```
**********************************************************
    5848 Tue Apr  2 11:54:22 2013
new/usr/src/cmd/mdb/common/modules/mr_sas/mr_sas.c
3500 Support LSI SAS2008 (Falcon) Skinny FW for mr_sas(7D)
**********************************************************
```

```
/*
 * 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 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2013 Nexenta Systems, Inc. All rights reserved.
 */

#include <limits.h>
#include <sys/mdb_modapi.h>
#include <sys/sysinfo.h>
#include <sys/sunmdi.h>
#include <sys/scsi/scsi.h>
#include "mr_sas.h"

int
construct_path(uintptr_t addr, char *result)
{
        struct  dev_info        d;
        char    devi_node[PATH_MAX];
        char    devi_addr[PATH_MAX];

        if (mdb_vread(&d, sizeof (d), addr) == -1) {
                mdb_warn("couldn't read dev_info");
                return (DCMD_ERR);
        }

        if (d.devi_parent) {
                construct_path((uintptr_t)d.devi_parent, result);
                mdb_readstr(devi_node, sizeof (devi_node),
                    (uintptr_t)d.devi_node_name);
                mdb_readstr(devi_addr, sizeof (devi_addr),
                    (uintptr_t)d.devi_addr);
                mdb_snprintf(result+strlen(result),
                    PATH_MAX-strlen(result),
                    "/%s%s%s", devi_node, (*devi_addr ? "@" : ""),
                    devi_addr);
        }
        return (DCMD_OK);
}
```



```
void
display_targets(struct mrsas_instance *m, int verbose)
display_targets(struct mrsas_instance m, int verbose)
{
        int     tgt;
        struct mrsas_ld mr_ldp[MRDRV_MAX_LD];
        struct mrsas_tbolt_pd mr_pdp[MRSAS_TBOLT_PD_TGT_MAX];
        struct mrsas_ld *mr_ldp;
        char    device_path[PATH_MAX];

        if (verbose) {
                *device_path = 0;
                if (construct_path((uintptr_t)m->dip, device_path) != DCMD_OK) {
                if (construct_path((uintptr_t)m.dip, device_path) != DCMD_OK) {
                        strcpy(device_path, "couldn't determine device path");
                }
        }

        mdb_printf("\n");
        if (verbose)
                mdb_printf("%s\n", device_path);
        mdb_printf("Physical/Logical Target\n");
        mdb_printf("-----------------------\n");

        if (mdb_vread(&mr_ldp, sizeof (mr_ldp), (uintptr_t)m->mr_ld_list)
            == -1 ||
            mdb_vread(&mr_pdp, sizeof (mr_pdp), (uintptr_t)m->mr_tbolt_pd_list)
            == -1) {
                mdb_warn("can't read list of disks");
                return;
        }

        mdb_printf("dev_type target\n");
        mdb_printf("----------");
        mdb_printf("\n");
        for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
                if (mr_ldp[tgt].dip != NULL &&
                    mr_ldp[tgt].lun_type == MRSAS_LD_LUN) {
                        mdb_printf("Logical          sd %d\n", tgt);
                mr_ldp = (struct mrsas_ld *)&m.mr_ld_list[tgt];
                if ((mr_ldp != NULL) && (mr_ldp->dip != NULL) &&
                    (mr_ldp->lun_type == MRSAS_LD_LUN)) {
                        mdb_printf("sd %d", tgt);
                        mdb_printf("\n");
                }
        }
        for (tgt = 0; tgt < MRSAS_TBOLT_PD_TGT_MAX; tgt++) {
                if (mr_pdp[tgt].dip != NULL &&
                    mr_pdp[tgt].lun_type == MRSAS_TBOLT_PD_LUN) {
                        mdb_printf("Physical         sd %d\n", tgt);
                }
        }
        mdb_printf("\n");
}

void
display_deviceinfo(struct mrsas_instance *m)
display_deviceinfo(struct mrsas_instance m)
{
        uint16_t vid, did, svid, sid;

        vid = m->vendor_id;
        did = m->device_id;
        svid = m->subsysvid;
        sid = m->subsysid;
```

```
 95          vid = m.vendor_id;
 96          did = m.device_id;
 97          svid = m.subsysvid;
 98          sid = m.subsysid;

117         mdb_printf("\n");
118         mdb_printf("vendor_id device_id subsysvid subsysid");
119         mdb_printf("\n");
120         mdb_printf("---------------------------------------");
121         mdb_printf("\n");
122         mdb_printf("    0x%x   0x%x    0x%x    0x%x",
123             vid, did, svid, sid);
124         mdb_printf("\n");
125 }

127 static int
128 mr_sas_dcmd(uintptr_t addr, uint_t flags, int argc, const mdb_arg_t *argv)
129 {
130         struct mrsas_instance m;

132         int     instance;
133         uint16_t ncmds;
134         uint_t  verbose = FALSE;
135         uint_t  device_info = FALSE;
136         uint_t  target_info = FALSE;
137         int     rv = DCMD_OK;
138         void    *mrsas_state;

140         if (!(flags & DCMD_ADDRSPEC)) {
141                 mrsas_state = NULL;
142                 if (mdb_readvar(&mrsas_state, "mrsas_state") == -1) {
143                         mdb_warn("can't read mrsas_state");
144                         return (DCMD_ERR);
145                 }
146                 if (mdb_pwalk_dcmd("genunix`softstate", "mr_sas`mr_sas",
147                     argc, argv, (uintptr_t)mrsas_state) == -1) {
148                         mdb_warn("mdb_pwalk_dcmd failed");
149                         return (DCMD_ERR);
150                 }
151                 return (DCMD_OK);
152         }

154         if (mdb_getopts(argc, argv,
155             'd', MDB_OPT_SETBITS, TRUE, &device_info,
156             't', MDB_OPT_SETBITS, TRUE, &target_info,
157             'v', MDB_OPT_SETBITS, TRUE, &verbose,
158             NULL) != argc)
159                 return (DCMD_USAGE);

161         if (mdb_vread(&m, sizeof (m), addr) == -1) {
162                 mdb_warn("couldn't read mrsas_instance struct at 0x%p", addr);
163                 return (DCMD_ERR);
164         }
165         instance = m.instance;

167         /* cmd slot info */
168         ncmds = m.max_fw_cmds;

170         /* processing completed */
171         if (((flags & DCMD_ADDRSPEC) && !(flags & DCMD_LOOP)) ||
172             (flags & DCMD_LOOPFIRST)) {
173                 if ((flags & DCMD_LOOP) && !(flags & DCMD_LOOPFIRST))
174                         mdb_printf("\n");
175                 mdb_printf("         mrsas_t inst max_fw_cmds intr_type");
176                 mdb_printf("\n");
177                 mdb_printf("=============================================");
```



```
178                 mdb_printf("\n");
179         }

181         mdb_printf("%16p %4d      %4d    ", addr, instance, ncmds);
182         switch (m.intr_type) {
183                 case DDI_INTR_TYPE_MSIX:
184                         mdb_printf("MSI-X");
185                         break;
186                 case DDI_INTR_TYPE_MSI:
187                         mdb_printf("MSI");
188                         break;
189                 case DDI_INTR_TYPE_FIXED:
190                         mdb_printf("FIXED");
191                         break;
192                 default:
193                         mdb_printf("INVALD");
194         }
195         mdb_printf("\n");

197         if (target_info)
198                 display_targets(&m, verbose);
181                 display_targets(m, verbose);

200         if (device_info)
201                 display_deviceinfo(&m);
184                 display_deviceinfo(m);

203         return (rv);
204 }
```

_____unchanged_portion_omitted_

```
**********************************************************
    2549 Tue Apr  2 11:54:23 2013
new/usr/src/pkg/manifests/driver-storage-mr_sas.mf
3500 Support LSI SAS2008 (Falcon) Skinny FW for mr_sas(7D)
**********************************************************
#
# 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 2012 Nexenta Systems, Inc. All rights reserved.
#

#
# 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=pciex1000,71 \
    alias=pciex1000,73 \
    alias=pciex1000,78 \
    alias=pciex1000,79
$(i386_ONLY)driver name=mr_sas class=scsi-self-identifying \
    alias=pciex1000,5b \
    alias=pciex1000,5d \
    alias=pciex1000,71 \
    alias=pciex1000,73 \
    alias=pciex1000,78 \
    alias=pciex1000,79
file path=kernel/drv/$(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
```

```
**********************************************************
  221542 Tue Apr  2 11:54:23 2013
new/usr/src/uts/common/io/mr_sas/mr_sas.c
3500 Support LSI SAS2008 (Falcon) Skinny FW for mr_sas(7D)
**********************************************************
```

```
/*
 * 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 2012 Nexenta System, Inc. All rights reserved.
 */

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



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

#include "mr_sas.h"

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

/* 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 *);
```

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

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



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

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

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


424 /*
425  * ************************************************************************** *
426  *                                                                            *
427  *               common entry points - for autoconfiguration                  *
428  *                                                                            *
429  * ************************************************************************** *
430  */
431 /*
432  * attach - adds a device to the system as part of initialization
433  * @dip:
434  * @cmd:
435  *
436  * The kernel calls a driver's attach() entry point to attach an instance of
437  * a device (for MegaRAID, it is instance of a controller) or to resume
438  * operation for an instance of a device that has been suspended or has been
439  * shut down by the power management framework
440  * The attach() entry point typically includes the following types of
441  * processing:
442  * - allocate a soft-state structure for the device instance (for MegaRAID,
443  *   controller instance)
444  * - initialize per-instance mutexes
445  * - initialize condition variables
446  * - register the device's interrupts (for MegaRAID, controller's interrupts)
447  * - map the registers and memory of the device instance (for MegaRAID,
448  *   controller instance)
449  * - create minor device nodes for the device instance (for MegaRAID,
450  *   controller instance)
451  * - report that the device instance (for MegaRAID, controller instance) has
452  *   attached
453  */
454 static int
455 mrsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
456 {
457         int             instance_no;
458         int             nregs;
```

```
459         int             i = 0;
460         uint8_t         irq;
461         uint16_t        vendor_id;
462         uint16_t        device_id;
463         uint16_t        subsysvid;
464         uint16_t        subsysid;
465         uint16_t        command;
466         off_t           reglength = 0;
467         int             intr_types = 0;
468         char            *data;

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

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

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

479         instance_no = ddi_get_instance(dip);

481         /*
482          * check to see whether this device is in a DMA-capable slot.
483          */
484         if (ddi_slaveonly(dip) == DDI_SUCCESS) {
485                 cmn_err(CE_WARN,
486                     "mr_sas%d: Device in slave-only slot, unused",
487                     instance_no);
488                 return (DDI_FAILURE);
489         }

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

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

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

512                 instance->unroll.softs  = 1;

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

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



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

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

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

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

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

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

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

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

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

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

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

579                         instance->func_ptr =
580                             &mrsas_function_template_fusion;
581                         instance->tbolt = 1;
582                         break;

584                 case PCI_DEVICE_ID_LSI_SKINNY:
585                 case PCI_DEVICE_ID_LSI_SKINNY_NEW:
586                         /*
587                          * FALLTHRU to PPC-style functions, but mark this
588                          * instance as Skinny, because the register set is
589                          * slightly different (See WR_IB_PICK_QPORT), and
590                          * certain other features are available to a Skinny
```

```
591                          * HBA.
592                          */
593                         instance->skinny = 1;
594                         /* FALLTHRU */

596                 case PCI_DEVICE_ID_LSI_2108VDE:
597                 case PCI_DEVICE_ID_LSI_2108V:
598                         con_log(CL_ANN, (CE_NOTE,
599                             "mr_sas: 2108 Liberator device detected"));

601                         instance->func_ptr =
602                             &mrsas_function_template_ppc;
603                         break;

605                 default:
606                         cmn_err(CE_WARN,
607                             "mr_sas: Invalid device detected");

609                         pci_config_teardown(&instance->pci_handle);
610                         ddi_soft_state_free(mrsas_state, instance_no);
611                         return (DDI_FAILURE);
612                 }

614                 instance->baseaddress = pci_config_get32(
615                     instance->pci_handle, PCI_CONF_BASE0);
616                 instance->baseaddress &= 0x0fffc;

618                 instance->dip           = dip;
619                 instance->vendor_id     = vendor_id;
620                 instance->device_id     = device_id;
621                 instance->subsysvid     = subsysvid;
622                 instance->subsysid      = subsysid;
623                 instance->instance      = instance_no;

625                 /* Initialize FMA */
626                 instance->fm_capabilities = ddi_prop_get_int(
627                     DDI_DEV_T_ANY, instance->dip, DDI_PROP_DONTPASS,
628                     "fm-capable", DDI_FM_EREPORT_CAPABLE |
629                     DDI_FM_ACCCHK_CAPABLE | DDI_FM_DMACHK_CAPABLE
630                     | DDI_FM_ERRCB_CAPABLE);

632                 mrsas_fm_init(instance);

634                 /* Setup register map */
635                 if ((ddi_dev_regsize(instance->dip,
636                     REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
637                     reglength < MINIMUM_MFI_MEM_SZ) {
638                         goto fail_attach;
639                 }
640                 if (reglength > DEFAULT_MFI_MEM_SZ) {
641                         reglength = DEFAULT_MFI_MEM_SZ;
642                         con_log(CL_DLEVEL1, (CE_NOTE,
643                             "mr_sas: register length to map is 0x%lx bytes",
644                             reglength));
645                 }
646                 if (ddi_regs_map_setup(instance->dip,
647                     REGISTER_SET_IO_2108, &instance->regmap, 0,
648                     reglength, &endian_attr, &instance->regmap_handle)
649                     != DDI_SUCCESS) {
650                         cmn_err(CE_WARN,
651                             "mr_sas: couldn't map control registers");
652                         goto fail_attach;
653                 }

655                 instance->unroll.regs = 1;
```



```
657                 /*
658                  * Disable Interrupt Now.
659                  * Setup Software interrupt
660                  */
661                 instance->func_ptr->disable_intr(instance);

663                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
664                     "mrsas-enable-msi", &data) == DDI_SUCCESS) {
665                         if (strncmp(data, "no", 3) == 0) {
666                                 msi_enable = 0;
667                                 con_log(CL_ANN1, (CE_WARN,
668                                     "msi_enable = %d disabled", msi_enable));
669                         }
670                         ddi_prop_free(data);
671                 }

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

675                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
676                     "mrsas-enable-fp", &data) == DDI_SUCCESS) {
677                         if (strncmp(data, "no", 3) == 0) {
678                                 enable_fp = 0;
679                                 cmn_err(CE_NOTE,
680                                     "enable_fp = %d, Fast-Path disabled.\n",
681                                     enable_fp);
682                         }

684                         ddi_prop_free(data);
685                 }

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

689                 /* Check for all supported interrupt types */
690                 if (ddi_intr_get_supported_types(
691                     dip, &intr_types) != DDI_SUCCESS) {
692                         cmn_err(CE_WARN,
693                             "ddi_intr_get_supported_types() failed");
694                         goto fail_attach;
695                 }

697                 con_log(CL_DLEVEL1, (CE_NOTE,
698                     "ddi_intr_get_supported_types() ret: 0x%x", intr_types));

700                 /* Initialize and Setup Interrupt handler */
701                 if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
702                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
703                             DDI_SUCCESS) {
704                                 cmn_err(CE_WARN,
705                                     "MSIX interrupt query failed");
706                                 goto fail_attach;
707                         }
708                         instance->intr_type = DDI_INTR_TYPE_MSIX;
709                 } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
710                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
711                             DDI_SUCCESS) {
712                                 cmn_err(CE_WARN,
713                                     "MSI interrupt query failed");
714                                 goto fail_attach;
715                         }
716                         instance->intr_type = DDI_INTR_TYPE_MSI;
717                 } else if (intr_types & DDI_INTR_TYPE_FIXED) {
718                         msi_enable = 0;
719                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
720                             DDI_SUCCESS) {
721                                 cmn_err(CE_WARN,
722                                     "FIXED interrupt query failed");
```

```
723                             goto fail_attach;
724                     }
725                     instance->intr_type = DDI_INTR_TYPE_FIXED;
726             } else {
727                     cmn_err(CE_WARN, "Device cannot "
728                         "suppport either FIXED or MSI/X "
729                         "interrupts");
730                     goto fail_attach;
731             }

733             instance->unroll.intr = 1;

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

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

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

752                     goto fail_attach;
753             }

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

760             mutex_init(&instance->sync_map_mtx, NULL,
761                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

763             mutex_init(&instance->app_cmd_pool_mtx, NULL,
764                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

766             mutex_init(&instance->config_dev_mtx, NULL,
767                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

769             mutex_init(&instance->cmd_pend_mtx, NULL,
770                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

772             mutex_init(&instance->ocr_flags_mtx, NULL,
773                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

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

779             mutex_init(&instance->cmd_pool_mtx, NULL,
780                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

782             mutex_init(&instance->reg_write_mtx, NULL,
783                 MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

785             if (instance->tbolt) {
786                     mutex_init(&instance->cmd_app_pool_mtx, NULL,
787                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
```



```
789                     mutex_init(&instance->chip_mtx, NULL,
790                         MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

792             }

794             instance->unroll.mutexs = 1;

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

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

810                                     goto fail_attach;
811                             }

813                             instance->unroll.soft_isr = 1;

815                     }
816             }

818             instance->softint_running = 0;

820             /* Allocate a transport structure */
821             tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

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

829             instance->tran = tran;
830             instance->unroll.tran = 1;

832             tran->tran_hba_private  = instance;
833             tran->tran_tgt_init     = mrsas_tran_tgt_init;
834             tran->tran_tgt_probe    = scsi_hba_probe;
835             tran->tran_tgt_free     = mrsas_tran_tgt_free;
818             if (instance->tbolt) {
819                     tran->tran_init_pkt     =
820                         mrsas_tbolt_tran_init_pkt;
821                     tran->tran_start        =
822                         mrsas_tbolt_tran_start;
823             } else {
836                     tran->tran_init_pkt     = mrsas_tran_init_pkt;
837                     if (instance->tbolt)
838                             tran->tran_start = mrsas_tbolt_tran_start;
839                     else
840                             tran->tran_start = mrsas_tran_start;
826             }
841             tran->tran_abort        = mrsas_tran_abort;
842             tran->tran_reset        = mrsas_tran_reset;
843             tran->tran_getcap       = mrsas_tran_getcap;
844             tran->tran_setcap       = mrsas_tran_setcap;
845             tran->tran_destroy_pkt  = mrsas_tran_destroy_pkt;
846             tran->tran_dmafree      = mrsas_tran_dmafree;
847             tran->tran_sync_pkt     = mrsas_tran_sync_pkt;
```

