```
**********************************************************
  413826 Tue Dec  4 12:30:12 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
XXXX Nexenta fixes for mpt_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) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * Copyright (c) 2000 to 2010, LSI Corporation.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms of all code within
 * this file that is exclusively owned by LSI, with or without
 * modification, is permitted provided that, in addition to the CDDL 1.0
 * License requirements, the following conditions are met:
 *
 *    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.
 */

/*
 * mptsas - This is a driver based on LSI Logic's MPT2.0 interface.
 *
 */

#if defined(lint) || defined(DEBUG)
#define MPTSAS_DEBUG
#endif
```



```
/*
 * standard header files.
 */
#include <sys/note.h>
#include <sys/scsi/scsi.h>
#include <sys/pci.h>
#include <sys/file.h>
#include <sys/cpuvar.h>
#include <sys/policy.h>
#include <sys/sysevent.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dr.h>
#include <sys/sata/sata_defs.h>
#include <sys/scsi/generic/sas.h>
#include <sys/scsi/impl/scsi_sas.h>

#pragma pack(1)
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_type.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_init.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_ioc.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_sas.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_raid.h>
#pragma pack()

/*
 * private header files.
 *
 */
#include <sys/scsi/impl/scsi_reset_notify.h>
#include <sys/scsi/adapters/mpt_sas/mptsas_var.h>
#include <sys/scsi/adapters/mpt_sas/mptsas_ioctl.h>
#include <sys/scsi/adapters/mpt_sas/mptsas_smhba.h>

#include <sys/raidioctl.h>

#include <sys/fs/dv_node.h>     /* devfs_clean */

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

/*
 * For anyone who would modify the code in mptsas_driver, it must be awared
 * that from snv_145 where CR6910752(mpt_sas driver performance can be
 * improved) is integrated, the per_instance mutex m_mutex is not hold
 * in the key IO code path, including mptsas_scsi_start(), mptsas_intr()
 * and all of the recursive functions called in them, so don't
 * make it for granted that all operations are sync/exclude correctly. Before
 * doing any modification in key code path, and even other code path such as
 * DR, watchsubr, ioctl, passthrough etc, make sure the elements modified have
 * no releationship to elements shown in the fastpath
 * (function mptsas_handle_io_fastpath()) in ISR and its recursive functions.
 * otherwise, you have to use the new introduced mutex to protect them.
 * As to how to do correctly, refer to the comments in mptsas_intr().
 */

/*
 * autoconfiguration data and routines.
```

```
111  */
112 static int mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
113 static int mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
114 static int mptsas_power(dev_info_t *dip, int component, int level);

116 /*
117  * cb_ops function
118  */
119 static int mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
120         cred_t *credp, int *rval);
121 #ifdef __sparc
122 static int mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd);
123 #else  /* __sparc */
124 static int mptsas_quiesce(dev_info_t *devi);
125 #endif  /* __sparc */

127 /*
128  * Resource initilaization for hardware
129  */
130 static void mptsas_setup_cmd_reg(mptsas_t *mpt);
131 static void mptsas_disable_bus_master(mptsas_t *mpt);
132 static void mptsas_hba_fini(mptsas_t *mpt);
133 static void mptsas_cfg_fini(mptsas_t *mptsas_blkp);
134 static int mptsas_hba_setup(mptsas_t *mpt);
135 static void mptsas_hba_teardown(mptsas_t *mpt);
136 static int mptsas_config_space_init(mptsas_t *mpt);
137 static void mptsas_config_space_fini(mptsas_t *mpt);
138 static void mptsas_iport_register(mptsas_t *mpt);
139 static int mptsas_smp_setup(mptsas_t *mpt);
140 static void mptsas_smp_teardown(mptsas_t *mpt);
141 static int mptsas_cache_create(mptsas_t *mpt);
142 static void mptsas_cache_destroy(mptsas_t *mpt);
143 static int mptsas_alloc_request_frames(mptsas_t *mpt);
144 static int mptsas_alloc_reply_frames(mptsas_t *mpt);
145 static int mptsas_alloc_free_queue(mptsas_t *mpt);
146 static int mptsas_alloc_post_queue(mptsas_t *mpt);
147 static void mptsas_alloc_reply_args(mptsas_t *mpt);
148 static int mptsas_alloc_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd);
149 static void mptsas_free_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd);
150 static int mptsas_init_chip(mptsas_t *mpt, int first_time);

152 /*
153  * SCSA function prototypes
154  */
155 static int mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt);
156 static int mptsas_scsi_reset(struct scsi_address *ap, int level);
157 static int mptsas_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
158 static int mptsas_scsi_getcap(struct scsi_address *ap, char *cap, int tgtonly);
159 static int mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value,
160     int tgtonly);
161 static void mptsas_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt);
162 static struct scsi_pkt *mptsas_scsi_init_pkt(struct scsi_address *ap,
163     struct scsi_pkt *pkt, struct buf *bp, int cmdlen, int statuslen,
164         int tgtlen, int flags, int (*callback)(), caddr_t arg);
165 static void mptsas_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt);
166 static void mptsas_scsi_destroy_pkt(struct scsi_address *ap,
167     struct scsi_pkt *pkt);
168 static int mptsas_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
169     scsi_hba_tran_t *hba_tran, struct scsi_device *sd);
170 static void mptsas_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
171     scsi_hba_tran_t *hba_tran, struct scsi_device *sd);
172 static int mptsas_scsi_reset_notify(struct scsi_address *ap, int flag,
173     void (*callback)(caddr_t), caddr_t arg);
174 static int mptsas_get_name(struct scsi_device *sd, char *name, int len);
175 static int mptsas_get_bus_addr(struct scsi_device *sd, char *name, int len);
176 static int mptsas_scsi_quiesce(dev_info_t *dip);
```



```
177 static int mptsas_scsi_unquiesce(dev_info_t *dip);
178 static int mptsas_bus_config(dev_info_t *pdip, uint_t flags,
179     ddi_bus_config_op_t op, void *arg, dev_info_t **childp);

181 /*
182  * SMP functions
183  */
184 static int mptsas_smp_start(struct smp_pkt *smp_pkt);

186 /*
187  * internal function prototypes.
188  */
189 static void mptsas_list_add(mptsas_t *mpt);
190 static void mptsas_list_del(mptsas_t *mpt);

192 static int mptsas_quiesce_bus(mptsas_t *mpt);
193 static int mptsas_unquiesce_bus(mptsas_t *mpt);

195 static int mptsas_alloc_handshake_msg(mptsas_t *mpt, size_t alloc_size);
196 static void mptsas_free_handshake_msg(mptsas_t *mpt);

198 static void mptsas_ncmds_checkdrain(void *arg);

200 static int mptsas_prepare_pkt(mptsas_cmd_t *cmd);
201 static int mptsas_accept_pkt(mptsas_t *mpt, mptsas_cmd_t *sp);
202 static int mptsas_accept_txwq_and_pkt(mptsas_t *mpt, mptsas_cmd_t *sp);
203 static void mptsas_accept_tx_waitq(mptsas_t *mpt);

205 static int mptsas_do_detach(dev_info_t *dev);
206 static int mptsas_do_scsi_reset(mptsas_t *mpt, uint16_t devhdl);
207 static int mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun,
208     struct scsi_pkt *pkt);
209 static int mptsas_scsi_capchk(char *cap, int tgtonly, int *cidxp);

211 static void mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd);
212 static void mptsas_handle_event(void *args);
213 static int mptsas_handle_event_sync(void *args);
214 static void mptsas_handle_dr(void *args);
215 static void mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
216     dev_info_t *pdip);

218 static void mptsas_restart_cmd(void *);

220 static void mptsas_flush_hba(mptsas_t *mpt);
221 static void mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun,
222         uint8_t tasktype);
223 static void mptsas_set_pkt_reason(mptsas_t *mpt, mptsas_cmd_t *cmd,
224     uchar_t reason, uint_t stat);

226 static uint_t mptsas_intr(caddr_t arg1, caddr_t arg2);
227 static void mptsas_process_intr(mptsas_t *mpt,
228     pMpi2ReplyDescriptorsUnion_t reply_desc_union);
244 static int mptsas_handle_io_fastpath(mptsas_t *mpt, uint16_t SMID);
229 static void mptsas_handle_scsi_io_success(mptsas_t *mpt,
230     pMpi2ReplyDescriptorsUnion_t reply_desc);
231 static void mptsas_handle_address_reply(mptsas_t *mpt,
232     pMpi2ReplyDescriptorsUnion_t reply_desc);
233 static int mptsas_wait_intr(mptsas_t *mpt, int polltime);
234 static void mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd,
235     uint32_t *control, pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl);

237 static void mptsas_watch(void *arg);
238 static void mptsas_watchsubr(mptsas_t *mpt);
239 static void mptsas_cmd_timeout(mptsas_t *mpt, uint16_t devhdl);
240 static void mptsas_kill_target(mptsas_t *mpt, mptsas_target_t *ptgt);
```

```
static void mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd);
static int mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
    uint8_t *data, uint32_t request_size, uint32_t reply_size,
    uint32_t data_size, uint32_t direction, uint8_t *dataout,
    uint32_t dataout_size, short timeout, int mode);
static int mptsas_free_devhdl(mptsas_t *mpt, uint16_t devhdl);

static uint8_t mptsas_get_fw_diag_buffer_number(mptsas_t *mpt,
    uint32_t unique_id);
static void mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd);
static int mptsas_post_fw_diag_buffer(mptsas_t *mpt,
    mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code);
static int mptsas_release_fw_diag_buffer(mptsas_t *mpt,
    mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
    uint32_t diag_type);
static int mptsas_diag_register(mptsas_t *mpt,
    mptsas_fw_diag_register_t *diag_register, uint32_t *return_code);
static int mptsas_diag_unregister(mptsas_t *mpt,
    mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code);
static int mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
    uint32_t *return_code);
static int mptsas_diag_read_buffer(mptsas_t *mpt,
    mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
    uint32_t *return_code, int ioctl_mode);
static int mptsas_diag_release(mptsas_t *mpt,
    mptsas_fw_diag_release_t *diag_release, uint32_t *return_code);
static int mptsas_do_diag_action(mptsas_t *mpt, uint32_t action,
    uint8_t *diag_action, uint32_t length, uint32_t *return_code,
    int ioctl_mode);
static int mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *data,
    int mode);

static int mptsas_pkt_alloc_extern(mptsas_t *mpt, mptsas_cmd_t *cmd,
    int cmdlen, int tgtlen, int statuslen, int kf);
static void mptsas_pkt_destroy_extern(mptsas_t *mpt, mptsas_cmd_t *cmd);

static int mptsas_kmem_cache_constructor(void *buf, void *cdrarg, int kmflags);
static void mptsas_kmem_cache_destructor(void *buf, void *cdrarg);

static int mptsas_cache_frames_constructor(void *buf, void *cdrarg,
    int kmflags);
static void mptsas_cache_frames_destructor(void *buf, void *cdrarg);

static void mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
    mptsas_cmd_t *cmd);
static void mptsas_check_task_mgt(mptsas_t *mpt,
    pMpi2SCSIManagementReply_t reply, mptsas_cmd_t *cmd);
static int mptsas_send_scsi_cmd(mptsas_t *mpt, struct scsi_address *ap,
    mptsas_target_t *ptgt, uchar_t *cdb, int cdblen, struct buf *data_bp,
    int *resid);

static int mptsas_alloc_active_slots(mptsas_t *mpt, int flag);
static void mptsas_free_active_slots(mptsas_t *mpt);
static int mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);
static int mptsas_start_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd);

static void mptsas_restart_hba(mptsas_t *mpt);
static void mptsas_restart_waitq(mptsas_t *mpt);

static void mptsas_deliver_doneq_thread(mptsas_t *mpt);
static void mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
static inline void mptsas_doneq_add0(mptsas_t *mpt, mptsas_cmd_t *cmd);
static void mptsas_doneq_mv(mptsas_t *mpt, uint64_t t);

static mptsas_cmd_t *mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t);
static void mptsas_doneq_empty(mptsas_t *mpt);
```



```
static void mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg);

static mptsas_cmd_t *mptsas_waitq_rm(mptsas_t *mpt);
static void mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);
static mptsas_cmd_t *mptsas_tx_waitq_rm(mptsas_t *mpt);
static void mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);


static void mptsas_start_watch_reset_delay();
static void mptsas_setup_bus_reset_delay(mptsas_t *mpt);
static void mptsas_watch_reset_delay(void *arg);
static int mptsas_watch_reset_delay_subr(mptsas_t *mpt);

static int mptsas_outstanding_cmds_n(mptsas_t *mpt);
/*
 * helper functions
 */
static void mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);

static dev_info_t *mptsas_find_child(dev_info_t *pdip, char *name);
static dev_info_t *mptsas_find_child_phy(dev_info_t *pdip, uint8_t phy);
static dev_info_t *mptsas_find_child_addr(dev_info_t *pdip, uint64_t sasaddr,
    int lun);
static mdi_pathinfo_t *mptsas_find_path_addr(dev_info_t *pdip, uint64_t sasaddr,
    int lun);
static mdi_pathinfo_t *mptsas_find_path_phy(dev_info_t *pdip, uint8_t phy);
static dev_info_t *mptsas_find_smp_child(dev_info_t *pdip, char *str_wwn);

static int mptsas_parse_address(char *name, uint64_t *wwid, uint8_t *phy,
    int *lun);
static int mptsas_parse_smp_name(char *name, uint64_t *wwn);

static mptsas_target_t *mptsas_phy_to_tgt(mptsas_t *mpt, int phymask,
    uint8_t phy);
static mptsas_target_t *mptsas_wwid_to_ptgt(mptsas_t *mpt, int phymask,
    uint64_t wwid);
static mptsas_smp_t *mptsas_wwid_to_psmp(mptsas_t *mpt, int phymask,
    uint64_t wwid);

static int mptsas_inquiry(mptsas_t *mpt, mptsas_target_t *ptgt, int lun,
    uchar_t page, unsigned char *buf, int len, int *rlen, uchar_t evpd);

static int mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
    uint16_t *handle, mptsas_target_t **pptgt);
static void mptsas_update_phymask(mptsas_t *mpt);
static inline void mptsas_remove_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd);

static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
    uint32_t *status, uint8_t cmd);
static dev_info_t *mptsas_get_dip_from_dev(dev_t dev,
    mptsas_phymask_t *phymask);
static mptsas_target_t *mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr,
    mptsas_phymask_t phymask);
static int mptsas_set_led_status(mptsas_t *mpt, mptsas_target_t *ptgt,
    uint32_t slotstatus);


/*
 * Enumeration / DR functions
 */
static void mptsas_config_all(dev_info_t *pdip);
static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
    dev_info_t **lundip);
static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
    dev_info_t **lundip);
```

```
364 static int mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt);
365 static int mptsas_offline_target(dev_info_t *pdip, char *name);

367 static int mptsas_config_raid(dev_info_t *pdip, uint16_t target,
368     dev_info_t **dip);

370 static int mptsas_config_luns(dev_info_t *pdip, mptsas_target_t *ptgt);
371 static int mptsas_probe_lun(dev_info_t *pdip, int lun,
372     dev_info_t **dip, mptsas_target_t *ptgt);

374 static int mptsas_create_lun(dev_info_t *pdip, struct scsi_inquiry *sd_inq,
375     dev_info_t **dip, mptsas_target_t *ptgt, int lun);

377 static int mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
378     char *guid, dev_info_t **dip, mptsas_target_t *ptgt, int lun);
379 static int mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
380     char *guid, dev_info_t **dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt,
381     int lun);

383 static void mptsas_offline_missed_luns(dev_info_t *pdip,
384     uint16_t *repluns, int lun_cnt, mptsas_target_t *ptgt);
385 static int mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,
386     mdi_pathinfo_t *rpip, uint_t flags);

388 static int mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn,
389     dev_info_t **smp_dip);
390 static int mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
391     uint_t flags);

393 static int mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data,
394     int mode, int *rval);
395 static int mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data,
396     int mode, int *rval);
397 static int mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data,
398     int mode, int *rval);
399 static void mptsas_record_event(void *args);
400 static int mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data,
401     int mode);

403 static void mptsas_hash_init(mptsas_hash_table_t *hashtab);
404 static void mptsas_hash_uninit(mptsas_hash_table_t *hashtab, size_t datalen);
405 static void mptsas_hash_add(mptsas_hash_table_t *hashtab, void *data);
406 static void * mptsas_hash_rem(mptsas_hash_table_t *hashtab, uint64_t key1,
407     mptsas_phymask_t key2);
408 static void * mptsas_hash_search(mptsas_hash_table_t *hashtab, uint64_t key1,
409     mptsas_phymask_t key2);
410 static void * mptsas_hash_traverse(mptsas_hash_table_t *hashtab, int pos);

412 mptsas_target_t *mptsas_tgt_alloc(mptsas_hash_table_t *, uint16_t, uint64_t,
413     uint32_t, mptsas_phymask_t, uint8_t);
434     uint32_t, mptsas_phymask_t, uint8_t, mptsas_t *);
414 static mptsas_smp_t *mptsas_smp_alloc(mptsas_hash_table_t *hashtab,
415     mptsas_smp_t *data);
416 static void mptsas_smp_free(mptsas_hash_table_t *hashtab, uint64_t wwid,
417     mptsas_phymask_t phymask);
418 static void mptsas_tgt_free(mptsas_hash_table_t *, uint64_t, mptsas_phymask_t);
419 static void * mptsas_search_by_devhdl(mptsas_hash_table_t *, uint16_t);
420 static int mptsas_online_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
421     dev_info_t **smp_dip);

423 /*
424  * Power management functions
425  */
426 static int mptsas_get_pci_cap(mptsas_t *mpt);
427 static int mptsas_init_pm(mptsas_t *mpt);
```



```
429 /*
430  * MPT MSI tunable:
431  *
432  * By default MSI is enabled on all supported platforms.
433  */
434 boolean_t mptsas_enable_msi = B_TRUE;
435 boolean_t mptsas_physical_bind_failed_page_83 = B_FALSE;

437 static int mptsas_register_intrs(mptsas_t *);
438 static void mptsas_unregister_intrs(mptsas_t *);
439 static int mptsas_add_intrs(mptsas_t *, int);
440 static void mptsas_rem_intrs(mptsas_t *);

442 /*
443  * FMA Prototypes
444  */
445 static void mptsas_fm_init(mptsas_t *mpt);
446 static void mptsas_fm_fini(mptsas_t *mpt);
447 static int mptsas_fm_error_cb(dev_info_t *, ddi_fm_error_t *, const void *);

449 extern pri_t minclsyspri, maxclsyspri;

451 /*
452  * This device is created by the SCSI pseudo nexus driver (SCSI vHCI).  It is
453  * under this device that the paths to a physical device are created when
454  * MPxIO is used.
455  */
456 extern dev_info_t       *scsi_vhci_dip;

458 /*
459  * Tunable timeout value for Inquiry VPD page 0x83
460  * By default the value is 30 seconds.
461  */
462 int mptsas_inq83_retry_timeout = 30;
463 /*
464  * Maximum number of command timeouts (0 - 255) considered acceptable.
465  */
466 int mptsas_timeout_threshold = 2;
467 /*
468  * Timeouts exceeding threshold within this period are considered excessive.
469  */
470 int mptsas_timeout_interval = 30;

472 /*
473  * This is used to allocate memory for message frame storage, not for
474  * data I/O DMA. All message frames must be stored in the first 4G of
475  * physical memory.
476  */
477 ddi_dma_attr_t mptsas_dma_attrs = {
478         DMA_ATTR_V0,    /* attribute layout version            */
479         0x0ull,         /* address low - should be 0 (longlong) */
480         0xffffffffull,  /* address high - 32-bit max range      */
481         0x00ffffffull,  /* count max - max DMA object size      */
482         4,              /* allocation alignment requirements    */
483         0x78,           /* burstsizes - binary encoded values   */
484         1,              /* minxfer - gran. of DMA engine        */
485         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
486         0xffffffffull,  /* max segment size (DMA boundary)      */
487         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
488         512,            /* granularity - device transfer size   */
489         0               /* flags, set to 0                      */
490 };
______unchanged_portion_omitted_

929 /*
930  * Notes:
```

```
 931  *      Set up all device state and allocate data structures,
 932  *      mutexes, condition variables, etc. for device operation.
 933  *      Add interrupts needed.
 934  *      Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
 935  */
 936 static int
 937 mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
 938 {
 939         mptsas_t                *mpt = NULL;
 940         int                     instance, i, j;
 941         int                     doneq_thread_num;
 942         char                    intr_added = 0;
 943         char                    map_setup = 0;
 944         char                    config_setup = 0;
 945         char                    hba_attach_setup = 0;
 946         char                    smp_attach_setup = 0;
 947         char                    mutex_init_done = 0;
 948         char                    event_taskq_create = 0;
 949         char                    dr_taskq_create = 0;
 950         char                    doneq_thread_create = 0;
 951         scsi_hba_tran_t         *hba_tran;
 952         uint_t                  mem_bar = MEM_SPACE;
 953         int                     rval = DDI_FAILURE;

 955         /* CONSTCOND */
 956         ASSERT(NO_COMPETING_THREADS);

 958         if (scsi_hba_iport_unit_address(dip)) {
 959                 return (mptsas_iport_attach(dip, cmd));
 960         }

 962         switch (cmd) {
 963         case DDI_ATTACH:
 964                 break;

 966         case DDI_RESUME:
 967                 if ((hba_tran = ddi_get_driver_private(dip)) == NULL)
 968                         return (DDI_FAILURE);

 970                 mpt = TRAN2MPT(hba_tran);

 972                 if (!mpt) {
 973                         return (DDI_FAILURE);
 974                 }

 976                 /*
 977                  * Reset hardware and softc to "no outstanding commands"
 978                  * Note that a check condition can result on first command
 979                  * to a target.
 980                  */
 981                 mutex_enter(&mpt->m_mutex);

 983                 /*
 984                  * raise power.
 985                  */
 986                 if (mpt->m_options & MPTSAS_OPT_PM) {
 987                         mutex_exit(&mpt->m_mutex);
 988                         (void) pm_busy_component(dip, 0);
 989                         rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
 990                         if (rval == DDI_SUCCESS) {
 991                                 mutex_enter(&mpt->m_mutex);
 992                         } else {
 993                                 /*
 994                                  * The pm_raise_power() call above failed,
 995                                  * and that can only occur if we were unable
 996                                  * to reset the hardware.  This is probably
```



```
 997                                  * due to unhealty hardware, and because
 998                                  * important filesystems(such as the root
 999                                  * filesystem) could be on the attached disks,
1000                                  * it would not be a good idea to continue,
1001                                  * as we won't be entirely certain we are
1002                                  * writing correct data.  So we panic() here
1003                                  * to not only prevent possible data corruption,
1004                                  * but to give developers or end users a hope
1005                                  * of identifying and correcting any problems.
1006                                  */
1007                                 fm_panic("mptsas could not reset hardware "
1008                                     "during resume");
1009                         }
1010                 }

1012                 mpt->m_suspended = 0;

1014                 /*
1015                  * Reinitialize ioc
1016                  */
1017                 mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1018                 if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
1019                         mutex_exit(&mpt->m_mutex);
1020                         if (mpt->m_options & MPTSAS_OPT_PM) {
1021                                 (void) pm_idle_component(dip, 0);
1022                         }
1023                         fm_panic("mptsas init chip fail during resume");
1024                 }
1025                 /*
1026                  * mptsas_update_driver_data needs interrupts so enable them
1027                  * first.
1028                  */
1029                 MPTSAS_ENABLE_INTR(mpt);
1030                 mptsas_update_driver_data(mpt);

1032                 /* start requests, if possible */
1033                 mptsas_restart_hba(mpt);

1035                 mutex_exit(&mpt->m_mutex);

1037                 /*
1038                  * Restart watch thread
1039                  */
1040                 mutex_enter(&mptsas_global_mutex);
1041                 if (mptsas_timeout_id == 0) {
1042                         mptsas_timeout_id = timeout(mptsas_watch, NULL,
1043                             mptsas_tick);
1044                         mptsas_timeouts_enabled = 1;
1045                 }
1046                 mutex_exit(&mptsas_global_mutex);

1048                 /* report idle status to pm framework */
1049                 if (mpt->m_options & MPTSAS_OPT_PM) {
1050                         (void) pm_idle_component(dip, 0);
1051                 }

1053                 return (DDI_SUCCESS);

1055         default:
1056                 return (DDI_FAILURE);

1058         }

1060         instance = ddi_get_instance(dip);

1062         /*
```

```
1063          * Allocate softc information.
1064          */
1065         if (ddi_soft_state_zalloc(mptsas_state, instance) != DDI_SUCCESS) {
1066                 mptsas_log(NULL, CE_WARN,
1067                     "mptsas%d: cannot allocate soft state", instance);
1068                 goto fail;
1069         }

1071         mpt = ddi_get_soft_state(mptsas_state, instance);

1073         if (mpt == NULL) {
1074                 mptsas_log(NULL, CE_WARN,
1075                     "mptsas%d: cannot get soft state", instance);
1076                 goto fail;
1077         }

1079         /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
1080         scsi_size_clean(dip);

1082         mpt->m_dip = dip;
1083         mpt->m_instance = instance;

1085         /* Make a per-instance copy of the structures */
1086         mpt->m_io_dma_attr = mptsas_dma_attrs64;
1087         mpt->m_msg_dma_attr = mptsas_dma_attrs;
1088         mpt->m_reg_acc_attr = mptsas_dev_attr;
1089         mpt->m_dev_acc_attr = mptsas_dev_attr;

1091         /*
1092          * Initialize FMA
1093          */
1094         mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
1095             DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
1096             DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
1097             DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);

1099         mptsas_fm_init(mpt);

1101         if (mptsas_alloc_handshake_msg(mpt,
1102             sizeof (Mpi2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
1103                 mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
1104                 goto fail;
1105         }

1107         /*
1108          * Setup configuration space
1109          */
1110         if (mptsas_config_space_init(mpt) == FALSE) {
1111                 mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
1112                 goto fail;
1113         }
1114         config_setup++;

1116         if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
1117             0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
1118                 mptsas_log(mpt, CE_WARN, "map setup failed");
1119                 goto fail;
1120         }
1121         map_setup++;

1123         /*
1124          * A taskq is created for dealing with the event handler
1125          */
1126         if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
1127             1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1128                 mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
```



```
1129                 goto fail;
1130         }
1131         event_taskq_create++;

1133         /*
1134          * A taskq is created for dealing with dr events
1135          */
1136         if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
1137             "mptsas_dr_taskq",
1138             1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1139                 mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
1140                     "failed");
1141                 goto fail;
1142         }
1143         dr_taskq_create++;

1145         mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1146             0, "mptsas_doneq_thread_threshold_prop", 10);
1147         mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1148             0, "mptsas_doneq_length_threshold_prop", 8);
1149         mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1150             0, "mptsas_doneq_thread_n_prop", 8);

1152         if (mpt->m_doneq_thread_n) {
1153                 cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
1154                 mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);

1156                 mutex_enter(&mpt->m_doneq_mutex);
1157                 mpt->m_doneq_thread_id =
1158                     kmem_zalloc(sizeof (mptsas_doneq_thread_list_t)
1159                     * mpt->m_doneq_thread_n, KM_SLEEP);

1161                 for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1162                         cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
1163                             CV_DRIVER, NULL);
1164                         mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
1165                             MUTEX_DRIVER, NULL);
1166                         mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1167                         mpt->m_doneq_thread_id[j].flag |=
1168                             MPTSAS_DONEQ_THREAD_ACTIVE;
1169                         mpt->m_doneq_thread_id[j].arg.mpt = mpt;
1170                         mpt->m_doneq_thread_id[j].arg.t = j;
1171                         mpt->m_doneq_thread_id[j].threadp =
1172                             thread_create(NULL, 0, mptsas_doneq_thread,
1173                             &mpt->m_doneq_thread_id[j].arg,
1174                             0, &p0, TS_RUN, minclsyspri);
1175                         mpt->m_doneq_thread_id[j].donetail =
1176                             &mpt->m_doneq_thread_id[j].doneq;
1177                         mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1178                 }
1179                 mutex_exit(&mpt->m_doneq_mutex);
1180                 doneq_thread_create++;
1181         }

1183         /* Initialize mutex used in interrupt handler */
1184         mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
1185             DDI_INTR_PRI(mpt->m_intr_pri));
1186         mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
1187         mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
1200         mutex_init(&mpt->m_intr_mutex, NULL, MUTEX_DRIVER,
1188             DDI_INTR_PRI(mpt->m_intr_pri));
1189         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1190                 mutex_init(&mpt->m_phy_info[i].smhba_info.phy_mutex,
1191                     NULL, MUTEX_DRIVER,
1192                     DDI_INTR_PRI(mpt->m_intr_pri));
1193         }
```

```
1195         cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
1196         cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
1197         cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
1198         cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
1199         cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
1200         mutex_init_done++;

1202         /*
1203          * Disable hardware interrupt since we're not ready to
1204          * handle it yet.
1205          */
1206         MPTSAS_DISABLE_INTR(mpt);
1207         if (mptsas_register_intrs(mpt) == FALSE)
1208                 goto fail;
1209         intr_added++;

1211         mutex_enter(&mpt->m_mutex);
1212         /*
1213          * Initialize power management component
1214          */
1215         if (mpt->m_options & MPTSAS_OPT_PM) {
1216                 if (mptsas_init_pm(mpt)) {
1217                         mutex_exit(&mpt->m_mutex);
1218                         mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
1219                             "failed");
1220                         goto fail;
1221                 }
1222         }

1224         /*
1225          * Initialize chip using Message Unit Reset, if allowed
1226          */
1227         mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1228         if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
1229                 mutex_exit(&mpt->m_mutex);
1230                 mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
1231                 goto fail;
1232         }

1234         /*
1235          * Fill in the phy_info structure and get the base WWID
1236          */
1237         if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
1238                 mptsas_log(mpt, CE_WARN,
1239                     "mptsas_get_manufacture_page5 failed!");
1240                 goto fail;
1241         }

1243         if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {
1244                 mptsas_log(mpt, CE_WARN,
1245                     "mptsas_get_sas_io_unit_page_hndshk failed!");
1246                 goto fail;
1247         }

1249         if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
1250                 mptsas_log(mpt, CE_WARN,
1251                     "mptsas_get_manufacture_page0 failed!");
1252                 goto fail;
1253         }

1255         mutex_exit(&mpt->m_mutex);

1257         /*
1258          * Register the iport for multiple port HBA
1259          */
```



```
1260         mptsas_iport_register(mpt);

1262         /*
1263          * initialize SCSI HBA transport structure
1264          */
1265         if (mptsas_hba_setup(mpt) == FALSE)
1266                 goto fail;
1267         hba_attach_setup++;

1269         if (mptsas_smp_setup(mpt) == FALSE)
1270                 goto fail;
1271         smp_attach_setup++;

1273         if (mptsas_cache_create(mpt) == FALSE)
1274                 goto fail;

1276         mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
1277             dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
1278         if (mpt->m_scsi_reset_delay == 0) {
1279                 mptsas_log(mpt, CE_NOTE,
1280                     "scsi_reset_delay of 0 is not recommended,"
1281                     " resetting to SCSI_DEFAULT_RESET_DELAY\n");
1282                 mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
1283         }

1285         /*
1286          * Initialize the wait and done FIFO queue
1287          */
1288         mpt->m_donetail = &mpt->m_doneq;
1289         mpt->m_waitqtail = &mpt->m_waitq;
1290         mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
1291         mpt->m_tx_draining = 0;

1293         /*
1294          * ioc cmd queue initialize
1295          */
1296         mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
1297         mpt->m_dev_handle = 0xFFFF;

1299         MPTSAS_ENABLE_INTR(mpt);

1301         /*
1302          * enable event notification
1303          */
1304         mutex_enter(&mpt->m_mutex);
1305         if (mptsas_ioc_enable_event_notification(mpt)) {
1306                 mutex_exit(&mpt->m_mutex);
1307                 goto fail;
1308         }
1309         mutex_exit(&mpt->m_mutex);

1311         /*
1312          * Initialize PHY info for smhba
1313          */
1314         if (mptsas_smhba_setup(mpt)) {
1315                 mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
1316                     "failed");
1317                 goto fail;
1318         }

1320         /* Check all dma handles allocated in attach */
1321         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
1322             != DDI_SUCCESS) ||
1323             (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
1324             != DDI_SUCCESS) ||
1325             (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
```

```
1326             != DDI_SUCCESS) ||
1327             (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
1328             != DDI_SUCCESS) ||
1329             (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
1330             != DDI_SUCCESS)) {
1331                 goto fail;
1332         }

1334         /* Check all acc handles allocated in attach */
1335         if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
1336             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
1337             != DDI_SUCCESS) ||
1338             (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
1339             != DDI_SUCCESS) ||
1340             (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
1341             != DDI_SUCCESS) ||
1342             (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
1343             != DDI_SUCCESS) ||
1344             (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
1345             != DDI_SUCCESS) ||
1346             (mptsas_check_acc_handle(mpt->m_config_handle)
1347             != DDI_SUCCESS)) {
1348                 goto fail;
1349         }

1351         /*
1352          * After this point, we are not going to fail the attach.
1353          */
1354         /*
1355          * used for mptsas_watch
1356          */
1357         mptsas_list_add(mpt);

1359         mutex_enter(&mptsas_global_mutex);
1360         if (mptsas_timeouts_enabled == 0) {
1361                 mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
1362                     dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);

1364                 mptsas_tick = mptsas_scsi_watchdog_tick *
1365                     drv_usectohz((clock_t)1000000);

1367                 mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
1368                 mptsas_timeouts_enabled = 1;
1369         }
1370         mutex_exit(&mptsas_global_mutex);

1372         /* Print message of HBA present */
1373         ddi_report_dev(dip);

1375         /* report idle status to pm framework */
1376         if (mpt->m_options & MPTSAS_OPT_PM) {
1377                 (void) pm_idle_component(dip, 0);
1378         }

1380         return (DDI_SUCCESS);

1382 fail:
1383         mptsas_log(mpt, CE_WARN, "attach failed");
1384         mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
1385         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
1386         if (mpt) {
1387                 mutex_enter(&mptsas_global_mutex);

1389                 if (mptsas_timeout_id && (mptsas_head == NULL)) {
1390                         timeout_id_t tid = mptsas_timeout_id;
1391                         mptsas_timeouts_enabled = 0;
```



```
1392                         mptsas_timeout_id = 0;
1393                         mutex_exit(&mptsas_global_mutex);
1394                         (void) untimeout(tid);
1395                         mutex_enter(&mptsas_global_mutex);
1396                 }
1397                 mutex_exit(&mptsas_global_mutex);
1398                 /* deallocate in reverse order */
1399                 mptsas_cache_destroy(mpt);

1401                 if (smp_attach_setup) {
1402                         mptsas_smp_teardown(mpt);
1403                 }
1404                 if (hba_attach_setup) {
1405                         mptsas_hba_teardown(mpt);
1406                 }

1408                 if (mpt->m_active) {
1409                         mptsas_hash_uninit(&mpt->m_active->m_smptbl,
1410                             sizeof (mptsas_smp_t));
1411                         mptsas_hash_uninit(&mpt->m_active->m_tgttbl,
1412                             sizeof (mptsas_target_t));
1413                         mptsas_free_active_slots(mpt);
1414                 }
1415                 if (intr_added) {
1416                         mptsas_unregister_intrs(mpt);
1417                 }

1419                 if (doneq_thread_create) {
1420                         mutex_enter(&mpt->m_doneq_mutex);
1421                         doneq_thread_num = mpt->m_doneq_thread_n;
1422                         for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1423                                 mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1424                                 mpt->m_doneq_thread_id[j].flag &=
1425                                     (~MPTSAS_DONEQ_THREAD_ACTIVE);
1426                                 cv_signal(&mpt->m_doneq_thread_id[j].cv);
1427                                 mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1428                         }
1429                         while (mpt->m_doneq_thread_n) {
1430                                 cv_wait(&mpt->m_doneq_thread_cv,
1431                                     &mpt->m_doneq_mutex);
1432                         }
1433                         for (j = 0; j < doneq_thread_num; j++) {
1434                                 cv_destroy(&mpt->m_doneq_thread_id[j].cv);
1435                                 mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
1436                         }
1437                         kmem_free(mpt->m_doneq_thread_id,
1438                             sizeof (mptsas_doneq_thread_list_t)
1439                             * doneq_thread_num);
1440                         mutex_exit(&mpt->m_doneq_mutex);
1441                         cv_destroy(&mpt->m_doneq_thread_cv);
1442                         mutex_destroy(&mpt->m_doneq_mutex);
1443                 }
1444                 if (event_taskq_create) {
1445                         ddi_taskq_destroy(mpt->m_event_taskq);
1446                 }
1447                 if (dr_taskq_create) {
1448                         ddi_taskq_destroy(mpt->m_dr_taskq);
1449                 }
1450                 if (mutex_init_done) {
1451                         mutex_destroy(&mpt->m_tx_waitq_mutex);
1462                         mutex_destroy(&mpt->m_intr_mutex);
1452                         mutex_destroy(&mpt->m_passthru_mutex);
1453                         mutex_destroy(&mpt->m_mutex);
1454                         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1455                                 mutex_destroy(
1456                                     &mpt->m_phy_info[i].smhba_info.phy_mutex);
```

```
1457                        }
1458                        cv_destroy(&mpt->m_cv);
1459                        cv_destroy(&mpt->m_passthru_cv);
1460                        cv_destroy(&mpt->m_fw_cv);
1461                        cv_destroy(&mpt->m_config_cv);
1462                        cv_destroy(&mpt->m_fw_diag_cv);
1463                }

1465                if (map_setup) {
1466                        mptsas_cfg_fini(mpt);
1467                }
1468                if (config_setup) {
1469                        mptsas_config_space_fini(mpt);
1470                }
1471                mptsas_free_handshake_msg(mpt);
1472                mptsas_hba_fini(mpt);

1474                mptsas_fm_fini(mpt);
1475                ddi_soft_state_free(mptsas_state, instance);
1476                ddi_prop_remove_all(dip);
1477        }
1478        return (DDI_FAILURE);
1479 }
```

unchanged_portion_omitted_

```
1680 static int
1681 mptsas_do_detach(dev_info_t *dip)
1682 {
1683        mptsas_t        *mpt;
1684        scsi_hba_tran_t *tran;
1685        int             circ = 0;
1686        int             circ1 = 0;
1687        mdi_pathinfo_t  *pip = NULL;
1688        int             i;
1689        int             doneq_thread_num = 0;

1691        NDBG0(("mptsas_do_detach: dip=0x%p", (void *)dip));

1693        if ((tran = ndi_flavorv_get(dip, SCSA_FLAVOR_SCSI_DEVICE)) == NULL)
1694                return (DDI_FAILURE);

1696        mpt = TRAN2MPT(tran);
1697        if (!mpt) {
1698                return (DDI_FAILURE);
1699        }
1700        /*
1701         * Still have pathinfo child, should not detach mpt driver
1702         */
1703        if (scsi_hba_iport_unit_address(dip)) {
1704                if (mpt->m_mpxio_enable) {
1705                        /*
1706                         * MPxIO enabled for the iport
1707                         */
1708                        ndi_devi_enter(scsi_vhci_dip, &circ1);
1709                        ndi_devi_enter(dip, &circ);
1710                        while (pip = mdi_get_next_client_path(dip, NULL)) {
1711                                if (mdi_pi_free(pip, 0) == MDI_SUCCESS) {
1712                                        continue;
1713                                }
1714                                ndi_devi_exit(dip, circ);
1715                                ndi_devi_exit(scsi_vhci_dip, circ1);
1716                                NDBG12(("detach failed because of "
1717                                    "outstanding path info"));
1718                                return (DDI_FAILURE);
1719                        }
1720                        ndi_devi_exit(dip, circ);
```



```
1721                        ndi_devi_exit(scsi_vhci_dip, circ1);
1722                        (void) mdi_phci_unregister(dip, 0);
1723                }

1725                ddi_prop_remove_all(dip);

1727                return (DDI_SUCCESS);
1728        }

1730        /* Make sure power level is D0 before accessing registers */
1731        if (mpt->m_options & MPTSAS_OPT_PM) {
1732                (void) pm_busy_component(dip, 0);
1733                if (mpt->m_power_level != PM_LEVEL_D0) {
1734                        if (pm_raise_power(dip, 0, PM_LEVEL_D0) !=
1735                            DDI_SUCCESS) {
1736                                mptsas_log(mpt, CE_WARN,
1737                                    "mptsas%d: Raise power request failed.",
1738                                    mpt->m_instance);
1739                                (void) pm_idle_component(dip, 0);
1740                                return (DDI_FAILURE);
1741                        }
1742                }
1743        }

1745        /*
1746         * Send RAID action system shutdown to sync IR.  After action, send a
1747         * Message Unit Reset. Since after that DMA resource will be freed,
1748         * set ioc to READY state will avoid HBA initiated DMA operation.
1749         */
1750        mutex_enter(&mpt->m_mutex);
1751        MPTSAS_DISABLE_INTR(mpt);
1752        mptsas_raid_action_system_shutdown(mpt);
1753        mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1754        (void) mptsas_ioc_reset(mpt, FALSE);
1755        mutex_exit(&mpt->m_mutex);
1756        mptsas_rem_intrs(mpt);
1757        ddi_taskq_destroy(mpt->m_event_taskq);
1758        ddi_taskq_destroy(mpt->m_dr_taskq);

1760        if (mpt->m_doneq_thread_n) {
1761                mutex_enter(&mpt->m_doneq_mutex);
1762                doneq_thread_num = mpt->m_doneq_thread_n;
1763                for (i = 0; i < mpt->m_doneq_thread_n; i++) {
1764                        mutex_enter(&mpt->m_doneq_thread_id[i].mutex);
1765                        mpt->m_doneq_thread_id[i].flag &=
1766                            (~MPTSAS_DONEQ_THREAD_ACTIVE);
1767                        cv_signal(&mpt->m_doneq_thread_id[i].cv);
1768                        mutex_exit(&mpt->m_doneq_thread_id[i].mutex);
1769                }
1770                while (mpt->m_doneq_thread_n) {
1771                        cv_wait(&mpt->m_doneq_thread_cv,
1772                            &mpt->m_doneq_mutex);
1773                }
1774                for (i = 0;  i < doneq_thread_num; i++) {
1775                        cv_destroy(&mpt->m_doneq_thread_id[i].cv);
1776                        mutex_destroy(&mpt->m_doneq_thread_id[i].mutex);
1777                }
1778                kmem_free(mpt->m_doneq_thread_id,
1779                    sizeof (mptsas_doneq_thread_list_t)
1780                    * doneq_thread_num);
1781                mutex_exit(&mpt->m_doneq_mutex);
1782                cv_destroy(&mpt->m_doneq_thread_cv);
1783                mutex_destroy(&mpt->m_doneq_mutex);
1784        }

1786        scsi_hba_reset_notify_tear_down(mpt->m_reset_notify_listf);
```

```
1788         mptsas_list_del(mpt);

1790         /*
1791          * Cancel timeout threads for this mpt
1792          */
1793         mutex_enter(&mpt->m_mutex);
1794         if (mpt->m_quiesce_timeid) {
1795                 timeout_id_t tid = mpt->m_quiesce_timeid;
1796                 mpt->m_quiesce_timeid = 0;
1797                 mutex_exit(&mpt->m_mutex);
1798                 (void) untimeout(tid);
1799                 mutex_enter(&mpt->m_mutex);
1800         }

1802         if (mpt->m_restart_cmd_timeid) {
1803                 timeout_id_t tid = mpt->m_restart_cmd_timeid;
1804                 mpt->m_restart_cmd_timeid = 0;
1805                 mutex_exit(&mpt->m_mutex);
1806                 (void) untimeout(tid);
1807                 mutex_enter(&mpt->m_mutex);
1808         }

1810         mutex_exit(&mpt->m_mutex);

1812         /*
1813          * last mpt? ... if active, CANCEL watch threads.
