****
17098 Fri Oct 12 14:59:02 2012
new/usr/src/uts/common/io/mr_sas/fusion.h
Code review comments
****
_____***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
 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
 527          * 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 Fri Oct 12 14:59:03 2012
new/usr/src/uts/common/io/mr_sas/ld_pd_map.c
Code review comments
**********************************************************
```

```
  1 /*
  2  * ***************************************************************************
  3  *
  4  * ld_pd_map.c
  5  *
  6  * Solaris MegaRAID device driver for SAS2.0 controllers
  7  * Copyright (c) 2008-2012, LSI Logic Corporation.
  8  * All rights reserved.
  9  *
 10  * Version:
 11  * Author:
 12  *              Swaminathan K S
 13  *              Arun Chandrashekhar
 14  *              Manju R
 15  *              Rasheed
 16  *              Shakeel Bukhari
 17  *
 18  *
 19  * This module contains functions for device drivers
 20  * to get pd-ld mapping information.
 21  *
 22  * ***************************************************************************
 23  */

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

 29 /*
 30  * This function will check if FAST IO is possible on this logical drive
 31  * by checking the EVENT information available in the driver
 31  * by checking the EVENT information availabe in the driver
 32  */
 33 #define MR_LD_STATE_OPTIMAL 3
 34 #define ABS_DIFF(a, b)   (((a) > (b)) ? ((a) - (b)) : ((b) - (a)))

 36 static void mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *,
 37     PLD_LOAD_BALANCE_INFO);

 39 #define FALSE 0
 40 #define TRUE 1

 42 typedef U64     REGION_KEY;
 43 typedef U32     REGION_LEN;
 44 extern int      debug_level_g;


 47 MR_LD_RAID
 48 *MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
 49 {
 50         return (&map->raidMap.ldSpanMap[ld].ldRaid);
 51 }
_____unchanged_portion_omitted_


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)
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;
208         U8              retval = TRUE;
209         U32             rowMod;
210         U32             armQ;
211         U32             arm;

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

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

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

220                 if (raid->rowSize == 0) {
221                         return (FALSE);
222                 }
223                 rowMod = (row % (raid->rowSize));
224                 armQ = raid->rowSize-1-rowMod;
225                 arm = armQ + 1 + logArm;
223                 arm = armQ+1+logArm;
226                 if (arm >= raid->rowSize)
227                         arm -= raid->rowSize;
228                 physArm = (U8)arm;
229         } else {
230                 if (raid->modFactor == 0)
231                         return (FALSE);
232                 physArm = MR_LdDataArmGet(ld,
233                     (stripRow % (raid->modFactor)), map);
234         }
235         if (raid->spanDepth == 1) {
236                 span = 0;
237                 *pdBlock = row << raid->stripeShift;
238         } else
239                 span = (U8)MR_GetSpanBlock(ld, row, pdBlock, map, &error_code);

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

244         /* Get the array on which this span is present. */
245         arRef           = MR_LdSpanArrayGet(ld, span, map);
246         /* Get the Pd. */
247         pd              = MR_ArPdGet(arRef, physArm, map);
248         /* Get dev handle from Pd. */
249         if (pd != MR_PD_INVALID) {
250                 *pDevHandle     = MR_PdDevHandleGet(pd, map);
251         } else {
```

```
252                 *pDevHandle = MR_PD_INVALID; /* set dev handle as invalid. */
253                 if ((raid->level >= 5) &&
254                     ((instance->device_id != PCI_DEVICE_ID_LSI_INVADER) ||
255                     (instance->device_id == PCI_DEVICE_ID_LSI_INVADER &&
256                     raid->regTypeReqOnRead != REGION_TYPE_UNUSED))) {
257                         pRAID_Context->regLockFlags = REGION_TYPE_EXCLUSIVE;
258                 } else if (raid->level == 1) {
259                         /* Get Alternate Pd. */
260                         pd = MR_ArPdGet(arRef, physArm + 1, map);
261                         /* Get dev handle from Pd. */
262                         if (pd != MR_PD_INVALID)
263                                 *pDevHandle = MR_PdDevHandleGet(pd, map);
264                 }
265         }

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

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

272         return (retval);
273 }



277 /*
278  * ****************************************************************************
279  *
280  * MR_BuildRaidContext function
281  *
282  * This function will initiate command processing.  The start/end row and strip
283  * information is calculated then the lock is acquired.
284  * This function will return 0 if region lock
285  * was acquired OR return num strips ???
286  */

288 U8
289 MR_BuildRaidContext(struct mrsas_instance *instance,
290     struct IO_REQUEST_INFO *io_info, MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context,
291     MR_FW_RAID_MAP_ALL *map)
292 {
293         MR_LD_RAID      *raid;
294         U32             ld, stripSize, stripe_mask;
295         U64             endLba, endStrip, endRow;
296         U64             start_row, start_strip;
297         REGION_KEY      regStart;
298         REGION_LEN      regSize;
299         U8              num_strips, numRows;
300         U16             ref_in_start_stripe;
301         U16             ref_in_end_stripe;

303         U64             ldStartBlock;
304         U32             numBlocks, ldTgtId;
305         U8              isRead;
306         U8              retval = 0;

308         ldStartBlock = io_info->ldStartBlock;
309         numBlocks = io_info->numBlocks;
310         ldTgtId = io_info->ldTgtId;
311         isRead = io_info->isRead;

313         if (map == NULL) {
314                 io_info->fpOkForIo = FALSE;
315                 return (FALSE);
316         }
```



```
318         ld = MR_TargetIdToLdGet(ldTgtId, map);

320         if (ld >= MAX_LOGICAL_DRIVES) {
321                 io_info->fpOkForIo = FALSE;
322                 return (FALSE);
323         }

325         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 */
336         /* Check to make sure is not deviding by zero */
339         if (raid->rowDataSize == 0)
340                 return (FALSE);
341         start_row               =  (start_strip / raid->rowDataSize);
342         endRow                  =  (endStrip  / raid->rowDataSize);
343         /* 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) {
352                 if (isRead)
355                         io_info->fpOkForIo = (raid->capability.fpReadCapable &&
356                             ((num_strips == 1) ||
357                             raid->capability.fpReadAcrossStripe));
358                 } else {
356                 else
359                         io_info->fpOkForIo =
360                             (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) {
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;
386                         regSize += (numRows-2) << raid->stripeShift;
390                 }

392                 if (endStrip == endRow * raid->rowDataSize) {
393                         regSize += ref_in_end_stripe + 1;
389                 if (endStrip == endRow*raid->rowDataSize) {
390                         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         }

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);

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

428                 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);
439                         }
440                 }
441         }
442         return (TRUE);
443 }
```

_____***unchanged_portion_omitted_***

****
6952 Fri Oct 12 14:59:04 2012
new/usr/src/uts/common/io/mr_sas/ld_pd_map.h
Code review comments
****
_____***unchanged_portion_omitted_***

```
 69 /*
 70  * Raid Context structure which describes MegaRAID specific IO Parameters
 70  * Raid Context structure which describes MegaRAID specific IO Paramenters
 71  * This resides at offset 0x60 where the SGL normally starts in MPT IO Frames
 72  */
 73 typedef struct _MPI2_SCSI_IO_VENDOR_UNIQUE {
 74         U8 nsegType;            /* 0x00 nseg[7:4], Type[3:0] */
 75         U8 resvd0;              /* 0x01 */
 76         U16 timeoutValue;       /* 0x02 -0x03 */
 77         U8 regLockFlags;        /* 0x04 */
 78         U8 reservedForHw1;      /* 0x05 */
 79         U16 ldTargetId;         /* 0x06 - 0x07 */
 80         U64 regLockRowLBA;      /* 0x08 - 0x0F */
 81         U32 regLockLength;      /* 0x10 - 0x13 */
 82         U16 nextLMId;           /* 0x14 - 0x15 */
 83         U8 extStatus;           /* 0x16 */
 84         U8 status;              /* 0x17 status */
 85         U8 RAIDFlags;           /* 0x18 resvd[7:6], ioSubType[5:4], */
 86                                 /* resvd[3:1], preferredCpu[0] */
 87         U8 numSGE;              /* 0x19 numSge; not including chain entries */
 88         U16 configSeqNum;       /* 0x1A -0x1B */
 89         U8 spanArm;             /* 0x1C span[7:5], arm[4:0] */
 90         U8 resvd2[3];           /* 0x1D-0x1f */
 91 } MPI2_SCSI_IO_VENDOR_UNIQUE, MPI25_SCSI_IO_VENDOR_UNIQUE;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  222548 Fri Oct 12 14:59:05 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.c
Code review comments
**********************************************************
/*
 * 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;
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; */
/* 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

#if 0
/* Enable OCR on firmware fault */
static volatile int  debug_support_ocr_isr_g  = 0;
#endif
#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 *,
```