```
848                 tran->tran_quiesce      = mrsas_tran_quiesce;
849                 tran->tran_unquiesce    = mrsas_tran_unquiesce;
850                 tran->tran_bus_config   = mrsas_tran_bus_config;

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


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

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

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

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

879                         goto fail_attach;
880                 }

882                 instance->unroll.devctl = 1;

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

891                         goto fail_attach;
892                 }

894                 instance->unroll.scsictl = 1;

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

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

909                         goto fail_attach;
910                 }

912                 instance->unroll.ioctl = 1;
```



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

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

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

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

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

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

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

970                         instance->unroll.pdlist_buff = 1;
971                 }
972 #endif
973                 break;
974         case DDI_PM_RESUME:
975                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_PM_RESUME"));
976                 break;
977         case DDI_RESUME:
978                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_RESUME"));
```

```
 979                 break;
 980         default:
 981                 con_log(CL_ANN,
 982                     (CE_WARN, "mr_sas: invalid attach cmd=%x", cmd));
 983                 return (DDI_FAILURE);
 984         }


 987         con_log(CL_DLEVEL1,
 988             (CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
 989             instance_no));
 990         return (DDI_SUCCESS);

 992 fail_attach:

 994         mrsas_undo_resources(dip, instance);

 996         mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
 997         ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

 999         mrsas_fm_fini(instance);

1001         pci_config_teardown(&instance->pci_handle);
1002         ddi_soft_state_free(mrsas_state, instance_no);

1004         con_log(CL_ANN, (CE_WARN, "mr_sas: return failure from mrsas_attach"));

1006         cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d",
1007             instance_no);

1009         return (DDI_FAILURE);
1010 }
_____unchanged_portion_omitted_
1608 #endif  /* __sparc */

1610 /*
1611  * ************************************************************************** *
1612  *                                                                            *
1613  *                         entry points (SCSI HBA)                            *
1614  *                                                                            *
1615  * ************************************************************************** *
1616  */
1617 /*
1618  * tran_tgt_init - initialize a target device instance
1619  * @hba_dip:
1620  * @tgt_dip:
1621  * @tran:
1622  * @sd:
1623  *
1624  * The tran_tgt_init() entry point enables the HBA to allocate and initialize
1625  * any per-target resources. tran_tgt_init() also enables the HBA to qualify
1626  * the device's address as valid and supportable for that particular HBA.
1627  * By returning DDI_FAILURE, the instance of the target driver for that device
1628  * is not probed or attached.
1629  */
1630 /*ARGSUSED*/
1631 static int
1632 mrsas_tran_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1633                 scsi_hba_tran_t *tran, struct scsi_device *sd)
1634 {
1635         struct mrsas_instance *instance;
1636         uint16_t tgt = sd->sd_address.a_target;
1637         uint8_t lun = sd->sd_address.a_lun;
1638         dev_info_t *child = NULL;

1640         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init target %d lun %d",
```



```
1641             tgt, lun));

1643         instance = ADDR2MR(&sd->sd_address);

1645         if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
1646                 /*
1647                  * If no persistent node exists, we don't allow .conf node
1648                  * to be created.
1649                  */
1650                 if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
1651                         con_log(CL_DLEVEL2,
1652                             (CE_NOTE, "mrsas_tgt_init find child ="
1653                             " %p t = %d l = %d", (void *)child, tgt, lun));
1654                         if (ndi_merge_node(tgt_dip, mrsas_name_node) !=
1655                             DDI_SUCCESS)
1656                                 /* Create this .conf node */
1657                                 return (DDI_SUCCESS);
1658                 }
1659                 con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init in ndi_per "
1660                     "DDI_FAILURE t = %d l = %d", tgt, lun));
1661                 return (DDI_FAILURE);

1663         }

1665         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init dev_dip %p tgt_dip %p",
1666             (void *)instance->mr_ld_list[tgt].dip, (void *)tgt_dip));

1668         if (tgt < MRDRV_MAX_LD && lun == 0) {
1669                 if (instance->mr_ld_list[tgt].dip == NULL &&
1670                     strcmp(ddi_driver_name(sd->sd_dev), "sd") == 0) {
1671                         mutex_enter(&instance->config_dev_mtx);
1672                         instance->mr_ld_list[tgt].dip = tgt_dip;
1673                         instance->mr_ld_list[tgt].lun_type = MRSAS_LD_LUN;
1674                         instance->mr_ld_list[tgt].flag = MRDRV_TGT_VALID;
1675                         mutex_exit(&instance->config_dev_mtx);
1676                 }
1677         }

1679 #ifdef PDSUPPORT
1680         else if (instance->tbolt || instance->skinny) {
1666         else if (instance->tbolt) {
1681                 if (instance->mr_tbolt_pd_list[tgt].dip == NULL) {
1682                         mutex_enter(&instance->config_dev_mtx);
1683                         instance->mr_tbolt_pd_list[tgt].dip = tgt_dip;
1684                         instance->mr_tbolt_pd_list[tgt].flag =
1685                             MRDRV_TGT_VALID;
1686                         mutex_exit(&instance->config_dev_mtx);
1687                         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_tgt_init:"
1688                             "t%xl%x", tgt, lun));
1689                 }
1690         }
1691 #endif

1693         return (DDI_SUCCESS);
1694 }

1696 /*ARGSUSED*/
1697 static void
1698 mrsas_tran_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1699     scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1700 {
1701         struct mrsas_instance *instance;
1702         int tgt = sd->sd_address.a_target;
1703         int lun = sd->sd_address.a_lun;

1705         instance = ADDR2MR(&sd->sd_address);
```

```
1707         con_log(CL_DLEVEL2, (CE_NOTE, "tgt_free t = %d l = %d", tgt, lun));

1709         if (tgt < MRDRV_MAX_LD && lun == 0) {
1710                 if (instance->mr_ld_list[tgt].dip == tgt_dip) {
1711                         mutex_enter(&instance->config_dev_mtx);
1712                         instance->mr_ld_list[tgt].dip = NULL;
1713                         mutex_exit(&instance->config_dev_mtx);
1714                 }
1715         }

1717 #ifdef PDSUPPORT
1718         else if (instance->tbolt || instance->skinny) {
1704         else if (instance->tbolt) {
1719                 mutex_enter(&instance->config_dev_mtx);
1720                 instance->mr_tbolt_pd_list[tgt].dip = NULL;
1721                 mutex_exit(&instance->config_dev_mtx);
1722                 con_log(CL_ANN1, (CE_NOTE, "tgt_free: Setting dip = NULL"
1723                     "for tgt:%x", tgt));
1724         }
1725 #endif

1727 }
______unchanged_portion_omitted_

1878 /*
1879  * tran_start - transport a SCSI command to the addressed target
1880  * @ap:
1881  * @pkt:
1882  *
1883  * The tran_start() entry point for a SCSI HBA driver is called to transport a
1884  * SCSI command to the addressed target. The SCSI command is described
1885  * entirely within the scsi_pkt structure, which the target driver allocated
1886  * through the HBA driver's tran_init_pkt() entry point. If the command
1887  * involves a data transfer, DMA resources must also have been allocated for
1888  * the scsi_pkt structure.
1889  *
1890  * Return Values :
1891  *      TRAN_BUSY - request queue is full, no more free scbs
1892  *      TRAN_ACCEPT - pkt has been submitted to the instance
1893  */
1894 static int
1895 mrsas_tran_start(struct scsi_address *ap, register struct scsi_pkt *pkt)
1896 {
1897         uchar_t         cmd_done = 0;

1899         struct mrsas_instance   *instance = ADDR2MR(ap);
1900         struct mrsas_cmd        *cmd;

1902         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1903         if (instance->deadadapter == 1) {
1904                 con_log(CL_ANN1, (CE_WARN,
1905                     "mrsas_tran_start: return TRAN_FATAL_ERROR "
1906                     "for IO, as the HBA doesnt take any more IOs"));
1907                 if (pkt) {
1908                         pkt->pkt_reason         = CMD_DEV_GONE;
1909                         pkt->pkt_statistics     = STAT_DISCON;
1910                 }
1911                 return (TRAN_FATAL_ERROR);
1912         }

1914         if (instance->adapterresetinprogress) {
1915                 con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_start: Reset flag set, "
1916                     "returning mfi_pkt and setting TRAN_BUSY\n"));
1917                 return (TRAN_BUSY);
1918         }
```



```
1920         con_log(CL_ANN1, (CE_CONT, "chkpnt:%s:%d:SCSI CDB[0]=0x%x time:%x",
1921             __func__, __LINE__, pkt->pkt_cdbp[0], pkt->pkt_time));

1923         pkt->pkt_reason = CMD_CMPLT;
1924         *pkt->pkt_scbp = STATUS_GOOD; /* clear arq scsi_status */

1926         cmd = build_cmd(instance, ap, pkt, &cmd_done);

1928         /*
1929          * Check if the command is already completed by the mrsas_build_cmd()
1930          * routine. In which case the busy_flag would be clear and scb will be
1931          * NULL and appropriate reason provided in pkt_reason field
1932          */
1933         if (cmd_done) {
1934                 pkt->pkt_reason = CMD_CMPLT;
1935                 pkt->pkt_scbp[0] = STATUS_GOOD;
1936                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1937                     | STATE_SENT_CMD;
1938                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1939                         (*pkt->pkt_comp)(pkt);
1940                 }

1942                 return (TRAN_ACCEPT);
1943         }

1945         if (cmd == NULL) {
1946                 return (TRAN_BUSY);
1947         }

1949         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1950                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1951                         con_log(CL_ANN, (CE_CONT, "mr_sas:Firmware busy"));
1952                         DTRACE_PROBE2(start_tran_err,
1953                             uint16_t, instance->fw_outstanding,
1954                             uint16_t, instance->max_fw_cmds);
1955                         mrsas_return_mfi_pkt(instance, cmd);
1941                         return_mfi_pkt(instance, cmd);
1956                         return (TRAN_BUSY);
1957                 }

1959                 /* Synchronize the Cmd frame for the controller */
1960                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1961                     DDI_DMA_SYNC_FORDEV);
1962                 con_log(CL_ANN, (CE_CONT, "issue_cmd_ppc: SCSI CDB[0]=0x%x"
1963                     "cmd->index:%x\n", pkt->pkt_cdbp[0], cmd->index));
1964                 instance->func_ptr->issue_cmd(cmd, instance);

1966         } else {
1967                 struct mrsas_header *hdr = &cmd->frame->hdr;

1969                 instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);

1971                 pkt->pkt_reason         = CMD_CMPLT;
1972                 pkt->pkt_statistics     = 0;
1973                 pkt->pkt_state |= STATE_XFERRED_DATA | STATE_GOT_STATUS;

1975                 switch (ddi_get8(cmd->frame_dma_obj.acc_handle,
1976                     &hdr->cmd_status)) {
1977                 case MFI_STAT_OK:
1978                         pkt->pkt_scbp[0] = STATUS_GOOD;
1979                         break;

1981                 case MFI_STAT_SCSI_DONE_WITH_ERROR:
1982                         con_log(CL_ANN, (CE_CONT,
1983                             "mrsas_tran_start: scsi done with error"));
```

```
1984                        pkt->pkt_reason = CMD_CMPLT;
1985                        pkt->pkt_statistics = 0;

1987                        ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;
1988                        break;

1990                case MFI_STAT_DEVICE_NOT_FOUND:
1991                        con_log(CL_ANN, (CE_CONT,
1992                            "mrsas_tran_start: device not found error"));
1993                        pkt->pkt_reason         = CMD_DEV_GONE;
1994                        pkt->pkt_statistics     = STAT_DISCON;
1995                        break;

1997                default:
1998                        ((struct scsi_status *)pkt->pkt_scbp)->sts_busy = 1;
1999                }

2001                (void) mrsas_common_check(instance, cmd);
2002                DTRACE_PROBE2(start_nointr_done, uint8_t, hdr->cmd,
2003                    uint8_t, hdr->cmd_status);
2004                mrsas_return_mfi_pkt(instance, cmd);
1990                return_mfi_pkt(instance, cmd);

2006                if (pkt->pkt_comp) {
2007                        (*pkt->pkt_comp)(pkt);
2008                }

2010        }

2012        return (TRAN_ACCEPT);
2013 }
______unchanged_portion_omitted_


2461 /*
2462  * ************************************************************************** *
2463  *                                                                            *
2464  *                                libraries                                   *
2465  *                                                                            *
2466  * ************************************************************************** *
2467  */
2468 /*
2469  * get_mfi_pkt : Get a command from the free pool
2470  * After successful allocation, the caller of this routine
2471  * must clear the frame buffer (memset to zero) before
2472  * using the packet further.
2473  *
2474  * ***** Note *****
2475  * After clearing the frame buffer the context id of the
2476  * frame buffer SHOULD be restored back.
2477  */
2478 struct mrsas_cmd *
2479 mrsas_get_mfi_pkt(struct mrsas_instance *instance)
2464 static struct mrsas_cmd *
2465 get_mfi_pkt(struct mrsas_instance *instance)
2480 {
2481         mlist_t                 *head = &instance->cmd_pool_list;
2482         struct mrsas_cmd        *cmd = NULL;

2484         mutex_enter(&instance->cmd_pool_mtx);

2486         if (!mlist_empty(head)) {
2487                 cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2488                 mlist_del_init(head->next);
2489         }
2490         if (cmd != NULL) {
```



```
2491                 cmd->pkt = NULL;
2492                 cmd->retry_count_for_ocr = 0;
2493                 cmd->drv_pkt_time = 0;

2495         }
2496         mutex_exit(&instance->cmd_pool_mtx);

2498         return (cmd);
2499 }
______unchanged_portion_omitted_
2523 /*
2524  * return_mfi_pkt : Return a cmd to free command pool
2525  */
2526 void
2527 mrsas_return_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2512 static void
2513 return_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2528 {
2529         mutex_enter(&instance->cmd_pool_mtx);
2530         /* use mlist_add_tail for debug assistance */
2531         mlist_add_tail(&cmd->list, &instance->cmd_pool_list);

2533         mutex_exit(&instance->cmd_pool_mtx);
2534 }
______unchanged_portion_omitted_


3137 /*
3138  * mrsas_alloc_cmd_pool
3139  */
3140 int
3141 mrsas_alloc_cmd_pool(struct mrsas_instance *instance)
3142 {
3143         int             i;
3144         int             count;
3145         uint32_t        max_cmd;
3146         uint32_t        reserve_cmd;
3147         size_t          sz;

3149         struct mrsas_cmd        *cmd;

3151         max_cmd = instance->max_fw_cmds;
3152         con_log(CL_ANN1, (CE_NOTE, "mrsas_alloc_cmd_pool: "
3153             "max_cmd %x", max_cmd));


3156         sz = sizeof (struct mrsas_cmd *) * max_cmd;

3158         /*
3159          * instance->cmd_list is an array of struct mrsas_cmd pointers.
3160          * Allocate the dynamic array first and then allocate individual
3161          * commands.
3162          */
3163         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
3164         ASSERT(instance->cmd_list);

3166         /* create a frame pool and assign one frame to each cmd */
3167         for (count = 0; count < max_cmd; count++) {
3168                 instance->cmd_list[count] =
3169                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
3170                 ASSERT(instance->cmd_list[count]);
3171         }

3173         /* add all the commands to command pool */

3175         INIT_LIST_HEAD(&instance->cmd_pool_list);
```

```
3176         INIT_LIST_HEAD(&instance->cmd_pend_list);
3177         INIT_LIST_HEAD(&instance->app_cmd_pool_list);

3179         /*
3180          * When max_cmd is lower than MRSAS_APP_RESERVED_CMDS, how do I split
3181          * into app_cmd and regular cmd?  For now, just take
3182          * max(1/8th of max, 4);
3183          */
3184         reserve_cmd = min(MRSAS_APP_RESERVED_CMDS,
3185             max(max_cmd >> 3, MRSAS_APP_MIN_RESERVED_CMDS));
3165         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

3187         for (i = 0; i < reserve_cmd; i++) {
3188                 cmd = instance->cmd_list[i];
3189                 cmd->index = i;
3190                 mlist_add_tail(&cmd->list, &instance->app_cmd_pool_list);
3191         }


3194         for (i = reserve_cmd; i < max_cmd; i++) {
3195                 cmd = instance->cmd_list[i];
3196                 cmd->index = i;
3197                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
3198         }

3200         return (DDI_SUCCESS);

3202 mrsas_undo_cmds:
3203         if (count > 0) {
3204                 /* free each cmd */
3205                 for (i = 0; i < count; i++) {
3206                         if (instance->cmd_list[i] != NULL) {
3207                                 kmem_free(instance->cmd_list[i],
3208                                     sizeof (struct mrsas_cmd));
3209                         }
3210                         instance->cmd_list[i] = NULL;
3211                 }
3212         }

3214 mrsas_undo_cmd_list:
3215         if (instance->cmd_list != NULL)
3216                 kmem_free(instance->cmd_list, sz);
3217         instance->cmd_list = NULL;

3219         return (DDI_FAILURE);
3220 }
_____unchanged_portion_omitted_



3283 /*
3284  * get_ctrl_info
3285  */
3286 static int
3287 get_ctrl_info(struct mrsas_instance *instance,
3288     struct mrsas_ctrl_info *ctrl_info)
3289 {
3290         int     ret = 0;

3292         struct mrsas_cmd                *cmd;
3293         struct mrsas_dcmd_frame *dcmd;
3294         struct mrsas_ctrl_info  *ci;

3296         if (instance->tbolt) {
3297                 cmd = get_raid_msg_mfi_pkt(instance);
3298         } else {
```



```
3299                 cmd = mrsas_get_mfi_pkt(instance);
3279                 cmd = get_mfi_pkt(instance);
3300         }

3302         if (!cmd) {
3303                 con_log(CL_ANN, (CE_WARN,
3304                     "Failed to get a cmd for ctrl info"));
3305                 DTRACE_PROBE2(info_mfi_err, uint16_t, instance->fw_outstanding,
3306                     uint16_t, instance->max_fw_cmds);
3307                 return (DDI_FAILURE);
3308         }

3310         /* Clear the frame buffer and assign back the context id */
3311         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3312         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3313             cmd->index);

3315         dcmd = &cmd->frame->dcmd;

3317         ci = (struct mrsas_ctrl_info *)instance->internal_buf;

3319         if (!ci) {
3320                 cmn_err(CE_WARN,
3321                     "Failed to alloc mem for ctrl info");
3322                 mrsas_return_mfi_pkt(instance, cmd);
3302                 return_mfi_pkt(instance, cmd);
3323                 return (DDI_FAILURE);
3324         }

3326         (void) memset(ci, 0, sizeof (struct mrsas_ctrl_info));

3328         /* for( i = 0; i < DCMD_MBOX_SZ; i++ ) dcmd->mbox.b[i] = 0; */
3329         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

3331         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3332         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status,
3333             MFI_CMD_STATUS_POLL_MODE);
3334         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3335         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3336             MFI_FRAME_DIR_READ);
3337         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3338         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3339             sizeof (struct mrsas_ctrl_info));
3340         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3341             MR_DCMD_CTRL_GET_INFO);
3342         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3343             instance->internal_buf_dmac_add);
3344         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3345             sizeof (struct mrsas_ctrl_info));

3347         cmd->frame_count = 1;

3349         if (instance->tbolt) {
3350                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3351         }

3353         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3354                 ret = 0;

3356                 ctrl_info->max_request_size = ddi_get32(
3357                     cmd->frame_dma_obj.acc_handle, &ci->max_request_size);

3359                 ctrl_info->ld_present_count = ddi_get16(
3360                     cmd->frame_dma_obj.acc_handle, &ci->ld_present_count);

3362                 ctrl_info->properties.on_off_properties = ddi_get32(
```

```
3363                    cmd->frame_dma_obj.acc_handle,
3364                    &ci->properties.on_off_properties);
3365                ddi_rep_get8(cmd->frame_dma_obj.acc_handle,
3366                    (uint8_t *)(ctrl_info->product_name),
3367                    (uint8_t *)(ci->product_name), 80 * sizeof (char),
3368                    DDI_DEV_AUTOINCR);
3369                /* should get more members of ci with ddi_get when needed */
3370        } else {
3371                cmn_err(CE_WARN, "get_ctrl_info: Ctrl info failed");
3372                ret = -1;
3373        }

3375        if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3376                ret = -1;
3377        }
3378        if (instance->tbolt) {
3379                return_raid_msg_mfi_pkt(instance, cmd);
3380        } else {
3381                mrsas_return_mfi_pkt(instance, cmd);
3361                return_mfi_pkt(instance, cmd);
