instance);
2030 void    mrsas_tbolt_kill_adapter(struct mrsas_instance *instance);
2116 int     mrsas_tbolt_kill_adapter(struct mrsas_instance *instance);
2031 int     abort_syncmap_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2032 void    mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2033     struct IO_REQUEST_INFO *, Mpi2RaidSCSIIORequest_t *, U32);
2118 void    mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],struct
2119         Mpi2RaidSCSIIORequest_t *, U32);


2036 int mrsas_init_adapter_ppc(struct mrsas_instance *instance);
2037 int mrsas_init_adapter_tbolt(struct mrsas_instance *instance);
2038 int mrsas_init_adapter(struct mrsas_instance *instance);
2122 static int
2123 mrsas_undo_resources (dev_info_t *dip, struct mrsas_instance *instance);

2125 int mrsas_init_adapter_ppc (struct mrsas_instance *instance);
2126 int mrsas_init_adapter_tbolt (struct mrsas_instance *instance);
```



```
2127 int mrsas_init_adapter (struct mrsas_instance *instance);

2040 int mrsas_alloc_cmd_pool(struct mrsas_instance *instance);
2041 void mrsas_free_cmd_pool(struct mrsas_instance *instance);

2043 void mrsas_print_cmd_details(struct mrsas_instance *, struct mrsas_cmd *, int);
2044 struct mrsas_cmd *get_raid_msg_pkt(struct mrsas_instance *);
2132 void mrsas_print_cmd_details(struct mrsas_instance *, struct mrsas_cmd *, int );

2046 int mfi_state_transition_to_ready(struct mrsas_instance *);


2049 /* FMA functions. */
2050 int mrsas_common_check(struct mrsas_instance *, struct  mrsas_cmd *);
2051 void mrsas_fm_ereport(struct mrsas_instance *, char *);


2054 #ifdef  __cplusplus
2055 }
_____unchanged_portion_omitted_
```

```
**********************************************************
    3875 Tue Nov  6 14:29:40 2012
new/./mr_sas_list.h
**********************************************************
```

```
/*
 * mr_sas_list.h: header for mr_sas
 *
 * Solaris MegaRAID driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * Copyright (c) 2008-20012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. Neither the name of the author nor the names of its contributors may be
 *    used to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
 * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 */

#ifndef _MR_SAS_LIST_H_
#define _MR_SAS_LIST_H_

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

struct mlist_head {
        struct mlist_head *next, *prev;
};
```

*unchanged_portion_omitted_*

```
void mlist_add(struct mlist_head *, struct mlist_head *);
void mlist_add_tail(struct mlist_head *, struct mlist_head *);
/*
```



```
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static void __list_add(struct mlist_head *new,
        struct mlist_head *prev,
        struct mlist_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}


/*
 * mlist_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static void mlist_add(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head, head->next);
}


/*
 * mlist_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static void mlist_add_tail(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head->prev, head);
}



/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static void __list_del(struct mlist_head *prev,
                        struct mlist_head *next)
{
        next->prev = prev;
        prev->next = next;
}


#if 0
void mlist_del(struct mlist_head *);
/*
 * mlist_del - deletes entry from list.
 * @entry:      the element to delete from the list.
 * Note:        list_empty on entry does not return true after this, the entry
```

```
 * is in an undefined state.
 */

static void mlist_del(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = entry->prev = 0;
}
#endif
void mlist_del_init(struct mlist_head *);
int mlist_empty(struct mlist_head *);
void mlist_splice(struct mlist_head *, struct mlist_head *);

/*
 * mlist_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
static void mlist_del_init(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        INIT_LIST_HEAD(entry);
}


/*
 * mlist_empty - tests whether a list is empty
 * @head: the list to test.
 */
static int mlist_empty(struct mlist_head *head)
{
        return (head->next == head);
}


/*
 * mlist_splice - join two lists
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
static void mlist_splice(struct mlist_head *list, struct mlist_head *head)
{
        struct mlist_head *first = list->next;

        if (first != list) {
                struct mlist_head *last = list->prev;
                struct mlist_head *at = head->next;

                first->prev = head;
                head->next = first;

                last->next = at;
                at->prev = last;
        }
}



/* TODO: set this */
#if 0
#pragma inline(list_add, list_add_tail, __list_del, list_del,
                list_del_init, list_empty, list_splice)
#endif


/*
 * mlist_entry - get the struct for this entry
```



```
 * @ptr:        the &struct mlist_head pointer.
 * @type:       the type of the struct this is embedded in.
 * @member:     the name of the list_struct within the struct.
 */
#define mlist_entry(ptr, type, member) \
        ((type *)((size_t)(ptr) - offsetof(type, member)))


/*
 * mlist_for_each       -       iterate over a list
 * @pos:        the &struct mlist_head to use as a loop counter.
 * @head:       the head for your list.
 */
#define mlist_for_each(pos, head) \
        for (pos = (head)->next, prefetch(pos->next); pos != (head); \
                pos = pos->next, prefetch(pos->next))


/*
 * mlist_for_each_safe - iterate over a list safe against removal of list entry
 * @pos:        the &struct mlist_head to use as a loop counter.
 * @n:          another &struct mlist_head to use as temporary storage
 * @head:       the head for your list.
 */
#define mlist_for_each_safe(pos, n, head) \
        for (pos = (head)->next, n = pos->next; pos != (head); \
                pos = n, n = pos->next)

#ifdef __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
    2874 Tue Nov  6 14:29:40 2012
new/./mr_sas_list.c
**********************************************************
```

```
/*
 * mr_sas_list.h: header for mr_sas
 *
 * Solaris MegaRAID driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 */

/* Copyright 2012 Nexenta Systems, Inc. All rights reserved. */

/*
 * Extract C functions from LSI-provided mr_sas_list.h such that we can both
 * be lint-clean and provide a slightly better source organizational model
 * beyond preprocessor abuse.
 */

#include "mr_sas_list.h"

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void
__list_add(struct mlist_head *new, struct mlist_head *prev,
    struct mlist_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

/*
 * mlist_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
void
mlist_add(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head, head->next);
}

/*
 * mlist_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
void
mlist_add_tail(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head->prev, head);
}
```



```
/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void
__list_del(struct mlist_head *prev, struct mlist_head *next)
{
        next->prev = prev;
        prev->next = next;
}

#if 0
/*
 * mlist_del - deletes entry from list.
 * @entry:      the element to delete from the list.
 * Note:        list_empty on entry does not return true after this, the entry
 * is in an undefined state.
 */

void
mlist_del(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = entry->prev = 0;
}
#endif

/*
 * mlist_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
void
mlist_del_init(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        INIT_LIST_HEAD(entry);
}

/*
 * mlist_empty - tests whether a list is empty
 * @head: the list to test.
 */
int
mlist_empty(struct mlist_head *head)
{
        return (head->next == head);
}

/*
 * mlist_splice - join two lists
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
void
mlist_splice(struct mlist_head *list, struct mlist_head *head)
{
        struct mlist_head *first = list->next;

        if (first != list) {
                struct mlist_head *last = list->prev;
                struct mlist_head *at = head->next;

                first->prev = head;
```

```
129                 head->next = first;

131                 last->next = at;
132                 at->prev = last;
133         }
134 }
```

```
**********************************************************
  106990 Tue Nov  6 14:29:40 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"

/*
 * FMA header files
 */
#include <sys/ddifm.h>
#include <sys/fm/protocol.h>
#include <sys/fm/util.h>
#include <sys/fm/io/ddi.h>


/* Pre-TB command size and TB command size. */
#define MR_COMMAND_SIZE (64*20) /* 1280 bytes */
// 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, struct IO_REQUEST_INFO *);
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 *);

/* Local static prototypes. */
static struct mrsas_cmd *mrsas_tbolt_build_cmd(struct mrsas_instance *,
    struct scsi_address *, struct scsi_pkt *, uchar_t *);
static void mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len_ptr,
    U64 start_blk, U32 num_blocks);
static int mrsas_tbolt_check_map_info(struct mrsas_instance *);
static int mrsas_tbolt_sync_map_info(struct mrsas_instance *);
static int mrsas_tbolt_prepare_pkt(struct scsa_cmd *);
static int mrsas_tbolt_ioc_init(struct mrsas_instance *, dma_obj_t *);
#ifdef PDSUPPORT
static void mrsas_tbolt_get_pd_info(struct mrsas_instance *,
    struct mrsas_tbolt_pd_info *, int);
#endif /* PDSUPPORT */
static int debug_tbolt_fw_faults_after_ocr_g  = 0;

static 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) {
                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_


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

        if (instance->mfi_internal_dma_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->mfi_internal_dma_obj);
                instance->mfi_internal_dma_obj.status = DMA_OBJ_FREED;
        }
        if (instance->mfi_evt_detail_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->mfi_evt_detail_obj);
                instance->mfi_evt_detail_obj.status = DMA_OBJ_FREED;
        }

        for (i = 0; i < 2; i++) {
                if (instance->ld_map_obj[i].status == DMA_OBJ_ALLOCATED) {
                        (void) mrsas_free_dma_obj(instance,
                            instance->ld_map_obj[i]);
```

```
134                         instance->ld_map_obj[i].status = DMA_OBJ_FREED;
135                 }
136         }
137 }
_____unchanged_portion_omitted_


162 /*
163  * ThunderBolt(TB) Request Message Frame Pool
164  */
165 int
166 create_mpi2_frame_pool(struct mrsas_instance *instance)
167 {
168         int             i = 0;
148         int             cookie_cnt;
169         uint16_t        max_cmd;
170         uint32_t        sgl_sz;
171         uint32_t        raid_msg_size;
172         uint32_t        total_size;
173         uint32_t        offset;
174         uint32_t        io_req_base_phys;
175         uint8_t         *io_req_base;
176         struct mrsas_cmd        *cmd;

178         max_cmd = instance->max_fw_cmds;

180         sgl_sz          = 1024;
181         raid_msg_size   = MRSAS_THUNDERBOLT_MSG_SIZE;

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

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

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

193         /*
194          * ThunderBolt(TB) We need to create a single chunk of DMA'ble memory
195          * and then split the memory to 1024 commands. Each command should be
196          * able to contain a RAID MESSAGE FRAME which will embed a MFI_FRAME
197          * within it. Further refer the "alloc_req_rep_desc" function where
198          * we allocate request/reply descriptors queues for a clue.
199          */

201         instance->mpi2_frame_pool_dma_obj.size = total_size;
202         instance->mpi2_frame_pool_dma_obj.dma_attr = mrsas_generic_dma_attr;
203         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_addr_hi =
204             0xFFFFFFFFU;
205         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_count_max =
206             0xFFFFFFFFU;
207         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_sgllen = 1;
208         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_align = 256;

210         if (mrsas_alloc_dma_obj(instance, &instance->mpi2_frame_pool_dma_obj,
211             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
212                 cmn_err(CE_WARN,
213                     "mr_sas: could not alloc mpi2 frame pool");
214                 return (DDI_FAILURE);
215         }

217         bzero(instance->mpi2_frame_pool_dma_obj.buffer, total_size);
```



```
218         instance->mpi2_frame_pool_dma_obj.status |= DMA_OBJ_ALLOCATED;

220         instance->io_request_frames =
221             (uint8_t *)instance->mpi2_frame_pool_dma_obj.buffer;
222         instance->io_request_frames_phy =
223             (uint32_t)
224             instance->mpi2_frame_pool_dma_obj.dma_cookie[0].dmac_address;

226         con_log(CL_DLEVEL3, (CE_NOTE, "io_request_frames 0x%p",
227             (void *)instance->io_request_frames));
206         con_log(CL_DLEVEL3, (CE_NOTE,
207             "io_request_frames 0x%x",
208             instance->io_request_frames));

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

232         io_req_base = (uint8_t *)instance->io_request_frames +
233             MRSAS_THUNDERBOLT_MSG_SIZE;
234         io_req_base_phys = instance->io_request_frames_phy +
235             MRSAS_THUNDERBOLT_MSG_SIZE;

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

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

243                 offset = i * MRSAS_THUNDERBOLT_MSG_SIZE;

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

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

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

255                 cmd->sense1 = (uint8_t *)((uint8_t *)io_req_base +
239                 cmd->sense1 = (uint8_t *)
240                     ((uint8_t *)io_req_base +
256                     (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
257                     (i * SENSE_LENGTH));

259                 cmd->sense_phys_addr1 = (io_req_base_phys +
244                 cmd->sense_phys_addr1 =
245                     (io_req_base_phys +
260                     (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
261                     (i * SENSE_LENGTH));


264                 cmd->SMID = i + 1;
250                 cmd->SMID = i+1;

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

```
254                     cmd->index, cmd->scsi_io_request));

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

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

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

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

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

273                 con_log(CL_DLEVEL3, (CE_NOTE,
274                     "Sgl bufffers phys  [%x]0x%x",
282                     cmd->index, cmd->sgl_phys_addr));
283         }

285         return (DDI_SUCCESS);

287 }


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

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


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

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



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

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

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

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

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

347         instance->mfi_evt_detail_obj.status |= DMA_OBJ_ALLOCATED;

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

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

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

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

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

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

```
367                 instance->ld_map_phy[i] =
368                         (uint32_t)instance->ld_map_obj[i].dma_cookie[0].dmac_add

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

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

383         }

385         return (DDI_SUCCESS);

387 fail_tbolt_additional_buff:
388         mrsas_tbolt_free_additional_dma_buffer(instance);

390         return (DDI_FAILURE);
391 }

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

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

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

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

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

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


421 /*
422  * Allocate Request and Reply  Queue Descriptors.
423  */
424 int
425 alloc_req_rep_desc(struct mrsas_instance *instance)
426 {
427         uint32_t        request_q_sz, reply_q_sz;
428         int             i, max_reply_q_sz;
422         int             i, max_request_q_sz, max_reply_q_sz;
423         uint64_t        request_desc;
429         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
425         uint64_t        *reply_ptr;