```
128                     register struct scsi_pkt *);
129 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
130 static int      mrsas_tran_reset(struct scsi_address *, int);
131 #if 0
132 static int      mrsas_tran_bus_reset(dev_info_t *, int);
133 #endif
134 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
135 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
136 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
137                     struct scsi_pkt *);
138 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
139 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
140 static int      mrsas_tran_quiesce(dev_info_t *dip);
141 static int      mrsas_tran_unquiesce(dev_info_t *dip);
142 static uint_t   mrsas_isr();
143 static uint_t   mrsas_softintr();
144 static void     mrsas_undo_resources(dev_info_t *, struct mrsas_instance *);
145 static struct mrsas_cmd *get_mfi_pkt(struct mrsas_instance *);
146 static void     return_mfi_pkt(struct mrsas_instance *,
147                     struct mrsas_cmd *);

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

191 static void     mrsas_rem_intrs(struct mrsas_instance *instance);
192 static int      mrsas_add_intrs(struct mrsas_instance *instance, int intr_type);
```



```
194 static void     mrsas_tran_tgt_free(dev_info_t *, dev_info_t *,
195                     scsi_hba_tran_t *, struct scsi_device *);
196 static int      mrsas_tran_bus_config(dev_info_t *, uint_t,
197                     ddi_bus_config_op_t, void *, dev_info_t **);
198 static int      mrsas_parse_devname(char *, int *, int *);
199 static int      mrsas_config_all_devices(struct mrsas_instance *);
200 static int      mrsas_config_ld(struct mrsas_instance *, uint16_t,
201                         uint8_t, dev_info_t **);
202 static int      mrsas_name_node(dev_info_t *, char *, int);
203 static void     mrsas_issue_evt_taskq(struct mrsas_eventinfo *);
204 static void     free_additional_dma_buffer(struct mrsas_instance *);
205 static void io_timeout_checker(void *);
206 static void mrsas_fm_init(struct mrsas_instance *);
207 static void mrsas_fm_fini(struct mrsas_instance *);

209 static struct mrsas_function_template mrsas_function_template_ppc = {
210         .read_fw_status_reg = read_fw_status_reg_ppc,
211         .issue_cmd = issue_cmd_ppc,
212         .issue_cmd_in_sync_mode = issue_cmd_in_sync_mode_ppc,
213         .issue_cmd_in_poll_mode = issue_cmd_in_poll_mode_ppc,
214         .enable_intr = enable_intr_ppc,
215         .disable_intr = disable_intr_ppc,
216         .intr_ack = intr_ack_ppc,
217         .init_adapter = mrsas_init_adapter_ppc
218 /*      .reset_adapter = mrsas_reset_adapter_ppc */
218 };
______unchanged_portion_omitted_

281 /*
282  * dev_ops contains configuration routines
283  */
284 static struct dev_ops mrsas_ops = {
285         DEVO_REV,               /* rev, */
286         0,                      /* refcnt */
287         mrsas_getinfo,          /* getinfo */
288         nulldev,                /* identify */
289         nulldev,                /* probe */
290         mrsas_attach,           /* attach */
291         mrsas_detach,           /* detach */
292 #ifdef  __sparc
293         mrsas_reset,            /* reset */
294 #else   /* __sparc */
295         nodev,
296 #endif  /* __sparc */
297         &mrsas_cb_ops,          /* char/block ops */
298         NULL,                   /* bus ops */
299         NULL,                   /* power */
300 #ifdef __sparc
301         ddi_quiesce_not_needed
302 #else   /* __sparc */
303         mrsas_quiesce   /* quiesce */
304 #endif  /* __sparc */

305 };
______unchanged_portion_omitted_

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

327 unsigned int enable_fp = 1;


330 /*
331  * ************************************************************************** *
332  *                                                                            *
333  *         common entry points - for loadable kernel modules                  *
334  *                                                                            *
```

```
 335  * ********************************************************************** *
 336  */

 338 /*
 339  * _init - initialize a loadable module
 340  * @void
 341  *
 342  * The driver should perform any one-time resource allocation or data
 343  * initialization during driver loading in _init(). For example, the driver
 344  * should initialize any mutexes global to the driver in this routine.
 345  * The driver should not, however, use _init() to allocate or initialize
 346  * anything that has to do with a particular instance of the device.
 347  * Per-instance initialization must be done in attach().
 348  */
 349 int
 350 _init(void)
 351 {
 352         int ret;

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

 356         ret = ddi_soft_state_init(&mrsas_state,
 357             sizeof (struct mrsas_instance), 0);

 359         if (ret != DDI_SUCCESS) {
 360                 cmn_err(CE_WARN, "mr_sas: could not init state");
 361                 return (ret);
 362         }

 364         if ((ret = scsi_hba_init(&modlinkage)) != DDI_SUCCESS) {
 365                 cmn_err(CE_WARN, "mr_sas: could not init scsi hba");
 366                 ddi_soft_state_fini(&mrsas_state);
 367                 return (ret);
 368         }

 370         ret = mod_install(&modlinkage);

 372         if (ret != DDI_SUCCESS) {
 373                 cmn_err(CE_WARN, "mr_sas: mod_install failed");
 374                 scsi_hba_fini(&modlinkage);
 375                 ddi_soft_state_fini(&mrsas_state);
 376         }

 378         return (ret);
 379 }
_____unchanged_portion_omitted_


 426 /*
 427  * ********************************************************************** *
 428  *                                                                        *
 429  *               common entry points - for autoconfiguration              *
 430  *                                                                        *
 431  * ********************************************************************** *
 432  */
 433 /*
 434  * attach - adds a device to the system as part of initialization
 435  * @dip:
 436  * @cmd:
 437  *
 438  * The kernel calls a driver's attach() entry point to attach an instance of
 439  * a device (for MegaRAID, it is instance of a controller) or to resume
 440  * operation for an instance of a device that has been suspended or has been
 441  * shut down by the power management framework
 442  * The attach() entry point typically includes the following types of
 443  * processing:
```



```
 444  * - allocate a soft-state structure for the device instance (for MegaRAID,
 445  *   controller instance)
 446  * - initialize per-instance mutexes
 447  * - initialize condition variables
 448  * - register the device's interrupts (for MegaRAID, controller's interrupts)
 449  * - map the registers and memory of the device instance (for MegaRAID,
 450  *   controller instance)
 451  * - create minor device nodes for the device instance (for MegaRAID,
 452  *   controller instance)
 453  * - report that the device instance (for MegaRAID, controller instance) has
 454  *   attached
 455  */
 456 static int
 457 mrsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 458 {
 459         int             instance_no;
 460         int             nregs;
 461         int             i = 0;
 462         uint8_t         irq;
 463         uint16_t        vendor_id;
 464         uint16_t        device_id;
 465         uint16_t        subsysvid;
 466         uint16_t        subsysid;
 467         uint16_t        command;
 468         off_t           reglength = 0;
 469         int             intr_types = 0;
 470         char            *data;

 472         scsi_hba_tran_t         *tran;
 473         ddi_dma_attr_t  tran_dma_attr;
 474         struct mrsas_instance   *instance;

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

 478         /* CONSTCOND */
 479         ASSERT(NO_COMPETING_THREADS);

 481         instance_no = ddi_get_instance(dip);

 483         /*
 484          * check to see whether this device is in a DMA-capable slot.
 485          */
 486         if (ddi_slaveonly(dip) == DDI_SUCCESS) {
 487                 cmn_err(CE_WARN,
 488                     "mr_sas%d: Device in slave-only slot, unused",
 489                     instance_no);
 490                 return (DDI_FAILURE);
 491         }

 493         switch (cmd) {
 494         case DDI_ATTACH:
 495                 /* allocate the soft state for the instance */
 496                 if (ddi_soft_state_zalloc(mrsas_state, instance_no)
 497                     != DDI_SUCCESS) {
 498                         cmn_err(CE_WARN,
 499                             "mr_sas%d: Failed to allocate soft state",
 500                             instance_no);
 501                         return (DDI_FAILURE);
 502                 }

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

 507                 if (instance == NULL) {
 508                         cmn_err(CE_WARN,
 509                             "mr_sas%d: Bad soft state", instance_no);
```

```
510                     ddi_soft_state_free(mrsas_state, instance_no);
511                     return (DDI_FAILURE);
512             }

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

514             instance->unroll.softs  = 1;

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

523                     ddi_soft_state_free(mrsas_state, instance_no);
524                     return (DDI_FAILURE);
525             }
530             if (instance->pci_handle == NULL) {
531                     cmn_err(CE_WARN,
532                         "mr_sas%d: pci config setup failed ",
533                         instance_no);
534                     ddi_soft_state_free(mrsas_state, instance_no);
535                     return (DDI_FAILURE);
536             }



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

531                     pci_config_teardown(&instance->pci_handle);
532                     ddi_soft_state_free(mrsas_state, instance_no);
533                     return (DDI_FAILURE);
534             }

536             vendor_id = pci_config_get16(instance->pci_handle,
537                 PCI_CONF_VENID);
538             device_id = pci_config_get16(instance->pci_handle,
539                 PCI_CONF_DEVID);