1814          */
1815         mutex_enter(&mptsas_global_mutex);
1816         if (mptsas_head == NULL) {
1817                 timeout_id_t tid;
1818                 /*
1819                  * Clear mptsas_timeouts_enable so that the watch thread
1820                  * gets restarted on DDI_ATTACH
1821                  */
1822                 mptsas_timeouts_enabled = 0;
1823                 if (mptsas_timeout_id) {
1824                         tid = mptsas_timeout_id;
1825                         mptsas_timeout_id = 0;
1826                         mutex_exit(&mptsas_global_mutex);
1827                         (void) untimeout(tid);
1828                         mutex_enter(&mptsas_global_mutex);
1829                 }
1830                 if (mptsas_reset_watch) {
1831                         tid = mptsas_reset_watch;
1832                         mptsas_reset_watch = 0;
1833                         mutex_exit(&mptsas_global_mutex);
1834                         (void) untimeout(tid);
1835                         mutex_enter(&mptsas_global_mutex);
1836                 }
1837         }
1838         mutex_exit(&mptsas_global_mutex);

1840         /*
1841          * Delete Phy stats
1842          */
1843         mptsas_destroy_phy_stats(mpt);

1845         /*
1846          * Delete nt_active.
1847          */
1848         mutex_enter(&mpt->m_mutex);
1849         mptsas_hash_uninit(&mpt->m_active->m_tgttbl, sizeof (mptsas_target_t));
1850         mptsas_hash_uninit(&mpt->m_active->m_smptbl, sizeof (mptsas_smp_t));
1851         mptsas_free_active_slots(mpt);
1852         mutex_exit(&mpt->m_mutex);
```



```
1854         /* deallocate everything that was allocated in mptsas_attach */
1855         mptsas_cache_destroy(mpt);

1857         mptsas_hba_fini(mpt);
1858         mptsas_cfg_fini(mpt);

1860         /* Lower the power informing PM Framework */
1861         if (mpt->m_options & MPTSAS_OPT_PM) {
1862                 if (pm_lower_power(dip, 0, PM_LEVEL_D3) != DDI_SUCCESS)
1863                         mptsas_log(mpt, CE_WARN,
1864                             "!mptsas%d: Lower power request failed "
1865                             "during detach, ignoring.",
1866                             mpt->m_instance);
1867         }

1869         mutex_destroy(&mpt->m_tx_waitq_mutex);
1880         mutex_destroy(&mpt->m_intr_mutex);
1870         mutex_destroy(&mpt->m_passthru_mutex);
1871         mutex_destroy(&mpt->m_mutex);
1872         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1873                 mutex_destroy(&mpt->m_phy_info[i].smhba_info.phy_mutex);
1874         }
1875         cv_destroy(&mpt->m_cv);
1876         cv_destroy(&mpt->m_passthru_cv);
1877         cv_destroy(&mpt->m_fw_cv);
1878         cv_destroy(&mpt->m_config_cv);
1879         cv_destroy(&mpt->m_fw_diag_cv);


1882         mptsas_smp_teardown(mpt);
1883         mptsas_hba_teardown(mpt);

1885         mptsas_config_space_fini(mpt);

1887         mptsas_free_handshake_msg(mpt);

1889         mptsas_fm_fini(mpt);
1890         ddi_soft_state_free(mptsas_state, ddi_get_instance(dip));
1891         ddi_prop_remove_all(dip);

1893         return (DDI_SUCCESS);
1894 }
_____unchanged_portion_omitted_

2177 static int
2178 mptsas_power(dev_info_t *dip, int component, int level)
2179 {
2180 #ifndef __lock_lint
2181         _NOTE(ARGUNUSED(component))
2182 #endif
2183         mptsas_t        *mpt;
2184         int             rval = DDI_SUCCESS;
2185         int             polls = 0;
2186         uint32_t        ioc_status;

2188         if (scsi_hba_iport_unit_address(dip) != 0)
2189                 return (DDI_SUCCESS);

2191         mpt = ddi_get_soft_state(mptsas_state, ddi_get_instance(dip));
2192         if (mpt == NULL) {
2193                 return (DDI_FAILURE);
2194         }

2196         mutex_enter(&mpt->m_mutex);
```

```
2198         /*
2199          * If the device is busy, don't lower its power level
2200          */
2201         if (mpt->m_busy && (mpt->m_power_level > level)) {
2202                 mutex_exit(&mpt->m_mutex);
2203                 return (DDI_FAILURE);
2204         }
2205         switch (level) {
2206         case PM_LEVEL_D0:
2207                 NDBG11(("mptsas%d: turning power ON.", mpt->m_instance));
2208                 MPTSAS_POWER_ON(mpt);
2209                 /*
2210                  * Wait up to 30 seconds for IOC to come out of reset.
2211                  */
2212                 while (((ioc_status = ddi_get32(mpt->m_datap,
2213                     &mpt->m_reg->Doorbell)) &
2214                     MPI2_IOC_STATE_MASK) == MPI2_IOC_STATE_RESET) {
2215                         if (polls++ > 3000) {
2216                                 break;
2217                         }
2218                         delay(drv_usectohz(10000));
2219                 }
2220                 /*
2221                  * If IOC is not in operational state, try to hard reset it.
2222                  */
2223                 if ((ioc_status & MPI2_IOC_STATE_MASK) !=
2224                     MPI2_IOC_STATE_OPERATIONAL) {
2225                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
2226                         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
2227                                 mptsas_log(mpt, CE_WARN,
2228                                     "mptsas_power: hard reset failed");
2229                                 mutex_exit(&mpt->m_mutex);
2230                                 return (DDI_FAILURE);
2231                         }
2232                 }
2244                 mutex_enter(&mpt->m_intr_mutex);
2233                 mpt->m_power_level = PM_LEVEL_D0;
2246                 mutex_exit(&mpt->m_intr_mutex);
2234                 break;
2235         case PM_LEVEL_D3:
2236                 NDBG11(("mptsas%d: turning power OFF.", mpt->m_instance));
2237                 MPTSAS_POWER_OFF(mpt);
2238                 break;
2239         default:
2240                 mptsas_log(mpt, CE_WARN, "mptsas%d: unknown power level <%x>.",
2241                     mpt->m_instance, level);
2242                 rval = DDI_FAILURE;
2243                 break;
2244         }
2245         mutex_exit(&mpt->m_mutex);
2246         return (rval);
2247 }
```

*_____unchanged_portion_omitted_*

```
2621 static void
2622 mptsas_alloc_reply_args(mptsas_t *mpt)
2623 {
2624         if (mpt->m_replyh_args == NULL) {
2637         if (mpt->m_replyh_args != NULL) {
2638                 kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
2639                     * mpt->m_max_replies);
2640                 mpt->m_replyh_args = NULL;
2641         }
2625                 mpt->m_replyh_args = kmem_zalloc(sizeof (m_replyh_arg_t) *
2626                     mpt->m_max_replies, KM_SLEEP);
2627         }
```



```
2628 }
```

*_____unchanged_portion_omitted_*

```
2933 /*
2934  * scsi_pkt handling
2935  *
2936  * Visible to the external world via the transport structure.
2937  */

2939 /*
2940  * Notes:
2941  *      - transport the command to the addressed SCSI target/lun device
2942  *      - normal operation is to schedule the command to be transported,
2943  *        and return TRAN_ACCEPT if this is successful.
2944  *      - if NO_INTR, tran_start must poll device for command completion
2945  */
2946 static int
2947 mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt)
2948 {
2949 #ifndef __lock_lint
2950         _NOTE(ARGUNUSED(ap))
2951 #endif
2952         mptsas_t        *mpt = PKT2MPT(pkt);
2953         mptsas_cmd_t    *cmd = PKT2CMD(pkt);
2954         int             rval;
2955         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

2957         NDBG1(("mptsas_scsi_start: pkt=0x%p", (void *)pkt));
2958         ASSERT(ptgt);
2959         if (ptgt == NULL)
2960                 return (TRAN_FATAL_ERROR);

2962         /*
2963          * prepare the pkt before taking mutex.
2964          */
2965         rval = mptsas_prepare_pkt(cmd);
2966         if (rval != TRAN_ACCEPT) {
2967                 return (rval);
2968         }

2970         /*
2971          * Send the command to target/lun, however your HBA requires it.
2972          * If busy, return TRAN_BUSY; if there's some other formatting error
2973          * in the packet, return TRAN_BADPKT; otherwise, fall through to the
2974          * return of TRAN_ACCEPT.
2975          *
2976          * Remember that access to shared resources, including the mptsas_t
2977          * data structure and the HBA hardware registers, must be protected
2978          * with mutexes, here and everywhere.
2979          *
2980          * Also remember that at interrupt time, you'll get an argument
2981          * to the interrupt handler which is a pointer to your mptsas_t
2982          * structure; you'll have to remember which commands are outstanding
2983          * and which scsi_pkt is the currently-running command so the
2984          * interrupt handler can refer to the pkt to set completion
2985          * status, call the target driver back through pkt_comp, etc.
2986          *
2987          * If the instance lock is held by other thread, don't spin to wait
2988          * for it. Instead, queue the cmd and next time when the instance lock
2989          * is not held, accept all the queued cmd. A extra tx_waitq is
2990          * introduced to protect the queue.
2991          *
2992          * The polled cmd will not be queud and accepted as usual.
2993          *
2994          * Under the tx_waitq mutex, record whether a thread is draining
2995          * the tx_waitq.  An IO requesting thread that finds the instance
```

```
2996          * mutex contended appends to the tx_waitq and while holding the
2997          * tx_wait mutex, if the draining flag is not set, sets it and then
2998          * proceeds to spin for the instance mutex. This scheme ensures that
2999          * the last cmd in a burst be processed.
3000          *
3001          * we enable this feature only when the helper threads are enabled,
3002          * at which we think the loads are heavy.
3003          *
3004          * per instance mutex m_tx_waitq_mutex is introduced to protect the
3005          * m_tx_waitqtail, m_tx_waitq, m_tx_draining.
3006          */

3008         if (mpt->m_doneq_thread_n) {
3009                 if (mutex_tryenter(&mpt->m_mutex) != 0) {
3010                         rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3011                         mutex_exit(&mpt->m_mutex);
3012                 } else if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3013                         mutex_enter(&mpt->m_mutex);
3014                         rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3015                         mutex_exit(&mpt->m_mutex);
3016                 } else {
3017                         mutex_enter(&mpt->m_tx_waitq_mutex);
3018                         /*
3019                          * ptgt->m_dr_flag is protected by m_mutex or
3020                          * m_tx_waitq_mutex. In this case, m_tx_waitq_mutex
3021                          * is acquired.
3022                          */
3004         mutex_enter(&ptgt->m_tgt_intr_mutex);
3023                         if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3024                                 if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3025                                         /*
3026                                          * The command should be allowed to
3027                                          * retry by returning TRAN_BUSY to
3028                                          * to stall the I/O's which come from
3029                                          * scsi_vhci since the device/path is
3030                                          * in unstable state now.
3031                                          */
3032                                         mutex_exit(&mpt->m_tx_waitq_mutex);
3033                                         return (TRAN_BUSY);
3034                                 } else {
3035                                         /*
3036                                          * The device is offline, just fail the
3037                                          * command by returning
3038                                          * TRAN_FATAL_ERROR.
3039                                          */
3040                                         mutex_exit(&mpt->m_tx_waitq_mutex);
3041                                         return (TRAN_FATAL_ERROR);
3042                                 }
3043                         }
3044                         if (mpt->m_tx_draining) {
3045                                 cmd->cmd_flags |= CFLAG_TXQ;
3046                                 *mpt->m_tx_waitqtail = cmd;
3047                                 mpt->m_tx_waitqtail = &cmd->cmd_linkp;
3048                                 mutex_exit(&mpt->m_tx_waitq_mutex);
3049                         } else { /* drain the queue */
3050                                 mpt->m_tx_draining = 1;
3051                                 mutex_exit(&mpt->m_tx_waitq_mutex);
3052                                 mutex_enter(&mpt->m_mutex);
3053                                 rval = mptsas_accept_txwq_and_pkt(mpt, cmd);
3054                                 mutex_exit(&mpt->m_mutex);
3055                         }
3056                 }
3057         } else {
3058                 mutex_enter(&mpt->m_mutex);
3059                 /*
3060                  * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
```



```
3061                  * in this case, m_mutex is acquired.
3062                  */
3063                 if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3064                         if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3065                                 /*
3066                                  * commands should be allowed to retry by
3067                                  * returning TRAN_BUSY to stall the I/O's
3068                                  * which come from scsi_vhci since the device/
3069                                  * path is in unstable state now.
3070                                  */
3071                                 mutex_exit(&mpt->m_mutex);
3013                         mutex_exit(&ptgt->m_tgt_intr_mutex);
3072                                 return (TRAN_BUSY);
3073                         } else {
3074                                 /*
3075                                  * The device is offline, just fail the
3076                                  * command by returning TRAN_FATAL_ERROR.
3077                                  */
3078                                 mutex_exit(&mpt->m_mutex);
3020                         mutex_exit(&ptgt->m_tgt_intr_mutex);
3079                                 return (TRAN_FATAL_ERROR);
3080                         }
3081                 }
3024         mutex_exit(&ptgt->m_tgt_intr_mutex);
3082                 rval = mptsas_accept_pkt(mpt, cmd);
3083                 mutex_exit(&mpt->m_mutex);
3084         }

3086         return (rval);
3087 }

3089 /*
3090  * Accept all the queued cmds(if any) before accept the current one.
3091  */
3092 static int
3093 mptsas_accept_txwq_and_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3094 {
3095         int rval;
3096         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

3098         ASSERT(mutex_owned(&mpt->m_mutex));
3099         /*
3100          * The call to mptsas_accept_tx_waitq() must always be performed
3101          * because that is where mpt->m_tx_draining is cleared.
3102          */
3103         mutex_enter(&mpt->m_tx_waitq_mutex);
3104         mptsas_accept_tx_waitq(mpt);
3105         mutex_exit(&mpt->m_tx_waitq_mutex);
3106         /*
3107          * ptgt->m_dr_flag is protected by m_mutex or m_tx_waitq_mutex
3108          * in this case, m_mutex is acquired.
3109          */
3110         if (ptgt->m_dr_flag == MPTSAS_DR_INTRANSITION) {
3111                 if (cmd->cmd_pkt_flags & FLAG_NOQUEUE) {
3112                         /*
3113                          * The command should be allowed to retry by returning
3114                          * TRAN_BUSY to stall the I/O's which come from
3115                          * scsi_vhci since the device/path is in unstable state
3116                          * now.
3117                          */
3118                         return (TRAN_BUSY);
3119                 } else {
3120                         /*
3121                          * The device is offline, just fail the command by
3122                          * return TRAN_FATAL_ERROR.
3123                          */
```

```
3124                        return (TRAN_FATAL_ERROR);
3125                }
3126        }
3127        rval = mptsas_accept_pkt(mpt, cmd);

3129        return (rval);
3130 }

3132 static int
3133 mptsas_accept_pkt(mptsas_t *mpt, mptsas_cmd_t *cmd)
3134 {
3135        int             rval = TRAN_ACCEPT;
3136        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

3138        NDBG1(("mptsas_accept_pkt: cmd=0x%p", (void *)cmd));

3140        ASSERT(mutex_owned(&mpt->m_mutex));

3142        if ((cmd->cmd_flags & CFLAG_PREPARED) == 0) {
3143                rval = mptsas_prepare_pkt(cmd);
3144                if (rval != TRAN_ACCEPT) {
3145                        cmd->cmd_flags &= ~CFLAG_TRANFLAG;
3146                        return (rval);
3147                }
3148        }

3150        /*
3151         * reset the throttle if we were draining
3152         */
3049        mutex_enter(&ptgt->m_tgt_intr_mutex);
3153        if ((ptgt->m_t_ncmds == 0) &&
3154            (ptgt->m_t_throttle == DRAIN_THROTTLE)) {
3155                NDBG23(("reset throttle"));
3156                ASSERT(ptgt->m_reset_delay == 0);
3157                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
3158        }

3160        /*
3161         * If HBA is being reset, the DevHandles are being re-initialized,
3162         * which means that they could be invalid even if the target is still
3163         * attached.  Check if being reset and if DevHandle is being
3164         * re-initialized.  If this is the case, return BUSY so the I/O can be
3165         * retried later.
3166         */
3167        if ((ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) && mpt->m_in_reset) {
3168                mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
3169                if (cmd->cmd_flags & CFLAG_TXQ) {
3170                        mptsas_doneq_add(mpt, cmd);
3171                        mptsas_doneq_empty(mpt);
3172                        return (rval);
3173                } else {
3174                        return (TRAN_BUSY);
3175                }
3176        }

3178        /*
3179         * If device handle has already been invalidated, just
3180         * fail the command. In theory, command from scsi_vhci
3181         * client is impossible send down command with invalid
3182         * devhdl since devhdl is set after path offline, target
3183         * driver is not suppose to select a offlined path.
3184         */
3185        if (ptgt->m_devhdl == MPTSAS_INVALID_DEVHDL) {
3186                NDBG20(("rejecting command, it might because invalid devhdl "
3187                    "request."));
3067                mutex_exit(&ptgt->m_tgt_intr_mutex);
```



```
3068                mutex_enter(&mpt->m_mutex);
3069                /*
3070                 * If HBA is being reset, the DevHandles are being
3071                 * re-initialized, which means that they could be invalid
3072                 * even if the target is still attached. Check if being reset
3073                 * and if DevHandle is being re-initialized. If this is the
3074                 * case, return BUSY so the I/O can be retried later.
3075                 */
3076                if (mpt->m_in_reset) {
3077                        mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
3078                            STAT_BUS_RESET);
3079                        if (cmd->cmd_flags & CFLAG_TXQ) {
3080                                mptsas_doneq_add(mpt, cmd);
3081                                mptsas_doneq_empty(mpt);
3082                                mutex_exit(&mpt->m_mutex);
3083                                return (rval);
3084                        } else {
3085                                mutex_exit(&mpt->m_mutex);
3086                                return (TRAN_BUSY);
3087                        }
3088                }
3188                mptsas_set_pkt_reason(mpt, cmd, CMD_DEV_GONE, STAT_TERMINATED);
3189                if (cmd->cmd_flags & CFLAG_TXQ) {
3190                        mptsas_doneq_add(mpt, cmd);
3191                        mptsas_doneq_empty(mpt);
3093                        mutex_exit(&mpt->m_mutex);
3192                        return (rval);
3193                } else {
3096                        mutex_exit(&mpt->m_mutex);
3194                        return (TRAN_FATAL_ERROR);
3195                }
3196        }
3100        mutex_exit(&ptgt->m_tgt_intr_mutex);
3197        /*
3198         * The first case is the normal case.  mpt gets a command from the
3199         * target driver and starts it.
3200         * Since SMID 0 is reserved and the TM slot is reserved, the actual max
3201         * commands is m_max_requests - 2.
3202         */
3203        if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
3204            (ptgt->m_t_throttle > HOLD_THROTTLE) &&
3107        mutex_enter(&ptgt->m_tgt_intr_mutex);
3108        if ((ptgt->m_t_throttle > HOLD_THROTTLE) &&
3205            (ptgt->m_t_ncmds < ptgt->m_t_throttle) &&
3206            (ptgt->m_reset_delay == 0) &&
3207            (ptgt->m_t_nwait == 0) &&
3208            ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0)) {
3113                mutex_exit(&ptgt->m_tgt_intr_mutex);
3209                if (mptsas_save_cmd(mpt, cmd) == TRUE) {
3210                        (void) mptsas_start_cmd(mpt, cmd);
3115                        (void) mptsas_start_cmd0(mpt, cmd);
3211                } else {
3117                        mutex_enter(&mpt->m_mutex);
3212                        mptsas_waitq_add(mpt, cmd);
3119                        mutex_exit(&mpt->m_mutex);
3213                }
3214        } else {
3215                /*
3216                 * Add this pkt to the work queue
3217                 */
3125                mutex_exit(&ptgt->m_tgt_intr_mutex);
3126                mutex_enter(&mpt->m_mutex);
3218                mptsas_waitq_add(mpt, cmd);

3220                if (cmd->cmd_pkt_flags & FLAG_NOINTR) {
3221                        (void) mptsas_poll(mpt, cmd, MPTSAS_POLL_TIME);
```

```
3223                        /*
3224                         * Only flush the doneq if this is not a TM
3225                         * cmd.  For TM cmds the flushing of the
3226                         * doneq will be done in those routines.
3227                         */
3228                        if ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
3229                                mptsas_doneq_empty(mpt);
3230                        }
3231                }
3141                mutex_exit(&mpt->m_mutex);
3232        }
3233        return (rval);
3234 }

3236 int
3237 mptsas_save_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
3238 {
3239        mptsas_slots_t  *slots;
3240        int             slot;
3241        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
3152        mptsas_slot_free_e_t    *pe;
3153        int             qn, qn_first;

3243        ASSERT(mutex_owned(&mpt->m_mutex));
3244        slots = mpt->m_active;

3246        /*
3247         * Account for reserved TM request slot and reserved SMID of 0.
3248         */
3249        ASSERT(slots->m_n_slots == (mpt->m_max_requests - 2));

3251        /*
3252         * m_tags is equivalent to the SMID when sending requests.  Since the
3253         * SMID cannot be 0, start out at one if rolling over past the size
3254         * of the request queue depth.  Also, don't use the last SMID, which is
3255         * reserved for TM requests.
3256         */
3257        slot = (slots->m_tags)++;
3258        if (slots->m_tags > slots->m_n_slots) {
3259                slots->m_tags = 1;
3260        }
3162        qn = qn_first = CPU->cpu_seqid & (mpt->m_slot_freeq_pair_n - 1);

3262 alloc_tag:
3263        /* Validate tag, should never fail. */
3264        if (slots->m_slot[slot] == NULL) {
3164 qpair_retry:
3165        ASSERT(qn < mpt->m_slot_freeq_pair_n);
3166        mutex_enter(&mpt->m_slot_freeq_pairp[qn].m_slot_allocq.s.m_fq_mutex);
3167        pe = list_head(&mpt->m_slot_freeq_pairp[qn].m_slot_allocq.
3168            s.m_fq_list);
3169        if (!pe) { /* switch the allocq and releq */
3170                mutex_enter(&mpt->m_slot_freeq_pairp[qn].m_slot_releq.
3171                    s.m_fq_mutex);
3172                if (mpt->m_slot_freeq_pairp[qn].m_slot_releq.s.m_fq_n) {
3173                        mpt->m_slot_freeq_pairp[qn].
3174                            m_slot_allocq.s.m_fq_n =
3175                            mpt->m_slot_freeq_pairp[qn].
3176                            m_slot_releq.s.m_fq_n;
3177                        mpt->m_slot_freeq_pairp[qn].
3178                            m_slot_allocq.s.m_fq_list.list_head.list_next =
3179                            mpt->m_slot_freeq_pairp[qn].
3180                            m_slot_releq.s.m_fq_list.list_head.list_next;
3181                        mpt->m_slot_freeq_pairp[qn].
3182                            m_slot_allocq.s.m_fq_list.list_head.list_prev =
```



```
3183                            mpt->m_slot_freeq_pairp[qn].
3184                            m_slot_releq.s.m_fq_list.list_head.list_prev;
3185                        mpt->m_slot_freeq_pairp[qn].
3186                            m_slot_releq.s.m_fq_list.list_head.list_prev->
3187                            list_next =
3188                            &mpt->m_slot_freeq_pairp[qn].
3189                            m_slot_allocq.s.m_fq_list.list_head;
3190                        mpt->m_slot_freeq_pairp[qn].
3191                            m_slot_releq.s.m_fq_list.list_head.list_next->
3192                            list_prev =
3193                            &mpt->m_slot_freeq_pairp[qn].
3194                            m_slot_allocq.s.m_fq_list.list_head;

3196                        mpt->m_slot_freeq_pairp[qn].
3197                            m_slot_releq.s.m_fq_list.list_head.list_next =
3198                            mpt->m_slot_freeq_pairp[qn].
3199                            m_slot_releq.s.m_fq_list.list_head.list_prev =
3200                            &mpt->m_slot_freeq_pairp[qn].
3201                            m_slot_releq.s.m_fq_list.list_head;
3202                        mpt->m_slot_freeq_pairp[qn].
3203                            m_slot_releq.s.m_fq_n = 0;
3204                } else {
3205                        mutex_exit(&mpt->m_slot_freeq_pairp[qn].
3206                            m_slot_releq.s.m_fq_mutex);
3207                        mutex_exit(&mpt->m_slot_freeq_pairp[qn].
3208                            m_slot_allocq.s.m_fq_mutex);
3209                        qn = (qn + 1) & (mpt->m_slot_freeq_pair_n - 1);
3210                        if (qn == qn_first)
3211                                return (FALSE);
3212                        else
3213                                goto qpair_retry;
3214                }
3215                mutex_exit(&mpt->m_slot_freeq_pairp[qn].
3216                    m_slot_releq.s.m_fq_mutex);
3217                pe = list_head(&mpt->m_slot_freeq_pairp[qn].
3218                    m_slot_allocq.s.m_fq_list);
3219                ASSERT(pe);
3220        }
3221        list_remove(&mpt->m_slot_freeq_pairp[qn].
3222            m_slot_allocq.s.m_fq_list, pe);
3223        slot = pe->slot;
3265                /*
3266                 * Make sure SMID is not using reserved value of 0
3267                 * and the TM request slot.
3268                 */
3269                ASSERT((slot > 0) && (slot <= slots->m_n_slots));
3228        ASSERT((slot > 0) && (slot <= slots->m_n_slots) &&
3229            mpt->m_slot_freeq_pairp[qn].m_slot_allocq.s.m_fq_n > 0);
3270                cmd->cmd_slot = slot;
3271                slots->m_slot[slot] = cmd;
3272                mpt->m_ncmds++;
3231        mpt->m_slot_freeq_pairp[qn].m_slot_allocq.s.m_fq_n--;
3232        ASSERT(mpt->m_slot_freeq_pairp[qn].m_slot_allocq.s.m_fq_n >= 0);

3234        mutex_exit(&mpt->m_slot_freeq_pairp[qn].m_slot_allocq.s.m_fq_mutex);
3274                /*
3275                 * only increment per target ncmds if this is not a
3276                 * command that has no target associated with it (i.e. a
3277                 * event acknoledgment)
3278                 */
3279                if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
3241                mutex_enter(&ptgt->m_tgt_intr_mutex);
3280                        ptgt->m_t_ncmds++;
3243                mutex_exit(&ptgt->m_tgt_intr_mutex);
3281                }
3282                cmd->cmd_active_timeout = cmd->cmd_pkt->pkt_time;
```

```
3284                 /*
3285                  * If initial timout is less than or equal to one tick, bump
3286                  * the timeout by a tick so that command doesn't timeout before
3287                  * its allotted time.
3288                  */
3289                 if (cmd->cmd_active_timeout <= mptsas_scsi_watchdog_tick) {
3290                         cmd->cmd_active_timeout += mptsas_scsi_watchdog_tick;
3291                 }
3292                 return (TRUE);
3293         } else {
3294                 int i;

3296                 /*
3297                  * If slot in use, scan until a free one is found. Don't use 0
3298                  * or final slot, which is reserved for TM requests.
3299                  */
3300                 for (i = 0; i < slots->m_n_slots; i++) {
3301                         slot = slots->m_tags;
3302                         if (++(slots->m_tags) > slots->m_n_slots) {
3303                                 slots->m_tags = 1;
3304                         }
3305                         if (slots->m_slot[slot] == NULL) {
3306                                 NDBG22(("found free slot %d", slot));
3307                                 goto alloc_tag;
3308                         }
3309                 }
3310         }
3311         return (FALSE);
3312 }
_____unchanged_portion_omitted_

3363 /*
3364  * tran_init_pkt(9E) - allocate scsi_pkt(9S) for command
3365  *
3366  * One of three possibilities:
3367  *      - allocate scsi_pkt
3368  *      - allocate scsi_pkt and DMA resources
3369  *      - allocate DMA resources to an already-allocated pkt
3370  */
3371 static struct scsi_pkt *
3372 mptsas_scsi_init_pkt(struct scsi_address *ap, struct scsi_pkt *pkt,
3373     struct buf *bp, int cmdlen, int statuslen, int tgtlen, int flags,
3374     int (*callback)(), caddr_t arg)
3375 {
3376         mptsas_cmd_t            *cmd, *new_cmd;
3377         mptsas_t                *mpt = ADDR2MPT(ap);
3378         int                     failure = 1;
3323 #ifndef __sparc
3379         uint_t                  oldcookiec;
3325 #endif  /* __sparc */
3380         mptsas_target_t         *ptgt = NULL;
3381         int                     rval;
3382         mptsas_tgt_private_t    *tgt_private;
3383         int                     kf;

3385         kf = (callback == SLEEP_FUNC)? KM_SLEEP: KM_NOSLEEP;

3387         tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->
3388             tran_tgt_private;
3389         ASSERT(tgt_private != NULL);
3390         if (tgt_private == NULL) {
3391                 return (NULL);
3392         }
3393         ptgt = tgt_private->t_private;
3394         ASSERT(ptgt != NULL);
```



```
3395         if (ptgt == NULL)
3396                 return (NULL);
3397         ap->a_target = ptgt->m_devhdl;
3398         ap->a_lun = tgt_private->t_lun;

3400         ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);
3401 #ifdef MPTSAS_TEST_EXTRN_ALLOC
3402         statuslen *= 100; tgtlen *= 4;
3403 #endif
3404         NDBG3(("mptsas_scsi_init_pkt:\n"
3405             "\ttgt=%d in=0x%p bp=0x%p clen=%d slen=%d tlen=%d flags=%x",
3406             ap->a_target, (void *)pkt, (void *)bp,
3407             cmdlen, statuslen, tgtlen, flags));

3409         /*
3410          * Allocate the new packet.
3411          */
3412         if (pkt == NULL) {
3413                 ddi_dma_handle_t        save_dma_handle;
3414                 ddi_dma_handle_t        save_arq_dma_handle;
3415                 struct buf              *save_arq_bp;
3416                 ddi_dma_cookie_t        save_arqcookie;
3363 #ifdef  __sparc
3364                 mptti_t                 *save_sg;
3365 #endif  /* __sparc */

3418                 cmd = kmem_cache_alloc(mpt->m_kmem_cache, kf);

3420                 if (cmd) {
3421                         save_dma_handle = cmd->cmd_dmahandle;
3422                         save_arq_dma_handle = cmd->cmd_arqhandle;
3423                         save_arq_bp = cmd->cmd_arq_buf;
3424                         save_arqcookie = cmd->cmd_arqcookie;
3374 #ifdef  __sparc
3375                         save_sg = cmd->cmd_sg;
3376 #endif  /* __sparc */
3425                         bzero(cmd, sizeof (*cmd) + scsi_pkt_size());
3426                         cmd->cmd_dmahandle = save_dma_handle;
3427                         cmd->cmd_arqhandle = save_arq_dma_handle;
3428                         cmd->cmd_arq_buf = save_arq_bp;
3429                         cmd->cmd_arqcookie = save_arqcookie;

3382 #ifdef  __sparc
3383                         cmd->cmd_sg = save_sg;
3384 #endif  /* __sparc */
3431                         pkt = (void *)((uchar_t *)cmd +
3432                             sizeof (struct mptsas_cmd));
3433                         pkt->pkt_ha_private = (opaque_t)cmd;
3434                         pkt->pkt_address = *ap;
3435                         pkt->pkt_private = (opaque_t)cmd->cmd_pkt_private;
3436                         pkt->pkt_scbp = (opaque_t)&cmd->cmd_scb;
3437                         pkt->pkt_cdbp = (opaque_t)&cmd->cmd_cdb;
3438                         cmd->cmd_pkt = (struct scsi_pkt *)pkt;
3439                         cmd->cmd_cdblen = (uchar_t)cmdlen;
3440                         cmd->cmd_scblen = statuslen;
3441                         cmd->cmd_rqslen = SENSE_LENGTH;
3442                         cmd->cmd_tgt_addr = ptgt;
3443                         failure = 0;
3444                 }

3446                 if (failure || (cmdlen > sizeof (cmd->cmd_cdb)) ||
3447                     (tgtlen > PKT_PRIV_LEN) ||
3448                     (statuslen > EXTCMDS_STATUS_SIZE)) {
3449                         if (failure == 0) {
3450                                 /*
3451                                  * if extern alloc fails, all will be
```

```
3452                                 * deallocated, including cmd
3453                                 */
3454                                failure = mptsas_pkt_alloc_extern(mpt, cmd,
3455                                    cmdlen, tgtlen, statuslen, kf);
3456                        }
3457                        if (failure) {
3458                                /*
3459                                 * if extern allocation fails, it will
3460                                 * deallocate the new pkt as well
3461                                 */
3462                                return (NULL);
3463                        }
3464                }
3465                new_cmd = cmd;

3467        } else {
3468                cmd = PKT2CMD(pkt);
3469                new_cmd = NULL;
3470        }


3427 #ifndef __sparc
3473        /* grab cmd->cmd_cookiec here as oldcookiec */

3475        oldcookiec = cmd->cmd_cookiec;
3431 #endif  /* __sparc */

3477        /*
3478         * If the dma was broken up into PARTIAL transfers cmd_nwin will be
3479         * greater than 0 and we'll need to grab the next dma window
3480         */
3481        /*
3482         * SLM-not doing extra command frame right now; may add later
3483         */

3485        if (cmd->cmd_nwin > 0) {

3487                /*
3488                 * Make sure we havn't gone past the the total number
3489                 * of windows
3490                 */
3491                if (++cmd->cmd_winindex >= cmd->cmd_nwin) {
3492                        return (NULL);
3493                }
3494                if (ddi_dma_getwin(cmd->cmd_dmahandle, cmd->cmd_winindex,
3495                    &cmd->cmd_dma_offset, &cmd->cmd_dma_len,
3496                    &cmd->cmd_cookie, &cmd->cmd_cookiec) == DDI_FAILURE) {
3497                        return (NULL);
3498                }
3499                goto get_dma_cookies;
3500        }


3503        if (flags & PKT_XARQ) {
3504                cmd->cmd_flags |= CFLAG_XARQ;
3505        }

3507        /*
3508         * DMA resource allocation.  This version assumes your
3509         * HBA has some sort of bus-mastering or onboard DMA capability, with a
3510         * scatter-gather list of length MPTSAS_MAX_DMA_SEGS, as given in the
3511         * ddi_dma_attr_t structure and passed to scsi_impl_dmaget.
3512         */
3513        if (bp && (bp->b_bcount != 0) &&
3514            (cmd->cmd_flags & CFLAG_DMAVALID) == 0) {
```



```
3516                int     cnt, dma_flags;
3517                mptti_t *dmap;          /* ptr to the S/G list */

3519                /*
3520                 * Set up DMA memory and position to the next DMA segment.
3521                 */
3522                ASSERT(cmd->cmd_dmahandle != NULL);

3524                if (bp->b_flags & B_READ) {
3525                        dma_flags = DDI_DMA_READ;
3526                        cmd->cmd_flags &= ~CFLAG_DMASEND;
3527                } else {
3528                        dma_flags = DDI_DMA_WRITE;
3529                        cmd->cmd_flags |= CFLAG_DMASEND;
3530                }
3531                if (flags & PKT_CONSISTENT) {
3532                        cmd->cmd_flags |= CFLAG_CMDIOPB;
3533                        dma_flags |= DDI_DMA_CONSISTENT;
3534                }

3536                if (flags & PKT_DMA_PARTIAL) {
3537                        dma_flags |= DDI_DMA_PARTIAL;
3538                }

3540                /*
3541                 * workaround for byte hole issue on psycho and
3542                 * schizo pre 2.1
3543                 */
3544                if ((bp->b_flags & B_READ) && ((bp->b_flags &
3545                    (B_PAGEIO|B_REMAPPED)) != B_PAGEIO) &&
3546                    ((uintptr_t)bp->b_un.b_addr & 0x7)) {
3547                        dma_flags |= DDI_DMA_CONSISTENT;
3548                }

3550                rval = ddi_dma_buf_bind_handle(cmd->cmd_dmahandle, bp,
3551                    dma_flags, callback, arg,
3552                    &cmd->cmd_cookie, &cmd->cmd_cookiec);
3553                if (rval == DDI_DMA_PARTIAL_MAP) {
3554                        (void) ddi_dma_numwin(cmd->cmd_dmahandle,
3555                            &cmd->cmd_nwin);
3556                        cmd->cmd_winindex = 0;
3557                        (void) ddi_dma_getwin(cmd->cmd_dmahandle,
3558                            cmd->cmd_winindex, &cmd->cmd_dma_offset,
3559                            &cmd->cmd_dma_len, &cmd->cmd_cookie,
3560                            &cmd->cmd_cookiec);
3561                } else if (rval && (rval != DDI_DMA_MAPPED)) {
3562                        switch (rval) {
3563                        case DDI_DMA_NORESOURCES:
3564                                bioerror(bp, 0);
3565                                break;
3566                        case DDI_DMA_BADATTR:
3567                        case DDI_DMA_NOMAPPING:
3568                                bioerror(bp, EFAULT);
3569                                break;
3570                        case DDI_DMA_TOOBIG:
3571                        default:
3572                                bioerror(bp, EINVAL);
3573                                break;
3574                        }
3575                        cmd->cmd_flags &= ~CFLAG_DMAVALID;
3576                        if (new_cmd) {
3577                                mptsas_scsi_destroy_pkt(ap, pkt);
3578                        }
3579                        return ((struct scsi_pkt *)NULL);
3580                }
```

```
3582 get_dma_cookies:
3583                 cmd->cmd_flags |= CFLAG_DMAVALID;
3584                 ASSERT(cmd->cmd_cookiec > 0);

3586                 if (cmd->cmd_cookiec > MPTSAS_MAX_CMD_SEGS) {
3587                         mptsas_log(mpt, CE_NOTE, "large cookiec received %d\n",
3588                             cmd->cmd_cookiec);
3589                         bioerror(bp, EINVAL);
3590                         if (new_cmd) {
3591                                 mptsas_scsi_destroy_pkt(ap, pkt);
3592                         }
3593                         return ((struct scsi_pkt *)NULL);
3594                 }

3596                 /*
3597                  * Allocate extra SGL buffer if needed.
3598                  */
3599                 if ((cmd->cmd_cookiec > MPTSAS_MAX_FRAME_SGES64(mpt)) &&
3600                     (cmd->cmd_extra_frames == NULL)) {
3601                         if (mptsas_alloc_extra_sgl_frame(mpt, cmd) ==
3602                             DDI_FAILURE) {
3603                                 mptsas_log(mpt, CE_WARN, "MPT SGL mem alloc "
3604                                     "failed");
3605                                 bioerror(bp, ENOMEM);
3606                                 if (new_cmd) {
3607                                         mptsas_scsi_destroy_pkt(ap, pkt);
3608                                 }
3609                                 return ((struct scsi_pkt *)NULL);
3610                         }
3611                 }

3613                 /*
3614                  * Always use scatter-gather transfer
3615                  * Use the loop below to store physical addresses of
3616                  * DMA segments, from the DMA cookies, into your HBA's
3617                  * scatter-gather list.
3618                  * We need to ensure we have enough kmem alloc'd
3619                  * for the sg entries since we are no longer using an
3620                  * array inside mptsas_cmd_t.
3621                  *
3622                  * We check cmd->cmd_cookiec against oldcookiec so
3623                  * the scatter-gather list is correctly allocated
3624                  */

3581 #ifndef __sparc
3626                 if (oldcookiec != cmd->cmd_cookiec) {
3627                         if (cmd->cmd_sg != (mptti_t *)NULL) {
3628                                 kmem_free(cmd->cmd_sg, sizeof (mptti_t) *
3629                                     oldcookiec);
3630                                 cmd->cmd_sg = NULL;
3631                         }
3632                 }

3634                 if (cmd->cmd_sg == (mptti_t *)NULL) {
3635                         cmd->cmd_sg = kmem_alloc((size_t)(sizeof (mptti_t)*
3636                             cmd->cmd_cookiec), kf);

3638                         if (cmd->cmd_sg == (mptti_t *)NULL) {
3639                                 mptsas_log(mpt, CE_WARN,
3640                                     "unable to kmem_alloc enough memory "
3641                                     "for scatter/gather list");
3642                 /*
3643                  * if we have an ENOMEM condition we need to behave
3644                  * the same way as the rest of this routine
3645                  */
```



```
3647                                 bioerror(bp, ENOMEM);
3648                                 if (new_cmd) {
3649                                         mptsas_scsi_destroy_pkt(ap, pkt);
3650                                 }
3651                                 return ((struct scsi_pkt *)NULL);
3652                         }
3653                 }

3610 #endif  /* __sparc */
3655                 dmap = cmd->cmd_sg;

3657                 ASSERT(cmd->cmd_cookie.dmac_size != 0);

3659                 /*
3660                  * store the first segment into the S/G list
3661                  */
3662                 dmap->count = cmd->cmd_cookie.dmac_size;
3663                 dmap->addr.address64.Low = (uint32_t)
3664                     (cmd->cmd_cookie.dmac_laddress & 0xffffffffull);
3665                 dmap->addr.address64.High = (uint32_t)
3666                     (cmd->cmd_cookie.dmac_laddress >> 32);

3668                 /*
3669                  * dmacount counts the size of the dma for this window
3670                  * (if partial dma is being used).  totaldmacount
3671                  * keeps track of the total amount of dma we have
3672                  * transferred for all the windows (needed to calculate
3673                  * the resid value below).
3674                  */
3675                 cmd->cmd_dmacount = cmd->cmd_cookie.dmac_size;
3676                 cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;

3678                 /*
3679                  * We already stored the first DMA scatter gather segment,
3680                  * start at 1 if we need to store more.