431         /*
```



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

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

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

445         reply_q_sz = 8 * max_reply_q_sz;


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

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

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

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

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

472         instance->reply_q_depth = max_reply_q_sz;

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

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

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

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


489         instance->reply_frame_pool_phy =
```

```
490             (uint32_t)instance->reply_desc_dma_obj.dma_cookie[0].dmac_address;

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


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

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


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

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

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

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

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

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

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

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

533         max_request_q_sz = instance->max_fw_cmds;

538         return (DDI_SUCCESS);

540 fail_undo_reply_queue:
541         if (instance->reply_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
542                 (void) mrsas_free_dma_obj(instance,
543                     instance->reply_desc_dma_obj);
544                 instance->reply_desc_dma_obj.status = DMA_OBJ_FREED;
545         }

547         return (DDI_FAILURE);
548 }
```



```
550 /*
551  * mrsas_alloc_cmd_pool_tbolt
552  *
553  * TODO: merge tbolt-specific code into mrsas_alloc_cmd_pool() to have single
554  * routine
549  * TODO: merge tbolt-specific codee into mrsas_alloc_cmd_pool() to have single r
555  */
556 int
557 mrsas_alloc_cmd_pool_tbolt(struct mrsas_instance *instance)
558 {
559         int             i;
560         int             count;
561         uint32_t        max_cmd;
562         uint32_t        reserve_cmd;
563         size_t          sz;

565         struct mrsas_cmd        *cmd;

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


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

574         /*
575          * instance->cmd_list is an array of struct mrsas_cmd pointers.
576          * Allocate the dynamic array first and then allocate individual
577          * commands.
578          */
579         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
575         if (instance->cmd_list == NULL) {
576                 con_log(CL_NONE, (CE_WARN,
577                     "Failed to allocate memory for cmd_list"));
578                 return (DDI_FAILURE);
579         }

581         /* create a frame pool and assign one frame to each cmd */
582         for (count = 0; count < max_cmd; count++) {
583                 instance->cmd_list[count] =
584                     kmem_zalloc(sizeof (struct mrsas_cmd),
583                 instance->cmd_list[count] = kmem_zalloc(sizeof (struct mrsas_cmd
584                     KM_SLEEP);
585                 if (instance->cmd_list[count] == NULL) {
586                         con_log(CL_NONE, (CE_WARN,
587                             "Failed to allocate memory for mrsas_cmd"));
588                         goto mrsas_undo_cmds;
585                 }
590         }

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

589         INIT_LIST_HEAD(&instance->cmd_pool_list);
590         INIT_LIST_HEAD(&instance->cmd_pend_list);
591         INIT_LIST_HEAD(&instance->cmd_app_pool_list);

593         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

595         /* cmd index 0 reservered for IOC INIT */
596         for (i = 1; i < reserve_cmd; i++) {
600         for (i = 1; i < reserve_cmd; i++) {    //cmd index 0 reservered for IOC
597                 cmd             = instance->cmd_list[i];
598                 cmd->index      = i;
599                 mlist_add_tail(&cmd->list, &instance->cmd_app_pool_list);
600         }
```

```
603         for (i = reserve_cmd; i < max_cmd; i++) {
604                 cmd             = instance->cmd_list[i];
605                 cmd->index      = i;
606                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
607         }

609         return (DDI_SUCCESS);

611 mrsas_undo_cmds:
612         if (count > 0) {
613                 /* free each cmd */
614                 for (i = 0; i < count; i++) {
615                         if (instance->cmd_list[i] != NULL) {
616                                 kmem_free(instance->cmd_list[i],
617                                     sizeof (struct mrsas_cmd));
618                         }
619                         if (instance->cmd_list[i] != NULL)
620                                 kmem_free(instance->cmd_list[i],sizeof (struct m
619                         instance->cmd_list[i] = NULL;
620                 }
621         }

623 mrsas_undo_cmd_list:
624         if (instance->cmd_list != NULL)
625                 kmem_free(instance->cmd_list, sz);
627                 kmem_free(instance->cmd_list,sz);
626         instance->cmd_list = NULL;

628         return (DDI_FAILURE);
629 }
_____unchanged_portion_omitted_


661 /*
662  * ThunderBolt(TB) memory allocations for commands/messages/frames.
663  */
664 int
665 alloc_space_for_mpi2(struct mrsas_instance *instance)
666 {
667         /* Allocate command pool (memory for cmd_list & individual commands) */
669         /* Allocate command pool ( memory for cmd_list & individual commands )*/
668         if (mrsas_alloc_cmd_pool_tbolt(instance)) {
669                 cmn_err(CE_WARN, "Error creating cmd pool");
670                 return (DDI_FAILURE);
671         }

673         /* Initialize single reply size and Message size */
674         instance->reply_size = MRSAS_THUNDERBOLT_REPLY_SIZE;
675         instance->raid_io_msg_size = MRSAS_THUNDERBOLT_MSG_SIZE;

677         instance->max_sge_in_main_msg = (MRSAS_THUNDERBOLT_MSG_SIZE -
678             (sizeof (MPI2_RAID_SCSI_IO_REQUEST) -
679             sizeof (MPI2_SGE_IO_UNION)))/ sizeof (MPI2_SGE_IO_UNION);
680         instance->max_sge_in_chain = (MR_COMMAND_SIZE -
681             MRSAS_THUNDERBOLT_MSG_SIZE) / sizeof (MPI2_SGE_IO_UNION);

683         /* Reduce SG count by 1 to take care of group cmds feature in FW */
684         instance->max_num_sge = (instance->max_sge_in_main_msg +
685             instance->max_sge_in_chain - 2);
686         instance->chain_offset_mpt_msg =
687             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16;
688         instance->chain_offset_io_req = (MRSAS_THUNDERBOLT_MSG_SIZE -
689             sizeof (MPI2_SGE_IO_UNION)) / 16;
690         instance->reply_read_index = 0;
```



```
693         /* Allocate Request and Reply descriptors Array */
694         /* Make sure the buffer is aligned to 8 for req/rep  descriptor Pool */
695         if (alloc_req_rep_desc(instance)) {
696                 cmn_err(CE_WARN,
697                     "Error, allocating memory for descripter-pool");
698                 goto mpi2_undo_cmd_pool;
699         }
700         con_log(CL_ANN1, (CE_NOTE, "[request message pool phys addr]0x%x",
701             instance->request_message_pool_phy));


704         /* Allocate MFI Frame pool - for MPI-MFI passthru commands */
705         if (create_mfi_frame_pool(instance)) {
706                 cmn_err(CE_WARN,
707                     "Error, allocating memory for MFI frame-pool");
708                 goto mpi2_undo_descripter_pool;
709         }


712         /* Allocate MPI2 Message pool */
713         /*
714          * Make sure the buffer is alligned to 256 for raid message packet
715          * create a io request pool and assign one frame to each cmd
716          */

718         if (create_mpi2_frame_pool(instance)) {
719                 cmn_err(CE_WARN,
720                     "Error, allocating memory for MPI2 Message-pool");
721                 goto mpi2_undo_mfi_frame_pool;
722         }

724 #ifdef DEBUG
725         con_log(CL_ANN1, (CE_CONT, "[max_sge_in_main_msg]0x%x",
726             instance->max_sge_in_main_msg));
727         con_log(CL_ANN1, (CE_CONT, "[max_sge_in_chain]0x%x",
728             instance->max_sge_in_chain));
729         con_log(CL_ANN1, (CE_CONT,
730             "[max_sge]0x%x", instance->max_num_sge));
731         con_log(CL_ANN1, (CE_CONT, "[chain_offset_mpt_msg]0x%x",
732             instance->chain_offset_mpt_msg));
733         con_log(CL_ANN1, (CE_CONT, "[chain_offset_io_req]0x%x",
734             instance->chain_offset_io_req));
735 #endif


738         /* Allocate additional dma buffer */
739         if (mrsas_tbolt_alloc_additional_dma_buffer(instance)) {
740                 cmn_err(CE_WARN,
741                     "Error, allocating tbolt additional DMA buffer");
742                 goto mpi2_undo_message_pool;
743         }

745         return (DDI_SUCCESS);

747 mpi2_undo_message_pool:
748         destroy_mpi2_frame_pool(instance);

750 mpi2_undo_mfi_frame_pool:
751         destroy_mfi_frame_pool(instance);

753 mpi2_undo_descripter_pool:
754         free_req_rep_desc_pool(instance);

756 mpi2_undo_cmd_pool:
```

```
757         mrsas_free_cmd_pool(instance);

759         return (DDI_FAILURE);
760 }


763 /*
764  * mrsas_init_adapter_tbolt - Initialize fusion interface adapter.
765  */
766 int
767 mrsas_init_adapter_tbolt(struct mrsas_instance *instance)
769 mrsas_init_adapter_tbolt (struct mrsas_instance *instance)
768 {

770         /*
771          * Reduce the max supported cmds by 1. This is to ensure that the
772          * reply_q_sz (1 more than the max cmd that driver may send)
773          * does not exceed max cmds that the FW can support
774          */

776         if (instance->max_fw_cmds > 1008) {
777                 instance->max_fw_cmds = 1008;
778                 instance->max_fw_cmds = instance->max_fw_cmds-1;
779         }

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


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

790                 return (DDI_FAILURE);
791         }

793         /* Send ioc init message */
794         /* NOTE: the issue_init call does FMA checking already. */
795         if (mrsas_issue_init_mpi2(instance) != DDI_SUCCESS) {
796         if ( mrsas_issue_init_mpi2(instance) != DDI_SUCCESS) {
796                 cmn_err(CE_WARN,
797                     " mrsas_issue_init_mpi2() failed.");

799                 goto fail_init_fusion;
800         }

802         instance->unroll.alloc_space_mpi2 = 1;

804         con_log(CL_ANN, (CE_NOTE,
805             "mrsas_init_adapter_tbolt: SUCCESSFUL"));
806                 "mrsas_init_adapter_tbolt: SUCCESSFULL\n"));

807         return (DDI_SUCCESS);

809 fail_init_fusion:

812 fail_undo_alloc_mpi2:
810         free_space_for_mpi2(instance);

812         return (DDI_FAILURE);
813 }
```



```
817 /*
818  * init_mpi2
819  */
820 int
821 mrsas_issue_init_mpi2(struct mrsas_instance *instance)
822 {
823         dma_obj_t init2_dma_obj;
824         int ret_val = DDI_SUCCESS;

826         /* allocate DMA buffer for IOC INIT message */
827         init2_dma_obj.size = sizeof (Mpi2IOCInitRequest_t);
828         init2_dma_obj.dma_attr = mrsas_generic_dma_attr;
829         init2_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
830         init2_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
831         init2_dma_obj.dma_attr.dma_attr_sgllen = 1;
832         init2_dma_obj.dma_attr.dma_attr_align = 256;

834         if (mrsas_alloc_dma_obj(instance, &init2_dma_obj,
835             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
836                 cmn_err(CE_WARN, "mr_sas_issue_init_mpi2 "
837                     "could not allocate data transfer buffer.");
838                 return (DDI_FAILURE);
839         }
840         (void) memset(init2_dma_obj.buffer, 2, sizeof (Mpi2IOCInitRequest_t));
843         (void) memset(init2_dma_obj.buffer, 2,
844             sizeof (Mpi2IOCInitRequest_t));

842         con_log(CL_ANN1, (CE_NOTE,
843             "mrsas_issue_init_mpi2 _phys adr: %x",
847             "mrsas_issue_init_mpi2 _phys adr: %x \n",
844             init2_dma_obj.dma_cookie[0].dmac_address));


847         /* Initialize and send ioc init message */
848         ret_val = mrsas_tbolt_ioc_init(instance, &init2_dma_obj);
852         ret_val = mrsas_tbolt_ioc_init(instance, &init2_dma_obj,
853             init2_dma_obj.acc_handle);
849         if (ret_val == DDI_FAILURE) {
850                 con_log(CL_ANN1, (CE_WARN,
851                     "mrsas_issue_init_mpi2: Failed"));
856                     "mrsas_issue_init_mpi2: Failed\n"));
852                 goto fail_init_mpi2;
853         }

855         /* free IOC init DMA buffer */
856         if (mrsas_free_dma_obj(instance, init2_dma_obj)
857             != DDI_SUCCESS) {
858                 con_log(CL_ANN1, (CE_WARN,
859                     "mrsas_issue_init_mpi2: Free Failed"));
864                     "mrsas_issue_init_mpi2: Free Failed\n"));
860                 return (DDI_FAILURE);
861         }


863         /* Get/Check and sync ld_map info */
864         instance->map_id = 0;
865         if (mrsas_tbolt_check_map_info(instance) == DDI_SUCCESS)
866                 (void) mrsas_tbolt_sync_map_info(instance);
871         if( mrsas_tbolt_check_map_info(instance) == DDI_SUCCESS )
872                 mrsas_tbolt_sync_map_info(instance);


869         /* No mrsas_cmd to send, so send NULL. */
870         if (mrsas_common_check(instance, NULL) != DDI_SUCCESS)
871                 goto fail_init_mpi2;
```

```
873         con_log(CL_ANN, (CE_NOTE,
874             "mrsas_issue_init_mpi2: SUCCESSFUL"));
875                     "mrsas_issue_init_mpi2: SUCCESSFULL\n"));

876         return (DDI_SUCCESS);

878 fail_init_mpi2:
879         (void) mrsas_free_dma_obj(instance, init2_dma_obj);
880         mrsas_free_dma_obj(instance, init2_dma_obj);

881         return (DDI_FAILURE);
882 }

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


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


899 #ifdef DEBUG
900         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
901             (int)sizeof (*mfiFrameInit2)));
902         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", (int)sizeof (*init)));
909             sizeof (*mfiFrameInit2)));
910         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", sizeof (*init)));
903         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
904             (int)sizeof (struct mrsas_init_frame2)));
912             sizeof (struct mrsas_init_frame2)));
905         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n",
906             (int)sizeof (Mpi2IOCInitRequest_t)));
914             sizeof (Mpi2IOCInitRequest_t)));
907 #endif

909         init = (Mpi2IOCInitRequest_t *)mpi2_dma_obj->buffer;
910         numbytes = sizeof (*init);
911         bzero(init, numbytes);

913         ddi_put8(mpi2_dma_obj->acc_handle, &init->Function,
914             MPI2_FUNCTION_IOC_INIT);

916         ddi_put8(mpi2_dma_obj->acc_handle, &init->WhoInit,
917             MPI2_WHOINIT_HOST_DRIVER);

919         /* set MsgVersion and HeaderVersion host driver was built with */
920         ddi_put16(mpi2_dma_obj->acc_handle, &init->MsgVersion,
921             MPI2_VERSION);
```



```
923         ddi_put16(mpi2_dma_obj->acc_handle, &init->HeaderVersion,
924             MPI2_HEADER_VERSION);

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

929         ddi_put16(mpi2_dma_obj->acc_handle, &init->ReplyFreeQueueDepth,
930             0);

932         ddi_put16(mpi2_dma_obj->acc_handle,
933             &init->ReplyDescriptorPostQueueDepth,
934             instance->reply_q_depth);
935         /*
936          * These addresses are set using the DMA cookie addresses from when the
937          * memory was allocated.  Sense buffer hi address should be 0.
938          * ddi_put32(accessp, &init->SenseBufferAddressHigh, 0);
939          */

941         ddi_put32(mpi2_dma_obj->acc_handle,
942             &init->SenseBufferAddressHigh, 0);

944         ddi_put64(mpi2_dma_obj->acc_handle,
945             (uint64_t *)&init->SystemRequestFrameBaseAddress,
946             instance->io_request_frames_phy);

948         ddi_put64(mpi2_dma_obj->acc_handle,
949             &init->ReplyDescriptorPostQueueAddress,
950             instance->reply_frame_pool_phy);

952         ddi_put64(mpi2_dma_obj->acc_handle,
953             &init->ReplyFreeQueueAddress, 0);

955         cmd = instance->cmd_list[0];
956         if (cmd == NULL) {
957                 return (DDI_FAILURE);
958         }
959         cmd->retry_count_for_ocr = 0;
960         cmd->pkt = NULL;
961         cmd->drv_pkt_time = 0;

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

966         frame_hdr = &cmd->frame->hdr;

968         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
969             MFI_CMD_STATUS_POLL_MODE);

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

973         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;

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

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

980         /* Init the MFI Header */
988         // Init the MFI Header
981         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
982             &mfiFrameInit2->cmd, MFI_CMD_OP_INIT);

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

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