541             subsysvid = pci_config_get16(instance->pci_handle,
542                 PCI_CONF_SUBVENID);
543             subsysid = pci_config_get16(instance->pci_handle,
544                 PCI_CONF_SUBSYSID);

546             pci_config_put16(instance->pci_handle, PCI_CONF_COMM,
547                 (pci_config_get16(instance->pci_handle,
548                 PCI_CONF_COMM) | PCI_COMM_ME));
549             irq = pci_config_get8(instance->pci_handle,
550                 PCI_CONF_ILINE);

552             con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
553                 "0x%x:0x%x 0x%x:0x%x, irq:%d drv-ver:%s",
554                 instance_no, vendor_id, device_id, subsysvid,
555                 subsysid, irq, MRSAS_VERSION));

557             /* enable bus-mastering */
558             command = pci_config_get16(instance->pci_handle,
559                 PCI_CONF_COMM);

561             if (!(command & PCI_COMM_ME)) {
562                     command |= PCI_COMM_ME;

564                     pci_config_put16(instance->pci_handle,
```



```
565                         PCI_CONF_COMM, command);

567                     con_log(CL_ANN, (CE_CONT, "mr_sas%d: "
568                         "enable bus-mastering", instance_no));
569             } else {
570                     con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
571                         "bus-mastering already set", instance_no));
572             }

574             /* initialize function pointers */
575             switch (device_id) {
576             case PCI_DEVICE_ID_LSI_TBOLT:
577             case PCI_DEVICE_ID_LSI_INVADER:
578                     con_log(CL_ANN, (CE_NOTE,
579                         "mr_sas: 2208 T.B. device detected"));

581                     instance->func_ptr =
582                         &mrsas_function_template_fusion;
583                     instance->tbolt = 1;
584                     break;

586             case PCI_DEVICE_ID_LSI_2108VDE:
587             case PCI_DEVICE_ID_LSI_2108V:
588                     con_log(CL_ANN, (CE_NOTE,
589                         "mr_sas: 2108 Liberator device detected"));

591                     instance->func_ptr =
592                         &mrsas_function_template_ppc;
593                     break;

595             default:
596                     cmn_err(CE_WARN,
597                         "mr_sas: Invalid device detected");

599                     pci_config_teardown(&instance->pci_handle);
600                     ddi_soft_state_free(mrsas_state, instance_no);
601                     return (DDI_FAILURE);
602             }

604             instance->baseaddress = pci_config_get32(
605                 instance->pci_handle, PCI_CONF_BASE0);
606             instance->baseaddress &= 0x0fffc;

608             instance->dip           = dip;
609             instance->vendor_id     = vendor_id;
610             instance->device_id     = device_id;
611             instance->subsysvid     = subsysvid;
612             instance->subsysid      = subsysid;
613             instance->instance      = instance_no;

615             /* Initialize FMA */
616             instance->fm_capabilities = ddi_prop_get_int(
617                 DDI_DEV_T_ANY, instance->dip, DDI_PROP_DONTPASS,
618                 "fm-capable", DDI_FM_EREPORT_CAPABLE |
619                 DDI_FM_ACCCHK_CAPABLE | DDI_FM_DMACHK_CAPABLE
620                 | DDI_FM_ERRCB_CAPABLE);

622             mrsas_fm_init(instance);

624             /* Setup register map */
625             if ((ddi_dev_regsize(instance->dip,
626                 REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
627                 reglength < MINIMUM_MFI_MEM_SZ) {
628                     goto fail_attach;
629             }
630             if (reglength > DEFAULT_MFI_MEM_SZ) {
```

```
631                         reglength = DEFAULT_MFI_MEM_SZ;
632                         con_log(CL_DLEVEL1, (CE_NOTE,
633                             "mr_sas: register length to map is 0x%lx bytes",
634                             reglength));
635                 }
636                 if (ddi_regs_map_setup(instance->dip,
637                     REGISTER_SET_IO_2108, &instance->regmap, 0,
638                     reglength, &endian_attr, &instance->regmap_handle)
639                     != DDI_SUCCESS) {
640                         cmn_err(CE_WARN,
641                             "mr_sas: couldn't map control registers");
642                         goto fail_attach;
643                 }
657                 if (instance->regmap_handle == NULL) {
658                         cmn_err(CE_WARN,
659                             "mr_sas: couldn't map control registers");
660                         goto fail_attach;
661                 }

645                 instance->unroll.regs = 1;

647                 /*
648                  * Disable Interrupt Now.
649                  * Setup Software interrupt
650                  */
651                 instance->func_ptr->disable_intr(instance);

653                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
654                     "mrsas-enable-msi", &data) == DDI_SUCCESS) {
655                         if (strncmp(data, "no", 3) == 0) {
656                                 msi_enable = 0;
657                                 con_log(CL_ANN1, (CE_WARN,
658                                     "msi_enable = %d disabled", msi_enable));
659                         }
660                         ddi_prop_free(data);
661                 }

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

665                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
666                     "mrsas-enable-fp", &data) == DDI_SUCCESS) {
667                         if (strncmp(data, "no", 3) == 0) {
668                                 enable_fp = 0;
669                                 cmn_err(CE_NOTE,
670                                     "enable_fp = %d, Fast-Path disabled.\n",
671                                     enable_fp);
672                         }

674                         ddi_prop_free(data);
675                 }

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

679                 /* Check for all supported interrupt types */
680                 if (ddi_intr_get_supported_types(
681                     dip, &intr_types) != DDI_SUCCESS) {
682                         cmn_err(CE_WARN,
683                             "ddi_intr_get_supported_types() failed");
684                         goto fail_attach;
685                 }

687                 con_log(CL_DLEVEL1, (CE_NOTE,
688                     "ddi_intr_get_supported_types() ret: 0x%x", intr_types));

690                 /* Initialize and Setup Interrupt handler */
691                 if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
```



```
692                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
693                             DDI_SUCCESS) {
694                                 cmn_err(CE_WARN,
695                                     "MSIX interrupt query failed");
696                                 goto fail_attach;
697                         }
698                         instance->intr_type = DDI_INTR_TYPE_MSIX;
699                 } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
700                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
701                             DDI_SUCCESS) {
702                                 cmn_err(CE_WARN,
703                                     "MSI interrupt query failed");
704                                 goto fail_attach;
705                         }
706                         instance->intr_type = DDI_INTR_TYPE_MSI;
707                 } else if (intr_types & DDI_INTR_TYPE_FIXED) {
708                         msi_enable = 0;
709                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
710                             DDI_SUCCESS) {
711                                 cmn_err(CE_WARN,
712                                     "FIXED interrupt query failed");
713                                 goto fail_attach;
714                         }
715                         instance->intr_type = DDI_INTR_TYPE_FIXED;
716                 } else {
717                         cmn_err(CE_WARN, "Device cannot "
718                             "suppport either FIXED or MSI/X "
719                             "interrupts");
720                         goto fail_attach;
721                 }

723                 instance->unroll.intr = 1;

725                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
726                     "mrsas-enable-ctio", &data) == DDI_SUCCESS) {
727                         if (strncmp(data, "no", 3) == 0) {
728                                 ctio_enable = 0;
729                                 con_log(CL_ANN1, (CE_WARN,
730                                     "ctio_enable = %d disabled", ctio_enable));
731                         }
732                         ddi_prop_free(data);
733                 }

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

737                 /* setup the mfi based low level driver */
738                 if (mrsas_init_adapter(instance) != DDI_SUCCESS) {
739                         cmn_err(CE_WARN, "mr_sas: "
740                             "could not initialize the low level driver");

742                         goto fail_attach;
743                 }

745                 /* Initialize all Mutex */
746                 INIT_LIST_HEAD(&instance->completed_pool_list);
747                 mutex_init(&instance->completed_pool_mtx,
748                     "completed_pool_mtx", MUTEX_DRIVER,
749                     DDI_INTR_PRI(instance->intr_pri));

751                 mutex_init(&instance->sync_map_mtx,
752                     "sync_map_mtx", MUTEX_DRIVER,
753                     DDI_INTR_PRI(instance->intr_pri));

755                 mutex_init(&instance->app_cmd_pool_mtx,
756                     "app_cmd_pool_mtx", MUTEX_DRIVER,
757                     DDI_INTR_PRI(instance->intr_pri));
```

```
759                 mutex_init(&instance->config_dev_mtx, "config_dev_mtx",
760                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

762                 mutex_init(&instance->cmd_pend_mtx, "cmd_pend_mtx",
763                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

765                 mutex_init(&instance->ocr_flags_mtx, "ocr_flags_mtx",
766                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

768                 mutex_init(&instance->int_cmd_mtx, "int_cmd_mtx",
769                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
770                 cv_init(&instance->int_cmd_cv, NULL, CV_DRIVER, NULL);

772                 mutex_init(&instance->cmd_pool_mtx, "cmd_pool_mtx",
773                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

