```
**********************************************************
    2653 Sun Mar 15 09:04:48 2015
new/usr/src/pkg/manifests/driver-storage-mr_sas.mf
5719 Add support for LSI Fury adapters
**********************************************************
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#

#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2013 Nexenta Systems, Inc. All rights reserved.
# Copyright 2015 Garrett D'Amore <garrett@damore.org>
#

#
# The default for payload-bearing actions in this package is to appear in the
# global zone only.  See the include file for greater detail, as well as
# information about overriding the defaults.
#
<include global_zone_only_component>
set name=pkg.fmri value=pkg:/driver/storage/mr_sas@$(PKGVERS)
set name=pkg.description value="LSI MegaRAID SAS2.0 Controller HBA Driver"
set name=pkg.summary value="LSI MegaRAID SAS2.0 HBA Driver"
set name=info.classification \
    value=org.opensolaris.category.2008:Drivers/Storage
set name=variant.arch value=$(ARCH)
dir path=kernel group=sys
dir path=kernel/drv group=sys
dir path=kernel/drv/$(ARCH64) group=sys
dir path=usr/share/man
dir path=usr/share/man/man7d
$(sparc_ONLY)driver name=mr_sas class=scsi-self-identifying \
    alias=pci1000,78 \
    alias=pci1000,79 \
    alias=pciex1000,5b \
    alias=pciex1000,5d \
    alias=pciex1000,5f \
    alias=pciex1000,71 \
    alias=pciex1000,73 \
    alias=pciex1000,78 \
    alias=pciex1000,79
$(i386_ONLY)driver name=mr_sas class=scsi-self-identifying \
    alias=pciex1000,5b \
    alias=pciex1000,5d \
    alias=pciex1000,5f \
    alias=pciex1000,71 \
    alias=pciex1000,73 \
    alias=pciex1000,78 \
```



```
    alias=pciex1000,79
file path=kernel/drv/$(ARCH64)/mr_sas group=sys
$(i386_ONLY)file path=kernel/drv/mr_sas group=sys
file path=kernel/drv/mr_sas.conf group=sys
file path=usr/share/man/man7d/mr_sas.7d
legacy pkg=SUNWmrsas desc="LSI MegaRAID SAS2.0 Controller HBA Driver" \
    name="LSI MegaRAID SAS2.0 HBA Driver"
license cr_Sun license=cr_Sun
license usr/src/uts/common/io/mr_sas/THIRDPARTYLICENSE \
    license=usr/src/uts/common/io/mr_sas/THIRDPARTYLICENSE
```

```
**********************************************************
   13354 Sun Mar 15 09:04:48 2015
new/usr/src/uts/common/io/mr_sas/ld_pd_map.c
5719 Add support for LSI Fury adapters
**********************************************************
```

```
   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  */
  24 /*
  25  * Copyright 2015 Garrett D'Amore <garrett@damore.org>
  26  */

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

  32 /*
  33  * This function will check if FAST IO is possible on this logical drive
  34  * by checking the EVENT information available in the driver
  35  */
  36 #define MR_LD_STATE_OPTIMAL 3
  37 #define ABS_DIFF(a, b)   (((a) > (b)) ? ((a) - (b)) : ((b) - (a)))

  39 static void mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *,
  40     PLD_LOAD_BALANCE_INFO);

  42 #define FALSE 0
  43 #define TRUE 1

  45 typedef U64     REGION_KEY;
  46 typedef U32     REGION_LEN;
  47 extern int      debug_level_g;


  50 MR_LD_RAID
  51 *MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
  52 {
  53         return (&map->raidMap.ldSpanMap[ld].ldRaid);
  54 }
```

*unchanged_portion_omitted_*

```
 184 /*
 185  * ***********************************************************
 186  *
 187  * This routine calculates the arm, span and block for
```



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

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

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

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

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

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

 248         /* Get the array on which this span is present. */
 249         arRef           = MR_LdSpanArrayGet(ld, span, map);
 250         /* Get the Pd. */
 251         pd              = MR_ArPdGet(arRef, physArm, map);
 252         /* Get dev handle from Pd. */
 253         if (pd != MR_PD_INVALID) {
```

```
254                 *pDevHandle     = MR_PdDevHandleGet(pd, map);
255         } else {
256                 *pDevHandle = MR_PD_INVALID; /* set dev handle as invalid. */
257                 if ((raid->level >= 5) &&
258                     ((devid != PCI_DEVICE_ID_LSI_INVADER) ||
259                     ((devid == PCI_DEVICE_ID_LSI_INVADER ||
260                     (devid == PCI_DEVICE_ID_LSI_FURY)) &&
254                     ((instance->device_id != PCI_DEVICE_ID_LSI_INVADER) ||
255                     (instance->device_id == PCI_DEVICE_ID_LSI_INVADER &&
261                     raid->regTypeReqOnRead != REGION_TYPE_UNUSED))) {
262                         pRAID_Context->regLockFlags = REGION_TYPE_EXCLUSIVE;
263                 } else if (raid->level == 1) {
264                         /* Get Alternate Pd. */
265                         pd = MR_ArPdGet(arRef, physArm + 1, map);
266                         /* Get dev handle from Pd. */
267                         if (pd != MR_PD_INVALID)
268                                 *pDevHandle = MR_PdDevHandleGet(pd, map);
269                 }
270         }

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

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

277         return (retval);
278 }



282 /*
283  * ***************************************************************
284  *
285  * MR_BuildRaidContext function
286  *
287  * This function will initiate command processing.  The start/end row and strip
288  * information is calculated then the lock is acquired.
289  * This function will return 0 if region lock
290  * was acquired OR return num strips ???
291  */

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

308         U64             ldStartBlock;
309         U32             numBlocks, ldTgtId;
310         U8              isRead;
311         U8              retval = 0;

313         ldStartBlock = io_info->ldStartBlock;
314         numBlocks = io_info->numBlocks;
315         ldTgtId = io_info->ldTgtId;
316         isRead = io_info->isRead;
```



```
318         if (map == NULL) {
319                 io_info->fpOkForIo = FALSE;
320                 return (FALSE);
321         }

323         ld = MR_TargetIdToLdGet(ldTgtId, map);

325         if (ld >= MAX_LOGICAL_DRIVES) {
326                 io_info->fpOkForIo = FALSE;
327                 return (FALSE);
328         }

330         raid = MR_LdRaidGet(ld, map);

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

351         /*
352          * calculate region info.
353          */
354         regStart        = start_row << raid->stripeShift;
355         regSize         = stripSize;

357         /* Check if we can send this I/O via FastPath */
358         if (raid->capability.fpCapable) {
359                 if (isRead) {
360                         io_info->fpOkForIo = (raid->capability.fpReadCapable &&
361                             ((num_strips == 1) ||
362                             raid->capability.fpReadAcrossStripe));
363                 } else {
364                         io_info->fpOkForIo =
365                             (raid->capability.fpWriteCapable &&
366                             ((num_strips == 1) ||
367                             raid->capability.fpWriteAcrossStripe));
368                 }
369         } else
370                 io_info->fpOkForIo = FALSE;


373         /*
374          * Check for DIF support
375          */
376         if (!raid->capability.ldPiMode) {
377                 io_info->ldPI = FALSE;
378         } else {
379                 io_info->ldPI = TRUE;
380         }

382         if (numRows == 1) {
383                 if (num_strips == 1) {
```

```
384                         regStart += ref_in_start_stripe;
385                         regSize = numBlocks;
386                 }
387         } else {
388                 if (start_strip == (start_row + 1) * raid->rowDataSize - 1) {
389                         regStart += ref_in_start_stripe;
390                 regSize = stripSize - ref_in_start_stripe;
391                 }