```
 987             &mfiFrameInit2->cmd_status,
 988             MFI_STAT_INVALID_STATUS);

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

 992         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
 993             &mfiFrameInit2->queue_info_new_phys_addr_lo,
 994             mpi2_dma_obj->dma_cookie[0].dmac_address);

 996         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
 997             &mfiFrameInit2->data_xfer_len,
 998             sizeof (Mpi2IOCInitRequest_t));

1000         con_log(CL_ANN1, (CE_CONT, "[reply q desc addr]%x",
1001             (int)init->ReplyDescriptorPostQueueAddress));
1009             init->ReplyDescriptorPostQueueAddress));

1003         /* fill driver version information */
1011         /* fill driver version information*/
1004         fill_up_drv_ver(&drv_ver_info);

1006         /* allocate the driver version data transfer buffer */
1007         instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
1015         instance->drv_ver_dma_obj.size = sizeof(drv_ver_info.drv_ver);
1008         instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
1009         instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
1010         instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
1011         instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
1012         instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;

1014         if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
1015             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
1016                 cmn_err(CE_WARN,
1017                     "fusion init: Could not allocate driver version buffer.");
1018                 return (DDI_FAILURE);
1019         }
1020         /* copy driver version to dma buffer */
1021         bzero(instance->drv_ver_dma_obj.buffer, sizeof (drv_ver_info.drv_ver));
1028         /* copy driver version to dma  buffer*/
1029         (void) memset(instance->drv_ver_dma_obj.buffer, 0,sizeof(drv_ver_info.dr
1022         ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
1023             (uint8_t *)drv_ver_info.drv_ver,
1024             (uint8_t *)instance->drv_ver_dma_obj.buffer,
1025             sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);
1033                 sizeof(drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);

1027         /* send driver version physical address to firmware */
1028         ddi_put64(cmd->frame_dma_obj.acc_handle, &mfiFrameInit2->driverversion,
1029             instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);
1035         /*send driver version physical address to firmware*/
1036         ddi_put64(cmd->frame_dma_obj.acc_handle,
1037             &mfiFrameInit2->driverversion, instance->drv_ver_dma_obj.dma_cookie[

1031         con_log(CL_ANN1, (CE_CONT, "[MPIINIT2 frame Phys addr ]0x%x len = %x",
1032             mfiFrameInit2->queue_info_new_phys_addr_lo,
1033             (int)sizeof (Mpi2IOCInitRequest_t)));
1041             sizeof (Mpi2IOCInitRequest_t)));

1035         con_log(CL_ANN1, (CE_CONT, "[Length]%x", mfiFrameInit2->data_xfer_len));

1037         con_log(CL_ANN1, (CE_CONT, "[MFI frame Phys Address]%x len = %x",
1038             cmd->scsi_io_request_phys_addr,
1039             (int)sizeof (struct mrsas_init_frame2)));
1046             cmd->scsi_io_request_phys_addr, sizeof (struct mrsas_init_frame2)));

1041         /* disable interrupts before sending INIT2 frame */
```



```
1042         instance->func_ptr->disable_intr(instance);

1044         req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
1045             instance->request_message_pool;
1046         req_desc->Words = cmd->scsi_io_request_phys_addr;
1047         req_desc->MFAIo.RequestFlags =
1048             (MPI2_REQ_DESCRIPT_FLAGS_MFA << MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1050         cmd->request_desc = req_desc;

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

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

1059         if (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1060             &mfiFrameInit2->cmd_status) == 0) {
1061                 con_log(CL_ANN, (CE_NOTE, "INIT2 Success"));
1069                 ret = DDI_SUCCESS;
1062         } else {
1063                 con_log(CL_ANN, (CE_WARN, "INIT2 Fail"));
1064                 mrsas_dump_reply_desc(instance);
1065                 goto fail_ioc_init;
1066         }

1068         mrsas_dump_reply_desc(instance);

1070         instance->unroll.verBuff = 1;

1072         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_ioc_init: SUCCESSFUL"));
1080         con_log(CL_ANN, (CE_NOTE,
1081                     "mrsas_tbolt_ioc_init: SUCCESSFULL\n"));


1074         return (DDI_SUCCESS);


1077 fail_ioc_init:

1079         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
1089         mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

1081         return (DDI_FAILURE);
1082 }

1084 int
1085 wait_for_outstanding_poll_io(struct mrsas_instance *instance)
1094 int wait_for_outstanding_poll_io(struct mrsas_instance *instance)
1086 {
1087         int i;
1088         uint32_t wait_time = dump_io_wait_time;
1089         for (i = 0; i < wait_time; i++) {
1090                 /*
1091                  * Check For Outstanding poll Commands
1092                  * except ldsync command and aen command
1093                  */
1094                 if (instance->fw_outstanding <= 2) {
1095                         break;
1096                 }
1097                 drv_usecwait(10*MILLISEC);
1098                 /* complete commands from reply queue */
1099                 (void) mr_sas_tbolt_process_outstanding_cmd(instance);
1100         }
1101         if (instance->fw_outstanding > 2) {
```

```
1102                 return (1);
1103         }
1104         return (0);
1105 }
1106 /*
1107  * scsi_pkt handling
1108  *
1109  * Visible to the external world via the transport structure.
1110  */

1112 int
1113 mrsas_tbolt_tran_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1114 {
1115         struct mrsas_instance   *instance = ADDR2MR(ap);
1116         struct scsa_cmd         *acmd = PKT2CMD(pkt);
1117         struct mrsas_cmd        *cmd = NULL;
1127         int                     rval, i;
1118         uchar_t                 cmd_done = 0;
1129         Mpi2RaidSCSIIORequest_t *scsi_raid_io;
1130         uint32_t                msecs = 120 * MILLISEC;

1120         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1121         if (instance->deadadapter == 1) {
1122                 cmn_err(CE_WARN,
1123                     "mrsas_tran_start:TBOLT return TRAN_FATAL_ERROR "
1124                     "for IO, as the HBA doesnt take any more IOs");
1125                 if (pkt) {
1126                         pkt->pkt_reason         = CMD_DEV_GONE;
1127                         pkt->pkt_statistics     = STAT_DISCON;
1128                 }
1129                 return (TRAN_FATAL_ERROR);
1130         }
1131         if (instance->adapterresetinprogress) {
1132                 con_log(CL_ANN, (CE_NOTE, "Reset flag set, "
1133                     "returning mfi_pkt and setting TRAN_BUSY\n"));
1134                 return (TRAN_BUSY);
1135         }
1136         (void) mrsas_tbolt_prepare_pkt(acmd);
1148         rval = mrsas_tbolt_prepare_pkt(acmd);

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

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

1154                 return (TRAN_ACCEPT);
1155         }

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


1162         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1163                 if (instance->fw_outstanding > instance->max_fw_cmds) {
```



```
1164                         cmn_err(CE_WARN,
1165                             "Command Queue Full... Returning BUSY");
1177                         "Command Queue Full... Returning BUSY \n");
1166                         return_raid_msg_pkt(instance, cmd);
1167                         return (TRAN_BUSY);
1168                 }

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

1174                 con_log(CL_ANN, (CE_CONT, "tbolt_issue_cmd: SCSI CDB[0]=0x%x "
1175                     "cmd->index:0x%x SMID 0x%x\n", pkt->pkt_cdbp[0],
1176                     cmd->index, cmd->SMID));
1187                     "cmd->index:0x%x SMID %0x%x\n", pkt->pkt_cdbp[0], cmd->index

1178                 instance->func_ptr->issue_cmd(cmd, instance);

1191                 return (TRAN_ACCEPT);

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

1185         return (TRAN_ACCEPT);
1197         }
1186 }

1188 /*
1189  * prepare the pkt:
1190  * the pkt may have been resubmitted or just reused so
1191  * initialize some fields and do some checks.
1192  */
1193 static int
1205 int
1194 mrsas_tbolt_prepare_pkt(struct scsa_cmd *acmd)
1195 {
1196         struct scsi_pkt *pkt = CMD2PKT(acmd);


1199         /*
1200          * Reinitialize some fields that need it; the packet may
1201          * have been resubmitted
1202          */
1203         pkt->pkt_reason = CMD_CMPLT;
1204         pkt->pkt_state = 0;
1205         pkt->pkt_statistics = 0;
1206         pkt->pkt_resid = 0;

1208         /*
1209          * zero status byte.
1210          */
1211         *(pkt->pkt_scbp) = 0;

1213         return (0);
1214 }


1217 int
1218 mr_sas_tbolt_build_sgl(struct mrsas_instance *instance,
1219     struct scsa_cmd *acmd,
1220     struct mrsas_cmd *cmd,
1221     Mpi2RaidSCSIIORequest_t *scsi_raid_io,
1222     uint32_t *datalen)
```

```
{
        uint32_t                MaxSGEs;
        int                     sg_to_process;
        uint32_t                i, j;
        uint32_t                i, j, SGEdwords = 0;
        uint32_t                numElements, endElement;
        Mpi25IeeeSgeChain64_t   *ieeeChainElement = NULL;
        Mpi25IeeeSgeChain64_t   *scsi_raid_io_sgl_ieee = NULL;
        ddi_acc_handle_t acc_handle =
            instance->mpi2_frame_pool_dma_obj.acc_handle;
        uint32_t                SGLFlags = 0;

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

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

        SGEdwords = sizeof (Mpi25IeeeSgeChain64_t) / 4;

        MaxSGEs = instance->max_sge_in_main_msg;

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

        /* set data transfer flag. */
        // set data transfer flag.
        if (acmd->cmd_flags & CFLAG_DMASEND) {
                ddi_put32(acc_handle, &scsi_raid_io->Control,
                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->Control,
                    MPI2_SCSIIO_CONTROL_WRITE);
        } else {
                ddi_put32(acc_handle, &scsi_raid_io->Control,
                    MPI2_SCSIIO_CONTROL_READ);
                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->Control, MPI2_SCSIIO_CONTROL_READ);
        }


        numElements = acmd->cmd_cookiecnt;

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

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

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

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

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

        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
```



```
                Mpi25IeeeSgeChain64_t *sgl_ptr_end = scsi_raid_io_sgl_ieee;
                sgl_ptr_end += instance->max_sge_in_main_msg - 1;

                ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &sgl_ptr_end->Flags, 0);
        }

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

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

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

                if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
                        if (i == (numElements - 1)) {
                                ddi_put8(acc_handle,
                                    &scsi_raid_io_sgl_ieee->Flags,
                                    IEEE_SGE_FLAGS_END_OF_LIST);
                        if (i == (numElements - 1))
                                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
                                        &scsi_raid_io_sgl_ieee->Flags, IEEE_SGE_
                        }
                }

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

#ifdef DEBUG
                con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]: %" PRIx64,
                con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]:%llx",
                    scsi_raid_io_sgl_ieee->Address));
                con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Length]:%x",
                    scsi_raid_io_sgl_ieee->Length));
                con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Flags]:%x",
                    scsi_raid_io_sgl_ieee->Flags));
#endif

        }

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

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

                con_log(CL_ANN1, (CE_NOTE, "[Chain Element index]:%x", i));

                if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
                        uint16_t ioFlags =
                            ddi_get16(acc_handle, &scsi_raid_io->IoFlags);
                        uint16_t        ioFlags = ddi_get16(instance->mpi2_frame
                                                    &scsi_raid_io->IoFlags);

                        if ((ioFlags &
                            MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) !=
                            MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
```

```
1326                                ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1327                                    (U8)instance->chain_offset_io_req);
1328                        } else {
1329                                ddi_put8(acc_handle,
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
1330                                    &scsi_raid_io->ChainOffset, 0);
1331                        }
1332                } else {
1333                        ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1334                            (U8)instance->chain_offset_io_req);
1348                else {
1349                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1350                                &scsi_raid_io->ChainOffset, (U8)instance->chain_
1335                }

1337                /* prepare physical chain element */
1338                ieeeChainElement = scsi_raid_io_sgl_ieee;

1340                ddi_put8(acc_handle, &ieeeChainElement->NextChainOffset, 0);
1356                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1357                    &ieeeChainElement->NextChainOffset, 0);

1342                if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1343                        ddi_put8(acc_handle, &ieeeChainElement->Flags,
1344                            IEEE_SGE_FLAGS_CHAIN_ELEMENT);
1345                } else {
1346                        ddi_put8(acc_handle, &ieeeChainElement->Flags,
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,
1347                            (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
1348                            MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
1349                }

1351                ddi_put32(acc_handle, &ieeeChainElement->Length,
1368                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1369                    &ieeeChainElement->Length,
1352                    (sizeof (MPI2_SGE_IO_UNION) * (numElements - i)));

1354                ddi_put64(acc_handle, &ieeeChainElement->Address,
1372                ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
1373                    &ieeeChainElement->Address,
1355                    (U64)cmd->sgl_phys_addr);

1357                sg_to_process = numElements - i;

1359                con_log(CL_ANN1, (CE_NOTE,
1360                    "[Additional SGE Count]:%x", endElement));

1362                /* point to the chained SGL buffer */
1363                scsi_raid_io_sgl_ieee = (Mpi25IeeeSgeChain64_t *)cmd->sgl;

1365                /* build rest of the SGL in chained buffer */
1366                for (j = 0; j < sg_to_process; j++, scsi_raid_io_sgl_ieee++) {
1367                        con_log(CL_DLEVEL3, (CE_NOTE, "[remaining SGL]:%x", i));

1369                        ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
1388                        ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
1389                            &scsi_raid_io_sgl_ieee->Address,
```



```
1370                            acmd->cmd_dmacookies[i].dmac_laddress);

1372                        ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length,
1392                        ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1393                            &scsi_raid_io_sgl_ieee->Length,
1373                            acmd->cmd_dmacookies[i].dmac_size);

1375                        ddi_put8(acc_handle, &scsi_raid_io_sgl_ieee->Flags, 0);
1396                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1397                            &scsi_raid_io_sgl_ieee->Flags, 0);

1377                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1378                                if (i == (numElements - 1)) {
1379                                        ddi_put8(acc_handle,
1380                                            &scsi_raid_io_sgl_ieee->Flags,
1381                                            IEEE_SGE_FLAGS_END_OF_LIST);
1400                                if (i == (numElements - 1))
1401                                        ddi_put8(instance->mpi2_frame_pool_dma_o
1402                                                &scsi_raid_io_sgl_ieee->Flags, I
1382                                }
1383                        }