775                 mutex_init(&instance->reg_write_mtx, "reg_write_mtx",
776                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

778                 if (instance->tbolt) {
779                         mutex_init(&instance->cmd_app_pool_mtx,
780                             "cmd_app_pool_mtx", MUTEX_DRIVER,
781                             DDI_INTR_PRI(instance->intr_pri));

783                         mutex_init(&instance->chip_mtx,
784                             "chip_mtx", MUTEX_DRIVER,
785                             DDI_INTR_PRI(instance->intr_pri));

787                 }

789                 instance->unroll.mutexs = 1;

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

793                 /* Register our soft-isr for highlevel interrupts. */
794                 instance->isr_level = instance->intr_pri;
795                 if (!(instance->tbolt)) {
796                         if (instance->isr_level == HIGH_LEVEL_INTR) {
797                                 if (ddi_add_softintr(dip,
798                                     DDI_SOFTINT_HIGH,
799                                     &instance->soft_intr_id, NULL, NULL,
800                                     mrsas_softintr, (caddr_t)instance) !=
801                                     DDI_SUCCESS) {
802                                         cmn_err(CE_WARN,
803                                             "Software ISR did not register");

805                                         goto fail_attach;
806                                 }

808                                 instance->unroll.soft_isr = 1;

810                         }
811                 }

813                 instance->softint_running = 0;

815                 /* Allocate a transport structure */
816                 tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

818                 if (tran == NULL) {
819                         cmn_err(CE_WARN,
820                             "scsi_hba_tran_alloc failed");
821                         goto fail_attach;
822                 }
```



```
824                 instance->tran = tran;
825                 instance->unroll.tran = 1;

827                 tran->tran_hba_private  = instance;
828                 tran->tran_tgt_init     = mrsas_tran_tgt_init;
829                 tran->tran_tgt_probe    = scsi_hba_probe;
830                 tran->tran_tgt_free     = mrsas_tran_tgt_free;
831                 if (instance->tbolt) {
832                         tran->tran_init_pkt     =
833                             mrsas_tbolt_tran_init_pkt;
834                         tran->tran_start        =
835                             mrsas_tbolt_tran_start;
836                 } else {
837                         tran->tran_init_pkt     = mrsas_tran_init_pkt;
838                         tran->tran_start        = mrsas_tran_start;
839                 }
840                 tran->tran_abort        = mrsas_tran_abort;
841                 tran->tran_reset        = mrsas_tran_reset;
842                 tran->tran_getcap       = mrsas_tran_getcap;
843                 tran->tran_setcap       = mrsas_tran_setcap;
844                 tran->tran_destroy_pkt  = mrsas_tran_destroy_pkt;
845                 tran->tran_dmafree      = mrsas_tran_dmafree;
846                 tran->tran_sync_pkt     = mrsas_tran_sync_pkt;
847                 tran->tran_quiesce      = mrsas_tran_quiesce;
848                 tran->tran_unquiesce    = mrsas_tran_unquiesce;
849                 tran->tran_bus_config   = mrsas_tran_bus_config;

851                 if (mrsas_relaxed_ordering)
852                         mrsas_generic_dma_attr.dma_attr_flags |=
853                             DDI_DMA_RELAXED_ORDERING;


856                 tran_dma_attr = mrsas_generic_dma_attr;
857                 tran_dma_attr.dma_attr_sgllen = instance->max_num_sge;

859                 /* Attach this instance of the hba */
860                 if (scsi_hba_attach_setup(dip, &tran_dma_attr, tran, 0)
861                     != DDI_SUCCESS) {
862                         cmn_err(CE_WARN,
863                             "scsi_hba_attach failed");

865                         goto fail_attach;
866                 }
867                 instance->unroll.tranSetup = 1;
868                 con_log(CL_ANN1,
869                     (CE_CONT, "scsi_hba_attach_setup()  done."));

871                 /* create devctl node for cfgadm command */
872                 if (ddi_create_minor_node(dip, "devctl",
873                     S_IFCHR, INST2DEVCTL(instance_no),
874                     DDI_NT_SCSI_NEXUS, 0) == DDI_FAILURE) {
875                         cmn_err(CE_WARN,
876                             "mr_sas: failed to create devctl node.");

878                         goto fail_attach;
879                 }

881                 instance->unroll.devctl = 1;

883                 /* create scsi node for cfgadm command */
884                 if (ddi_create_minor_node(dip, "scsi", S_IFCHR,
885                     INST2SCSI(instance_no), DDI_NT_SCSI_ATTACHMENT_POINT, 0) ==
886                     DDI_FAILURE) {
887                         cmn_err(CE_WARN,
888                             "mr_sas: failed to create scsi node.");
```

```
890                     goto fail_attach;
891             }

893             instance->unroll.scsictl = 1;

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

898             /*
899              * Create a node for applications
900              * for issuing ioctl to the driver.
901              */
902             if (ddi_create_minor_node(dip, instance->iocnode,
903                 S_IFCHR, INST2LSIRDCTL(instance_no), DDI_PSEUDO, 0) ==
904                 DDI_FAILURE) {
905                     cmn_err(CE_WARN,
906                         "mr_sas: failed to create ioctl node.");

908                     goto fail_attach;
909             }

911             instance->unroll.ioctl = 1;

913             /* Create a taskq to handle dr events */
914             if ((instance->taskq = ddi_taskq_create(dip,
915                 "mrsas_dr_taskq", 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
916                     cmn_err(CE_WARN,
917                         "mr_sas: failed to create taskq ");
918                     instance->taskq = NULL;
919                     goto fail_attach;
920             }
921             instance->unroll.taskq = 1;
922             con_log(CL_ANN1, (CE_CONT, "ddi_taskq_create()  done."));

924             /* enable interrupt */
925             instance->func_ptr->enable_intr(instance);

927             /* initiate AEN */
928             if (start_mfi_aen(instance)) {
929                     cmn_err(CE_WARN,
930                         "mr_sas: failed to initiate AEN.");
931                     goto fail_attach;
932             }
933             instance->unroll.aenPend = 1;
934             con_log(CL_ANN1,
935                 (CE_CONT, "AEN started for instance %d.", instance_no));

937             /* Finally! We are on the air.  */
938             ddi_report_dev(dip);

940             /* FMA handle checking. */
941             if (mrsas_check_acc_handle(instance->regmap_handle) !=
942                 DDI_SUCCESS) {
943                     goto fail_attach;
944             }
945             if (mrsas_check_acc_handle(instance->pci_handle) !=
946                 DDI_SUCCESS) {
947                     goto fail_attach;
948             }

950             instance->mr_ld_list =
951                 kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
952                 KM_SLEEP);
953             if (instance->mr_ld_list == NULL) {
954                     cmn_err(CE_WARN, "mr_sas attach(): "
955                         "failed to allocate ld_list array");
```



```
956                     goto fail_attach;
957             }
958             instance->unroll.ldlist_buff = 1;

960 #ifdef PDSUPPORT
961             if (instance->tbolt) {
962                     instance->mr_tbolt_pd_max = MRSAS_TBOLT_PD_TGT_MAX;
963                     instance->mr_tbolt_pd_list =
964                         kmem_zalloc(MRSAS_TBOLT_GET_PD_MAX(instance) *
965                         sizeof (struct mrsas_tbolt_pd), KM_SLEEP);
966                     ASSERT(instance->mr_tbolt_pd_list);
967                     for (i = 0; i < instance->mr_tbolt_pd_max; i++) {
968                             instance->mr_tbolt_pd_list[i].lun_type =
969                                 MRSAS_TBOLT_PD_LUN;
970                             instance->mr_tbolt_pd_list[i].dev_id =
971                                 (uint8_t)i;
972                     }

974                     instance->unroll.pdlist_buff = 1;
975             }
976 #endif
977             break;
978         case DDI_PM_RESUME:
979             con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_PM_RESUME"));
980             break;
981         case DDI_RESUME:
982             con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_RESUME"));
983             break;
984         default:
985             con_log(CL_ANN,
986                 (CE_WARN, "mr_sas: invalid attach cmd=%x", cmd));
987             return (DDI_FAILURE);
988         }


991         cmn_err(CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
992             instance_no);
993         return (DDI_SUCCESS);

995 fail_attach:

997         mrsas_undo_resources(dip, instance);

999         mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
1000        ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

1002        mrsas_fm_fini(instance);

1004        pci_config_teardown(&instance->pci_handle);
1005        ddi_soft_state_free(mrsas_state, instance_no);

1007        con_log(CL_ANN, (CE_WARN, "mr_sas: return failure from mrsas_attach"));

1009        cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d",
1010            instance_no);

1012        return (DDI_FAILURE);
1013 }
```

*unchanged_portion_omitted_*

```
1168 static void
1169 mrsas_undo_resources(dev_info_t *dip, struct mrsas_instance *instance)
1170 {
1171         int     instance_no;
```

```
1173         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


1176         instance_no = ddi_get_instance(dip);