393                 if (numRows > 2) {
394                         regSize += (numRows - 2) << raid->stripeShift;
395                 }

397                 if (endStrip == endRow * raid->rowDataSize) {
398                         regSize += ref_in_end_stripe + 1;
399                 } else {
400                         regSize += stripSize;
401                 }
402         }

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

406         if ((instance->device_id == PCI_DEVICE_ID_LSI_INVADER) ||
407             (instance->device_id == PCI_DEVICE_ID_LSI_FURY)) {
401         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
408                 pRAID_Context->regLockFlags = (isRead) ?
409                     raid->regTypeReqOnRead : raid->regTypeReqOnWrite;
410         } else {
411                 pRAID_Context->regLockFlags = (isRead) ?
412                     REGION_TYPE_SHARED_READ : raid->regTypeReqOnWrite;
413         }

415         pRAID_Context->ldTargetId = raid->targetId;
416         pRAID_Context->regLockRowLBA = regStart;
417         pRAID_Context->regLockLength = regSize;
418         pRAID_Context->configSeqNum = raid->seqNum;

420         /*
421          * Get Phy Params only if FP capable,
422          * or else leave it to MR firmware to do the calculation.
423          */
424         if (io_info->fpOkForIo) {
425                 /* if fast path possible then get the physical parameters */
426                 retval = MR_GetPhyParams(instance, ld, start_strip,
427                     ref_in_start_stripe, &io_info->pdBlock,
428                     &io_info->devHandle, pRAID_Context, map);

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

434                 return (retval);

436         } else if (isRead) {
437                 uint_t stripIdx;

439                 for (stripIdx = 0; stripIdx < num_strips; stripIdx++) {
440                         if (!MR_GetPhyParams(instance, ld,
441                             start_strip + stripIdx, ref_in_start_stripe,
442                             &io_info->pdBlock, &io_info->devHandle,
443                             pRAID_Context, map)) {
444                                 return (TRUE);
445                         }
446                 }
447         }
448         return (TRUE);
```



```
449 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  221625 Sun Mar 15 09:04:49 2015
new/usr/src/uts/common/io/mr_sas/mr_sas.c
5719 Add support for LSI Fury adapters
**********************************************************
```

```
/*
 * 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 2013 Nexenta Systems, Inc. All rights reserved.
 * Copyright 2015 Garrett D'Amore <garrett@damore.org>
 */

#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>

/* Macros to help Skinny and stock 2108/MFI live together. */
#define WR_IB_PICK_QPORT(addr, instance) \
        if ((instance)->skinny) { \
                WR_IB_LOW_QPORT((addr), (instance)); \
                WR_IB_HIGH_QPORT(0, (instance)); \
        } else { \
                WR_IB_QPORT((addr), (instance)); \
        }

/*
 * 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

#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 *);
```

```
128 static int      mrsas_tran_tgt_init(dev_info_t *, dev_info_t *,
129                       scsi_hba_tran_t *, struct scsi_device *);
130 static struct scsi_pkt *mrsas_tran_init_pkt(struct scsi_address *, register
131                       struct scsi_pkt *, struct buf *, int, int, int, int,
132                       int (*)(), caddr_t);
133 static int      mrsas_tran_start(struct scsi_address *,
134                       register struct scsi_pkt *);
135 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
136 static int      mrsas_tran_reset(struct scsi_address *, int);
137 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
138 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
139 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
140                       struct scsi_pkt *);
141 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
142 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
143 static int      mrsas_tran_quiesce(dev_info_t *dip);
144 static int      mrsas_tran_unquiesce(dev_info_t *dip);
145 static uint_t   mrsas_isr();
146 static uint_t   mrsas_softintr();
147 static void     mrsas_undo_resources(dev_info_t *, struct mrsas_instance *);

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 };
_____unchanged_portion_omitted_