3382        }

3384        return (ret);
3385 }

3387 /*
3388  * abort_aen_cmd
3389  */
3390 static int
3391 abort_aen_cmd(struct mrsas_instance *instance,
3392     struct mrsas_cmd *cmd_to_abort)
3393 {
3394        int     ret = 0;

3396        struct mrsas_cmd                *cmd;
3397        struct mrsas_abort_frame        *abort_fr;

3399        con_log(CL_ANN1, (CE_NOTE, "chkpnt: abort_aen:%d", __LINE__));

3401        if (instance->tbolt) {
3402                cmd = get_raid_msg_mfi_pkt(instance);
3403        } else {
3404                cmd = mrsas_get_mfi_pkt(instance);
3384                cmd = get_mfi_pkt(instance);
3405        }

3407        if (!cmd) {
3408                con_log(CL_ANN1, (CE_WARN,
3409                    "abort_aen_cmd():Failed to get a cmd for abort_aen_cmd"));
3410                DTRACE_PROBE2(abort_mfi_err, uint16_t, instance->fw_outstanding,
3411                    uint16_t, instance->max_fw_cmds);
3412                return (DDI_FAILURE);
3413        }

3415        /* Clear the frame buffer and assign back the context id */
3416        (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3417        ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3418            cmd->index);

3420        abort_fr = &cmd->frame->abort;

3422        /* prepare and issue the abort frame */
3423        ddi_put8(cmd->frame_dma_obj.acc_handle,
3424            &abort_fr->cmd, MFI_CMD_OP_ABORT);
3425        ddi_put8(cmd->frame_dma_obj.acc_handle, &abort_fr->cmd_status,
3426            MFI_CMD_STATUS_SYNC_MODE);
```



```
3427        ddi_put16(cmd->frame_dma_obj.acc_handle, &abort_fr->flags, 0);
3428        ddi_put32(cmd->frame_dma_obj.acc_handle, &abort_fr->abort_context,
3429            cmd_to_abort->index);
3430        ddi_put32(cmd->frame_dma_obj.acc_handle,
3431            &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3432        ddi_put32(cmd->frame_dma_obj.acc_handle,
3433            &abort_fr->abort_mfi_phys_addr_hi, 0);

3435        instance->aen_cmd->abort_aen = 1;

3437        cmd->frame_count = 1;

3439        if (instance->tbolt) {
3440                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3441        }

3443        if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3444                con_log(CL_ANN1, (CE_WARN,
3445                    "abort_aen_cmd: issue_cmd_in_poll_mode failed"));
3446                ret = -1;
3447        } else {
3448                ret = 0;
3449        }

3451        instance->aen_cmd->abort_aen = 1;
3452        instance->aen_cmd = 0;

3454        if (instance->tbolt) {
3455                return_raid_msg_mfi_pkt(instance, cmd);
3456        } else {
3457                mrsas_return_mfi_pkt(instance, cmd);
3437                return_mfi_pkt(instance, cmd);
3458        }

3460        atomic_add_16(&instance->fw_outstanding, (-1));

3462        return (ret);
3463 }
______unchanged_portion_omitted_


3573 /*
3574  * mrsas_init_adapter_ppc - Initialize MFI interface adapter.
3575  */
3576 int
3577 mrsas_init_adapter_ppc(struct mrsas_instance *instance)
3578 {
3579        struct mrsas_cmd                *cmd;

3581        /*
3582         * allocate memory for mfi adapter(cmd pool, individual commands, mfi
3583         * frames etc
3584         */
3585        if (alloc_space_for_mfi(instance) != DDI_SUCCESS) {
3586                con_log(CL_ANN, (CE_NOTE,
3587                    "Error, failed to allocate memory for MFI adapter"));
3588                return (DDI_FAILURE);
3589        }

3591        /* Build INIT command */
3592        cmd = mrsas_get_mfi_pkt(instance);
3593        if (cmd == NULL) {
3594                DTRACE_PROBE2(init_adapter_mfi_err, uint16_t,
3595                    instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
3596                return (DDI_FAILURE);
3597        }
```

```
3572         cmd = get_mfi_pkt(instance);

3599         if (mrsas_build_init_cmd(instance, &cmd) != DDI_SUCCESS) {
3600                 con_log(CL_ANN,
3601                     (CE_NOTE, "Error, failed to build INIT command"));

3603                 goto fail_undo_alloc_mfi_space;
3604         }

3606         /*
3607          * Disable interrupt before sending init frame ( see linux driver code)
3608          * send INIT MFI frame in polled mode
3609          */
3610         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3611                 con_log(CL_ANN, (CE_WARN, "failed to init firmware"));
3612                 goto fail_fw_init;
3613         }

3615         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
3616                 goto fail_fw_init;
3617         mrsas_return_mfi_pkt(instance, cmd);
3592         return_mfi_pkt(instance, cmd);

3619         if (ctio_enable &&
3620             (instance->func_ptr->read_fw_status_reg(instance) & 0x04000000)) {
3621                 con_log(CL_ANN, (CE_NOTE, "mr_sas: IEEE SGL's supported"));
3622                 instance->flag_ieee = 1;
3623         } else {
3624                 instance->flag_ieee = 0;
3625         }

3627         ASSERT(!instance->skinny || instance->flag_ieee);

3629         instance->unroll.alloc_space_mfi = 1;
3630         instance->unroll.verBuff = 1;

3632         return (DDI_SUCCESS);


3635 fail_fw_init:
3636         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

3638 fail_undo_alloc_mfi_space:
3639         mrsas_return_mfi_pkt(instance, cmd);
3612         return_mfi_pkt(instance, cmd);
3640         free_space_for_mfi(instance);

3642         return (DDI_FAILURE);

3644 }
______unchanged_portion_omitted_



3709 static int
3710 mrsas_issue_init_mfi(struct mrsas_instance *instance)
3711 {
3712         struct mrsas_cmd                *cmd;
3713         struct mrsas_init_frame         *init_frame;
3714         struct mrsas_init_queue_info    *initq_info;

3716 /*
3717  * Prepare a init frame. Note the init frame points to queue info
3718  * structure. Each frame has SGL allocated after first 64 bytes. For
3719  * this frame - since we don't need any SGL - we use SGL's space as
3720  * queue info structure
```



```
3721  */
3722         con_log(CL_ANN1, (CE_NOTE,
3723             "mrsas_issue_init_mfi: entry\n"));
3724         cmd = get_mfi_app_pkt(instance);

3726         if (!cmd) {
3727                 con_log(CL_ANN1, (CE_WARN,
3728                     "mrsas_issue_init_mfi: get_pkt failed\n"));
3729                 return (DDI_FAILURE);
3730         }

3732         /* Clear the frame buffer and assign back the context id */
3733         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3734         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3735             cmd->index);

3737         init_frame = (struct mrsas_init_frame *)cmd->frame;
3738         initq_info = (struct mrsas_init_queue_info *)
3739             ((unsigned long)init_frame + 64);

3741         (void) memset(init_frame, 0, MRMFI_FRAME_SIZE);
3742         (void) memset(initq_info, 0, sizeof (struct mrsas_init_queue_info));

3744         ddi_put32(cmd->frame_dma_obj.acc_handle, &initq_info->init_flags, 0);

3746         ddi_put32(cmd->frame_dma_obj.acc_handle,
3747             &initq_info->reply_queue_entries, instance->max_fw_cmds + 1);
3748         ddi_put32(cmd->frame_dma_obj.acc_handle,
3749             &initq_info->producer_index_phys_addr_hi, 0);
3750         ddi_put32(cmd->frame_dma_obj.acc_handle,
3751             &initq_info->producer_index_phys_addr_lo,
3752             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address);
3753         ddi_put32(cmd->frame_dma_obj.acc_handle,
3754             &initq_info->consumer_index_phys_addr_hi, 0);
3755         ddi_put32(cmd->frame_dma_obj.acc_handle,
3756             &initq_info->consumer_index_phys_addr_lo,
3757             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 4);

3759         ddi_put32(cmd->frame_dma_obj.acc_handle,
3760             &initq_info->reply_queue_start_phys_addr_hi, 0);
3761         ddi_put32(cmd->frame_dma_obj.acc_handle,
3762             &initq_info->reply_queue_start_phys_addr_lo,
3763             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 8);

3765         ddi_put8(cmd->frame_dma_obj.acc_handle,
3766             &init_frame->cmd, MFI_CMD_OP_INIT);
3767         ddi_put8(cmd->frame_dma_obj.acc_handle, &init_frame->cmd_status,
3768             MFI_CMD_STATUS_POLL_MODE);
3769         ddi_put16(cmd->frame_dma_obj.acc_handle, &init_frame->flags, 0);
3770         ddi_put32(cmd->frame_dma_obj.acc_handle,
3771             &init_frame->queue_info_new_phys_addr_lo,
3772             cmd->frame_phys_addr + 64);
3773         ddi_put32(cmd->frame_dma_obj.acc_handle,
3774             &init_frame->queue_info_new_phys_addr_hi, 0);

3776         ddi_put32(cmd->frame_dma_obj.acc_handle, &init_frame->data_xfer_len,
3777             sizeof (struct mrsas_init_queue_info));

3779         cmd->frame_count = 1;

3781         /* issue the init frame in polled mode */
3782         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3783                 con_log(CL_ANN1, (CE_WARN,
3784                     "mrsas_issue_init_mfi():failed to "
3785                     "init firmware"));
3786                 return_mfi_app_pkt(instance, cmd);
```

```
3787                 return (DDI_FAILURE);
3788         }

3790         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3791                 return_mfi_app_pkt(instance, cmd);
3764                 return_mfi_pkt(instance, cmd);
3792                 return (DDI_FAILURE);
3793         }

3795         return_mfi_app_pkt(instance, cmd);
3796         con_log(CL_ANN1, (CE_CONT, "mrsas_issue_init_mfi: Done"));

3798         return (DDI_SUCCESS);
3799 }
3800 /*
3801  * mfi_state_transition_to_ready        : Move the FW to READY state
3802  *
3803  * @reg_set                    : MFI register set
3804  */
3805 int
3806 mfi_state_transition_to_ready(struct mrsas_instance *instance)
3807 {
3808         int             i;
3809         uint8_t         max_wait;
3810         uint32_t        fw_ctrl = 0;
3811         uint32_t        fw_state;
3812         uint32_t        cur_state;
3813         uint32_t        cur_abs_reg_val;
3814         uint32_t        prev_abs_reg_val;
3815         uint32_t        status;

3817         cur_abs_reg_val =
3818             instance->func_ptr->read_fw_status_reg(instance);
3819         fw_state =
3820             cur_abs_reg_val & MFI_STATE_MASK;
3821         con_log(CL_ANN1, (CE_CONT,
3822             "mfi_state_transition_to_ready:FW state = 0x%x", fw_state));

3824         while (fw_state != MFI_STATE_READY) {
3825                 con_log(CL_ANN, (CE_CONT,
3826                     "mfi_state_transition_to_ready:FW state%x", fw_state));

3828                 switch (fw_state) {
3829                 case MFI_STATE_FAULT:
3830                         con_log(CL_ANN, (CE_NOTE,
3831                             "mr_sas: FW in FAULT state!!"));

3833                         return (ENODEV);
3834                 case MFI_STATE_WAIT_HANDSHAKE:
3835                         /* set the CLR bit in IMR0 */
3836                         con_log(CL_ANN1, (CE_NOTE,
3837                             "mr_sas: FW waiting for HANDSHAKE"));
3838                         /*
3839                          * PCI_Hot Plug: MFI F/W requires
3840                          * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3841                          * to be set
3842                          */
3843                         /* WR_IB_MSG_0(MFI_INIT_CLEAR_HANDSHAKE, instance); */
3844                         if (!instance->tbolt && !instance->skinny) {
3817                         if (!instance->tbolt) {
3845                                 WR_IB_DOORBELL(MFI_INIT_CLEAR_HANDSHAKE |
3846                                     MFI_INIT_HOTPLUG, instance);
3847                         } else {
3848                                 WR_RESERVED0_REGISTER(MFI_INIT_CLEAR_HANDSHAKE |
3849                                     MFI_INIT_HOTPLUG, instance);
3850                         }
```



```
3851                         max_wait        = (instance->tbolt == 1) ? 180 : 2;
3852                         cur_state       = MFI_STATE_WAIT_HANDSHAKE;
3853                         break;
3854                 case MFI_STATE_BOOT_MESSAGE_PENDING:
3855                         /* set the CLR bit in IMR0 */
3856                         con_log(CL_ANN1, (CE_NOTE,
3857                             "mr_sas: FW state boot message pending"));
3858                         /*
3859                          * PCI_Hot Plug: MFI F/W requires
3860                          * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3861                          * to be set
3862                          */
3863                         if (!instance->tbolt && !instance->skinny) {
3836                         if (!instance->tbolt) {
3864                                 WR_IB_DOORBELL(MFI_INIT_HOTPLUG, instance);
3865                         } else {
3866                                 WR_RESERVED0_REGISTER(MFI_INIT_HOTPLUG,
3867                                     instance);
3868                         }
3869                         max_wait        = (instance->tbolt == 1) ? 180 : 10;
3870                         cur_state       = MFI_STATE_BOOT_MESSAGE_PENDING;
3871                         break;
3872                 case MFI_STATE_OPERATIONAL:
3873                         /* bring it to READY state; assuming max wait 2 secs */
3874                         instance->func_ptr->disable_intr(instance);
3875                         con_log(CL_ANN1, (CE_NOTE,
3876                             "mr_sas: FW in OPERATIONAL state"));
3877                         /*
3878                          * PCI_Hot Plug: MFI F/W requires
3879                          * (MFI_INIT_READY | MFI_INIT_MFIMODE | MFI_INIT_ABORT)
3880                          * to be set
3881                          */
3882                         /* WR_IB_DOORBELL(MFI_INIT_READY, instance); */
3883                         if (!instance->tbolt && !instance->skinny) {
3856                         if (!instance->tbolt) {
3884                                 WR_IB_DOORBELL(MFI_RESET_FLAGS, instance);
3885                         } else {
3886                                 WR_RESERVED0_REGISTER(MFI_RESET_FLAGS,
3887                                     instance);

3889                                 for (i = 0; i < (10 * 1000); i++) {
3890                                         status =
3891                                             RD_RESERVED0_REGISTER(instance);
3892                                         if (status & 1) {
3893                                                 delay(1 *
3894                                                     drv_usectohz(MILLISEC));
3895                                         } else {
3896                                                 break;
3897                                         }
3898                                 }

3900                         }
3901                         max_wait        = (instance->tbolt == 1) ? 180 : 10;
3902                         cur_state       = MFI_STATE_OPERATIONAL;
3903                         break;
3904                 case MFI_STATE_UNDEFINED:
3905                         /* this state should not last for more than 2 seconds */
3906                         con_log(CL_ANN1, (CE_NOTE, "FW state undefined"));

3908                         max_wait        = (instance->tbolt == 1) ? 180 : 2;
3909                         cur_state       = MFI_STATE_UNDEFINED;
3910                         break;
3911                 case MFI_STATE_BB_INIT:
3912                         max_wait        = (instance->tbolt == 1) ? 180 : 2;
3913                         cur_state       = MFI_STATE_BB_INIT;
3914                         break;
```

```
3915                 case MFI_STATE_FW_INIT:
3916                         max_wait        = (instance->tbolt == 1) ? 180 : 2;
3917                         cur_state       = MFI_STATE_FW_INIT;
3918                         break;
3919                 case MFI_STATE_FW_INIT_2:
3920                         max_wait        = 180;
3921                         cur_state       = MFI_STATE_FW_INIT_2;
3922                         break;
3923                 case MFI_STATE_DEVICE_SCAN:
3924                         max_wait        = 180;
3925                         cur_state       = MFI_STATE_DEVICE_SCAN;
3926                         prev_abs_reg_val = cur_abs_reg_val;
3927                         con_log(CL_NONE, (CE_NOTE,
3928                             "Device scan in progress ...\n"));
3929                         break;
3930                 case MFI_STATE_FLUSH_CACHE:
3931                         max_wait        = 180;
3932                         cur_state       = MFI_STATE_FLUSH_CACHE;
3933                         break;
3934                 default:
3935                         con_log(CL_ANN1, (CE_NOTE,
3936                             "mr_sas: Unknown state 0x%x", fw_state));
3937                         return (ENODEV);
3938                 }

3940                 /* the cur_state should not last for more than max_wait secs */
3941                 for (i = 0; i < (max_wait * MILLISEC); i++) {
3942                         /* fw_state = RD_OB_MSG_0(instance) & MFI_STATE_MASK; */
3943                         cur_abs_reg_val =
3944                             instance->func_ptr->read_fw_status_reg(instance);
3945                         fw_state = cur_abs_reg_val & MFI_STATE_MASK;

3947                         if (fw_state == cur_state) {
3948                                 delay(1 * drv_usectohz(MILLISEC));
3949                         } else {
3950                                 break;
3951                         }
3952                 }
3953                 if (fw_state == MFI_STATE_DEVICE_SCAN) {
3954                         if (prev_abs_reg_val != cur_abs_reg_val) {
3955                                 continue;
3956                         }
3957                 }

3959                 /* return error if fw_state hasn't changed after max_wait */
3960                 if (fw_state == cur_state) {
3961                         con_log(CL_ANN1, (CE_WARN,
3962                             "FW state hasn't changed in %d secs", max_wait));
3963                         return (ENODEV);
3964                 }
3965         };

3967         /* This may also need to apply to Skinny, but for now, don't worry. */
3968         if (!instance->tbolt && !instance->skinny) {
3940         if (!instance->tbolt) {
3969                 fw_ctrl = RD_IB_DOORBELL(instance);
3970                 con_log(CL_ANN1, (CE_CONT,
3971                     "mfi_state_transition_to_ready:FW ctrl = 0x%x", fw_ctrl));

3973                 /*
3974                  * Write 0xF to the doorbell register to do the following.
3975                  * - Abort all outstanding commands (bit 0).
3976                  * - Transition from OPERATIONAL to READY state (bit 1).
3977                  * - Discard (possible) low MFA posted in 64-bit mode (bit-2).
3978                  * - Set to release FW to continue running (i.e. BIOS handshake
3979                  *   (bit 3).
```



```
3980                  */
3981                 WR_IB_DOORBELL(0xF, instance);
3982         }

3984         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
3985                 return (EIO);
3986         }

3988         return (DDI_SUCCESS);
3989 }

3991 /*
3992  * get_seq_num
3993  */
3994 static int
3995 get_seq_num(struct mrsas_instance *instance,
3996     struct mrsas_evt_log_info *eli)
3997 {
3998         int     ret = DDI_SUCCESS;

4000         dma_obj_t                       dcmd_dma_obj;
4001         struct mrsas_cmd                *cmd;
4002         struct mrsas_dcmd_frame         *dcmd;
4003         struct mrsas_evt_log_info *eli_tmp;
4004         if (instance->tbolt) {
4005                 cmd = get_raid_msg_mfi_pkt(instance);
4006         } else {
4007                 cmd = mrsas_get_mfi_pkt(instance);
3979                 cmd = get_mfi_pkt(instance);
4008         }

4010         if (!cmd) {
4011                 cmn_err(CE_WARN, "mr_sas: failed to get a cmd");
4012                 DTRACE_PROBE2(seq_num_mfi_err, uint16_t,
4013                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
4014                 return (ENOMEM);
4015         }

4017         /* Clear the frame buffer and assign back the context id */
4018         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4019         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4020             cmd->index);

4022         dcmd = &cmd->frame->dcmd;

4024         /* allocate the data transfer buffer */
4025         dcmd_dma_obj.size = sizeof (struct mrsas_evt_log_info);
4026         dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
4027         dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
4028         dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
4029         dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
4030         dcmd_dma_obj.dma_attr.dma_attr_align = 1;

4032         if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
4033             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
4034                 cmn_err(CE_WARN,
4035                     "get_seq_num: could not allocate data transfer buffer.");
4036                 return (DDI_FAILURE);
4037         }

4039         (void) memset(dcmd_dma_obj.buffer, 0,
4040             sizeof (struct mrsas_evt_log_info));

4042         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4044         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
```

```
4045         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
4046         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
4047         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4048             MFI_FRAME_DIR_READ);
4049         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4050         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
4051             sizeof (struct mrsas_evt_log_info));
4052         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
4053             MR_DCMD_CTRL_EVENT_GET_INFO);
4054         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
4055             sizeof (struct mrsas_evt_log_info));
4056         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
4057             dcmd_dma_obj.dma_cookie[0].dmac_address);

4059         cmd->sync_cmd = MRSAS_TRUE;
4060         cmd->frame_count = 1;

4062         if (instance->tbolt) {
4063                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4064         }

4066         if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
4067                 cmn_err(CE_WARN, "get_seq_num: "
4068                     "failed to issue MRSAS_DCMD_CTRL_EVENT_GET_INFO");
4069                 ret = DDI_FAILURE;
4070         } else {
4071                 eli_tmp = (struct mrsas_evt_log_info *)dcmd_dma_obj.buffer;
4072                 eli->newest_seq_num = ddi_get32(cmd->frame_dma_obj.acc_handle,
4073                     &eli_tmp->newest_seq_num);
4074                 ret = DDI_SUCCESS;
4075         }

4077         if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
4078                 ret = DDI_FAILURE;

4080         if (instance->tbolt) {
4081                 return_raid_msg_mfi_pkt(instance, cmd);
4082         } else {
4083                 mrsas_return_mfi_pkt(instance, cmd);
4055                 return_mfi_pkt(instance, cmd);
4084         }

4086         return (ret);
4087 }
```