3681                  */
3682                 for (cnt = 1; cnt < cmd->cmd_cookiec; cnt++) {
3683                         /*
3684                          * Get next DMA cookie
3685                          */
3686                         ddi_dma_nextcookie(cmd->cmd_dmahandle,
3687                             &cmd->cmd_cookie);
3688                         dmap++;

3690                         cmd->cmd_dmacount += cmd->cmd_cookie.dmac_size;
3691                         cmd->cmd_totaldmacount += cmd->cmd_cookie.dmac_size;

3693                         /*
3694                          * store the segment parms into the S/G list
3695                          */
3696                         dmap->count = cmd->cmd_cookie.dmac_size;
3697                         dmap->addr.address64.Low = (uint32_t)
3698                             (cmd->cmd_cookie.dmac_laddress & 0xffffffffull);
3699                         dmap->addr.address64.High = (uint32_t)
3700                             (cmd->cmd_cookie.dmac_laddress >> 32);
3701                 }

3703                 /*
3704                  * If this was partially allocated we set the resid
3705                  * the amount of data NOT transferred in this window
3706                  * If there is only one window, the resid will be 0
3707                  */
3708                 pkt->pkt_resid = (bp->b_bcount - cmd->cmd_totaldmacount);
3709                 NDBG16(("mptsas_dmaget: cmd_dmacount=%d.", cmd->cmd_dmacount));
3710         }
3711         return (pkt);
```

```
3712 }

3714 /*
3715  * tran_destroy_pkt(9E) - scsi_pkt(9s) deallocation
3716  *
3717  * Notes:
3718  *      - also frees DMA resources if allocated
3719  *      - implicit DMA synchonization
3720  */
3721 static void
3722 mptsas_scsi_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
3723 {
3724         mptsas_cmd_t    *cmd = PKT2CMD(pkt);
3725         mptsas_t        *mpt = ADDR2MPT(ap);

3727         NDBG3(("mptsas_scsi_destroy_pkt: target=%d pkt=0x%p",
3728             ap->a_target, (void *)pkt));

3730         if (cmd->cmd_flags & CFLAG_DMAVALID) {
3731                 (void) ddi_dma_unbind_handle(cmd->cmd_dmahandle);
3732                 cmd->cmd_flags &= ~CFLAG_DMAVALID;
3733         }

3690 #ifndef __sparc
3735         if (cmd->cmd_sg) {
3736                 kmem_free(cmd->cmd_sg, sizeof (mptti_t) * cmd->cmd_cookiec);
3737                 cmd->cmd_sg = NULL;
3738         }

3695 #endif  /* __sparc */
3740         mptsas_free_extra_sgl_frame(mpt, cmd);

3742         if ((cmd->cmd_flags &
3743             (CFLAG_FREE | CFLAG_CDBEXTERN | CFLAG_PRIVEXTERN |
3744             CFLAG_SCBEXTERN)) == 0) {
3745                 cmd->cmd_flags = CFLAG_FREE;
3746                 kmem_cache_free(mpt->m_kmem_cache, (void *)cmd);
3747         } else {
3748                 mptsas_pkt_destroy_extern(mpt, cmd);
3749         }
3750 }

3752 /*
3753  * kmem cache constructor and destructor:
3754  * When constructing, we bzero the cmd and allocate the dma handle
3755  * When destructing, just free the dma handle
3756  */
3757 static int
3758 mptsas_kmem_cache_constructor(void *buf, void *cdrarg, int kmflags)
3759 {
3760         mptsas_cmd_t            *cmd = buf;
3761         mptsas_t                *mpt  = cdrarg;
3762         struct scsi_address     ap;
3763         uint_t                  cookiec;
3764         ddi_dma_attr_t          arq_dma_attr;
3765         int                     (*callback)(caddr_t);

3767         callback = (kmflags == KM_SLEEP)? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;

3769         NDBG4(("mptsas_kmem_cache_constructor"));

3771         ap.a_hba_tran = mpt->m_tran;
3772         ap.a_target = 0;
3773         ap.a_lun = 0;

3775         /*
```



```
3776          * allocate a dma handle
3777          */
3778         if ((ddi_dma_alloc_handle(mpt->m_dip, &mpt->m_io_dma_attr, callback,
3779             NULL, &cmd->cmd_dmahandle)) != DDI_SUCCESS) {
3780                 cmd->cmd_dmahandle = NULL;
3781                 return (-1);
3782         }

3784         cmd->cmd_arq_buf = scsi_alloc_consistent_buf(&ap, (struct buf *)NULL,
3785             SENSE_LENGTH, B_READ, callback, NULL);
3786         if (cmd->cmd_arq_buf == NULL) {
3787                 ddi_dma_free_handle(&cmd->cmd_dmahandle);
3788                 cmd->cmd_dmahandle = NULL;
3789                 return (-1);
3790         }

3792         /*
3793          * allocate a arq handle
3794          */
3795         arq_dma_attr = mpt->m_msg_dma_attr;
3796         arq_dma_attr.dma_attr_sgllen = 1;
3797         if ((ddi_dma_alloc_handle(mpt->m_dip, &arq_dma_attr, callback,
3798             NULL, &cmd->cmd_arqhandle)) != DDI_SUCCESS) {
3799                 ddi_dma_free_handle(&cmd->cmd_dmahandle);
3800                 scsi_free_consistent_buf(cmd->cmd_arq_buf);
3801                 cmd->cmd_dmahandle = NULL;
3802                 cmd->cmd_arqhandle = NULL;
3803                 return (-1);
3804         }

3806         if (ddi_dma_buf_bind_handle(cmd->cmd_arqhandle,
3807             cmd->cmd_arq_buf, (DDI_DMA_READ | DDI_DMA_CONSISTENT),
3808             callback, NULL, &cmd->cmd_arqcookie, &cookiec) != DDI_SUCCESS) {
3809                 ddi_dma_free_handle(&cmd->cmd_dmahandle);
3810                 ddi_dma_free_handle(&cmd->cmd_arqhandle);
3811                 scsi_free_consistent_buf(cmd->cmd_arq_buf);
3812                 cmd->cmd_dmahandle = NULL;
3813                 cmd->cmd_arqhandle = NULL;
3814                 cmd->cmd_arq_buf = NULL;
3815                 return (-1);
3816         }
3773         /*
3774          * In sparc, the sgl length in most of the cases would be 1, so we
3775          * pre-allocate it in cache. On x86, the max number would be 256,
3776          * pre-allocate a maximum would waste a lot of memory especially
3777          * when many cmds are put onto waitq.
3778          */
3779 #ifdef  __sparc
3780         cmd->cmd_sg = kmem_alloc((size_t)(sizeof (mptti_t)*
3781             MPTSAS_MAX_CMD_SEGS), KM_SLEEP);
3782 #endif  /* __sparc */

3818         return (0);
3819 }

3821 static void
3822 mptsas_kmem_cache_destructor(void *buf, void *cdrarg)
3823 {
3824 #ifndef __lock_lint
3825         _NOTE(ARGUNUSED(cdrarg))
3826 #endif
3827         mptsas_cmd_t    *cmd = buf;

3829         NDBG4(("mptsas_kmem_cache_destructor"));

3831         if (cmd->cmd_arqhandle) {
```

```
3832                 (void) ddi_dma_unbind_handle(cmd->cmd_arqhandle);
3833                 ddi_dma_free_handle(&cmd->cmd_arqhandle);
3834                 cmd->cmd_arqhandle = NULL;
3835         }
3836         if (cmd->cmd_arq_buf) {
3837                 scsi_free_consistent_buf(cmd->cmd_arq_buf);
3838                 cmd->cmd_arq_buf = NULL;
3839         }
3840         if (cmd->cmd_dmahandle) {
3841                 ddi_dma_free_handle(&cmd->cmd_dmahandle);
3842                 cmd->cmd_dmahandle = NULL;
3843         }
3810 #ifdef  __sparc
3811         if (cmd->cmd_sg) {
3812                 kmem_free(cmd->cmd_sg, sizeof (mptti_t)* MPTSAS_MAX_CMD_SEGS);
3813                 cmd->cmd_sg = NULL;
3814         }
3815 #endif  /* __sparc */
3844 }
_____unchanged_portion_omitted_

4493 /*
4494  * Interrupt handling
4495  * Utility routine.  Poll for status of a command sent to HBA
4496  * without interrupts (a FLAG_NOINTR command).
4497  */
4498 int
4499 mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
4500 {
4501         int     rval = TRUE;

4503         NDBG5(("mptsas_poll: cmd=0x%p", (void *)poll_cmd));

4477         /*
4478          * In order to avoid using m_mutex in ISR(a new separate mutex
4479          * m_intr_mutex is introduced) and keep the same lock logic,
4480          * the m_intr_mutex should be used to protect the getting and
4481          * setting of the ReplyDescriptorIndex.
4482          *
4483          * Since the m_intr_mutex would be released during processing the poll
4484          * cmd, so we should set the poll flag earlier here to make sure the
4485          * polled cmd be handled in this thread/context. A side effect is other
4486          * cmds during the period between the flag set and reset are also
4487          * handled in this thread and not the ISR. Since the poll cmd is not
4488          * so common, so the performance degradation in this case is not a big
4489          * issue.
4490          */
4491         mutex_enter(&mpt->m_intr_mutex);
4492         mpt->m_polled_intr = 1;
4493         mutex_exit(&mpt->m_intr_mutex);

4505         if ((poll_cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
4506                 mptsas_restart_hba(mpt);
4507         }

4509         /*
4510          * Wait, using drv_usecwait(), long enough for the command to
4511          * reasonably return from the target if the target isn't
4512          * "dead".  A polled command may well be sent from scsi_poll, and
4513          * there are retries built in to scsi_poll if the transport
4514          * accepted the packet (TRAN_ACCEPT).  scsi_poll waits 1 second
4515          * and retries the transport up to scsi_poll_busycnt times
4516          * (currently 60) if
4517          * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
4518          * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
4519          *
```



```
4520          * limit the waiting to avoid a hang in the event that the
4521          * cmd never gets started but we are still receiving interrupts
4522          */
4523         while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
4524                 if (mptsas_wait_intr(mpt, polltime) == FALSE) {
4525                         NDBG5(("mptsas_poll: command incomplete"));
4526                         rval = FALSE;
4527                         break;
4528                 }
4529         }

4521         mutex_enter(&mpt->m_intr_mutex);
4522         mpt->m_polled_intr = 0;
4523         mutex_exit(&mpt->m_intr_mutex);

4531         if (rval == FALSE) {

4533                 /*
4534                  * this isn't supposed to happen, the hba must be wedged
4535                  * Mark this cmd as a timeout.
4536                  */
4537                 mptsas_set_pkt_reason(mpt, poll_cmd, CMD_TIMEOUT,
4538                     (STAT_TIMEOUT|STAT_ABORTED));

4540                 if (poll_cmd->cmd_queued == FALSE) {

4542                         NDBG5(("mptsas_poll: not on waitq"));

4544                         poll_cmd->cmd_pkt->pkt_state |=
4545                             (STATE_GOT_BUS|STATE_GOT_TARGET|STATE_SENT_CMD);
4546                 } else {

4548                         /* find and remove it from the waitq */
4549                         NDBG5(("mptsas_poll: delete from waitq"));
4550                         mptsas_waitq_delete(mpt, poll_cmd);
4551                 }

4553         }
4554         mptsas_fma_check(mpt, poll_cmd);
4555         NDBG5(("mptsas_poll: done"));
4556         return (rval);
4557 }

4559 /*
4560  * Used for polling cmds and TM function
4561  */
4562 static int
4563 mptsas_wait_intr(mptsas_t *mpt, int polltime)
4564 {
4565         int                             cnt;
4566         pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
4561         Mpi2ReplyDescriptorsUnion_t     reply_desc_union_v;
4567         uint32_t                        int_mask;
4563         uint8_t reply_type;

4569         NDBG5(("mptsas_wait_intr"));

4571         mpt->m_polled_intr = 1;

4573         /*
4574          * Get the current interrupt mask and disable interrupts.  When
4575          * re-enabling ints, set mask to saved value.
4576          */
4577         int_mask = ddi_get32(mpt->m_datap, &mpt->m_reg->HostInterruptMask);
4578         MPTSAS_DISABLE_INTR(mpt);
```

```
4580         /*
4581          * Keep polling for at least (polltime * 1000) seconds
4582          */
4583         for (cnt = 0; cnt < polltime; cnt++) {
4579                 mutex_enter(&mpt->m_intr_mutex);
4584                 (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
4585                     DDI_DMA_SYNC_FORCPU);

4587                 reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
4588                     MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

4590                 if (ddi_get32(mpt->m_acc_post_queue_hdl,
4591                     &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
4592                     ddi_get32(mpt->m_acc_post_queue_hdl,
4593                     &reply_desc_union->Words.High) == 0xFFFFFFFF) {
4590                         mutex_exit(&mpt->m_intr_mutex);
4594                         drv_usecwait(1000);
4595                         continue;
4596                 }

4595                 reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
4596                     &reply_desc_union->Default.ReplyFlags);
4597                 reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
4598                 reply_desc_union_v.Default.ReplyFlags = reply_type;
4599                 if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS) {
4600                         reply_desc_union_v.SCSIIOSuccess.SMID =
4601                             ddi_get16(mpt->m_acc_post_queue_hdl,
4602                             &reply_desc_union->SCSIIOSuccess.SMID);
4603                 } else if (reply_type ==
4604                     MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
4605                         reply_desc_union_v.AddressReply.ReplyFrameAddress =
4606                             ddi_get32(mpt->m_acc_post_queue_hdl,
4607                             &reply_desc_union->AddressReply.ReplyFrameAddress);
4608                         reply_desc_union_v.AddressReply.SMID =
4609                             ddi_get16(mpt->m_acc_post_queue_hdl,
4610                             &reply_desc_union->AddressReply.SMID);
4611                 }
4598                 /*
4599                  * The reply is valid, process it according to its
4600                  * type.
4613                  * Clear the reply descriptor for re-use and increment
4614                  * index.
4601                  */
4602                 mptsas_process_intr(mpt, reply_desc_union);
4616                 ddi_put64(mpt->m_acc_post_queue_hdl,
4617                     &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
4618                     0xFFFFFFFFFFFFFFFF);
4619                 (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
4620                     DDI_DMA_SYNC_FORDEV);

4604                 if (++mpt->m_post_index == mpt->m_post_queue_depth) {
4605                         mpt->m_post_index = 0;
4606                 }

4608                 /*
4609                  * Update the global reply index
4610                  */
4611                 ddi_put32(mpt->m_datap,
4612                     &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);
4613                 mpt->m_polled_intr = 0;
4631                 mutex_exit(&mpt->m_intr_mutex);

4615                 /*
4634                  * The reply is valid, process it according to its
4635                  * type.
4636                  */
```



```
4637                 mptsas_process_intr(mpt, &reply_desc_union_v);


4640                 /*
4616                  * Re-enable interrupts and quit.
4617                  */
4618                 ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask,
4619                     int_mask);
4620                 return (TRUE);

4622         }

4624         /*
4625          * Clear polling flag, re-enable interrupts and quit.
4626          */
4627         mpt->m_polled_intr = 0;
4628         ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, int_mask);
4629         return (FALSE);
4630 }

4656 /*
4657  * For fastpath, the m_intr_mutex should be held from the begining to the end,
4658  * so we only treat those cmds that need not release m_intr_mutex(even just for
4659  * a moment) as candidate for fast processing. otherwise, we don't handle them
4660  * and just return, then in ISR, those cmds would be handled later with m_mutex
4661  * held and m_intr_mutex not held.
4662  */
4663 static int
4664 mptsas_handle_io_fastpath(mptsas_t *mpt,
4665     uint16_t SMID)
4666 {
4667         mptsas_slots_t                          *slots = mpt->m_active;
4668         mptsas_cmd_t                            *cmd = NULL;
4669         struct scsi_pkt                         *pkt;

4671         /*
4672          * This is a success reply so just complete the IO.  First, do a sanity
4673          * check on the SMID.  The final slot is used for TM requests, which
4674          * would not come into this reply handler.
4675          */
4676         if ((SMID == 0) || (SMID > slots->m_n_slots)) {
4677                 mptsas_log(mpt, CE_WARN, "?Received invalid SMID of %d\n",
4678                     SMID);
4679                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4680                 return (TRUE);
4681         }

4683         cmd = slots->m_slot[SMID];

4685         /*
4686          * print warning and return if the slot is empty
4687          */
4688         if (cmd == NULL) {
4689                 mptsas_log(mpt, CE_WARN, "?NULL command for successful SCSI IO "
4690                     "in slot %d", SMID);
4691                 return (TRUE);
4692         }

4694         pkt = CMD2PKT(cmd);
4695         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET | STATE_SENT_CMD |
4696             STATE_GOT_STATUS);
4697         if (cmd->cmd_flags & CFLAG_DMAVALID) {
4698                 pkt->pkt_state |= STATE_XFERRED_DATA;
4699         }
4700         pkt->pkt_resid = 0;
```

```
4702         /*
4703          * If the cmd is a IOC, or a passthrough, then we don't process it in
4704          * fastpath, and later it would be handled by mptsas_process_intr()
4705          * with m_mutex protected.
4706          */
4707         if (cmd->cmd_flags & (CFLAG_PASSTHRU | CFLAG_CMDIOC)) {
4708                 return (FALSE);
4709         } else {
4710                 mptsas_remove_cmd0(mpt, cmd);
4711         }

4713         if (cmd->cmd_flags & CFLAG_RETRY) {
4714                 /*
4715                  * The target returned QFULL or busy, do not add tihs
4716                  * pkt to the doneq since the hba will retry
4717                  * this cmd.
4718                  *
4719                  * The pkt has already been resubmitted in
4720                  * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4721                  * Remove this cmd_flag here.
4722                  */
4723                 cmd->cmd_flags &= ~CFLAG_RETRY;
4724         } else {
4725                 mptsas_doneq_add0(mpt, cmd);
4726         }

4728         /*
4729          * In fastpath, the cmd should only be a context reply, so just check
4730          * the post queue of the reply descriptor and the dmahandle of the cmd
4731          * is enough. No sense data in this case and no need to check the dma
4732          * handle where sense data dma info is saved, the dma handle of the
4733          * reply frame, and the dma handle of the reply free queue.
4734          * For the dma handle of the request queue. Check fma here since we
4735          * are sure the request must have already been sent/DMAed correctly.
4736          * otherwise checking in mptsas_scsi_start() is not correct since
4737          * at that time the dma may not start.
4738          */
4739         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
4740             DDI_SUCCESS) ||
4741             (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl) !=
4742             DDI_SUCCESS)) {
4743                 ddi_fm_service_impact(mpt->m_dip,
4744                     DDI_SERVICE_UNAFFECTED);
4745                 pkt->pkt_reason = CMD_TRAN_ERR;
4746                 pkt->pkt_statistics = 0;
4747         }
4748         if (cmd->cmd_dmahandle &&
4749             (mptsas_check_dma_handle(cmd->cmd_dmahandle) != DDI_SUCCESS)) {
4750                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4751                 pkt->pkt_reason = CMD_TRAN_ERR;
4752                 pkt->pkt_statistics = 0;
4753         }
4754         if ((cmd->cmd_extra_frames &&
4755             ((mptsas_check_dma_handle(cmd->cmd_extra_frames->m_dma_hdl) !=
4756             DDI_SUCCESS) ||
4757             (mptsas_check_acc_handle(cmd->cmd_extra_frames->m_acc_hdl) !=
4758             DDI_SUCCESS)))) {
4759                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4760                 pkt->pkt_reason = CMD_TRAN_ERR;
4761                 pkt->pkt_statistics = 0;
4762         }

4764         return (TRUE);
4765 }
```

```
4632 static void
```



```
4633 mptsas_handle_scsi_io_success(mptsas_t *mpt,
4634     pMpi2ReplyDescriptorsUnion_t reply_desc)
4635 {
4636         pMpi2SCSIIOSuccessReplyDescriptor_t     scsi_io_success;
4637         uint16_t                                SMID;
4638         mptsas_slots_t                          *slots = mpt->m_active;
4639         mptsas_cmd_t                            *cmd = NULL;
4640         struct scsi_pkt                         *pkt;

4642         ASSERT(mutex_owned(&mpt->m_mutex));

4644         scsi_io_success = (pMpi2SCSIIOSuccessReplyDescriptor_t)reply_desc;
4645         SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &scsi_io_success->SMID);
4778         SMID = scsi_io_success->SMID;

4647         /*
4648          * This is a success reply so just complete the IO.  First, do a sanity
4649          * check on the SMID.  The final slot is used for TM requests, which
4650          * would not come into this reply handler.
4651          */
4652         if ((SMID == 0) || (SMID > slots->m_n_slots)) {
4653                 mptsas_log(mpt, CE_WARN, "?Received invalid SMID of %d\n",
4654                     SMID);
4655                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4656                 return;
4657         }

4659         cmd = slots->m_slot[SMID];

4661         /*
4662          * print warning and return if the slot is empty
4663          */
4664         if (cmd == NULL) {
4665                 mptsas_log(mpt, CE_WARN, "?NULL command for successful SCSI IO "
4666                     "in slot %d", SMID);
4667                 return;
4668         }

4670         pkt = CMD2PKT(cmd);
4671         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET | STATE_SENT_CMD |
4672             STATE_GOT_STATUS);
4673         if (cmd->cmd_flags & CFLAG_DMAVALID) {
4674                 pkt->pkt_state |= STATE_XFERRED_DATA;
4675         }
4676         pkt->pkt_resid = 0;

4678         if (cmd->cmd_flags & CFLAG_PASSTHRU) {
4679                 cmd->cmd_flags |= CFLAG_FINISHED;
4680                 cv_broadcast(&mpt->m_passthru_cv);
4681                 return;
4682         } else {
4683                 mptsas_remove_cmd(mpt, cmd);
4684         }

4686         if (cmd->cmd_flags & CFLAG_RETRY) {
4687                 /*
4688                  * The target returned QFULL or busy, do not add tihs
4689                  * pkt to the doneq since the hba will retry
4690                  * this cmd.
4691                  *
4692                  * The pkt has already been resubmitted in
4693                  * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4694                  * Remove this cmd_flag here.
4695                  */
4696                 cmd->cmd_flags &= ~CFLAG_RETRY;
4697         } else {
```

```
4698                 mptsas_doneq_add(mpt, cmd);
4699         }
4700 }

4702 static void
4703 mptsas_handle_address_reply(mptsas_t *mpt,
4704     pMpi2ReplyDescriptorsUnion_t reply_desc)
4705 {
4706         pMpi2AddressReplyDescriptor_t   address_reply;
4707         pMPI2DefaultReply_t             reply;
4708         mptsas_fw_diagnostic_buffer_t   *pBuffer;
4709         uint32_t                        reply_addr;
4710         uint16_t                        SMID, iocstatus;
4711         mptsas_slots_t                  *slots = mpt->m_active;
4712         mptsas_cmd_t                    *cmd = NULL;
4713         uint8_t                         function, buffer_type;
4714         m_replyh_arg_t                  *args;
4715         int                             reply_frame_no;

4717         ASSERT(mutex_owned(&mpt->m_mutex));

4719         address_reply = (pMpi2AddressReplyDescriptor_t)reply_desc;
4720         reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
4721             &address_reply->ReplyFrameAddress);
4722         SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

4854         reply_addr = address_reply->ReplyFrameAddress;
4855         SMID = address_reply->SMID;
4724         /*
4725          * If reply frame is not in the proper range we should ignore this
4726          * message and exit the interrupt handler.
4727          */
4728         if ((reply_addr < mpt->m_reply_frame_dma_addr) ||
4729             (reply_addr >= (mpt->m_reply_frame_dma_addr +
4730             (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
4731             ((reply_addr - mpt->m_reply_frame_dma_addr) %
4732             mpt->m_reply_frame_size != 0)) {
4733                 mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
4734                     "address 0x%x\n", reply_addr);
4735                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4736                 return;
4737         }

4739         (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
4740             DDI_DMA_SYNC_FORCPU);
4741         reply = (pMPI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
4742             mpt->m_reply_frame_dma_addr));
4743         function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

4745         /*
4746          * don't get slot information and command for events since these values
4747          * don't exist
4748          */
4749         if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
4750             (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
4751                 /*
4752                  * This could be a TM reply, which use the last allocated SMID,
4753                  * so allow for that.
4754                  */
4755                 if ((SMID == 0) || (SMID > (slots->m_n_slots + 1))) {
4756                         mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
4757                             "%d\n", SMID);
4758                         ddi_fm_service_impact(mpt->m_dip,
4759                             DDI_SERVICE_UNAFFECTED);
4760                         return;
4761                 }
```



```
4763                 cmd = slots->m_slot[SMID];

4765                 /*
4766                  * print warning and return if the slot is empty
4767                  */
4768                 if (cmd == NULL) {
4769                         mptsas_log(mpt, CE_WARN, "?NULL command for address "
4770                             "reply in slot %d", SMID);
4771                         return;
4772                 }
4773                 if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
4774                     (cmd->cmd_flags & CFLAG_CONFIG) ||
4775                     (cmd->cmd_flags & CFLAG_FW_DIAG)) {
4776                         cmd->cmd_rfm = reply_addr;
4777                         cmd->cmd_flags |= CFLAG_FINISHED;
4778                         cv_broadcast(&mpt->m_passthru_cv);
4779                         cv_broadcast(&mpt->m_config_cv);
4780                         cv_broadcast(&mpt->m_fw_diag_cv);
4781                         return;
4782                 } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
4783                         mptsas_remove_cmd(mpt, cmd);
4784                 }
4785                 NDBG31(("\t\tmptsas_process_intr: slot=%d", SMID));
4786         }
4787         /*
4788          * Depending on the function, we need to handle
4789          * the reply frame (and cmd) differently.
4790          */
4791         switch (function) {
4792         case MPI2_FUNCTION_SCSI_IO_REQUEST:
4793                 mptsas_check_scsi_io_error(mpt, (pMpi2SCSIIOReply_t)reply, cmd);
4794                 break;
4795         case MPI2_FUNCTION_SCSI_TASK_MGMT:
4796                 cmd->cmd_rfm = reply_addr;
4797                 mptsas_check_task_mgt(mpt, (pMpi2SCSIManagementReply_t)reply,
4798                     cmd);
4799                 break;
4800         case MPI2_FUNCTION_FW_DOWNLOAD:
4801                 cmd->cmd_flags |= CFLAG_FINISHED;
4802                 cv_signal(&mpt->m_fw_cv);
4803                 break;
4804         case MPI2_FUNCTION_EVENT_NOTIFICATION:
4805                 reply_frame_no = (reply_addr - mpt->m_reply_frame_dma_addr) /
4806                     mpt->m_reply_frame_size;
4807                 args = &mpt->m_replyh_args[reply_frame_no];
4808                 args->mpt = (void *)mpt;
4809                 args->rfm = reply_addr;

4811                 /*
4812                  * Record the event if its type is enabled in
4813                  * this mpt instance by ioctl.
4814                  */
4815                 mptsas_record_event(args);

4817                 /*
4818                  * Handle time critical events
4819                  * NOT_RESPONDING/ADDED only now
4820                  */
4821                 if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {
4822                         /*
4823                          * Would not return main process,
4824                          * just let taskq resolve ack action
4825                          * and ack would be sent in taskq thread
4826                          */
4827                         NDBG20(("send mptsas_handle_event_sync success"));
```

```
4828                }

4830                if (mpt->m_in_reset) {
4831                        NDBG20(("dropping event received during reset"));
4832                        return;
4833                }

4835                if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
4836                    (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
4837                        mptsas_log(mpt, CE_WARN, "No memory available"
4838                        "for dispatch taskq");
4839                        /*
4840                         * Return the reply frame to the free queue.
4841                         */
4842                        ddi_put32(mpt->m_acc_free_queue_hdl,
4843                            &((uint32_t *)(void *)
4844                            mpt->m_free_queue)[mpt->m_free_index], reply_addr);
4845                        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4846                            DDI_DMA_SYNC_FORDEV);
4847                        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4848                                mpt->m_free_index = 0;
4849                        }

4851                        ddi_put32(mpt->m_datap,
4852                            &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
4853                }
4854                return;
4855        case MPI2_FUNCTION_DIAG_BUFFER_POST:
4856                /*
4857                 * If SMID is 0, this implies that the reply is due to a
4858                 * release function with a status that the buffer has been
4859                 * released.  Set the buffer flags accordingly.
4860                 */
4861                if (SMID == 0) {
4862                        iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
4863                            &reply->IOCStatus);
4864                        buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
4865                            &(((pMpi2DiagBufferPostReply_t)reply)->BufferType));
4866                        if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
4867                                pBuffer =
4868                                    &mpt->m_fw_diag_buffer_list[buffer_type];
4869                                pBuffer->valid_data = TRUE;
4870                                pBuffer->owned_by_firmware = FALSE;
4871                                pBuffer->immediate = FALSE;
4872                        }
4873                } else {
4874                        /*
4875                         * Normal handling of diag post reply with SMID.
4876                         */
4877                        cmd = slots->m_slot[SMID];

4879                        /*
4880                         * print warning and return if the slot is empty
4881                         */
4882                        if (cmd == NULL) {
4883                                mptsas_log(mpt, CE_WARN, "?NULL command for "
4884                                    "address reply in slot %d", SMID);
4885                                return;
4886                        }
4887                        cmd->cmd_rfm = reply_addr;
4888                        cmd->cmd_flags |= CFLAG_FINISHED;
4889                        cv_broadcast(&mpt->m_fw_diag_cv);
4890                }
4891                return;
4892        default:
4893                mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
```



```
4894                        break;
4895        }

4897        /*
4898         * Return the reply frame to the free queue.
4899         */
4900        ddi_put32(mpt->m_acc_free_queue_hdl,
4901            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
4902            reply_addr);
4903        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4904            DDI_DMA_SYNC_FORDEV);
4905        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4906                mpt->m_free_index = 0;
4907        }
4908        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
4909            mpt->m_free_index);

4911        if (cmd->cmd_flags & CFLAG_FW_CMD)
4912                return;

4914        if (cmd->cmd_flags & CFLAG_RETRY) {
4915                /*
4916                 * The target returned QFULL or busy, do not add tihs
4917                 * pkt to the doneq since the hba will retry
4918                 * this cmd.
4919                 *
4920                 * The pkt has already been resubmitted in
4921                 * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4922                 * Remove this cmd_flag here.
4923                 */
4924                cmd->cmd_flags &= ~CFLAG_RETRY;
4925        } else {
4926                mptsas_doneq_add(mpt, cmd);
4927        }
4928 }

4930 static void
4931 mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
4932     mptsas_cmd_t *cmd)
4933 {
4934        uint8_t                 scsi_status, scsi_state;
4935        uint16_t                ioc_status;
4936        uint32_t                xferred, sensecount, responsedata, loginfo = 0;
4937        struct scsi_pkt         *pkt;
4938        struct scsi_arq_status  *arqstat;
4939        struct buf              *bp;
4940        mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
4941        uint8_t                 *sensedata = NULL;

4943        if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
4944            (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
4945                bp = cmd->cmd_ext_arq_buf;
4946        } else {
4947                bp = cmd->cmd_arq_buf;
4948        }

4950        scsi_status = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIStatus);
4951        ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
4952        scsi_state = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIState);
4953        xferred = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->TransferCount);
4954        sensecount = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->SenseCount);
4955        responsedata = ddi_get32(mpt->m_acc_reply_frame_hdl,
4956            &reply->ResponseInfo);

4958        if (ioc_status & MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
4959                loginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
```

```
4960                    &reply->IOCLogInfo);
4961                mptsas_log(mpt, CE_NOTE,
4962                    "?Log info 0x%x received for target %d.\n"
4963                    "\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
4964                    loginfo, Tgt(cmd), scsi_status, ioc_status,
4965                    scsi_state);
4966        }

4968        NDBG31(("\t\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
4969            scsi_status, ioc_status, scsi_state));

4971        pkt = CMD2PKT(cmd);
4972        *(pkt->pkt_scbp) = scsi_status;

4974        if (loginfo == 0x31170000) {
4975                /*
4976                 * if loginfo PL_LOGINFO_CODE_IO_DEVICE_MISSING_DELAY_RETRY
4977                 * 0x31170000 comes, that means the device missing delay
4978                 * is in progressing, the command need retry later.
4979                 */
4980                *(pkt->pkt_scbp) = STATUS_BUSY;
4981                return;
4982        }

4984        if ((scsi_state & MPI2_SCSI_STATE_NO_SCSI_STATUS) &&
4985            ((ioc_status & MPI2_IOCSTATUS_MASK) ==
4986            MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE)) {
4987                pkt->pkt_reason = CMD_INCOMPLETE;
4988                pkt->pkt_state |= STATE_GOT_BUS;
5115                mutex_enter(&ptgt->m_tgt_intr_mutex);
4989                if (ptgt->m_reset_delay == 0) {
4990                        mptsas_set_throttle(mpt, ptgt,
4991                            DRAIN_THROTTLE);
4992                }
5120                mutex_exit(&ptgt->m_tgt_intr_mutex);
4993                return;
4994        }

4996        if (scsi_state & MPI2_SCSI_STATE_RESPONSE_INFO_VALID) {
4997                responsedata &= 0x000000FF;
4998                if (responsedata & MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF) {
4999                        mptsas_log(mpt, CE_NOTE, "Do not support the TLR\n");
5000                        pkt->pkt_reason = CMD_TLR_OFF;
5001                        return;
5002                }
5003        }


5006        switch (scsi_status) {
5007        case MPI2_SCSI_STATUS_CHECK_CONDITION:
5008                pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5009                arqstat = (void*)(pkt->pkt_scbp);
5010                arqstat->sts_rqpkt_status = *((struct scsi_status *)
5011                    (pkt->pkt_scbp));
5012                pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET |
5013                    STATE_SENT_CMD | STATE_GOT_STATUS | STATE_ARQ_DONE);
5014                if (cmd->cmd_flags & CFLAG_XARQ) {
5015                        pkt->pkt_state |= STATE_XARQ_DONE;
5016                }
5017                if (pkt->pkt_resid != cmd->cmd_dmacount) {
5018                        pkt->pkt_state |= STATE_XFERRED_DATA;
5019                }
5020                arqstat->sts_rqpkt_reason = pkt->pkt_reason;
5021                arqstat->sts_rqpkt_state  = pkt->pkt_state;
5022                arqstat->sts_rqpkt_state |= STATE_XFERRED_DATA;
5023                arqstat->sts_rqpkt_statistics = pkt->pkt_statistics;
```



```
5024                sensedata = (uint8_t *)&arqstat->sts_sensedata;

5026                bcopy((uchar_t *)bp->b_un.b_addr, sensedata,
5027                    ((cmd->cmd_rqslen >= sensecount) ? sensecount :
5028                    cmd->cmd_rqslen));
5029                arqstat->sts_rqpkt_resid = (cmd->cmd_rqslen - sensecount);
5030                cmd->cmd_flags |= CFLAG_CMDARQ;
5031                /*
5032                 * Set proper status for pkt if autosense was valid
5033                 */
5034                if (scsi_state & MPI2_SCSI_STATE_AUTOSENSE_VALID) {
5035                        struct scsi_status zero_status = { 0 };
5036                        arqstat->sts_rqpkt_status = zero_status;
5037                }

5039                /*
5040                 * ASC=0x47 is parity error
5041                 * ASC=0x48 is initiator detected error received
5042                 */
5043                if ((scsi_sense_key(sensedata) == KEY_ABORTED_COMMAND) &&
5044                    ((scsi_sense_asc(sensedata) == 0x47) ||
5045                    (scsi_sense_asc(sensedata) == 0x48))) {
5046                        mptsas_log(mpt, CE_NOTE, "Aborted_command!");
5047                }

5049                /*
5050                 * ASC/ASCQ=0x3F/0x0E means report_luns data changed
5051                 * ASC/ASCQ=0x25/0x00 means invalid lun
5052                 */
5053                if (((scsi_sense_key(sensedata) == KEY_UNIT_ATTENTION) &&
5054                    (scsi_sense_asc(sensedata) == 0x3F) &&
5055                    (scsi_sense_ascq(sensedata) == 0x0E)) ||
5056                    ((scsi_sense_key(sensedata) == KEY_ILLEGAL_REQUEST) &&
5057                    (scsi_sense_asc(sensedata) == 0x25) &&
5058                    (scsi_sense_ascq(sensedata) == 0x00))) {
5059                        mptsas_topo_change_list_t *topo_node = NULL;

5061                        topo_node = kmem_zalloc(
5062                            sizeof (mptsas_topo_change_list_t),
5063                            KM_NOSLEEP);
5064                        if (topo_node == NULL) {
5065                                mptsas_log(mpt, CE_NOTE, "No memory"
5066                                    "resource for handle SAS dynamic"
5067                                    "reconfigure.\n");
5068                                break;
5069                        }
5070                        topo_node->mpt = mpt;
5071                        topo_node->event = MPTSAS_DR_EVENT_RECONFIG_TARGET;
5072                        topo_node->un.phymask = ptgt->m_phymask;
5073                        topo_node->devhdl = ptgt->m_devhdl;
5074                        topo_node->object = (void *)ptgt;
5075                        topo_node->flags = MPTSAS_TOPO_FLAG_LUN_ASSOCIATED;

5077                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
5078                            mptsas_handle_dr,
5079                            (void *)topo_node,
5080                            DDI_NOSLEEP)) != DDI_SUCCESS) {
5081                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq"
5082                                    "for handle SAS dynamic reconfigure"
5083                                    "failed. \n");
5084                        }
5085                }
5086                break;
5087        case MPI2_SCSI_STATUS_GOOD:
5088                switch (ioc_status & MPI2_IOCSTATUS_MASK) {
5089                case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
```

```
5090                        pkt->pkt_reason = CMD_DEV_GONE;
5091                        pkt->pkt_state |= STATE_GOT_BUS;
5220                        mutex_enter(&ptgt->m_tgt_intr_mutex);
5092                        if (ptgt->m_reset_delay == 0) {
5093                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5094                        }
5224                        mutex_exit(&ptgt->m_tgt_intr_mutex);
5095                        NDBG31(("lost disk for target%d, command:%x",
5096                            Tgt(cmd), pkt->pkt_cdbp[0]));
5097                        break;
5098                case MPI2_IOCSTATUS_SCSI_DATA_OVERRUN:
5099                        NDBG31(("data overrun: xferred=%d", xferred));
5100                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5101                        pkt->pkt_reason = CMD_DATA_OVR;
5102                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5103                            | STATE_SENT_CMD | STATE_GOT_STATUS
5104                            | STATE_XFERRED_DATA);
5105                        pkt->pkt_resid = 0;
5106                        break;
5107                case MPI2_IOCSTATUS_SCSI_RESIDUAL_MISMATCH:
5108                case MPI2_IOCSTATUS_SCSI_DATA_UNDERRUN:
5109                        NDBG31(("data underrun: xferred=%d", xferred));
5110                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5111                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5112                            | STATE_SENT_CMD | STATE_GOT_STATUS);
5113                        pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5114                        if (pkt->pkt_resid != cmd->cmd_dmacount) {
5115                                pkt->pkt_state |= STATE_XFERRED_DATA;
5116                        }
5117                        break;
5118                case MPI2_IOCSTATUS_SCSI_TASK_TERMINATED:
5119                        mptsas_set_pkt_reason(mpt,
5120                            cmd, CMD_RESET, STAT_BUS_RESET);
5121                        break;
5122                case MPI2_IOCSTATUS_SCSI_IOC_TERMINATED:
5123                case MPI2_IOCSTATUS_SCSI_EXT_TERMINATED:
5124                        mptsas_set_pkt_reason(mpt,
5125                            cmd, CMD_RESET, STAT_DEV_RESET);
5126                        break;
5127                case MPI2_IOCSTATUS_SCSI_IO_DATA_ERROR:
5128                case MPI2_IOCSTATUS_SCSI_PROTOCOL_ERROR:
5129                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET);
5130                        mptsas_set_pkt_reason(mpt,
5131                            cmd, CMD_TERMINATED, STAT_TERMINATED);
5132                        break;
5133                case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
5134                case MPI2_IOCSTATUS_BUSY:
5135                        /*
5136                         * set throttles to drain
5137                         */
5138                        ptgt = (mptsas_target_t *)mptsas_hash_traverse(
5139                            &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
5140                        while (ptgt != NULL) {
5271                                mutex_enter(&ptgt->m_tgt_intr_mutex);
5141                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5273                                mutex_exit(&ptgt->m_tgt_intr_mutex);

5143                                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
5144                                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
5145                        }

5147                        /*
5148                         * retry command
5149                         */
5150                        cmd->cmd_flags |= CFLAG_RETRY;
5151                        cmd->cmd_pkt_flags |= FLAG_HEAD;
```



```
5285                        mutex_exit(&mpt->m_mutex);
5153                        (void) mptsas_accept_pkt(mpt, cmd);
5287                        mutex_enter(&mpt->m_mutex);
5154                        break;
5155                default:
5156                        mptsas_log(mpt, CE_WARN,
5157                            "unknown ioc_status = %x\n", ioc_status);
5158                        mptsas_log(mpt, CE_CONT, "scsi_state = %x, transfer "
5159                            "count = %x, scsi_status = %x", scsi_state,
5160                            xferred, scsi_status);
5161                        break;
5162                }
5163                break;
5164        case MPI2_SCSI_STATUS_TASK_SET_FULL:
5165                mptsas_handle_qfull(mpt, cmd);
5166                break;
5167        case MPI2_SCSI_STATUS_BUSY:
5168                NDBG31(("scsi_status busy received"));
5169                break;
5170        case MPI2_SCSI_STATUS_RESERVATION_CONFLICT:
5171                NDBG31(("scsi_status reservation conflict received"));
5172                break;
5173        default:
5174                mptsas_log(mpt, CE_WARN, "scsi_status=%x, ioc_status=%x\n",
5175                    scsi_status, ioc_status);
5176                mptsas_log(mpt, CE_WARN,
5177                    "mptsas_process_intr: invalid scsi status\n");
5178                break;
5179        }
5180 }
______unchanged_portion_omitted_


5269 /*
5270  * mpt interrupt handler.
5271  */
5272 static uint_t
5273 mptsas_intr(caddr_t arg1, caddr_t arg2)
5274 {
5275        mptsas_t                        *mpt = (void *)arg1;
5276        pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
5277        uchar_t                         did_reply = FALSE;
5411        int                             i = 0, j;
5412        uint8_t                         reply_type;
5413        uint16_t                        SMID;

5279        NDBG1(("mptsas_intr: arg1 0x%p arg2 0x%p", (void *)arg1, (void *)arg2));

5281        mutex_enter(&mpt->m_mutex);
5417        /*
5418         * 1.
5419         * To avoid using m_mutex in the ISR(ISR referes not only mptsas_intr,
5420         * but all of the recursive called functions in it. the same below),
5421         * separate mutexs are introduced to protect the elements shown in ISR.
5422         * 3 type of mutex are involved here:
5423         *      a)per instance mutex m_intr_mutex.
5424         *      b)per target mutex m_tgt_intr_mutex.
5425         *      c)mutex that protect the free slot.
5426         *
5427         * a)per instance mutex m_intr_mutex:
5428         * used to protect m_options, m_power, m_waitq, etc that would be
5429         * checked/modified in ISR; protect the getting and setting the reply
5430         * descriptor index; protect the m_slots[];
5431         *
5432         * b)per target mutex m_tgt_intr_mutex:
```

```
5433          * used to protect per target element which has relationship to ISR.
5434          * contention for the new per target mutex is just as high as it in
5435          * sd(7d) driver.
5436          *
5437          * c)mutexs that protect the free slots:
5438          * those mutexs are introduced to minimize the mutex contentions
5439          * between the IO request threads where free slots are allocated
5440          * for sending cmds and ISR where slots holding outstanding cmds
5441          * are returned to the free pool.
5442          * the idea is like this:
5443          * 1) Partition all of the free slot into NCPU groups. For example,
5444          * In system where we have 15 slots, and 4 CPU, then slot s1,s5,s9,s13
5445          * are marked belonging to CPU1, s2,s6,s10,s14 to CPU2, s3,s7,s11,s15
5446          * to CPU3, and s4,s8,s12 to CPU4.