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

```
463         uint16_t        device_id;
464         uint16_t        subsysvid;
465         uint16_t        subsysid;
466         uint16_t        command;
467         off_t           reglength = 0;
468         int             intr_types = 0;
469         char            *data;

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

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

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

480         instance_no = ddi_get_instance(dip);

482         /*
483          * check to see whether this device is in a DMA-capable slot.
484          */
485         if (ddi_slaveonly(dip) == DDI_SUCCESS) {
486                 cmn_err(CE_WARN,
487                     "mr_sas%d: Device in slave-only slot, unused",
488                     instance_no);
489                 return (DDI_FAILURE);
490         }

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

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

506                 if (instance == NULL) {
507                         cmn_err(CE_WARN,
508                             "mr_sas%d: Bad soft state", instance_no);
509                         ddi_soft_state_free(mrsas_state, instance_no);
510                         return (DDI_FAILURE);
511                 }

513                 instance->unroll.softs  = 1;

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

522                         ddi_soft_state_free(mrsas_state, instance_no);
523                         return (DDI_FAILURE);
524                 }

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



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

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

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

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

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

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

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

563                         pci_config_put16(instance->pci_handle,
564                             PCI_CONF_COMM, command);

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

573                 /* initialize function pointers */
574                 switch (device_id) {
575                 case PCI_DEVICE_ID_LSI_TBOLT:
576                 case PCI_DEVICE_ID_LSI_INVADER:
577                 case PCI_DEVICE_ID_LSI_FURY:
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_SKINNY:
587                 case PCI_DEVICE_ID_LSI_SKINNY_NEW:
588                         /*
589                          * FALLTHRU to PPC-style functions, but mark this
590                          * instance as Skinny, because the register set is
591                          * slightly different (See WR_IB_PICK_QPORT), and
592                          * certain other features are available to a Skinny
593                          * HBA.
594                          */
```

```
595                         instance->skinny = 1;
596                         /* FALLTHRU */
597
598                 case PCI_DEVICE_ID_LSI_2108VDE:
599                 case PCI_DEVICE_ID_LSI_2108V:
600                         con_log(CL_ANN, (CE_NOTE,
601                             "mr_sas: 2108 Liberator device detected"));
602
603                         instance->func_ptr =
604                             &mrsas_function_template_ppc;
605                         break;
606
607                 default:
608                         cmn_err(CE_WARN,
609                             "mr_sas: Invalid device detected");
610
611                         pci_config_teardown(&instance->pci_handle);
612                         ddi_soft_state_free(mrsas_state, instance_no);
613                         return (DDI_FAILURE);
614                 }
615
616                 instance->baseaddress = pci_config_get32(
617                     instance->pci_handle, PCI_CONF_BASE0);
618                 instance->baseaddress &= 0x0fffc;
619
620                 instance->dip           = dip;
621                 instance->vendor_id     = vendor_id;
622                 instance->device_id     = device_id;
623                 instance->subsysvid     = subsysvid;
624                 instance->subsysid      = subsysid;
625                 instance->instance      = instance_no;
626
627                 /* Initialize FMA */
628                 instance->fm_capabilities = ddi_prop_get_int(
629                     DDI_DEV_T_ANY, instance->dip, DDI_PROP_DONTPASS,
630                     "fm-capable", DDI_FM_EREPORT_CAPABLE |
631                     DDI_FM_ACCCHK_CAPABLE | DDI_FM_DMACHK_CAPABLE
632                     | DDI_FM_ERRCB_CAPABLE);
633
634                 mrsas_fm_init(instance);
635
636                 /* Setup register map */
637                 if ((ddi_dev_regsize(instance->dip,
638                     REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
639                     reglength < MINIMUM_MFI_MEM_SZ) {
640                         goto fail_attach;
641                 }
642                 if (reglength > DEFAULT_MFI_MEM_SZ) {
643                         reglength = DEFAULT_MFI_MEM_SZ;
644                         con_log(CL_DLEVEL1, (CE_NOTE,
645                             "mr_sas: register length to map is 0x%lx bytes",
646                             reglength));
647                 }
648                 if (ddi_regs_map_setup(instance->dip,
649                     REGISTER_SET_IO_2108, &instance->regmap, 0,
650                     reglength, &endian_attr, &instance->regmap_handle)
651                     != DDI_SUCCESS) {
652                         cmn_err(CE_WARN,
653                             "mr_sas: couldn't map control registers");
654                         goto fail_attach;
655                 }
656
657                 instance->unroll.regs = 1;
658
659                 /*
660                  * Disable Interrupt Now.
```



```
661                  * Setup Software interrupt
662                  */
663                 instance->func_ptr->disable_intr(instance);
664
665                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
666                     "mrsas-enable-msi", &data) == DDI_SUCCESS) {
667                         if (strncmp(data, "no", 3) == 0) {
668                                 msi_enable = 0;
669                                 con_log(CL_ANN1, (CE_WARN,
670                                     "msi_enable = %d disabled", msi_enable));
671                         }
672                         ddi_prop_free(data);
673                 }
674
675                 con_log(CL_DLEVEL1, (CE_NOTE, "msi_enable = %d", msi_enable));
676
677                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
678                     "mrsas-enable-fp", &data) == DDI_SUCCESS) {
679                         if (strncmp(data, "no", 3) == 0) {
680                                 enable_fp = 0;
681                                 cmn_err(CE_NOTE,
682                                     "enable_fp = %d, Fast-Path disabled.\n",
683                                     enable_fp);
684                         }
685
686                         ddi_prop_free(data);
687                 }
688
689                 con_log(CL_DLEVEL1, (CE_NOTE, "enable_fp = %d\n", enable_fp));
690
691                 /* Check for all supported interrupt types */
692                 if (ddi_intr_get_supported_types(
693                     dip, &intr_types) != DDI_SUCCESS) {
694                         cmn_err(CE_WARN,
695                             "ddi_intr_get_supported_types() failed");
696                         goto fail_attach;
697                 }
698
699                 con_log(CL_DLEVEL1, (CE_NOTE,
700                     "ddi_intr_get_supported_types() ret: 0x%x", intr_types));
701
702                 /* Initialize and Setup Interrupt handler */
703                 if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
704                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
705                             DDI_SUCCESS) {
706                                 cmn_err(CE_WARN,
707                                     "MSIX interrupt query failed");
708                                 goto fail_attach;
709                         }
710                         instance->intr_type = DDI_INTR_TYPE_MSIX;
711                 } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
712                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
713                             DDI_SUCCESS) {
714                                 cmn_err(CE_WARN,
715                                     "MSI interrupt query failed");
716                                 goto fail_attach;
717                         }
718                         instance->intr_type = DDI_INTR_TYPE_MSI;
719                 } else if (intr_types & DDI_INTR_TYPE_FIXED) {
720                         msi_enable = 0;
721                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
722                             DDI_SUCCESS) {
723                                 cmn_err(CE_WARN,
724                                     "FIXED interrupt query failed");
725                                 goto fail_attach;
726                         }
```

```
727                     instance->intr_type = DDI_INTR_TYPE_FIXED;
728             } else {
729                     cmn_err(CE_WARN, "Device cannot "
730                         "suppport either FIXED or MSI/X "
731                         "interrupts");
732                     goto fail_attach;
733             }

735             instance->unroll.intr = 1;

737             if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
738                 "mrsas-enable-ctio", &data) == DDI_SUCCESS) {
739                     if (strncmp(data, "no", 3) == 0) {
740                             ctio_enable = 0;
741                             con_log(CL_ANN1, (CE_WARN,
742                                 "ctio_enable = %d disabled", ctio_enable));
743                     }
744                     ddi_prop_free(data);
745             }

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

749             /* setup the mfi based low level driver */
750             if (mrsas_init_adapter(instance) != DDI_SUCCESS) {
751                     cmn_err(CE_WARN, "mr_sas: "
752                         "could not initialize the low level driver");

754                     goto fail_attach;
755             }

757             /* Initialize all Mutex */
758             INIT_LIST_HEAD(&instance->completed_pool_list);
759             mutex_init(&instance->completed_pool_mtx, NULL,
760                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

762             mutex_init(&instance->sync_map_mtx, NULL,
763                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

765             mutex_init(&instance->app_cmd_pool_mtx, NULL,
766                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

768             mutex_init(&instance->config_dev_mtx, NULL,
769                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

771             mutex_init(&instance->cmd_pend_mtx, NULL,
772                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

774             mutex_init(&instance->ocr_flags_mtx, NULL,
775                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

777             mutex_init(&instance->int_cmd_mtx, NULL,
778                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
779             cv_init(&instance->int_cmd_cv, NULL, CV_DRIVER, NULL);

781             mutex_init(&instance->cmd_pool_mtx, NULL,