1179         if (instance->unroll.ioctl == 1) {
1180                 ddi_remove_minor_node(dip, instance->iocnode);
1181                 instance->unroll.ioctl = 0;
1182         }

1184         if (instance->unroll.scsictl == 1) {
1185                 ddi_remove_minor_node(dip, "scsi");
1186                 instance->unroll.scsictl = 0;
1187         }

1189         if (instance->unroll.devctl == 1) {
1190                 ddi_remove_minor_node(dip, "devctl");
1191                 instance->unroll.devctl = 0;
1192         }

1194         if (instance->unroll.tranSetup == 1) {
1195                 if (scsi_hba_detach(dip) != DDI_SUCCESS) {
1196                         cmn_err(CE_WARN,
1197                             "mr_sas2%d: failed to detach", instance_no);
1198                         return;  /* DDI_FAILURE */
1199                 }
1200                 instance->unroll.tranSetup = 0;
1201                 con_log(CL_ANN1, (CE_CONT, "scsi_hba_dettach()  done."));
1202         }

1204         if (instance->unroll.tran == 1)  {
1205                 scsi_hba_tran_free(instance->tran);
1206                 instance->unroll.tran = 0;
1207                 con_log(CL_ANN1, (CE_CONT, "scsi_hba_tran_free()  done."));
1208         }

1210         if (instance->unroll.syncCmd == 1) {
1211                 if (instance->tbolt) {
1212                         if (abort_syncmap_cmd(instance,
1213                             instance->map_update_cmd)) {
1214                                 cmn_err(CE_WARN, "mrsas_detach: "
1215                                     "failed to abort previous syncmap command");
1216                         }

1218                         instance->unroll.syncCmd = 0;
1219                         con_log(CL_ANN1, (CE_CONT, "sync cmd aborted, done."));
1220                 }
1221         }

1223         if (instance->unroll.aenPend == 1) {
1224                 if (abort_aen_cmd(instance, instance->aen_cmd))
1225                         cmn_err(CE_WARN, "mrsas_detach: "
1226                             "failed to abort prevous AEN command");

1228                 instance->unroll.aenPend = 0;
1229                 con_log(CL_ANN1, (CE_CONT, "aen cmd aborted, done."));
1230                 /* This means the controller is fully initialized and running */
1248                 /* This means the controller is fully initialzed and running */
1231                 /* Shutdown should be a last command to controller. */
1232                 /* shutdown_controller(); */
1233         }


1236         if (instance->unroll.timer == 1)         {
1237                 if (instance->timeout_id != (timeout_id_t)-1) {
```



```
1238                         (void) untimeout(instance->timeout_id);
1239                         instance->timeout_id = (timeout_id_t)-1;

1241                         instance->unroll.timer = 0;
1242                 }
1243         }

1245         instance->func_ptr->disable_intr(instance);


1248         if (instance->unroll.mutexs == 1) {
1249                 mutex_destroy(&instance->cmd_pool_mtx);
1250                 mutex_destroy(&instance->app_cmd_pool_mtx);
1251                 mutex_destroy(&instance->cmd_pend_mtx);
1252                 mutex_destroy(&instance->completed_pool_mtx);
1253                 mutex_destroy(&instance->sync_map_mtx);
1254                 mutex_destroy(&instance->int_cmd_mtx);
1255                 cv_destroy(&instance->int_cmd_cv);
1256                 mutex_destroy(&instance->config_dev_mtx);
1257                 mutex_destroy(&instance->ocr_flags_mtx);
1258                 mutex_destroy(&instance->reg_write_mtx);

1260                 if (instance->tbolt) {
1261                         mutex_destroy(&instance->cmd_app_pool_mtx);
1262                         mutex_destroy(&instance->chip_mtx);
1263                 }

1265                 instance->unroll.mutexs = 0;
1266                 con_log(CL_ANN1, (CE_CONT, "Destroy mutex & cv,  done."));
1267         }


1270         if (instance->unroll.soft_isr == 1) {
1271                 ddi_remove_softintr(instance->soft_intr_id);
1272                 instance->unroll.soft_isr = 0;
1273         }

1275         if (instance->unroll.intr == 1) {
1276                 mrsas_rem_intrs(instance);
1277                 instance->unroll.intr = 0;
1278         }


1281         if (instance->unroll.taskq == 1)         {
1282                 if (instance->taskq) {
1283                         ddi_taskq_destroy(instance->taskq);
1284                         instance->unroll.taskq = 0;
1285                 }

1287         }

1289         /*
1290          * free dma memory allocated for
1291          * cmds/frames/queues/driver version etc
1292          */
1293         if (instance->unroll.verBuff == 1) {
1294                 (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
1295                 instance->unroll.verBuff = 0;
1296         }

1298         if (instance->unroll.pdlist_buff == 1)  {
1299                 if (instance->mr_tbolt_pd_list != NULL) {
1300                         kmem_free(instance->mr_tbolt_pd_list,
1301                             MRSAS_TBOLT_GET_PD_MAX(instance) *
1302                             sizeof (struct mrsas_tbolt_pd));
1303                 }
```

```
1305                instance->mr_tbolt_pd_list = NULL;
1306                instance->unroll.pdlist_buff = 0;
1307        }

1309        if (instance->unroll.ldlist_buff == 1)  {
1310                if (instance->mr_ld_list != NULL) {
1311                        kmem_free(instance->mr_ld_list, MRDRV_MAX_LD
1312                            * sizeof (struct mrsas_ld));
1313                }

1315                instance->mr_ld_list = NULL;
1316                instance->unroll.ldlist_buff = 0;
1317        }

1319        if (instance->tbolt) {
1320                if (instance->unroll.alloc_space_mpi2 == 1) {
1321                        free_space_for_mpi2(instance);
1322                        instance->unroll.alloc_space_mpi2 = 0;
1323                }
1324        } else {
1325                if (instance->unroll.alloc_space_mfi == 1) {
1326                        free_space_for_mfi(instance);
1327                        instance->unroll.alloc_space_mfi = 0;
1328                }
1329        }

1331        if (instance->unroll.regs == 1)  {
1332                ddi_regs_map_free(&instance->regmap_handle);
1333                instance->unroll.regs = 0;
1334                con_log(CL_ANN1, (CE_CONT, "ddi_regs_map_free()  done."));
1335        }
1336 }
```

*_____unchanged_portion_omitted_*

```
1417 /*
1418  * ioctl - performs a range of I/O commands for character drivers
1419  * @dev:
1420  * @cmd:
1421  * @arg:
1422  * @mode:
1423  * @credp:
1424  * @rvalp:
1425  *
1426  * ioctl() routine must make sure that user data is copied into or out of the
1427  * kernel address space explicitly using copyin(), copyout(), ddi_copyin(),
1428  * and ddi_copyout(), as appropriate.
1429  * This is a wrapper routine to serialize access to the actual ioctl routine.
1430  * ioctl() should return 0 on success, or the appropriate error number. The
1431  * driver may also set the value returned to the calling process through rvalp.
1432  */

1434 static int
1435 mrsas_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp,
1436     int *rvalp)
1437 {
1438        int     rval = 0;

1440        struct mrsas_instance   *instance;
1441        struct mrsas_ioctl      *ioctl;
1442        struct mrsas_aen        aen;
1443        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1445        instance = ddi_get_soft_state(mrsas_state, MINOR2INST(getminor(dev)));

1447        if (instance == NULL) {
```



```
1448                /* invalid minor number */
1449                con_log(CL_ANN, (CE_WARN, "mr_sas: adapter not found."));
1450                return (ENXIO);
1451        }

1453        ioctl = (struct mrsas_ioctl *)kmem_zalloc(sizeof (struct mrsas_ioctl),
1454            KM_SLEEP);
1455        if (ioctl == NULL) {
1456                /* Failed to allocate memory for ioctl  */
1457                con_log(CL_ANN, (CE_WARN, "mr_sas_ioctl: "
1458                    "failed to allocate memory for ioctl"));
1459                return (ENOMEM);
1477                return (ENXIO);
1460        }

1462        switch ((uint_t)cmd) {
1463                case MRSAS_IOCTL_FIRMWARE:
1464                        if (ddi_copyin((void *)arg, ioctl,
1465                            sizeof (struct mrsas_ioctl), mode)) {
1466                                con_log(CL_ANN, (CE_WARN, "mrsas_ioctl: "
1467                                    "ERROR IOCTL copyin"));
1468                                kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1469                                return (EFAULT);
1470                        }

1472                        if (ioctl->control_code == MRSAS_DRIVER_IOCTL_COMMON) {
1473                                rval = handle_drv_ioctl(instance, ioctl, mode);
1474                        } else {
1475                                rval = handle_mfi_ioctl(instance, ioctl, mode);
1476                        }

1478                        if (ddi_copyout((void *)ioctl, (void *)arg,
1479                            (sizeof (struct mrsas_ioctl) - 1), mode)) {
1480                                con_log(CL_ANN, (CE_WARN,
1481                                    "mrsas_ioctl: copy_to_user failed"));
1482                                rval = 1;
1483                        }