5447          * 2) In each of the group, an alloc/release queue pair is created,
5448          * and both the allocq and the releaseq have a dedicated mutex.
5449          * 3) When init, all of the slots in a CPU group are inserted into the
5450          * allocq of its CPU's pair.
5451          * 4) When doing IO,
5452          * mptsas_scsi_start()
5453          * {
5454          *      cpuid = the cpu NO of the cpu where this thread is running on
5455          * retry:
5456          *      mutex_enter(&allocq[cpuid]);
5457          *      if (get free slot = success) {
5458          *              remove the slot from the allocq
5459          *              mutex_exit(&allocq[cpuid]);
5460          *              return(success);
5461          *      } else { // exchange allocq and releaseq and try again
5462          *              mutex_enter(&releq[cpuid]);
5463          *              exchange the allocq and releaseq of this pair;
5464          *              mutex_exit(&releq[cpuid]);
5465          *              if (try to get free slot again = success) {
5466          *                      remove the slot from the allocq
5467          *                      mutex_exit(&allocq[cpuid]);
5468          *                      return(success);
5469          *              } else {
5470          *                      MOD(cpuid)++;
5471          *                      goto retry;
5472          *                      if (all CPU groups tried)
5473          *                              mutex_exit(&allocq[cpuid]);
5474          *                              return(failure);
5475          *              }
5476          *      }
5477          * }
5478          * ISR()
5479          * {
5480          *              cpuid = the CPU group id where the slot sending the
5481          *              cmd belongs;
5482          *              mutex_enter(&releq[cpuid]);
5483          *              remove the slot from the releaseq
5484          *              mutex_exit(&releq[cpuid]);
5485          * }
5486          * This way, only when the queue pair doing exchange have mutex
5487          * contentions.
5488          *
5489          * For mutex m_intr_mutex and m_tgt_intr_mutex, there are 2 scenarios:
5490          *
5491          * a)If the elements are only checked but not modified in the ISR, then
5492          * only the places where those elements are modifed(outside of ISR)
5493          * need to be protected by the new introduced mutex.
5494          * For example, data A is only read/checked in ISR, then we need do
5495          * like this:
5496          * In ISR:
5497          * {
5498          *      mutex_enter(&new_mutex);
```



```
5499          *      read(A);
5500          *      mutex_exit(&new_mutex);
5501          *      //the new_mutex here is either the m_tgt_intr_mutex or
5502          *      //the m_intr_mutex.
5503          * }
5504          * In non-ISR
5505          * {
5506          *      mutex_enter(&m_mutex); //the stock driver already did this
5507          *      mutex_enter(&new_mutex);
5508          *      write(A);
5509          *      mutex_exit(&new_mutex);
5510          *      mutex_exit(&m_mutex); //the stock driver already did this
5511          *
5512          *      read(A);
5513          *      // read(A) in non-ISR is not required to be protected by new
5514          *      // mutex since 'A' has already been protected by m_mutex
5515          *      // outside of the ISR
5516          * }
5517          *
5518          * Those fields in mptsas_target_t/ptgt which are only read in ISR
5519          * fall into this catergory. So they, together with the fields which
5520          * are never read in ISR, are not necessary to be protected by
5521          * m_tgt_intr_mutex, don't bother.
5522          * checking of m_waitq also falls into this catergory. so all of the
5523          * place outside of ISR where the m_waitq is modified, such as in
5524          * mptsas_waitq_add(), mptsas_waitq_delete(), mptsas_waitq_rm(),
5525          * m_intr_mutex should be used.
5526          *
5527          * b)If the elements are modified in the ISR, then each place where
5528          * those elements are referred(outside of ISR) need to be protected
5529          * by the new introduced mutex. Of course, if those elements only
5530          * appear in the non-key code path, that is, they don't affect
5531          * performance, then the m_mutex can still be used as before.
5532          * For example, data B is modified in key code path in ISR, and data C
5533          * is modified in non-key code path in ISR, then we can do like this:
5534          * In ISR:
5535          * {
5536          *      mutex_enter(&new_mutex);
5537          *      wirte(B);
5538          *      mutex_exit(&new_mutex);
5539          *      if (seldom happen) {
5540          *              mutex_enter(&m_mutex);
5541          *              write(C);
5542          *              mutex_exit(&m_mutex);
5543          *      }
5544          *      //the new_mutex here is either the m_tgt_intr_mutex or
5545          *      //the m_intr_mutex.
5546          * }
5547          * In non-ISR
5548          * {
5549          *      mutex_enter(&new_mutex);
5550          *      write(B);
5551          *      mutex_exit(&new_mutex);
5552          *
5553          *      mutex_enter(&new_mutex);
5554          *      read(B);
5555          *      mutex_exit(&new_mutex);
5556          *      // both write(B) and read(B) in non-ISR is required to be
5557          *      // protected by new mutex outside of the ISR
5558          *
5559          *      mutex_enter(&m_mutex); //the stock driver already did this
5560          *      read(C);
5561          *      write(C);
5562          *      mutex_exit(&m_mutex); //the stock driver already did this
5563          *      // both write(C) and read(C) in non-ISR have been already
5564          *      // been protected by m_mutex outside of the ISR
```

```
5565          * }
5566          *
5567          * For example, ptgt->m_t_ncmds fall into 'B' of this catergory, and
5568          * elements shown in address reply, restart_hba, passthrough, IOC
5569          * fall into 'C' of  this catergory.
5570          *
5571          * In any case where mutexs are nested, make sure in the following
5572          * order:
5573          *      m_mutex -> m_intr_mutex -> m_tgt_intr_mutex
5574          *      m_intr_mutex -> m_tgt_intr_mutex
5575          *      m_mutex -> m_intr_mutex
5576          *      m_mutex -> m_tgt_intr_mutex
5577          *
5578          * 2.
5579          * Make sure at any time, getting the ReplyDescriptor by m_post_index
5580          * and setting m_post_index to the ReplyDescriptorIndex register are
5581          * atomic. Since m_mutex is not used for this purpose in ISR, the new
5582          * mutex m_intr_mutex must play this role. So mptsas_poll(), where this
5583          * kind of getting/setting is also performed, must use m_intr_mutex.
5584          * Note, since context reply in ISR/process_intr is the only code path
5585          * which affect performance, a fast path is introduced to only handle
5586          * the read/write IO having context reply. For other IOs such as
5587          * passthrough and IOC with context reply and all address reply, we
5588          * use the as-is process_intr() to handle them. In order to keep the
5589          * same semantics in process_intr(), make sure any new mutex is not held
5590          * before enterring it.
5591          */

5593         mutex_enter(&mpt->m_intr_mutex);

5283         /*
5284          * If interrupts are shared by two channels then check whether this
5285          * interrupt is genuinely for this channel by making sure first the
5286          * chip is in high power state.
5287          */
5288         if ((mpt->m_options & MPTSAS_OPT_PM) &&
5289             (mpt->m_power_level != PM_LEVEL_D0)) {
5290                 mutex_exit(&mpt->m_mutex);
5602                 mutex_exit(&mpt->m_intr_mutex);
5291                 return (DDI_INTR_UNCLAIMED);
5292         }

5294         /*
5295          * If polling, interrupt was triggered by some shared interrupt because
5296          * IOC interrupts are disabled during polling, so polling routine will
5297          * handle any replies.  Considering this, if polling is happening,
5298          * return with interrupt unclaimed.
5299          */
5300         if (mpt->m_polled_intr) {
5301                 mutex_exit(&mpt->m_mutex);
5613                 mutex_exit(&mpt->m_intr_mutex);
5302                 mptsas_log(mpt, CE_WARN, "mpt_sas: Unclaimed interrupt");
5303                 return (DDI_INTR_UNCLAIMED);
5304         }

5306         /*
5307          * Read the istat register.
5308          */
5309         if ((INTPENDING(mpt)) != 0) {
5310                 /*
5311                  * read fifo until empty.
5312                  */
5313 #ifndef __lock_lint
5314                 _NOTE(CONSTCOND)
5315 #endif
5316                 while (TRUE) {
```



```
5317                         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5318                             DDI_DMA_SYNC_FORCPU);
5319                         reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
5320                             MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

5322                         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5323                             &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5324                             ddi_get32(mpt->m_acc_post_queue_hdl,
5325                             &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5326                                 break;
5327                         }

5329                         /*
5330                          * The reply is valid, process it according to its
5331                          * type.  Also, set a flag for updating the reply index
5332                          * after they've all been processed.
5333                          */
5334                         did_reply = TRUE;

5336                         mptsas_process_intr(mpt, reply_desc_union);
5648                         reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5649                             &reply_desc_union->Default.ReplyFlags);
5650                         reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
5651                         mpt->m_reply[i].Default.ReplyFlags = reply_type;
5652                         if (reply_type ==
5653                             MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS) {
5654                                 SMID = ddi_get16(mpt->m_acc_post_queue_hdl,
5655                                     &reply_desc_union->SCSIIOSuccess.SMID);
5656                                 if (mptsas_handle_io_fastpath(mpt, SMID) !=
5657                                     TRUE) {
5658                                         mpt->m_reply[i].SCSIIOSuccess.SMID =
5659                                             SMID;
5660                                         i++;
5661                                 }
5662                         } else if (reply_type ==
5663                             MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
5664                                 mpt->m_reply[i].AddressReply.ReplyFrameAddress =
5665                                     ddi_get32(mpt->m_acc_post_queue_hdl,
5666                                     &reply_desc_union->AddressReply.
5667                                     ReplyFrameAddress);
5668                                 mpt->m_reply[i].AddressReply.SMID =
5669                                     ddi_get16(mpt->m_acc_post_queue_hdl,
5670                                     &reply_desc_union->AddressReply.SMID);
5671                                 i++;
5672                         }
5673                         /*
5674                          * Clear the reply descriptor for re-use and increment
5675                          * index.
5676                          */
5677                         ddi_put64(mpt->m_acc_post_queue_hdl,
5678                             &((uint64_t *)(void *)mpt->m_post_queue)
5679                             [mpt->m_post_index], 0xFFFFFFFFFFFFFFFF);
5680                         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5681                             DDI_DMA_SYNC_FORDEV);

5338                         /*
5339                          * Increment post index and roll over if needed.
5340                          */
5341                         if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5342                                 mpt->m_post_index = 0;
5343                         }
5689                         if (i >= MPI_ADDRESS_COALSCE_MAX)
5690                                 break;
5344                 }

5346                 /*
```

```
5347                  * Update the global reply index if at least one reply was
5348                  * processed.
5349                  */
5350                 if (did_reply) {
5351                         ddi_put32(mpt->m_datap,
5352                             &mpt->m_reg->ReplyPostHostIndex, mpt->m_post_index);

5701                         /*
5702                          * For fma, only check the PIO is required and enough
5703                          * here. Those cases where fastpath is not hit, the
5704                          * mptsas_fma_check() check all of the types of
5705                          * fma. That is not necessary and sometimes not
5706                          * correct. fma check should only be done after
5707                          * the PIO and/or dma is performed.
5708                          */
5709                         if ((mptsas_check_acc_handle(mpt->m_datap) !=
5710                             DDI_SUCCESS)) {
5711                                 ddi_fm_service_impact(mpt->m_dip,
5712                                     DDI_SERVICE_UNAFFECTED);
5353                 }

5715                 }
5354         } else {
5355                 mutex_exit(&mpt->m_mutex);
5717                 mutex_exit(&mpt->m_intr_mutex);
5356                 return (DDI_INTR_UNCLAIMED);
5357         }
5358         NDBG1(("mptsas_intr complete"));
5721         mutex_exit(&mpt->m_intr_mutex);

5360         /*
5724          * Since most of the cmds(read and write IO with success return.)
5725          * have already been processed in fast path in which the m_mutex
5726          * is not held, handling here the address reply and other context reply
5727          * such as passthrough and IOC cmd with m_mutex held should be a big
5728          * issue for performance.
5729          * If holding m_mutex to process these cmds was still an obvious issue,
5730          * we can process them in a taskq.
5731          */
5732         for (j = 0; j < i; j++) {
5733                 mutex_enter(&mpt->m_mutex);
5734                 mptsas_process_intr(mpt, &mpt->m_reply[j]);
5735                 mutex_exit(&mpt->m_mutex);
5736         }

5738         /*
5361          * If no helper threads are created, process the doneq in ISR. If
5362          * helpers are created, use the doneq length as a metric to measure the
5363          * load on the interrupt CPU. If it is long enough, which indicates the
5364          * load is heavy, then we deliver the IO completions to the helpers.
5365          * This measurement has some limitations, although it is simple and
5366          * straightforward and works well for most of the cases at present.
5367          */
5368         if (!mpt->m_doneq_thread_n ||
5369             (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)) {
5746         if (!mpt->m_doneq_thread_n) {
5370                 mptsas_doneq_empty(mpt);
5371         } else {
5749                 int helper = 1;
5750                 mutex_enter(&mpt->m_intr_mutex);
5751                 if (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)
5752                         helper = 0;
5753                 mutex_exit(&mpt->m_intr_mutex);
5754                 if (helper) {
5372                 mptsas_deliver_doneq_thread(mpt);
5756                 } else {
```



```
5757                         mptsas_doneq_empty(mpt);
5373         }
5759         }

5375         /*
5376          * If there are queued cmd, start them now.
5377          */
5764         mutex_enter(&mpt->m_intr_mutex);
5378         if (mpt->m_waitq != NULL) {
5379                 mptsas_restart_waitq(mpt);
5380         }

5766                 mutex_exit(&mpt->m_intr_mutex);
5767                 mutex_enter(&mpt->m_mutex);
5768                 mptsas_restart_hba(mpt);
5382         mutex_exit(&mpt->m_mutex);
5383         return (DDI_INTR_CLAIMED);
5771         }
5772         mutex_exit(&mpt->m_intr_mutex);
5773         return (DDI_INTR_CLAIMED);
5384 }

5776 /*
5777  * In ISR, the successfully completed read and write IO are processed in a
5778  * fast path. This function is only used to handle non-fastpath IO, including
5779  * all of the address reply, and the context reply for IOC cmd, passthrough,
5780  * etc.
5781  * This function is also used to process polled cmd.
5782  */
5386 static void
5387 mptsas_process_intr(mptsas_t *mpt,
5388     pMpi2ReplyDescriptorsUnion_t reply_desc_union)
5389 {
5390         uint8_t reply_type;

5392         ASSERT(mutex_owned(&mpt->m_mutex));

5394         /*
5395          * The reply is valid, process it according to its
5396          * type.  Also, set a flag for updated the reply index
5397          * after they've all been processed.
5398          */
5399         reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5400             &reply_desc_union->Default.ReplyFlags);
5401         reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
5794         reply_type = reply_desc_union->Default.ReplyFlags;
5402         if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS) {
5403                 mptsas_handle_scsi_io_success(mpt, reply_desc_union);
5404         } else if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
5405                 mptsas_handle_address_reply(mpt, reply_desc_union);
5406         } else {
5407                 mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
5408                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5409         }

5411         /*
5412          * Clear the reply descriptor for re-use and increment
5413          * index.
5414          */
5415         ddi_put64(mpt->m_acc_post_queue_hdl,
5416             &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
5417             0xFFFFFFFFFFFFFFFF);
5418         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5419             DDI_DMA_SYNC_FORDEV);
5420 }
```

```
5422 /*
5423  * handle qfull condition
5424  */
5425 static void
5426 mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd)
5427 {
5428         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

5430         if ((++cmd->cmd_qfull_retries > ptgt->m_qfull_retries) ||
5431             (ptgt->m_qfull_retries == 0)) {
5432                 /*
5433                  * We have exhausted the retries on QFULL, or,
5434                  * the target driver has indicated that it
5435                  * wants to handle QFULL itself by setting
5436                  * qfull-retries capability to 0. In either case
5437                  * we want the target driver's QFULL handling
5438                  * to kick in. We do this by having pkt_reason
5439                  * as CMD_CMPLT and pkt_scbp as STATUS_QFULL.
5440                  */
5824                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5441                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5826                 mutex_exit(&ptgt->m_tgt_intr_mutex);
5442         } else {
5828                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5443                 if (ptgt->m_reset_delay == 0) {
5444                         ptgt->m_t_throttle =
5445                             max((ptgt->m_t_ncmds - 2), 0);
5446                 }
5833                 mutex_exit(&ptgt->m_tgt_intr_mutex);

5448                 cmd->cmd_pkt_flags |= FLAG_HEAD;
5449                 cmd->cmd_flags &= ~(CFLAG_TRANFLAG);
5450                 cmd->cmd_flags |= CFLAG_RETRY;

5839                 mutex_exit(&mpt->m_mutex);
5452                 (void) mptsas_accept_pkt(mpt, cmd);
5841                 mutex_enter(&mpt->m_mutex);

5454                 /*
5455                  * when target gives queue full status with no commands
5456                  * outstanding (m_t_ncmds == 0), throttle is set to 0
5457                  * (HOLD_THROTTLE), and the queue full handling start
5458                  * (see psarc/1994/313); if there are commands outstanding,
5459                  * throttle is set to (m_t_ncmds - 2)
5460                  */
5850                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5461                 if (ptgt->m_t_throttle == HOLD_THROTTLE) {
5462                         /*
5463                          * By setting throttle to QFULL_THROTTLE, we
5464                          * avoid submitting new commands and in
5465                          * mptsas_restart_cmd find out slots which need
5466                          * their throttles to be cleared.
5467                          */
5468                         mptsas_set_throttle(mpt, ptgt, QFULL_THROTTLE);
5469                         if (mpt->m_restart_cmd_timeid == 0) {
5470                                 mpt->m_restart_cmd_timeid =
5471                                     timeout(mptsas_restart_cmd, mpt,
5472                                     ptgt->m_qfull_retry_interval);
5473                         }
5474                 }
5865                 mutex_exit(&ptgt->m_tgt_intr_mutex);
5475         }
5476 }
```

*unchanged_portion_omitted_*

```
5606 /*
```



```
5607  * mptsas_handle_dr is a task handler for DR, the DR action includes:
5608  * 1. Directly attched Device Added/Removed.
5609  * 2. Expander Device Added/Removed.
5610  * 3. Indirectly Attached Device Added/Expander.
5611  * 4. LUNs of a existing device status change.
5612  * 5. RAID volume created/deleted.
5613  * 6. Member of RAID volume is released because of RAID deletion.
5614  * 7. Physical disks are removed because of RAID creation.
5615  */
5616 static void
5617 mptsas_handle_dr(void *args) {
5618         mptsas_topo_change_list_t       *topo_node = NULL;
5619         mptsas_topo_change_list_t       *save_node = NULL;
5620         mptsas_t                        *mpt;
5621         dev_info_t                      *parent = NULL;
5622         mptsas_phymask_t                phymask = 0;
5623         char                            *phy_mask_name;
5624         uint8_t                         flags = 0, physport = 0xff;
5625         uint8_t                         port_update = 0;
5626         uint_t                          event;

5628         topo_node = (mptsas_topo_change_list_t *)args;

5630         mpt = topo_node->mpt;
5631         event = topo_node->event;
5632         flags = topo_node->flags;

5634         phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);

5636         NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));

5638         switch (event) {
5639         case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5640                 if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5641                     (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
5642                     (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
5643                         /*
5644                          * Direct attached or expander attached device added
5645                          * into system or a Phys Disk that is being unhidden.
5646                          */
5647                         port_update = 1;
5648                 }
5649                 break;
5650         case MPTSAS_DR_EVENT_RECONFIG_SMP:
5651                 /*
5652                  * New expander added into system, it must be the head
5653                  * of topo_change_list_t
5654                  */
5655                 port_update = 1;
5656                 break;
5657         default:
5658                 port_update = 0;
5659                 break;
5660         }
5661         /*
5662          * All cases port_update == 1 may cause initiator port form change
5663          */
5664         mutex_enter(&mpt->m_mutex);
5665         if (mpt->m_port_chng && port_update) {
5666                 /*
5667                  * mpt->m_port_chng flag indicates some PHYs of initiator
5668                  * port have changed to online. So when expander added or
5669                  * directly attached device online event come, we force to
5670                  * update port information by issueing SAS IO Unit Page and
5671                  * update PHYMASKs.
5672                  */
```

```
5673                (void) mptsas_update_phymask(mpt);
5674                mpt->m_port_chng = 0;

5676        }
5677        mutex_exit(&mpt->m_mutex);
5678        while (topo_node) {
5679                phymask = 0;
5680                if (parent == NULL) {
5681                        physport = topo_node->un.physport;
5682                        event = topo_node->event;
5683                        flags = topo_node->flags;
5684                        if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
5685                            MPTSAS_DR_EVENT_OFFLINE_SMP)) {
5686                                /*
5687                                 * For all offline events, phymask is known
5688                                 */
5689                                phymask = topo_node->un.phymask;
5690                                goto find_parent;
5691                        }
5692                        if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
5693                                goto handle_topo_change;
5694                        }
5695                        if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
5696                                phymask = topo_node->un.phymask;
5697                                goto find_parent;
5698                        }

5700                        if ((flags ==
5701                            MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
5702                            (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
5703                                /*
5704                                 * There is no any field in IR_CONFIG_CHANGE
5705                                 * event indicate physport/phynum, let's get
5706                                 * parent after SAS Device Page0 request.
5707                                 */
5708                                goto handle_topo_change;
5709                        }

5711                        mutex_enter(&mpt->m_mutex);
5712                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5713                                /*
5714                                 * If the direct attached device added or a
5715                                 * phys disk is being unhidden, argument
5716                                 * physport actually is PHY#, so we have to get
5717                                 * phymask according PHY#.
5718                                 */
5719                                physport = mpt->m_phy_info[physport].port_num;
5720                        }

5722                        /*
5723                         * Translate physport to phymask so that we can search
5724                         * parent dip.
5725                         */
5726                        phymask = mptsas_physport_to_phymask(mpt,
5727                            physport);
5728                        mutex_exit(&mpt->m_mutex);

5730 find_parent:
5731                        bzero(phy_mask_name, MPTSAS_MAX_PHYS);
5732                        /*
5733                         * For RAID topology change node, write the iport name
5734                         * as v0.
5735                         */
5736                        if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5737                                (void) sprintf(phy_mask_name, "v0");
5738                        } else {
```



```
5739                                /*
5740                                 * phymask can bo 0 if the drive has been
5741                                 * pulled by the time an add event is
5742                                 * processed.  If phymask is 0, just skip this
5743                                 * event and continue.
5744                                 */
5745                                if (phymask == 0) {
5746                                        mutex_enter(&mpt->m_mutex);
5747                                        save_node = topo_node;
5748                                        topo_node = topo_node->next;
5749                                        ASSERT(save_node);
5750                                        kmem_free(save_node,
5751                                            sizeof (mptsas_topo_change_list_t));
5752                                        mutex_exit(&mpt->m_mutex);

5754                                        parent = NULL;
5755                                        continue;
5756                                }
5757                                (void) sprintf(phy_mask_name, "%x", phymask);
5758                        }
5759                        parent = scsi_hba_iport_find(mpt->m_dip,
5760                            phy_mask_name);
5761                        if (parent == NULL) {
5762                                mptsas_log(mpt, CE_WARN, "Failed to find an "
5763                                    "iport, should not happen!");
5764                                goto out;
5765                        }

5767                }
5768                ASSERT(parent);
5769 handle_topo_change:

5771                mutex_enter(&mpt->m_mutex);
5772                /*
5773                 * If HBA is being reset, don't perform operations depending
5774                 * on the IOC. We must free the topo list, however.
5775                 */
5776                if (!mpt->m_in_reset)

5777                        mptsas_handle_topo_change(topo_node, parent);
5778                else
5779                        NDBG20(("skipping topo change received during reset"));
5780                save_node = topo_node;
5781                topo_node = topo_node->next;
5782                ASSERT(save_node);
5783                kmem_free(save_node, sizeof (mptsas_topo_change_list_t));
5784                mutex_exit(&mpt->m_mutex);

5786                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5787                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
5788                    (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) {
5789                        /*
5790                         * If direct attached device associated, make sure
5791                         * reset the parent before start the next one. But
5792                         * all devices associated with expander shares the
5793                         * parent.  Also, reset parent if this is for RAID.
5794                         */
5795                        parent = NULL;
5796                }
5797        }
5798 out:
5799        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5800 }

5802 static void
5803 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
```

```
5804     dev_info_t *parent)
5805 {
5806         mptsas_target_t *ptgt = NULL;
5807         mptsas_smp_t    *psmp = NULL;
5808         mptsas_t        *mpt = (void *)topo_node->mpt;
5809         uint16_t        devhdl;
5810         uint16_t        attached_devhdl;
5811         uint64_t        sas_wwn = 0;
5812         int             rval = 0;
5813         uint32_t        page_address;
5814         uint8_t         phy, flags;
5815         char            *addr = NULL;
5816         dev_info_t      *lundip;
5817         int             circ = 0, circ1 = 0;
5818         char            attached_wwnstr[MPTSAS_WWN_STRLEN];

5820         NDBG20(("mptsas%d handle_topo_change enter", mpt->m_instance));

5822         ASSERT(mutex_owned(&mpt->m_mutex));

5824         switch (topo_node->event) {
5825         case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5826         {
5827                 char *phy_mask_name;
5828                 mptsas_phymask_t phymask = 0;

5830                 if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5831                         /*
5832                          * Get latest RAID info.
5833                          */
5834                         (void) mptsas_get_raid_info(mpt);
5835                         ptgt = mptsas_search_by_devhdl(
5836                             &mpt->m_active->m_tgttbl, topo_node->devhdl);
5837                         if (ptgt == NULL)
5838                                 break;
5839                 } else {
5840                         ptgt = (void *)topo_node->object;
5841                 }

5843                 if (ptgt == NULL) {
5844                         /*
5845                          * If a Phys Disk was deleted, RAID info needs to be
5846                          * updated to reflect the new topology.
5847                          */
5848                         (void) mptsas_get_raid_info(mpt);

5850                         /*
5851                          * Get sas device page 0 by DevHandle to make sure if
5852                          * SSP/SATA end device exist.
5853                          */
5854                         page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
5855                             MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
5856                             topo_node->devhdl;

5858                         rval = mptsas_get_target_device_info(mpt, page_address,
5859                             &devhdl, &ptgt);
5860                         if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
5861                                 mptsas_log(mpt, CE_NOTE,
5862                                     "mptsas_handle_topo_change: target %d is "
5863                                     "not a SAS/SATA device. \n",
5864                                     topo_node->devhdl);
5865                         } else if (rval == DEV_INFO_FAIL_ALLOC) {
5866                                 mptsas_log(mpt, CE_NOTE,
5867                                     "mptsas_handle_topo_change: could not "
5868                                     "allocate memory. \n");
5869                         }
```



```
5870                         /*
5871                          * If rval is DEV_INFO_PHYS_DISK than there is nothing
5872                          * else to do, just leave.
5873                          */
5874                         if (rval != DEV_INFO_SUCCESS) {
5875                                 return;
5876                         }
5877                 }

5879                 ASSERT(ptgt->m_devhdl == topo_node->devhdl);

5881                 mutex_exit(&mpt->m_mutex);
5882                 flags = topo_node->flags;

5884                 if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
5885                         phymask = ptgt->m_phymask;
5886                         phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
5887                         (void) sprintf(phy_mask_name, "%x", phymask);
5888                         parent = scsi_hba_iport_find(mpt->m_dip,
5889                             phy_mask_name);
5890                         kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5891                         if (parent == NULL) {
5892                                 mptsas_log(mpt, CE_WARN, "Failed to find a "
5893                                     "iport for PD, should not happen!");
5894                                 mutex_enter(&mpt->m_mutex);
5895                                 break;
5896                         }
5897                 }

5899                 if (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5900                         ndi_devi_enter(parent, &circ1);
5901                         (void) mptsas_config_raid(parent, topo_node->devhdl,
5902                             &lundip);
5903                         ndi_devi_exit(parent, circ1);
5904                 } else {
5905                         /*
5906                          * hold nexus for bus configure
5907                          */
5908                         ndi_devi_enter(scsi_vhci_dip, &circ);
5909                         ndi_devi_enter(parent, &circ1);
5910                         rval = mptsas_config_target(parent, ptgt);
5911                         /*
5912                          * release nexus for bus configure
5913                          */
5914                         ndi_devi_exit(parent, circ1);
5915                         ndi_devi_exit(scsi_vhci_dip, circ);

5917                         /*
5918                          * Add parent's props for SMHBA support
5919                          */
5920                         if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5921                                 bzero(attached_wwnstr,
5922                                     sizeof (attached_wwnstr));
5923                                 (void) sprintf(attached_wwnstr, "w%016"PRIx64,
5924                                     ptgt->m_sas_wwn);
5925                                 if (ddi_prop_update_string(DDI_DEV_T_NONE,
5926                                     parent,
5927                                     SCSI_ADDR_PROP_ATTACHED_PORT,
5928                                     attached_wwnstr)
5929                                     != DDI_PROP_SUCCESS) {
5930                                         (void) ddi_prop_remove(DDI_DEV_T_NONE,
5931                                             parent,
5932                                             SCSI_ADDR_PROP_ATTACHED_PORT);
5933                                         mptsas_log(mpt, CE_WARN, "Failed to"
5934                                             "attached-port props");
5935                                         return;
```

```
5936                                }
5937                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5938                                    MPTSAS_NUM_PHYS, 1) !=
5939                                    DDI_PROP_SUCCESS) {
5940                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5941                                            parent, MPTSAS_NUM_PHYS);
5942                                        mptsas_log(mpt, CE_WARN, "Failed to"
5943                                            " create num-phys props");
5944                                        return;
5945                                }

5947                                /*
5948                                 * Update PHY info for smhba
5949                                 */
5950                                mutex_enter(&mpt->m_mutex);
5951                                if (mptsas_smhba_phy_init(mpt)) {
5952                                        mutex_exit(&mpt->m_mutex);
5953                                        mptsas_log(mpt, CE_WARN, "mptsas phy"
5954                                            " update failed");
5955                                        return;
5956                                }
5957                                mutex_exit(&mpt->m_mutex);
5958                                mptsas_smhba_set_phy_props(mpt,
5959                                    ddi_get_name_addr(parent), parent,
5960                                    1, &attached_devhdl);
5961                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5962                                    MPTSAS_VIRTUAL_PORT, 0) !=
5963                                    DDI_PROP_SUCCESS) {
5964                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5965                                            parent, MPTSAS_VIRTUAL_PORT);
5966                                        mptsas_log(mpt, CE_WARN,
5967                                            "mptsas virtual-port"
5968                                            "port prop update failed");
5969                                        return;
5970                                }
5971                        }
5972                }
5973                mutex_enter(&mpt->m_mutex);

5975                NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
5976                    "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
5977                    ptgt->m_phymask));
5978                break;
5979        }
5980        case MPTSAS_DR_EVENT_OFFLINE_TARGET:
5981        {
5982                mptsas_hash_table_t *tgttbl = &mpt->m_active->m_tgttbl;
5983                devhdl = topo_node->devhdl;
5984                ptgt = mptsas_search_by_devhdl(tgttbl, devhdl);
5985                if (ptgt == NULL)
5986                        break;

5988                sas_wwn = ptgt->m_sas_wwn;
5989                phy = ptgt->m_phynum;

5991                addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

5993                if (sas_wwn) {
5994                        (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
5995                } else {
5996                        (void) sprintf(addr, "p%x", phy);
5997                }
5998                ASSERT(ptgt->m_devhdl == devhdl);

6000                if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
6001                    (topo_node->flags ==
```



```
6002                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6003                        /*
6004                         * Get latest RAID info if RAID volume status changes
6005                         * or Phys Disk status changes
6006                         */
6007                        (void) mptsas_get_raid_info(mpt);
6008                }
6009                /*
6010                 * Abort all outstanding command on the device
6011                 */
6012                rval = mptsas_do_scsi_reset(mpt, devhdl);
6013                if (rval) {
6014                        NDBG20(("mptsas%d handle_topo_change to reset target "
6015                            "before offline devhdl:%x, phymask:%x, rval:%x",
6016                            mpt->m_instance, ptgt->m_devhdl, ptgt->m_phymask,
6017                            rval));
6018                }

6020                mutex_exit(&mpt->m_mutex);

6022                ndi_devi_enter(scsi_vhci_dip, &circ);
6023                ndi_devi_enter(parent, &circ1);
6024                rval = mptsas_offline_target(parent, addr);
6025                ndi_devi_exit(parent, circ1);
6026                ndi_devi_exit(scsi_vhci_dip, circ);
6027                NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6028                    "phymask:%x, rval:%x", mpt->m_instance,
6029                    ptgt->m_devhdl, ptgt->m_phymask, rval));

6031                kmem_free(addr, SCSI_MAXNAMELEN);

6033                /*
6034                 * Clear parent's props for SMHBA support
6035                 */
6036                flags = topo_node->flags;
6037                if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6038                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6039                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6040                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6041                            DDI_PROP_SUCCESS) {
6042                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6043                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6044                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6045                                    "prop update failed");
6046                                break;
6047                        }
6048                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6049                            MPTSAS_NUM_PHYS, 0) !=
6050                            DDI_PROP_SUCCESS) {
6051                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6052                                    MPTSAS_NUM_PHYS);
6053                                mptsas_log(mpt, CE_WARN, "mptsas num phys "
6054                                    "prop update failed");
6055                                break;
6056                        }
6057                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6058                            MPTSAS_VIRTUAL_PORT, 1) !=
6059                            DDI_PROP_SUCCESS) {
6060                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6061                                    MPTSAS_VIRTUAL_PORT);
6062                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6063                                    "prop update failed");
6064                                break;
6065                        }
6066                }
```

```
6068                mutex_enter(&mpt->m_mutex);
6454                if (mptsas_set_led_status(mpt, ptgt, 0) != DDI_SUCCESS) {
6455                        NDBG14(("mptsas: clear LED for tgt %x failed",
6456                            ptgt->m_slot_num));
6457                }
6069                if (rval == DDI_SUCCESS) {
6070                        mptsas_tgt_free(&mpt->m_active->m_tgttbl,
6071                            ptgt->m_sas_wwn, ptgt->m_phymask);
6072                        ptgt = NULL;
6073                } else {
6074                        /*
6075                         * clean DR_INTRANSITION flag to allow I/O down to
6076                         * PHCI driver since failover finished.
6077                         * Invalidate the devhdl
6078                         */
6468                        mutex_enter(&ptgt->m_tgt_intr_mutex);
6079                        ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6080                        ptgt->m_tgt_unconfigured = 0;
6081                        mutex_enter(&mpt->m_tx_waitq_mutex);
6082                        ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
6083                        mutex_exit(&mpt->m_tx_waitq_mutex);
6472                        mutex_exit(&ptgt->m_tgt_intr_mutex);
6084                }

6086                /*
6087                 * Send SAS IO Unit Control to free the dev handle
6088                 */
6089                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6090                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
6091                        rval = mptsas_free_devhdl(mpt, devhdl);

6093                        NDBG20(("mptsas%d handle_topo_change to remove "
6094                            "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6095                            rval));
6096                }

6098                break;
6099        }
6100        case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
6101        {
6102                devhdl = topo_node->devhdl;
6103                /*
6104                 * If this is the remove handle event, do a reset first.
6105                 */
6106                if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6107                        rval = mptsas_do_scsi_reset(mpt, devhdl);
6108                        if (rval) {
6109                                NDBG20(("mpt%d reset target before remove "
6110                                    "devhdl:%x, rval:%x", mpt->m_instance,
6111                                    devhdl, rval));
6112                        }
6113                }

6115                /*
6116                 * Send SAS IO Unit Control to free the dev handle
6117                 */
6118                rval = mptsas_free_devhdl(mpt, devhdl);
6119                NDBG20(("mptsas%d handle_topo_change to remove "
6120                    "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6121                    rval));
6122                break;
6123        }
6124        case MPTSAS_DR_EVENT_RECONFIG_SMP:
6125        {
6126                mptsas_smp_t smp;
6127                dev_info_t *smpdip;
```



```
6128                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;

6130                devhdl = topo_node->devhdl;

6132                page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
6133                    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
6134                rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
6135                if (rval != DDI_SUCCESS) {
6136                        mptsas_log(mpt, CE_WARN, "failed to online smp, "
6137                            "handle %x", devhdl);
6138                        return;
6139                }

6141                psmp = mptsas_smp_alloc(smptbl, &smp);
6142                if (psmp == NULL) {
6143                        return;
6144                }

6146                mutex_exit(&mpt->m_mutex);
6147                ndi_devi_enter(parent, &circ1);
6148                (void) mptsas_online_smp(parent, psmp, &smpdip);
6149                ndi_devi_exit(parent, circ1);

6151                mutex_enter(&mpt->m_mutex);
6152                break;
6153        }
6154        case MPTSAS_DR_EVENT_OFFLINE_SMP:
6155        {
6156                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;
6157                devhdl = topo_node->devhdl;
6158                uint32_t dev_info;

6160                psmp = mptsas_search_by_devhdl(smptbl, devhdl);
6161                if (psmp == NULL)
6162                        break;
6163                /*
6164                 * The mptsas_smp_t data is released only if the dip is offlined
6165                 * successfully.
6166                 */
6167                mutex_exit(&mpt->m_mutex);

6169                ndi_devi_enter(parent, &circ1);
6170                rval = mptsas_offline_smp(parent, psmp, NDI_DEVI_REMOVE);
6171                ndi_devi_exit(parent, circ1);

6173                dev_info = psmp->m_deviceinfo;
6174                if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
6175                    DEVINFO_DIRECT_ATTACHED) {
6176                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6177                            MPTSAS_VIRTUAL_PORT, 1) !=
6178                            DDI_PROP_SUCCESS) {
6179                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6180                                    MPTSAS_VIRTUAL_PORT);
6181                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6182                                    "prop update failed");
6183                                return;
6184                        }
6185                        /*
6186                         * Check whether the smp connected to the iport,
6187                         */
6188                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6189                            MPTSAS_NUM_PHYS, 0) !=
6190                            DDI_PROP_SUCCESS) {
6191                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6192                                    MPTSAS_NUM_PHYS);
6193                                mptsas_log(mpt, CE_WARN, "mptsas num phys"
```

```
6194                                    "prop update failed");
6195                                return;
6196                        }
6197                        /*
6198                         * Clear parent's attached-port props
6199                         */
6200                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6201                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6202                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6203                            DDI_PROP_SUCCESS) {
6204                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6205                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6206                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6207                                    "prop update failed");
6208                                return;
6209                        }
6210                }

6212                mutex_enter(&mpt->m_mutex);
6213                NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6214                    "rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
6215                if (rval == DDI_SUCCESS) {
6216                        mptsas_smp_free(smptbl, psmp->m_sasaddr,
6217                            psmp->m_phymask);
6218                } else {
6219                        psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
6220                }

6222                bzero(attached_wwnstr, sizeof (attached_wwnstr));

6224                break;
6225        }
6226        default:
6227                return;
6228        }
6229 }
```

*unchanged_portion_omitted_*

```
6315 #define SMP_RESET_IN_PROGRESS MPI2_EVENT_SAS_TOPO_LR_SMP_RESET_IN_PROGRESS
6316 /*
6317  * handle sync events from ioc in interrupt
6318  * return value:
6319  * DDI_SUCCESS: The event is handled by this func
6320  * DDI_FAILURE: Event is not handled
6321  */
6322 static int
6323 mptsas_handle_event_sync(void *args)
6324 {
6325        m_replyh_arg_t                  *replyh_arg;
6326        pMpi2EventNotificationReply_t   eventreply;
6327        uint32_t                        event, rfm;
6328        mptsas_t                        *mpt;
6329        uint_t                          iocstatus;

6331        replyh_arg = (m_replyh_arg_t *)args;
6332        rfm = replyh_arg->rfm;
6333        mpt = replyh_arg->mpt;

6335        ASSERT(mutex_owned(&mpt->m_mutex));

6337        eventreply = (pMpi2EventNotificationReply_t)
6338            (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
6339        event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6341        if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
6342            &eventreply->IOCStatus)) {
```



```
6343                if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
6344                        mptsas_log(mpt, CE_WARN,
6345                            "!mptsas_handle_event_sync: IOCStatus=0x%x, "
6346                            "IOCLogInfo=0x%x", iocstatus,
6347                            ddi_get32(mpt->m_acc_reply_frame_hdl,
6348                            &eventreply->IOCLogInfo));
6349                } else {
6350                        mptsas_log(mpt, CE_WARN,
6351                            "mptsas_handle_event_sync: IOCStatus=0x%x, "
6352                            "IOCLogInfo=0x%x", iocstatus,
6353                            ddi_get32(mpt->m_acc_reply_frame_hdl,
6354                            &eventreply->IOCLogInfo));
6355                }
6356        }

6358        /*
6359         * figure out what kind of event we got and handle accordingly
6360         */
6361        switch (event) {
6362        case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
6363        {
6364                pMpi2EventDataSasTopologyChangeList_t   sas_topo_change_list;
6365                uint8_t                         num_entries, expstatus, phy;
6366                uint8_t                         phystatus, physport, state, i;
6367                uint8_t                         start_phy_num, link_rate;
6368                uint16_t                        dev_handle, reason_code;
6369                uint16_t                        enc_handle, expd_handle;
6370                char                            string[80], curr[80], prev[80];
6371                mptsas_topo_change_list_t       *topo_head = NULL;
6372                mptsas_topo_change_list_t       *topo_tail = NULL;
6373                mptsas_topo_change_list_t       *topo_node = NULL;
6374                mptsas_target_t                 *ptgt;
6375                mptsas_smp_t                    *psmp;
6376                mptsas_hash_table_t             *tgttbl, *smptbl;
6377                uint8_t                         flags = 0, exp_flag;
6378                smhba_info_t                    *pSmhba = NULL;

6380                NDBG20(("mptsas_handle_event_sync: SAS topology change"));

6382                tgttbl = &mpt->m_active->m_tgttbl;
6383                smptbl = &mpt->m_active->m_smptbl;

6385                sas_topo_change_list = (pMpi2EventDataSasTopologyChangeList_t)
6386                    eventreply->EventData;

6388                enc_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6389                    &sas_topo_change_list->EnclosureHandle);
6390                expd_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6391                    &sas_topo_change_list->ExpanderDevHandle);
6392                num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
6393                    &sas_topo_change_list->NumEntries);
6394                start_phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
6395                    &sas_topo_change_list->StartPhyNum);
6396                expstatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
6397                    &sas_topo_change_list->ExpStatus);
6398                physport = ddi_get8(mpt->m_acc_reply_frame_hdl,
6399                    &sas_topo_change_list->PhysicalPort);

6401                string[0] = 0;
6402                if (expd_handle) {
6403                        flags = MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED;
6404                        switch (expstatus) {
6405                        case MPI2_EVENT_SAS_TOPO_ES_ADDED:
6406                                (void) sprintf(string, " added");
6407                                /*
6408                                 * New expander device added
```

```
6409                                 */
6410                                mpt->m_port_chng = 1;
6411                                topo_node = kmem_zalloc(
6412                                    sizeof (mptsas_topo_change_list_t),
6413                                    KM_SLEEP);
6414                                topo_node->mpt = mpt;
6415                                topo_node->event = MPTSAS_DR_EVENT_RECONFIG_SMP;
6416                                topo_node->un.physport = physport;
6417                                topo_node->devhdl = expd_handle;
6418                                topo_node->flags = flags;
6419                                topo_node->object = NULL;
6420                                if (topo_head == NULL) {
6421                                        topo_head = topo_tail = topo_node;
6422                                } else {
6423                                        topo_tail->next = topo_node;
6424                                        topo_tail = topo_node;
6425                                }
6426                                break;
6427                        case MPI2_EVENT_SAS_TOPO_ES_NOT_RESPONDING:
6428                                (void) sprintf(string, " not responding, "
6429                                    "removed");
6430                                psmp = mptsas_search_by_devhdl(smptbl,
6431                                    expd_handle);
6432                                if (psmp == NULL)
6433                                        break;

6435                                topo_node = kmem_zalloc(
6436                                    sizeof (mptsas_topo_change_list_t),
6437                                    KM_SLEEP);
6438                                topo_node->mpt = mpt;
6439                                topo_node->un.phymask = psmp->m_phymask;
6440                                topo_node->event = MPTSAS_DR_EVENT_OFFLINE_SMP;
6441                                topo_node->devhdl = expd_handle;
6442                                topo_node->flags = flags;
6443                                topo_node->object = NULL;
6444                                if (topo_head == NULL) {
6445                                        topo_head = topo_tail = topo_node;
6446                                } else {
6447                                        topo_tail->next = topo_node;
6448                                        topo_tail = topo_node;
6449                                }
6450                                break;
6451                        case MPI2_EVENT_SAS_TOPO_ES_RESPONDING:
6452                                break;
6453                        case MPI2_EVENT_SAS_TOPO_ES_DELAY_NOT_RESPONDING:
6454                                (void) sprintf(string, " not responding, "
6455                                    "delaying removal");
6456                                break;
6457                        default:
6458                                break;
6459                        }
6460                } else {
6461                        flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
6462                }

6464                NDBG20(("SAS TOPOLOGY CHANGE for enclosure %x expander %x%s\n",
6465                    enc_handle, expd_handle, string));
6466                for (i = 0; i < num_entries; i++) {
6467                        phy = i + start_phy_num;
6468                        phystatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
6469                            &sas_topo_change_list->PHY[i].PhyStatus);
6470                        dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6471                            &sas_topo_change_list->PHY[i].AttachedDevHandle);
6472                        reason_code = phystatus & MPI2_EVENT_SAS_TOPO_RC_MASK;
6473                        /*
6474                         * Filter out processing of Phy Vacant Status unless
```



```
6475                         * the reason code is "Not Responding".  Process all
6476                         * other combinations of Phy Status and Reason Codes.