782                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

784             mutex_init(&instance->reg_write_mtx, NULL,
785                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

787             if (instance->tbolt) {
788                     mutex_init(&instance->cmd_app_pool_mtx, NULL,
789                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

791                     mutex_init(&instance->chip_mtx, NULL,
792                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
```



```
794             }

796             instance->unroll.mutexs = 1;

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

800             /* Register our soft-isr for highlevel interrupts. */
801             instance->isr_level = instance->intr_pri;
802             if (!(instance->tbolt)) {
803                     if (instance->isr_level == HIGH_LEVEL_INTR) {
804                             if (ddi_add_softintr(dip,
805                                 DDI_SOFTINT_HIGH,
806                                 &instance->soft_intr_id, NULL, NULL,
807                                 mrsas_softintr, (caddr_t)instance) !=
808                                 DDI_SUCCESS) {
809                                     cmn_err(CE_WARN,
810                                         "Software ISR did not register");

812                                     goto fail_attach;
813                             }

815                             instance->unroll.soft_isr = 1;

817                     }
818             }

820             instance->softint_running = 0;

822             /* Allocate a transport structure */
823             tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

825             if (tran == NULL) {
826                     cmn_err(CE_WARN,
827                         "scsi_hba_tran_alloc failed");
828                     goto fail_attach;
829             }

831             instance->tran = tran;
832             instance->unroll.tran = 1;

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

854             if (mrsas_relaxed_ordering)
855                     mrsas_generic_dma_attr.dma_attr_flags |=
856                         DDI_DMA_RELAXED_ORDERING;
```

```
859                 tran_dma_attr = mrsas_generic_dma_attr;
860                 tran_dma_attr.dma_attr_sgllen = instance->max_num_sge;

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

868                         goto fail_attach;
869                 }
870                 instance->unroll.tranSetup = 1;
871                 con_log(CL_ANN1,
872                     (CE_CONT, "scsi_hba_attach_setup()  done."));

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

881                         goto fail_attach;
882                 }

884                 instance->unroll.devctl = 1;

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

893                         goto fail_attach;
894                 }

896                 instance->unroll.scsictl = 1;

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

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

911                         goto fail_attach;
912                 }

914                 instance->unroll.ioctl = 1;

916                 /* Create a taskq to handle dr events */
917                 if ((instance->taskq = ddi_taskq_create(dip,
918                     "mrsas_dr_taskq", 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
919                         cmn_err(CE_WARN,
920                             "mr_sas: failed to create taskq ");
921                         instance->taskq = NULL;
922                         goto fail_attach;
923                 }
924                 instance->unroll.taskq = 1;
```



```
925                 con_log(CL_ANN1, (CE_CONT, "ddi_taskq_create()  done."));

927                 /* enable interrupt */
928                 instance->func_ptr->enable_intr(instance);

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

940                 /* Finally! We are on the air.  */
941                 ddi_report_dev(dip);

943                 /* FMA handle checking. */
944                 if (mrsas_check_acc_handle(instance->regmap_handle) !=
945                     DDI_SUCCESS) {
946                         goto fail_attach;
947                 }
948                 if (mrsas_check_acc_handle(instance->pci_handle) !=
949                     DDI_SUCCESS) {
950                         goto fail_attach;
951                 }

953                 instance->mr_ld_list =
954                     kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
955                     KM_SLEEP);
956                 instance->unroll.ldlist_buff = 1;

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

972                         instance->unroll.pdlist_buff = 1;
973                 }
974 #endif
975                 break;
976         case DDI_PM_RESUME:
977                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_PM_RESUME"));
978                 break;
979         case DDI_RESUME:
980                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_RESUME"));
981                 break;
982         default:
983                 con_log(CL_ANN,
984                     (CE_WARN, "mr_sas: invalid attach cmd=%x", cmd));
985                 return (DDI_FAILURE);
986         }


989         con_log(CL_DLEVEL1,
990             (CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
```

```
 991             instance_no));
 992         return (DDI_SUCCESS);

 994 fail_attach:

 996         mrsas_undo_resources(dip, instance);

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

1001         mrsas_fm_fini(instance);

1003         pci_config_teardown(&instance->pci_handle);
1004         ddi_soft_state_free(mrsas_state, instance_no);

1006         con_log(CL_ANN, (CE_WARN, "mr_sas: return failure from mrsas_attach"));

1008         cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d",
1009             instance_no);

1011         return (DDI_FAILURE);
1012 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   56298 Sun Mar 15 09:04:49 2015
new/usr/src/uts/common/io/mr_sas/mr_sas.h
5719 Add support for LSI Fury adapters
**********************************************************
```

```
   1 /*
   2  * mr_sas.h: header for mr_sas
   3  *
   4  * Solaris MegaRAID driver for SAS2.0 controllers
   5  * Copyright (c) 2008-2012, LSI Logic Corporation.
   6  * All rights reserved.
   7  *
   8  * Version:
   9  * Author:
  10  *              Swaminathan K S
  11  *              Arun Chandrashekhar
  12  *              Manju R
  13  *              Rasheed
  14  *              Shakeel Bukhari
  15  *
  16  * Redistribution and use in source and binary forms, with or without
  17  * modification, are permitted provided that the following conditions are met:
  18  *
  19  * 1. Redistributions of source code must retain the above copyright notice,
  20  *    this list of conditions and the following disclaimer.
  21  *
  22  * 2. Redistributions in binary form must reproduce the above copyright notice,
  23  *    this list of conditions and the following disclaimer in the documentation
  24  *    and/or other materials provided with the distribution.
  25  *
  26  * 3. Neither the name of the author nor the names of its contributors may be
  27  *    used to endorse or promote products derived from this software without
  28  *    specific prior written permission.
  29  *
  30  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  31  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  32  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  33  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
  34  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
  35  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
  36  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
  37  * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
  38  * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  39  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
  40  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
  41  * DAMAGE.
  42  */

  44 /*
  45  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
  46  * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
  47  * Copyright 2015 Garrett D'Amore <garrett@damore.org>
  48  */

  50 #ifndef _MR_SAS_H_
  51 #define _MR_SAS_H_

  53 #ifdef  __cplusplus
  54 extern "C" {
  55 #endif

  57 #include <sys/scsi/scsi.h>
  58 #include "mr_sas_list.h"
  59 #include "ld_pd_map.h"

  61 /*
```



```
  62  * MegaRAID SAS2.0 Driver meta data
  63  */
  64 #define MRSAS_VERSION                           "6.503.00.00ILLUMOS"
  65 #define MRSAS_RELDATE                           "July 30, 2012"

  67 #define MRSAS_TRUE                              1
  68 #define MRSAS_FALSE                             0

  70 #define ADAPTER_RESET_NOT_REQUIRED              0
  71 #define ADAPTER_RESET_REQUIRED                  1

  73 #define PDSUPPORT       1

  75 /*
  76  * MegaRAID SAS2.0 device id conversion definitions.
  77  */
  78 #define INST2LSIRDCTL(x)                ((x) << INST_MINOR_SHIFT)
  79 #define MRSAS_GET_BOUNDARY_ALIGNED_LEN(len, new_len, boundary_len)  { \
  80         int rem; \
  81         rem = (len / boundary_len); \
  82         if ((rem * boundary_len) != len) { \
  83                 new_len = len + ((rem + 1) * boundary_len - len); \
  84         } else { \
  85                 new_len = len; \
  86         } \
  87 }


  90 /*
  91  * MegaRAID SAS2.0 supported controllers
  92  */
  93 #define PCI_DEVICE_ID_LSI_2108VDE               0x0078
  94 #define PCI_DEVICE_ID_LSI_2108V                 0x0079
  95 #define PCI_DEVICE_ID_LSI_SKINNY                0x0071
  96 #define PCI_DEVICE_ID_LSI_SKINNY_NEW            0x0073
  97 #define PCI_DEVICE_ID_LSI_TBOLT                 0x005b
  98 #define PCI_DEVICE_ID_LSI_INVADER               0x005d
  99 #define PCI_DEVICE_ID_LSI_FURY                  0x005f

 101 /*
 102  * Register Index for 2108 Controllers.
 103  */
 104 #define REGISTER_SET_IO_2108                    (2)

 106 #define MRSAS_MAX_SGE_CNT                       0x50
 107 #define MRSAS_APP_RESERVED_CMDS                 32
 108 #define MRSAS_APP_MIN_RESERVED_CMDS             4

 110 #define MRSAS_IOCTL_DRIVER                      0x12341234
 111 #define MRSAS_IOCTL_FIRMWARE                    0x12345678
 112 #define MRSAS_IOCTL_AEN                         0x87654321

 114 #define MRSAS_1_SECOND                          1000000

 116 #ifdef PDSUPPORT

 118 #define UNCONFIGURED_GOOD                       0x0
 119 #define PD_SYSTEM                               0x40
 120 #define MR_EVT_PD_STATE_CHANGE                  0x0072
 121 #define MR_EVT_PD_REMOVED_EXT           0x00f8
 122 #define MR_EVT_PD_INSERTED_EXT          0x00f7
 123 #define MR_DCMD_PD_GET_INFO                     0x02020000
 124 #define MRSAS_TBOLT_PD_LUN              1
 125 #define MRSAS_TBOLT_PD_TGT_MAX  255
 126 #define MRSAS_TBOLT_GET_PD_MAX(s)       ((s)->mr_tbolt_pd_max)
```

```
 128 #endif

 130 /* Raid Context Flags */
 131 #define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT 0x4
 132 #define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_MASK 0x30
 133 typedef enum MR_RAID_FLAGS_IO_SUB_TYPE {
 134         MR_RAID_FLAGS_IO_SUB_TYPE_NONE = 0,
 135         MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD = 1
 136 } MR_RAID_FLAGS_IO_SUB_TYPE;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  105571 Sun Mar 15 09:04:49 2015
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
5719 Add support for LSI Fury adapters
**********************************************************
```