*unchanged_portion_omitted_*

```
4125 /*
4126  * flush_cache
4127  */
4128 static void
4129 flush_cache(struct mrsas_instance *instance)
4130 {
4131         struct mrsas_cmd                *cmd = NULL;
4132         struct mrsas_dcmd_frame         *dcmd;
4133         if (instance->tbolt) {
4134                 cmd = get_raid_msg_mfi_pkt(instance);
4135         } else {
4136                 cmd = mrsas_get_mfi_pkt(instance);
4108                 cmd = get_mfi_pkt(instance);
4137         }

4139         if (!cmd) {
4140                 con_log(CL_ANN1, (CE_WARN,
4141                     "flush_cache():Failed to get a cmd for flush_cache"));
4142                 DTRACE_PROBE2(flush_cache_err, uint16_t,
4143                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
```



```
4144                 return;
4145         }

4147         /* Clear the frame buffer and assign back the context id */
4148         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4149         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4150             cmd->index);

4152         dcmd = &cmd->frame->dcmd;

4154         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4156         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
4157         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
4158         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 0);
4159         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4160             MFI_FRAME_DIR_NONE);
4161         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4162         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len, 0);
4163         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
4164             MR_DCMD_CTRL_CACHE_FLUSH);
4165         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.b[0],
4166             MR_FLUSH_CTRL_CACHE | MR_FLUSH_DISK_CACHE);

4168         cmd->frame_count = 1;

4170         if (instance->tbolt) {
4171                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4172         }

4174         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
4175                 con_log(CL_ANN1, (CE_WARN,
4176             "flush_cache: failed to issue MFI_DCMD_CTRL_CACHE_FLUSH"));
4177         }
4178         con_log(CL_ANN1, (CE_CONT, "flush_cache done"));
4179         if (instance->tbolt) {
4180                 return_raid_msg_mfi_pkt(instance, cmd);
4181         } else {
4182                 mrsas_return_mfi_pkt(instance, cmd);
4154                 return_mfi_pkt(instance, cmd);
4183         }

4185 }

4187 /*
4188  * service_mfi_aen-     Completes an AEN command
4189  * @instance:                  Adapter soft state
4190  * @cmd:                       Command to be completed
4191  *
4192  */
4193 void
4194 service_mfi_aen(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
4195 {
4196         uint32_t        seq_num;
4197         struct mrsas_evt_detail *evt_detail =
4198             (struct mrsas_evt_detail *)instance->mfi_evt_detail_obj.buffer;
4199         int             rval = 0;
4200         int             tgt = 0;
4201         uint8_t         dtype;
4202 #ifdef PDSUPPORT
4203         mrsas_pd_address_t      *pd_addr;
4204 #endif
4205         ddi_acc_handle_t                acc_handle;

4207         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
```

```
4209         acc_handle = cmd->frame_dma_obj.acc_handle;
4210         cmd->cmd_status = ddi_get8(acc_handle, &cmd->frame->io.cmd_status);
4211         if (cmd->cmd_status == ENODATA) {
4212                 cmd->cmd_status = 0;
4213         }
4214 
4215         /*
4216          * log the MFI AEN event to the sysevent queue so that
4217          * application will get noticed
4218          */
4219         if (ddi_log_sysevent(instance->dip, DDI_VENDOR_LSI, "LSIMEGA", "SAS",
4220             NULL, NULL, DDI_NOSLEEP) != DDI_SUCCESS) {
4221                 int     instance_no = ddi_get_instance(instance->dip);
4222                 con_log(CL_ANN, (CE_WARN,
4223                     "mr_sas%d: Failed to log AEN event", instance_no));
4224         }
4225         /*
4226          * Check for any ld devices that has changed state. i.e. online
4227          * or offline.
4228          */
4229         con_log(CL_ANN1, (CE_CONT,
4230             "AEN: code = %x class = %x locale = %x args = %x",
4231             ddi_get32(acc_handle, &evt_detail->code),
4232             evt_detail->cl.members.class,
4233             ddi_get16(acc_handle, &evt_detail->cl.members.locale),
4234             ddi_get8(acc_handle, &evt_detail->arg_type)));
4235 
4236         switch (ddi_get32(acc_handle, &evt_detail->code)) {
4237         case MR_EVT_CFG_CLEARED: {
4238                 for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
4239                         if (instance->mr_ld_list[tgt].dip != NULL) {
4240                                 mutex_enter(&instance->config_dev_mtx);
4241                                 instance->mr_ld_list[tgt].flag =
4242                                     (uint8_t)~MRDRV_TGT_VALID;
4243                                 mutex_exit(&instance->config_dev_mtx);
4244                                 rval = mrsas_service_evt(instance, tgt, 0,
4245                                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4246                                 con_log(CL_ANN1, (CE_WARN,
4247                                     "mr_sas: CFG CLEARED AEN rval = %d "
4248                                     "tgt id = %d", rval, tgt));
4249                         }
4250                 }
4251                 break;
4252         }
4253 
4254         case MR_EVT_LD_DELETED: {
4255                 tgt = ddi_get16(acc_handle, &evt_detail->args.ld.target_id);
4256                 mutex_enter(&instance->config_dev_mtx);
4257                 instance->mr_ld_list[tgt].flag = (uint8_t)~MRDRV_TGT_VALID;
4258                 mutex_exit(&instance->config_dev_mtx);
4259                 rval = mrsas_service_evt(instance,
4260                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4261                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4262                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD DELETED AEN rval = %d "
4263                     "tgt id = %d index = %d", rval,
4264                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4265                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4266                 break;
4267         } /* End of MR_EVT_LD_DELETED */
4268 
4269         case MR_EVT_LD_CREATED: {
4270                 rval = mrsas_service_evt(instance,
4271                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4272                     MRSAS_EVT_CONFIG_TGT, NULL);
4273                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD CREATED AEN rval = %d "
4274                     "tgt id = %d index = %d", rval,
```



```
4275                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4276                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4277                 break;
4278         } /* End of MR_EVT_LD_CREATED */
4279 
4280 #ifdef PDSUPPORT
4281         case MR_EVT_PD_REMOVED_EXT: {
4282                 if (instance->tbolt || instance->skinny) {
4254                 if (instance->tbolt) {
4283                         pd_addr = &evt_detail->args.pd_addr;
4284                         dtype = pd_addr->scsi_dev_type;
4285                         con_log(CL_DLEVEL1, (CE_NOTE,
4286                             " MR_EVT_PD_REMOVED_EXT: dtype = %x,"
4287                             " arg_type = %d ", dtype, evt_detail->arg_type));
4288                         tgt = ddi_get16(acc_handle,
4289                             &evt_detail->args.pd.device_id);
4290                         mutex_enter(&instance->config_dev_mtx);
4291                         instance->mr_tbolt_pd_list[tgt].flag =
4292                             (uint8_t)~MRDRV_TGT_VALID;
4293                         mutex_exit(&instance->config_dev_mtx);
4294                         rval = mrsas_service_evt(instance, ddi_get16(
4295                             acc_handle, &evt_detail->args.pd.device_id),
4296                             1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4297                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4298                             "rval = %d tgt id = %d ", rval,
4299                             ddi_get16(acc_handle,
4300                             &evt_detail->args.pd.device_id)));
4301                 }
4302                 break;
4303         } /* End of MR_EVT_PD_REMOVED_EXT */
4304 
4305         case MR_EVT_PD_INSERTED_EXT: {
4306                 if (instance->tbolt || instance->skinny) {
4278                 if (instance->tbolt) {
4307                         rval = mrsas_service_evt(instance,
4308                             ddi_get16(acc_handle,
4309                             &evt_detail->args.pd.device_id),
4310                             1, MRSAS_EVT_CONFIG_TGT, NULL);
4311                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_INSERTEDi_EXT:"
4312                             "rval = %d tgt id = %d ", rval,
4313                             ddi_get16(acc_handle,
4314                             &evt_detail->args.pd.device_id)));
4315                 }
4316                 break;
4317         } /* End of MR_EVT_PD_INSERTED_EXT */
4318 
4319         case MR_EVT_PD_STATE_CHANGE: {
4320                 if (instance->tbolt || instance->skinny) {
4292                 if (instance->tbolt) {
4321                         tgt = ddi_get16(acc_handle,
4322                             &evt_detail->args.pd.device_id);
4323                         if ((evt_detail->args.pd_state.prevState ==
4324                             PD_SYSTEM) &&
4325                             (evt_detail->args.pd_state.newState != PD_SYSTEM)) {
4326                                 mutex_enter(&instance->config_dev_mtx);
4327                                 instance->mr_tbolt_pd_list[tgt].flag =
4328                                     (uint8_t)~MRDRV_TGT_VALID;
4329                                 mutex_exit(&instance->config_dev_mtx);
4330                                 rval = mrsas_service_evt(instance,
4331                                     ddi_get16(acc_handle,
4332                                     &evt_detail->args.pd.device_id),
4333                                     1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4334                                 con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4335                                     "rval = %d tgt id = %d ", rval,
4336                                     ddi_get16(acc_handle,
4337                                     &evt_detail->args.pd.device_id)));
```

```
4338                                    break;
4339                            }
4340                            if ((evt_detail->args.pd_state.prevState
4341                                == UNCONFIGURED_GOOD) &&
4342                                (evt_detail->args.pd_state.newState == PD_SYSTEM)) {
4343                                    rval = mrsas_service_evt(instance,
4344                                        ddi_get16(acc_handle,
4345                                        &evt_detail->args.pd.device_id),
4346                                        1, MRSAS_EVT_CONFIG_TGT, NULL);
4347                                    con_log(CL_ANN1, (CE_WARN,
4348                                        "mr_sas: PD_INSERTED: rval = %d "
4349                                        " tgt id = %d ", rval,
4350                                        ddi_get16(acc_handle,
4351                                        &evt_detail->args.pd.device_id)));
4352                                    break;
4353                            }
4354                    }
4355                    break;
4356            }
4357 #endif

4359            } /* End of Main Switch */

4361            /* get copy of seq_num and class/locale for re-registration */
4362            seq_num = ddi_get32(acc_handle, &evt_detail->seq_num);
4363            seq_num++;
4364            (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
4365                sizeof (struct mrsas_evt_detail));

4367            ddi_put8(acc_handle, &cmd->frame->dcmd.cmd_status, 0x0);
4368            ddi_put32(acc_handle, &cmd->frame->dcmd.mbox.w[0], seq_num);

4370            instance->aen_seq_num = seq_num;

4372            cmd->frame_count = 1;

4374            cmd->retry_count_for_ocr = 0;
4375            cmd->drv_pkt_time = 0;

4377            /* Issue the aen registration frame */
4378            instance->func_ptr->issue_cmd(cmd, instance);
4379 }
```