6477                         */
6478                        if ((phystatus &
6479                            MPI2_EVENT_SAS_TOPO_PHYSTATUS_VACANT) &&
6480                            (reason_code !=
6481                            MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING)) {
6482                                continue;
6483                        }
6484                        curr[0] = 0;
6485                        prev[0] = 0;
6486                        string[0] = 0;
6487                        switch (reason_code) {
6488                        case MPI2_EVENT_SAS_TOPO_RC_TARG_ADDED:
6489                        {
6490                                NDBG20(("mptsas%d phy %d physical_port %d "
6491                                    "dev_handle %d added", mpt->m_instance, phy,
6492                                    physport, dev_handle));
6493                                link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
6494                                    &sas_topo_change_list->PHY[i].LinkRate);
6495                                state = (link_rate &
6496                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
6497                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
6498                                switch (state) {
6499                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
6500                                        (void) sprintf(curr, "is disabled");
6501                                        break;
6502                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6503                                        (void) sprintf(curr, "is offline, "
6504                                            "failed speed negotiation");
6505                                        break;
6506                                case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6507                                        (void) sprintf(curr, "SATA OOB "
6508                                            "complete");
6509                                        break;
6510                                case SMP_RESET_IN_PROGRESS:
6511                                        (void) sprintf(curr, "SMP reset in "
6512                                            "progress");
6513                                        break;
6514                                case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6515                                        (void) sprintf(curr, "is online at "
6516                                            "1.5 Gbps");
6517                                        break;
6518                                case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6519                                        (void) sprintf(curr, "is online at 3.0 "
6520                                            "Gbps");
6521                                        break;
6522                                case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6523                                        (void) sprintf(curr, "is online at 6.0 "
6524                                            "Gbps");
6525                                        break;
6526                                default:
6527                                        (void) sprintf(curr, "state is "
6528                                            "unknown");
6529                                        break;
6530                                }
6531                                /*
6532                                 * New target device added into the system.
6533                                 * Set association flag according to if an
6534                                 * expander is used or not.
6535                                 */
6536                                exp_flag =
6537                                    MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
6538                                if (flags ==
6539                                    MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
6540                                        flags = exp_flag;
```

```
6541                                }
6542                                topo_node = kmem_zalloc(
6543                                    sizeof (mptsas_topo_change_list_t),
6544                                    KM_SLEEP);
6545                                topo_node->mpt = mpt;
6546                                topo_node->event =
6547                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
6548                                if (expd_handle == 0) {
6549                                        /*
6550                                         * Per MPI 2, if expander dev handle
6551                                         * is 0, it's a directly attached
6552                                         * device. So driver use PHY to decide
6553                                         * which iport is associated
6554                                         */
6555                                        physport = phy;
6556                                        mpt->m_port_chng = 1;
6557                                }
6558                                topo_node->un.physport = physport;
6559                                topo_node->devhdl = dev_handle;
6560                                topo_node->flags = flags;
6561                                topo_node->object = NULL;
6562                                if (topo_head == NULL) {
6563                                        topo_head = topo_tail = topo_node;
6564                                } else {
6565                                        topo_tail->next = topo_node;
6566                                        topo_tail = topo_node;
6567                                }
6568                                break;
6569                        }
6570                        case MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING:
6571                        {
6572                                NDBG20(("mptsas%d phy %d physical_port %d "
6573                                    "dev_handle %d removed", mpt->m_instance,
6574                                    phy, physport, dev_handle));
6575                                /*
6576                                 * Set association flag according to if an
6577                                 * expander is used or not.
6578                                 */
6579                                exp_flag =
6580                                    MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
6581                                if (flags ==
6582                                    MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
6583                                        flags = exp_flag;
6584                                }
6585                                /*
6586                                 * Target device is removed from the system
6587                                 * Before the device is really offline from
6588                                 * from system.
6589                                 */
6590                                ptgt = mptsas_search_by_devhdl(tgttbl,
6591                                    dev_handle);
6592                                /*
6593                                 * If ptgt is NULL here, it means that the
6594                                 * DevHandle is not in the hash table.  This is
6595                                 * reasonable sometimes.  For example, if a
6596                                 * disk was pulled, then added, then pulled
6597                                 * again, the disk will not have been put into
6598                                 * the hash table because the add event will
6599                                 * have an invalid phymask.  BUT, this does not
6600                                 * mean that the DevHandle is invalid.  The
6601                                 * controller will still have a valid DevHandle
6602                                 * that must be removed.  To do this, use the
6603                                 * MPTSAS_TOPO_FLAG_REMOVE_HANDLE event.
6604                                 */
6605                                if (ptgt == NULL) {
6606                                        topo_node = kmem_zalloc(
```



```
6607                                            sizeof (mptsas_topo_change_list_t),
6608                                            KM_SLEEP);
6609                                        topo_node->mpt = mpt;
6610                                        topo_node->un.phymask = 0;
6611                                        topo_node->event =
6612                                            MPTSAS_TOPO_FLAG_REMOVE_HANDLE;
6613                                        topo_node->devhdl = dev_handle;
6614                                        topo_node->flags = flags;
6615                                        topo_node->object = NULL;
6616                                        if (topo_head == NULL) {
6617                                                topo_head = topo_tail =
6618                                                    topo_node;
6619                                        } else {
6620                                                topo_tail->next = topo_node;
6621                                                topo_tail = topo_node;
6622                                        }
6623                                        break;
6624                                }

6626                                /*
6627                                 * Update DR flag immediately avoid I/O failure
6628                                 * before failover finish. Pay attention to the
6629                                 * mutex protect, we need grab m_tx_waitq_mutex
6630                                 * during set m_dr_flag because we won't add
6631                                 * the following command into waitq, instead,
6632                                 * we need return TRAN_BUSY in the tran_start
6633                                 * context.
7018                                 * mutex protect, we need grab the per target
7019                                 * mutex during set m_dr_flag because the
7020                                 * m_mutex would not be held all the time in
7021                                 * mptsas_scsi_start().
6634                                 */
6635                                mutex_enter(&mpt->m_tx_waitq_mutex);
7023                                mutex_enter(&ptgt->m_tgt_intr_mutex);
6636                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6637                                mutex_exit(&mpt->m_tx_waitq_mutex);
7025                                mutex_exit(&ptgt->m_tgt_intr_mutex);

6639                                topo_node = kmem_zalloc(
6640                                    sizeof (mptsas_topo_change_list_t),
6641                                    KM_SLEEP);
6642                                topo_node->mpt = mpt;
6643                                topo_node->un.phymask = ptgt->m_phymask;
6644                                topo_node->event =
6645                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
6646                                topo_node->devhdl = dev_handle;
6647                                topo_node->flags = flags;
6648                                topo_node->object = NULL;
6649                                if (topo_head == NULL) {
6650                                        topo_head = topo_tail = topo_node;
6651                                } else {
6652                                        topo_tail->next = topo_node;
6653                                        topo_tail = topo_node;
6654                                }
6655                                break;
6656                        }
6657                        case MPI2_EVENT_SAS_TOPO_RC_PHY_CHANGED:
6658                                link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
6659                                    &sas_topo_change_list->PHY[i].LinkRate);
6660                                state = (link_rate &
6661                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
6662                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
6663                                pSmhba = &mpt->m_phy_info[i].smhba_info;
6664                                pSmhba->negotiated_link_rate = state;
6665                                switch (state) {
6666                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
```

```
6667                                    (void) sprintf(curr, "is disabled");
6668                                    mptsas_smhba_log_sysevent(mpt,
6669                                        ESC_SAS_PHY_EVENT,
6670                                        SAS_PHY_REMOVE,
6671                                        &mpt->m_phy_info[i].smhba_info);
6672                                    mpt->m_phy_info[i].smhba_info.
6673                                        negotiated_link_rate
6674                                        = 0x1;
6675                                    break;
6676                            case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6677                                    (void) sprintf(curr, "is offline, "
6678                                        "failed speed negotiation");
6679                                    mptsas_smhba_log_sysevent(mpt,
6680                                        ESC_SAS_PHY_EVENT,
6681                                        SAS_PHY_OFFLINE,
6682                                        &mpt->m_phy_info[i].smhba_info);
6683                                    break;
6684                            case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6685                                    (void) sprintf(curr, "SATA OOB "
6686                                        "complete");
6687                                    break;
6688                            case SMP_RESET_IN_PROGRESS:
6689                                    (void) sprintf(curr, "SMP reset in "
6690                                        "progress");
6691                                    break;
6692                            case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6693                                    (void) sprintf(curr, "is online at "
6694                                        "1.5 Gbps");
6695                                    if ((expd_handle == 0) &&
6696                                        (enc_handle == 1)) {
6697                                            mpt->m_port_chng = 1;
6698                                    }
6699                                    mptsas_smhba_log_sysevent(mpt,
6700                                        ESC_SAS_PHY_EVENT,
6701                                        SAS_PHY_ONLINE,
6702                                        &mpt->m_phy_info[i].smhba_info);
6703                                    break;
6704                            case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6705                                    (void) sprintf(curr, "is online at 3.0 "
6706                                        "Gbps");
6707                                    if ((expd_handle == 0) &&
6708                                        (enc_handle == 1)) {
6709                                            mpt->m_port_chng = 1;
6710                                    }
6711                                    mptsas_smhba_log_sysevent(mpt,
6712                                        ESC_SAS_PHY_EVENT,
6713                                        SAS_PHY_ONLINE,
6714                                        &mpt->m_phy_info[i].smhba_info);
6715                                    break;
6716                            case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6717                                    (void) sprintf(curr, "is online at "
6718                                        "6.0 Gbps");
6719                                    if ((expd_handle == 0) &&
6720                                        (enc_handle == 1)) {
6721                                            mpt->m_port_chng = 1;
6722                                    }
6723                                    mptsas_smhba_log_sysevent(mpt,
6724                                        ESC_SAS_PHY_EVENT,
6725                                        SAS_PHY_ONLINE,
6726                                        &mpt->m_phy_info[i].smhba_info);
6727                                    break;
6728                            default:
6729                                    (void) sprintf(curr, "state is "
6730                                        "unknown");
6731                                    break;
6732                            }
```



```
6734                                    state = (link_rate &
6735                                        MPI2_EVENT_SAS_TOPO_LR_PREV_MASK) >>
6736                                        MPI2_EVENT_SAS_TOPO_LR_PREV_SHIFT;
6737                                    switch (state) {
6738                                    case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
6739                                            (void) sprintf(prev, ", was disabled");
6740                                            break;
6741                                    case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6742                                            (void) sprintf(prev, ", was offline, "
6743                                                "failed speed negotiation");
6744                                            break;
6745                                    case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6746                                            (void) sprintf(prev, ", was SATA OOB "
6747                                                "complete");
6748                                            break;
6749                                    case SMP_RESET_IN_PROGRESS:
6750                                            (void) sprintf(prev, ", was SMP reset "
6751                                                "in progress");
6752                                            break;
6753                                    case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6754                                            (void) sprintf(prev, ", was online at "
6755                                                "1.5 Gbps");
6756                                            break;
6757                                    case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6758                                            (void) sprintf(prev, ", was online at "
6759                                                "3.0 Gbps");
6760                                            break;
6761                                    case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6762                                            (void) sprintf(prev, ", was online at "
6763                                                "6.0 Gbps");
6764                                            break;
6765                                    default:
6766                                    break;
6767                                    }
6768                                    (void) sprintf(&string[strlen(string)], "link "
6769                                        "changed, ");
6770                                    break;
6771                            case MPI2_EVENT_SAS_TOPO_RC_NO_CHANGE:
6772                                    continue;
6773                            case MPI2_EVENT_SAS_TOPO_RC_DELAY_NOT_RESPONDING:
6774                                    (void) sprintf(&string[strlen(string)],
6775                                        "target not responding, delaying "
6776                                        "removal");
6777                                    break;
6778                            }
6779                            NDBG20(("mptsas%d phy %d DevHandle %x, %s%s%s\n",
6780                                mpt->m_instance, phy, dev_handle, string, curr,
6781                                prev));
6782                    }
6783                    if (topo_head != NULL) {
6784                            /*
6785                             * Launch DR taskq to handle topology change
6786                             */
6787                            if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
6788                                mptsas_handle_dr, (void *)topo_head,
6789                                DDI_NOSLEEP)) != DDI_SUCCESS) {
6790                                    mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
6791                                        "for handle SAS DR event failed. \n");
6792                            }
6793                    }
6794                    break;
6795            }
6796            case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
6797            {
6798                    Mpi2EventDataIrConfigChangeList_t       *irChangeList;
```

```
6799                mptsas_topo_change_list_t              *topo_head = NULL;
6800                mptsas_topo_change_list_t              *topo_tail = NULL;
6801                mptsas_topo_change_list_t              *topo_node = NULL;
6802                mptsas_target_t                        *ptgt;
6803                mptsas_hash_table_t                    *tgttbl;
6804                uint8_t                                num_entries, i, reason;
6805                uint16_t                               volhandle, diskhandle;

6807                irChangeList = (pMpi2EventDataIrConfigChangeList_t)
6808                    eventreply->EventData;
6809                num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
6810                    &irChangeList->NumElements);

6812                tgttbl = &mpt->m_active->m_tgttbl;

6814                NDBG20(("mptsas%d IR_CONFIGURATION_CHANGE_LIST event received",
6815                    mpt->m_instance));

6817                for (i = 0; i < num_entries; i++) {
6818                        reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
6819                            &irChangeList->ConfigElement[i].ReasonCode);
6820                        volhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6821                            &irChangeList->ConfigElement[i].VolDevHandle);
6822                        diskhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6823                            &irChangeList->ConfigElement[i].PhysDiskDevHandle);

6825                        switch (reason) {
6826                        case MPI2_EVENT_IR_CHANGE_RC_ADDED:
6827                        case MPI2_EVENT_IR_CHANGE_RC_VOLUME_CREATED:
6828                        {
6829                                NDBG20(("mptsas %d volume added\n",
6830                                    mpt->m_instance));

6832                                topo_node = kmem_zalloc(
6833                                    sizeof (mptsas_topo_change_list_t),
6834                                    KM_SLEEP);

6836                                topo_node->mpt = mpt;
6837                                topo_node->event =
6838                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
6839                                topo_node->un.physport = 0xff;
6840                                topo_node->devhdl = volhandle;
6841                                topo_node->flags =
6842                                    MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
6843                                topo_node->object = NULL;
6844                                if (topo_head == NULL) {
6845                                        topo_head = topo_tail = topo_node;
6846                                } else {
6847                                        topo_tail->next = topo_node;
6848                                        topo_tail = topo_node;
6849                                }
6850                                break;
6851                        }
6852                        case MPI2_EVENT_IR_CHANGE_RC_REMOVED:
6853                        case MPI2_EVENT_IR_CHANGE_RC_VOLUME_DELETED:
6854                        {
6855                                NDBG20(("mptsas %d volume deleted\n",
6856                                    mpt->m_instance));
6857                                ptgt = mptsas_search_by_devhdl(tgttbl,
6858                                    volhandle);
6859                                if (ptgt == NULL)
6860                                        break;

6862                                /*
6863                                 * Clear any flags related to volume
6864                                 */
```



```
6865                                (void) mptsas_delete_volume(mpt, volhandle);

6867                                /*
6868                                 * Update DR flag immediately avoid I/O failure
6869                                 */
6870                                mutex_enter(&mpt->m_tx_waitq_mutex);
7258                                mutex_enter(&ptgt->m_tgt_intr_mutex);
6871                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6872                                mutex_exit(&mpt->m_tx_waitq_mutex);
7260                                mutex_exit(&ptgt->m_tgt_intr_mutex);

6874                                topo_node = kmem_zalloc(
6875                                    sizeof (mptsas_topo_change_list_t),
6876                                    KM_SLEEP);
6877                                topo_node->mpt = mpt;
6878                                topo_node->un.phymask = ptgt->m_phymask;
6879                                topo_node->event =
6880                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
6881                                topo_node->devhdl = volhandle;
6882                                topo_node->flags =
6883                                    MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
6884                                topo_node->object = (void *)ptgt;
6885                                if (topo_head == NULL) {
6886                                        topo_head = topo_tail = topo_node;
6887                                } else {
6888                                        topo_tail->next = topo_node;
6889                                        topo_tail = topo_node;
6890                                }
6891                                break;
6892                        }
6893                        case MPI2_EVENT_IR_CHANGE_RC_PD_CREATED:
6894                        case MPI2_EVENT_IR_CHANGE_RC_HIDE:
6895                        {
6896                                ptgt = mptsas_search_by_devhdl(tgttbl,
6897                                    diskhandle);
6898                                if (ptgt == NULL)
6899                                        break;

6901                                /*
6902                                 * Update DR flag immediately avoid I/O failure
6903                                 */
6904                                mutex_enter(&mpt->m_tx_waitq_mutex);
7292                                mutex_enter(&ptgt->m_tgt_intr_mutex);
6905                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6906                                mutex_exit(&mpt->m_tx_waitq_mutex);
7294                                mutex_exit(&ptgt->m_tgt_intr_mutex);

6908                                topo_node = kmem_zalloc(
6909                                    sizeof (mptsas_topo_change_list_t),
6910                                    KM_SLEEP);
6911                                topo_node->mpt = mpt;
6912                                topo_node->un.phymask = ptgt->m_phymask;
6913                                topo_node->event =
6914                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
6915                                topo_node->devhdl = diskhandle;
6916                                topo_node->flags =
6917                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
6918                                topo_node->object = (void *)ptgt;
6919                                if (topo_head == NULL) {
6920                                        topo_head = topo_tail = topo_node;
6921                                } else {
6922                                        topo_tail->next = topo_node;
6923                                        topo_tail = topo_node;
6924                                }
6925                                break;
6926                        }
```

```
6927                        case MPI2_EVENT_IR_CHANGE_RC_UNHIDE:
6928                        case MPI2_EVENT_IR_CHANGE_RC_PD_DELETED:
6929                        {
6930                                /*
6931                                 * The physical drive is released by a IR
6932                                 * volume. But we cannot get the the physport
6933                                 * or phynum from the event data, so we only
6934                                 * can get the physport/phynum after SAS
6935                                 * Device Page0 request for the devhdl.
6936                                 */
6937                                topo_node = kmem_zalloc(
6938                                    sizeof (mptsas_topo_change_list_t),
6939                                    KM_SLEEP);
6940                                topo_node->mpt = mpt;
6941                                topo_node->un.phymask = 0;
6942                                topo_node->event =
6943                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
6944                                topo_node->devhdl = diskhandle;
6945                                topo_node->flags =
6946                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
6947                                topo_node->object = NULL;
6948                                mpt->m_port_chng = 1;
6949                                if (topo_head == NULL) {
6950                                        topo_head = topo_tail = topo_node;
6951                                } else {
6952                                        topo_tail->next = topo_node;
6953                                        topo_tail = topo_node;
6954                                }
6955                                break;
6956                        }
6957                        default:
6958                                break;
6959                        }
6960                }

6962                if (topo_head != NULL) {
6963                        /*
6964                         * Launch DR taskq to handle topology change
6965                         */
6966                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
6967                            mptsas_handle_dr, (void *)topo_head,
6968                            DDI_NOSLEEP)) != DDI_SUCCESS) {
6969                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
6970                                    "for handle SAS DR event failed. \n");
6971                        }
6972                }
6973                break;
6974        }
6975        default:
6976                return (DDI_FAILURE);
6977        }

6979        return (DDI_SUCCESS);
6980 }

6982 /*
6983  * handle events from ioc
6984  */
6985 static void
6986 mptsas_handle_event(void *args)
6987 {
6988        m_replyh_arg_t                  *replyh_arg;
6989        pMpi2EventNotificationReply_t   eventreply;
6990        uint32_t                        event, iocloginfo, rfm;
6991        uint32_t                        status;
6992        uint8_t                         port;
```



```
6993        mptsas_t                        *mpt;
6994        uint_t                          iocstatus;

6996        replyh_arg = (m_replyh_arg_t *)args;
6997        rfm = replyh_arg->rfm;
6998        mpt = replyh_arg->mpt;

7000        mutex_enter(&mpt->m_mutex);
7001        /*
7002         * If HBA is being reset, drop incoming event.
7003         */
7004        if (mpt->m_in_reset) {
7005                NDBG20(("dropping event received prior to reset"));
7006                mutex_exit(&mpt->m_mutex);
7007                return;
7008        }

7010        eventreply = (pMpi2EventNotificationReply_t)
7011            (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
7012        event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

7014        if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
7015            &eventreply->IOCStatus)) {
7016                if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
7017                        mptsas_log(mpt, CE_WARN,
7018                            "!mptsas_handle_event: IOCStatus=0x%x, "
7019                            "IOCLogInfo=0x%x", iocstatus,
7020                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7021                            &eventreply->IOCLogInfo));
7022                } else {
7023                        mptsas_log(mpt, CE_WARN,
7024                            "mptsas_handle_event: IOCStatus=0x%x, "
7025                            "IOCLogInfo=0x%x", iocstatus,
7026                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7027                            &eventreply->IOCLogInfo));
7028                }
7029        }

7031        /*
7032         * figure out what kind of event we got and handle accordingly
7033         */
7034        switch (event) {
7035        case MPI2_EVENT_LOG_ENTRY_ADDED:
7036                break;
7037        case MPI2_EVENT_LOG_DATA:
7038                iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7039                    &eventreply->IOCLogInfo);
7040                NDBG20(("mptsas %d log info %x received.\n", mpt->m_instance,
7041                    iocloginfo));
7042                break;
7043        case MPI2_EVENT_STATE_CHANGE:
7044                NDBG20(("mptsas%d state change.", mpt->m_instance));
7045                break;
7046        case MPI2_EVENT_HARD_RESET_RECEIVED:
7047                NDBG20(("mptsas%d event change.", mpt->m_instance));
7048                break;
7049        case MPI2_EVENT_SAS_DISCOVERY:
7050        {
7051                MPI2_EVENT_DATA_SAS_DISCOVERY   *sasdiscovery;
7052                char                            string[80];
7053                uint8_t                         rc;

7055                sasdiscovery =
7056                    (pMpi2EventDataSasDiscovery_t)eventreply->EventData;

7058                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
```

```
7059                    &sasdiscovery->ReasonCode);
7060                port = ddi_get8(mpt->m_acc_reply_frame_hdl,
7061                    &sasdiscovery->PhysicalPort);
7062                status = ddi_get32(mpt->m_acc_reply_frame_hdl,
7063                    &sasdiscovery->DiscoveryStatus);

7065                string[0] = 0;
7066                switch (rc) {
7067                case MPI2_EVENT_SAS_DISC_RC_STARTED:
7068                        (void) sprintf(string, "STARTING");
7069                        break;
7070                case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
7071                        (void) sprintf(string, "COMPLETED");
7072                        break;
7073                default:
7074                        (void) sprintf(string, "UNKNOWN");
7075                        break;
7076                }

7078                NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
7079                    port, status));

7081                break;
7082        }
7083        case MPI2_EVENT_EVENT_CHANGE:
7084                NDBG20(("mptsas%d event change.", mpt->m_instance));
7085                break;
7086        case MPI2_EVENT_TASK_SET_FULL:
7087        {
7088                pMpi2EventDataTaskSetFull_t     taskfull;

7090                taskfull = (pMpi2EventDataTaskSetFull_t)eventreply->EventData;

7092                NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
7093                    mpt->m_instance,  ddi_get16(mpt->m_acc_reply_frame_hdl,
7094                    &taskfull->CurrentDepth)));
7095                break;
7096        }
7097        case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
7098        {
7099                /*
7100                 * SAS TOPOLOGY CHANGE LIST Event has already been handled
7101                 * in mptsas_handle_event_sync() of interrupt context
7102                 */
7103                break;
7104        }
7105        case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
7106        {
7107                pMpi2EventDataSasEnclDevStatusChange_t  encstatus;
7108                uint8_t                                 rc;
7109                char                                    string[80];

7111                encstatus = (pMpi2EventDataSasEnclDevStatusChange_t)
7112                    eventreply->EventData;

7114                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7115                    &encstatus->ReasonCode);
7116                switch (rc) {
7117                case MPI2_EVENT_SAS_ENCL_RC_ADDED:
7118                        (void) sprintf(string, "added");
7119                        break;
7120                case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
7121                        (void) sprintf(string, ", not responding");
7122                        break;
7123                default:
7124                break;
```



```
7125                }
7126                NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure %x%s\n",
7127                    mpt->m_instance, ddi_get16(mpt->m_acc_reply_frame_hdl,
7128                    &encstatus->EnclosureHandle), string));
7129                break;
7130        }

7132        /*
7133         * MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by
7134         * mptsas_handle_event_sync,in here just send ack message.
7135         */
7136        case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
7137        {
7138                pMpi2EventDataSasDeviceStatusChange_t   statuschange;
7139                uint8_t                                 rc;
7140                uint16_t                                devhdl;
7141                uint64_t                                wwn = 0;
7142                uint32_t                                wwn_lo, wwn_hi;

7144                statuschange = (pMpi2EventDataSasDeviceStatusChange_t)
7145                    eventreply->EventData;
7146                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7147                    &statuschange->ReasonCode);
7148                wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7149                    (uint32_t *)(void *)&statuschange->SASAddress);
7150                wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
7151                    (uint32_t *)(void *)&statuschange->SASAddress + 1);
7152                wwn = ((uint64_t)wwn_hi << 32) | wwn_lo;
7153                devhdl =  ddi_get16(mpt->m_acc_reply_frame_hdl,
7154                    &statuschange->DevHandle);

7156                NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wwn is %"PRIx64,
7157                    wwn));

7159                switch (rc) {
7160                case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
7161                        NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
7162                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7163                            &statuschange->ASC),
7164                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7165                            &statuschange->ASCQ)));
7166                        break;

7168                case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
7169                        NDBG20(("Device not supported"));
7170                        break;

7172                case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
7173                        NDBG20(("IOC internally generated the Target Reset "
7174                            "for devhdl:%x", devhdl));
7175                        break;

7177                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
7178                        NDBG20(("IOC's internally generated Target Reset "
7179                            "completed for devhdl:%x", devhdl));
7180                        break;

7182                case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
7183                        NDBG20(("IOC internally generated Abort Task"));
7184                        break;

7186                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
7187                        NDBG20(("IOC's internally generated Abort Task "
7188                            "completed"));
7189                        break;
```

```
7191                case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
7192                        NDBG20(("IOC internally generated Abort Task Set"));
7193                        break;

7195                case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
7196                        NDBG20(("IOC internally generated Clear Task Set"));
7197                        break;

7199                case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
7200                        NDBG20(("IOC internally generated Query Task"));
7201                        break;

7203                case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
7204                        NDBG20(("Device sent an Asynchronous Notification"));
7205                        break;

7207                default:
7208                        break;
7209                }
7210                break;
7211        }
7212        case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7213        {
7214                /*
7215                 * IR TOPOLOGY CHANGE LIST Event has already been handled
7216                 * in mpt_handle_event_sync() of interrupt context
7217                 */
7218                break;
7219        }
7220        case MPI2_EVENT_IR_OPERATION_STATUS:
7221        {
7222                Mpi2EventDataIrOperationStatus_t        *irOpStatus;
7223                char                                    reason_str[80];
7224                uint8_t                                 rc, percent;
7225                uint16_t                                handle;

7227                irOpStatus = (pMpi2EventDataIrOperationStatus_t)
7228                    eventreply->EventData;
7229                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7230                    &irOpStatus->RAIDOperation);
7231                percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
7232                    &irOpStatus->PercentComplete);
7233                handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7234                    &irOpStatus->VolDevHandle);

7236                switch (rc) {
7237                        case MPI2_EVENT_IR_RAIDOP_RESYNC:
7238                                (void) sprintf(reason_str, "resync");
7239                                break;
7240                        case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
7241                                (void) sprintf(reason_str, "online capacity "
7242                                    "expansion");
7243                                break;
7244                        case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
7245                                (void) sprintf(reason_str, "consistency check");
7246                                break;
7247                        default:
7248                                (void) sprintf(reason_str, "unknown reason %x",
7249                                    rc);
7250                }

7252                NDBG20(("mptsas%d raid operational status: (%s)"
7253                    "\thandle(0x%04x), percent complete(%d)\n",
7254                    mpt->m_instance, reason_str, handle, percent));
7255                break;
7256        }
```



```
7257        case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
7258        {
7259                pMpi2EventDataSasBroadcastPrimitive_t   sas_broadcast;
7260                uint8_t                                 phy_num;
7261                uint8_t                                 primitive;

7263                sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
7264                    eventreply->EventData;

7266                phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
7267                    &sas_broadcast->PhyNum);
7268                primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
7269                    &sas_broadcast->Primitive);

7271                switch (primitive) {
7272                case MPI2_EVENT_PRIMITIVE_CHANGE:
7273                        mptsas_smhba_log_sysevent(mpt,
7274                            ESC_SAS_HBA_PORT_BROADCAST,
7275                            SAS_PORT_BROADCAST_CHANGE,
7276                            &mpt->m_phy_info[phy_num].smhba_info);
7277                        break;
7278                case MPI2_EVENT_PRIMITIVE_SES:
7279                        mptsas_smhba_log_sysevent(mpt,
7280                            ESC_SAS_HBA_PORT_BROADCAST,
7281                            SAS_PORT_BROADCAST_SES,
7282                            &mpt->m_phy_info[phy_num].smhba_info);
7283                        break;
7284                case MPI2_EVENT_PRIMITIVE_EXPANDER:
7285                        mptsas_smhba_log_sysevent(mpt,
7286                            ESC_SAS_HBA_PORT_BROADCAST,
7287                            SAS_PORT_BROADCAST_D01_4,
7288                            &mpt->m_phy_info[phy_num].smhba_info);
7289                        break;
7290                case MPI2_EVENT_PRIMITIVE_ASYNCHRONOUS_EVENT:
7291                        mptsas_smhba_log_sysevent(mpt,
7292                            ESC_SAS_HBA_PORT_BROADCAST,
7293                            SAS_PORT_BROADCAST_D04_7,
7294                            &mpt->m_phy_info[phy_num].smhba_info);
7295                        break;
7296                case MPI2_EVENT_PRIMITIVE_RESERVED3:
7297                        mptsas_smhba_log_sysevent(mpt,
7298                            ESC_SAS_HBA_PORT_BROADCAST,
7299                            SAS_PORT_BROADCAST_D16_7,
7300                            &mpt->m_phy_info[phy_num].smhba_info);
7301                        break;
7302                case MPI2_EVENT_PRIMITIVE_RESERVED4:
7303                        mptsas_smhba_log_sysevent(mpt,
7304                            ESC_SAS_HBA_PORT_BROADCAST,
7305                            SAS_PORT_BROADCAST_D29_7,
7306                            &mpt->m_phy_info[phy_num].smhba_info);
7307                        break;
7308                case MPI2_EVENT_PRIMITIVE_CHANGE0_RESERVED:
7309                        mptsas_smhba_log_sysevent(mpt,
7310                            ESC_SAS_HBA_PORT_BROADCAST,
7311                            SAS_PORT_BROADCAST_D24_0,
7312                            &mpt->m_phy_info[phy_num].smhba_info);
7313                        break;
7314                case MPI2_EVENT_PRIMITIVE_CHANGE1_RESERVED:
7315                        mptsas_smhba_log_sysevent(mpt,
7316                            ESC_SAS_HBA_PORT_BROADCAST,
7317                            SAS_PORT_BROADCAST_D27_4,
7318                            &mpt->m_phy_info[phy_num].smhba_info);
7319                        break;
7320                default:
7321                        NDBG20(("mptsas%d: unknown BROADCAST PRIMITIVE"
7322                            " %x received",
```

```
7323                            mpt->m_instance, primitive));
7324                    break;
7325            }
7326            NDBG20(("mptsas%d sas broadcast primitive: "
7327                "\tprimitive(0x%04x), phy(%d) complete\n",
7328                mpt->m_instance, primitive, phy_num));
7329            break;
7330        }
7331        case MPI2_EVENT_IR_VOLUME:
7332        {
7333            Mpi2EventDataIrVolume_t         *irVolume;
7334            uint16_t                        devhandle;
7335            uint32_t                        state;
7336            int                             config, vol;
7337            mptsas_slots_t                  *slots = mpt->m_active;
7338            uint8_t                         found = FALSE;

7340            irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
7341            state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7342                &irVolume->NewValue);
7343            devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7344                &irVolume->VolDevHandle);

7346            NDBG20(("EVENT_IR_VOLUME event is received"));

7348            /*
7349             * Get latest RAID info and then find the DevHandle for this
7350             * event in the configuration.  If the DevHandle is not found
7351             * just exit the event.
7352             */
7353            (void) mptsas_get_raid_info(mpt);
7354            for (config = 0; (config < slots->m_num_raid_configs) &&
7355                (!found); config++) {
7356                for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
7357                    if (slots->m_raidconfig[config].m_raidvol[vol].
7358                        m_raidhandle == devhandle) {
7359                            found = TRUE;
7360                            break;
7361                    }
7362                }
7363            }
7364            if (!found) {
7365                    break;
7366            }

7368            switch (irVolume->ReasonCode) {
7369            case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
7370            {
7371                    uint32_t i;
7372                    slots->m_raidconfig[config].m_raidvol[vol].m_settings =
7373                        state;

7375                    i = state & MPI2_RAIDVOL0_SETTING_MASK_WRITE_CACHING;
7376                    mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
7377                        ", auto-config of hot-swap drives is %s"
7378                        ", write caching is %s"
7379                        ", hot-spare pool mask is %02x\n",
7380                        vol, state &
7381                        MPI2_RAIDVOL0_SETTING_AUTO_CONFIG_HSWAP_DISABLE
7382                        ? "disabled" : "enabled",
7383                        i == MPI2_RAIDVOL0_SETTING_UNCHANGED
7384                        ? "controlled by member disks" :
7385                        i == MPI2_RAIDVOL0_SETTING_DISABLE_WRITE_CACHING
7386                        ? "disabled" :
7387                        i == MPI2_RAIDVOL0_SETTING_ENABLE_WRITE_CACHING
7388                        ? "enabled" :
```



```
7389                            "incorrectly set",
7390                            (state >> 16) & 0xff);
7391                                break;
7392                }
7393                case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
7394                {
7395                        slots->m_raidconfig[config].m_raidvol[vol].m_state =
7396                            (uint8_t)state;

7398                        mptsas_log(mpt, CE_NOTE,
7399                            "Volume %d is now %s\n", vol,
7400                            state == MPI2_RAID_VOL_STATE_OPTIMAL
7401                            ? "optimal" :
7402                            state == MPI2_RAID_VOL_STATE_DEGRADED
7403                            ? "degraded" :
7404                            state == MPI2_RAID_VOL_STATE_ONLINE
7405                            ? "online" :
7406                            state == MPI2_RAID_VOL_STATE_INITIALIZING
7407                            ? "initializing" :
7408                            state == MPI2_RAID_VOL_STATE_FAILED
7409                            ? "failed" :
7410                            state == MPI2_RAID_VOL_STATE_MISSING
7411                            ? "missing" :
7412                            "state unknown");
7413                        break;
7414                }
7415                case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
7416                {
7417                        slots->m_raidconfig[config].m_raidvol[vol].
7418                            m_statusflags = state;

7420                        mptsas_log(mpt, CE_NOTE,
7421                            " Volume %d is now %s%s%s%s%s%s%s%s\n",
7422                            vol,
7423                            state & MPI2_RAIDVOL0_STATUS_FLAG_ENABLED
7424                            ? ", enabled" : ", disabled",
7425                            state & MPI2_RAIDVOL0_STATUS_FLAG_QUIESCED
7426                            ? ", quiesced" : "",
7427                            state & MPI2_RAIDVOL0_STATUS_FLAG_VOLUME_INACTIVE
7428                            ? ", inactive" : ", active",
7429                            state &
7430                            MPI2_RAIDVOL0_STATUS_FLAG_BAD_BLOCK_TABLE_FULL
7431                            ? ", bad block table is full" : "",
7432                            state &
7433                            MPI2_RAIDVOL0_STATUS_FLAG_RESYNC_IN_PROGRESS
7434                            ? ", resync in progress" : "",
7435                            state & MPI2_RAIDVOL0_STATUS_FLAG_BACKGROUND_INIT
7436                            ? ", background initialization in progress" : "",
7437                            state &
7438                            MPI2_RAIDVOL0_STATUS_FLAG_CAPACITY_EXPANSION
7439                            ? ", capacity expansion in progress" : "",
7440                            state &
7441                            MPI2_RAIDVOL0_STATUS_FLAG_CONSISTENCY_CHECK
7442                            ? ", consistency check in progress" : "",
7443                            state & MPI2_RAIDVOL0_STATUS_FLAG_DATA_SCRUB
7444                            ? ", data scrub in progress" : "");
7445                        break;
7446                }
7447                default:
7448                        break;
7449                }
7450                break;
7451        }
7452        case MPI2_EVENT_IR_PHYSICAL_DISK:
7453        {
7454                Mpi2EventDataIrPhysicalDisk_t   *irPhysDisk;
```

```
7455                uint16_t                        devhandle, enchandle, slot;
7456                uint32_t                        status, state;
7457                uint8_t                         physdisknum, reason;

7459                irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
7460                    eventreply->EventData;
7461                physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,
7462                    &irPhysDisk->PhysDiskNum);
7463                devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7464                    &irPhysDisk->PhysDiskDevHandle);
7465                enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7466                    &irPhysDisk->EnclosureHandle);
7467                slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
7468                    &irPhysDisk->Slot);
7469                state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7470                    &irPhysDisk->NewValue);
7471                reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
7472                    &irPhysDisk->ReasonCode);

7474                NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));

7476                switch (reason) {
7477                case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
7478                        mptsas_log(mpt, CE_NOTE,
7479                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7480                            "for enclosure with handle 0x%x is now in hot "
7481                            "spare pool %d",
7482                            physdisknum, devhandle, slot, enchandle,
7483                            (state >> 16) & 0xff);
7484                        break;

7486                case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
7487                        status = state;
7488                        mptsas_log(mpt, CE_NOTE,
7489                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7490                            "for enclosure with handle 0x%x is now "
7491                            "%s%s%s%s%s\n", physdisknum, devhandle, slot,
7492                            enchandle,
7493                            status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
7494                            ? ", inactive" : ", active",
7495                            status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
7496                            ? ", out of sync" : "",
7497                            status & MPI2_PHYSDISK0_STATUS_FLAG_QUIESCED
7498                            ? ", quiesced" : "",
7499                            status &
7500                            MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
7501                            ? ", write cache enabled" : "",
7502                            status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
7503                            ? ", capacity expansion target" : "");
7504                        break;

7506                case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
7507                        mptsas_log(mpt, CE_NOTE,
7508                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7509                            "for enclosure with handle 0x%x is now %s\n",
7510                            physdisknum, devhandle, slot, enchandle,
7511                            state == MPI2_RAID_PD_STATE_OPTIMAL
7512                            ? "optimal" :
7513                            state == MPI2_RAID_PD_STATE_REBUILDING
7514                            ? "rebuilding" :
7515                            state == MPI2_RAID_PD_STATE_DEGRADED
7516                            ? "degraded" :
7517                            state == MPI2_RAID_PD_STATE_HOT_SPARE
7518                            ? "a hot spare" :
7519                            state == MPI2_RAID_PD_STATE_ONLINE
7520                            ? "online" :
```



```
7521                            state == MPI2_RAID_PD_STATE_OFFLINE
7522                            ? "offline" :
7523                            state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
7524                            ? "not compatible" :
7525                            state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
7526                            ? "not configured" :
7527                            "state unknown");
7528                        break;
7529                }
7530                break;
7531        }
7532        default:
7533                NDBG20(("mptsas%d: unknown event %x received",
7534                    mpt->m_instance, event));
7535                break;
7536        }

7538        /*
7539         * Return the reply frame to the free queue.
7540         */
7541        ddi_put32(mpt->m_acc_free_queue_hdl,
7542            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index], rfm);
7543        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
7544            DDI_DMA_SYNC_FORDEV);
7545        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
7546                mpt->m_free_index = 0;
7547        }
7548        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
7549            mpt->m_free_index);
7550        mutex_exit(&mpt->m_mutex);
7551 }

7553 /*
7554  * invoked from timeout() to restart qfull cmds with throttle == 0
7555  */
7556 static void
7557 mptsas_restart_cmd(void *arg)
7558 {
7559        mptsas_t        *mpt = arg;
7560        mptsas_target_t *ptgt = NULL;

7562        mutex_enter(&mpt->m_mutex);

7564        mpt->m_restart_cmd_timeid = 0;

7566        ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
7567            MPTSAS_HASH_FIRST);
7568        while (ptgt != NULL) {
7949                mutex_enter(&ptgt->m_tgt_intr_mutex);
7569                if (ptgt->m_reset_delay == 0) {
7570                        if (ptgt->m_t_throttle == QFULL_THROTTLE) {
7571                                mptsas_set_throttle(mpt, ptgt,
7572                                    MAX_THROTTLE);
7573                        }
7574                }
7956                mutex_exit(&ptgt->m_tgt_intr_mutex);

7576                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
7577                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
7578        }
7579        mptsas_restart_hba(mpt);
7580        mutex_exit(&mpt->m_mutex);
7581 }

7965 /*
7966  * mptsas_remove_cmd0 is similar to mptsas_remove_cmd except that it is called
```

```
 * where m_intr_mutex has already been held.