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

/*
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2015 Garrett D'Amore <garrett@damore.org>
 */


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

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


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



```
/* Local static prototypes. */
static struct mrsas_cmd *mrsas_tbolt_build_cmd(struct mrsas_instance *,
    struct scsi_address *, struct scsi_pkt *, uchar_t *);
static void mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len_ptr,
    U64 start_blk, U32 num_blocks);
static int mrsas_tbolt_check_map_info(struct mrsas_instance *);
static int mrsas_tbolt_sync_map_info(struct mrsas_instance *);
static int mrsas_tbolt_prepare_pkt(struct scsa_cmd *);
static int mrsas_tbolt_ioc_init(struct mrsas_instance *, dma_obj_t *);
#ifdef PDSUPPORT
static void mrsas_tbolt_get_pd_info(struct mrsas_instance *,
    struct mrsas_tbolt_pd_info *, int);
#endif /* PDSUPPORT */

static int debug_tbolt_fw_faults_after_ocr_g = 0;

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

        struct mrsas_cmd        *cmd;

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

*unchanged_portion_omitted_*

```
int
mr_sas_tbolt_build_sgl(struct mrsas_instance *instance,
    struct scsa_cmd *acmd,
    struct mrsas_cmd *cmd,
    Mpi2RaidSCSIIORequest_t *scsi_raid_io,
    uint32_t *datalen)
{
        uint32_t                MaxSGEs;
        int                     sg_to_process;
        uint32_t                i, j;
        uint32_t                numElements, endElement;
        Mpi25IeeeSgeChain64_t   *ieeeChainElement = NULL;
        Mpi25IeeeSgeChain64_t   *scsi_raid_io_sgl_ieee = NULL;
        ddi_acc_handle_t acc_handle =
            instance->mpi2_frame_pool_dma_obj.acc_handle;
        uint16_t                devid = instance->device_id;

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

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

        MaxSGEs = instance->max_sge_in_main_msg;

        ddi_put16(acc_handle, &scsi_raid_io->SGLFlags,
```

```
1247             MPI2_SGE_FLAGS_64_BIT_ADDRESSING);

1249         /* set data transfer flag. */
1250         if (acmd->cmd_flags & CFLAG_DMASEND) {
1251                 ddi_put32(acc_handle, &scsi_raid_io->Control,
1252                     MPI2_SCSIIO_CONTROL_WRITE);
1253         } else {
1254                 ddi_put32(acc_handle, &scsi_raid_io->Control,
1255                     MPI2_SCSIIO_CONTROL_READ);
1256         }


1259         numElements = acmd->cmd_cookiecnt;

1261         con_log(CL_DLEVEL1, (CE_NOTE, "[SGE Count]:%x", numElements));

1263         if (numElements > instance->max_num_sge) {
1264                 con_log(CL_ANN, (CE_NOTE,
1265                     "[Max SGE Count Exceeded]:%x", numElements));
1266                 return (numElements);
1267         }

1269         ddi_put8(acc_handle, &scsi_raid_io->RaidContext.numSGE,
1270             (uint8_t)numElements);

1272         /* set end element in main message frame */
1273         endElement = (numElements <= MaxSGEs) ? numElements : (MaxSGEs - 1);

1275         /* prepare the scatter-gather list for the firmware */
1276         scsi_raid_io_sgl_ieee =
1277             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

1279         if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1280             (devid == PCI_DEVICE_ID_LSI_FURY)) {
1277         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1281                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = scsi_raid_io_sgl_ieee;
1282                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;

1284                 ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
1285         }

1287         for (i = 0; i < endElement; i++, scsi_raid_io_sgl_ieee++) {
1288                 ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
1289                     acmd->cmd_dmacookies[i].dmac_laddress);

1291                 ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length,
1292                     acmd->cmd_dmacookies[i].dmac_size);

1294                 ddi_put8(acc_handle, &scsi_raid_io_sgl_ieee->Flags, 0);

1296                 if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1297                     (devid == PCI_DEVICE_ID_LSI_FURY)) {
1293                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1298                         if (i == (numElements - 1)) {
1299                                 ddi_put8(acc_handle,
1300                                     &scsi_raid_io_sgl_ieee->Flags,
1301                                     IEEE_SGE_FLAGS_END_OF_LIST);
1302                         }
1303                 }

1305                 *datalen += acmd->cmd_dmacookies[i].dmac_size;

1307 #ifdef DEBUG
1308                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]: %" PRIx64,
1309                     scsi_raid_io_sgl_ieee->Address));
1310                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Length]:%x",
```



```
1311                     scsi_raid_io_sgl_ieee->Length));
1312                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Flags]:%x",
1313                     scsi_raid_io_sgl_ieee->Flags));
1314 #endif

1316         }

1318         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset, 0);

1320         /* check if chained SGL required */
1321         if (i < numElements) {

1323                 con_log(CL_ANN1, (CE_NOTE, "[Chain Element index]:%x", i));

1325                 if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1326                     (devid == PCI_DEVICE_ID_LSI_FURY)) {
1321                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1327                         uint16_t ioFlags =
1328                             ddi_get16(acc_handle, &scsi_raid_io->IoFlags);

1330                         if ((ioFlags &
1331                             MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) !=
1332                             MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
1333                                 ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1334                                     (U8)instance->chain_offset_io_req);
1335                         } else {
1336                                 ddi_put8(acc_handle,
1337                                     &scsi_raid_io->ChainOffset, 0);
1338                         }
1339                 } else {
1340                         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1341                             (U8)instance->chain_offset_io_req);
1342                 }

1344                 /* prepare physical chain element */
1345                 ieeeChainElement = scsi_raid_io_sgl_ieee;

1347                 ddi_put8(acc_handle, &ieeeChainElement->NextChainOffset, 0);

1349                 if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1350                     (devid == PCI_DEVICE_ID_LSI_FURY)) {
1344                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1351                         ddi_put8(acc_handle, &ieeeChainElement->Flags,
1352                             IEEE_SGE_FLAGS_CHAIN_ELEMENT);
1353                 } else {
1354                         ddi_put8(acc_handle, &ieeeChainElement->Flags,
1355                             (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
1356                             MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
1357                 }

1359                 ddi_put32(acc_handle, &ieeeChainElement->Length,
1360                     (sizeof (MPI2_SGE_IO_UNION) * (numElements - i)));

1362                 ddi_put64(acc_handle, &ieeeChainElement->Address,
1363                     (U64)cmd->sgl_phys_addr);

1365                 sg_to_process = numElements - i;

1367                 con_log(CL_ANN1, (CE_NOTE,
1368                     "[Additional SGE Count]:%x", endElement));

1370                 /* point to the chained SGL buffer */
1371                 scsi_raid_io_sgl_ieee = (Mpi25IeeeSgeChain64_t *)cmd->sgl;

1373                 /* build rest of the SGL in chained buffer */
1374                 for (j = 0; j < sg_to_process; j++, scsi_raid_io_sgl_ieee++) {
```

```
1375                        con_log(CL_DLEVEL3, (CE_NOTE, "[remaining SGL]:%x", i));

1377                        ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
1378                            acmd->cmd_dmacookies[i].dmac_laddress);

1380                        ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length,
1381                            acmd->cmd_dmacookies[i].dmac_size);

1383                        ddi_put8(acc_handle, &scsi_raid_io_sgl_ieee->Flags, 0);

1385                        if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1386                            (devid == PCI_DEVICE_ID_LSI_FURY)) {
1379                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1387                                if (i == (numElements - 1)) {
1388                                        ddi_put8(acc_handle,
1389                                            &scsi_raid_io_sgl_ieee->Flags,
1390                                            IEEE_SGE_FLAGS_END_OF_LIST);
1391                                }
1392                        }