1485                        break;
1486                case MRSAS_IOCTL_AEN:
1487                        con_log(CL_ANN,
1488                            (CE_NOTE, "mrsas_ioctl: IOCTL Register AEN.\n"));

1490                        if (ddi_copyin((void *) arg, &aen,
1491                            sizeof (struct mrsas_aen), mode)) {
1492                                con_log(CL_ANN, (CE_WARN,
1493                                    "mrsas_ioctl: ERROR AEN copyin"));
1494                                kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1495                                return (EFAULT);
1496                        }

1498                        rval = handle_mfi_aen(instance, &aen);

1500                        if (ddi_copyout((void *) &aen, (void *)arg,
1501                            sizeof (struct mrsas_aen), mode)) {
1502                                con_log(CL_ANN, (CE_WARN,
1503                                    "mrsas_ioctl: copy_to_user failed"));
1504                                rval = 1;
1505                        }

1507                        break;
1508                default:
1509                        rval = scsi_hba_ioctl(dev, cmd, arg,
1510                            mode, credp, rvalp);

1512                        con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_ioctl: "
```

```
1513                            "scsi_hba_ioctl called, ret = %x.", rval));
1514        }

1516        kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1517        return (rval);
1518 }
_____unchanged_portion_omitted_


2509 /*
2510  * ************************************************************************** *
2511  *                                                                            *
2512  *                                  libraries                                 *
2513  *                                                                            *
2514  * ************************************************************************** *
2515  */
2516 /*
2517  * get_mfi_pkt : Get a command from the free pool
2518  * After successful allocation, the caller of this routine
2519  * must clear the frame buffer (memset to zero) before
2520  * using the packet further.
2521  *
2522  * ***** Note *****
2523  * After clearing the frame buffer the context id of the
2524  * frame buffer SHOULD be restored back.
2525  */
2526 static struct mrsas_cmd *
2527 get_mfi_pkt(struct mrsas_instance *instance)
2528 {
2529        mlist_t                 *head = &instance->cmd_pool_list;
2530        struct mrsas_cmd        *cmd = NULL;

2532        mutex_enter(&instance->cmd_pool_mtx);
2551        ASSERT(mutex_owned(&instance->cmd_pool_mtx));

2534        if (!mlist_empty(head)) {
2535                cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2536                mlist_del_init(head->next);
2537        }
2538        if (cmd != NULL) {
2539                cmd->pkt = NULL;
2540                cmd->retry_count_for_ocr = 0;
2541                cmd->drv_pkt_time = 0;

2543        }
2544        mutex_exit(&instance->cmd_pool_mtx);

2546        return (cmd);
2547 }

2549 static struct mrsas_cmd *
2550 get_mfi_app_pkt(struct mrsas_instance *instance)
2551 {
2552        mlist_t                         *head = &instance->app_cmd_pool_list;
2553        struct mrsas_cmd        *cmd = NULL;

2555        mutex_enter(&instance->app_cmd_pool_mtx);
2575        ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2557        if (!mlist_empty(head)) {
2558                cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2559                mlist_del_init(head->next);
2560        }
2561        if (cmd != NULL) {
2562                cmd->pkt = NULL;
2563                cmd->retry_count_for_ocr = 0;
```



```
2564                cmd->drv_pkt_time = 0;
2565        }

2567        mutex_exit(&instance->app_cmd_pool_mtx);

2569        return (cmd);
2570 }
2571 /*
2572  * return_mfi_pkt : Return a cmd to free command pool
2573  */
2574 static void
2575 return_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2576 {
2577        mutex_enter(&instance->cmd_pool_mtx);
2598        ASSERT(mutex_owned(&instance->cmd_pool_mtx));
2578        /* use mlist_add_tail for debug assistance */
2579        mlist_add_tail(&cmd->list, &instance->cmd_pool_list);

2581        mutex_exit(&instance->cmd_pool_mtx);
2582 }

2584 static void
2585 return_mfi_app_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2586 {
2587        mutex_enter(&instance->app_cmd_pool_mtx);
2609        ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2589        mlist_add(&cmd->list, &instance->app_cmd_pool_list);

2591        mutex_exit(&instance->app_cmd_pool_mtx);
2592 }
2593 void
2594 push_pending_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2595 {
2596        struct scsi_pkt *pkt;
2597        struct mrsas_header     *hdr;
2598        con_log(CL_DLEVEL2, (CE_NOTE, "push_pending_pkt(): Called\n"));
2599        mutex_enter(&instance->cmd_pend_mtx);
2622        ASSERT(mutex_owned(&instance->cmd_pend_mtx));
2600        mlist_del_init(&cmd->list);
2601        mlist_add_tail(&cmd->list, &instance->cmd_pend_list);
2602        if (cmd->sync_cmd == MRSAS_TRUE) {
2603                hdr = (struct mrsas_header *)&cmd->frame->hdr;
2604                if (hdr) {
2605                        con_log(CL_ANN1, (CE_CONT,
2606                            "push_pending_mfi_pkt: "
2607                            "cmd %p index %x "
2608                            "time %llx",
2609                            (void *)cmd, cmd->index,
2610                            gethrtime()));
2611                        /* Wait for specified interval  */
2612                        cmd->drv_pkt_time = ddi_get16(
2613                            cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2614                        if (cmd->drv_pkt_time < debug_timeout_g)
2615                                cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2616                                con_log(CL_ANN1, (CE_CONT,
2617                                    "push_pending_pkt(): "
2618                                    "Called IO Timeout Value %x\n",
2619                                    cmd->drv_pkt_time));
2620                }
2621                if (hdr && instance->timeout_id == (timeout_id_t)-1) {
2622                        instance->timeout_id = timeout(io_timeout_checker,
2623                            (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2624                }
2625        } else {
2626                pkt = cmd->pkt;
```

```
2627                 if (pkt) {
2628                         con_log(CL_ANN1, (CE_CONT,
2629                             "push_pending_mfi_pkt: "
2630                             "cmd %p index %x pkt %p, "
2631                             "time %llx",
2632                             (void *)cmd, cmd->index, (void *)pkt,
2633                             gethrtime()));
2634                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2635                 }
2636                 if (pkt && instance->timeout_id == (timeout_id_t)-1) {
2637                         instance->timeout_id = timeout(io_timeout_checker,
2638                             (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2639                 }
2640         }

2642         mutex_exit(&instance->cmd_pend_mtx);

2644 }

2646 int
2647 mrsas_print_pending_cmds(struct mrsas_instance *instance)
2648 {
2649         mlist_t *head = &instance->cmd_pend_list;
2650         mlist_t *tmp = head;
2651         struct mrsas_cmd *cmd = NULL;
2652         struct mrsas_header     *hdr;
2653         unsigned int            flag = 1;
2654         struct scsi_pkt *pkt;
2655         int saved_level;
2656         int cmd_count = 0;


2658         saved_level = debug_level_g;
2659         debug_level_g = CL_ANN1;

2661         cmn_err(CE_NOTE, "mrsas_print_pending_cmds(): Called\n");

2663         while (flag) {
2664                 mutex_enter(&instance->cmd_pend_mtx);
2665                 tmp     =       tmp->next;
2666                 if (tmp == head) {
2667                         mutex_exit(&instance->cmd_pend_mtx);
2668                         flag = 0;
2669                         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds():"
2670                             " NO MORE CMDS PENDING....\n"));
2671                         break;
2672                 } else {
2673                         cmd = mlist_entry(tmp, struct mrsas_cmd, list);
2674                         mutex_exit(&instance->cmd_pend_mtx);
2675                         if (cmd) {
2676                                 if (cmd->sync_cmd == MRSAS_TRUE) {
2677                                         hdr = (struct mrsas_header *)
2678                                             &cmd->frame->hdr;
2679                                         if (hdr) {
2680                                                 con_log(CL_ANN1, (CE_CONT,
2681                                                     "print: cmd %p index 0x%x "
2682                                                     "drv_pkt_time 0x%x (NO-PKT)"
2683                                                     " hdr %p\n", (void *)cmd,
2684                                                     cmd->index,
2685                                                     cmd->drv_pkt_time,
2686                                                     (void *)hdr));
2687                                         }
2688                                 } else {
2689                                         pkt = cmd->pkt;
2690                                         if (pkt) {
2691                                         con_log(CL_ANN1, (CE_CONT,
```



```
2692                                             "print: cmd %p index 0x%x "
2693                                             "drv_pkt_time 0x%x pkt %p \n",
2694                                             (void *)cmd, cmd->index,
2695                                             cmd->drv_pkt_time, (void *)pkt));
2696                                         }
2697                                 }

2699                                 if (++cmd_count == 1) {
2700                                         mrsas_print_cmd_details(instance, cmd,
2701                                             0xDD);
2702                                 } else {
2703                                         mrsas_print_cmd_details(instance, cmd,
2704                                             1);
2705                                 }