*_____unchanged_portion_omitted_*

```
4445 /*
4446  * mrsas_softintr - The Software ISR
4447  * @param arg   : HBA soft state
4448  *
4449  * called from high-level interrupt if hi-level interrupt are not there,
4450  * otherwise triggered as a soft interrupt
4451  */
4452 static uint_t
4453 mrsas_softintr(struct mrsas_instance *instance)
4454 {
4455            struct scsi_pkt         *pkt;
4456            struct scsa_cmd         *acmd;
4457            struct mrsas_cmd        *cmd;
4458            struct mlist_head       *pos, *next;
4459            mlist_t                 process_list;
4460            struct mrsas_header     *hdr;
4461            struct scsi_arq_status  *arqstat;

4463            con_log(CL_ANN1, (CE_NOTE, "mrsas_softintr() called."));

4465            ASSERT(instance);
```



```
4467            mutex_enter(&instance->completed_pool_mtx);

4469            if (mlist_empty(&instance->completed_pool_list)) {
4470                    mutex_exit(&instance->completed_pool_mtx);
4471                    return (DDI_INTR_CLAIMED);
4472            }

4474            instance->softint_running = 1;

4476            INIT_LIST_HEAD(&process_list);
4477            mlist_splice(&instance->completed_pool_list, &process_list);
4478            INIT_LIST_HEAD(&instance->completed_pool_list);

4480            mutex_exit(&instance->completed_pool_mtx);

4482            /* perform all callbacks first, before releasing the SCBs */
4483            mlist_for_each_safe(pos, next, &process_list) {
4484                    cmd = mlist_entry(pos, struct mrsas_cmd, list);

4486                    /* syncronize the Cmd frame for the controller */
4487                    (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle,
4488                        0, 0, DDI_DMA_SYNC_FORCPU);

4490                    if (mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
4491                        DDI_SUCCESS) {
4492                            mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
4493                            ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4494                            con_log(CL_ANN1, (CE_WARN,
4495                                "mrsas_softintr: "
4496                                "FMA check reports DMA handle failure"));
4497                            return (DDI_INTR_CLAIMED);
4498                    }

4500                    hdr = &cmd->frame->hdr;

4502                    /* remove the internal command from the process list */
4503                    mlist_del_init(&cmd->list);

4505                    switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
4506                    case MFI_CMD_OP_PD_SCSI:
4507                    case MFI_CMD_OP_LD_SCSI:
4508                    case MFI_CMD_OP_LD_READ:
4509                    case MFI_CMD_OP_LD_WRITE:
4510                            /*
4511                             * MFI_CMD_OP_PD_SCSI and MFI_CMD_OP_LD_SCSI
4512                             * could have been issued either through an
4513                             * IO path or an IOCTL path. If it was via IOCTL,
4514                             * we will send it to internal completion.
4515                             */
4516                            if (cmd->sync_cmd == MRSAS_TRUE) {
4517                                    complete_cmd_in_sync_mode(instance, cmd);
4518                                    break;
4519                            }

4521                            /* regular commands */
4522                            acmd =  cmd->cmd;
4523                            pkt =   CMD2PKT(acmd);

4525                            if (acmd->cmd_flags & CFLAG_DMAVALID) {
4526                                    if (acmd->cmd_flags & CFLAG_CONSISTENT) {
4527                                            (void) ddi_dma_sync(acmd->cmd_dmahandle,
4528                                                acmd->cmd_dma_offset,
4529                                                acmd->cmd_dma_len,
4530                                                DDI_DMA_SYNC_FORCPU);
4531                                    }
4532                            }
```

```
4534                        pkt->pkt_reason         = CMD_CMPLT;
4535                        pkt->pkt_statistics     = 0;
4536                        pkt->pkt_state = STATE_GOT_BUS
4537                            | STATE_GOT_TARGET | STATE_SENT_CMD
4538                            | STATE_XFERRED_DATA | STATE_GOT_STATUS;

4540                        con_log(CL_ANN, (CE_CONT,
4541                            "CDB[0] = %x completed for %s: size %lx context %x",
4542                            pkt->pkt_cdbp[0], ((acmd->islogical) ? "LD" : "PD"),
4543                            acmd->cmd_dmacount, hdr->context));
4544                        DTRACE_PROBE3(softintr_cdb, uint8_t, pkt->pkt_cdbp[0],
4545                            uint_t, acmd->cmd_cdblen, ulong_t,
4546                            acmd->cmd_dmacount);

4548                        if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
4549                                struct scsi_inquiry     *inq;

4551                                if (acmd->cmd_dmacount != 0) {
4552                                        bp_mapin(acmd->cmd_buf);
4553                                        inq = (struct scsi_inquiry *)
4554                                            acmd->cmd_buf->b_un.b_addr;

4556 #ifdef PDSUPPORT
4557                                        if (hdr->cmd_status == MFI_STAT_OK) {
4558                                                display_scsi_inquiry(
4559                                                    (caddr_t)inq);
4560                                        }
4561 #else
4562                                        /* don't expose physical drives to OS */
4563                                        if (acmd->islogical &&
4564                                            (hdr->cmd_status == MFI_STAT_OK)) {
4565                                                display_scsi_inquiry(
4566                                                    (caddr_t)inq);
4567                                        } else if ((hdr->cmd_status ==
4568                                            MFI_STAT_OK) && inq->inq_dtype ==
4569                                            DTYPE_DIRECT) {

4571                                                display_scsi_inquiry(
4572                                                    (caddr_t)inq);

4574                                                /* for physical disk */
4575                                                hdr->cmd_status =
4576                                                    MFI_STAT_DEVICE_NOT_FOUND;
4577                                        }
4578 #endif /* PDSUPPORT */
4579                                }
4580                        }

4582                        DTRACE_PROBE2(softintr_done, uint8_t, hdr->cmd,
4583                            uint8_t, hdr->cmd_status);

4585                        switch (hdr->cmd_status) {
4586                        case MFI_STAT_OK:
4587                                pkt->pkt_scbp[0] = STATUS_GOOD;
4588                                break;
4589                        case MFI_STAT_LD_CC_IN_PROGRESS:
4590                        case MFI_STAT_LD_RECON_IN_PROGRESS:
4591                                pkt->pkt_scbp[0] = STATUS_GOOD;
4592                                break;
4593                        case MFI_STAT_LD_INIT_IN_PROGRESS:
4594                                con_log(CL_ANN,
4595                                    (CE_WARN, "Initialization in Progress"));
4596                                pkt->pkt_reason = CMD_TRAN_ERR;

4598                                break;
```



```
4599                        case MFI_STAT_SCSI_DONE_WITH_ERROR:
4600                                con_log(CL_ANN, (CE_CONT, "scsi_done error"));

4602                                pkt->pkt_reason = CMD_CMPLT;
4603                                ((struct scsi_status *)
4604                                    pkt->pkt_scbp)->sts_chk = 1;

4606                                if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
4607                                        con_log(CL_ANN,
4608                                            (CE_WARN, "TEST_UNIT_READY fail"));
4609                                } else {
4610                                        pkt->pkt_state |= STATE_ARQ_DONE;
4611                                        arqstat = (void *)(pkt->pkt_scbp);
4612                                        arqstat->sts_rqpkt_reason = CMD_CMPLT;
4613                                        arqstat->sts_rqpkt_resid = 0;
4614                                        arqstat->sts_rqpkt_state |=
4615                                            STATE_GOT_BUS | STATE_GOT_TARGET
4616                                            | STATE_SENT_CMD
4617                                            | STATE_XFERRED_DATA;
4618                                        *(uint8_t *)&arqstat->sts_rqpkt_status =
4619                                            STATUS_GOOD;
4620                                        ddi_rep_get8(
4621                                            cmd->frame_dma_obj.acc_handle,
4622                                            (uint8_t *)
4623                                            &(arqstat->sts_sensedata),
4624                                            cmd->sense,
4625                                            sizeof (struct scsi_extended_sense),
4626                                            DDI_DEV_AUTOINCR);
4627                                }
4628                                break;
4629                        case MFI_STAT_LD_OFFLINE:
4630                        case MFI_STAT_DEVICE_NOT_FOUND:
4631                                con_log(CL_ANN, (CE_CONT,
4632                                "mrsas_softintr:device not found error"));
4633                                pkt->pkt_reason = CMD_DEV_GONE;
4634                                pkt->pkt_statistics  = STAT_DISCON;
4635                                break;
4636                        case MFI_STAT_LD_LBA_OUT_OF_RANGE:
4637                                pkt->pkt_state |= STATE_ARQ_DONE;
4638                                pkt->pkt_reason = CMD_CMPLT;
4639                                ((struct scsi_status *)
4640                                    pkt->pkt_scbp)->sts_chk = 1;

4642                                arqstat = (void *)(pkt->pkt_scbp);
4643                                arqstat->sts_rqpkt_reason = CMD_CMPLT;
4644                                arqstat->sts_rqpkt_resid = 0;
4645                                arqstat->sts_rqpkt_state |= STATE_GOT_BUS
4646                                    | STATE_GOT_TARGET | STATE_SENT_CMD
4647                                    | STATE_XFERRED_DATA;
4648                                *(uint8_t *)&arqstat->sts_rqpkt_status =
4649                                    STATUS_GOOD;

4651                                arqstat->sts_sensedata.es_valid = 1;
4652                                arqstat->sts_sensedata.es_key =
4653                                    KEY_ILLEGAL_REQUEST;
4654                                arqstat->sts_sensedata.es_class =
4655                                    CLASS_EXTENDED_SENSE;

4657                                /*
4658                                 * LOGICAL BLOCK ADDRESS OUT OF RANGE:
4659                                 * ASC: 0x21h; ASCQ: 0x00h;
4660                                 */
4661                                arqstat->sts_sensedata.es_add_code = 0x21;
4662                                arqstat->sts_sensedata.es_qual_code = 0x00;

4664                                break;
```

```
4666                        default:
4667                                con_log(CL_ANN, (CE_CONT, "Unknown status!"));
4668                                pkt->pkt_reason = CMD_TRAN_ERR;

4670                                break;
4671                        }

4673                        atomic_add_16(&instance->fw_outstanding, (-1));

4675                        (void) mrsas_common_check(instance, cmd);

4677                        if (acmd->cmd_dmahandle) {
4678                                if (mrsas_check_dma_handle(
4679                                    acmd->cmd_dmahandle) != DDI_SUCCESS) {
4680                                        ddi_fm_service_impact(instance->dip,
4681                                            DDI_SERVICE_UNAFFECTED);
4682                                        pkt->pkt_reason = CMD_TRAN_ERR;
4683                                        pkt->pkt_statistics = 0;
4684                                }
4685                        }

4687                        mrsas_return_mfi_pkt(instance, cmd);

4689                        /* Call the callback routine */
4690                        if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4691                            pkt->pkt_comp) {

4656                                con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_softintr: "
4657                                    "posting to scsa cmd %p index %x pkt %p "
4658                                    "time %llx", (void *)cmd, cmd->index,
4659                                    (void *)pkt, gethrtime()));
4692                                (*pkt->pkt_comp)(pkt);

4693                        }

4664                        return_mfi_pkt(instance, cmd);
4695                        break;

4697                case MFI_CMD_OP_SMP:
4698                case MFI_CMD_OP_STP:
4699                        complete_cmd_in_sync_mode(instance, cmd);
4700                        break;

4702                case MFI_CMD_OP_DCMD:
4703                        /* see if got an event notification */
4704                        if (ddi_get32(cmd->frame_dma_obj.acc_handle,
4705                            &cmd->frame->dcmd.opcode) ==
4706                            MR_DCMD_CTRL_EVENT_WAIT) {
4707                                if ((instance->aen_cmd == cmd) &&
4708                                    (instance->aen_cmd->abort_aen)) {
4709                                        con_log(CL_ANN, (CE_WARN,
4710                                            "mrsas_softintr: "
4711                                            "aborted_aen returned"));
4712                                } else {
4713                                        atomic_add_16(&instance->fw_outstanding,
4714                                            (-1));
4715                                        service_mfi_aen(instance, cmd);
4716                                }
4717                        } else {
4718                                complete_cmd_in_sync_mode(instance, cmd);
4719                        }

4721                        break;

4723                case MFI_CMD_OP_ABORT:
```



```
4724                        con_log(CL_ANN, (CE_NOTE, "MFI_CMD_OP_ABORT complete"));
4725                        /*
4726                         * MFI_CMD_OP_ABORT successfully completed
4727                         * in the synchronous mode
4728                         */
4729                        complete_cmd_in_sync_mode(instance, cmd);
4730                        break;

4732                default:
4733                        mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
4734                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

4736                        if (cmd->pkt != NULL) {
4737                                pkt = cmd->pkt;
4738                                if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4739                                    pkt->pkt_comp) {

4741                                        con_log(CL_ANN1, (CE_CONT, "posting to "
4742                                            "scsa cmd %p index %x pkt %p"
4743                                            "time %llx, default ", (void *)cmd,
4744                                            cmd->index, (void *)pkt,
4745                                            gethrtime()));

4747                                        (*pkt->pkt_comp)(pkt);

4749                                }
4750                        }
4751                        con_log(CL_ANN, (CE_WARN, "Cmd type unknown !"));
4752                        break;
4753                }
4754        }

4756        instance->softint_running = 0;

4758        return (DDI_INTR_CLAIMED);
4759 }
```