1394                        *datalen += acmd->cmd_dmacookies[i].dmac_size;

1396 #if DEBUG
1397                        con_log(CL_DLEVEL1, (CE_NOTE,
1398                            "[SGL Address]: %" PRIx64,
1399                            scsi_raid_io_sgl_ieee->Address));
1400                        con_log(CL_DLEVEL1, (CE_NOTE,
1401                            "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
1402                        con_log(CL_DLEVEL1, (CE_NOTE,
1403                            "[SGL Flags]:%x", scsi_raid_io_sgl_ieee->Flags));
1404 #endif

1406                        i++;
1407                }
1408        }

1410        return (0);
1411 } /*end of BuildScatterGather */


1414 /*
1415  * build_cmd
1416  */
1417 static struct mrsas_cmd *
1418 mrsas_tbolt_build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
1419     struct scsi_pkt *pkt, uchar_t *cmd_done)
1420 {
1421        uint8_t         fp_possible = 0;
1422        uint32_t        index;
1423        uint32_t        lba_count = 0;
1424        uint32_t        start_lba_hi = 0;
1425        uint32_t        start_lba_lo = 0;
1426        uint16_t        devid = instance->device_id;
1427        ddi_acc_handle_t acc_handle =
1428            instance->mpi2_frame_pool_dma_obj.acc_handle;
1429        struct mrsas_cmd                *cmd = NULL;
1430        struct scsa_cmd                 *acmd = PKT2CMD(pkt);
1431        MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
1432        Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
1433        uint32_t                        datalen;
1434        struct IO_REQUEST_INFO io_info;
1435        MR_FW_RAID_MAP_ALL *local_map_ptr;
1436        uint16_t pd_cmd_cdblen;

1438        con_log(CL_DLEVEL1, (CE_NOTE,
1439            "chkpnt: Entered mrsas_tbolt_build_cmd:%d", __LINE__));
```



```
1441        /* find out if this is logical or physical drive command.  */
1442        acmd->islogical = MRDRV_IS_LOGICAL(ap);
1443        acmd->device_id = MAP_DEVICE_ID(instance, ap);

1445        *cmd_done = 0;

1447        /* get the command packet */
1448        if (!(cmd = get_raid_msg_pkt(instance))) {
1449                return (NULL);
1450        }

1452        index = cmd->index;
1453        ReqDescUnion =  mr_sas_get_request_descriptor(instance, index);
1454        ReqDescUnion->Words = 0;
1455        ReqDescUnion->SCSIIO.SMID = cmd->SMID;
1456        ReqDescUnion->SCSIIO.RequestFlags =
1457            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1458            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);


1461        cmd->request_desc = ReqDescUnion;
1462        cmd->pkt = pkt;
1463        cmd->cmd = acmd;

1465        /* lets get the command directions */
1466        if (acmd->cmd_flags & CFLAG_DMASEND) {
1467                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1468                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1469                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1470                            DDI_DMA_SYNC_FORDEV);
1471                }
1472        } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
1473                if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1474                        (void) ddi_dma_sync(acmd->cmd_dmahandle,
1475                            acmd->cmd_dma_offset, acmd->cmd_dma_len,
1476                            DDI_DMA_SYNC_FORCPU);
1477                }
1478        } else {
1479                con_log(CL_ANN, (CE_NOTE, "NO DMA"));
1480        }


1483        /* get SCSI_IO raid message frame pointer */
1484        scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

1486        /* zero out SCSI_IO raid message frame */
1487        bzero(scsi_raid_io, sizeof (Mpi2RaidSCSIIORequest_t));

1489        /* Set the ldTargetId set by BuildRaidContext() */
1490        ddi_put16(acc_handle, &scsi_raid_io->RaidContext.ldTargetId,
1491            acmd->device_id);

1493        /*  Copy CDB to scsi_io_request message frame */
1494        ddi_rep_put8(acc_handle,
1495            (uint8_t *)pkt->pkt_cdbp, (uint8_t *)scsi_raid_io->CDB.CDB32,
1496            acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

1498        /*
1499         * Just the CDB length, rest of the Flags are zero
1500         * This will be modified later.
1501         */
1502        ddi_put16(acc_handle, &scsi_raid_io->IoFlags, acmd->cmd_cdblen);

1504        pd_cmd_cdblen = acmd->cmd_cdblen;
```

```
1506         switch (pkt->pkt_cdbp[0]) {
1507         case SCMD_READ:
1508         case SCMD_WRITE:
1509         case SCMD_READ_G1:
1510         case SCMD_WRITE_G1:
1511         case SCMD_READ_G4:
1512         case SCMD_WRITE_G4:
1513         case SCMD_READ_G5:
1514         case SCMD_WRITE_G5:

1516                 if (acmd->islogical) {
1517                         /* Initialize sense Information */
1518                         if (cmd->sense1 == NULL) {
1519                                 con_log(CL_ANN, (CE_NOTE, "tbolt_build_cmd: "
1520                                     "Sense buffer ptr NULL "));
1521                         }
1522                         bzero(cmd->sense1, SENSE_LENGTH);
1523                         con_log(CL_DLEVEL2, (CE_NOTE, "tbolt_build_cmd "
1524                             "CDB[0] = %x\n", pkt->pkt_cdbp[0]));

1526                         if (acmd->cmd_cdblen == CDB_GROUP0) {
1527                                 /* 6-byte cdb */
1528                                 lba_count = (uint16_t)(pkt->pkt_cdbp[4]);
1529                                 start_lba_lo = ((uint32_t)(pkt->pkt_cdbp[3]) |
1530                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
1531                                     ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
1532                                     << 16));
1533                         } else if (acmd->cmd_cdblen == CDB_GROUP1) {
1534                                 /* 10-byte cdb */
1535                                 lba_count =
1536                                     (((uint16_t)(pkt->pkt_cdbp[8])) |
1537                                     ((uint16_t)(pkt->pkt_cdbp[7]) << 8));

1539                                 start_lba_lo =
1540                                     (((uint32_t)(pkt->pkt_cdbp[5])) |
1541                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1542                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1543                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1545                         } else if (acmd->cmd_cdblen == CDB_GROUP5) {
1546                                 /* 12-byte cdb */
1547                                 lba_count = (
1548                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
1549                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1550                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1551                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24));

1553                                 start_lba_lo =
1554                                     (((uint32_t)(pkt->pkt_cdbp[5])) |
1555                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1556                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1557                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1559                         } else if (acmd->cmd_cdblen == CDB_GROUP4) {
1560                                 /* 16-byte cdb */
1561                                 lba_count = (
1562                                     ((uint32_t)(pkt->pkt_cdbp[13])) |
1563                                     ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
1564                                     ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
1565                                     ((uint32_t)(pkt->pkt_cdbp[10]) << 24));

1567                                 start_lba_lo = (
1568                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
1569                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1570                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1571                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24));
```



```
1573                                 start_lba_hi = (
1574                                     ((uint32_t)(pkt->pkt_cdbp[5])) |
1575                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1576                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1577                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));
1578                         }

1580                         if (instance->tbolt &&
1581                             ((lba_count * 512) > mrsas_tbolt_max_cap_maxxfer)) {
1582                                 cmn_err(CE_WARN, " IO SECTOR COUNT exceeds "
1583                                     "controller limit 0x%x sectors",
1584                                     lba_count);
1585                         }