 */
void
mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        ASSERT(mutex_owned(&mpt->m_mutex));

        /*
         * With new fine-grained lock mechanism, the outstanding cmd is only
         * linked to m_active before the dma is triggerred(MPTSAS_START_CMD)
         * to send it. that is, mptsas_save_cmd() doesn't link the outstanding
         * cmd now. So when mptsas_remove_cmd is called, a mptsas_save_cmd must
         * have been called, but the cmd may have not been linked.
         * For mptsas_remove_cmd0, the cmd must have been linked.
         * In order to keep the same semantic, we link the cmd to the
         * outstanding cmd list.
         */
        mpt->m_active->m_slot[cmd->cmd_slot] = cmd;

        mutex_enter(&mpt->m_intr_mutex);
        mptsas_remove_cmd0(mpt, cmd);
        mutex_exit(&mpt->m_intr_mutex);
}

static inline void
mptsas_remove_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        int             slot;
        mptsas_slots_t  *slots = mpt->m_active;
        int             t;
        mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
        mptsas_slot_free_e_t    *pe;

        ASSERT(cmd != NULL);
        ASSERT(cmd->cmd_queued == FALSE);

        /*
         * Task Management cmds are removed in their own routines.  Also,
         * we don't want to modify timeout based on TM cmds.
         */
        if (cmd->cmd_flags & CFLAG_TM_CMD) {
                return;
        }

        t = Tgt(cmd);
        slot = cmd->cmd_slot;
        pe = mpt->m_slot_free_ae + slot - 1;
        ASSERT(cmd == slots->m_slot[slot]);
        ASSERT((slot > 0) && slot < (mpt->m_max_requests - 1));

        /*
         * remove the cmd.
         */
        if (cmd == slots->m_slot[slot]) {
                NDBG31(("mptsas_remove_cmd: removing cmd=0x%p", (void *)cmd));
        mutex_enter(&mpt->m_slot_freeq_pairp[pe->cpuid].
            m_slot_releq.s.m_fq_mutex);
        NDBG31(("mptsas_remove_cmd0: removing cmd=0x%p", (void *)cmd));
                slots->m_slot[slot] = NULL;
                mpt->m_ncmds--;
        ASSERT(pe->slot == slot);
        list_insert_tail(&mpt->m_slot_freeq_pairp[pe->cpuid].
            m_slot_releq.s.m_fq_list, pe);
        mpt->m_slot_freeq_pairp[pe->cpuid].m_slot_releq.s.m_fq_n++;
        ASSERT(mpt->m_slot_freeq_pairp[pe->cpuid].
            m_slot_releq.s.m_fq_n <= mpt->m_max_requests - 2);
```



```
        mutex_exit(&mpt->m_slot_freeq_pairp[pe->cpuid].
            m_slot_releq.s.m_fq_mutex);

                /*
                 * only decrement per target ncmds if command
                 * has a target associated with it.
                 */
                if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
                mutex_enter(&ptgt->m_tgt_intr_mutex);
                        ptgt->m_t_ncmds--;
                        /*
                         * reset throttle if we just ran an untagged command
                         * to a tagged target
                         */
                        if ((ptgt->m_t_ncmds == 0) &&
                            ((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0)) {
                                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
                        }
                mutex_exit(&ptgt->m_tgt_intr_mutex);
                }

        }

        /*
         * This is all we need to do for ioc commands.
         * The ioc cmds would never be handled in fastpath in ISR, so we make
         * sure the mptsas_return_to_pool() would always be called with
         * m_mutex protected.
         */
        if (cmd->cmd_flags & CFLAG_CMDIOC) {
                ASSERT(mutex_owned(&mpt->m_mutex));
                mptsas_return_to_pool(mpt, cmd);
                return;
        }

        /*
         * Figure out what to set tag Q timeout for...
         *
         * Optimize: If we have duplicate's of same timeout
         * we're using, then we'll use it again until we run
         * out of duplicates.  This should be the normal case
         * for block and raw I/O.
         * If no duplicates, we have to scan through tag que and
         * find the longest timeout value and use it.  This is
         * going to take a while...
         * Add 1 to m_n_slots to account for TM request.
         */
        mutex_enter(&ptgt->m_tgt_intr_mutex);
        if (cmd->cmd_pkt->pkt_time == ptgt->m_timebase) {
                if (--(ptgt->m_dups) == 0) {
                        if (ptgt->m_t_ncmds) {
                                mptsas_cmd_t *ssp;
                                uint_t n = 0;
                                ushort_t nslots = (slots->m_n_slots + 1);
                                ushort_t i;
                                /*
                                 * This crude check assumes we don't do
                                 * this too often which seems reasonable
                                 * for block and raw I/O.
                                 */
                                for (i = 0; i < nslots; i++) {
                                        ssp = slots->m_slot[i];
                                        if (ssp && (Tgt(ssp) == t) &&
                                            (ssp->cmd_pkt->pkt_time > n)) {
                                                n = ssp->cmd_pkt->pkt_time;
                                                ptgt->m_dups = 1;
```

```
7669                                    } else if (ssp && (Tgt(ssp) == t) &&
7670                                        (ssp->cmd_pkt->pkt_time == n)) {
7671                                            ptgt->m_dups++;
7672                                    }
7673                                }
7674                                ptgt->m_timebase = n;
7675                        } else {
7676                                ptgt->m_dups = 0;
7677                                ptgt->m_timebase = 0;
7678                        }
7679                }
7680        }
7681        ptgt->m_timeout = ptgt->m_timebase;

7683        ASSERT(cmd != slots->m_slot[cmd->cmd_slot]);
8109        mutex_exit(&ptgt->m_tgt_intr_mutex);
7684 }

7686 /*
7687  * accept all cmds on the tx_waitq if any and then
7688  * start a fresh request from the top of the device queue.
7689  *
7690  * since there are always cmds queued on the tx_waitq, and rare cmds on
7691  * the instance waitq, so this function should not be invoked in the ISR,
7692  * the mptsas_restart_waitq() is invoked in the ISR instead. otherwise, the
7693  * burden belongs to the IO dispatch CPUs is moved the interrupt CPU.
7694  */
7695 static void
7696 mptsas_restart_hba(mptsas_t *mpt)
7697 {
7698        ASSERT(mutex_owned(&mpt->m_mutex));

7700        mutex_enter(&mpt->m_tx_waitq_mutex);
7701        if (mpt->m_tx_waitq) {
7702                mptsas_accept_tx_waitq(mpt);
7703        }
7704        mutex_exit(&mpt->m_tx_waitq_mutex);
7705        mptsas_restart_waitq(mpt);
7706 }

7708 /*
7709  * start a fresh request from the top of the device queue
7710  */
7711 static void
7712 mptsas_restart_waitq(mptsas_t *mpt)
7713 {
7714        mptsas_cmd_t    *cmd, *next_cmd;
7715        mptsas_target_t *ptgt = NULL;

7717        NDBG1(("mptsas_restart_waitq: mpt=0x%p", (void *)mpt));
8121        NDBG1(("mptsas_restart_hba: mpt=0x%p", (void *)mpt));

7719        ASSERT(mutex_owned(&mpt->m_mutex));

7721        /*
7722         * If there is a reset delay, don't start any cmds.  Otherwise, start
7723         * as many cmds as possible.
7724         * Since SMID 0 is reserved and the TM slot is reserved, the actual max
7725         * commands is m_max_requests - 2.
7726         */
7727        cmd = mpt->m_waitq;

7729        while (cmd != NULL) {
7730                next_cmd = cmd->cmd_linkp;
7731                if (cmd->cmd_flags & CFLAG_PASSTHRU) {
7732                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
```



```
7733                                /*
7734                                 * passthru command get slot need
7735                                 * set CFLAG_PREPARED.
7736                                 */
7737                                cmd->cmd_flags |= CFLAG_PREPARED;
7738                                mptsas_waitq_delete(mpt, cmd);
7739                                mptsas_start_passthru(mpt, cmd);
7740                        }
7741                        cmd = next_cmd;
7742                        continue;
7743                }
7744                if (cmd->cmd_flags & CFLAG_CONFIG) {
7745                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7746                                /*
7747                                 * Send the config page request and delete it
7748                                 * from the waitq.
7749                                 */
7750                                cmd->cmd_flags |= CFLAG_PREPARED;
7751                                mptsas_waitq_delete(mpt, cmd);
7752                                mptsas_start_config_page_access(mpt, cmd);
7753                        }
7754                        cmd = next_cmd;
7755                        continue;
7756                }
7757                if (cmd->cmd_flags & CFLAG_FW_DIAG) {
7758                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7759                                /*
7760                                 * Send the FW Diag request and delete if from
7761                                 * the waitq.
7762                                 */
7763                                cmd->cmd_flags |= CFLAG_PREPARED;
7764                                mptsas_waitq_delete(mpt, cmd);
7765                                mptsas_start_diag(mpt, cmd);
7766                        }
7767                        cmd = next_cmd;
7768                        continue;
7769                }

7771                ptgt = cmd->cmd_tgt_addr;
7772                if (ptgt && (ptgt->m_t_throttle == DRAIN_THROTTLE) &&
8176                if (ptgt) {
8177                        mutex_enter(&mpt->m_intr_mutex);
8178                        mutex_enter(&ptgt->m_tgt_intr_mutex);
8179                        if ((ptgt->m_t_throttle == DRAIN_THROTTLE) &&
7773                    (ptgt->m_t_ncmds == 0)) {
7774                                mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
7775                        }
7776                        if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
7777                            (ptgt && (ptgt->m_reset_delay == 0)) &&
7778                            (ptgt && (ptgt->m_t_ncmds <
7779                            ptgt->m_t_throttle))) {
8183                                if ((ptgt->m_reset_delay == 0) &&
8184                                    (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
8185                                        mutex_exit(&ptgt->m_tgt_intr_mutex);
8186                                        mutex_exit(&mpt->m_intr_mutex);
7780                                if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7781                                        mptsas_waitq_delete(mpt, cmd);
7782                                        (void) mptsas_start_cmd(mpt, cmd);
7783                                }
8191                                        goto out;
7784                        }
8193                                mutex_exit(&ptgt->m_tgt_intr_mutex);
8194                                mutex_exit(&mpt->m_intr_mutex);
8195                        }
8196 out:
7785                cmd = next_cmd;
```

```
7786         }
7787 }

7788 /*
7789  * Cmds are queued if tran_start() doesn't get the m_mutexlock(no wait).
7790  * Accept all those queued cmds before new cmd is accept so that the
7791  * cmds are sent in order.
8202  * mpt tag type lookup
7792  */
7793 static void
7794 mptsas_accept_tx_waitq(mptsas_t *mpt)
8204 static char mptsas_tag_lookup[] =
8205         {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};

8207 /*
8208  * mptsas_start_cmd0 is similar to mptsas_start_cmd, except that, it is called
8209  * without ANY mutex protected, while, mptsas_start_cmd is called with m_mutex
8210  * protected.
8211  *
8212  * the relevant field in ptgt should be protected by m_tgt_intr_mutex in both
8213  * functions.
8214  *
8215  * before the cmds are linked on the slot for monitor as outstanding cmds, they
8216  * are accessed as slab objects, so slab framework ensures the exclusive access,
8217  * and no other mutex is requireed. Linking for monitor and the trigger of dma
8218  * must be done exclusively.
8219  */
8220 static int
8221 mptsas_start_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd)
7795 {
7796         mptsas_cmd_t *cmd;
8223         struct scsi_pkt         *pkt = CMD2PKT(cmd);
8224         uint32_t                control = 0;
8225         int                     n;
8226         caddr_t                 mem;
8227         pMpi2SCSIIORequest_t    io_request;
8228         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
8229         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;
8230         mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
8231         uint16_t                SMID, io_flags = 0;
8232         uint32_t                request_desc_low, request_desc_high;

7798         ASSERT(mutex_owned(&mpt->m_mutex));
7799         ASSERT(mutex_owned(&mpt->m_tx_waitq_mutex));
8234         NDBG1(("mptsas_start_cmd0: cmd=0x%p", (void *)cmd));

7801         /*
7802          * A Bus Reset could occur at any time and flush the tx_waitq,
7803          * so we cannot count on the tx_waitq to contain even one cmd.
7804          * And when the m_tx_waitq_mutex is released and run
7805          * mptsas_accept_pkt(), the tx_waitq may be flushed.
8237          * Set SMID and increment index.  Rollover to 1 instead of 0 if index
8238          * is at the max.  0 is an invalid SMID, so we call the first index 1.
7806          */
7807         cmd = mpt->m_tx_waitq;
7808         for (;;) {
7809                 if ((cmd = mpt->m_tx_waitq) == NULL) {
7810                         mpt->m_tx_draining = 0;
8240         SMID = cmd->cmd_slot;

8242         /*
8243          * It is possible for back to back device reset to
8244          * happen before the reset delay has expired.  That's
8245          * ok, just let the device reset go out on the bus.
8246          */
8247         if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
```



```
8248                 ASSERT(ptgt->m_reset_delay == 0);
8249         }

8251         /*
8252          * if a non-tagged cmd is submitted to an active tagged target
8253          * then drain before submitting this cmd; SCSI-2 allows RQSENSE
8254          * to be untagged
8255          */
8256         mutex_enter(&ptgt->m_tgt_intr_mutex);
8257         if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
8258             (ptgt->m_t_ncmds > 1) &&
8259             ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
8260             (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
8261                 if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
8262                         NDBG23(("target=%d, untagged cmd, start draining\n",
8263                             ptgt->m_devhdl));

8265                         if (ptgt->m_reset_delay == 0) {
8266                                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
8267                         }
8268                         mutex_exit(&ptgt->m_tgt_intr_mutex);

8270                         mutex_enter(&mpt->m_mutex);
8271                         mptsas_remove_cmd(mpt, cmd);
8272                         cmd->cmd_pkt_flags |= FLAG_HEAD;
8273                         mptsas_waitq_add(mpt, cmd);
8274                         mutex_exit(&mpt->m_mutex);
8275                         return (DDI_FAILURE);
8276                 }
8277                 mutex_exit(&ptgt->m_tgt_intr_mutex);
8278                 return (DDI_FAILURE);
8279         }
8280         mutex_exit(&ptgt->m_tgt_intr_mutex);

8282         /*
8283          * Set correct tag bits.
8284          */
8285         if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
8286                 switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
8287                     FLAG_TAGMASK) >> 12)]) {
8288                 case MSG_SIMPLE_QTAG:
8289                         control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
7811                         break;
8291                 case MSG_HEAD_QTAG:
8292                         control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
8293                         break;
8294                 case MSG_ORDERED_QTAG:
8295                         control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
8296                         break;
8297                 default:
8298                         mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
8299                         break;
7812                 }
7813                 if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL) {
7814                         mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
8301         } else {
8302                 if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
8303                         ptgt->m_t_throttle = 1;
7815                 }
7816                 cmd->cmd_linkp = NULL;
7817                 mutex_exit(&mpt->m_tx_waitq_mutex);
7818                 if (mptsas_accept_pkt(mpt, cmd) != TRAN_ACCEPT)
7819                         cmn_err(CE_WARN, "mpt: mptsas_accept_tx_waitq: failed "
7820                             "to accept cmd on queue\n");
7821                 mutex_enter(&mpt->m_tx_waitq_mutex);
8305                 control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
```

```
7822         }
7823 }

8308         if (cmd->cmd_pkt_flags & FLAG_TLR) {
8309                 control |= MPI2_SCSIIO_CONTROL_TLR_ON;
8310         }

7826 /*
7827  * mpt tag type lookup
8312         mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
8313         io_request = (pMpi2SCSIIORequest_t)mem;

8315         bzero(io_request, sizeof (Mpi2SCSIIORequest_t));
8316         ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
8317             (MPI2_SCSI_IO_REQUEST, SGL) / 4);
8318         mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
8319             MPI2_FUNCTION_SCSI_IO_REQUEST);

8321         (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
8322             io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);

8324         io_flags = cmd->cmd_cdblen;
8325         ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
8326         /*
8327          * setup the Scatter/Gather DMA list for this request
7828  */
7829 static char mptsas_tag_lookup[] =
7830         {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};
8329         if (cmd->cmd_cookiec > 0) {
8330                 mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
8331         } else {
8332                 ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
8333                     ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
8334                     MPI2_SGE_FLAGS_END_OF_BUFFER |
8335                     MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
8336                     MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
8337         }

8339         /*
8340          * save ARQ information
8341          */
8342         ddi_put8(acc_hdl, &io_request->SenseBufferLength, cmd->cmd_rqslen);
8343         if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
8344             (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
8345                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
8346                     cmd->cmd_ext_arqcookie.dmac_address);
8347         } else {
8348                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
8349                     cmd->cmd_arqcookie.dmac_address);
8350         }

8352         ddi_put32(acc_hdl, &io_request->Control, control);

8354         NDBG31(("starting message=0x%p, with cmd=0x%p",
8355             (void *)(uintptr_t)mpt->m_req_frame_dma_addr, (void *)cmd));

8357         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);

8359         /*
8360          * Build request descriptor and write it to the request desc post reg.
8361          */
8362         request_desc_low = (SMID << 16) + MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
8363         request_desc_high = ptgt->m_devhdl << 16;

8365         mutex_enter(&mpt->m_mutex);
8366         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
```



```
8367         MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
8368         mutex_exit(&mpt->m_mutex);

8370         /*
8371          * Start timeout.
8372          */
8373         mutex_enter(&ptgt->m_tgt_intr_mutex);
8374 #ifdef MPTSAS_TEST
8375         /*
8376          * Temporarily set timebase = 0;  needed for
8377          * timeout torture test.
8378          */
8379         if (mptsas_test_timeouts) {
8380                 ptgt->m_timebase = 0;
8381         }
8382 #endif
8383         n = pkt->pkt_time - ptgt->m_timebase;

8385         if (n == 0) {
8386                 (ptgt->m_dups)++;
8387                 ptgt->m_timeout = ptgt->m_timebase;
8388         } else if (n > 0) {
8389                 ptgt->m_timeout =
8390                     ptgt->m_timebase = pkt->pkt_time;
8391                 ptgt->m_dups = 1;
8392         } else if (n < 0) {
8393                 ptgt->m_timeout = ptgt->m_timebase;
8394         }
8395 #ifdef MPTSAS_TEST
8396         /*
8397          * Set back to a number higher than
8398          * mptsas_scsi_watchdog_tick
8399          * so timeouts will happen in mptsas_watchsubr
8400          */
8401         if (mptsas_test_timeouts) {
8402                 ptgt->m_timebase = 60;
8403         }
8404 #endif
8405         mutex_exit(&ptgt->m_tgt_intr_mutex);

8407         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
8408             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
8409                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8410                 return (DDI_FAILURE);
8411         }
8412         return (DDI_SUCCESS);
8413 }

7832 static int
7833 mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
7834 {
7835         struct scsi_pkt         *pkt = CMD2PKT(cmd);
7836         uint32_t                control = 0;
7837         int                     n;
7838         caddr_t                 mem;
7839         pMpi2SCSIIORequest_t    io_request;
7840         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
7841         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;
7842         mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
7843         uint16_t                SMID, io_flags = 0;
7844         uint32_t                request_desc_low, request_desc_high;

7846         NDBG1(("mptsas_start_cmd: cmd=0x%p", (void *)cmd));

7848         /*
7849          * Set SMID and increment index.  Rollover to 1 instead of 0 if index
```

```
7850          * is at the max.  0 is an invalid SMID, so we call the first index 1.
7851          */
7852         SMID = cmd->cmd_slot;

7854         /*
7855          * It is possible for back to back device reset to
7856          * happen before the reset delay has expired.  That's
7857          * ok, just let the device reset go out on the bus.
7858          */
7859         if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
7860                 ASSERT(ptgt->m_reset_delay == 0);
7861         }

7863         /*
7864          * if a non-tagged cmd is submitted to an active tagged target
7865          * then drain before submitting this cmd; SCSI-2 allows RQSENSE
7866          * to be untagged
7867          */
8451         mutex_enter(&ptgt->m_tgt_intr_mutex);
7868         if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
7869             (ptgt->m_t_ncmds > 1) &&
7870             ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
7871             (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
7872                 if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
7873                         NDBG23(("target=%d, untagged cmd, start draining\n",
7874                             ptgt->m_devhdl));

7876                         if (ptgt->m_reset_delay == 0) {
7877                                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
7878                         }
8463                         mutex_exit(&ptgt->m_tgt_intr_mutex);

7880                         mptsas_remove_cmd(mpt, cmd);
7881                         cmd->cmd_pkt_flags |= FLAG_HEAD;
7882                         mptsas_waitq_add(mpt, cmd);
8468                         return (DDI_FAILURE);
7883                 }
8470                 mutex_exit(&ptgt->m_tgt_intr_mutex);
7884                 return (DDI_FAILURE);
7885         }
8473         mutex_exit(&ptgt->m_tgt_intr_mutex);

7887         /*
7888          * Set correct tag bits.
7889          */
7890         if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
7891                 switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
7892                     FLAG_TAGMASK) >> 12)]) {
7893                 case MSG_SIMPLE_QTAG:
7894                         control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
7895                         break;
7896                 case MSG_HEAD_QTAG:
7897                         control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
7898                         break;
7899                 case MSG_ORDERED_QTAG:
7900                         control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
7901                         break;
7902                 default:
7903                         mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
7904                         break;
7905                 }
7906         } else {
7907                 if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
7908                         ptgt->m_t_throttle = 1;
7909                 }
7910                 control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
```



```
7911         }

7913         if (cmd->cmd_pkt_flags & FLAG_TLR) {
7914                 control |= MPI2_SCSIIO_CONTROL_TLR_ON;
7915         }

7917         mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
7918         io_request = (pMpi2SCSIIORequest_t)mem;

7920         bzero(io_request, sizeof (Mpi2SCSIIORequest_t));
7921         ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
7922             (MPI2_SCSI_IO_REQUEST, SGL) / 4);
7923         mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
7924             MPI2_FUNCTION_SCSI_IO_REQUEST);

7926         (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
7927             io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);

7929         io_flags = cmd->cmd_cdblen;
7930         ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
7931         /*
7932          * setup the Scatter/Gather DMA list for this request
7933          */
7934         if (cmd->cmd_cookiec > 0) {
7935                 mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
7936         } else {
7937                 ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
7938                     ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
7939                     MPI2_SGE_FLAGS_END_OF_BUFFER |
7940                     MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
7941                     MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
7942         }

7944         /*
7945          * save ARQ information
7946          */
7947         ddi_put8(acc_hdl, &io_request->SenseBufferLength, cmd->cmd_rqslen);
7948         if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
7949             (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
7950                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
7951                     cmd->cmd_ext_arqcookie.dmac_address);
7952         } else {
7953                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
7954                     cmd->cmd_arqcookie.dmac_address);
7955         }

7957         ddi_put32(acc_hdl, &io_request->Control, control);

7959         NDBG31(("starting message=0x%p, with cmd=0x%p",
7960             (void *)(uintptr_t)mpt->m_req_frame_dma_addr, (void *)cmd));

7962         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);

7964         /*
7965          * Build request descriptor and write it to the request desc post reg.
7966          */
7967         request_desc_low = (SMID << 16) + MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
7968         request_desc_high = ptgt->m_devhdl << 16;

8558         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
7969         MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);

7971         /*
7972          * Start timeout.
7973          */
8564         mutex_enter(&ptgt->m_tgt_intr_mutex);
```

```
7974 #ifdef MPTSAS_TEST
7975         /*
7976          * Temporarily set timebase = 0;  needed for
7977          * timeout torture test.
7978          */
7979         if (mptsas_test_timeouts) {
7980                 ptgt->m_timebase = 0;
7981         }
7982 #endif
7983         n = pkt->pkt_time - ptgt->m_timebase;

7985         if (n == 0) {
7986                 (ptgt->m_dups)++;
7987                 ptgt->m_timeout = ptgt->m_timebase;
7988         } else if (n > 0) {
7989                 ptgt->m_timeout =
7990                     ptgt->m_timebase = pkt->pkt_time;
7991                 ptgt->m_dups = 1;
7992         } else if (n < 0) {
7993                 ptgt->m_timeout = ptgt->m_timebase;
7994         }
7995 #ifdef MPTSAS_TEST
7996         /*
7997          * Set back to a number higher than
7998          * mptsas_scsi_watchdog_tick
7999          * so timeouts will happen in mptsas_watchsubr
8000          */
8001         if (mptsas_test_timeouts) {
8002                 ptgt->m_timebase = 60;
8003         }
8004 #endif
8596         mutex_exit(&ptgt->m_tgt_intr_mutex);

8006         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
8007             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
8008                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
8009                 return (DDI_FAILURE);
8010         }
8011         return (DDI_SUCCESS);
8012 }
```

_____unchanged_portion_omitted_

```
8050 /*
8051  * move the current global doneq to the doneq of thead[t]
8643  * move the current global doneq to the doneq of thread[t]
8052  */
8053 static void
8054 mptsas_doneq_mv(mptsas_t *mpt, uint64_t t)
8055 {
8056         mptsas_cmd_t                    *cmd;
8057         mptsas_doneq_thread_list_t      *item = &mpt->m_doneq_thread_id[t];

8059         ASSERT(mutex_owned(&item->mutex));
8652         mutex_enter(&mpt->m_intr_mutex);
8060         while ((cmd = mpt->m_doneq) != NULL) {
8061                 if ((mpt->m_doneq = cmd->cmd_linkp) == NULL) {
8062                         mpt->m_donetail = &mpt->m_doneq;
8063                 }
8064                 cmd->cmd_linkp = NULL;
8065                 *item->donetail = cmd;
8066                 item->donetail = &cmd->cmd_linkp;
8067                 mpt->m_doneq_len--;
8068                 item->len++;
8069         }
8663         mutex_exit(&mpt->m_intr_mutex);
8070 }
```

_____unchanged_portion_omitted_


```
8143 /*
8144  * These routines manipulate the queue of commands that
8145  * are waiting for their completion routines to be called.
8146  * The queue is usually in FIFO order but on an MP system
8147  * it's possible for the completion routines to get out
8148  * of order. If that's a problem you need to add a global
8149  * mutex around the code that calls the completion routine
8150  * in the interrupt handler.
8738  * mptsas_doneq_add0 is similar to mptsas_doneq_add except that it is called
8739  * where m_intr_mutex has already been held.
8151  */
8152 static void
8153 mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8741 static inline void
8742 mptsas_doneq_add0(mptsas_t *mpt, mptsas_cmd_t *cmd)
8154 {
8155         struct scsi_pkt *pkt = CMD2PKT(cmd);

8157         NDBG31(("mptsas_doneq_add: cmd=0x%p", (void *)cmd));
8746         NDBG31(("mptsas_doneq_add0: cmd=0x%p", (void *)cmd));

8159         ASSERT((cmd->cmd_flags & CFLAG_COMPLETED) == 0);
8160         cmd->cmd_linkp = NULL;
8161         cmd->cmd_flags |= CFLAG_FINISHED;
8162         cmd->cmd_flags &= ~CFLAG_IN_TRANSPORT;

8164         mptsas_fma_check(mpt, cmd);

8166         /*
8167          * only add scsi pkts that have completion routines to
8168          * the doneq.  no intr cmds do not have callbacks.
8169          */
8170         if (pkt && (pkt->pkt_comp)) {
8171                 *mpt->m_donetail = cmd;
8172                 mpt->m_donetail = &cmd->cmd_linkp;
8173                 mpt->m_doneq_len++;
8174         }
8175 }

8764 /*
8765  * These routines manipulate the queue of commands that
8766  * are waiting for their completion routines to be called.
8767  * The queue is usually in FIFO order but on an MP system
8768  * it's possible for the completion routines to get out
8769  * of order. If that's a problem you need to add a global
8770  * mutex around the code that calls the completion routine
8771  * in the interrupt handler.
8772  */
8773 static void
8774 mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8775 {
8776         ASSERT(mutex_owned(&mpt->m_mutex));

8778         mptsas_fma_check(mpt, cmd);

8780         mutex_enter(&mpt->m_intr_mutex);
8781         mptsas_doneq_add0(mpt, cmd);
8782         mutex_exit(&mpt->m_intr_mutex);
8783 }

8177 static mptsas_cmd_t *
8178 mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t)
8179 {
8180         mptsas_cmd_t                    *cmd;
8181         mptsas_doneq_thread_list_t      *item = &mpt->m_doneq_thread_id[t];
```

```
8183         /* pop one off the done queue */
8184         if ((cmd = item->doneq) != NULL) {
8185                 /* if the queue is now empty fix the tail pointer */
8186                 NDBG31(("mptsas_doneq_thread_rm: cmd=0x%p", (void *)cmd));
8187                 if ((item->doneq = cmd->cmd_linkp) == NULL) {
8188                         item->donetail = &item->doneq;
8189                 }
8190                 cmd->cmd_linkp = NULL;
8191                 item->len--;
8192         }
8193         return (cmd);
8194 }

8196 static void
8197 mptsas_doneq_empty(mptsas_t *mpt)
8198 {
8807         mutex_enter(&mpt->m_intr_mutex);
8199         if (mpt->m_doneq && !mpt->m_in_callback) {
8200                 mptsas_cmd_t    *cmd, *next;
8201                 struct scsi_pkt *pkt;

8203                 mpt->m_in_callback = 1;
8204                 cmd = mpt->m_doneq;
8205                 mpt->m_doneq = NULL;
8206                 mpt->m_donetail = &mpt->m_doneq;
8207                 mpt->m_doneq_len = 0;

8818                 mutex_exit(&mpt->m_intr_mutex);

8820                 /*
8821                  * ONLY in ISR, is it called without m_mutex held, otherwise,
8822                  * it is always called with m_mutex held.
8823                  */
8824                 if ((curthread->t_flag & T_INTR_THREAD) == 0)
8209                 mutex_exit(&mpt->m_mutex);
8210                 /*
8211                  * run the completion routines of all the
8212                  * completed commands
8213                  */
8214                 while (cmd != NULL) {
8215                         next = cmd->cmd_linkp;
8216                         cmd->cmd_linkp = NULL;
8217                         /* run this command's completion routine */
8218                         cmd->cmd_flags |= CFLAG_COMPLETED;
8219                         pkt = CMD2PKT(cmd);
8220                         mptsas_pkt_comp(pkt, cmd);
8221                         cmd = next;
8222                 }
8839                 if ((curthread->t_flag & T_INTR_THREAD) == 0)
8223                 mutex_enter(&mpt->m_mutex);
8224                 mpt->m_in_callback = 0;
8842                 return;
8225         }
8844         mutex_exit(&mpt->m_intr_mutex);
8226 }

8228 /*
8229  * These routines manipulate the target's queue of pending requests
8230  */
8231 void
8232 mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8233 {
8234         NDBG7(("mptsas_waitq_add: cmd=0x%p", (void *)cmd));
8235         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8236         cmd->cmd_queued = TRUE;
```



```
8237         if (ptgt)
8238                 ptgt->m_t_nwait++;
8239         if (cmd->cmd_pkt_flags & FLAG_HEAD) {
8859                 mutex_enter(&mpt->m_intr_mutex);
8240                 if ((cmd->cmd_linkp = mpt->m_waitq) == NULL) {
8241                         mpt->m_waitqtail = &cmd->cmd_linkp;
8242                 }
8243                 mpt->m_waitq = cmd;
8864                 mutex_exit(&mpt->m_intr_mutex);
8244         } else {
8245                 cmd->cmd_linkp = NULL;
8246                 *(mpt->m_waitqtail) = cmd;
8247                 mpt->m_waitqtail = &cmd->cmd_linkp;
8248         }
8249 }

8251 static mptsas_cmd_t *
8252 mptsas_waitq_rm(mptsas_t *mpt)
8253 {
8254         mptsas_cmd_t    *cmd;
8255         mptsas_target_t *ptgt;
8256         NDBG7(("mptsas_waitq_rm"));

8879         mutex_enter(&mpt->m_intr_mutex);
8258         MPTSAS_WAITQ_RM(mpt, cmd);
8881         mutex_exit(&mpt->m_intr_mutex);

8260         NDBG7(("mptsas_waitq_rm: cmd=0x%p", (void *)cmd));
8261         if (cmd) {
8262                 ptgt = cmd->cmd_tgt_addr;
8263                 if (ptgt) {
8264                         ptgt->m_t_nwait--;
8265                         ASSERT(ptgt->m_t_nwait >= 0);
8266                 }
8267         }
8268         return (cmd);
8269 }

8271 /*
8272  * remove specified cmd from the middle of the wait queue.
8273  */
8274 static void
8275 mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8276 {
8277         mptsas_cmd_t    *prevp = mpt->m_waitq;
8278         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

8280         NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8281             (void *)mpt, (void *)cmd));
8282         if (ptgt) {
8283                 ptgt->m_t_nwait--;
8284                 ASSERT(ptgt->m_t_nwait >= 0);
8285         }

8287         if (prevp == cmd) {
8911                 mutex_enter(&mpt->m_intr_mutex);
8288                 if ((mpt->m_waitq = cmd->cmd_linkp) == NULL)
8289                         mpt->m_waitqtail = &mpt->m_waitq;
8914                 mutex_exit(&mpt->m_intr_mutex);

8291                 cmd->cmd_linkp = NULL;
8292                 cmd->cmd_queued = FALSE;
8293                 NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8294                     (void *)mpt, (void *)cmd));
8295                 return;
8296         }
```

```
8298         while (prevp != NULL) {
8299                 if (prevp->cmd_linkp == cmd) {
8300                         if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8301                                 mpt->m_waitqtail = &prevp->cmd_linkp;

8303                         cmd->cmd_linkp = NULL;
8304                         cmd->cmd_queued = FALSE;
8305                         NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8306                             (void *)mpt, (void *)cmd));
8307                         return;
8308                 }
8309                 prevp = prevp->cmd_linkp;
8310         }
8311         cmn_err(CE_PANIC, "mpt: mptsas_waitq_delete: queue botch");
8312 }

8314 static mptsas_cmd_t *
8315 mptsas_tx_waitq_rm(mptsas_t *mpt)
8316 {
8317         mptsas_cmd_t *cmd;
8318         NDBG7(("mptsas_tx_waitq_rm"));

8320         MPTSAS_TX_WAITQ_RM(mpt, cmd);

8322         NDBG7(("mptsas_tx_waitq_rm: cmd=0x%p", (void *)cmd));

8324         return (cmd);
8325 }

8327 /*
8328  * remove specified cmd from the middle of the tx_waitq.
8329  */
8330 static void
8331 mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8332 {
8333         mptsas_cmd_t *prevp = mpt->m_tx_waitq;

8335         NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8336             (void *)mpt, (void *)cmd));

8338         if (prevp == cmd) {
8339                 if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL)
8340                         mpt->m_tx_waitqtail = &mpt->m_tx_waitq;

8342                 cmd->cmd_linkp = NULL;
8343                 cmd->cmd_queued = FALSE;
8344                 NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8345                     (void *)mpt, (void *)cmd));
8346                 return;
8347         }

8349         while (prevp != NULL) {
8350                 if (prevp->cmd_linkp == cmd) {
8351                         if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8352                                 mpt->m_tx_waitqtail = &prevp->cmd_linkp;

8354                         cmd->cmd_linkp = NULL;
8355                         cmd->cmd_queued = FALSE;
8356                         NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8357                             (void *)mpt, (void *)cmd));
8358                         return;
8359                 }
8360                 prevp = prevp->cmd_linkp;
8361         }
8362         cmn_err(CE_PANIC, "mpt: mptsas_tx_waitq_delete: queue botch");
```



```
8363 }

8365 /*
8366  * device and bus reset handling
8367  *
8368  * Notes:
8369  *      - RESET_ALL:    reset the controller
8370  *      - RESET_TARGET: reset the target specified in scsi_address
8371  */
8372 static int
8373 mptsas_scsi_reset(struct scsi_address *ap, int level)
8374 {
8375         mptsas_t                *mpt = ADDR2MPT(ap);
8376         int                     rval;
8377         mptsas_tgt_private_t    *tgt_private;
8378         mptsas_target_t         *ptgt = NULL;

8380         tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->tran_tgt_private;
8381         ptgt = tgt_private->t_private;
8382         if (ptgt == NULL) {
8383                 return (FALSE);
8384         }
8385         NDBG22(("mptsas_scsi_reset: target=%d level=%d", ptgt->m_devhdl,
8386             level));

8388         mutex_enter(&mpt->m_mutex);
8389         /*
8390          * if we are not in panic set up a reset delay for this target
8391          */
8392         if (!ddi_in_panic()) {
8393                 mptsas_setup_bus_reset_delay(mpt);
8394         } else {
8395                 drv_usecwait(mpt->m_scsi_reset_delay * 1000);
8396         }
8397         rval = mptsas_do_scsi_reset(mpt, ptgt->m_devhdl);
8398         mutex_exit(&mpt->m_mutex);

8400         /*
8401          * The transport layer expect to only see TRUE and
8402          * FALSE. Therefore, we will adjust the return value
8403          * if mptsas_do_scsi_reset returns FAILED.
8404          */
8405         if (rval == FAILED)
8406                 rval = FALSE;
8407         return (rval);
8408 }
_____unchanged_portion_omitted_

8505 /*
8506  * Clean up from a device reset.
8507  * For the case of target reset, this function clears the waitq of all
8508  * commands for a particular target.   For the case of abort task set, this
8509  * function clears the waitq of all commonds for a particular target/lun.
8510  */
8511 static void
8512 mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
8513 {
8514         mptsas_slots_t  *slots = mpt->m_active;
8515         mptsas_cmd_t    *cmd, *next_cmd;
8516         int             slot;
8517         uchar_t         reason;
8518         uint_t          stat;

8520         NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));

8522         /*
```

```
8523          * Make sure the I/O Controller has flushed all cmds
8524          * that are associated with this target for a target reset
8525          * and target/lun for abort task set.
8526          * Account for TM requests, which use the last SMID.
8527          */
9102         mutex_enter(&mpt->m_intr_mutex);
8528         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8529                 if ((cmd = slots->m_slot[slot]) == NULL)
9104                 if ((cmd = slots->m_slot[slot]) == NULL) {
8530                         continue;
9106                 }
8531                 reason = CMD_RESET;
8532                 stat = STAT_DEV_RESET;
8533                 switch (tasktype) {
8534                 case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8535                         if (Tgt(cmd) == target) {
8536                                 if (cmd->cmd_tgt_addr->m_timeout < 0) {
8537                                         /*
8538                                          * When timeout requested, propagate
8539                                          * proper reason and statistics to
8540                                          * target drivers.
8541                                          */
8542                                         reason = CMD_TIMEOUT;
8543                                         stat |= STAT_TIMEOUT;
8544                                 }
8545                                 NDBG25(("mptsas_flush_target discovered non-"
8546                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8547                                     tasktype));
8548                                 mptsas_dump_cmd(mpt, cmd);
8549                                 mptsas_remove_cmd(mpt, cmd);
9116                                 mptsas_remove_cmd0(mpt, cmd);
8550                                 mptsas_set_pkt_reason(mpt, cmd, reason, stat);
8551                                 mptsas_doneq_add(mpt, cmd);
9118                                 mptsas_doneq_add0(mpt, cmd);
8552                         }
8553                         break;
8554                 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8555                         reason = CMD_ABORTED;
8556                         stat = STAT_ABORTED;
8557                         /*FALLTHROUGH*/
8558                 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8559                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {

8561                                 NDBG25(("mptsas_flush_target discovered non-"
8562                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8563                                     tasktype));
8564                                 mptsas_dump_cmd(mpt, cmd);
8565                                 mptsas_remove_cmd(mpt, cmd);
9132                                 mptsas_remove_cmd0(mpt, cmd);
8566                                 mptsas_set_pkt_reason(mpt, cmd, reason,
8567                                     stat);
8568                                 mptsas_doneq_add(mpt, cmd);
9135                                 mptsas_doneq_add0(mpt, cmd);
8569                         }
8570                         break;
8571                 default:
8572                         break;
8573                 }
8574         }
9142         mutex_exit(&mpt->m_intr_mutex);

8576         /*
8577          * Flush the waitq and tx_waitq of this target's cmds
9145          * Flush the waitq of this target's cmds
8578          */
8579         cmd = mpt->m_waitq;
```



```
8581         reason = CMD_RESET;
8582         stat = STAT_DEV_RESET;

8584         switch (tasktype) {
8585         case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8586                 while (cmd != NULL) {
8587                         next_cmd = cmd->cmd_linkp;
8588                         if (Tgt(cmd) == target) {
8589                                 mptsas_waitq_delete(mpt, cmd);
8590                                 mptsas_set_pkt_reason(mpt, cmd,
8591                                     reason, stat);
8592                                 mptsas_doneq_add(mpt, cmd);
8593                         }
8594                         cmd = next_cmd;
8595                 }
8596                 mutex_enter(&mpt->m_tx_waitq_mutex);
8597                 cmd = mpt->m_tx_waitq;
8598                 while (cmd != NULL) {
8599                         next_cmd = cmd->cmd_linkp;
8600                         if (Tgt(cmd) == target) {
8601                                 mptsas_tx_waitq_delete(mpt, cmd);
8602                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8603                                 mptsas_set_pkt_reason(mpt, cmd,
8604                                     reason, stat);
8605                                 mptsas_doneq_add(mpt, cmd);
8606                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8607                         }
8608                         cmd = next_cmd;
8609                 }
8610                 mutex_exit(&mpt->m_tx_waitq_mutex);
8611                 break;
8612         case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8613                 reason = CMD_ABORTED;
8614                 stat =  STAT_ABORTED;
8615                 /*FALLTHROUGH*/
8616         case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8617                 while (cmd != NULL) {
8618                         next_cmd = cmd->cmd_linkp;
8619                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8620                                 mptsas_waitq_delete(mpt, cmd);
8621                                 mptsas_set_pkt_reason(mpt, cmd,
8622                                     reason, stat);
8623                                 mptsas_doneq_add(mpt, cmd);
8624                         }
8625                         cmd = next_cmd;
8626                 }
8627                 mutex_enter(&mpt->m_tx_waitq_mutex);
8628                 cmd = mpt->m_tx_waitq;
8629                 while (cmd != NULL) {
8630                         next_cmd = cmd->cmd_linkp;
8631                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8632                                 mptsas_tx_waitq_delete(mpt, cmd);
8633                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8634                                 mptsas_set_pkt_reason(mpt, cmd,
8635                                     reason, stat);
8636                                 mptsas_doneq_add(mpt, cmd);
8637                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8638                         }
8639                         cmd = next_cmd;
8640                 }
8641                 mutex_exit(&mpt->m_tx_waitq_mutex);
8642                 break;
8643         default:
8644                 mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
8645                     tasktype);
```

```
8646                 break;
8647         }
8648 }

8650 /*
8651  * Clean up hba state, abort all outstanding command and commands in waitq
8652  * reset timeout of all targets.