_____***unchanged_portion_omitted_***

```
5095 /*
5096  * build_cmd
5097  */
5098 static struct mrsas_cmd *
5099 build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
5100     struct scsi_pkt *pkt, uchar_t *cmd_done)
5101 {
5102         uint16_t        flags = 0;
5103         uint32_t        i;
5104         uint32_t        context;
5105         uint32_t        sge_bytes;
5106         uint32_t        tmp_data_xfer_len;
5107         ddi_acc_handle_t acc_handle;
5108         struct mrsas_cmd                *cmd;
5109         struct mrsas_sge64              *mfi_sgl;
5110         struct mrsas_sge_ieee           *mfi_sgl_ieee;
5111         struct scsa_cmd                 *acmd = PKT2CMD(pkt);
5112         struct mrsas_pthru_frame        *pthru;
5113         struct mrsas_io_frame           *ldio;

5115         /* find out if this is logical or physical drive command.  */
5116         acmd->islogical = MRDRV_IS_LOGICAL(ap);
5117         acmd->device_id = MAP_DEVICE_ID(instance, ap);
5118         *cmd_done = 0;

5120         /* get the command packet */
5121         if (!(cmd = mrsas_get_mfi_pkt(instance))) {
5091         if (!(cmd = get_mfi_pkt(instance))) {
```

```
5122                 DTRACE_PROBE2(build_cmd_mfi_err, uint16_t,
5123                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
5124                 return (NULL);
5125         }

5127         acc_handle = cmd->frame_dma_obj.acc_handle;

5129         /* Clear the frame buffer and assign back the context id */
5130         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
5131         ddi_put32(acc_handle, &cmd->frame->hdr.context, cmd->index);

5133         cmd->pkt = pkt;
5134         cmd->cmd = acmd;
5135         DTRACE_PROBE3(build_cmds, uint8_t, pkt->pkt_cdbp[0],
5136             ulong_t, acmd->cmd_dmacount, ulong_t, acmd->cmd_dma_len);

5138         /* lets get the command directions */
5139         if (acmd->cmd_flags & CFLAG_DMASEND) {
5140                 flags = MFI_FRAME_DIR_WRITE;

5142                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5143                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5144                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5145                             DDI_DMA_SYNC_FORDEV);
5146                 }
5147         } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
5148                 flags = MFI_FRAME_DIR_READ;

5150                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5151                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5152                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5153                             DDI_DMA_SYNC_FORCPU);
5154                 }
5155         } else {
5156                 flags = MFI_FRAME_DIR_NONE;
5157         }

5159         if (instance->flag_ieee) {
5160                 flags |= MFI_FRAME_IEEE;
5161         }
5162         flags |= MFI_FRAME_SGL64;

5164         switch (pkt->pkt_cdbp[0]) {

5166         /*
5167          * case SCMD_SYNCHRONIZE_CACHE:
5168          *      flush_cache(instance);
5169          *      mrsas_return_mfi_pkt(instance, cmd);
5139          *      return_mfi_pkt(instance, cmd);
5170          *      *cmd_done = 1;
5171          *
5172          *      return (NULL);
5173          */

5175         case SCMD_READ:
5176         case SCMD_WRITE:
5177         case SCMD_READ_G1:
5178         case SCMD_WRITE_G1:
5179         case SCMD_READ_G4:
5180         case SCMD_WRITE_G4:
5181         case SCMD_READ_G5:
5182         case SCMD_WRITE_G5:
5183                 if (acmd->islogical) {
5184                         ldio = (struct mrsas_io_frame *)cmd->frame;

5186                         /*
```



```
5187                          * preare the Logical IO frame:
5188                          * 2nd bit is zero for all read cmds
5189                          */
5190                         ddi_put8(acc_handle, &ldio->cmd,
5191                             (pkt->pkt_cdbp[0] & 0x02) ? MFI_CMD_OP_LD_WRITE
5192                             : MFI_CMD_OP_LD_READ);
5193                         ddi_put8(acc_handle, &ldio->cmd_status, 0x0);
5194                         ddi_put8(acc_handle, &ldio->scsi_status, 0x0);
5195                         ddi_put8(acc_handle, &ldio->target_id, acmd->device_id);
5196                         ddi_put16(acc_handle, &ldio->timeout, 0);
5197                         ddi_put8(acc_handle, &ldio->reserved_0, 0);
5198                         ddi_put16(acc_handle, &ldio->pad_0, 0);
5199                         ddi_put16(acc_handle, &ldio->flags, flags);

5201                         /* Initialize sense Information */
5202                         bzero(cmd->sense, SENSE_LENGTH);
5203                         ddi_put8(acc_handle, &ldio->sense_len, SENSE_LENGTH);
5204                         ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_hi, 0);
5205                         ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_lo,
5206                             cmd->sense_phys_addr);
5207                         ddi_put32(acc_handle, &ldio->start_lba_hi, 0);
5208                         ddi_put8(acc_handle, &ldio->access_byte,
5209                             (acmd->cmd_cdblen != 6) ? pkt->pkt_cdbp[1] : 0);
5210                         ddi_put8(acc_handle, &ldio->sge_count,
5211                             acmd->cmd_cookiecnt);
5212                         if (instance->flag_ieee) {
5213                                 mfi_sgl_ieee =
5214                                     (struct mrsas_sge_ieee *)&ldio->sgl;
5215                         } else {
5216                                 mfi_sgl = (struct mrsas_sge64   *)&ldio->sgl;
5217                         }

5219                         context = ddi_get32(acc_handle, &ldio->context);

5221                         if (acmd->cmd_cdblen == CDB_GROUP0) {
5222                                 /* 6-byte cdb */
5223                                 ddi_put32(acc_handle, &ldio->lba_count, (
5224                                     (uint16_t)(pkt->pkt_cdbp[4])));

5226                                 ddi_put32(acc_handle, &ldio->start_lba_lo, (
5227                                     ((uint32_t)(pkt->pkt_cdbp[3])) |
5228                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
5229                                     ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
5230                                     << 16)));
5231                         } else if (acmd->cmd_cdblen == CDB_GROUP1) {
5232                                 /* 10-byte cdb */
5233                                 ddi_put32(acc_handle, &ldio->lba_count, (
5234                                     ((uint16_t)(pkt->pkt_cdbp[8])) |
5235                                     ((uint16_t)(pkt->pkt_cdbp[7]) << 8)));

5237                                 ddi_put32(acc_handle, &ldio->start_lba_lo, (
5238                                     ((uint32_t)(pkt->pkt_cdbp[5])) |
5239                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5240                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5241                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5242                         } else if (acmd->cmd_cdblen == CDB_GROUP5) {
5243                                 /* 12-byte cdb */
5244                                 ddi_put32(acc_handle, &ldio->lba_count, (
5245                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
5246                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5247                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5248                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));

5250                                 ddi_put32(acc_handle, &ldio->start_lba_lo, (
5251                                     ((uint32_t)(pkt->pkt_cdbp[5])) |
5252                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
```

```
5253                                    ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5254                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5255                            } else if (acmd->cmd_cdblen == CDB_GROUP4) {
5256                                    /* 16-byte cdb */
5257                                    ddi_put32(acc_handle, &ldio->lba_count, (
5258                                        ((uint32_t)(pkt->pkt_cdbp[13])) |
5259                                        ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
5260                                        ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
5261                                        ((uint32_t)(pkt->pkt_cdbp[10]) << 24)));

5263                                    ddi_put32(acc_handle, &ldio->start_lba_lo, (
5264                                        ((uint32_t)(pkt->pkt_cdbp[9])) |
5265                                        ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5266                                        ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5267                                        ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));

5269                                    ddi_put32(acc_handle, &ldio->start_lba_hi, (
5270                                        ((uint32_t)(pkt->pkt_cdbp[5])) |
5271                                        ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5272                                        ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5273                                        ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5274                            }

5276                            break;
5277                    }
5278                    /* fall through For all non-rd/wr and physical disk cmds */
5248                    /* fall through For all non-rd/wr cmds */
5279            default:

5281                    switch (pkt->pkt_cdbp[0]) {
5282                    case SCMD_MODE_SENSE:
5283                    case SCMD_MODE_SENSE_G1: {
5284                            union scsi_cdb  *cdbp;
5285                            uint16_t        page_code;

5287                            cdbp = (void *)pkt->pkt_cdbp;
5288                            page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
5289                            switch (page_code) {
5290                            case 0x3:
5291                            case 0x4:
5292                                    (void) mrsas_mode_sense_build(pkt);
5293                                    mrsas_return_mfi_pkt(instance, cmd);
5263                                    return_mfi_pkt(instance, cmd);
5294                                    *cmd_done = 1;
5295                                    return (NULL);
5296                            }
5297                            break;
5298                    }
5299                    default:
5300                            break;
5301                    }

5303                    pthru   = (struct mrsas_pthru_frame *)cmd->frame;

5305                    /* prepare the DCDB frame */
5306                    ddi_put8(acc_handle, &pthru->cmd, (acmd->islogical) ?
5307                        MFI_CMD_OP_LD_SCSI : MFI_CMD_OP_PD_SCSI);
5308                    ddi_put8(acc_handle, &pthru->cmd_status, 0x0);
5309                    ddi_put8(acc_handle, &pthru->scsi_status, 0x0);
5310                    ddi_put8(acc_handle, &pthru->target_id, acmd->device_id);
5311                    ddi_put8(acc_handle, &pthru->lun, 0);
5312                    ddi_put8(acc_handle, &pthru->cdb_len, acmd->cmd_cdblen);
5313                    ddi_put16(acc_handle, &pthru->timeout, 0);
5314                    ddi_put16(acc_handle, &pthru->flags, flags);
5315                    tmp_data_xfer_len = 0;
5316                    for (i = 0; i < acmd->cmd_cookiecnt; i++) {
```



```
5317                            tmp_data_xfer_len += acmd->cmd_dmacookies[i].dmac_size;
5318                    }
5319                    ddi_put32(acc_handle, &pthru->data_xfer_len,
5320                        tmp_data_xfer_len);
5321                    ddi_put8(acc_handle, &pthru->sge_count, acmd->cmd_cookiecnt);
5322                    if (instance->flag_ieee) {
5323                            mfi_sgl_ieee = (struct mrsas_sge_ieee *)&pthru->sgl;
5324                    } else {
5325                            mfi_sgl = (struct mrsas_sge64 *)&pthru->sgl;
5326                    }

5328                    bzero(cmd->sense, SENSE_LENGTH);
5329                    ddi_put8(acc_handle, &pthru->sense_len, SENSE_LENGTH);
5330                    ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
5331                    ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo,
5332                        cmd->sense_phys_addr);

5334                    context = ddi_get32(acc_handle, &pthru->context);
5335                    ddi_rep_put8(acc_handle, (uint8_t *)pkt->pkt_cdbp,
5336                        (uint8_t *)pthru->cdb, acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

5338                    break;
5339            }
5340 #ifdef lint
5341            context = context;
5342 #endif
5343            /* prepare the scatter-gather list for the firmware */
5344            if (instance->flag_ieee) {
5345                    for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl_ieee++) {
5346                            ddi_put64(acc_handle, &mfi_sgl_ieee->phys_addr,
5347                                acmd->cmd_dmacookies[i].dmac_laddress);
5348                            ddi_put32(acc_handle, &mfi_sgl_ieee->length,
5349                                acmd->cmd_dmacookies[i].dmac_size);
5350                    }
5351                    sge_bytes = sizeof (struct mrsas_sge_ieee)*acmd->cmd_cookiecnt;
5352            } else {
5353                    for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl++) {
5354                            ddi_put64(acc_handle, &mfi_sgl->phys_addr,
5355                                acmd->cmd_dmacookies[i].dmac_laddress);
5356                            ddi_put32(acc_handle, &mfi_sgl->length,
5357                                acmd->cmd_dmacookies[i].dmac_size);
5358                    }
5359                    sge_bytes = sizeof (struct mrsas_sge64)*acmd->cmd_cookiecnt;
5360            }

5362            cmd->frame_count = (sge_bytes / MRMFI_FRAME_SIZE) +
5363                ((sge_bytes % MRMFI_FRAME_SIZE) ? 1 : 0) + 1;

5365            if (cmd->frame_count >= 8) {
5366                    cmd->frame_count = 8;
5367            }

5369            return (cmd);
5370 }
______unchanged_portion_omitted_

6312 /*
6313  * handle_mfi_ioctl
6314  */
6315 static int
6316 handle_mfi_ioctl(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
6317     int mode)
6318 {
6319            int     rval = DDI_SUCCESS;

6321            struct mrsas_header     *hdr;
```

```
6322         struct mrsas_cmd        *cmd;

6324         if (instance->tbolt) {
6325                 cmd = get_raid_msg_mfi_pkt(instance);
6326         } else {
6327                 cmd = mrsas_get_mfi_pkt(instance);
6297                 cmd = get_mfi_pkt(instance);
6328         }
6329         if (!cmd) {
6330                 con_log(CL_ANN, (CE_WARN, "mr_sas: "
6331                     "failed to get a cmd packet"));
6332                 DTRACE_PROBE2(mfi_ioctl_err, uint16_t,
6333                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
6334                 return (DDI_FAILURE);
6335         }

6337         /* Clear the frame buffer and assign back the context id */
6338         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6339         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6340             cmd->index);

6342         hdr = (struct mrsas_header *)&ioctl->frame[0];

6344         switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
6345         case MFI_CMD_OP_DCMD:
6346                 rval = issue_mfi_dcmd(instance, ioctl, cmd, mode);
6347                 break;
6348         case MFI_CMD_OP_SMP:
6349                 rval = issue_mfi_smp(instance, ioctl, cmd, mode);
6350                 break;
6351         case MFI_CMD_OP_STP:
6352                 rval = issue_mfi_stp(instance, ioctl, cmd, mode);
6353                 break;
6354         case MFI_CMD_OP_LD_SCSI:
6355         case MFI_CMD_OP_PD_SCSI:
6356                 rval = issue_mfi_pthru(instance, ioctl, cmd, mode);
6357                 break;
6358         default:
6359                 con_log(CL_ANN, (CE_WARN, "handle_mfi_ioctl: "
6360                     "invalid mfi ioctl hdr->cmd = %d", hdr->cmd));
6361                 rval = DDI_FAILURE;
6362                 break;
6363         }

6365         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
6366                 rval = DDI_FAILURE;

6368         if (instance->tbolt) {
6369                 return_raid_msg_mfi_pkt(instance, cmd);
6370         } else {
6371                 mrsas_return_mfi_pkt(instance, cmd);
6341                 return_mfi_pkt(instance, cmd);
6372         }

6374         return (rval);
6375 }
_____unchanged_portion_omitted_

6393 static int
6394 register_mfi_aen(struct mrsas_instance *instance, uint32_t seq_num,
6395     uint32_t class_locale_word)
6396 {
6397         int     ret_val;

6399         struct mrsas_cmd        *cmd, *aen_cmd;
6400         struct mrsas_dcmd_frame *dcmd;
```



```
6401         union mrsas_evt_class_locale    curr_aen;
6402         union mrsas_evt_class_locale    prev_aen;

6404         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
6405         /*
6406          * If there an AEN pending already (aen_cmd), check if the
6407          * class_locale of that pending AEN is inclusive of the new
6408          * AEN request we currently have. If it is, then we don't have
6409          * to do anything. In other words, whichever events the current
6410          * AEN request is subscribing to, have already been subscribed
6411          * to.
6412          *
6413          * If the old_cmd is _not_ inclusive, then we have to abort
6414          * that command, form a class_locale that is superset of both
6415          * old and current and re-issue to the FW
6416          */

6418         curr_aen.word = LE_32(class_locale_word);
6419         curr_aen.members.locale = LE_16(curr_aen.members.locale);
6420         aen_cmd = instance->aen_cmd;
6421         if (aen_cmd) {
6422                 prev_aen.word = ddi_get32(aen_cmd->frame_dma_obj.acc_handle,
6423                     &aen_cmd->frame->dcmd.mbox.w[1]);
6424                 prev_aen.word = LE_32(prev_aen.word);
6425                 prev_aen.members.locale = LE_16(prev_aen.members.locale);
6426                 /*
6427                  * A class whose enum value is smaller is inclusive of all
6428                  * higher values. If a PROGRESS (= -1) was previously
6429                  * registered, then a new registration requests for higher
6430                  * classes need not be sent to FW. They are automatically
6431                  * included.
6432                  *
6433                  * Locale numbers don't have such hierarchy. They are bitmap
6434                  * values
6435                  */
6436                 if ((prev_aen.members.class <= curr_aen.members.class) &&
6437                     !((prev_aen.members.locale & curr_aen.members.locale) ^
6438                     curr_aen.members.locale)) {
6439                         /*
6440                          * Previously issued event registration includes
6441                          * current request. Nothing to do.
6442                          */

6444                         return (0);
6445                 } else {
6446                         curr_aen.members.locale |= prev_aen.members.locale;

6448                         if (prev_aen.members.class < curr_aen.members.class)
6449                                 curr_aen.members.class = prev_aen.members.class;

6451                         ret_val = abort_aen_cmd(instance, aen_cmd);

6453                         if (ret_val) {
6454                                 con_log(CL_ANN, (CE_WARN, "register_mfi_aen: "
6455                                     "failed to abort prevous AEN command"));

6457                                 return (ret_val);
6458                         }
6459                 }
6460         } else {
6461                 curr_aen.word = LE_32(class_locale_word);
6462                 curr_aen.members.locale = LE_16(curr_aen.members.locale);
6463         }

6465         if (instance->tbolt) {
6466                 cmd = get_raid_msg_mfi_pkt(instance);
```

```
6467         } else {
6468                 cmd = mrsas_get_mfi_pkt(instance);
6438                 cmd = get_mfi_pkt(instance);
6469         }

6471         if (!cmd) {
6472                 DTRACE_PROBE2(mfi_aen_err, uint16_t, instance->fw_outstanding,
6473                     uint16_t, instance->max_fw_cmds);
6474                 return (ENOMEM);
6475         }

6477         /* Clear the frame buffer and assign back the context id */
6478         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6479         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6480             cmd->index);

6482         dcmd = &cmd->frame->dcmd;

6484         /* for(i = 0; i < DCMD_MBOX_SZ; i++) dcmd->mbox.b[i] = 0; */
6485         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

6487         (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
6488             sizeof (struct mrsas_evt_detail));

6490         /* Prepare DCMD for aen registration */
6491         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
6492         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
6493         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
6494         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
6495             MFI_FRAME_DIR_READ);
6496         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
6497         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
6498             sizeof (struct mrsas_evt_detail));
6499         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
6500             MR_DCMD_CTRL_EVENT_WAIT);
6501         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], seq_num);
6502         curr_aen.members.locale = LE_16(curr_aen.members.locale);
6503         curr_aen.word = LE_32(curr_aen.word);
6504         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[1],
6505             curr_aen.word);
6506         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
6507             instance->mfi_evt_detail_obj.dma_cookie[0].dmac_address);
6508         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
6509             sizeof (struct mrsas_evt_detail));

6511         instance->aen_seq_num = seq_num;


6514         /*
6515          * Store reference to the cmd used to register for AEN. When an
6516          * application wants us to register for AEN, we have to abort this
6517          * cmd and re-register with a new EVENT LOCALE supplied by that app
6518          */
6519         instance->aen_cmd = cmd;

6521         cmd->frame_count = 1;

6523         /* Issue the aen registration frame */
6524         /* atomic_add_16 (&instance->fw_outstanding, 1); */
6525         if (instance->tbolt) {
6526                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
6527         }
6528         instance->func_ptr->issue_cmd(cmd, instance);

6530         return (0);
6531 }
```

_____*unchanged_portion_omitted_*


```
6733 static void
6734 issue_cmd_ppc(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
6735 {
6736         struct scsi_pkt *pkt;
6737         atomic_add_16(&instance->fw_outstanding, 1);

6739         pkt = cmd->pkt;
6740         if (pkt) {
6741                 con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6742                     "ISSUED CMD TO FW : called : cmd:"
6743                     ": %p instance : %p pkt : %p pkt_time : %x\n",
6744                     gethrtime(), (void *)cmd, (void *)instance,
6745                     (void *)pkt, cmd->drv_pkt_time));
6746                 if (instance->adapterresetinprogress) {
6747                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
6748                         con_log(CL_ANN1, (CE_NOTE, "Reset the scsi_pkt timer"));
6749                 } else {
6750                         push_pending_mfi_pkt(instance, cmd);
6751                 }

6753         } else {
6754                 con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
6755                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
6756                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
6757         }

6759         mutex_enter(&instance->reg_write_mtx);
6760         /* Issue the command to the FW */
6761         WR_IB_PICK_QPORT((cmd->frame_phys_addr) |
6731         WR_IB_QPORT((cmd->frame_phys_addr) |
6762             (((cmd->frame_count - 1) << 1) | 1), instance);
6763         mutex_exit(&instance->reg_write_mtx);

6765 }

6767 /*
6768  * issue_cmd_in_sync_mode
6769  */
6770 static int
6771 issue_cmd_in_sync_mode_ppc(struct mrsas_instance *instance,
6772     struct mrsas_cmd *cmd)
6742 struct mrsas_cmd *cmd)
6773 {
6774         int     i;
6775         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
6745         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * (10 * MILLISEC);
6776         struct mrsas_header *hdr = &cmd->frame->hdr;

6778         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: called"));

6780         if (instance->adapterresetinprogress) {
6781                 cmd->drv_pkt_time = ddi_get16(
6782                     cmd->frame_dma_obj.acc_handle, &hdr->timeout);
6783                 if (cmd->drv_pkt_time < debug_timeout_g)
6784                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

6786                 con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: "
6787                     "issue and return in reset case\n"));
6788                 WR_IB_PICK_QPORT((cmd->frame_phys_addr) |
6758                 WR_IB_QPORT((cmd->frame_phys_addr) |
6789                     (((cmd->frame_count - 1) << 1) | 1), instance);

6791                 return (DDI_SUCCESS);
6792         } else {
6793                 con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: pushing the pkt\n"));
```

```
6794                 push_pending_mfi_pkt(instance, cmd);
6795         }

6797         cmd->cmd_status = ENODATA;

6799         mutex_enter(&instance->reg_write_mtx);
6800         /* Issue the command to the FW */
6801         WR_IB_PICK_QPORT((cmd->frame_phys_addr) |
6771         WR_IB_QPORT((cmd->frame_phys_addr) |
6802             (((cmd->frame_count - 1) << 1) | 1), instance);
6803         mutex_exit(&instance->reg_write_mtx);

6805         mutex_enter(&instance->int_cmd_mtx);
6806         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
6807                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
6808         }
6809         mutex_exit(&instance->int_cmd_mtx);

6811         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: done"));

6813         if (i < (msecs -1)) {
6814                 return (DDI_SUCCESS);
6815         } else {
6816                 return (DDI_FAILURE);
6817         }
6818 }

6820 /*
6821  * issue_cmd_in_poll_mode
6822  */
6823 static int
6824 issue_cmd_in_poll_mode_ppc(struct mrsas_instance *instance,
6825     struct mrsas_cmd *cmd)
6826 {
6827         int             i;
6828         uint16_t        flags;
6829         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
6830         struct mrsas_header *frame_hdr;

6832         con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_poll_mode_ppc: called"));

6834         frame_hdr = (struct mrsas_header *)cmd->frame;
6835         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
6836             MFI_CMD_STATUS_POLL_MODE);
6837         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
6838         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;

6840         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

6842         /* issue the frame using inbound queue port */
6843         WR_IB_PICK_QPORT((cmd->frame_phys_addr) |
6813         WR_IB_QPORT((cmd->frame_phys_addr) |
6844             (((cmd->frame_count - 1) << 1) | 1), instance);

6846         /* wait for cmd_status to change from 0xFF */
6847         for (i = 0; i < msecs && (
6848             ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
6849             == MFI_CMD_STATUS_POLL_MODE); i++) {
6850                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
6851         }

6853         if (ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
6854             == MFI_CMD_STATUS_POLL_MODE) {
6855                 con_log(CL_ANN, (CE_NOTE, "issue_cmd_in_poll_mode: "
6856                     "cmd polling timed out"));
6857                 return (DDI_FAILURE);
```



```
6858         }

6860         return (DDI_SUCCESS);
6861 }

6863 static void
6864 enable_intr_ppc(struct mrsas_instance *instance)
6865 {
6866         uint32_t        mask;

6868         con_log(CL_ANN1, (CE_NOTE, "enable_intr_ppc: called"));

6870         if (instance->skinny) {
6871                 /* For SKINNY, write ~0x1, from BSD's mfi driver. */
6872                 WR_OB_INTR_MASK(0xfffffffe, instance);
6873         } else {
6874                 /* WR_OB_DOORBELL_CLEAR(0xFFFFFFFF, instance); */
6875                 WR_OB_DOORBELL_CLEAR(OB_DOORBELL_CLEAR_MASK, instance);

6877                 /* WR_OB_INTR_MASK(~0x80000000, instance); */
6878                 WR_OB_INTR_MASK(~(MFI_REPLY_2108_MESSAGE_INTR_MASK), instance);
6879         }

6881         /* dummy read to force PCI flush */
6882         mask = RD_OB_INTR_MASK(instance);

6884         con_log(CL_ANN1, (CE_NOTE, "enable_intr_ppc: "
6885             "outbound_intr_mask = 0x%x", mask));
6886 }

6888 static void
6889 disable_intr_ppc(struct mrsas_instance *instance)
6890 {
6891         uint32_t        mask;

6893         con_log(CL_ANN1, (CE_NOTE, "disable_intr_ppc: called"));

6895         con_log(CL_ANN1, (CE_NOTE, "disable_intr_ppc: before : "
6896             "outbound_intr_mask = 0x%x", RD_OB_INTR_MASK(instance)));

6898         /* For now, assume there are no extras needed for Skinny support. */

6863         /* WR_OB_INTR_MASK(0xFFFFFFFF, instance); */
6900         WR_OB_INTR_MASK(OB_INTR_MASK, instance);

6902         con_log(CL_ANN1, (CE_NOTE, "disable_intr_ppc: after : "
6903             "outbound_intr_mask = 0x%x", RD_OB_INTR_MASK(instance)));

6905         /* dummy read to force PCI flush */
6906         mask = RD_OB_INTR_MASK(instance);
6907 #ifdef lint
6908         mask = mask;
6909 #endif
6910 }

6912 static int
6913 intr_ack_ppc(struct mrsas_instance *instance)
6914 {
6915         uint32_t        status;
6916         int ret = DDI_INTR_CLAIMED;

6918         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: called"));

6920         /* check if it is our interrupt */
6921         status = RD_OB_INTR_STATUS(instance);
```

```
6923         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: status = 0x%x", status));

6925         /*
6926          * NOTE:  Some drivers call out SKINNY here, but the return is the same
6927          * for SKINNY and 2108.
6928          */
6929         if (!(status & MFI_REPLY_2108_MESSAGE_INTR)) {
6930                 ret = DDI_INTR_UNCLAIMED;
6931         }

6933         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
6934                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
6935                 ret = DDI_INTR_UNCLAIMED;
6936         }

6938         if (ret == DDI_INTR_UNCLAIMED) {
6939                 return (ret);
6940         }

6942         /*
6943          * Clear the interrupt by writing back the same value.
6944          * Another case where SKINNY is slightly different.
6945          */
6946         if (instance->skinny) {
6947                 WR_OB_INTR_STATUS(status, instance);
6948         } else {
6901         /* clear the interrupt by writing back the same value */
6949                 WR_OB_DOORBELL_CLEAR(status, instance);
6950         }

6952         /* dummy READ */
6953         status = RD_OB_INTR_STATUS(instance);

6955         con_log(CL_ANN1, (CE_NOTE, "intr_ack_ppc: interrupt cleared"));

6957         return (ret);
6958 }
```