1587                         bzero(&io_info, sizeof (struct IO_REQUEST_INFO));
1588                         io_info.ldStartBlock = ((uint64_t)start_lba_hi << 32) |
1589                             start_lba_lo;
1590                         io_info.numBlocks = lba_count;
1591                         io_info.ldTgtId = acmd->device_id;

1593                         if (acmd->cmd_flags & CFLAG_DMASEND)
1594                                 io_info.isRead = 0;
1595                         else
1596                                 io_info.isRead = 1;


1599                         /* Acquire SYNC MAP UPDATE lock */
1600                         mutex_enter(&instance->sync_map_mtx);

1602                         local_map_ptr =
1603                             instance->ld_map[(instance->map_id & 1)];

1605                         if ((MR_TargetIdToLdGet(
1606                             acmd->device_id, local_map_ptr) >=
1607                             MAX_LOGICAL_DRIVES) || !instance->fast_path_io) {
1608                                 cmn_err(CE_NOTE, "Fast Path NOT Possible, "
1609                                     "targetId >= MAX_LOGICAL_DRIVES || "
1610                                     "!instance->fast_path_io");
1611                                 fp_possible = 0;
1612                                 /* Set Regionlock flags to BYPASS */
1613                                 /* io_request->RaidContext.regLockFlags  = 0; */
1614                                 ddi_put8(acc_handle,
1615                                     &scsi_raid_io->RaidContext.regLockFlags, 0);
1616                         } else {
1617                                 if (MR_BuildRaidContext(instance, &io_info,
1618                                     &scsi_raid_io->RaidContext, local_map_ptr))
1619                                         fp_possible = io_info.fpOkForIo;
1620                         }

1622                         if (!enable_fp)
1623                                 fp_possible = 0;

1625                         con_log(CL_ANN1, (CE_NOTE, "enable_fp %d  "
1626                             "instance->fast_path_io %d fp_possible %d",
1627                             enable_fp, instance->fast_path_io, fp_possible));

1629                 if (fp_possible) {

1631                         /* Check for DIF enabled LD */
1632                         if (MR_CheckDIF(acmd->device_id, local_map_ptr)) {
1633                                 /* Prepare 32 Byte CDB for DIF capable Disk */
1634                                 mrsas_tbolt_prepare_cdb(instance,
1635                                     scsi_raid_io->CDB.CDB32,
1636                                     &io_info, scsi_raid_io, start_lba_lo);
1637                         } else {
```

```
1638                                mrsas_tbolt_set_pd_lba(scsi_raid_io->CDB.CDB32,
1639                                    (uint8_t *)&pd_cmd_cdblen,
1640                                    io_info.pdBlock, io_info.numBlocks);
1641                                ddi_put16(acc_handle,
1642                                    &scsi_raid_io->IoFlags, pd_cmd_cdblen);
1643                        }

1645                        ddi_put8(acc_handle, &scsi_raid_io->Function,
1646                            MPI2_FUNCTION_SCSI_IO_REQUEST);

1648                        ReqDescUnion->SCSIIO.RequestFlags =
1649                            (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
1650                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1652                        if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1653                            (devid == PCI_DEVICE_ID_LSI_FURY)) {
1644                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1654                                uint8_t regLockFlags = ddi_get8(acc_handle,
1655                                    &scsi_raid_io->RaidContext.regLockFlags);
1656                                uint16_t IoFlags = ddi_get16(acc_handle,
1657                                    &scsi_raid_io->IoFlags);

1659                                if (regLockFlags == REGION_TYPE_UNUSED)
1660                                        ReqDescUnion->SCSIIO.RequestFlags =
1661                                            (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1662                                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1664                                IoFlags |=
1665                                    MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH;
1666                                regLockFlags |=
1667                                    (MR_RL_FLAGS_GRANT_DESTINATION_CUDA |
1668                                    MR_RL_FLAGS_SEQ_NUM_ENABLE);

1670                                ddi_put8(acc_handle,
1671                                    &scsi_raid_io->ChainOffset, 0);
1672                                ddi_put8(acc_handle,
1673                                    &scsi_raid_io->RaidContext.nsegType,
1674                                    ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1675                                    MPI2_TYPE_CUDA));
1676                                ddi_put8(acc_handle,
1677                                    &scsi_raid_io->RaidContext.regLockFlags,
1678                                    regLockFlags);
1679                                ddi_put16(acc_handle,
1680                                    &scsi_raid_io->IoFlags, IoFlags);
1681                        }

1683                        if ((instance->load_balance_info[
1684                            acmd->device_id].loadBalanceFlag) &&
1685                            (io_info.isRead)) {
1686                                io_info.devHandle =
1687                                    get_updated_dev_handle(&instance->
1688                                    load_balance_info[acmd->device_id],
1689                                    &io_info);
1690                                cmd->load_balance_flag |=
1691                                    MEGASAS_LOAD_BALANCE_FLAG;
1692                        } else {
1693                                cmd->load_balance_flag &=
1694                                    ~MEGASAS_LOAD_BALANCE_FLAG;
1695                        }

1697                        ReqDescUnion->SCSIIO.DevHandle = io_info.devHandle;
1698                        ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1699                            io_info.devHandle);

1701                } else {
1702                        ddi_put8(acc_handle, &scsi_raid_io->Function,
```



```
1703                            MPI2_FUNCTION_LD_IO_REQUEST);

1705                        ddi_put16(acc_handle,
1706                            &scsi_raid_io->DevHandle, acmd->device_id);

1708                        ReqDescUnion->SCSIIO.RequestFlags =
1709                            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1710                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1712                        ddi_put16(acc_handle,
1713                            &scsi_raid_io->RaidContext.timeoutValue,
1714                            local_map_ptr->raidMap.fpPdIoTimeoutSec);

1716                        if ((devid == PCI_DEVICE_ID_LSI_INVADER) ||
1717                            (devid == PCI_DEVICE_ID_LSI_FURY)) {
1707                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1718                                uint8_t regLockFlags = ddi_get8(acc_handle,
1719                                    &scsi_raid_io->RaidContext.regLockFlags);

1721                                if (regLockFlags == REGION_TYPE_UNUSED) {
1722                                        ReqDescUnion->SCSIIO.RequestFlags =
1723                                            (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1724                                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1725                                }

1727                                regLockFlags |=
1728                                    (MR_RL_FLAGS_GRANT_DESTINATION_CPU0 |
1729                                    MR_RL_FLAGS_SEQ_NUM_ENABLE);

1731                                ddi_put8(acc_handle,
1732                                    &scsi_raid_io->RaidContext.nsegType,
1733                                    ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1734                                    MPI2_TYPE_CUDA));
1735                                ddi_put8(acc_handle,
1736                                    &scsi_raid_io->RaidContext.regLockFlags,
1737                                    regLockFlags);
1738                        }
1739                } /* Not FP */

1741                /* Release SYNC MAP UPDATE lock */
1742                mutex_exit(&instance->sync_map_mtx);


1745                /*
1746                 * Set sense buffer physical address/length in scsi_io_request.
1747                 */
1748                ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
1749                    cmd->sense_phys_addr1);
1750                ddi_put8(acc_handle, &scsi_raid_io->SenseBufferLength,
1751                    SENSE_LENGTH);

1753                /* Construct SGL */
1754                ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1755                    offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1757                (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1758                    scsi_raid_io, &datalen);

1760                ddi_put32(acc_handle, &scsi_raid_io->DataLength, datalen);

1762                break;
1763 #ifndef PDSUPPORT       /* if PDSUPPORT, skip break and fall through */
1764        } else {
1765                break;
1766 #endif
1767        }
```