8653  */
8654 static void
8655 mptsas_flush_hba(mptsas_t *mpt)
8656 {
8657         mptsas_slots_t  *slots = mpt->m_active;
8658         mptsas_cmd_t    *cmd;
8659         int             slot;

8661         NDBG25(("mptsas_flush_hba"));

8663         /*
8664          * The I/O Controller should have already sent back
8665          * all commands via the scsi I/O reply frame.  Make
8666          * sure all commands have been flushed.
8667          * Account for TM request, which use the last SMID.
8668          */
9207         mutex_enter(&mpt->m_intr_mutex);
8669         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8670                 if ((cmd = slots->m_slot[slot]) == NULL)
9209                 if ((cmd = slots->m_slot[slot]) == NULL) {
8671                         continue;
9211                 }

8673                 if (cmd->cmd_flags & CFLAG_CMDIOC) {
8674                         /*
8675                          * Need to make sure to tell everyone that might be
8676                          * waiting on this command that it's going to fail.  If
8677                          * we get here, this command will never timeout because
8678                          * the active command table is going to be re-allocated,
8679                          * so there will be nothing to check against a time out.
8680                          * Instead, mark the command as failed due to reset.
8681                          */
8682                         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
8683                             STAT_BUS_RESET);
8684                         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8685                             (cmd->cmd_flags & CFLAG_CONFIG) ||
8686                             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8687                                 cmd->cmd_flags |= CFLAG_FINISHED;
8688                                 cv_broadcast(&mpt->m_passthru_cv);
8689                                 cv_broadcast(&mpt->m_config_cv);
8690                                 cv_broadcast(&mpt->m_fw_diag_cv);
8691                         }
8692                         continue;
8693                 }

8695                 NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
8696                     slot));
8697                 mptsas_dump_cmd(mpt, cmd);

8699                 mptsas_remove_cmd(mpt, cmd);
9239                 mptsas_remove_cmd0(mpt, cmd);
8700                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8701                 mptsas_doneq_add(mpt, cmd);
9241                 mptsas_doneq_add0(mpt, cmd);
8702         }
9243         mutex_exit(&mpt->m_intr_mutex);

8704         /*
8705          * Flush the waitq.
```



```
8706          */
8707         while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
8708                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8709                 if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8710                     (cmd->cmd_flags & CFLAG_CONFIG) ||
8711                     (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8712                         cmd->cmd_flags |= CFLAG_FINISHED;
8713                         cv_broadcast(&mpt->m_passthru_cv);
8714                         cv_broadcast(&mpt->m_config_cv);
8715                         cv_broadcast(&mpt->m_fw_diag_cv);
8716                 } else {
8717                         mptsas_doneq_add(mpt, cmd);
8718                 }
8719         }

8721         /*
8722          * Flush the tx_waitq
8723          */
8724         mutex_enter(&mpt->m_tx_waitq_mutex);
8725         while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
8726                 mutex_exit(&mpt->m_tx_waitq_mutex);
8727                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8728                 mptsas_doneq_add(mpt, cmd);
8729                 mutex_enter(&mpt->m_tx_waitq_mutex);
8730         }
8731         mutex_exit(&mpt->m_tx_waitq_mutex);

8733         /*
8734          * Drain the taskqs prior to reallocating resources.
8735          */
8736         mutex_exit(&mpt->m_mutex);
8737         ddi_taskq_wait(mpt->m_event_taskq);
8738         ddi_taskq_wait(mpt->m_dr_taskq);
8739         mutex_enter(&mpt->m_mutex);
8740 }
______unchanged_portion_omitted_

8779 static void
8780 mptsas_setup_bus_reset_delay(mptsas_t *mpt)
8781 {
8782         mptsas_target_t *ptgt = NULL;

8784         NDBG22(("mptsas_setup_bus_reset_delay"));
8785         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
8786             MPTSAS_HASH_FIRST);
8787         while (ptgt != NULL) {
9309                 mutex_enter(&ptgt->m_tgt_intr_mutex);
8788                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
8789                 ptgt->m_reset_delay = mpt->m_scsi_reset_delay;
9312                 mutex_exit(&ptgt->m_tgt_intr_mutex);

8791                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
8792                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
8793         }

8795         mptsas_start_watch_reset_delay();
8796 }
______unchanged_portion_omitted_

8833 static int
8834 mptsas_watch_reset_delay_subr(mptsas_t *mpt)
8835 {
8836         int             done = 0;
8837         int             restart = 0;
8838         mptsas_target_t *ptgt = NULL;
```

```
8840         NDBG22(("mptsas_watch_reset_delay_subr: mpt=0x%p", (void *)mpt));

8842         ASSERT(mutex_owned(&mpt->m_mutex));

8844         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
8845             MPTSAS_HASH_FIRST);
8846         while (ptgt != NULL) {
9370                 mutex_enter(&ptgt->m_tgt_intr_mutex);
8847                 if (ptgt->m_reset_delay != 0) {
8848                         ptgt->m_reset_delay -=
8849                             MPTSAS_WATCH_RESET_DELAY_TICK;
8850                         if (ptgt->m_reset_delay <= 0) {
8851                                 ptgt->m_reset_delay = 0;
8852                                 mptsas_set_throttle(mpt, ptgt,
8853                                     MAX_THROTTLE);
8854                                 restart++;
8855                         } else {
8856                                 done = -1;
8857                         }
8858                 }
9383                 mutex_exit(&ptgt->m_tgt_intr_mutex);

8860                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
8861                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
8862         }

8864         if (restart > 0) {
8865                 mptsas_restart_hba(mpt);
8866         }
8867         return (done);
8868 }
_____unchanged_portion_omitted_

8916 static int
8917 mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun, struct scsi_pkt *pkt)
8918 {
8919         mptsas_cmd_t    *sp = NULL;
8920         mptsas_slots_t  *slots = mpt->m_active;
8921         int             rval = FALSE;

8923         ASSERT(mutex_owned(&mpt->m_mutex));

8925         /*
8926          * Abort the command pkt on the target/lun in ap.  If pkt is
8927          * NULL, abort all outstanding commands on that target/lun.
8928          * If you can abort them, return 1, else return 0.
8929          * Each packet that's aborted should be sent back to the target
8930          * driver through the callback routine, with pkt_reason set to
8931          * CMD_ABORTED.
8932          *
8933          * abort cmd pkt on HBA hardware; clean out of outstanding
8934          * command lists, etc.
8935          */
8936         if (pkt != NULL) {
8937                 /* abort the specified packet */
8938                 sp = PKT2CMD(pkt);

8940                 if (sp->cmd_queued) {
8941                         NDBG23(("mptsas_do_scsi_abort: queued sp=0x%p aborted",
8942                             (void *)sp));
8943                         mptsas_waitq_delete(mpt, sp);
8944                         mptsas_set_pkt_reason(mpt, sp, CMD_ABORTED,
8945                             STAT_ABORTED);
8946                         mptsas_doneq_add(mpt, sp);
8947                         rval = TRUE;
8948                         goto done;
```



```
8949                 }

8951                 /*
8952                  * Have mpt firmware abort this command
8953                  */

9479                 mutex_enter(&mpt->m_intr_mutex);
8955                 if (slots->m_slot[sp->cmd_slot] != NULL) {
9481                         mutex_exit(&mpt->m_intr_mutex);
8956                         rval = mptsas_ioc_task_management(mpt,
8957                             MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK, target,
8958                             lun, NULL, 0, 0);

8960                         /*
8961                          * The transport layer expects only TRUE and FALSE.
8962                          * Therefore, if mptsas_ioc_task_management returns
8963                          * FAILED we will return FALSE.
8964                          */
8965                         if (rval == FAILED)
8966                                 rval = FALSE;
8967                         goto done;
8968                 }
9495                 mutex_exit(&mpt->m_intr_mutex);
8969         }

8971         /*
8972          * If pkt is NULL then abort task set
8973          */
8974         rval = mptsas_ioc_task_management(mpt,
8975             MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET, target, lun, NULL, 0, 0);

8977         /*
8978          * The transport layer expects only TRUE and FALSE.
8979          * Therefore, if mptsas_ioc_task_management returns
8980          * FAILED we will return FALSE.
8981          */
8982         if (rval == FAILED)
8983                 rval = FALSE;

8985 #ifdef MPTSAS_TEST
8986         if (rval && mptsas_test_stop) {
8987                 debug_enter("mptsas_do_scsi_abort");
8988         }
8989 #endif

8991 done:
8992         mptsas_doneq_empty(mpt);
8993         return (rval);
8994 }
_____unchanged_portion_omitted_

9071 /*
9072  * (*tran_setcap).  Set the capability named to the value given.
9073  */
9074 static int
9075 mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value, int tgtonly)
9076 {
9077         mptsas_t        *mpt = ADDR2MPT(ap);
9078         int             ckey;
9079         int             rval = FALSE;
9607         mptsas_target_t *ptgt;

9081         NDBG24(("mptsas_scsi_setcap: target=%d, cap=%s value=%x tgtonly=%x",
9082             ap->a_target, cap, value, tgtonly));

9084         if (!tgtonly) {
```

```
9085                    return (rval);
9086            }

9088            mutex_enter(&mpt->m_mutex);

9090            if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9091                    mutex_exit(&mpt->m_mutex);
9092                    return (UNDEFINED);
9093            }

9095            switch (ckey) {
9096            case SCSI_CAP_DMA_MAX:
9097            case SCSI_CAP_MSG_OUT:
9098            case SCSI_CAP_PARITY:
9099            case SCSI_CAP_INITIATOR_ID:
9100            case SCSI_CAP_LINKED_CMDS:
9101            case SCSI_CAP_UNTAGGED_QING:
9102            case SCSI_CAP_RESET_NOTIFICATION:
9103                    /*
9104                     * None of these are settable via
9105                     * the capability interface.
9106                     */
9107                    break;
9108            case SCSI_CAP_ARQ:
9109                    /*
9110                     * We cannot turn off arq so return false if asked to
9111                     */
9112                    if (value) {
9113                            rval = TRUE;
9114                    } else {
9115                            rval = FALSE;
9116                    }
9117                    break;
9118            case SCSI_CAP_TAGGED_QING:
9119                    mptsas_set_throttle(mpt, ((mptsas_tgt_private_t *)
9120                        (ap->a_hba_tran->tran_tgt_private))->t_private,
9121                        MAX_THROTTLE);
9647                    ptgt = ((mptsas_tgt_private_t *)
9648                        (ap->a_hba_tran->tran_tgt_private))->t_private;
9649                    mutex_enter(&ptgt->m_tgt_intr_mutex);
9650                    mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9651                    mutex_exit(&ptgt->m_tgt_intr_mutex);
9122                    rval = TRUE;
9123                    break;
9124            case SCSI_CAP_QFULL_RETRIES:
9125                    ((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
9126                        t_private->m_qfull_retries = (uchar_t)value;
9127                    rval = TRUE;
9128                    break;
9129            case SCSI_CAP_QFULL_RETRY_INTERVAL:
9130                    ((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
9131                        t_private->m_qfull_retry_interval =
9132                        drv_usectohz(value * 1000);
9133                    rval = TRUE;
9134                    break;
9135            default:
9136                    rval = UNDEFINED;
9137                    break;
9138            }
9139            mutex_exit(&mpt->m_mutex);
9140            return (rval);
9141 }
```

*_____unchanged_portion_omitted_*

```
9159 static int
9160 mptsas_alloc_active_slots(mptsas_t *mpt, int flag)
```



```
9161 {
9162            mptsas_slots_t  *old_active = mpt->m_active;
9163            mptsas_slots_t  *new_active;
9164            size_t          size;
9165            int             rval = -1, i;
9695            int             rval = -1, nslot, i;
9696            mptsas_slot_free_e_t    *pe;

9167            /*
9168             * if there are active commands, then we cannot
9169             * change size of active slots array.
9170             */
9171            ASSERT(mpt->m_ncmds == 0);
9698            if (mptsas_outstanding_cmds_n(mpt)) {
9699                    NDBG9(("cannot change size of active slots array"));
9700                    return (rval);
9701            }

9173            size = MPTSAS_SLOTS_SIZE(mpt);
9174            new_active = kmem_zalloc(size, flag);
9175            if (new_active == NULL) {
9176                    NDBG1(("new active alloc failed"));
9177                    return (rval);
9178            }
9179            /*
9180             * Since SMID 0 is reserved and the TM slot is reserved, the
9181             * number of slots that can be used at any one time is
9182             * m_max_requests - 2.
9183             */
9184            new_active->m_n_slots = (mpt->m_max_requests - 2);
9714            new_active->m_n_slots = nslot = (mpt->m_max_requests - 2);
9185            new_active->m_size = size;
9186            new_active->m_tags = 1;

9187            if (old_active) {
9188                    new_active->m_tgttbl = old_active->m_tgttbl;
9189                    new_active->m_smptbl = old_active->m_smptbl;
9190                    new_active->m_num_raid_configs =
9191                        old_active->m_num_raid_configs;
9192                    for (i = 0; i < new_active->m_num_raid_configs; i++) {
9193                            new_active->m_raidconfig[i] =
9194                                old_active->m_raidconfig[i];
9195                    }
9196                    mptsas_free_active_slots(mpt);
9197            }

9730            if (max_ncpus & (max_ncpus - 1)) {
9731                    mpt->m_slot_freeq_pair_n = (1 << highbit(max_ncpus));
9732            } else {
9733                    mpt->m_slot_freeq_pair_n = max_ncpus;
9734            }
9735            mpt->m_slot_freeq_pairp = kmem_zalloc(
9736                mpt->m_slot_freeq_pair_n *
9737                sizeof (mptsas_slot_freeq_pair_t), KM_SLEEP);
9738            for (i = 0; i < mpt->m_slot_freeq_pair_n; i++) {
9739                    list_create(&mpt->m_slot_freeq_pairp[i].
9740                        m_slot_allocq.s.m_fq_list,
9741                        sizeof (mptsas_slot_free_e_t),
9742                        offsetof(mptsas_slot_free_e_t, node));
9743                    list_create(&mpt->m_slot_freeq_pairp[i].
9744                        m_slot_releq.s.m_fq_list,
9745                        sizeof (mptsas_slot_free_e_t),
9746                        offsetof(mptsas_slot_free_e_t, node));
9747                    mpt->m_slot_freeq_pairp[i].m_slot_allocq.s.m_fq_n = 0;
9748                    mpt->m_slot_freeq_pairp[i].m_slot_releq.s.m_fq_n = 0;
9749                    mutex_init(&mpt->m_slot_freeq_pairp[i].
```

```
                    m_slot_allocq.s.m_fq_mutex, NULL, MUTEX_DRIVER,
                    DDI_INTR_PRI(mpt->m_intr_pri));
                mutex_init(&mpt->m_slot_freeq_pairp[i].
                    m_slot_releq.s.m_fq_mutex, NULL, MUTEX_DRIVER,
                    DDI_INTR_PRI(mpt->m_intr_pri));
        }
        pe = mpt->m_slot_free_ae = kmem_zalloc(nslot *
            sizeof (mptsas_slot_free_e_t), KM_SLEEP);
        /*
         * An array of Mpi2ReplyDescriptorsUnion_t is defined here.
         * We are trying to eliminate the m_mutex in the context
         * reply code path in the ISR. Since the read of the
         * ReplyDescriptor and update/write of the ReplyIndex must
         * be atomic (since the poll thread may also update them at
         * the same time) so we first read out of the ReplyDescriptor
         * into this array and update the ReplyIndex register with a
         * separate mutex m_intr_mutex protected, and then release the
         * mutex and process all of them. the length of the array is
         * defined as max as 128(128*64=8k), which is
         * assumed as the maxmium depth of the interrupt coalese.
         */
        mpt->m_reply = kmem_zalloc(MPI_ADDRESS_COALSCE_MAX *
            sizeof (Mpi2ReplyDescriptorsUnion_t), KM_SLEEP);
        for (i = 0; i < nslot; i++, pe++) {
                pe->slot = i + 1; /* SMID 0 is reserved */
                pe->cpuid = i % mpt->m_slot_freeq_pair_n;
                list_insert_tail(&mpt->m_slot_freeq_pairp
                    [i % mpt->m_slot_freeq_pair_n]
                    .m_slot_allocq.s.m_fq_list, pe);
                mpt->m_slot_freeq_pairp[i % mpt->m_slot_freeq_pair_n]
                    .m_slot_allocq.s.m_fq_n++;
                mpt->m_slot_freeq_pairp[i % mpt->m_slot_freeq_pair_n]
                    .m_slot_allocq.s.m_fq_n_init++;
        }

        mpt->m_active = new_active;
        rval = 0;

        return (rval);
}

static void
mptsas_free_active_slots(mptsas_t *mpt)
{
        mptsas_slots_t  *active = mpt->m_active;
        size_t          size;
        mptsas_slot_free_e_t    *pe;
        int     i;

        if (active == NULL)
                return;

        if (mpt->m_slot_freeq_pairp) {
                for (i = 0; i < mpt->m_slot_freeq_pair_n; i++) {
                        while ((pe = list_head(&mpt->m_slot_freeq_pairp
                            [i].m_slot_allocq.s.m_fq_list)) != NULL) {
                                list_remove(&mpt->m_slot_freeq_pairp[i]
                                    .m_slot_allocq.s.m_fq_list, pe);
                        }
                        list_destroy(&mpt->m_slot_freeq_pairp
                            [i].m_slot_allocq.s.m_fq_list);
                        while ((pe = list_head(&mpt->m_slot_freeq_pairp
                            [i].m_slot_releq.s.m_fq_list)) != NULL) {
                                list_remove(&mpt->m_slot_freeq_pairp[i]
                                    .m_slot_releq.s.m_fq_list, pe);
                        }
```



```
                        list_destroy(&mpt->m_slot_freeq_pairp
                            [i].m_slot_releq.s.m_fq_list);
                        mutex_destroy(&mpt->m_slot_freeq_pairp
                            [i].m_slot_allocq.s.m_fq_mutex);
                        mutex_destroy(&mpt->m_slot_freeq_pairp
                            [i].m_slot_releq.s.m_fq_mutex);
                }
                kmem_free(mpt->m_slot_freeq_pairp, mpt->m_slot_freeq_pair_n *
                    sizeof (mptsas_slot_freeq_pair_t));
        }
        if (mpt->m_slot_free_ae)
                kmem_free(mpt->m_slot_free_ae, mpt->m_active->m_n_slots *
                    sizeof (mptsas_slot_free_e_t));

        if (mpt->m_reply)
                kmem_free(mpt->m_reply, MPI_ADDRESS_COALSCE_MAX *
                    sizeof (Mpi2ReplyDescriptorsUnion_t));

        size = active->m_size;
        kmem_free(active, size);
        mpt->m_active = NULL;
}
______unchanged_portion_omitted_

static void
mptsas_watchsubr(mptsas_t *mpt)
{
        int             i;
        mptsas_cmd_t    *cmd;
        mptsas_target_t *ptgt = NULL;

        NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

#ifdef MPTSAS_TEST
        if (mptsas_enable_untagged) {
                mptsas_test_untagged++;
        }
#endif

        /*
         * Check for commands stuck in active slot
         * Account for TM requests, which use the last SMID.
         */
        mutex_enter(&mpt->m_intr_mutex);
        for (i = 0; i <= mpt->m_active->m_n_slots; i++) {
                if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
                        if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
                                cmd->cmd_active_timeout -=
                                    mptsas_scsi_watchdog_tick;
                                if (cmd->cmd_active_timeout <= 0) {
                                        /*
                                         * There seems to be a command stuck
                                         * in the active slot.  Drain throttle.
                                         */
                                        mptsas_set_throttle(mpt,
                                            cmd->cmd_tgt_addr,
                                        ptgt = cmd->cmd_tgt_addr;
                                        mutex_enter(&ptgt->m_tgt_intr_mutex);
                                        mptsas_set_throttle(mpt, ptgt,
                                            DRAIN_THROTTLE);
                                        mutex_exit(&ptgt->m_tgt_intr_mutex);
                                }
                        }
                        if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
                            (cmd->cmd_flags & CFLAG_CONFIG) ||
                            (cmd->cmd_flags & CFLAG_FW_DIAG)) {
```

```
9376                                cmd->cmd_active_timeout -=
9377                                    mptsas_scsi_watchdog_tick;
9378                                if (cmd->cmd_active_timeout <= 0) {
9379                                        /*
9380                                         * passthrough command timeout
9381                                         */
9382                                        cmd->cmd_flags |= (CFLAG_FINISHED |
9383                                            CFLAG_TIMEOUT);
9384                                        cv_broadcast(&mpt->m_passthru_cv);
9385                                        cv_broadcast(&mpt->m_config_cv);
9386                                        cv_broadcast(&mpt->m_fw_diag_cv);
9387                                }
9388                        }
9389                }
9390        }
10016        mutex_exit(&mpt->m_intr_mutex);

9392        ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9393            MPTSAS_HASH_FIRST);
9394        while (ptgt != NULL) {
9395                /*
10022                 * In order to avoid using m_mutex in the key code path in ISR,
10023                 * separate mutexs are introduced to protect those elements
10024                 * shown in ISR.
10025                 */
10026                mutex_enter(&ptgt->m_tgt_intr_mutex);

10028                /*
9396                 * If we were draining due to a qfull condition,
9397                 * go back to full throttle.
9398                 */
9399                if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9400                    (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9401                    (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9402                        mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9403                        mptsas_restart_hba(mpt);
9404                }

9406                if ((ptgt->m_t_ncmds > 0) &&
9407                    (ptgt->m_timebase)) {

9409                        if (ptgt->m_timebase <=
9410                            mptsas_scsi_watchdog_tick) {
9411                                ptgt->m_timebase +=
9412                                    mptsas_scsi_watchdog_tick;
10046                                mutex_exit(&ptgt->m_tgt_intr_mutex);
9413                                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9414                                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9415                                continue;
9416                        }

9418                        ptgt->m_timeout -= mptsas_scsi_watchdog_tick;

9420                        if (ptgt->m_timeout_count > 0) {
9421                                ptgt->m_timeout_interval +=
9422                                    mptsas_scsi_watchdog_tick;
9423                        }
9424                        if (ptgt->m_timeout_interval >
9425                            mptsas_timeout_interval) {
9426                                ptgt->m_timeout_interval = 0;
9427                                ptgt->m_timeout_count = 0;
9428                        }

9430                        if (ptgt->m_timeout < 0) {
9431                                ptgt->m_timeout_count++;
9432                                if (ptgt->m_timeout_count >
```



```
9433                                    mptsas_timeout_threshold) {
9434                                        ptgt->m_timeout_count = 0;
9435                                        mptsas_kill_target(mpt, ptgt);
9436                                } else {
10055                                    mutex_exit(&ptgt->m_tgt_intr_mutex);
9437                                        mptsas_cmd_timeout(mpt, ptgt->m_devhdl);
9438                                }
9439                                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9440                                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9441                                continue;
9442                        }

9444                        if ((ptgt->m_timeout) <=
9445                            mptsas_scsi_watchdog_tick) {
9446                                NDBG23(("pending timeout"));
9447                                mptsas_set_throttle(mpt, ptgt,
9448                                    DRAIN_THROTTLE);
9449                        }
9450                }

10069                mutex_exit(&ptgt->m_tgt_intr_mutex);
9452                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9453                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9454        }
9455 }
______unchanged_portion_omitted_

9479 /*
9480  * target causing too many timeouts
9481  */
9482 static void
9483 mptsas_kill_target(mptsas_t *mpt, mptsas_target_t *ptgt)
9484 {
9485        mptsas_topo_change_list_t       *topo_node = NULL;

9487        NDBG29(("mptsas_tgt_kill: target=%d", ptgt->m_devhdl));
9488        mptsas_log(mpt, CE_WARN, "timeout threshold exceeded for "
9489            "Target %d", ptgt->m_devhdl);

9491        topo_node = kmem_zalloc(sizeof (mptsas_topo_change_list_t), KM_SLEEP);
9492        topo_node->mpt = mpt;
9493        topo_node->un.phymask = ptgt->m_phymask;
9494        topo_node->event = MPTSAS_DR_EVENT_OFFLINE_TARGET;
9495        topo_node->devhdl = ptgt->m_devhdl;
9496        if (ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
9497                topo_node->flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
9498        else
9499                topo_node->flags = MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
9500        topo_node->object = NULL;

9502        /*
9503         * Launch DR taskq to fake topology change
9504         */
9505        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
9506            mptsas_handle_dr, (void *)topo_node,
9507            DDI_NOSLEEP)) != DDI_SUCCESS) {
9508                mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
9509                    "for fake offline event failed. \n");
9510        }
9511 }

9513 /*
9514  * Device / Hotplug control
9515  */
9516 static int
9517 mptsas_scsi_quiesce(dev_info_t *dip)
```

```
9518 {
9519         mptsas_t        *mpt;
9520         scsi_hba_tran_t *tran;

9522         tran = ddi_get_driver_private(dip);
9523         if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
9524                 return (-1);

9526         return (mptsas_quiesce_bus(mpt));
9527 }
_____unchanged_portion_omitted_

9542 static int
9543 mptsas_quiesce_bus(mptsas_t *mpt)
9544 {
9545         mptsas_target_t *ptgt = NULL;

9547         NDBG28(("mptsas_quiesce_bus"));
9548         mutex_enter(&mpt->m_mutex);

9550         /* Set all the throttles to zero */
9551         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9552             MPTSAS_HASH_FIRST);
9553         while (ptgt != NULL) {
10138                 mutex_enter(&ptgt->m_tgt_intr_mutex);
9554                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10140                 mutex_exit(&ptgt->m_tgt_intr_mutex);

9556                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9557                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9558         }

9560         /* If there are any outstanding commands in the queue */
9561         if (mpt->m_ncmds) {
10147          mutex_enter(&mpt->m_intr_mutex);
10148          if (mptsas_outstanding_cmds_n(mpt)) {
10149                 mutex_exit(&mpt->m_intr_mutex);
9562                 mpt->m_softstate |= MPTSAS_SS_DRAINING;
9563                 mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
9564                     mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
9565                 if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
9566                         /*
9567                          * Quiesce has been interrupted
9568                          */
9569                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
9570                         ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9571                             &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
9572                         while (ptgt != NULL) {
10161                                 mutex_enter(&ptgt->m_tgt_intr_mutex);
9573                                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10163                                 mutex_exit(&ptgt->m_tgt_intr_mutex);

9575                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9576                                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9577                         }
9578                         mptsas_restart_hba(mpt);
9579                         if (mpt->m_quiesce_timeid != 0) {
9580                                 timeout_id_t tid = mpt->m_quiesce_timeid;
9581                                 mpt->m_quiesce_timeid = 0;
9582                                 mutex_exit(&mpt->m_mutex);
9583                                 (void) untimeout(tid);
9584                                 return (-1);
9585                         }
9586                         mutex_exit(&mpt->m_mutex);
9587                         return (-1);
9588                 } else {
```



```
9589                         /* Bus has been quiesced */
9590                         ASSERT(mpt->m_quiesce_timeid == 0);
9591                         mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
9592                         mpt->m_softstate |= MPTSAS_SS_QUIESCED;
9593                         mutex_exit(&mpt->m_mutex);
9594                         return (0);
9595                 }
9596         }
10187         mutex_exit(&mpt->m_intr_mutex);
9597         /* Bus was not busy - QUIESCED */
9598         mutex_exit(&mpt->m_mutex);

9600         return (0);
9601 }

9603 static int
9604 mptsas_unquiesce_bus(mptsas_t *mpt)
9605 {
9606         mptsas_target_t *ptgt = NULL;

9608         NDBG28(("mptsas_unquiesce_bus"));
9609         mutex_enter(&mpt->m_mutex);
9610         mpt->m_softstate &= ~MPTSAS_SS_QUIESCED;
9611         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9612             MPTSAS_HASH_FIRST);
9613         while (ptgt != NULL) {
10205                 mutex_enter(&ptgt->m_tgt_intr_mutex);
9614                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10207                 mutex_exit(&ptgt->m_tgt_intr_mutex);

9616                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9617                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9618         }
9619         mptsas_restart_hba(mpt);
9620         mutex_exit(&mpt->m_mutex);
9621         return (0);
9622 }

9624 static void
9625 mptsas_ncmds_checkdrain(void *arg)
9626 {
9627         mptsas_t        *mpt = arg;
9628         mptsas_target_t *ptgt = NULL;

9630         mutex_enter(&mpt->m_mutex);
9631         if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
9632                 mpt->m_quiesce_timeid = 0;
9633                 if (mpt->m_ncmds == 0) {
9634                         /* Command queue has been drained */
9635                         cv_signal(&mpt->m_cv);
9636                 } else {
10226                  mutex_enter(&mpt->m_intr_mutex);
10227                  if (mptsas_outstanding_cmds_n(mpt)) {
10228                          mutex_exit(&mpt->m_intr_mutex);
9637                          /*
9638                           * The throttle may have been reset because
9639                           * of a SCSI bus reset
9640                           */
9641                         ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9642                             &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
9643                         while (ptgt != NULL) {
10236                                 mutex_enter(&ptgt->m_tgt_intr_mutex);
9644                                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10238                                 mutex_exit(&ptgt->m_tgt_intr_mutex);

9646                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
```

```
9647                                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9648                            }

9650                            mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
9651                                mpt, (MPTSAS_QUIESCE_TIMEOUT *
9652                                drv_usectohz(1000000)));
10247                    } else {
10248                            mutex_exit(&mpt->m_intr_mutex);
10249                            /* Command queue has been drained */
10250                            cv_signal(&mpt->m_cv);
9653                    }
9654            }
9655            mutex_exit(&mpt->m_mutex);
9656 }
```

*unchanged_portion_omitted_*

```
9682 static void
9683 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
9684 {
9685            caddr_t                 memp;
9686            pMPI2RequestHeader_t    request_hdrp;
9687            struct scsi_pkt         *pkt = cmd->cmd_pkt;
9688            mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
9689            uint32_t                request_size, data_size, dataout_size;
9690            uint32_t                direction;
9691            ddi_dma_cookie_t        data_cookie;
9692            ddi_dma_cookie_t        dataout_cookie;
9693            uint32_t                request_desc_low, request_desc_high = 0;
9694            uint32_t                i, sense_bufp;
9695            uint8_t                 desc_type;
9696            uint8_t                 *request, function;
9697            ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
9698            ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

9700            desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

9702            request = pt->request;
9703            direction = pt->direction;
9704            request_size = pt->request_size;
9705            data_size = pt->data_size;
9706            dataout_size = pt->dataout_size;
9707            data_cookie = pt->data_cookie;
9708            dataout_cookie = pt->dataout_cookie;

9710            /*
9711             * Store the passthrough message in memory location
9712             * corresponding to our slot number
9713             */
9714            memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
9715            request_hdrp = (pMPI2RequestHeader_t)memp;
9716            bzero(memp, mpt->m_req_frame_size);

9718            for (i = 0; i < request_size; i++) {
9719                    bcopy(request + i, memp + i, 1);
9720            }

9722            if (data_size || dataout_size) {
9723                    pMpi2SGESimple64_t      sgep;
9724                    uint32_t                sge_flags;

9726                    sgep = (pMpi2SGESimple64_t)((uint8_t *)request_hdrp +
9727                        request_size);
9728                    if (dataout_size) {

9730                            sge_flags = dataout_size |
9731                                ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
```



```
9732                                    MPI2_SGE_FLAGS_END_OF_BUFFER |
9733                                    MPI2_SGE_FLAGS_HOST_TO_IOC |
9734                                    MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
9735                                    MPI2_SGE_FLAGS_SHIFT);
9736                            ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
9737                            ddi_put32(acc_hdl, &sgep->Address.Low,
9738                                (uint32_t)(dataout_cookie.dmac_laddress &
9739                                0xffffffffull));
9740                            ddi_put32(acc_hdl, &sgep->Address.High,
9741                                (uint32_t)(dataout_cookie.dmac_laddress
9742                                >> 32));
9743                            sgep++;
9744                    }
9745                    sge_flags = data_size;
9746                    sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
9747                        MPI2_SGE_FLAGS_LAST_ELEMENT |
9748                        MPI2_SGE_FLAGS_END_OF_BUFFER |
9749                        MPI2_SGE_FLAGS_END_OF_LIST |
9750                        MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
9751                        MPI2_SGE_FLAGS_SHIFT);
9752                    if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
9753                            sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
9754                                MPI2_SGE_FLAGS_SHIFT);
9755                    } else {
9756                            sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
9757                                MPI2_SGE_FLAGS_SHIFT);
9758                    }
9759                    ddi_put32(acc_hdl, &sgep->FlagsLength,
9760                        sge_flags);
9761                    ddi_put32(acc_hdl, &sgep->Address.Low,
9762                        (uint32_t)(data_cookie.dmac_laddress &
9763                        0xffffffffull));
9764                    ddi_put32(acc_hdl, &sgep->Address.High,
9765                        (uint32_t)(data_cookie.dmac_laddress >> 32));
9766            }

9768            function = request_hdrp->Function;
9769            if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
9770                (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
9771                    pMpi2SCSIIORequest_t    scsi_io_req;

9773                    scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;
9774                    /*
9775                     * Put SGE for data and data_out buffer at the end of
9776                     * scsi_io_request message header.(64 bytes in total)
9777                     * Following above SGEs, the residual space will be
9778                     * used by sense data.
9779                     */
9780                    ddi_put8(acc_hdl,
9781                        &scsi_io_req->SenseBufferLength,
9782                        (uint8_t)(request_size - 64));

9784                    sense_bufp = mpt->m_req_frame_dma_addr +
9785                        (mpt->m_req_frame_size * cmd->cmd_slot);
9786                    sense_bufp += 64;
9787                    ddi_put32(acc_hdl,
9788                        &scsi_io_req->SenseBufferLowAddress, sense_bufp);

9790                    /*
9791                     * Set SGLOffset0 value
9792                     */
9793                    ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
9794                        offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);

9796                    /*
9797                     * Setup descriptor info.  RAID passthrough must use the
```

```
9798                 * default request descriptor which is already set, so if this
9799                 * is a SCSI IO request, change the descriptor to SCSI IO.
9800                 */
9801                if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
9802                        desc_type = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
9803                        request_desc_high = (ddi_get16(acc_hdl,
9804                            &scsi_io_req->DevHandle) << 16);
9805                }
9806        }

9808        /*
9809         * We must wait till the message has been completed before
9810         * beginning the next message so we wait for this one to
9811         * finish.
9812         */
9813        (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
9814        request_desc_low = (cmd->cmd_slot << 16) + desc_type;
9815        cmd->cmd_rfm = NULL;
10414        mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
9816        MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
9817        if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
9818            (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
9819                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
9820        }
9821 }
```

*unchanged_portion_omitted_*

```
10183 static void
10184 mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
10185 {
10186        pMpi2DiagBufferPostRequest_t    pDiag_post_msg;
10187        pMpi2DiagReleaseRequest_t       pDiag_release_msg;
10188        struct scsi_pkt                 *pkt = cmd->cmd_pkt;
10189        mptsas_diag_request_t           *diag = pkt->pkt_ha_private;
10190        uint32_t                        request_desc_low, i;

10192        ASSERT(mutex_owned(&mpt->m_mutex));

10194        /*
10195         * Form the diag message depending on the post or release function.
10196         */
10197        if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
10198                pDiag_post_msg = (pMpi2DiagBufferPostRequest_t)
10199                    (mpt->m_req_frame + (mpt->m_req_frame_size *
10200                    cmd->cmd_slot));
10201                bzero(pDiag_post_msg, mpt->m_req_frame_size);
10202                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
10203                    diag->function);
10204                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
10205                    diag->pBuffer->buffer_type);
10206                ddi_put8(mpt->m_acc_req_frame_hdl,
10207                    &pDiag_post_msg->ExtendedType,
10208                    diag->pBuffer->extended_type);
10209                ddi_put32(mpt->m_acc_req_frame_hdl,
10210                    &pDiag_post_msg->BufferLength,
10211                    diag->pBuffer->buffer_data.size);
10212                for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);
10213                    i++) {
10214                        ddi_put32(mpt->m_acc_req_frame_hdl,
10215                            &pDiag_post_msg->ProductSpecific[i],
10216                            diag->pBuffer->product_specific[i]);
10217                }
10218                ddi_put32(mpt->m_acc_req_frame_hdl,
10219                    &pDiag_post_msg->BufferAddress.Low,
10220                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
10221                    & 0xffffffffull));
```



```
10222                ddi_put32(mpt->m_acc_req_frame_hdl,
10223                    &pDiag_post_msg->BufferAddress.High,
10224                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
10225                    >> 32));
10226        } else {
10227                pDiag_release_msg = (pMpi2DiagReleaseRequest_t)
10228                    (mpt->m_req_frame + (mpt->m_req_frame_size *
10229                    cmd->cmd_slot));
10230                bzero(pDiag_release_msg, mpt->m_req_frame_size);
10231                ddi_put8(mpt->m_acc_req_frame_hdl,
10232                    &pDiag_release_msg->Function, diag->function);
10233                ddi_put8(mpt->m_acc_req_frame_hdl,
10234                    &pDiag_release_msg->BufferType,
10235                    diag->pBuffer->buffer_type);
10236        }

10238        /*
10239         * Send the message
10240         */
10241        (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
10242            DDI_DMA_SYNC_FORDEV);
10243        request_desc_low = (cmd->cmd_slot << 16) +
10244            MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
10245        cmd->cmd_rfm = NULL;
10845        mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
10246        MPTSAS_START_CMD(mpt, request_desc_low, 0);
10247        if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
10248            DDI_SUCCESS) ||
10249            (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
10250            DDI_SUCCESS)) {
10251                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10252        }
10253 }
```

*unchanged_portion_omitted_*

```
11353 static int
11354 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
11355     int *rval)
11356 {
11357        int                     status = 0;
11358        mptsas_t                *mpt;
11359        mptsas_update_flash_t   flashdata;
11360        mptsas_pass_thru_t      passthru_data;
11361        mptsas_adapter_data_t   adapter_data;
11362        mptsas_pci_info_t       pci_info;
11363        int                     copylen;

11365        int                     iport_flag = 0;
11366        dev_info_t              *dip = NULL;
11367        mptsas_phymask_t        phymask = 0;
11968        struct devctl_iocdata   *dcp = NULL;
11969        uint32_t                slotstatus = 0;
11970        char                    *addr = NULL;
11971        mptsas_target_t         *ptgt = NULL;

11369        *rval = MPTIOCTL_STATUS_GOOD;
11370        if (secpolicy_sys_config(credp, B_FALSE) != 0) {
11371                return (EPERM);
11372        }

11374        mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
11375        if (mpt == NULL) {
11376                /*
11377                 * Called from iport node, get the states
11378                 */
11379                iport_flag = 1;
```

```
11380                 dip = mptsas_get_dip_from_dev(dev, &phymask);
11381                 if (dip == NULL) {
11382                         return (ENXIO);
11383                 }
11384                 mpt = DIP2MPT(dip);
11385         }
11386         /* Make sure power level is D0 before accessing registers */
11387         mutex_enter(&mpt->m_mutex);
11388         if (mpt->m_options & MPTSAS_OPT_PM) {
11389                 (void) pm_busy_component(mpt->m_dip, 0);
11390                 if (mpt->m_power_level != PM_LEVEL_D0) {
11391                         mutex_exit(&mpt->m_mutex);
11392                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
11393                             DDI_SUCCESS) {
11394                                 mptsas_log(mpt, CE_WARN,
11395                                     "mptsas%d: mptsas_ioctl: Raise power "
11396                                     "request failed.", mpt->m_instance);
11397                                 (void) pm_idle_component(mpt->m_dip, 0);
11398                                 return (ENXIO);
11399                         }
11400                 } else {
11401                         mutex_exit(&mpt->m_mutex);
11402                 }
11403         } else {
11404                 mutex_exit(&mpt->m_mutex);
11405         }

11407         if (iport_flag) {
11408                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
12013                 if (status != 0) {
11409                         goto out;
11410         }
12016                 /*
12017                  * The following code control the OK2RM LED, it doesn't affect
12018                  * the ioctl return status.
12019                  */
12020                 if ((cmd == DEVCTL_DEVICE_ONLINE) ||
12021                     (cmd == DEVCTL_DEVICE_OFFLINE)) {
12022                         if (ndi_dc_allochdl((void *)data, &dcp) !=
12023                             NDI_SUCCESS) {
12024                                 goto out;
12025                         }
12026                         addr = ndi_dc_getaddr(dcp);
12027                         ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
12028                         if (ptgt == NULL) {
12029                                 NDBG14(("mptsas_ioctl led control: tgt %s not "
12030                                     "found", addr));
12031                                 ndi_dc_freehdl(dcp);
12032                                 goto out;
12033                         }
12034                         mutex_enter(&mpt->m_mutex);
12035                         if (cmd == DEVCTL_DEVICE_ONLINE) {
12036                                 ptgt->m_tgt_unconfigured = 0;
12037                         } else if (cmd == DEVCTL_DEVICE_OFFLINE) {
12038                                 ptgt->m_tgt_unconfigured = 1;
12039                         }
12040                         slotstatus = 0;
12041 #ifdef MPTSAS_GET_LED
12042                         /*
12043                          * The get led status can't get a valid/reasonable
12044                          * state, so ignore the get led status, and write the
12045                          * required value directly
12046                          */
12047                         if (mptsas_get_led_status(mpt, ptgt, &slotstatus) !=
12048                             DDI_SUCCESS) {
12049                                 NDBG14(("mptsas_ioctl: get LED for tgt %s "
```



```
12050                                     "failed %x", addr, slotstatus));
12051                                 slotstatus = 0;
12052                         }
12053                         NDBG14(("mptsas_ioctl: LED status %x for %s",
12054                             slotstatus, addr));
12055 #endif
12056                         if (cmd == DEVCTL_DEVICE_OFFLINE) {
12057                                 slotstatus |=
12058                                     MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;
12059                         } else {
12060                                 slotstatus &=
12061                                     ~MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;
12062                         }
12063                         if (mptsas_set_led_status(mpt, ptgt, slotstatus) !=
12064                             DDI_SUCCESS) {
12065                                 NDBG14(("mptsas_ioctl: set LED for tgt %s "
12066                                     "failed %x", addr, slotstatus));
12067                         }
12068                         mutex_exit(&mpt->m_mutex);
12069                         ndi_dc_freehdl(dcp);
12070                 }
12071                 goto out;
12072         }
11411         switch (cmd) {
11412                 case MPTIOCTL_UPDATE_FLASH:
11413                         if (ddi_copyin((void *)data, &flashdata,
11414                                 sizeof (struct mptsas_update_flash), mode)) {
11415                                 status = EFAULT;
11416                                 break;
11417                         }

11419                         mutex_enter(&mpt->m_mutex);
11420                         if (mptsas_update_flash(mpt,
11421                             (caddr_t)(long)flashdata.PtrBuffer,
11422                             flashdata.ImageSize, flashdata.ImageType, mode)) {
11423                                 status = EFAULT;
11424                         }

11426                         /*
11427                          * Reset the chip to start using the new
11428                          * firmware.  Reset if failed also.
11429                          */
11430                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11431                         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
11432                                 status = EFAULT;
11433                         }
11434                         mutex_exit(&mpt->m_mutex);
11435                         break;
11436                 case MPTIOCTL_PASS_THRU:
11437                         /*
11438                          * The user has requested to pass through a command to
11439                          * be executed by the MPT firmware.  Call our routine
11440                          * which does this.  Only allow one passthru IOCTL at
11441                          * one time. Other threads will block on
11442                          * m_passthru_mutex, which is of adaptive variant.