2707                         }
2708                 }
2709         }
2710         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds(): Done\n"));


2713         debug_level_g = saved_level;

2715         return (DDI_SUCCESS);
2716 }
```

_____***unchanged_portion_omitted_***

```
3624 /*
3625  * mrsas_init_adapter_ppc - Initialize MFI interface adapter.
3626  */
3627 int
3628 mrsas_init_adapter_ppc(struct mrsas_instance *instance)
3629 {
3630         struct mrsas_cmd                *cmd;

3632         /*
3633          * allocate memory for mfi adapter(cmd pool, individual commands, mfi
3634          * frames etc
3635          */
3636         if (alloc_space_for_mfi(instance) != DDI_SUCCESS) {
3637                 con_log(CL_ANN, (CE_NOTE,
3638                     "Error, failed to allocate memory for MFI adapter"));
3639                 return (DDI_FAILURE);
3640         }

3642         /* Build INIT command */
3643         cmd = get_mfi_pkt(instance);

3645         if (mrsas_build_init_cmd(instance, &cmd) != DDI_SUCCESS) {
3646                 con_log(CL_ANN,
3647                     (CE_NOTE, "Error, failed to build INIT command"));

3649                 goto fail_undo_alloc_mfi_space;
3650         }

3652         /*
3653          * Disable interrupt before sending init frame ( see linux driver code)
3654          * send INIT MFI frame in polled mode
3655          */
3656         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3657                 con_log(CL_ANN, (CE_WARN, "failed to init firmware"));
3658                 goto fail_fw_init;
3659         }

3661         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
```

```
3662                goto fail_fw_init;
3663        return_mfi_pkt(instance, cmd);
3687        /* return_mfi_pkt(instance, cmd); */ /* XXX KEBE ASKS, inherit? */

3665        if (ctio_enable &&
3666            (instance->func_ptr->read_fw_status_reg(instance) & 0x04000000)) {
3667                con_log(CL_ANN, (CE_NOTE, "mr_sas: IEEE SGL's supported"));
3668                instance->flag_ieee = 1;
3669        } else {
3670                instance->flag_ieee = 0;
3671        }

3673        instance->unroll.alloc_space_mfi = 1;
3674        instance->unroll.verBuff = 1;

3676        return (DDI_SUCCESS);


3679 fail_fw_init:
3680        (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

3682 fail_undo_alloc_mfi_space:
3683        return_mfi_pkt(instance, cmd);
3684        free_space_for_mfi(instance);

3686        return (DDI_FAILURE);

3688 }
```

*unchanged_portion_omitted_*

```
4811 /*
4812  * mrsas_alloc_dma_obj
4813  *
4814  * Allocate the memory and other resources for an dma object.
4815  */
4816 int
4817 mrsas_alloc_dma_obj(struct mrsas_instance *instance, dma_obj_t *obj,
4818     uchar_t endian_flags)
4819 {
4820        int     i;
4821        size_t  alen = 0;
4822        uint_t  cookie_cnt;
4823        struct ddi_device_acc_attr tmp_endian_attr;

4825        tmp_endian_attr = endian_attr;
4826        tmp_endian_attr.devacc_attr_endian_flags = endian_flags;
4827        tmp_endian_attr.devacc_attr_access = DDI_DEFAULT_ACC;

4829        i = ddi_dma_alloc_handle(instance->dip, &obj->dma_attr,
4830            DDI_DMA_SLEEP, NULL, &obj->dma_handle);
4831        if (i != DDI_SUCCESS) {

4833                switch (i) {
4834                        case DDI_DMA_BADATTR :
4835                                con_log(CL_ANN, (CE_WARN,
4836                                "Failed ddi_dma_alloc_handle- Bad attribute"));
4837                                break;
4838                        case DDI_DMA_NORESOURCES :
4839                                con_log(CL_ANN, (CE_WARN,
4840                                "Failed ddi_dma_alloc_handle- No Resources"));
4841                                break;
4842                        default :
4843                                con_log(CL_ANN, (CE_WARN,
4844                                "Failed ddi_dma_alloc_handle: "
4845                                "unknown status %d", i));
4846                                break;
```



```
4847                }

4849                return (-1);
4850        }

4852        if ((ddi_dma_mem_alloc(obj->dma_handle, obj->size, &tmp_endian_attr,
4853            DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL,
4854            &obj->buffer, &alen, &obj->acc_handle) != DDI_SUCCESS) ||
4855            alen < obj->size) {

4857                ddi_dma_free_handle(&obj->dma_handle);

4859                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_mem_alloc"));

4861                return (-1);
4862        }
4887        if (obj->dma_handle == NULL) {
4888                /* XXX KEBE ASKS --> fm_service_impact()? */
4889                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_mem_alloc"));
4890                return (-1);
4891        }

4864        if (ddi_dma_addr_bind_handle(obj->dma_handle, NULL, obj->buffer,
4865            obj->size, DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP,
4866            NULL, &obj->dma_cookie[0], &cookie_cnt) != DDI_SUCCESS) {

4868                ddi_dma_mem_free(&obj->acc_handle);
4869                ddi_dma_free_handle(&obj->dma_handle);

4871                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_addr_bind_handle"));

4873                return (-1);
4874        }
4904        if (obj->acc_handle == NULL) {
4905                /* XXX KEBE ASKS --> fm_service_impact()? */
4906                ddi_dma_mem_free(&obj->acc_handle);
4907                ddi_dma_free_handle(&obj->dma_handle);

4909                con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_addr_bind_handle"));
4910                return (-1);
4911        }

4876        if (mrsas_check_dma_handle(obj->dma_handle) != DDI_SUCCESS) {
4877                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4878                return (-1);
4879        }

4881        if (mrsas_check_acc_handle(obj->acc_handle) != DDI_SUCCESS) {
4882                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4883                return (-1);
4884        }

4886        return (cookie_cnt);
4887 }
```

*unchanged_portion_omitted_*

```
6786 static void
6787 issue_cmd_ppc(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
6788 {
6789        struct scsi_pkt *pkt;
6790        atomic_add_16(&instance->fw_outstanding, 1);

6792        pkt = cmd->pkt;
6793        if (pkt) {
6794                con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6795                    "ISSUED CMD TO FW : called : cmd:"
```

```
6796                     ": %p instance : %p pkt : %p pkt_time : %x\n",
6797                     gethrtime(), (void *)cmd, (void *)instance,
6798                     (void *)pkt, cmd->drv_pkt_time));
6799                 if (instance->adapterresetinprogress) {
6800                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
6801                         con_log(CL_ANN1, (CE_NOTE, "Reset the scsi_pkt timer"));
6802                 } else {
6803                         push_pending_mfi_pkt(instance, cmd);
6804                 }

6806         } else {
6807                 con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6808                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
6809                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
6810         }

6812         mutex_enter(&instance->reg_write_mtx);
6850         ASSERT(mutex_owned(&instance->reg_write_mtx));
6813         /* Issue the command to the FW */
6814         WR_IB_QPORT((cmd->frame_phys_addr) |
6815             (((cmd->frame_count - 1) << 1) | 1), instance);
6816         mutex_exit(&instance->reg_write_mtx);

6818 }

6820 /*
6821  * issue_cmd_in_sync_mode
6822  */
6823 static int
6824 issue_cmd_in_sync_mode_ppc(struct mrsas_instance *instance,
6825 struct mrsas_cmd *cmd)
6826 {
6827         int     i;
6828         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * (10 * MILLISEC);
6829         struct mrsas_header *hdr = &cmd->frame->hdr;

6831         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: called"));

6833         if (instance->adapterresetinprogress) {
6834                 cmd->drv_pkt_time = ddi_get16(
6835                     cmd->frame_dma_obj.acc_handle, &hdr->timeout);
6836                 if (cmd->drv_pkt_time < debug_timeout_g)
6837                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

6839                 con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: "
6840                     "issue and return in reset case\n"));
6841                 WR_IB_QPORT((cmd->frame_phys_addr) |
6842                     (((cmd->frame_count - 1) << 1) | 1), instance);

6844                 return (DDI_SUCCESS);
6845         } else {
6846                 con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: pushing the pkt\n"));
6847                 push_pending_mfi_pkt(instance, cmd);
6848         }

6850         cmd->cmd_status = ENODATA;

6852         mutex_enter(&instance->reg_write_mtx);
6891         ASSERT(mutex_owned(&instance->reg_write_mtx));
6853         /* Issue the command to the FW */
6854         WR_IB_QPORT((cmd->frame_phys_addr) |
6855             (((cmd->frame_count - 1) << 1) | 1), instance);
6856         mutex_exit(&instance->reg_write_mtx);

6858         mutex_enter(&instance->int_cmd_mtx);
6859         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
```



```
6860                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
6861         }
6862         mutex_exit(&instance->int_cmd_mtx);

6864         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: done"));