1385                        *datalen += acmd->cmd_dmacookies[i].dmac_size;

1387 #if DEBUG
1388                        con_log(CL_DLEVEL1, (CE_NOTE,
1389                            "[SGL Address]: %" PRIx64,
1409                            "[SGL Address]:%llx",
1390                            scsi_raid_io_sgl_ieee->Address));
1391                        con_log(CL_DLEVEL1, (CE_NOTE,
1392                            "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
1393                        con_log(CL_DLEVEL1, (CE_NOTE,
1394                            "[SGL Flags]:%x", scsi_raid_io_sgl_ieee->Flags));
1395 #endif

1397                        i++;
1398                }
1399        }

1401        return (0);
1402 } /*end of BuildScatterGather */


1405 /*
1406  * build_cmd
1407  */
1408 static struct mrsas_cmd *
1428 struct mrsas_cmd *
1409 mrsas_tbolt_build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
1410     struct scsi_pkt *pkt, uchar_t *cmd_done)
1411 {
1412        uint8_t         fp_possible = 0;
1413        uint32_t        index;
1414        uint32_t        lba_count = 0;
1415        uint32_t        start_lba_hi = 0;
1416        uint32_t        start_lba_lo = 0;
1417        ddi_acc_handle_t acc_handle =
1418            instance->mpi2_frame_pool_dma_obj.acc_handle;
1433        uint16_t        flags = 0;
1434        uint32_t        i, index;
1435        uint32_t        context;
1436        uint32_t        sge_bytes;
1437        uint8_t         ChainOffsetValue;
1438        uint32_t        SGLFlags;
1439        uint32_t        lba_count=0;
1440        uint32_t        start_lba_hi=0;
```

```
1441        uint32_t        start_lba_lo=0;
1442        ddi_acc_handle_t acc_handle;
1419        struct mrsas_cmd                *cmd = NULL;
1420        struct scsa_cmd                 *acmd = PKT2CMD(pkt);
1421        MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
1422        Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
1447        Mpi25IeeeSgeChain64_t           *scsi_raid_io_sgl_ieee;
1423        uint32_t                        datalen;
1424        struct IO_REQUEST_INFO io_info;
1425        MR_FW_RAID_MAP_ALL *local_map_ptr;
1451        MR_LD_RAID *raid;
1452        U32     ld;
1426        uint16_t pd_cmd_cdblen;

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

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

1435        *cmd_done = 0;

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

1442        index = cmd->index;
1443        ReqDescUnion =  mr_sas_get_request_descriptor(instance, index);
1470        ReqDescUnion =  mr_sas_get_request_descriptor(instance, index, cmd);
1444        ReqDescUnion->Words = 0;
1445        ReqDescUnion->SCSIIO.SMID = cmd->SMID;
1446        ReqDescUnion->SCSIIO.RequestFlags =
1447            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1448            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);


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

1455        /* lets get the command directions */
1456        if (acmd->cmd_flags & CFLAG_DMASEND) {
1457                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1458                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1459                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1460                            DDI_DMA_SYNC_FORDEV);
1461                }
1462        } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
1463                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1464                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1465                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1466                            DDI_DMA_SYNC_FORCPU);
1467                }
1468        } else {
1469                con_log(CL_ANN, (CE_NOTE, "NO DMA"));
1496                con_log(CL_ANN, (CE_NOTE, "NO DMA\n"));
1470        }


1473        /* get SCSI_IO raid message frame pointer */
1500        // get SCSI_IO raid message frame pointer
1474        scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

1476        /* zero out SCSI_IO raid message frame */
```



```
1477        bzero(scsi_raid_io, sizeof (Mpi2RaidSCSIIORequest_t));
1504        memset(scsi_raid_io, 0, sizeof(Mpi2RaidSCSIIORequest_t));

1479        /* Set the ldTargetId set by BuildRaidContext() */
1480        ddi_put16(acc_handle, &scsi_raid_io->RaidContext.ldTargetId,
1506        /*Set the ldTargetId set by BuildRaidContext() */
1507        ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1508                    &scsi_raid_io->RaidContext.ldTargetId,
1481            acmd->device_id);

1483        /*  Copy CDB to scsi_io_request message frame */
1484        ddi_rep_put8(acc_handle,
1485            (uint8_t *)pkt->pkt_cdbp, (uint8_t *)scsi_raid_io->CDB.CDB32,
1512        ddi_rep_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1513                    (uint8_t *)pkt->pkt_cdbp,
1514                    (uint8_t *)scsi_raid_io->CDB.CDB32,
1486            acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

1488        /*
1489         * Just the CDB length, rest of the Flags are zero
1518        * Just the CDB length,rest of the Flags are zero
1490         * This will be modified later.
1491         */
1492        ddi_put16(acc_handle, &scsi_raid_io->IoFlags, acmd->cmd_cdblen);
1521        ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1522                    &scsi_raid_io->IoFlags,
1523                    acmd->cmd_cdblen);

1494        pd_cmd_cdblen = acmd->cmd_cdblen;

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

1506                if (acmd->islogical) {
1507                        /* Initialize sense Information */
1508                        if (cmd->sense1 == NULL) {
1509                                con_log(CL_ANN, (CE_NOTE, "tbolt_build_cmd: "
1510                                    "Sense buffer ptr NULL "));
1540                        con_log(CL_ANN, (CE_NOTE,
1541                                "tbolt_build_cmd: Sense buffer ptr NULL \n"));
1511                        }
1512                        bzero(cmd->sense1, SENSE_LENGTH);
1513                        con_log(CL_DLEVEL2, (CE_NOTE, "tbolt_build_cmd "
1514                            "CDB[0] = %x\n", pkt->pkt_cdbp[0]));
1544                con_log(CL_DLEVEL2, (CE_NOTE,
1545                    "tbolt_build_cmd CDB[0] = %x\n", pkt->pkt_cdbp[0]));

1516                        if (acmd->cmd_cdblen == CDB_GROUP0) {
1517                                /* 6-byte cdb */
1547                if (acmd->cmd_cdblen == CDB_GROUP0) {                   /* 6-byt
1518                                lba_count = (uint16_t)(pkt->pkt_cdbp[4]);
1519                                start_lba_lo = ((uint32_t)(pkt->pkt_cdbp[3]) |
1549                        start_lba_lo =
1550                            ((uint32_t)(pkt->pkt_cdbp[3]) |
1520                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
1521                                    ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
1522                                    << 16));
1523                        } else if (acmd->cmd_cdblen == CDB_GROUP1) {
1524                                /* 10-byte cdb */
```

```
                        ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F) << 16));
                } else if (acmd->cmd_cdblen == CDB_GROUP1) {            /* 10-by
                        lba_count =
                            (((uint16_t)(pkt->pkt_cdbp[8])) |
                            ((uint16_t)(pkt->pkt_cdbp[7]) << 8));

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

                } else if (acmd->cmd_cdblen == CDB_GROUP5) {
                        /* 12-byte cdb */
                } else if (acmd->cmd_cdblen == CDB_GROUP5) {            /* 12-by
                        lba_count = (
                            ((uint32_t)(pkt->pkt_cdbp[9])) |
                            ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
                            ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
                            ((uint32_t)(pkt->pkt_cdbp[6]) << 24));

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

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

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

                        start_lba_hi = (
                            ((uint32_t)(pkt->pkt_cdbp[5])) |
                            ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
                            ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
                            ((uint32_t)(pkt->pkt_cdbp[2]) << 24));
                }

                if (instance->tbolt &&
                    ((lba_count * 512) > mrsas_tbolt_max_cap_maxxfer)) {
                        cmn_err(CE_WARN, " IO SECTOR COUNT exceeds "
                            "controller limit 0x%x sectors",
                            lba_count);
                }
                        ((lba_count * 512) > mrsas_tbolt_max_cap_maxxfer
                cmn_err(CE_WARN," IO SECTOR COUNT exceeds controller lim

                bzero(&io_info, sizeof (struct IO_REQUEST_INFO));
                io_info.ldStartBlock = ((uint64_t)start_lba_hi << 32) |
                    start_lba_lo;
        memset(&io_info, 0, sizeof (struct IO_REQUEST_INFO));
        io_info.ldStartBlock = ((uint64_t)start_lba_hi << 32) | start_lb
                io_info.numBlocks = lba_count;
                io_info.ldTgtId = acmd->device_id;
```



```
                if (acmd->cmd_flags & CFLAG_DMASEND)
                        io_info.isRead = 0;
                else
                        io_info.isRead = 1;


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

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

                if ((MR_TargetIdToLdGet(
                    acmd->device_id, local_map_ptr) >=
                    MAX_LOGICAL_DRIVES) || !instance->fast_path_io) {
                        cmn_err(CE_NOTE, "Fast Path NOT Possible, "
                            "targetId >= MAX_LOGICAL_DRIVES || "
                            "!instance->fast_path_io");
        if  ( (MR_TargetIdToLdGet(acmd->device_id, local_map_ptr) >= MAX
                        cmn_err(CE_NOTE,
                            "Fast Path NOT Possible, targetId >= MAX_LOGICAL
                        fp_possible = 0;
                        /* Set Regionlock flags to BYPASS */
                        /* io_request->RaidContext.regLockFlags  = 0; */
                        ddi_put8(acc_handle,
                        /* Set Regionlock flags to BYPASS
                        io_request->RaidContext.regLockFlags  = 0; */
                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
                            &scsi_raid_io->RaidContext.regLockFlags, 0);
                } else {
                        if (MR_BuildRaidContext(instance, &io_info,
                            &scsi_raid_io->RaidContext, local_map_ptr))
                                fp_possible = io_info.fpOkForIo;
                }

                if (!enable_fp)
        if (!enable_fp) {
                        fp_possible = 0;

                con_log(CL_ANN1, (CE_NOTE, "enable_fp %d  "
                    "instance->fast_path_io %d fp_possible %d",
        }
        con_log(CL_ANN1, (CE_NOTE,
                            "enable_fp %d  instance->fast_path_io %d fp_poss
                    enable_fp, instance->fast_path_io, fp_possible));

        if (fp_possible) {

                        /* Check for DIF enabled LD */
                        if (MR_CheckDIF(acmd->device_id, local_map_ptr)) {
                                /* Prepare 32 Byte CDB for DIF capable Disk */
                                mrsas_tbolt_prepare_cdb(instance,
                                    scsi_raid_io->CDB.CDB32,
                                    &io_info, scsi_raid_io, start_lba_lo);
                                    &io_info,
                                    scsi_raid_io,
                                    start_lba_lo);
                        } else {
                                mrsas_tbolt_set_pd_lba(scsi_raid_io->CDB.CDB32,
                                    (uint8_t *)&pd_cmd_cdblen,
                                    io_info.pdBlock, io_info.numBlocks);
                                ddi_put16(acc_handle,
                                    &scsi_raid_io->IoFlags, pd_cmd_cdblen);
                                  (uint8_t *)&pd_cmd_cdblen, io_info.pdBlock, i
```

```
1651                                    ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
1652                                              &scsi_raid_io->IoFlags,
1653                                              pd_cmd_cdblen);
1633                            }

1635                            ddi_put8(acc_handle, &scsi_raid_io->Function,
1656                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1657                                &scsi_raid_io->Function,
1636                                MPI2_FUNCTION_SCSI_IO_REQUEST);

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

1642                            if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1643                                    uint8_t regLockFlags = ddi_get8(acc_handle,
1665                                    uint8_t regLockFlags = ddi_get8(instance->mpi2_f
1644                                        &scsi_raid_io->RaidContext.regLockFlags);
1645                                    uint16_t IoFlags = ddi_get16(acc_handle,
1667                                    uint16_t IoFlags = ddi_get16(instance->mpi2_fram
1646                                        &scsi_raid_io->IoFlags);

1648                                    if (regLockFlags == REGION_TYPE_UNUSED)
1649                                            ReqDescUnion->SCSIIO.RequestFlags =
1650                                                (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1651                                                MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1672                                                (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK

1653                                    IoFlags |=
1654                                        MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH;
1655                                    regLockFlags |=
1656                                        (MR_RL_FLAGS_GRANT_DESTINATION_CUDA |
1657                                        MR_RL_FLAGS_SEQ_NUM_ENABLE);
1674                                    IoFlags |= MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_
1675                                    regLockFlags |= (MR_RL_FLAGS_GRANT_DESTINATION_C

1659                                    ddi_put8(acc_handle,
1677                                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1660                                        &scsi_raid_io->ChainOffset, 0);
1661                                    ddi_put8(acc_handle,
1662                                        &scsi_raid_io->RaidContext.nsegType,
1663                                        ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1664                                        MPI2_TYPE_CUDA));
1665                                    ddi_put8(acc_handle,
1666                                        &scsi_raid_io->RaidContext.regLockFlags,
1667                                        regLockFlags);
1668                                    ddi_put16(acc_handle,
1679                                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1680                                                  &scsi_raid_io->RaidContext.n
1681                                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1682                                                  &scsi_raid_io->RaidContext.r
1683                                    ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
1669                                        &scsi_raid_io->IoFlags, IoFlags);
1670                            }

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



```
1683                                            ~MEGASAS_LOAD_BALANCE_FLAG;
1684                                    }
1687                                    if ((instance->load_balance_info[acmd->device_id].loadBa
1688                                            io_info.devHandle = get_updated_dev_handle(&inst
1689                                            cmd->load_balance_flag |= MEGASAS_LOAD_BALANCE_F
1690                                    } else
1691                                            cmd->load_balance_flag &= ~MEGASAS_LOAD_BALANCE_

1686                                    ReqDescUnion->SCSIIO.DevHandle = io_info.devHandle;
1687                                    ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1694                                    ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1695                                        &scsi_raid_io->DevHandle,
1688                                        io_info.devHandle);

1690                            } else {
1691                                    ddi_put8(acc_handle, &scsi_raid_io->Function,
1699                                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1700                                        &scsi_raid_io->Function,
1692                                        MPI2_FUNCTION_LD_IO_REQUEST);

1694                                    ddi_put16(acc_handle,
1703                                    ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1695                                        &scsi_raid_io->DevHandle, acmd->device_id);

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

1701                                    ddi_put16(acc_handle,
1702                                        &scsi_raid_io->RaidContext.timeoutValue,
1703                                        local_map_ptr->raidMap.fpPdIoTimeoutSec);
1710                                    ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
1711                                                  &scsi_raid_io->RaidContext.timeoutVa

1705                                    if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1706                                            uint8_t regLockFlags = ddi_get8(acc_handle,
1714                                            uint8_t regLockFlags = ddi_get8(instance->mpi2_f
1707                                                &scsi_raid_io->RaidContext.regLockFlags);

1709                                            if (regLockFlags == REGION_TYPE_UNUSED) {
1717                                            if (regLockFlags == REGION_TYPE_UNUSED)
1710                                                    ReqDescUnion->SCSIIO.RequestFlags =
1711                                                        (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1712                                                        MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1713                                            }
1719                                                    (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK

1715                                            regLockFlags |=
1716                                                (MR_RL_FLAGS_GRANT_DESTINATION_CPU0 |
1717                                                MR_RL_FLAGS_SEQ_NUM_ENABLE);
1721                                            regLockFlags |= (MR_RL_FLAGS_GRANT_DESTINATION_C

1719                                            ddi_put8(acc_handle,
1720                                                &scsi_raid_io->RaidContext.nsegType,
1721                                                ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1722                                                MPI2_TYPE_CUDA));
1723                                            ddi_put8(acc_handle,
1724                                                &scsi_raid_io->RaidContext.regLockFlags,
1725                                                regLockFlags);
1723                                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1724                                                          &scsi_raid_io->RaidContext.n
1725                                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1726                                                          &scsi_raid_io->RaidContext.r
1726                                    }
1728
1727                            } /* Not FP */
```