```
1768         /* fall through For all non-rd/wr cmds */
1769         default:
1770                 switch (pkt->pkt_cdbp[0]) {
1771                 case 0x35: { /* SCMD_SYNCHRONIZE_CACHE */
1772                         return_raid_msg_pkt(instance, cmd);
1773                         *cmd_done = 1;
1774                         return (NULL);
1775                 }

1777                 case SCMD_MODE_SENSE:
1778                 case SCMD_MODE_SENSE_G1: {
1779                         union scsi_cdb  *cdbp;
1780                         uint16_t        page_code;

1782                         cdbp = (void *)pkt->pkt_cdbp;
1783                         page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
1784                         switch (page_code) {
1785                         case 0x3:
1786                         case 0x4:
1787                                 (void) mrsas_mode_sense_build(pkt);
1788                                 return_raid_msg_pkt(instance, cmd);
1789                                 *cmd_done = 1;
1790                                 return (NULL);
1791                         }
1792                         break;
1793                 }

1795                 default: {
1796                         /*
1797                          * Here we need to handle PASSTHRU for
1798                          * Logical Devices. Like Inquiry etc.
1799                          */

1801                         if (!(acmd->islogical)) {

1803                                 /* Acquire SYNC MAP UPDATE lock */
1804                                 mutex_enter(&instance->sync_map_mtx);

1806                                 local_map_ptr =
1807                                     instance->ld_map[(instance->map_id & 1)];

1809                                 ddi_put8(acc_handle, &scsi_raid_io->Function,
1810                                     MPI2_FUNCTION_SCSI_IO_REQUEST);

1812                                 ReqDescUnion->SCSIIO.RequestFlags =
1813                                     (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
1814                                     MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1816                                 ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1817                                     local_map_ptr->raidMap.
1818                                     devHndlInfo[acmd->device_id].curDevHdl);


1821                                 /* Set regLockFlasgs to REGION_TYPE_BYPASS */
1822                                 ddi_put8(acc_handle,
1823                                     &scsi_raid_io->RaidContext.regLockFlags, 0);
1824                                 ddi_put64(acc_handle,
1825                                     &scsi_raid_io->RaidContext.regLockRowLBA,
1826                                     0);
1827                                 ddi_put32(acc_handle,
1828                                     &scsi_raid_io->RaidContext.regLockLength,
1829                                     0);
1830                                 ddi_put8(acc_handle,
1831                                     &scsi_raid_io->RaidContext.RAIDFlags,
1832                                     MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD <<
1833                                     MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT);
```



```
1834                                 ddi_put16(acc_handle,
1835                                     &scsi_raid_io->RaidContext.timeoutValue,
1836                                     local_map_ptr->raidMap.fpPdIoTimeoutSec);
1837                                 ddi_put16(acc_handle,
1838                                     &scsi_raid_io->RaidContext.ldTargetId,
1839                                     acmd->device_id);
1840                                 ddi_put8(acc_handle,
1841                                     &scsi_raid_io->LUN[1], acmd->lun);

1843                                 /* Release SYNC MAP UPDATE lock */
1844                                 mutex_exit(&instance->sync_map_mtx);

1846                         } else {
1847                                 ddi_put8(acc_handle, &scsi_raid_io->Function,
1848                                     MPI2_FUNCTION_LD_IO_REQUEST);
1849                                 ddi_put8(acc_handle,
1850                                     &scsi_raid_io->LUN[1], acmd->lun);
1851                                 ddi_put16(acc_handle,
1852                                     &scsi_raid_io->DevHandle, acmd->device_id);
1853                                 ReqDescUnion->SCSIIO.RequestFlags =
1854                                     (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
1855                                     MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1856                         }

1858                         /*
1859                          * Set sense buffer physical address/length in
1860                          * scsi_io_request.
1861                          */
1862                         ddi_put32(acc_handle,
1863                             &scsi_raid_io->SenseBufferLowAddress,
1864                             cmd->sense_phys_addr1);
1865                         ddi_put8(acc_handle,
1866                             &scsi_raid_io->SenseBufferLength, SENSE_LENGTH);

1868                         /* Construct SGL */
1869                         ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1870                             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1872                         (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1873                             scsi_raid_io, &datalen);

1875                         ddi_put32(acc_handle,
1876                             &scsi_raid_io->DataLength, datalen);


1879                         con_log(CL_ANN, (CE_CONT,
1880                             "tbolt_build_cmd CDB[0] =%x, TargetID =%x\n",
1881                             pkt->pkt_cdbp[0], acmd->device_id));
1882                         con_log(CL_DLEVEL1, (CE_CONT,
1883                             "data length = %x\n",
1884                             scsi_raid_io->DataLength));
1885                         con_log(CL_DLEVEL1, (CE_CONT,
1886                             "cdb length = %x\n",
1887                             acmd->cmd_cdblen));
1888                 }
1889                 break;
1890                 }

1892         }

1894         return (cmd);
1895 }
_____unchanged_portion_omitted_


2228 void
```

```
2229 mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *instance,
2230     struct mrsas_cmd *cmd)
2231 {
2232         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2233         Mpi25IeeeSgeChain64_t           *scsi_raid_io_sgl_ieee;
2234         MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
2235         uint32_t                        index;
2236         ddi_acc_handle_t acc_handle =
2237             instance->mpi2_frame_pool_dma_obj.acc_handle;

2239         if (!instance->tbolt) {
2240                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled."));
2241                 return;
2242         }

2244         index = cmd->index;

2246         ReqDescUnion = mr_sas_get_request_descriptor(instance, index);

2248         if (!ReqDescUnion) {
2249                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]"));
2250                 return;
2251         }

2253         con_log(CL_ANN1, (CE_NOTE, "[SMID]%x", cmd->SMID));

2255         ReqDescUnion->Words = 0;

2257         ReqDescUnion->SCSIIO.RequestFlags =
2258             (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
2259             MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

2261         ReqDescUnion->SCSIIO.SMID = cmd->SMID;

2263         cmd->request_desc = ReqDescUnion;

2265         /* get raid message frame pointer */
2266         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2268         if ((instance->device_id == PCI_DEVICE_ID_LSI_INVADER) ||
2269             (instance->device_id == PCI_DEVICE_ID_LSI_FURY)) {
2258         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
2270                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)
2271                     &scsi_raid_io->SGL.IeeeChain;
2272                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;
2273                 ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
2274         }

2276         ddi_put8(acc_handle, &scsi_raid_io->Function,
2277             MPI2_FUNCTION_PASSTHRU_IO_REQUEST);

2279         ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
2280             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

2282         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
2283             (U8)offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16);

2285         ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
2286             cmd->sense_phys_addr1);


2289         scsi_raid_io_sgl_ieee =
2290             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

2292         ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
2293             (U64)cmd->frame_phys_addr);
```



```
2295         ddi_put8(acc_handle,
2296             &scsi_raid_io_sgl_ieee->Flags, (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2297             MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
2298         /* LSI put hardcoded 1024 instead of MEGASAS_MAX_SZ_CHAIN_FRAME. */
2299         ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length, 1024);

2301         con_log(CL_ANN1, (CE_NOTE,
2302             "[MFI CMD PHY ADDRESS]:%" PRIx64,
2303             scsi_raid_io_sgl_ieee->Address));
2304         con_log(CL_ANN1, (CE_NOTE,
2305             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
2306         con_log(CL_ANN1, (CE_NOTE, "[SGL Flags]:%x",
2307             scsi_raid_io_sgl_ieee->Flags));
2308 }
```

_____***unchanged_portion_omitted_***