6866         if (i < (msecs -1)) {
6867                 return (DDI_SUCCESS);
6868         } else {
6869                 return (DDI_FAILURE);
6870         }
6871 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   55863 Fri Oct 12 14:59:06 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.h
Code review comments
**********************************************************
______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          */
 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. */
 562                 uint8_t soft_isr;
 563                 uint8_t ldlist_buff;
 564                 uint8_t pdlist_buff;
 565                 uint8_t syncCmd;
 566                 uint8_t verBuff;
 567                 uint8_t alloc_space_mfi;
 568                 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 */
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 */
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 */
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;
_____unchanged_portion_omitted_

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  */
666 #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__ ""
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 */
```



```
1956 int mrsas_config_scsi_device(struct mrsas_instance *,
1957     struct scsi_device *, dev_info_t **);

1959 #ifdef PDSUPPORT
1960 int mrsas_tbolt_config_pd(struct mrsas_instance *, uint16_t,
1961     uint8_t, dev_info_t **);
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);
1964 dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t,
1965                         uint8_t);
1966 int     mrsas_service_evt(struct mrsas_instance *, int, int, int,
1967                         uint64_t);
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);

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);
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 *);
2001 void    tbolt_complete_cmd_in_sync_mode(struct mrsas_instance *,
2002         struct mrsas_cmd *);
2003 int     alloc_req_rep_desc(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 *);
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 *);

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 *);
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 *);
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);
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);


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);

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 *);

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_

****
2874 Fri Oct 12 14:59:07 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_list.c
Code review comments
****

```
/*
 * 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.
 */

/* 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;
}
```

_____unchanged_portion_omitted_

```
**********************************************************
  107669 Fri Oct 12 14:59:07 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
Code review comments
**********************************************************
______unchanged_portion_omitted_

 551 /*
 552  * mrsas_alloc_cmd_pool_tbolt
 553  *
 554  * TODO: merge tbolt-specific code into mrsas_alloc_cmd_pool() to have single
 554  * TODO: merge tbolt-specific codee into mrsas_alloc_cmd_pool() to have single
 555  * routine
 556  */
 557 int
 558 mrsas_alloc_cmd_pool_tbolt(struct mrsas_instance *instance)
 559 {
 560         int             i;
 561         int             count;
 562         uint32_t        max_cmd;
 563         uint32_t        reserve_cmd;
 564         size_t          sz;

 566         struct mrsas_cmd        *cmd;

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


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

 575         /*
 576          * instance->cmd_list is an array of struct mrsas_cmd pointers.
 577          * Allocate the dynamic array first and then allocate individual
 578          * commands.
 579          */
 580         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
 581         if (instance->cmd_list == NULL) {
 582                 con_log(CL_NONE, (CE_WARN,
 583                     "Failed to allocate memory for cmd_list"));
 584                 return (DDI_FAILURE);
 585         }

 587         /* create a frame pool and assign one frame to each cmd */
 588         for (count = 0; count < max_cmd; count++) {
 589                 instance->cmd_list[count] =
 590                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
 591                 if (instance->cmd_list[count] == NULL) {
 592                         con_log(CL_NONE, (CE_WARN,
 593                             "Failed to allocate memory for mrsas_cmd"));
 594                         goto mrsas_undo_cmds;
 595                 }
 596         }

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

 600         INIT_LIST_HEAD(&instance->cmd_pool_list);
 601         INIT_LIST_HEAD(&instance->cmd_pend_list);
 602         INIT_LIST_HEAD(&instance->cmd_app_pool_list);

 604         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

 606         /* cmd index 0 reservered for IOC INIT */
 607         for (i = 1; i < reserve_cmd; i++) {
 608                 cmd             = instance->cmd_list[i];
```



```
 609                 cmd->index      = i;
 610                 mlist_add_tail(&cmd->list, &instance->cmd_app_pool_list);
 611         }


 614         for (i = reserve_cmd; i < max_cmd; i++) {
 615                 cmd             = instance->cmd_list[i];
 616                 cmd->index      = i;
 617                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
 618         }

 620         return (DDI_SUCCESS);

 622 mrsas_undo_cmds:
 623         if (count > 0) {
 624                 /* free each cmd */
 625                 for (i = 0; i < count; i++) {
 626                         if (instance->cmd_list[i] != NULL) {
 627                                 kmem_free(instance->cmd_list[i],
 628                                     sizeof (struct mrsas_cmd));
 629                         }
 630                         instance->cmd_list[i] = NULL;
 631                 }
 632         }

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

 639         return (DDI_FAILURE);
 640 }
______unchanged_portion_omitted_

2712 uint_t
2713 mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *instance)
2714 {
2715         uint8_t                         replyType;
2716         Mpi2SCSIIOSuccessReplyDescriptor_t *replyDesc;
2717         Mpi2ReplyDescriptorsUnion_t     *desc;
2718         uint16_t                        smid;
2719         union desc_value                d_val;
2720         struct mrsas_cmd                *cmd;

2722         struct mrsas_header     *hdr;
2723         struct scsi_pkt         *pkt;

2725         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2726             0, 0, DDI_DMA_SYNC_FORDEV);

2728         (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2729             0, 0, DDI_DMA_SYNC_FORCPU);

2731         desc = instance->reply_frame_pool;
2732         desc += instance->reply_read_index;

2734         replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;
2735         replyType = replyDesc->ReplyFlags &
2736             MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2738         if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2739                 return (DDI_INTR_UNCLAIMED);

2741         if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
2742             != DDI_SUCCESS) {
2743                 mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
```

```
2744                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2745                con_log(CL_ANN1,
2746                    (CE_WARN, "mr_sas_tbolt_process_outstanding_cmd(): "
2747                    "FMA check, returning DDI_INTR_UNCLAIMED"));
2748                return (DDI_INTR_CLAIMED);
2749        }

2751        con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %p  Words = %" PRIx64 " \n",
2752            (void *)desc, desc->Words));

2754        d_val.word = desc->Words;


2757        /* Read Reply descriptor */
2758        while ((d_val.u1.low != 0xffffffff) &&
2759            (d_val.u1.high != 0xffffffff)) {

2761                (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2762                    0, 0, DDI_DMA_SYNC_FORCPU);

2764                smid = replyDesc->SMID;

2766                if (!smid || smid > instance->max_fw_cmds + 1) {
2767                        con_log(CL_ANN1, (CE_NOTE,
2768                            "Reply Desc at Break  = %p  Words = %" PRIx64 " \n",
2769                            (void *)desc, desc->Words));
2770                        break;
2771                }

2773                cmd     = instance->cmd_list[smid - 1];
2774                if (!cmd) {
2775                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_tbolt_process_"
2776                            "outstanding_cmd: Invalid command "
2777                            " or Poll commad Received in completion path\n"));
2778                } else {
2779                        mutex_enter(&instance->cmd_pend_mtx);
2780                        if (cmd->sync_cmd == MRSAS_TRUE) {
2781                                hdr = (struct mrsas_header *)&cmd->frame->hdr;
2782                                if (hdr) {
2783                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2784                                            "tbolt_process_outstanding_cmd:"
2785                                            " mlist_del_init(&cmd->list).\n"));
2786                                        mlist_del_init(&cmd->list);
2787                                }
2788                        } else {
2789                                pkt = cmd->pkt;
2790                                if (pkt) {
2791                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2792                                            "tbolt_process_outstanding_cmd:"
2793                                            "mlist_del_init(&cmd->list).\n"));
2794                                        mlist_del_init(&cmd->list);
2795                                }
2796                        }

2798                        mutex_exit(&instance->cmd_pend_mtx);

2800                        tbolt_complete_cmd(instance, cmd);
2801                }
2802                /* set it back to all 1s. */
2803                desc->Words = -1LL;
2802                /* set it back to all 0xffffffff. */
2803                desc->Words = (uint64_t)~0;

2805                instance->reply_read_index++;

2807                if (instance->reply_read_index >= (instance->reply_q_depth)) {
```



```
2808                        con_log(CL_ANN1, (CE_NOTE, "wrap around"));
2809                        instance->reply_read_index = 0;
2810                }

2812                /* Get the next reply descriptor */
2813                if (!instance->reply_read_index)
2814                        desc = instance->reply_frame_pool;
2815                else
2816                        desc++;

2818                replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;

2820                d_val.word = desc->Words;

2822                con_log(CL_ANN1, (CE_NOTE,
2823                    "Next Reply Desc  = %p Words = %" PRIx64 "\n",
2824                    (void *)desc, desc->Words));

2826                replyType = replyDesc->ReplyFlags &
2827                    MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2829                if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2830                        break;

2832        } /* End of while loop. */

2834        /* update replyIndex to FW */
2835        WR_MPI2_REPLY_POST_INDEX(instance->reply_read_index, instance);


2838        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2839            0, 0, DDI_DMA_SYNC_FORDEV);

2841        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2842            0, 0, DDI_DMA_SYNC_FORCPU);
2843        return (DDI_INTR_CLAIMED);
2844 }
```

_____***unchanged_portion_omitted_***