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


                /*
                 * Set sense buffer physical address/length in scsi_io_request.
                 */
                ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
                /* Set sense buffer physical address/length in scsi_io_request.*
                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->SenseBufferLowAddress,
                    cmd->sense_phys_addr1);
                ddi_put8(acc_handle, &scsi_raid_io->SenseBufferLength,
                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->SenseBufferLength,
                    SENSE_LENGTH);

                /* Construct SGL */
                ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
                /* Construct SGL*/
                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->SGLOffset0,
                    offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

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

                ddi_put32(acc_handle, &scsi_raid_io->DataLength, datalen);
                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
                    &scsi_raid_io->DataLength, datalen);

                break;

        }
        else {
#ifndef PDSUPPORT       /* if PDSUPPORT, skip break and fall through */
        } else {
                break;
#endif
        }
        /* fall through For all non-rd/wr cmds */
        default:
                switch (pkt->pkt_cdbp[0]) {
                case 0x35: { /* SCMD_SYNCHRONIZE_CACHE */
                case 0x35: { // SCMD_SYNCHRONIZE_CACHE
                        return_raid_msg_pkt(instance, cmd);
                        *cmd_done = 1;
                        return (NULL);
                }

                case SCMD_MODE_SENSE:
                case SCMD_MODE_SENSE_G1: {
                        union scsi_cdb  *cdbp;
                        uint16_t        page_code;

                        cdbp = (void *)pkt->pkt_cdbp;
                        page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
                        switch (page_code) {
                        case 0x3:
                        case 0x4:
                                (void) mrsas_mode_sense_build(pkt);
                                return_raid_msg_pkt(instance, cmd);
                                *cmd_done = 1;
```



```
                                return (NULL);
                        }
                        break;
                }

                default: {
                        /*
                         * Here we need to handle PASSTHRU for
                         * Logical Devices. Like Inquiry etc.
                         */
                        /* Here we need to handle PASSTHRU for
                           Logical Devices. Like Inquiry etc.*/

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

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

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

                                ddi_put8(acc_handle, &scsi_raid_io->Function,
                                    MPI2_FUNCTION_SCSI_IO_REQUEST);
                                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
                                    &scsi_raid_io->Function, MPI2_FUNCTION_SCSI_

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

                                ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
                                    local_map_ptr->raidMap.
                                    devHndlInfo[acmd->device_id].curDevHdl);
                                ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
                                        &scsi_raid_io->DevHandle,
                                                local_map_ptr->raidMap.devHndlIn


                                /* Set regLockFlasgs to REGION_TYPE_BYPASS */
                                ddi_put8(acc_handle,
                                /*Set regLockFlasgs to REGION_TYPE_BYPASS */
                                ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
                                    &scsi_raid_io->RaidContext.regLockFlags, 0);
                                ddi_put64(acc_handle,
                                    &scsi_raid_io->RaidContext.regLockRowLBA,
                                    0);
                                ddi_put32(acc_handle,
                                    &scsi_raid_io->RaidContext.regLockLength,
                                    0);
                                ddi_put8(acc_handle,
                                    &scsi_raid_io->RaidContext.RAIDFlags,
                                    MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD <<
                                    MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT);
                                ddi_put16(acc_handle,
                                    &scsi_raid_io->RaidContext.timeoutValue,
                                    local_map_ptr->raidMap.fpPdIoTimeoutSec);
                                ddi_put16(acc_handle,
                                    &scsi_raid_io->RaidContext.ldTargetId,
                                    acmd->device_id);
                                ddi_put8(acc_handle,
                                ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_
                                        &scsi_raid_io->RaidContext.regLockRowLBA
                                ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_
```

```
1818                                    &scsi_raid_io->RaidContext.regLockLength
1819                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1820                                    MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD << M
1821                            ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
1822                                &scsi_raid_io->RaidContext.timeoutValue, loc
1823                            ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
1824                                    &scsi_raid_io->RaidContext.ldTargetId, a
1825                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1829                                &scsi_raid_io->LUN[1], acmd->lun);

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

1834                    } else {
1835                            ddi_put8(acc_handle, &scsi_raid_io->Function,
1836                                MPI2_FUNCTION_LD_IO_REQUEST);
1837                            ddi_put8(acc_handle,
1832                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1833                                    &scsi_raid_io->Function, MPI2_FU
1834                            ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_h
1838                                &scsi_raid_io->LUN[1], acmd->lun);
1839                            ddi_put16(acc_handle,
1836                            ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_
1840                                &scsi_raid_io->DevHandle, acmd->device_id);
1841                            ReqDescUnion->SCSIIO.RequestFlags =
1842                                (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
1843                                MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1839                                (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO << MPI2
1844                    }

1846                    /*
1847                     * Set sense buffer physical address/length in
1848                     * scsi_io_request.
1849                     */
1850                    ddi_put32(acc_handle,
1842                    /* Set sense buffer physical address/length in scsi_io_r
1843                    ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1851                        &scsi_raid_io->SenseBufferLowAddress,
1852                        cmd->sense_phys_addr1);
1853                    ddi_put8(acc_handle,
1846                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1854                        &scsi_raid_io->SenseBufferLength, SENSE_LENGTH);

1856                    /* Construct SGL */
1857                    ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1849                    /* Construct SGL*/
1850                    ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1851                        &scsi_raid_io->SGLOffset0,
1858                        offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1860                    (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1854                    mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1861                        scsi_raid_io, &datalen);

1863                    ddi_put32(acc_handle,
1857                    ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
1864                        &scsi_raid_io->DataLength, datalen);


1867                    con_log(CL_ANN, (CE_CONT,
1868                        "tbolt_build_cmd CDB[0] =%x, TargetID =%x\n",
1869                        pkt->pkt_cdbp[0], acmd->device_id));
1870                    con_log(CL_DLEVEL1, (CE_CONT,
1871                        "data length = %x\n",
1872                        scsi_raid_io->DataLength));
1873                    con_log(CL_DLEVEL1, (CE_CONT,
```



```
1874                                "cdb length = %x\n",
1875                                acmd->cmd_cdblen));
1876                    }
1877                            break;
1878                    }

1880            }
1875    #ifdef lint
1876            context = context;
1877    #endif

1882            return (cmd);
1883    }
______unchanged_portion_omitted_

1981    void
1982    tbolt_issue_cmd(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
1983    {
1984            MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;
1985            atomic_add_16(&instance->fw_outstanding, 1);

1987            struct scsi_pkt *pkt;

1989            con_log(CL_ANN1,
1990                (CE_NOTE, "tbolt_issue_cmd: cmd->[SMID]=0x%X", cmd->SMID));
1986            con_log(CL_ANN1, (CE_NOTE, "tbolt_issue_cmd: cmd->[SMID]=0x%X", cmd->SMI

1992            con_log(CL_DLEVEL1, (CE_CONT,
1993                " [req desc Words] %" PRIx64 " \n", req_desc->Words));
1989                " [req desc Words] %llx \n", req_desc->Words));
1994            con_log(CL_DLEVEL1, (CE_CONT,
1995                " [req desc low part] %x \n",
1996                (uint_t)(req_desc->Words & 0xffffffffff)));
1991                " [req desc low part] %x \n", req_desc->Words));
1997            con_log(CL_DLEVEL1, (CE_CONT,
1998                " [req desc high part] %x \n", (uint_t)(req_desc->Words >> 32)));
1993                " [req desc high part] %x \n", (req_desc->Words >> 32)));
1999            pkt = cmd->pkt;

2001            if (pkt) {
2002                    con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
2003                        "ISSUED CMD TO FW : called : cmd:"
2004                        ": %p instance : %p pkt : %p pkt_time : %x\n",
2005                        gethrtime(), (void *)cmd, (void *)instance,
2006                        (void *)pkt, cmd->drv_pkt_time));
2007                    if (instance->adapterresetinprogress) {
2008                            cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2009                            con_log(CL_ANN, (CE_NOTE,
2010                                "TBOLT Reset the scsi_pkt timer"));
2011                    } else {
2012                            push_pending_mfi_pkt(instance, cmd);
2013                    }

2015            } else {
2016                    con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
2017                        "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
2018                        "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
2019            }

2021            /* Issue the command to the FW */
2022            mutex_enter(&instance->reg_write_mtx);
2023            WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2024            WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2025            mutex_exit(&instance->reg_write_mtx);
2026    }
```

```
/*
 * issue_cmd_in_sync_mode
 */
int
tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *instance,
    struct mrsas_cmd *cmd)
{
        int             i;
        uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
        MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

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

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


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

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

                return (DDI_SUCCESS);
        } else {
                con_log(CL_ANN1, (CE_NOTE,
                    "tbolt_issue_cmd_in_sync_mode: pushing the pkt"));
                con_log(CL_ANN1, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode: pushin
                push_pending_mfi_pkt(instance, cmd);
        }

        con_log(CL_DLEVEL2, (CE_NOTE,
            "HighQport offset :%p",
            (void *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT)));
            "HighQport offset :%lx",
            (uint32_t *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT)));
        con_log(CL_DLEVEL2, (CE_NOTE,
            "LowQport offset :%p",
            (void *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT)));
            "LowQport offset :%lx",
            (uint32_t *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT)));

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


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

        con_log(CL_ANN1, (CE_NOTE,
            " req desc high part %x", (uint_t)(req_desc->Words >> 32)));
        con_log(CL_ANN1, (CE_NOTE, " req desc low part %x",
            (uint_t)(req_desc->Words & 0xffffffff)));
```



```
            " req desc high part %x \n", (req_desc->Words >> 32)));
        con_log(CL_ANN1, (CE_NOTE,
            " req desc low part %x \n", req_desc->Words));

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


        if (i < (msecs -1)) {
                return (DDI_SUCCESS);
        } else {
                return (DDI_FAILURE);
        }
}

/*
 * issue_cmd_in_poll_mode
 */
int
tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *instance,
    struct mrsas_cmd *cmd)
{
        int             i;
        uint16_t        flags;
        uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
        struct mrsas_header *frame_hdr;

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

        MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

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

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

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

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

        if (ddi_get8(cmd->frame_dma_obj.acc_handle,
```

```
2146             &frame_hdr->cmd_status) == MFI_CMD_STATUS_POLL_MODE) {
2147                 con_log(CL_ANN1, (CE_NOTE,
2148                     " cmd failed %" PRIx64, (req_desc->Words)));
2138                     " cmd failed %x \n", (req_desc->Words)));
2149                 return (DDI_FAILURE);
2150         }

2152         return (DDI_SUCCESS);
2153 }

2155 void
2156 tbolt_enable_intr(struct mrsas_instance *instance)
2157 {
2148         uint32_t        mask;

2158         /* TODO: For Thunderbolt/Invader also clear intr on enable */
2159         /* writel(~0, &regs->outbound_intr_status); */
2160         /* readl(&regs->outbound_intr_status); */
2151         //writel(~0, &regs->outbound_intr_status);
2152         //readl(&regs->outbound_intr_status);

2162         WR_OB_INTR_MASK(~(MFI_FUSION_ENABLE_INTERRUPT_MASK), instance);

2164         /* dummy read to force PCI flush */
2165         (void) RD_OB_INTR_MASK(instance);
2157         mask = RD_OB_INTR_MASK(instance);

2167 }

2169 void
2170 tbolt_disable_intr(struct mrsas_instance *instance)
2171 {
2172         uint32_t mask = 0xFFFFFFFF;
2165         uint32_t status;


2174         WR_OB_INTR_MASK(mask, instance);

2176         /* Dummy readl to force pci flush */

2178         (void) RD_OB_INTR_MASK(instance);
2172         status = RD_OB_INTR_MASK(instance);
2179 }


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

2187         /* check if it is our interrupt */
2188         status = RD_OB_INTR_STATUS(instance);
2189         con_log(CL_ANN1, (CE_NOTE,
2190             "chkpnt: Entered tbolt_intr_ack status = %d", status));
2184             "chkpnt: Entered tbolt_intr_ack status = %d \n", status));

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

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

2201         if ((status & 1) || (status & MFI_FUSION_ENABLE_INTERRUPT_MASK)) {
```