_____***unchanged_portion_omitted_***

```
7498 static int
7499 mrsas_tran_bus_config(dev_info_t *parent, uint_t flags,
7500     ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
7501 {
7502         struct mrsas_instance *instance;
7503         int config;
7504         int rval  = NDI_SUCCESS;

7506         char *ptr = NULL;
7507         int tgt, lun;

7509         con_log(CL_ANN1, (CE_NOTE, "Bus config called for op = %x", op));

7511         if ((instance = ddi_get_soft_state(mrsas_state,
7512             ddi_get_instance(parent))) == NULL) {
7513                 return (NDI_FAILURE);
7514         }

7516         /* Hold nexus during bus_config */
7517         ndi_devi_enter(parent, &config);
7518         switch (op) {
7519         case BUS_CONFIG_ONE: {

7521                 /* parse wwid/target name out of name given */
7522                 if ((ptr = strchr((char *)arg, '@')) == NULL) {
7523                         rval = NDI_FAILURE;
7524                         break;
```



```
7525                 }
7526                 ptr++;

7528                 if (mrsas_parse_devname(arg, &tgt, &lun) != 0) {
7529                         rval = NDI_FAILURE;
7530                         break;
7531                 }

7533                 if (lun == 0) {
7534                         rval = mrsas_config_ld(instance, tgt, lun, childp);
7535 #ifdef PDSUPPORT
7536                 } else if ((instance->tbolt || instance->skinny) && lun != 0) {
7488                 } else if (instance->tbolt == 1 && lun != 0) {
7537                         rval = mrsas_tbolt_config_pd(instance,
7538                             tgt, lun, childp);
7539 #endif
7540                 } else {
7541                         rval = NDI_FAILURE;
7542                 }

7544                 break;
7545         }
7546         case BUS_CONFIG_DRIVER:
7547         case BUS_CONFIG_ALL: {

7549                 rval = mrsas_config_all_devices(instance);

7551                 rval = NDI_SUCCESS;
7552                 break;
7553         }
7554         }

7556         if (rval == NDI_SUCCESS) {
7557                 rval = ndi_busop_bus_config(parent, flags, op, arg, childp, 0);

7559         }
7560         ndi_devi_exit(parent, config);

7562         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_bus_config: rval = %x",
7563             rval));
7564         return (rval);
7565 }

7567 static int
7568 mrsas_config_all_devices(struct mrsas_instance *instance)
7569 {
7570         int rval, tgt;

7572         for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
7573                 (void) mrsas_config_ld(instance, tgt, 0, NULL);

7575         }

7577 #ifdef PDSUPPORT
7578         /* Config PD devices connected to the card */
7579         if (instance->tbolt || instance->skinny) {
7531         if (instance->tbolt) {
7580                 for (tgt = 0; tgt < instance->mr_tbolt_pd_max; tgt++) {
7581                         (void) mrsas_tbolt_config_pd(instance, tgt, 1, NULL);
7582                 }
7583         }
7584 #endif

7586         rval = NDI_SUCCESS;
7587         return (rval);
7588 }
```

_____***unchanged_portion_omitted_***

```
7799 static void
7800 mrsas_issue_evt_taskq(struct mrsas_eventinfo *mrevt)
7801 {
7802         struct mrsas_instance *instance = mrevt->instance;
7803         dev_info_t *dip, *pdip;
7804         int circ1 = 0;
7805         char *devname;

7807         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_evt_taskq: called for"
7808             " tgt %d lun %d event %d",
7809             mrevt->tgt, mrevt->lun, mrevt->event));

7811         if (mrevt->tgt < MRDRV_MAX_LD && mrevt->lun == 0) {
7812                 mutex_enter(&instance->config_dev_mtx);
7813                 dip = instance->mr_ld_list[mrevt->tgt].dip;
7814                 mutex_exit(&instance->config_dev_mtx);
7815 #ifdef PDSUPPORT
7816         } else {
7817                 mutex_enter(&instance->config_dev_mtx);
7818                 dip = instance->mr_tbolt_pd_list[mrevt->tgt].dip;
7819                 mutex_exit(&instance->config_dev_mtx);
7820 #endif
7821         }


7824         ndi_devi_enter(instance->dip, &circ1);
7825         switch (mrevt->event) {
7826         case MRSAS_EVT_CONFIG_TGT:
7827                 if (dip == NULL) {

7829                         if (mrevt->lun == 0) {
7830                                 (void) mrsas_config_ld(instance, mrevt->tgt,
7831                                     0, NULL);
7832 #ifdef PDSUPPORT
7833                         } else if (instance->tbolt || instance->skinny) {
7785                         } else if (instance->tbolt) {
7834                                 (void) mrsas_tbolt_config_pd(instance,
7835                                     mrevt->tgt,
7836                                     1, NULL);
7837 #endif
7838                         }
7839                         con_log(CL_ANN1, (CE_NOTE,
7840                             "mr_sas: EVT_CONFIG_TGT called:"
7841                             " for tgt %d lun %d event %d",
7842                             mrevt->tgt, mrevt->lun, mrevt->event));

7844                 } else {
7845                         con_log(CL_ANN1, (CE_NOTE,
7846                             "mr_sas: EVT_CONFIG_TGT dip != NULL:"
7847                             " for tgt %d lun %d event %d",
7848                             mrevt->tgt, mrevt->lun, mrevt->event));
7849                 }
7850                 break;
7851         case MRSAS_EVT_UNCONFIG_TGT:
7852                 if (dip) {
7853                         if (i_ddi_devi_attached(dip)) {

7855                                 pdip = ddi_get_parent(dip);

7857                                 devname = kmem_zalloc(MAXNAMELEN + 1, KM_SLEEP);
7858                                 (void) ddi_deviname(dip, devname);

7860                                 (void) devfs_clean(pdip, devname + 1,
7861                                     DV_CLEAN_FORCE);
7862                                 kmem_free(devname, MAXNAMELEN + 1);
```



```
7863                         }
7864                         (void) ndi_devi_offline(dip, NDI_DEVI_REMOVE);
7865                         con_log(CL_ANN1, (CE_NOTE,
7866                             "mr_sas: EVT_UNCONFIG_TGT called:"
7867                             " for tgt %d lun %d event %d",
7868                             mrevt->tgt, mrevt->lun, mrevt->event));
7869                 } else {
7870                         con_log(CL_ANN1, (CE_NOTE,
7871                             "mr_sas: EVT_UNCONFIG_TGT dip == NULL:"
7872                             " for tgt %d lun %d event %d",
7873                             mrevt->tgt, mrevt->lun, mrevt->event));
7874                 }
7875                 break;
7876         }
7877         kmem_free(mrevt, sizeof (struct mrsas_eventinfo));
7878         ndi_devi_exit(instance->dip, circ1);
7879 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   56203 Tue Apr  2 11:54:25 2013
new/usr/src/uts/common/io/mr_sas/mr_sas.h
3500 Support LSI SAS2008 (Falcon) Skinny FW for mr_sas(7D)
**********************************************************
```

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

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _MR_SAS_H_
#define _MR_SAS_H_

#ifdef  __cplusplus
extern "C" {
#endif

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

/*
 * MegaRAID SAS2.0 Driver meta data
 */
```



```
 */
#define MRSAS_VERSION                           "6.503.00.00ILLUMOS"
#define MRSAS_RELDATE                           "July 30, 2012"

#define MRSAS_TRUE                              1
#define MRSAS_FALSE                             0

#define ADAPTER_RESET_NOT_REQUIRED              0
#define ADAPTER_RESET_REQUIRED                  1

#define PDSUPPORT       1

/*
 * MegaRAID SAS2.0 device id conversion definitions.
 */
#define INST2LSIRDCTL(x)                ((x) << INST_MINOR_SHIFT)
#define MRSAS_GET_BOUNDARY_ALIGNED_LEN(len, new_len, boundary_len)  { \
        int rem; \
        rem = (len / boundary_len); \
        if ((rem * boundary_len) != len) { \
                new_len = len + ((rem + 1) * boundary_len - len); \
        } else { \
                new_len = len; \
        } \
}


/*
 * MegaRAID SAS2.0 supported controllers
 */
#define PCI_DEVICE_ID_LSI_2108VDE               0x0078
#define PCI_DEVICE_ID_LSI_2108V                 0x0079
#define PCI_DEVICE_ID_LSI_SKINNY                0x0071
#define PCI_DEVICE_ID_LSI_SKINNY_NEW            0x0073
#define PCI_DEVICE_ID_LSI_TBOLT                 0x005b
#define PCI_DEVICE_ID_LSI_INVADER               0x005d

/*
 * Register Index for 2108 Controllers.
 */
#define REGISTER_SET_IO_2108                    (2)

#define MRSAS_MAX_SGE_CNT                       0x50
#define MRSAS_APP_RESERVED_CMDS                 32
#define MRSAS_APP_MIN_RESERVED_CMDS             4

#define MRSAS_IOCTL_DRIVER                      0x12341234
#define MRSAS_IOCTL_FIRMWARE                    0x12345678
#define MRSAS_IOCTL_AEN                         0x87654321

#define MRSAS_1_SECOND                          1000000

#ifdef PDSUPPORT

#define UNCONFIGURED_GOOD                       0x0
#define PD_SYSTEM                               0x40
#define MR_EVT_PD_STATE_CHANGE                  0x0072
#define MR_EVT_PD_REMOVED_EXT           0x00f8
#define MR_EVT_PD_INSERTED_EXT          0x00f7
#define MR_DCMD_PD_GET_INFO                     0x02020000
#define MRSAS_TBOLT_PD_LUN              1
#define MRSAS_TBOLT_PD_TGT_MAX  255
#define MRSAS_TBOLT_GET_PD_MAX(s)       ((s)->mr_tbolt_pd_max)