```
2202                 /* clear the interrupt by writing back the same value */
2203                 WR_OB_INTR_STATUS(status, instance);
2204                 /* dummy READ */
2205                 (void) RD_OB_INTR_STATUS(instance);
2194                 RD_OB_INTR_STATUS(instance);
2206         }
2207         return (DDI_INTR_CLAIMED);
2208 }
```

_____***unchanged_portion_omitted_***

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

2317         if (!instance->tbolt) {
2318                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled."));
2305                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled.\n"));
2319                 return;
2320         }

2322         index = cmd->index;

2324         ReqDescUnion = mr_sas_get_request_descriptor(instance, index);
2311         ReqDescUnion =
2312             mr_sas_get_request_descriptor(instance, index, cmd);

2326         if (!ReqDescUnion) {
2327                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]"));
2315                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]%x"));
2328                 return;
2329         }

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

2333         ReqDescUnion->Words = 0;

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

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

2341         cmd->request_desc = ReqDescUnion;

2343         /* get raid message frame pointer */
2331         // get raid message frame pointer
2344         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2346         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
2347                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)
2348                     &scsi_raid_io->SGL.IeeeChain;
2335                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)&s
2349                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;
2350                 ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
2337                 ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2338                     &sgl_ptr_end->Flags, 0);
2351         }
```

```
2353         ddi_put8(acc_handle, &scsi_raid_io->Function,
2341         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2342             &scsi_raid_io->Function,
2354             MPI2_FUNCTION_PASSTHRU_IO_REQUEST);

2356         ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
2345         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2346             &scsi_raid_io->SGLOffset0,
2357             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

2359         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
2349         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2350             &scsi_raid_io->ChainOffset,
2360             (U8)offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16);

2362         ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
2353         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2354             &scsi_raid_io->SenseBufferLowAddress,
2363             cmd->sense_phys_addr1);


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

2369         ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
2361         ddi_put64(instance->mpi2_frame_pool_dma_obj.acc_handle,
2362             &scsi_raid_io_sgl_ieee->Address,
2370             (U64)cmd->frame_phys_addr);

2372         ddi_put8(acc_handle,
2373             &scsi_raid_io_sgl_ieee->Flags, (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2365         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2366             &scsi_raid_io_sgl_ieee->Flags,
2367             (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2374             MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
2375         /* LSI put hardcoded 1024 instead of MEGASAS_MAX_SZ_CHAIN_FRAME. */
2376         ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length, 1024);
2369         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2370             &scsi_raid_io_sgl_ieee->Length, 1024); //MEGASAS_MAX_SZ_CHAIN_FRAME

2378         con_log(CL_ANN1, (CE_NOTE,
2379             "[MFI CMD PHY ADDRESS]:%" PRIx64,
2373             "[MFI CMD PHY ADDRESS]:%x",
2380             scsi_raid_io_sgl_ieee->Address));
2381         con_log(CL_ANN1, (CE_NOTE,
2382             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
2383         con_log(CL_ANN1, (CE_NOTE, "[SGL Flags]:%x",
2384             scsi_raid_io_sgl_ieee->Flags));
2385 }


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



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

2405         status = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.status);
2406         extStatus = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.extStatus);
2398         status = ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2399             &scsi_raid_io->RaidContext.status);
2400         extStatus = ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2401             &scsi_raid_io->RaidContext.extStatus);

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

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

2419         /* regular commands */

2421         switch (ddi_get8(acc_handle, &scsi_raid_io->Function)) {
2416         switch (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
2417             &scsi_raid_io->Function)) {

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

2427                 if (cmd->load_balance_flag & MEGASAS_LOAD_BALANCE_FLAG) {
2428                         arm = lbinfo->raid1DevHandle[0] ==
2429                             scsi_raid_io->DevHandle ? 0 : 1;
2424                             arm = lbinfo->raid1DevHandle[0] == scsi_raid_io->

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

2438         case MPI2_FUNCTION_LD_IO_REQUEST :   { /* Regular Path IO. */
2432                 case MPI2_FUNCTION_LD_IO_REQUEST :   {// Regular Path IO.
2439                 acmd =  (struct scsa_cmd *)cmd->cmd;
2440                 pkt =   (struct scsi_pkt *)CMD2PKT(acmd);

2442                 if (acmd->cmd_flags & CFLAG_DMAVALID) {
2443                         if (acmd->cmd_flags & CFLAG_CONSISTENT) {
2444                                 (void) ddi_dma_sync(acmd->cmd_dmahandle,
2445                                     acmd->cmd_dma_offset, acmd->cmd_dma_len,
2439                                         acmd->cmd_dma_offset,
2440                                         acmd->cmd_dma_len,
2446                                     DDI_DMA_SYNC_FORCPU);
2447                         }
2448                 }

2450                 pkt->pkt_reason         = CMD_CMPLT;
2451                 pkt->pkt_statistics     = 0;
2452                 pkt->pkt_state = STATE_GOT_BUS | STATE_GOT_TARGET |
2453                     STATE_SENT_CMD | STATE_XFERRED_DATA | STATE_GOT_STATUS;
2447                     pkt->pkt_state = STATE_GOT_BUS
2448                         | STATE_GOT_TARGET | STATE_SENT_CMD
2449                         | STATE_XFERRED_DATA | STATE_GOT_STATUS;
```

```
2455                 con_log(CL_ANN, (CE_CONT, " CDB[0] = %x completed for %s: "
2456                     "size %lx SMID %x cmd_status %x", pkt->pkt_cdbp[0],
2451                         con_log(CL_ANN, (CE_CONT,
2452                             " CDB[0] = %x completed for %s: size %lx SMID %x cmd
2453                             pkt->pkt_cdbp[0],
2457                     ((acmd->islogical) ? "LD" : "PD"),
2458                     acmd->cmd_dmacount, cmd->SMID, status));

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

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

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

2476                                                 display_scsi_inquiry(
2477                                                     (caddr_t)inq);
2478                                         }
2476 #endif
2477                                 } else {
2480                                         else {
2478                                         /* for physical disk */
2479                                         status = MFI_STAT_DEVICE_NOT_FOUND;
2482                                                 status =
2483                                                     MFI_STAT_DEVICE_NOT_FOUND;
2480                                 }
2481                         }
2482                 }

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

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



```
2508                                 ((struct scsi_status *)
2509                                     pkt->pkt_scbp)->sts_chk = 1;

2506                         if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
2507                                 con_log(CL_ANN,
2508                                     (CE_WARN, "TEST_UNIT_READY fail"));
2512                                     con_log(CL_ANN, (CE_WARN, "TEST_UNIT_REA
2509                         } else {
2510                                 pkt->pkt_state |= STATE_ARQ_DONE;
2511                                 arqstat = (void *)(pkt->pkt_scbp);
2512                                 arqstat->sts_rqpkt_reason = CMD_CMPLT;
2513                                 arqstat->sts_rqpkt_resid = 0;
2514                                 arqstat->sts_rqpkt_state |=
2515                                     STATE_GOT_BUS | STATE_GOT_TARGET
2516                                     | STATE_SENT_CMD
2517                                     | STATE_XFERRED_DATA;
2518                                 *(uint8_t *)&arqstat->sts_rqpkt_status =
2519                                     STATUS_GOOD;
2520                                 con_log(CL_ANN1,
2521                                     (CE_NOTE, "Copying Sense data %x",
2524                                         con_log(CL_ANN1, (CE_NOTE,
2525                                             "Copying Sense data %x",
2522                                         cmd->SMID));

2524                                 ddi_rep_get8(acc_handle,
2525                                     (uint8_t *)&(arqstat->sts_sensedata),
2528                                         ddi_rep_get8(
2529                                             instance->
2530                                             mpi2_frame_pool_dma_obj.acc_handle,
2531                                             (uint8_t *)
2532                                             &(arqstat->sts_sensedata),
2526                                         cmd->sense1,
2527                                         sizeof (struct scsi_extended_sense),
2528                                         DDI_DEV_AUTOINCR);

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

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

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

2542                                     "CDB[0]=0x%x targetId=0x%x devhandle=0x%
2543                                     ddi_get8(instance->mpi2_frame_pool_dma_o
2544                                     ddi_get16(instance->mpi2_frame_pool_dma_
2545                                     ddi_get16(instance->mpi2_frame_pool_dma_
2544                         pkt->pkt_reason = CMD_DEV_GONE;
2545                         pkt->pkt_statistics  = STAT_DISCON;
2546                         break;
2547                 case MFI_STAT_DEVICE_NOT_FOUND:
2548                         con_log(CL_ANN, (CE_CONT,
2549                             "tbolt_complete_cmd: device not found error"));
2550                         pkt->pkt_reason = CMD_DEV_GONE;
2551                         pkt->pkt_statistics  = STAT_DISCON;
2552                         break;

2554                 case MFI_STAT_LD_LBA_OUT_OF_RANGE:
2555                         pkt->pkt_state |= STATE_ARQ_DONE;
2556                         pkt->pkt_reason = CMD_CMPLT;
```

```
                        ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;
                                ((struct scsi_status *)
                                    pkt->pkt_scbp)->sts_chk = 1;

                        arqstat = (void *)(pkt->pkt_scbp);
                        arqstat->sts_rqpkt_reason = CMD_CMPLT;
                        arqstat->sts_rqpkt_resid = 0;
                        arqstat->sts_rqpkt_state |= STATE_GOT_BUS
                            | STATE_GOT_TARGET | STATE_SENT_CMD
                            | STATE_XFERRED_DATA;
                        *(uint8_t *)&arqstat->sts_rqpkt_status = STATUS_GOOD;
                                *(uint8_t *)&arqstat->sts_rqpkt_status =
                                    STATUS_GOOD;

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

                        /*
                         * LOGICAL BLOCK ADDRESS OUT OF RANGE:
                         * ASC: 0x21h; ASCQ: 0x00h;
                         */
                        arqstat->sts_sensedata.es_add_code = 0x21;
                        arqstat->sts_sensedata.es_qual_code = 0x00;
                        break;
                case MFI_STAT_INVALID_CMD:
                case MFI_STAT_INVALID_DCMD:
                case MFI_STAT_INVALID_PARAMETER:
                case MFI_STAT_INVALID_SEQUENCE_NUMBER:
                default:
                        cmn_err(CE_WARN, "tbolt_complete_cmd: Unknown status!");
                        pkt->pkt_reason = CMD_TRAN_ERR;

                        break;
                }

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

                (void) mrsas_common_check(instance, cmd);
                if (acmd->cmd_dmahandle) {
                        if (mrsas_check_dma_handle(acmd->cmd_dmahandle) !=
                            DDI_SUCCESS) {
                                ddi_fm_service_impact(instance->dip,
                                    DDI_SERVICE_UNAFFECTED);
                                pkt->pkt_reason = CMD_TRAN_ERR;
                                pkt->pkt_statistics = 0;
                        }
                }

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

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

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



```
                ddi_put8(acc_handle, &scsi_raid_io->RaidContext.extStatus, 0);
                        ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
                            &scsi_raid_io->RaidContext.extStatus, 0);

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

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

                        mutex_enter(&instance->sync_map_mtx);

                        con_log(CL_ANN, (CE_NOTE,
                            "LDMAP sync command SMID RECEIVED 0x%X",
                            cmd->SMID));
                        if (cmd->frame->hdr.cmd_status != 0) {
                                cmn_err(CE_WARN,
                                    "map sync failed, status = 0x%x.",
                                    cmd->frame->hdr.cmd_status);
                        } else {
                                    "map sync failed, status = 0x%x.\n",cmd-
                        }
                        else {
                                instance->map_id++;
                                cmn_err(CE_NOTE,
                                    "map sync received, switched map_id to %"
                                    PRIu64 " \n", instance->map_id);
                                    "map sync received, switched map_id to %
                        }

                        if (MR_ValidateMapInfo(instance->ld_map[
                            (instance->map_id & 1)],
                            instance->load_balance_info)) {
                        if (MR_ValidateMapInfo(instance->ld_map[(instance->map_i
                                instance->fast_path_io = 1;
                        } else {
                        else
                                instance->fast_path_io = 0;
                        }

                        con_log(CL_ANN, (CE_NOTE,
                            "instance->fast_path_io %d",
                            instance->fast_path_io));
                                "instance->fast_path_io %d \n",instance->fast_pa

                        instance->unroll.syncCmd = 0;

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

                        cmn_err(CE_NOTE, "LDMAP sync completed.");
                        cmn_err(CE_NOTE, "LDMAP sync completed.\n");
                        mutex_exit(&instance->sync_map_mtx);
                        break;
```

```
2659                }

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

2675                if (cmd->sync_cmd == MRSAS_TRUE) {
2674                if (cmd->sync_cmd == MRSAS_TRUE ) {
2676                        con_log(CL_ANN1, (CE_CONT,
2677                            "Sync-mode Command Response SMID RECEIVED 0x%X",
2678                            cmd->SMID));

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

2691                /* free message */
2692                con_log(CL_ANN,
2693                    (CE_NOTE, "tbolt_complete_cmd: Unknown Type!!!!!!!!"));
2690                con_log(CL_ANN, (CE_NOTE, "tbolt_complete_cmd: Unknown Type!!!!!
2694                break;
2695        }
2696 }

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

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

2711        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
```



```
2712            0, 0, DDI_DMA_SYNC_FORDEV);

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

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

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

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

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

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

2740        d_val.word = desc->Words;


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

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

2750                smid = replyDesc->SMID;

2752                if (!smid || smid > instance->max_fw_cmds + 1) {
2753                        con_log(CL_ANN1, (CE_NOTE,
2754                            "Reply Desc at Break  = %p  Words = %" PRIx64,
2755                            (void *)desc, desc->Words));
2744                        "Reply Desc at Break  = %llx  Words = %llx \n",
2745                        desc, desc->Words));
2756                        break;
2757                }

2759                cmd     = instance->cmd_list[smid - 1];
2760                if (!cmd) {
2761                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_tbolt_process_"
2762                            "outstanding_cmd: Invalid command "
2763                            " or Poll commad Received in completion path"));
2764                } else {
2750                if(!cmd ) {
2751                        con_log(CL_ANN1, (CE_NOTE,
2752                                "mr_sas_tbolt_process_outstanding_cmd: Invalid c
2753                                " or Poll commad Received in completion path\n")
2754                }
2755                else {
2765                        mutex_enter(&instance->cmd_pend_mtx);
2766                        if (cmd->sync_cmd == MRSAS_TRUE) {
2767                                hdr = (struct mrsas_header *)&cmd->frame->hdr;
```

```
2768                                if (hdr) {
2769                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2770                                            "tbolt_process_outstanding_cmd:"
2771                                            " mlist_del_init(&cmd->list)."));
2760                                        con_log(CL_ANN1, (CE_NOTE,
2761                                                "mr_sas_tbolt_process_outstandin
2762                                                " mlist_del_init(&cmd->list).\n"
2772                                        mlist_del_init(&cmd->list);
2773                                }
2774                        } else {
2775                                pkt = cmd->pkt;
2776                                if (pkt) {
2777                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2778                                            "tbolt_process_outstanding_cmd:"
2779                                            "mlist_del_init(&cmd->list)."));
2768                                            con_log(CL_ANN1, (CE_NOTE,
2769                                                    "mr_sas_tbolt_process_ou
2770                                                    "mlist_del_init(&cmd->li
2780                                        mlist_del_init(&cmd->list);
2781                                }
2782                        }

2784                        mutex_exit(&instance->cmd_pend_mtx);

2786                        tbolt_complete_cmd(instance, cmd);
2787                }
2788                /* set it back to all 1s. */
2789                desc->Words = -1LL;
2779                // set it back to all 0xffffffff.
2780                desc->Words = (uint64_t)~0;

2791                instance->reply_read_index++;

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

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

2804                replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;

2806                d_val.word = desc->Words;

2808                con_log(CL_ANN1, (CE_NOTE,
2809                    "Next Reply Desc  = %p Words = %" PRIx64,
2810                    (void *)desc, desc->Words));
2800                    "Next Reply Desc  = %llx Words = %llx\n",
2801                    desc, desc->Words));

2812                replyType = replyDesc->ReplyFlags &
2813                    MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2815                if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2816                        break;

2818        } /* End of while loop. */

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



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

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

2863 /*
2864  * mrsas_tbolt_get_ld_map_info -        Returns  ld_map structure
2865  * instance:                            Adapter soft state
2866  *
2867  * Issues an internal command (DCMD) to get the FW's controller PD
2868  * list structure.  This information is mainly used to find out SYSTEM
2869  * supported by the FW.
2870  */
2871 int
2872 mrsas_tbolt_get_ld_map_info(struct mrsas_instance *instance)
2873 {
2874        int ret = 0;
2875        struct mrsas_cmd        *cmd = NULL;
2876        struct mrsas_dcmd_frame *dcmd;
2877        MR_FW_RAID_MAP_ALL *ci;
2878        uint32_t ci_h = 0;
2879        U32 size_map_info;

2881        cmd = get_raid_msg_pkt(instance);

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

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

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

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

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

2901        if (!ci) {
2902                cmn_err(CE_WARN, "Failed to alloc mem for ld_map_info");
2893                cmn_err(CE_WARN,
2894                    "Failed to alloc mem for ld_map_info");
2903                return_raid_msg_pkt(instance, cmd);
2904                return (-1);
2905        }

2907        bzero(ci, sizeof (*ci));
2908        bzero(dcmd->mbox.b, DCMD_MBOX_SZ);
2899        memset(ci, 0, sizeof (*ci));
2900        memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

2910        dcmd->cmd = MFI_CMD_OP_DCMD;
2911        dcmd->cmd_status = 0xFF;
2912        dcmd->sge_count = 1;
2913        dcmd->flags = MFI_FRAME_DIR_READ;
2914        dcmd->timeout = 0;
2915        dcmd->pad_0 = 0;
```

```
2916         dcmd->data_xfer_len = size_map_info;
2917         dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
2918         dcmd->sgl.sge32[0].phys_addr = ci_h;
2919         dcmd->sgl.sge32[0].length = size_map_info;


2922         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

2924         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
2925                 ret = 0;
2926                 con_log(CL_ANN1, (CE_NOTE, "Get LD Map Info success"));
2918                 con_log(CL_ANN1, (CE_NOTE,
2919                     "Get LD Map Info success\n"));
2927         } else {
2928                 cmn_err(CE_WARN, "Get LD Map Info failed");
2921                 cmn_err(CE_WARN,
2922                     "Get LD Map Info failed\n");
2929                 ret = -1;
2930         }

2932         return_raid_msg_pkt(instance, cmd);

2934         return (ret);
2935 }
______unchanged_portion_omitted_

2954 /*
2948 /**
2955  * mrsas_tbolt_command_create - Create command for fast path.
2956  * @io_info:    MegaRAID IO request packet pointer.
2957  * @ref_tag:    Reference tag for RD/WRPROTECT
2958  *
2959  * Create the command for fast path.
2960  */
2961 void
2962 mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2963     struct IO_REQUEST_INFO *io_info, Mpi2RaidSCSIIORequest_t *scsi_io_request,
2964     U32 ref_tag)
2956 mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],struct IO_REQU
2965 {
2966         uint16_t                EEDPFlags;
2967         uint32_t                Control;
2968         ddi_acc_handle_t acc_handle =
2969             instance->mpi2_frame_pool_dma_obj.acc_handle;
2960         // Prepare 32-byte CDB if DIF is supported on this device
2961         con_log(CL_ANN, (CE_NOTE, "Prepare DIF CDB\n"));

2971         /* Prepare 32-byte CDB if DIF is supported on this device */
2972         con_log(CL_ANN, (CE_NOTE, "Prepare DIF CDB"));
2963         memset(cdb, 0, 32);

2974         bzero(cdb, 32);

2976         cdb[0] =  MRSAS_SCSI_VARIABLE_LENGTH_CMD;


2979         cdb[7] =  MRSAS_SCSI_ADDL_CDB_LEN;

2981         if (io_info->isRead)
2970         if (io_info->isRead) {
2982                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_READ32;
2983         else
2972         }
2973         else {
2984                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_WRITE32;
2975         }
```



```
2986         /* Verify within linux driver, set to MEGASAS_RD_WR_PROTECT_CHECK_ALL */
2987         cdb[10] = MRSAS_RD_WR_PROTECT;
2977         cdb[10] = MRSAS_RD_WR_PROTECT;  // Verify with in linux driver, set to M

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

2999         /* Logical block reference tag */
3000         ddi_put32(acc_handle, &scsi_io_request->CDB.EEDP32.PrimaryReferenceTag,
3001             BE_32(ref_tag));
2990         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2991                         &scsi_io_request->CDB.EEDP32.PrimaryReferenceTag,
2992                                                 BIG_ENDIAN(ref_tag));

3003         ddi_put16(acc_handle,
3004             &scsi_io_request->CDB.EEDP32.PrimaryApplicationTagMask, 0xffff);
2994         ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
2995             &scsi_io_request->CDB.EEDP32.PrimaryApplicationTagMask,
2996             0xffff);

3006         ddi_put32(acc_handle, &scsi_io_request->DataLength,
2998         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
2999             &scsi_io_request->DataLength,
3007             ((io_info->numBlocks)*512));
3008         /* Specify 32-byte cdb */
3009         ddi_put16(acc_handle, &scsi_io_request->IoFlags, 32);
3001         ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
3002                 &scsi_io_request->IoFlags,32);  /* Specify 32-byte cdb */

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

3017         /* set SCSI IO EEDPFlags */
3018         EEDPFlags = ddi_get16(acc_handle, &scsi_io_request->EEDPFlags);
3019         Control = ddi_get32(acc_handle, &scsi_io_request->Control);
3011         EEDPFlags = ddi_get16(instance->mpi2_frame_pool_dma_obj.acc_handle,
3012                                                 &scsi_io_request->EEDPFl
3013         Control = ddi_get32(instance->mpi2_frame_pool_dma_obj.acc_handle,
3014                                                 &scsi_io_request->Contro

3021         /* set SCSI IO EEDPFlags bits */
3016         // set SCSI IO EEDPFlags bits
3022         if (io_info->isRead) {
3023                 /*
3024                  * For READ commands, the EEDPFlags shall be set to specify to
3025                  * Increment the Primary Reference Tag, to Check the Reference
3026                  * Tag, and to Check and Remove the Protection Information
3027                  * fields.
3028                  */
3018                 // For READ commands, the EEDPFlags shall be set to specify to
3019                 // Increment the Primary Reference Tag, to Check the Reference
3020                 // Tag, and to Check and Remove the Protection Information
3021                 // fields.
3029                 EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3030                     MPI2_SCSIIO_EEDPFLAGS_CHECK_REFTAG  |
```

```
3031                    MPI2_SCSIIO_EEDPFLAGS_CHECK_REMOVE_OP       |
3032                    MPI2_SCSIIO_EEDPFLAGS_CHECK_APPTAG  |
3033                    MPI2_SCSIIO_EEDPFLAGS_CHECK_GUARD;
3034            } else {
3035                    /*
3036                     * For WRITE commands, the EEDPFlags shall be set to specify to
3037                     * Increment the Primary Reference Tag, and to Insert
3038                     * Protection Information fields.
3039                     */
3027            }
3028            else {
3029                    // For WRITE commands, the EEDPFlags shall be set to specify to
3030                    // Increment the Primary Reference Tag, and to Insert
3031                    // Protection Information fields.
3040                    EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3041                        MPI2_SCSIIO_EEDPFLAGS_INSERT_OP;
3042            }
3043            Control |= (0x4 << 26);

3045            ddi_put16(acc_handle, &scsi_io_request->EEDPFlags, EEDPFlags);
3046            ddi_put32(acc_handle, &scsi_io_request->Control, Control);
3047            ddi_put32(acc_handle,
3048                &scsi_io_request->EEDPBlockSize, MRSAS_EEDPBLOCKSIZE);
3037            ddi_put16(instance->mpi2_frame_pool_dma_obj.acc_handle,
3038                        &scsi_io_request->EEDPFlags, EEDPFlags);
3039            ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
3040                        &scsi_io_request->Control, Control);
3041            ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
3042                &scsi_io_request->EEDPBlockSize,
3043                MRSAS_EEDPBLOCKSIZE);
3049 }


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

3067            /* Some drives don't support 16/12 byte CDB's, convert to 10 */
3068            if (((cdb_len == 12) || (cdb_len == 16)) &&
3069                (start_blk <= 0xffffffff)) {
3070                    if (cdb_len == 16) {
3071                            con_log(CL_ANN,
3072                                (CE_NOTE, "Converting READ/WRITE(16) to READ10"));
3065                            con_log(CL_ANN, (CE_NOTE, "Converting READ/WRITE(16) to
3073                            opcode = cdb[0] == READ_16 ? READ_10 : WRITE_10;
3074                            flagvals = cdb[1];
3075                            groupnum = cdb[14];
3076                            control = cdb[15];
3077                    } else {
3078                            con_log(CL_ANN,
3079                                (CE_NOTE, "Converting READ/WRITE(12) to READ10"));
3071                            con_log(CL_ANN, (CE_NOTE, "Converting READ/WRITE(12) to
3080                            opcode = cdb[0] == READ_12 ? READ_10 : WRITE_10;
```



```
3081                            flagvals = cdb[1];
3082                            groupnum = cdb[10];
3083                            control = cdb[11];
3084                    }

3086                    bzero(cdb, sizeof (cdb));
3078                    memset(cdb, 0, sizeof(cdb));

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

3119                    bzero(cdb, sizeof (cdb));
3110                            memset(cdb, 0, sizeof(cdb));

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

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

3132                    /* Specify 16-byte cdb */
3133                    cdb_len = 16;
3134            } else if ((cdb_len == 6) && (start_blk > 0x1fffff)) {
3135                    /* convert to 10 byte CDB */
3136                    opcode = cdb[0] == READ_6 ? READ_10 : WRITE_10;
3137                    control = cdb[5];

3139                    bzero(cdb, sizeof (cdb));
3130                                memset(cdb, 0, sizeof(cdb));
3140                    cdb[0] = opcode;
3141                    cdb[9] = control;
```

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

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


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

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

3187         *cdb_len_ptr = cdb_len;
3188 }


3191 static int
3182 U8
3192 mrsas_tbolt_check_map_info(struct mrsas_instance *instance)
3193 {
3194         MR_FW_RAID_MAP_ALL *ld_map;

3196         if (!mrsas_tbolt_get_ld_map_info(instance)) {

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

3200                 con_log(CL_ANN1, (CE_NOTE, "ldCount=%d, map size=%d",
3191                 con_log(CL_ANN1, (CE_NOTE, "ldCount=%d, map size=%d\n",
3201                     ld_map->raidMap.ldCount, ld_map->raidMap.totalSize));