#endif
```

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

459 typedef struct mrsas_instance {
460         uint32_t        *producer;
461         uint32_t        *consumer;

463         uint32_t        *reply_queue;
464         dma_obj_t       mfi_internal_dma_obj;
465         uint16_t        adapterresetinprogress;
466         uint16_t        deadadapter;
467         /* ThunderBolt (TB) specific */
468         dma_obj_t       mpi2_frame_pool_dma_obj;
469         dma_obj_t       request_desc_dma_obj;
470         dma_obj_t       reply_desc_dma_obj;
471         dma_obj_t       ld_map_obj[2];

473         uint8_t         init_id;
474         uint8_t         flag_ieee;
475         uint8_t         disable_online_ctrl_reset;
476         uint8_t         fw_fault_count_after_ocr;

478         uint16_t        max_num_sge;
479         uint16_t        max_fw_cmds;
480         uint32_t        max_sectors_per_req;

482         struct mrsas_cmd **cmd_list;

484         mlist_t         cmd_pool_list;
485         kmutex_t        cmd_pool_mtx;
486         kmutex_t        sync_map_mtx;

488         mlist_t         app_cmd_pool_list;
489         kmutex_t        app_cmd_pool_mtx;
490         mlist_t         cmd_app_pool_list;
491         kmutex_t        cmd_app_pool_mtx;


494         mlist_t         cmd_pend_list;
495         kmutex_t        cmd_pend_mtx;

497         dma_obj_t       mfi_evt_detail_obj;
498         struct mrsas_cmd *aen_cmd;

500         uint32_t        aen_seq_num;
501         uint32_t        aen_class_locale_word;

503         scsi_hba_tran_t         *tran;

505         kcondvar_t      int_cmd_cv;
506         kmutex_t        int_cmd_mtx;

508         kcondvar_t      aen_cmd_cv;
509         kmutex_t        aen_cmd_mtx;

511         kcondvar_t      abort_cmd_cv;
512         kmutex_t        abort_cmd_mtx;

514         kmutex_t        reg_write_mtx;
```



```
515         kmutex_t        chip_mtx;

517         dev_info_t              *dip;
518         ddi_acc_handle_t        pci_handle;

520         timeout_id_t    timeout_id;
521         uint32_t        unique_id;
522         uint16_t        fw_outstanding;
523         caddr_t         regmap;
524         ddi_acc_handle_t        regmap_handle;
525         uint8_t         isr_level;
526         ddi_iblock_cookie_t     iblock_cookie;
527         ddi_iblock_cookie_t     soft_iblock_cookie;
528         ddi_softintr_t          soft_intr_id;
529         uint8_t         softint_running;
530         uint8_t         tbolt_softint_running;
531         kmutex_t        completed_pool_mtx;
532         mlist_t         completed_pool_list;

534         caddr_t         internal_buf;
535         uint32_t        internal_buf_dmac_add;
536         uint32_t        internal_buf_size;

538         uint16_t        vendor_id;
539         uint16_t        device_id;
540         uint16_t        subsysvid;
541         uint16_t        subsysid;
542         int             instance;
543         int             baseaddress;
544         char            iocnode[16];

546         int             fm_capabilities;
547         /*
548          * Driver resources unroll flags.  The flag is set for resources that
549          * are needed to be free'd at detach() time.
550          */
551         struct _unroll {
552                 uint8_t softs;          /* The software state was allocated. */
553                 uint8_t regs;           /* Controller registers mapped. */
554                 uint8_t intr;           /* Interrupt handler added. */
555                 uint8_t reqs;           /* Request structs allocated. */
556                 uint8_t mutexs;         /* Mutex's allocated. */
557                 uint8_t taskq;          /* Task q's created. */
558                 uint8_t tran;           /* Tran struct allocated */
559                 uint8_t tranSetup;      /* Tran attached to the ddi. */
560                 uint8_t devctl;         /* Device nodes for cfgadm created. */
561                 uint8_t scsictl;        /* Device nodes for cfgadm created. */
562                 uint8_t ioctl;          /* Device nodes for ioctl's created. */
563                 uint8_t timer;          /* Timer started. */
564                 uint8_t aenPend;        /* AEN cmd pending f/w. */
565                 uint8_t mapUpdate_pend; /* LD MAP update cmd pending f/w. */
566                 uint8_t soft_isr;       /* Soft interrupt handler allocated. */
567                 uint8_t ldlist_buff;    /* Logical disk list allocated. */
568                 uint8_t pdlist_buff;    /* Physical disk list allocated. */
569                 uint8_t syncCmd;        /* Sync map command allocated. */
570                 uint8_t verBuff;        /* 2108 MFI buffer allocated. */
571                 uint8_t alloc_space_mfi;  /* Allocated space for 2108 MFI. */
572                 uint8_t alloc_space_mpi2; /* Allocated space for 2208 MPI2. */
573         } unroll;


576         /* function template pointer */
577         struct mrsas_function_template *func_ptr;


580         /* MSI interrupts specific */
```

```
 581         ddi_intr_handle_t *intr_htable;         /* Interrupt handle array */
 582         size_t          intr_htable_size;       /* Int. handle array size */
 583         int             intr_type;
 584         int             intr_cnt;
 585         uint_t          intr_pri;
 586         int             intr_cap;

 588         ddi_taskq_t     *taskq;
 589         struct mrsas_ld *mr_ld_list;
 590         kmutex_t        config_dev_mtx;
 591         /* ThunderBolt (TB) specific */
 592         ddi_softintr_t  tbolt_soft_intr_id;

 594 #ifdef PDSUPPORT
 595         uint32_t        mr_tbolt_pd_max;
 596         struct mrsas_tbolt_pd *mr_tbolt_pd_list;
 597 #endif

 599         uint8_t         fast_path_io;

 601         uint8_t         skinny;
 602         uint8_t         tbolt;
 597         uint16_t        tbolt;
 603         uint16_t        reply_read_index;
 604         uint16_t        reply_size;             /* Single Reply struct size */
 605         uint16_t        raid_io_msg_size;       /* Single message size */
 606         uint32_t        io_request_frames_phy;
 607         uint8_t         *io_request_frames;
 608         /* Virtual address of request desc frame pool */
 609         MRSAS_REQUEST_DESCRIPTOR_UNION  *request_message_pool;
 610         /* Physical address of request desc frame pool */
 611         uint32_t        request_message_pool_phy;
 612         /* Virtual address of reply Frame */
 613         MPI2_REPLY_DESCRIPTORS_UNION    *reply_frame_pool;
 614         /* Physical address of reply Frame */
 615         uint32_t        reply_frame_pool_phy;
 616         uint8_t         *reply_pool_limit;      /* Last reply frame address */
 617         /* Physical address of Last reply frame */
 618         uint32_t        reply_pool_limit_phy;
 619         uint32_t        reply_q_depth;          /* Reply Queue Depth */
 620         uint8_t         max_sge_in_main_msg;
 621         uint8_t         max_sge_in_chain;
 622         uint8_t         chain_offset_io_req;
 623         uint8_t         chain_offset_mpt_msg;
 624         MR_FW_RAID_MAP_ALL *ld_map[2];
 625         uint32_t        ld_map_phy[2];
 626         uint32_t        size_map_info;
 627         uint64_t        map_id;
 628         LD_LOAD_BALANCE_INFO load_balance_info[MAX_LOGICAL_DRIVES];
 629         struct mrsas_cmd *map_update_cmd;
 630         uint32_t        SyncRequired;
 631         kmutex_t        ocr_flags_mtx;
 632         dma_obj_t       drv_ver_dma_obj;
 633 } mrsas_t;
_____unchanged_portion_omitted_

1955 #pragma pack()

1957 #ifndef DDI_VENDOR_LSI
1958 #define DDI_VENDOR_LSI          "LSI"
1959 #endif /* DDI_VENDOR_LSI */

1961 int mrsas_config_scsi_device(struct mrsas_instance *,
1962     struct scsi_device *, dev_info_t **);

1964 #ifdef PDSUPPORT
```



```
1965 int mrsas_tbolt_config_pd(struct mrsas_instance *, uint16_t,
1966     uint8_t, dev_info_t **);
1967 #endif

1969 dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t, uint8_t);
1970 int mrsas_service_evt(struct mrsas_instance *, int, int, int, uint64_t);
1971 void return_raid_msg_pkt(struct mrsas_instance *, struct mrsas_cmd *);
1972 struct mrsas_cmd *get_raid_msg_mfi_pkt(struct mrsas_instance *);
1973 void return_raid_msg_mfi_pkt(struct mrsas_instance *, struct mrsas_cmd *);

1975 int     alloc_space_for_mpi2(struct mrsas_instance *);
1976 void    fill_up_drv_ver(struct mrsas_drv_ver *dv);

1978 int     mrsas_issue_init_mpi2(struct mrsas_instance *);
1979 struct scsi_pkt *mrsas_tbolt_tran_init_pkt(struct scsi_address *, register
1980                     struct scsi_pkt *, struct buf *, int, int, int, int,
1981                     int (*)(), caddr_t);
1982 int     mrsas_tbolt_tran_start(struct scsi_address *,
1983                     register struct scsi_pkt *);
1984 uint32_t tbolt_read_fw_status_reg(struct mrsas_instance *);
1985 void    tbolt_issue_cmd(struct mrsas_cmd *, struct mrsas_instance *);
1986 int     tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *,
1987                     struct mrsas_cmd *);
1988 int     tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *,
1989                     struct mrsas_cmd *);
1990 void    tbolt_enable_intr(struct mrsas_instance *);
1991 void    tbolt_disable_intr(struct mrsas_instance *);
1992 int     tbolt_intr_ack(struct mrsas_instance *);
1993 uint_t  mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *);
1994     uint_t tbolt_softintr();
1995 int     mrsas_tbolt_dma(struct mrsas_instance *, uint32_t, int, int (*)());
1996 int     mrsas_check_dma_handle(ddi_dma_handle_t handle);
1997 int     mrsas_check_acc_handle(ddi_acc_handle_t handle);
1998 int     mrsas_dma_alloc(struct mrsas_instance *, struct scsi_pkt *,
1999                     struct buf *, int, int (*)());
2000 int     mrsas_dma_move(struct mrsas_instance *,
2001                         struct scsi_pkt *, struct buf *);
2002 int     mrsas_alloc_dma_obj(struct mrsas_instance *, dma_obj_t *,
2003                     uchar_t);
2004 void    mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2005 int     mrsas_dma_alloc_dmd(struct mrsas_instance *, dma_obj_t *);
2006 void    tbolt_complete_cmd_in_sync_mode(struct mrsas_instance *,
2007         struct mrsas_cmd *);
2008 int     alloc_req_rep_desc(struct mrsas_instance *);
2009 int             mrsas_mode_sense_build(struct scsi_pkt *);
2010 void            push_pending_mfi_pkt(struct mrsas_instance *,
2011                     struct mrsas_cmd *);
2012 int     mrsas_issue_pending_cmds(struct mrsas_instance *);
2013 int     mrsas_print_pending_cmds(struct mrsas_instance *);
2014 int     mrsas_complete_pending_cmds(struct mrsas_instance *);

2016 int     create_mfi_frame_pool(struct mrsas_instance *);
2017 void    destroy_mfi_frame_pool(struct mrsas_instance *);
2018 int     create_mfi_mpi_frame_pool(struct mrsas_instance *);
2019 void    destroy_mfi_mpi_frame_pool(struct mrsas_instance *);
2020 int     create_mpi2_frame_pool(struct mrsas_instance *);
2021 void    destroy_mpi2_frame_pool(struct mrsas_instance *);
2022 int     mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t);
2023 void    mrsas_tbolt_free_additional_dma_buffer(struct mrsas_instance *);
2024 void    free_req_desc_pool(struct mrsas_instance *);
2025 void    free_space_for_mpi2(struct mrsas_instance *);
2026 void    mrsas_dump_reply_desc(struct mrsas_instance *);
2027 void    tbolt_complete_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2028 void    display_scsi_inquiry(caddr_t);
2029 void    service_mfi_aen(struct mrsas_instance *, struct mrsas_cmd *);
2030 int     mrsas_mode_sense_build(struct scsi_pkt *);
```

```
2031 int     mrsas_tbolt_get_ld_map_info(struct mrsas_instance *);
2032 struct mrsas_cmd *mrsas_tbolt_build_poll_cmd(struct mrsas_instance *,
2033         struct scsi_address *, struct scsi_pkt *, uchar_t *);
2034 int     mrsas_tbolt_reset_ppc(struct mrsas_instance *instance);
2035 void    mrsas_tbolt_kill_adapter(struct mrsas_instance *instance);
2036 int     abort_syncmap_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2037 void    mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2038     struct IO_REQUEST_INFO *, Mpi2RaidSCSIIORequest_t *, U32);


2041 int mrsas_init_adapter_ppc(struct mrsas_instance *instance);
2042 int mrsas_init_adapter_tbolt(struct mrsas_instance *instance);
2043 int mrsas_init_adapter(struct mrsas_instance *instance);

2045 int mrsas_alloc_cmd_pool(struct mrsas_instance *instance);
2046 void mrsas_free_cmd_pool(struct mrsas_instance *instance);

2048 void mrsas_print_cmd_details(struct mrsas_instance *, struct mrsas_cmd *, int);
2049 struct mrsas_cmd *get_raid_msg_pkt(struct mrsas_instance *);

2051 int mfi_state_transition_to_ready(struct mrsas_instance *);

2053 struct mrsas_cmd *mrsas_get_mfi_pkt(struct mrsas_instance *);
2054 void mrsas_return_mfi_pkt(struct mrsas_instance *, struct mrsas_cmd *);


2057 /* FMA functions. */
2058 int mrsas_common_check(struct mrsas_instance *, struct  mrsas_cmd *);
2059 void mrsas_fm_ereport(struct mrsas_instance *, char *);


2062 #ifdef  __cplusplus
2063 }
```

*unchanged_portion_omitted_*

```
**********************************************************
  105163 Tue Apr  2 11:54:25 2013
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
3500 Support LSI SAS2008 (Falcon) Skinny FW for mr_sas(7D)
**********************************************************
```

```
/*
 * 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.
 */


#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 void
io_timeout_checker(void *arg);
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_***

```
/*
 * mrsas_tbolt_tran_init_pkt - allocate & initialize a scsi_pkt structure
 * @ap:
 * @pkt:
 * @bp:
 * @cmdlen:
 * @statuslen:
 * @tgtlen:
 * @flags:
 * @callback:
 *
 * The tran_init_pkt() entry point allocates and initializes a scsi_pkt
 * structure and DMA resources for a target driver request. The
 * tran_init_pkt() entry point is called when the target driver calls the
 * SCSA function scsi_init_pkt(). Each call of the tran_init_pkt() entry point
 * is a request to perform one or more of three possible services:
 *  - allocation and initialization of a scsi_pkt structure
 *  - allocation of DMA resources for data transfer
 *  - reallocation of DMA resources for the next portion of the data transfer
 */
struct scsi_pkt *
mrsas_tbolt_tran_init_pkt(struct scsi_address *ap,
        register struct scsi_pkt *pkt,
        struct buf *bp, int cmdlen, int statuslen, int tgtlen,
        int flags, int (*callback)(), caddr_t arg)
{
        struct scsa_cmd *acmd;
```

```
1912         struct mrsas_instance   *instance;
1913         struct scsi_pkt *new_pkt;

1915         instance = ADDR2MR(ap);

1917         /* step #1 : pkt allocation */
1918         if (pkt == NULL) {
1919                 pkt = scsi_hba_pkt_alloc(instance->dip, ap, cmdlen, statuslen,
1920                     tgtlen, sizeof (struct scsa_cmd), callback, arg);
1921                 if (pkt == NULL) {
1922                         return (NULL);
1923                 }

1925                 acmd = PKT2CMD(pkt);

1927                 /*
1928                  * Initialize the new pkt - we redundantly initialize
1929                  * all the fields for illustrative purposes.
1930                  */
1931                 acmd->cmd_pkt           = pkt;
1932                 acmd->cmd_flags         = 0;
1933                 acmd->cmd_scblen        = statuslen;
1934                 acmd->cmd_cdblen        = cmdlen;
1935                 acmd->cmd_dmahandle     = NULL;
1936                 acmd->cmd_ncookies      = 0;
1937                 acmd->cmd_cookie        = 0;
1938                 acmd->cmd_cookiecnt     = 0;
1939                 acmd->cmd_nwin          = 0;

1941                 pkt->pkt_address        = *ap;
1942                 pkt->pkt_comp           = (void (*)())NULL;
1943                 pkt->pkt_flags          = 0;
1944                 pkt->pkt_time           = 0;
1945                 pkt->pkt_resid          = 0;
1946                 pkt->pkt_state          = 0;
1947                 pkt->pkt_statistics     = 0;
1948                 pkt->pkt_reason         = 0;
1949                 new_pkt                 = pkt;
1950         } else {
1951                 acmd = PKT2CMD(pkt);
1952                 new_pkt = NULL;
1953         }

1955         /* step #2 : dma allocation/move */
1956         if (bp && bp->b_bcount != 0) {
1957                 if (acmd->cmd_dmahandle == NULL) {
1958                         if (mrsas_dma_alloc(instance, pkt, bp, flags,
1959                             callback) == DDI_FAILURE) {
1960                                 if (new_pkt) {
1961                                         scsi_hba_pkt_free(ap, new_pkt);
1962                                 }
1963                                 return ((struct scsi_pkt *)NULL);
1964                         }
1965                 } else {
1966                         if (mrsas_dma_move(instance, pkt, bp) == DDI_FAILURE) {
1967                                 return ((struct scsi_pkt *)NULL);
1968                         }
1969                 }
1970         }
1971         return (pkt);
1972 }


1887 uint32_t
1888 tbolt_read_fw_status_reg(struct mrsas_instance *instance)
1889 {
```



```
1890         return ((uint32_t)RD_OB_SCRATCH_PAD_0(instance));
1891 }
_______unchanged_portion_omitted_


3546 #ifdef PDSUPPORT
3547 /*
3548  * Even though these functions were originally intended for 2208 only, it
3549  * turns out they're useful for "Skinny" support as well.  In a perfect world,
3550  * these two functions would be either in mr_sas.c, or in their own new source
3551  * file.  Since this driver needs some cleanup anyway, keep this portion in
3552  * mind as well.
3553  */

3555 int
3556 mrsas_tbolt_config_pd(struct mrsas_instance *instance, uint16_t tgt,
3557     uint8_t lun, dev_info_t **ldip)
3558 {
3559         struct scsi_device *sd;
3560         dev_info_t *child;
3561         int rval, dtype;
3562         struct mrsas_tbolt_pd_info *pds = NULL;

3564         con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_config_pd: t = %d l = %d",
3565             tgt, lun));

3567         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
3568                 if (ldip) {
3569                         *ldip = child;
3570                 }
3571                 if (instance->mr_tbolt_pd_list[tgt].flag != MRDRV_TGT_VALID) {
3572                         rval = mrsas_service_evt(instance, tgt, 1,
3573                             MRSAS_EVT_UNCONFIG_TGT, NULL);
3574                         con_log(CL_ANN1, (CE_WARN,
3575                             "mr_sas:DELETING STALE ENTRY  rval = %d "
3576                             "tgt id = %d", rval, tgt));
3577                         return (NDI_FAILURE);
3578                 }
3579                 return (NDI_SUCCESS);
3580         }

3582         pds = (struct mrsas_tbolt_pd_info *)
3583             kmem_zalloc(sizeof (struct mrsas_tbolt_pd_info), KM_SLEEP);
3584         mrsas_tbolt_get_pd_info(instance, pds, tgt);
3585         dtype = pds->scsiDevType;

3587         /* Check for Disk */
3588         if ((dtype == DTYPE_DIRECT)) {
3589                 if ((dtype == DTYPE_DIRECT) &&
3590                     (LE_16(pds->fwState) != PD_SYSTEM)) {
3591                         kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3592                         return (NDI_FAILURE);
3593                 }
3594                 sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
3595                 sd->sd_address.a_hba_tran = instance->tran;
3596                 sd->sd_address.a_target = (uint16_t)tgt;
3597                 sd->sd_address.a_lun = (uint8_t)lun;

3599                 if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS) {
3600                         rval = mrsas_config_scsi_device(instance, sd, ldip);
3601                         con_log(CL_DLEVEL1, (CE_NOTE,
3602                             "Phys. device found: tgt %d dtype %d: %s",
3603                             tgt, dtype, sd->sd_inq->inq_vid));
3604                 } else {
3605                         rval = NDI_FAILURE;
3606                         con_log(CL_DLEVEL1, (CE_NOTE, "Phys. device Not found "
```

```
3607                            "scsi_hba_probe Failed: tgt %d dtype %d: %s",
3608                            tgt, dtype, sd->sd_inq->inq_vid));
3609                }

3611                /* sd_unprobe is blank now. Free buffer manually */
3612                if (sd->sd_inq) {
3613                        kmem_free(sd->sd_inq, SUN_INQSIZE);
3614                        sd->sd_inq = (struct scsi_inquiry *)NULL;
3615                }
3616                kmem_free(sd, sizeof (struct scsi_device));
3617                rval = NDI_SUCCESS;
3617        } else {
3618                con_log(CL_ANN1, (CE_NOTE,
3619                    "Device not supported: tgt %d lun %d dtype %d",
3620                    tgt, lun, dtype));
3621                rval = NDI_FAILURE;
3622        }

3624        kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3625        con_log(CL_ANN1, (CE_NOTE, "mrsas_config_pd: return rval = %d",
3626            rval));
3627        return (rval);
3628 }

3630 static void
3631 mrsas_tbolt_get_pd_info(struct mrsas_instance *instance,
3632     struct mrsas_tbolt_pd_info *pds, int tgt)
3633 {
3634        struct mrsas_cmd        *cmd;
3635        struct mrsas_dcmd_frame *dcmd;
3636        dma_obj_t               dcmd_dma_obj;

3638        ASSERT(instance->tbolt || instance->skinny);

3640        if (instance->tbolt)
3641                cmd = get_raid_msg_pkt(instance);
3642        else
3643                cmd = mrsas_get_mfi_pkt(instance);

3645        if (!cmd) {
3646                con_log(CL_ANN1,
3647                    (CE_WARN, "Failed to get a cmd for get pd info"));
3648                return;
3649        }

3651        /* Clear the frame buffer and assign back the context id */
3652        bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3653        ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3654            cmd->index);


3657        dcmd = &cmd->frame->dcmd;
3658        dcmd_dma_obj.size = sizeof (struct mrsas_tbolt_pd_info);
3659        dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
3660        dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xffffffff;
3661        dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xffffffff;
3662        dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
3663        dcmd_dma_obj.dma_attr.dma_attr_align = 1;

3665        (void) mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
3666            DDI_STRUCTURE_LE_ACC);
3667        bzero(dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info));
3668        bzero(dcmd->mbox.b, 12);
3669        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3670        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
3671        ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
```



```
3672        ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3673            MFI_FRAME_DIR_READ);
3674        ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3675        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3676            sizeof (struct mrsas_tbolt_pd_info));
3677        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3678            MR_DCMD_PD_GET_INFO);
3679        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], tgt);
3680        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3681            sizeof (struct mrsas_tbolt_pd_info));
3682        ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3683            dcmd_dma_obj.dma_cookie[0].dmac_address);

3685        cmd->sync_cmd = MRSAS_TRUE;
3686        cmd->frame_count = 1;

3688        if (instance->tbolt)
3764        if (instance->tbolt) {
3689                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3766        }

3691        instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd);

3693        ddi_rep_get8(cmd->frame_dma_obj.acc_handle, (uint8_t *)pds,
3694            (uint8_t *)dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info),
3695            DDI_DEV_AUTOINCR);
3696        (void) mrsas_free_dma_obj(instance, dcmd_dma_obj);

3698        if (instance->tbolt)
3699                return_raid_msg_pkt(instance, cmd);
3700        else
3701                mrsas_return_mfi_pkt(instance, cmd);
3702 }
______unchanged_portion_omitted_
```