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



```
3194                 if (MR_ValidateMapInfo(instance->ld_map[(instance->map_id & 1)],
3195                         con_log(CL_ANN, (CE_CONT,
3196                                 "MR_ValidateMapInfo success"));

3208                         instance->fast_path_io = 1;
3209                         con_log(CL_ANN,
3210                             (CE_NOTE, "instance->fast_path_io %d",
3211                             instance->fast_path_io));
3199                         con_log(CL_ANN, (CE_NOTE,
3200                                 "instance->fast_path_io %d \n",instance->fast_pa

3213                         return (DDI_SUCCESS);
3214                 }

3216         }

3218         instance->fast_path_io = 0;
3219         cmn_err(CE_WARN, "MR_ValidateMapInfo failed");
3220         con_log(CL_ANN, (CE_NOTE,
3221             "instance->fast_path_io %d", instance->fast_path_io));
3210             "instance->fast_path_io %d \n",instance->fast_path_io));

3212
3223         return (DDI_FAILURE);
3224 }

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

3232 void
3221 int
3233 mrsas_tbolt_kill_adapter(struct mrsas_instance *instance)
3234 {
3235         cmn_err(CE_NOTE, "TBOLT Kill adapter called");
3224         cmn_err(CE_WARN, "TBOLT Kill adapter called\n");

3237         if (instance->deadadapter == 1)
3238                 return;
3227                 return (DDI_FAILURE);

3240         con_log(CL_ANN1, (CE_NOTE, "tbolt_kill_adapter: "
3241             "Writing to doorbell with MFI_STOP_ADP "));
3242         mutex_enter(&instance->ocr_flags_mtx);
3243         instance->deadadapter = 1;
3244         mutex_exit(&instance->ocr_flags_mtx);
3245         instance->func_ptr->disable_intr(instance);
3246         WR_RESERVED0_REGISTER(MFI_STOP_ADP, instance);
3247         /* Flush */
3248         (void) RD_RESERVED0_REGISTER(instance);
3237         RD_RESERVED0_REGISTER(instance);

3250         (void) mrsas_print_pending_cmds(instance);
3251         (void) mrsas_complete_pending_cmds(instance);
3240         mrsas_complete_pending_cmds(instance);
3241         return (DDI_SUCCESS);
3252 }

3254 void
3255 mrsas_reset_reply_desc(struct mrsas_instance *instance)
3243 void  mrsas_reset_reply_desc(struct mrsas_instance *instance)
3256 {
3257         int i;
3258         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
```

```
3259         instance->reply_read_index = 0;
3247         instance->reply_read_index= 0;

3261         /* initializing reply address to 0xFFFFFFFF */
3262         reply_desc = instance->reply_frame_pool;

3264         for (i = 0; i < instance->reply_q_depth; i++) {
3265                 reply_desc->Words = (uint64_t)~0;
3266                 reply_desc++;
3267         }
3268 }

3270 int
3271 mrsas_tbolt_reset_ppc(struct mrsas_instance *instance)
3272 {
3273         uint32_t status = 0x00;
3261         uint32_t status=0x00;
3274         uint32_t retry = 0;
3263         uint32_t seq_num;
3275         uint32_t cur_abs_reg_val;
3276         uint32_t fw_state;
3266         union mrsas_evt_class_locale    class_locale;
3277         uint32_t abs_state;
3278         uint32_t i;

3280         con_log(CL_ANN, (CE_NOTE,
3281             "mrsas_tbolt_reset_ppc entered"));
3271             "mrsas_tbolt_reset_ppc entered\n "));

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

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

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

3297         /* Add delay inorder to complete the ioctl & io cmds in-flight */
3298         for (i = 0; i < 3000; i++) {
3287         /*Add delay inorder to complete the ioctl & io cmds in-flight */
3288         for (i = 0; i<3000; i++) {
3299                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
3300         }

3302         instance->reply_read_index = 0;
3292         instance->reply_read_index= 0;

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

3308         WR_TBOLT_IB_WRITE_SEQ(0xF, instance);
3309         WR_TBOLT_IB_WRITE_SEQ(4, instance);
3310         WR_TBOLT_IB_WRITE_SEQ(0xb, instance);
3311         WR_TBOLT_IB_WRITE_SEQ(2, instance);
3312         WR_TBOLT_IB_WRITE_SEQ(7, instance);
3313         WR_TBOLT_IB_WRITE_SEQ(0xd, instance);
3314         con_log(CL_ANN1, (CE_NOTE,
3315             "mrsas_tbolt_reset_ppc: magic number written "
3316             "to write sequence register"));
```



```
3306             "to write sequence register\n"));
3317         delay(100 * drv_usectohz(MILLISEC));
3318         status = RD_TBOLT_HOST_DIAG(instance);
3319         con_log(CL_ANN1, (CE_NOTE,
3320             "mrsas_tbolt_reset_ppc: READ HOSTDIAG SUCCESS "
3321             "to write sequence register"));
3311             "to write sequence register\n"));

3323         while (status & DIAG_TBOLT_RESET_ADAPTER) {
3324                 delay(100 * drv_usectohz(MILLISEC));
3325                 status = RD_TBOLT_HOST_DIAG(instance);
3326                 if (retry++ == 100) {
3327                         cmn_err(CE_WARN,
3328                             "mrsas_tbolt_reset_ppc:"
3329                             "resetadapter bit is set already "
3330                             "check retry count %d", retry);
3320                             "check retry count %d\n", retry);
3331                         return (DDI_FAILURE);
3332                 }
3333         }

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

3338         ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3339             (uint8_t *)((uintptr_t)(instance)->regmap +
3340             RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);
3330                 RESET_TBOLT_STATUS_OFF),4,DDI_DEV_AUTOINCR);

3342         while ((status & DIAG_TBOLT_RESET_ADAPTER)) {
3343                 delay(100 * drv_usectohz(MILLISEC));
3344                 ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3345                     (uint8_t *)((uintptr_t)(instance)->regmap +
3346                     RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);
3336                     RESET_TBOLT_STATUS_OFF),4,DDI_DEV_AUTOINCR);
3347                 if (retry++ == 100) {
3348                         /* Dont call kill adapter here */
3349                         /* RESET BIT ADAPTER is cleared by firmare */
3350                         /* mrsas_tbolt_kill_adapter(instance); */
3351                         cmn_err(CE_WARN,
3352                             "mr_sas %d: %s(): RESET FAILED; return failure!!!",
3353                             instance->instance, __func__);
3340                         //mrsas_tbolt_kill_adapter(instance);
3341                         cmn_err(CE_WARN, "mr_sas %d: %s(): RESET FAILED; return
3354                         return (DDI_FAILURE);
3355                 }
3356         }

3358         con_log(CL_ANN,
3359             (CE_NOTE, "mrsas_tbolt_reset_ppc: Adapter reset complete"));
3346         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: Adapter reset complete
3360         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3361             "Calling mfi_state_transition_to_ready"));

3363         abs_state = instance->func_ptr->read_fw_status_reg(instance);
3364         retry = 0;
3365         while ((abs_state <= MFI_STATE_FW_INIT) && (retry++ < 1000)) {
3366                 delay(100 * drv_usectohz(MILLISEC));
3367                 abs_state = instance->func_ptr->read_fw_status_reg(instance);
3368         }
3369         if (abs_state <= MFI_STATE_FW_INIT) {
3370                 cmn_err(CE_WARN,
3371                     "mrsas_tbolt_reset_ppc: firmware state < MFI_STATE_FW_INIT"
3372                     "state = 0x%x, RETRY RESET.", abs_state);
3357                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: firmware state < MFI_ST
3358                     "state = 0x%x, RETRY RESET.\n", abs_state);
```

```
3373                goto retry_reset;
3374        }

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

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

3389                con_log(CL_ANN,
3390                    (CE_NOTE, "mrsas_tbolt_reset_ppc : FW is not ready "
3375                con_log(CL_ANN, (CE_NOTE,  "mrsas_tbolt_reset_ppc : FW is not re
3391                    "FW state = 0x%x", fw_state));

3393                if (fw_state == MFI_STATE_FAULT) {
3394                        /* increment the count */
3379                        // increment the count
3395                        instance->fw_fault_count_after_ocr++;
3396                        if (instance->fw_fault_count_after_ocr
3397                            < MAX_FW_RESET_COUNT) {
3398                                cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3399                                    "FW is in fault after OCR count %d "
3400                                    "Retry Reset",
3401                                    instance->fw_fault_count_after_ocr);
3402                                goto retry_reset;

3404                        } else {
3405                                cmn_err(CE_WARN, "mrsas %d: %s:"
3406                                    "Max Reset Count exceeded >%d"
3407                                    "Mark HBA as bad, KILL adapter",
3408                                    instance->instance, __func__,
3409                                    MAX_FW_RESET_COUNT);
3393                                    instance->instance, __func__, MAX_FW_RES

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

3417        /* reset the counter as FW is up after OCR */
3401        // reset the counter as FW is up after OCR
3418        instance->fw_fault_count_after_ocr = 0;

3420        mrsas_reset_reply_desc(instance);


3423        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3424            "Calling mrsas_issue_init_mpi2"));
3425        abs_state = mrsas_issue_init_mpi2(instance);
3426        if (abs_state == (uint32_t)DDI_FAILURE) {
3410        if(abs_state == DDI_FAILURE) {
3427                cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3428                    "INIT failed Retrying Reset");
3429                goto retry_reset;
3430        }
3431        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3432            "mrsas_issue_init_mpi2 Done"));
```



```
3434        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3435            "Calling mrsas_print_pending_cmd"));
3436        (void) mrsas_print_pending_cmds(instance);
3419            "Calling mrsas_print_pending_cmd\n"));
3420        mrsas_print_pending_cmds(instance);
3437        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3438            "mrsas_print_pending_cmd done"));
3422            "mrsas_print_pending_cmd done\n"));

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

3443        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3444            "Calling mrsas_issue_pending_cmds"));
3445        (void) mrsas_issue_pending_cmds(instance);
3429        mrsas_issue_pending_cmds(instance);
3446        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3447        "issue_pending_cmds done."));
3431        "issue_pending_cmds done.\n"));

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

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

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

3457        con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag."));
3441        con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag.\n"));
3458        mutex_enter(&instance->ocr_flags_mtx);
3459        instance->adapterresetinprogress = 0;
3460        mutex_exit(&instance->ocr_flags_mtx);
3461        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3462            "adpterresetinprogress flag unset"));

3464        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc done"));
3448        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc done\n"));
3465        return (DDI_SUCCESS);

3467 }


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

3479 static int
3463 int
3480 mrsas_tbolt_sync_map_info(struct mrsas_instance *instance)
3481 {
3482        int                     ret = 0, i;
3483        struct mrsas_cmd        *cmd = NULL;
3484        struct mrsas_dcmd_frame *dcmd;
3485        uint32_t size_sync_info, num_lds;
3486        LD_TARGET_SYNC *ci = NULL;
3487        MR_FW_RAID_MAP_ALL *map;
3488        MR_LD_RAID  *raid;
3489        LD_TARGET_SYNC *ld_sync;
3490        uint32_t ci_h = 0;
3491        uint32_t size_map_info;
```

```
3493         cmd = get_raid_msg_pkt(instance);

3495         if (cmd == NULL) {
3496                 cmn_err(CE_WARN, "Failed to get a cmd from free-pool in "
3497                     "mrsas_tbolt_sync_map_info(). ");
3480                 cmn_err(CE_WARN,
3481                     "Failed to get a cmd from free-pool in mrsas_tbolt_sync_map_
3498                 return (DDI_FAILURE);
3499         }

3501         /* Clear the frame buffer and assign back the context id */
3502         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3486         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3503         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3504             cmd->index);
3505         bzero(cmd->scsi_io_request, sizeof (Mpi2RaidSCSIIORequest_t));


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

3510         num_lds = map->raidMap.ldCount;

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

3514         size_sync_info = sizeof (LD_TARGET_SYNC) * num_lds;

3516         con_log(CL_ANN, (CE_NOTE, "size_sync_info =0x%x ; ld count = 0x%x",
3500         con_log(CL_ANN, (CE_NOTE,
3501                     "size_sync_info =0x%x ; ld count = 0x%x \n ",
3517             size_sync_info, num_lds));

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

3521         bzero(ci, sizeof (MR_FW_RAID_MAP_ALL));
3506         memset(ci, 0, sizeof(MR_FW_RAID_MAP_ALL));
3522         ci_h = instance->ld_map_phy[(instance->map_id - 1) & 1];

3524         bzero(dcmd->mbox.b, DCMD_MBOX_SZ);
3509         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

3526         ld_sync = (LD_TARGET_SYNC *)ci;

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

3531                 con_log(CL_ANN1,
3532                     (CE_NOTE, "i : 0x%x, Seq Num : 0x%x, Sync Reqd : 0x%x",
3516                 con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, Seq Num : 0x%x, Sync Reqd
3533                     i, raid->seqNum, raid->flags.ldSyncRequired));

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

3537                 con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, tgt : 0x%x",
3521                 con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, tgt : 0x%x \n",
3538                     i, ld_sync->ldTargetId));

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


3544         size_map_info = sizeof (MR_FW_RAID_MAP) +
3545             (sizeof (MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));
3528         size_map_info = sizeof(MR_FW_RAID_MAP) +
3529                 (sizeof(MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));
```



```
3547         dcmd->cmd = MFI_CMD_OP_DCMD;
3548         dcmd->cmd_status = 0xFF;
3549         dcmd->sge_count = 1;
3550         dcmd->flags = MFI_FRAME_DIR_WRITE;
3551         dcmd->timeout = 0;
3552         dcmd->pad_0 = 0;
3553         dcmd->data_xfer_len = size_map_info;
3554         ASSERT(num_lds <= 255);
3555         dcmd->mbox.b[0] = (U8)num_lds;
3538         dcmd->mbox.b[0] = num_lds;
3556         dcmd->mbox.b[1] = 1; /* Pend */
3557         dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
3558         dcmd->sgl.sge32[0].phys_addr = ci_h;
3559         dcmd->sgl.sge32[0].length = size_map_info;


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

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

3567         instance->unroll.syncCmd = 1;
3568         con_log(CL_ANN1, (CE_NOTE, "sync cmd issued. [SMID]:%x", cmd->SMID));
3551         con_log(CL_ANN1, (CE_NOTE, "sync cmd issued. [SMID]:%x",cmd->SMID));

3570         return (ret);
3571 }

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

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

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

3587         cmd = get_raid_msg_mfi_pkt(instance);

3589         if (!cmd) {
3590                 cmn_err(CE_WARN,
3591                     "Failed to get a cmd from free-pool abort_syncmap_cmd().");
3592                 return (DDI_FAILURE);
3593         }
3594         /* Clear the frame buffer and assign back the context id */
3595         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3578         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3596         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3597             cmd->index);

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

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

```
3610             &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3611         ddi_put32(cmd->frame_dma_obj.acc_handle,
3612             &abort_fr->abort_mfi_phys_addr_hi, 0);

3614         cmd->frame_count = 1;

3616         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

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

3626         return_raid_msg_mfi_pkt(instance, cmd);

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

3630         return (ret);
3631 }


3634 #ifdef PDSUPPORT
3635 int
3636 mrsas_tbolt_config_pd(struct mrsas_instance *instance, uint16_t tgt,
3637     uint8_t lun, dev_info_t **ldip)
3638 {
3639         struct scsi_device *sd;
3640         dev_info_t *child;
3641         int rval, dtype;
3642         struct mrsas_tbolt_pd_info *pds = NULL;
3626         uint64_t *wwn;


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

3647         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
3648                 if (ldip) {
3649                         *ldip = child;
3650                 }
3651                 if (instance->mr_tbolt_pd_list[tgt].flag != MRDRV_TGT_VALID) {
3652                         rval = mrsas_service_evt(instance, tgt, 1,
3653                             MRSAS_EVT_UNCONFIG_TGT, NULL);
3654                         con_log(CL_ANN1, (CE_WARN,
3655                             "mr_sas:DELETING STALE ENTRY  rval = %d "
3656                             "tgt id = %d", rval, tgt));
3641                                 "tgt id = %d ", rval, tgt));
3657                         return (NDI_FAILURE);
3658                 }
3659                 return (NDI_SUCCESS);
3660         }

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

3667         /* Check for Disk */
3652         /* Check for Disk*/
3668         if ((dtype == DTYPE_DIRECT)) {
3669                 if ((dtype == DTYPE_DIRECT) &&
3670                     (LE_16(pds->fwState) != PD_SYSTEM)) {
3671                         kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
```



```
3672                         return (NDI_FAILURE);
3673                 }
3674                 sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
3675                 sd->sd_address.a_hba_tran = instance->tran;
3676                 sd->sd_address.a_target = (uint16_t)tgt;
3677                 sd->sd_address.a_lun = (uint8_t)lun;

3679                 if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS) {
3680                         rval = mrsas_config_scsi_device(instance, sd, ldip);
3681                         con_log(CL_DLEVEL1, (CE_NOTE,
3682                             "Phys. device found: tgt %d dtype %d: %s",
3683                             tgt, dtype, sd->sd_inq->inq_vid));
3684                 } else {
3685                         rval = NDI_FAILURE;
3686                         con_log(CL_DLEVEL1, (CE_NOTE, "Phys. device Not found "
3687                             "scsi_hba_probe Failed: tgt %d dtype %d: %s",
3670                         con_log(CL_DLEVEL1, (CE_NOTE,
3671                             "Phys. device Not found scsi_hba_probe Failed: tgt %
3688                             tgt, dtype, sd->sd_inq->inq_vid));
3689                 }

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

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

3711 static void
3712 mrsas_tbolt_get_pd_info(struct mrsas_instance *instance,
3713     struct mrsas_tbolt_pd_info *pds, int tgt)
3695 mrsas_tbolt_get_pd_info(struct mrsas_instance *instance, struct mrsas_tbolt_pd_i
3696     int tgt)
3714 {
3715         struct mrsas_cmd        *cmd;
3716         struct mrsas_dcmd_frame *dcmd;
3717         dma_obj_t               dcmd_dma_obj;

3719         cmd = get_raid_msg_pkt(instance);

3721         if (!cmd) {
3722                 con_log(CL_ANN1,
3723                     (CE_WARN, "Failed to get a cmd for get pd info"));
3705             con_log(CL_ANN1, (CE_WARN, "Failed to get a cmd for get pd info"));
3724                 return;
3725         }

3727         /* Clear the frame buffer and assign back the context id */
3728         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3710         memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3729         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3730             cmd->index);
```

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

3741         (void) mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
3742             DDI_STRUCTURE_LE_ACC);
3743         bzero(dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info));
3744         bzero(dcmd->mbox.b, 12);
3725         (void) memset(dcmd_dma_obj.buffer, 0, sizeof (struct mrsas_tbolt_pd_info
3726         (void) memset(dcmd->mbox.b, 0, 12);
3745         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3746         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
3747         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3748         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3749             MFI_FRAME_DIR_READ);
3730         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags, MFI_FRAME_DIR_REA
3750         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3751         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3752             sizeof (struct mrsas_tbolt_pd_info));
3753         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3754             MR_DCMD_PD_GET_INFO);
3755         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], tgt);
3756         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3757             sizeof (struct mrsas_tbolt_pd_info));
3758         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3759             dcmd_dma_obj.dma_cookie[0].dmac_address);

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

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

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

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

_____***unchanged_portion_omitted_***