11443                          */
11444                         if (ddi_copyin((void *)data, &passthru_data,
11445                             sizeof (mptsas_pass_thru_t), mode)) {
11446                                 status = EFAULT;
11447                                 break;
11448                         }
11449                         mutex_enter(&mpt->m_passthru_mutex);
11450                         mutex_enter(&mpt->m_mutex);
11451                         status = mptsas_pass_thru(mpt, &passthru_data, mode);
11452                         mutex_exit(&mpt->m_mutex);
11453                         mutex_exit(&mpt->m_passthru_mutex);
```

```
11455                   break;
11456           case MPTIOCTL_GET_ADAPTER_DATA:
11457                   /*
11458                    * The user has requested to read adapter data.  Call
11459                    * our routine which does this.
11460                    */
11461                   bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
11462                   if (ddi_copyin((void *)data, (void *)&adapter_data,
11463                       sizeof (mptsas_adapter_data_t), mode)) {
11464                       status = EFAULT;
11465                       break;
11466                   }
11467                   if (adapter_data.StructureLength >=
11468                       sizeof (mptsas_adapter_data_t)) {
11469                       adapter_data.StructureLength = (uint32_t)
11470                           sizeof (mptsas_adapter_data_t);
11471                       copylen = sizeof (mptsas_adapter_data_t);
11472                       mutex_enter(&mpt->m_mutex);
11473                       mptsas_read_adapter_data(mpt, &adapter_data);
11474                       mutex_exit(&mpt->m_mutex);
11475                   } else {
11476                       adapter_data.StructureLength = (uint32_t)
11477                           sizeof (mptsas_adapter_data_t);
11478                       copylen = sizeof (adapter_data.StructureLength);
11479                       *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
11480                   }
11481                   if (ddi_copyout((void *)(&adapter_data), (void *)data,
11482                       copylen, mode) != 0) {
11483                       status = EFAULT;
11484                   }
11485                   break;
11486           case MPTIOCTL_GET_PCI_INFO:
11487                   /*
11488                    * The user has requested to read pci info.  Call
11489                    * our routine which does this.
11490                    */
11491                   bzero(&pci_info, sizeof (mptsas_pci_info_t));
11492                   mutex_enter(&mpt->m_mutex);
11493                   mptsas_read_pci_info(mpt, &pci_info);
11494                   mutex_exit(&mpt->m_mutex);
11495                   if (ddi_copyout((void *)(&pci_info), (void *)data,
11496                       sizeof (mptsas_pci_info_t), mode) != 0) {
11497                       status = EFAULT;
11498                   }
11499                   break;
11500           case MPTIOCTL_RESET_ADAPTER:
11501                   mutex_enter(&mpt->m_mutex);
11502                   mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11503                   if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
11504                           mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
11505                               "failed");
11506                           status = EFAULT;
11507                   }
11508                   mutex_exit(&mpt->m_mutex);
11509                   break;
11510           case MPTIOCTL_DIAG_ACTION:
11511                   /*
11512                    * The user has done a diag buffer action.  Call our
11513                    * routine which does this.  Only allow one diag action
11514                    * at one time.
11515                    */
11516                   mutex_enter(&mpt->m_mutex);
11517                   if (mpt->m_diag_action_in_progress) {
11518                           mutex_exit(&mpt->m_mutex);
11519                           return (EBUSY);
```



```
11520                   }
11521                   mpt->m_diag_action_in_progress = 1;
11522                   status = mptsas_diag_action(mpt,
11523                       (mptsas_diag_action_t *)data, mode);
11524                   mpt->m_diag_action_in_progress = 0;
11525                   mutex_exit(&mpt->m_mutex);
11526                   break;
11527           case MPTIOCTL_EVENT_QUERY:
11528                   /*
11529                    * The user has done an event query. Call our routine
11530                    * which does this.
11531                    */
11532                   status = mptsas_event_query(mpt,
11533                       (mptsas_event_query_t *)data, mode, rval);
11534                   break;
11535           case MPTIOCTL_EVENT_ENABLE:
11536                   /*
11537                    * The user has done an event enable. Call our routine
11538                    * which does this.
11539                    */
11540                   status = mptsas_event_enable(mpt,
11541                       (mptsas_event_enable_t *)data, mode, rval);
11542                   break;
11543           case MPTIOCTL_EVENT_REPORT:
11544                   /*
11545                    * The user has done an event report. Call our routine
11546                    * which does this.
11547                    */
11548                   status = mptsas_event_report(mpt,
11549                       (mptsas_event_report_t *)data, mode, rval);
11550                   break;
11551           case MPTIOCTL_REG_ACCESS:
11552                   /*
11553                    * The user has requested register access.  Call our
11554                    * routine which does this.
11555                    */
11556                   status = mptsas_reg_access(mpt,
11557                       (mptsas_reg_access_t *)data, mode);
11558                   break;
11559           default:
11560                   status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
11561                       rval);
11562                   break;
11563           }

11565 out:
12228           if (mpt->m_options & MPTSAS_OPT_PM)
12229                   (void) pm_idle_component(mpt->m_dip, 0);
11566           return (status);
11567 }

11569 int
11570 mptsas_restart_ioc(mptsas_t *mpt)
11571 {
11572           int             rval = DDI_SUCCESS;
11573           mptsas_target_t *ptgt = NULL;

11575           ASSERT(mutex_owned(&mpt->m_mutex));

11577           /*
11578            * Set a flag telling I/O path that we're processing a reset.  This is
11579            * needed because after the reset is complete, the hash table still
11580            * needs to be rebuilt.  If I/Os are started before the hash table is
11581            * rebuilt, I/O errors will occur.  This flag allows I/Os to be marked
11582            * so that they can be retried.
11583            */
```

```
11584         mpt->m_in_reset = TRUE;

11586         /*
11587          * Set all throttles to HOLD
11588          */
11589         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
11590             MPTSAS_HASH_FIRST);
11591         while (ptgt != NULL) {
12256                 mutex_enter(&ptgt->m_tgt_intr_mutex);
11592                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
12258                 mutex_exit(&ptgt->m_tgt_intr_mutex);

11594                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
11595                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
11596         }

11598         /*
11599          * Disable interrupts
11600          */
11601         MPTSAS_DISABLE_INTR(mpt);

11603         /*
11604          * Abort all commands: outstanding commands, commands in waitq and
11605          * tx_waitq.
12270          * Abort all commands: outstanding commands, commands in waitq
11606          */
11607         mptsas_flush_hba(mpt);

11609         /*
11610          * Reinitialize the chip.
11611          */
11612         if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
11613                 rval = DDI_FAILURE;
11614         }

11616         /*
11617          * Enable interrupts again
11618          */
11619         MPTSAS_ENABLE_INTR(mpt);

11621         /*
11622          * If mptsas_init_chip was successful, update the driver data.
11623          */
11624         if (rval == DDI_SUCCESS) {
11625                 mptsas_update_driver_data(mpt);
11626         }

11628         /*
11629          * Reset the throttles
11630          */
11631         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
11632             MPTSAS_HASH_FIRST);
11633         while (ptgt != NULL) {
12299                 mutex_enter(&ptgt->m_tgt_intr_mutex);
11634                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
12301                 mutex_exit(&ptgt->m_tgt_intr_mutex);

11636                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
11637                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
11638         }

11640         mptsas_doneq_empty(mpt);
11641         mptsas_restart_hba(mpt);

11643         if (rval != DDI_SUCCESS) {
11644                 mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
```



```
11645                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
11646         }

11648         /*
11649          * Clear the reset flag so that I/Os can continue.
11650          */
11651         mpt->m_in_reset = FALSE;

11653         return (rval);
11654 }
______unchanged_portion_omitted_

11915 static int
11916 mptsas_init_pm(mptsas_t *mpt)
11917 {
11918         char            pmc_name[16];
11919         char            *pmc[] = {
11920                                 NULL,
11921                                 "0=Off (PCI D3 State)",
11922                                 "3=On (PCI D0 State)",
11923                                 NULL
11924                         };
11925         uint16_t        pmcsr_stat;

11927         if (mptsas_get_pci_cap(mpt) == FALSE) {
11928                 return (DDI_FAILURE);
11929         }
11930         /*
11931          * If PCI's capability does not support PM, then don't need
11932          * to registe the pm-components
11933          */
11934         if (!(mpt->m_options & MPTSAS_OPT_PM))
11935                 return (DDI_SUCCESS);
11936         /*
11937          * If power management is supported by this chip, create
11938          * pm-components property for the power management framework
11939          */
11940         (void) sprintf(pmc_name, "NAME=mptsas%d", mpt->m_instance);
11941         pmc[0] = pmc_name;
11942         if (ddi_prop_update_string_array(DDI_DEV_T_NONE, mpt->m_dip,
11943             "pm-components", pmc, 3) != DDI_PROP_SUCCESS) {
12611                 mutex_enter(&mpt->m_intr_mutex);
11944                 mpt->m_options &= ~MPTSAS_OPT_PM;
12613                 mutex_exit(&mpt->m_intr_mutex);
11945                 mptsas_log(mpt, CE_WARN,
11946                     "mptsas%d: pm-component property creation failed.",
11947                     mpt->m_instance);
11948                 return (DDI_FAILURE);
11949         }

11951         /*
11952          * Power on device.
11953          */
11954         (void) pm_busy_component(mpt->m_dip, 0);
11955         pmcsr_stat = pci_config_get16(mpt->m_config_handle,
11956             mpt->m_pmcsr_offset);
11957         if ((pmcsr_stat & PCI_PMCSR_STATE_MASK) != PCI_PMCSR_D0) {
11958                 mptsas_log(mpt, CE_WARN, "mptsas%d: Power up the device",
11959                     mpt->m_instance);
11960                 pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset,
11961                     PCI_PMCSR_D0);
11962         }
11963         if (pm_power_has_changed(mpt->m_dip, 0, PM_LEVEL_D0) != DDI_SUCCESS) {
11964                 mptsas_log(mpt, CE_WARN, "pm_power_has_changed failed");
11965                 return (DDI_FAILURE);
11966         }
```

```
12636         mutex_enter(&mpt->m_intr_mutex);
11967         mpt->m_power_level = PM_LEVEL_D0;
12638         mutex_exit(&mpt->m_intr_mutex);
11968         /*
11969          * Set pm idle delay.
11970          */
11971         mpt->m_pm_idle_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
11972             mpt->m_dip, 0, "mptsas-pm-idle-delay", MPTSAS_PM_IDLE_TIMEOUT);

11974         return (DDI_SUCCESS);
11975 }
______unchanged_portion_omitted_

12024 /*
12025  * mptsas_add_intrs:
12026  *
12027  * Register FIXED or MSI interrupts.
12028  */
12029 static int
12030 mptsas_add_intrs(mptsas_t *mpt, int intr_type)
12031 {
12032         dev_info_t      *dip = mpt->m_dip;
12033         int             avail, actual, count = 0;
12034         int             i, flag, ret;

12036         NDBG6(("mptsas_add_intrs:interrupt type 0x%x", intr_type));

12038         /* Get number of interrupts */
12039         ret = ddi_intr_get_nintrs(dip, intr_type, &count);
12040         if ((ret != DDI_SUCCESS) || (count <= 0)) {
12041                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_nintrs() failed, "
12042                     "ret %d count %d\n", ret, count);

12044                 return (DDI_FAILURE);
12045         }

12047         /* Get number of available interrupts */
12048         ret = ddi_intr_get_navail(dip, intr_type, &avail);
12049         if ((ret != DDI_SUCCESS) || (avail == 0)) {
12050                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_navail() failed, "
12051                     "ret %d avail %d\n", ret, avail);

12053                 return (DDI_FAILURE);
12054         }

12056         if (0 && avail < count) {
12727         if (avail < count) {
12057                 mptsas_log(mpt, CE_NOTE, "ddi_intr_get_nvail returned %d, "
12058                     "navail() returned %d", count, avail);
12059         }

12061         /* Mpt only have one interrupt routine */
12062         if ((intr_type == DDI_INTR_TYPE_MSI) && (count > 1)) {
12063                 count = 1;
12064         }

12066         /* Allocate an array of interrupt handles */
12067         mpt->m_intr_size = count * sizeof (ddi_intr_handle_t);
12068         mpt->m_htable = kmem_alloc(mpt->m_intr_size, KM_SLEEP);

12070         flag = DDI_INTR_ALLOC_NORMAL;

12072         /* call ddi_intr_alloc() */
12073         ret = ddi_intr_alloc(dip, mpt->m_htable, intr_type, 0,
12074             count, &actual, flag);
```



```
12076         if ((ret != DDI_SUCCESS) || (actual == 0)) {
12077                 mptsas_log(mpt, CE_WARN, "ddi_intr_alloc() failed, ret %d\n",
12078                     ret);
12079                 kmem_free(mpt->m_htable, mpt->m_intr_size);
12080                 return (DDI_FAILURE);
12081         }

12083         /* use interrupt count returned or abort? */
12084         if (actual < count) {
12085                 mptsas_log(mpt, CE_NOTE, "Requested: %d, Received: %d\n",
12086                     count, actual);
12087         }

12089         mpt->m_intr_cnt = actual;

12091         /*
12092          * Get priority for first msi, assume remaining are all the same
12093          */
12094         if ((ret = ddi_intr_get_pri(mpt->m_htable[0],
12095             &mpt->m_intr_pri)) != DDI_SUCCESS) {
12096                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_pri() failed %d\n", ret);

12098                 /* Free already allocated intr */
12099                 for (i = 0; i < actual; i++) {
12100                         (void) ddi_intr_free(mpt->m_htable[i]);
12101                 }

12103                 kmem_free(mpt->m_htable, mpt->m_intr_size);
12104                 return (DDI_FAILURE);
12105         }

12107         /* Test for high level mutex */
12108         if (mpt->m_intr_pri >= ddi_intr_get_hilevel_pri()) {
12109                 mptsas_log(mpt, CE_WARN, "mptsas_add_intrs: "
12110                     "Hi level interrupt not supported\n");

12112                 /* Free already allocated intr */
12113                 for (i = 0; i < actual; i++) {
12114                         (void) ddi_intr_free(mpt->m_htable[i]);
12115                 }

12117                 kmem_free(mpt->m_htable, mpt->m_intr_size);
12118                 return (DDI_FAILURE);
12119         }

12121         /* Call ddi_intr_add_handler() */
12122         for (i = 0; i < actual; i++) {
12123                 if ((ret = ddi_intr_add_handler(mpt->m_htable[i], mptsas_intr,
12124                     (caddr_t)mpt, (caddr_t)(uintptr_t)i)) != DDI_SUCCESS) {
12125                         mptsas_log(mpt, CE_WARN, "ddi_intr_add_handler() "
12126                             "failed %d\n", ret);

12128                         /* Free already allocated intr */
12129                         for (i = 0; i < actual; i++) {
12130                                 (void) ddi_intr_free(mpt->m_htable[i]);
12131                         }

12133                         kmem_free(mpt->m_htable, mpt->m_intr_size);
12134                         return (DDI_FAILURE);
12135                 }
12136         }

12138         if ((ret = ddi_intr_get_cap(mpt->m_htable[0], &mpt->m_intr_cap))
12139             != DDI_SUCCESS) {
12140                 mptsas_log(mpt, CE_WARN, "ddi_intr_get_cap() failed %d\n", ret);
```

```
12142                 /* Free already allocated intr */
12143                 for (i = 0; i < actual; i++) {
12144                         (void) ddi_intr_free(mpt->m_htable[i]);
12145                 }

12147                 kmem_free(mpt->m_htable, mpt->m_intr_size);
12148                 return (DDI_FAILURE);
12149         }

12151         /*
12152          * Enable interrupts
12153          */
12154         if (mpt->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
12155                 /* Call ddi_intr_block_enable() for MSI interrupts */
12156                 (void) ddi_intr_block_enable(mpt->m_htable, mpt->m_intr_cnt);
12157         } else {
12158                 /* Call ddi_intr_enable for MSI or FIXED interrupts */
12159                 for (i = 0; i < mpt->m_intr_cnt; i++) {
12160                         (void) ddi_intr_enable(mpt->m_htable[i]);
12161                 }
12162         }
12163         return (DDI_SUCCESS);
12164 }
______unchanged_portion_omitted_

12332 static int
12333 mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
12334     uint16_t *dev_handle, mptsas_target_t **pptgt)
12335 {
12336         int             rval;
12337         uint32_t        dev_info;
12338         uint64_t        sas_wwn;
12339         mptsas_phymask_t phymask;
12340         uint8_t         physport, phynum, config, disk;
12341         mptsas_slots_t  *slots = mpt->m_active;
12342         uint64_t        devicename;
12343         uint16_t        pdev_hdl;
12344         mptsas_target_t *tmp_tgt = NULL;
12345         uint16_t        bay_num, enclosure;

12347         ASSERT(*pptgt == NULL);

12349         rval = mptsas_get_sas_device_page0(mpt, page_address, dev_handle,
12350             &sas_wwn, &dev_info, &physport, &phynum, &pdev_hdl,
12351             &bay_num, &enclosure);
12352         if (rval != DDI_SUCCESS) {
12353                 rval = DEV_INFO_FAIL_PAGE0;
12354                 return (rval);
12355         }

12357         if ((dev_info & (MPI2_SAS_DEVICE_INFO_SSP_TARGET |
12358             MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
12359             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) == NULL) {
12360                 rval = DEV_INFO_WRONG_DEVICE_TYPE;
12361                 return (rval);
12362         }

12364         /*
12365          * Check if the dev handle is for a Phys Disk. If so, set return value
12366          * and exit.  Don't add Phys Disks to hash.
12367          */
12368         for (config = 0; config < slots->m_num_raid_configs; config++) {
12369                 for (disk = 0; disk < MPTSAS_MAX_DISKS_IN_CONFIG; disk++) {
12370                         if (*dev_handle == slots->m_raidconfig[config].
12371                             m_physdisk_devhdl[disk]) {
12372                                 rval = DEV_INFO_PHYS_DISK;
```



```
12373                                 return (rval);
12374                         }
12375                 }
12376         }

12378         /*
12379          * Get SATA Device Name from SAS device page0 for
12380          * sata device, if device name doesn't exist, set m_sas_wwn to
12381          * 0 for direct attached SATA. For the device behind the expander
12382          * we still can use STP address assigned by expander.
12383          */
12384         if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
12385             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
12386                 mutex_exit(&mpt->m_mutex);
12387                 /* alloc a tmp_tgt to send the cmd */
12388                 tmp_tgt = kmem_zalloc(sizeof (struct mptsas_target),
12389                     KM_SLEEP);
12390                 tmp_tgt->m_devhdl = *dev_handle;
12391                 tmp_tgt->m_deviceinfo = dev_info;
12392                 tmp_tgt->m_qfull_retries = QFULL_RETRIES;
12393                 tmp_tgt->m_qfull_retry_interval =
12394                     drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
12395                 tmp_tgt->m_t_throttle = MAX_THROTTLE;
12396                 devicename = mptsas_get_sata_guid(mpt, tmp_tgt, 0);
12397                 kmem_free(tmp_tgt, sizeof (struct mptsas_target));
12398                 mutex_enter(&mpt->m_mutex);
12399                 if (devicename != 0 && (((devicename >> 56) & 0xf0) == 0x50)) {
12400                         sas_wwn = devicename;
12401                 } else if (dev_info & MPI2_SAS_DEVICE_INFO_DIRECT_ATTACH) {
12402                         sas_wwn = 0;
12403                 }
12404         }

12406         phymask = mptsas_physport_to_phymask(mpt, physport);
12407         *pptgt = mptsas_tgt_alloc(&slots->m_tgttbl, *dev_handle, sas_wwn,
12408             dev_info, phymask, phynum);
13079             dev_info, phymask, phynum, mpt);
12409         if (*pptgt == NULL) {
12410                 mptsas_log(mpt, CE_WARN, "Failed to allocated target"
12411                     "structure!");
12412                 rval = DEV_INFO_FAIL_ALLOC;
12413                 return (rval);
12414         }
12415         (*pptgt)->m_enclosure = enclosure;
12416         (*pptgt)->m_slot_num = bay_num;
12417         return (DEV_INFO_SUCCESS);
12418 }
______unchanged_portion_omitted_

13802 static int
13803 mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *inq, char *guid,
13804     dev_info_t **lun_dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt, int lun)
13805 {
13806         int                     target;
13807         char                    *nodename = NULL;
13808         char                    **compatible = NULL;
13809         int                     ncompatible     = 0;
13810         int                     mdi_rtn = MDI_FAILURE;
13811         int                     rval = DDI_FAILURE;
13812         char                    *old_guid = NULL;
13813         mptsas_t                *mpt = DIP2MPT(pdip);
13814         char                    *lun_addr = NULL;
13815         char                    *wwn_str = NULL;
13816         char                    *attached_wwn_str = NULL;
13817         char                    *component = NULL;
13818         uint8_t                 phy = 0xFF;
```

```
13819         uint64_t                sas_wwn;
13820         int64_t                 lun64 = 0;
13821         uint32_t                devinfo;
13822         uint16_t                dev_hdl;
13823         uint16_t                pdev_hdl;
13824         uint64_t                dev_sas_wwn;
13825         uint64_t                pdev_sas_wwn;
13826         uint32_t                pdev_info;
13827         uint8_t                 physport;
13828         uint8_t                 phy_id;
13829         uint32_t                page_address;
13830         uint16_t                bay_num, enclosure;
13831         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
13832         uint32_t                dev_info;

13834         mutex_enter(&mpt->m_mutex);
13835         target = ptgt->m_devhdl;
13836         sas_wwn = ptgt->m_sas_wwn;
13837         devinfo = ptgt->m_deviceinfo;
13838         phy = ptgt->m_phynum;
13839         mutex_exit(&mpt->m_mutex);

13841         if (sas_wwn) {
13842                 *pip = mptsas_find_path_addr(pdip, sas_wwn, lun);
13843         } else {
13844                 *pip = mptsas_find_path_phy(pdip, phy);
13845         }

13847         if (*pip != NULL) {
13848                 *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
13849                 ASSERT(*lun_dip != NULL);
13850                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, *lun_dip,
13851                     (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM),
13852                     MDI_CLIENT_GUID_PROP, &old_guid) == DDI_SUCCESS) {
13853                         if (strncmp(guid, old_guid, strlen(guid)) == 0) {
13854                                 /*
13855                                  * Same path back online again.
13856                                  */
13857                                 (void) ddi_prop_free(old_guid);
13858                                 if ((!MDI_PI_IS_ONLINE(*pip)) &&
13859                                     (!MDI_PI_IS_STANDBY(*pip)) &&
13860                                     (ptgt->m_tgt_unconfigured == 0)) {
13861                                         rval = mdi_pi_online(*pip, 0);
14533                                         mutex_enter(&mpt->m_mutex);
14534                                         (void) mptsas_set_led_status(mpt, ptgt,
14535                                             0);
14536                                         mutex_exit(&mpt->m_mutex);
13862                                 } else {
13863                                         rval = DDI_SUCCESS;
13864                                 }
13865                                 if (rval != DDI_SUCCESS) {
13866                                         mptsas_log(mpt, CE_WARN, "path:target: "
13867                                             "%x, lun:%x online failed!", target,
13868                                             lun);
13869                                         *pip = NULL;
13870                                         *lun_dip = NULL;
13871                                 }
13872                                 return (rval);
13873                         } else {
13874                                 /*
13875                                  * The GUID of the LUN has changed which maybe
13876                                  * because customer mapped another volume to the
13877                                  * same LUN.
13878                                  */
13879                                 mptsas_log(mpt, CE_WARN, "The GUID of the "
13880                                     "target:%x, lun:%x was changed, maybe "
```



```
13881                                     "because someone mapped another volume "
13882                                     "to the same LUN", target, lun);
13883                                 (void) ddi_prop_free(old_guid);
13884                                 if (!MDI_PI_IS_OFFLINE(*pip)) {
13885                                         rval = mdi_pi_offline(*pip, 0);
13886                                         if (rval != MDI_SUCCESS) {
13887                                                 mptsas_log(mpt, CE_WARN, "path:"
13888                                                     "target:%x, lun:%x offline "
13889                                                     "failed!", target, lun);
13890                                                 *pip = NULL;
13891                                                 *lun_dip = NULL;
13892                                                 return (DDI_FAILURE);
13893                                         }
13894                                 }
13895                                 if (mdi_pi_free(*pip, 0) != MDI_SUCCESS) {
13896                                         mptsas_log(mpt, CE_WARN, "path:target:"
13897                                             "%x, lun:%x free failed!", target,
13898                                             lun);
13899                                         *pip = NULL;
13900                                         *lun_dip = NULL;
13901                                         return (DDI_FAILURE);
13902                                 }
13903                         }
13904                 } else {
13905                         mptsas_log(mpt, CE_WARN, "Can't get client-guid "
13906                             "property for path:target:%x, lun:%x", target, lun);
13907                         *pip = NULL;
13908                         *lun_dip = NULL;
13909                         return (DDI_FAILURE);
13910                 }
13911         }
13912         scsi_hba_nodename_compatible_get(inq, NULL,
13913             inq->inq_dtype, NULL, &nodename, &compatible, &ncompatible);

13915         /*
13916          * if nodename can't be determined then print a message and skip it
13917          */
13918         if (nodename == NULL) {
13919                 mptsas_log(mpt, CE_WARN, "mptsas driver found no compatible "
13920                     "driver for target%d lun %d dtype:0x%02x", target, lun,
13921                     inq->inq_dtype);
13922                 return (DDI_FAILURE);
13923         }

13925         wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
13926         /* The property is needed by MPAPI */
13927         (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);

13929         lun_addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13930         if (guid) {
13931                 (void) sprintf(lun_addr, "w%s,%x", wwn_str, lun);
13932                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
13933         } else {
13934                 (void) sprintf(lun_addr, "p%x,%x", phy, lun);
13935                 (void) sprintf(wwn_str, "p%x", phy);
13936         }

13938         mdi_rtn = mdi_pi_alloc_compatible(pdip, nodename,
13939             guid, lun_addr, compatible, ncompatible,
13940             0, pip);
13941         if (mdi_rtn == MDI_SUCCESS) {

13943                 if (mdi_prop_update_string(*pip, MDI_GUID,
13944                     guid) != DDI_SUCCESS) {
13945                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13946                             "create prop for target %d lun %d (MDI_GUID)",
```

```
13947                           target, lun);
13948                   mdi_rtn = MDI_FAILURE;
13949                   goto virt_create_done;
13950           }

13952           if (mdi_prop_update_int(*pip, LUN_PROP,
13953               lun) != DDI_SUCCESS) {
13954                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13955                       "create prop for target %d lun %d (LUN_PROP)",
13956                       target, lun);
13957                   mdi_rtn = MDI_FAILURE;
13958                   goto virt_create_done;
13959           }
13960           lun64 = (int64_t)lun;
13961           if (mdi_prop_update_int64(*pip, LUN64_PROP,
13962               lun64) != DDI_SUCCESS) {
13963                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13964                       "create prop for target %d (LUN64_PROP)",
13965                       target);
13966                   mdi_rtn = MDI_FAILURE;
13967                   goto virt_create_done;
13968           }
13969           if (mdi_prop_update_string_array(*pip, "compatible",
13970               compatible, ncompatible) !=
13971               DDI_PROP_SUCCESS) {
13972                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13973                       "create prop for target %d lun %d (COMPATIBLE)",
13974                       target, lun);
13975                   mdi_rtn = MDI_FAILURE;
13976                   goto virt_create_done;
13977           }
13978           if (sas_wwn && (mdi_prop_update_string(*pip,
13979               SCSI_ADDR_PROP_TARGET_PORT, wwn_str) != DDI_PROP_SUCCESS)) {
13980                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13981                       "create prop for target %d lun %d "
13982                       "(target-port)", target, lun);
13983                   mdi_rtn = MDI_FAILURE;
13984                   goto virt_create_done;
13985           } else if ((sas_wwn == 0) && (mdi_prop_update_int(*pip,
13986               "sata-phy", phy) != DDI_PROP_SUCCESS)) {
13987                   /*
13988                    * Direct attached SATA device without DeviceName
13989                    */
13990                   mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13991                       "create prop for SAS target %d lun %d "
13992                       "(sata-phy)", target, lun);
13993                   mdi_rtn = MDI_FAILURE;
13994                   goto virt_create_done;
13995           }
13996           mutex_enter(&mpt->m_mutex);

13998           page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
13999               MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14000               (uint32_t)ptgt->m_devhdl;
14001           rval = mptsas_get_sas_device_page0(mpt, page_address,
14002               &dev_hdl, &dev_sas_wwn, &dev_info, &physport,
14003               &phy_id, &pdev_hdl, &bay_num, &enclosure);
14004           if (rval != DDI_SUCCESS) {
14005                   mutex_exit(&mpt->m_mutex);
14006                   mptsas_log(mpt, CE_WARN, "mptsas unable to get "
14007                       "parent device for handle %d", page_address);
14008                   mdi_rtn = MDI_FAILURE;
14009                   goto virt_create_done;
14010           }

14012           page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
```



```
14013               MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
14014           rval = mptsas_get_sas_device_page0(mpt, page_address,
14015               &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
14016               &phy_id, &pdev_hdl, &bay_num, &enclosure);
14017           if (rval != DDI_SUCCESS) {
14018                   mutex_exit(&mpt->m_mutex);
14019                   mptsas_log(mpt, CE_WARN, "mptsas unable to get"
14020                       "device info for handle %d", page_address);
14021                   mdi_rtn = MDI_FAILURE;
14022                   goto virt_create_done;
14023           }

14025           mutex_exit(&mpt->m_mutex);

14027           /*
14028            * If this device direct attached to the controller
14029            * set the attached-port to the base wwid
14030            */
14031           if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
14032               != DEVINFO_DIRECT_ATTACHED) {
14033                   (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14034                       pdev_sas_wwn);
14035           } else {
14036                   /*
14037                    * Update the iport's attached-port to guid
14038                    */
14039                   if (sas_wwn == 0) {
14040                           (void) sprintf(wwn_str, "p%x", phy);
14041                   } else {
14042                           (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14043                   }
14044                   if (ddi_prop_update_string(DDI_DEV_T_NONE,
14045                       pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
14046                       DDI_PROP_SUCCESS) {
14047                           mptsas_log(mpt, CE_WARN,
14048                               "mptsas unable to create "
14049                               "property for iport target-port"
14050                               " %s (sas_wwn)",
14051                               wwn_str);
14052                           mdi_rtn = MDI_FAILURE;
14053                           goto virt_create_done;
14054                   }

14056                   (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14057                       mpt->un.m_base_wwid);
14058           }

14060           if (mdi_prop_update_string(*pip,
14061               SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
14062               DDI_PROP_SUCCESS) {
14063                   mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14064                       "property for iport attached-port %s (sas_wwn)",
14065                       attached_wwn_str);
14066                   mdi_rtn = MDI_FAILURE;
14067                   goto virt_create_done;
14068           }


14071           if (inq->inq_dtype == 0) {
14072                   component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
14073                   /*
14074                    * set obp path for pathinfo
14075                    */
14076                   (void) snprintf(component, MAXPATHLEN,
14077                       "disk@%s", lun_addr);
```

```
14079                         if (mdi_pi_pathname_obp_set(*pip, component) !=
14080                             DDI_SUCCESS) {
14081                                 mptsas_log(mpt, CE_WARN, "mpt_sas driver "
14082                                     "unable to set obp-path for object %s",
14083                                     component);
14084                                 mdi_rtn = MDI_FAILURE;
14085                                 goto virt_create_done;
14086                         }
14087                 }

14089                 *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
14090                 if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14091                     MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14092                         if ((ndi_prop_update_int(DDI_DEV_T_NONE, *lun_dip,
14093                             "pm-capable", 1)) !=
14094                             DDI_PROP_SUCCESS) {
14095                                 mptsas_log(mpt, CE_WARN, "mptsas driver"
14096                                     "failed to create pm-capable "
14097                                     "property, target %d", target);
14098                                 mdi_rtn = MDI_FAILURE;
14099                                 goto virt_create_done;
14100                         }
14101                 }
14102                 /*
14103                  * Create the phy-num property
14104                  */
14105                 if (mdi_prop_update_int(*pip, "phy-num",
14106                     ptgt->m_phynum) != DDI_SUCCESS) {
14107                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
14108                             "create phy-num property for target %d lun %d",
14109                             target, lun);
14110                         mdi_rtn = MDI_FAILURE;
14111                         goto virt_create_done;
14112                 }
14113                 NDBG20(("new path:%s onlining,", MDI_PI(*pip)->pi_addr));
14114                 mdi_rtn = mdi_pi_online(*pip, 0);
14790                 if (mdi_rtn == MDI_SUCCESS) {
14791                         mutex_enter(&mpt->m_mutex);
14792                         if (mptsas_set_led_status(mpt, ptgt, 0) !=
14793                             DDI_SUCCESS) {
14794                                 NDBG14(("mptsas: clear LED for slot %x "
14795                                     "failed", ptgt->m_slot_num));
14796                         }
14797                         mutex_exit(&mpt->m_mutex);
14798                 }
14115                 if (mdi_rtn == MDI_NOT_SUPPORTED) {
14116                         mdi_rtn = MDI_FAILURE;
14117                 }
14118 virt_create_done:
14119                 if (*pip && mdi_rtn != MDI_SUCCESS) {
14120                         (void) mdi_pi_free(*pip, 0);
14121                         *pip = NULL;
14122                         *lun_dip = NULL;
14123                 }
14124         }

14126         scsi_hba_nodename_compatible_free(nodename, compatible);
14127         if (lun_addr != NULL) {
14128                 kmem_free(lun_addr, SCSI_MAXNAMELEN);
14129         }
14130         if (wwn_str != NULL) {
14131                 kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14132         }
14133         if (component != NULL) {
14134                 kmem_free(component, MAXPATHLEN);
14135         }
```



```
14137         return ((mdi_rtn == MDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14138 }

14140 static int
14141 mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *inq,
14142     char *guid, dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14143 {
14144         int                     target;
14145         int                     rval;
14146         int                     ndi_rtn = NDI_FAILURE;
14147         uint64_t                be_sas_wwn;
14148         char                    *nodename = NULL;
14149         char                    **compatible = NULL;
14150         int                     ncompatible = 0;
14151         int                     instance = 0;
14152         mptsas_t                *mpt = DIP2MPT(pdip);
14153         char                    *wwn_str = NULL;
14154         char                    *component = NULL;
14155         char                    *attached_wwn_str = NULL;
14156         uint8_t                 phy = 0xFF;
14157         uint64_t                sas_wwn;
14158         uint32_t                devinfo;
14159         uint16_t                dev_hdl;
14160         uint16_t                pdev_hdl;
14161         uint64_t                pdev_sas_wwn;
14162         uint64_t                dev_sas_wwn;
14163         uint32_t                pdev_info;
14164         uint8_t                 physport;
14165         uint8_t                 phy_id;
14166         uint32_t                page_address;
14167         uint16_t                bay_num, enclosure;
14168         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
14169         uint32_t                dev_info;
14170         int64_t                 lun64 = 0;

14172         mutex_enter(&mpt->m_mutex);
14173         target = ptgt->m_devhdl;
14174         sas_wwn = ptgt->m_sas_wwn;
14175         devinfo = ptgt->m_deviceinfo;
14176         phy = ptgt->m_phynum;
14177         mutex_exit(&mpt->m_mutex);

14179         /*
14180          * generate compatible property with binding-set "mpt"
14181          */
14182         scsi_hba_nodename_compatible_get(inq, NULL, inq->inq_dtype, NULL,
14183             &nodename, &compatible, &ncompatible);

14185         /*
14186          * if nodename can't be determined then print a message and skip it
14187          */
14188         if (nodename == NULL) {
14189                 mptsas_log(mpt, CE_WARN, "mptsas found no compatible driver "
14190                     "for target %d lun %d", target, lun);
14191                 return (DDI_FAILURE);
14192         }

14194         ndi_rtn = ndi_devi_alloc(pdip, nodename,
14195             DEVI_SID_NODEID, lun_dip);

14197         /*
14198          * if lun alloc success, set props
14199          */
14200         if (ndi_rtn == NDI_SUCCESS) {
```

```
14202                 if (ndi_prop_update_int(DDI_DEV_T_NONE,
14203                     *lun_dip, LUN_PROP, lun) !=
14204                     DDI_PROP_SUCCESS) {
14205                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14206                             "property for target %d lun %d (LUN_PROP)",
14207                             target, lun);
14208                         ndi_rtn = NDI_FAILURE;
14209                         goto phys_create_done;
14210                 }

14212                 lun64 = (int64_t)lun;
14213                 if (ndi_prop_update_int64(DDI_DEV_T_NONE,
14214                     *lun_dip, LUN64_PROP, lun64) !=
14215                     DDI_PROP_SUCCESS) {
14216                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14217                             "property for target %d lun64 %d (LUN64_PROP)",
14218                             target, lun);
14219                         ndi_rtn = NDI_FAILURE;
14220                         goto phys_create_done;
14221                 }
14222                 if (ndi_prop_update_string_array(DDI_DEV_T_NONE,
14223                     *lun_dip, "compatible", compatible, ncompatible)
14224                     != DDI_PROP_SUCCESS) {
14225                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14226                             "property for target %d lun %d (COMPATIBLE)",
14227                             target, lun);
14228                         ndi_rtn = NDI_FAILURE;
14229                         goto phys_create_done;
14230                 }

14232                 /*
14233                  * We need the SAS WWN for non-multipath devices, so
14234                  * we'll use the same property as that multipathing
14235                  * devices need to present for MPAPI. If we don't have
14236                  * a WWN (e.g. parallel SCSI), don't create the prop.
14237                  */
14238                 wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
14239                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14240                 if (sas_wwn && ndi_prop_update_string(DDI_DEV_T_NONE,
14241                     *lun_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str)
14242                     != DDI_PROP_SUCCESS) {
14243                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14244                             "create property for SAS target %d lun %d "
14245                             "(target-port)", target, lun);
14246                         ndi_rtn = NDI_FAILURE;
14247                         goto phys_create_done;
14248                 }

14250                 be_sas_wwn = BE_64(sas_wwn);
14251                 if (sas_wwn && ndi_prop_update_byte_array(
14252                     DDI_DEV_T_NONE, *lun_dip, "port-wwn",
14253                     (uchar_t *)&be_sas_wwn, 8) != DDI_PROP_SUCCESS) {
14254                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14255                             "create property for SAS target %d lun %d "
14256                             "(port-wwn)", target, lun);
14257                         ndi_rtn = NDI_FAILURE;
14258                         goto phys_create_done;
14259                 } else if ((sas_wwn == 0) && (ndi_prop_update_int(
14260                     DDI_DEV_T_NONE, *lun_dip, "sata-phy", phy) !=
14261                     DDI_PROP_SUCCESS)) {
14262                         /*
14263                          * Direct attached SATA device without DeviceName
14264                          */
14265                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14266                             "create property for SAS target %d lun %d "
14267                             "(sata-phy)", target, lun);
```



```
14268                         ndi_rtn = NDI_FAILURE;
14269                         goto phys_create_done;
14270                 }

14272                 if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
14273                     *lun_dip, SAS_PROP) != DDI_PROP_SUCCESS) {
14274                         mptsas_log(mpt, CE_WARN, "mptsas unable to"
14275                             "create property for SAS target %d lun %d"
14276                             " (SAS_PROP)", target, lun);
14277                         ndi_rtn = NDI_FAILURE;
14278                         goto phys_create_done;
14279                 }
14280                 if (guid && (ndi_prop_update_string(DDI_DEV_T_NONE,
14281                     *lun_dip, NDI_GUID, guid) != DDI_SUCCESS)) {
14282                         mptsas_log(mpt, CE_WARN, "mptsas unable "
14283                             "to create guid property for target %d "
14284                             "lun %d", target, lun);
14285                         ndi_rtn = NDI_FAILURE;
14286                         goto phys_create_done;
14287                 }

14289                 /*
14290                  * The following code is to set properties for SM-HBA support,
14291                  * it doesn't apply to RAID volumes
14292                  */
14293                 if (ptgt->m_phymask == 0)
14294                         goto phys_raid_lun;

14296                 mutex_enter(&mpt->m_mutex);

14298                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14299                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14300                     (uint32_t)ptgt->m_devhdl;
14301                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14302                     &dev_hdl, &dev_sas_wwn, &dev_info,
14303                     &physport, &phy_id, &pdev_hdl,
14304                     &bay_num, &enclosure);
14305                 if (rval != DDI_SUCCESS) {
14306                         mutex_exit(&mpt->m_mutex);
14307                         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
14308                             "parent device for handle %d.", page_address);
14309                         ndi_rtn = NDI_FAILURE;
14310                         goto phys_create_done;
14311                 }

14313                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14314                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
14315                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14316                     &dev_hdl, &pdev_sas_wwn, &pdev_info,
14317                     &physport, &phy_id, &pdev_hdl, &bay_num, &enclosure);
14318                 if (rval != DDI_SUCCESS) {
14319                         mutex_exit(&mpt->m_mutex);
14320                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14321                             "device for handle %d.", page_address);
14322                         ndi_rtn = NDI_FAILURE;
14323                         goto phys_create_done;
14324                 }

14326                 mutex_exit(&mpt->m_mutex);

14328                 /*
14329                  * If this device direct attached to the controller
14330                  * set the attached-port to the base wwid
14331                  */
14332                 if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
14333                     != DEVINFO_DIRECT_ATTACHED) {
```

```
14334                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14335                               pdev_sas_wwn);
14336                   } else {
14337                           /*
14338                            * Update the iport's attached-port to guid
14339                            */
14340                           if (sas_wwn == 0) {
14341                                   (void) sprintf(wwn_str, "p%x", phy);
14342                           } else {
14343                                   (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14344                           }
14345                           if (ddi_prop_update_string(DDI_DEV_T_NONE,
14346                               pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
14347                               DDI_PROP_SUCCESS) {
14348                                   mptsas_log(mpt, CE_WARN,
14349                                       "mptsas unable to create "
14350                                       "property for iport target-port"
14351                                       " %s (sas_wwn)",
14352                                       wwn_str);
14353                                   ndi_rtn = NDI_FAILURE;
14354                                   goto phys_create_done;
14355                           }

14357                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14358                               mpt->un.m_base_wwid);
14359                   }

14361                   if (ndi_prop_update_string(DDI_DEV_T_NONE,
14362                       *lun_dip, SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
14363                       DDI_PROP_SUCCESS) {
14364                           mptsas_log(mpt, CE_WARN,
14365                               "mptsas unable to create "
14366                               "property for iport attached-port %s (sas_wwn)",
14367                               attached_wwn_str);
14368                           ndi_rtn = NDI_FAILURE;
14369                           goto phys_create_done;
14370                   }

14372                   if (IS_SATA_DEVICE(dev_info)) {
14373                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
14374                               *lun_dip, MPTSAS_VARIANT, "sata") !=
14375                               DDI_PROP_SUCCESS) {
14376                                   mptsas_log(mpt, CE_WARN,
14377                                       "mptsas unable to create "
14378                                       "property for device variant ");
14379                                   ndi_rtn = NDI_FAILURE;
14380                                   goto phys_create_done;
14381                           }
14382                   }

14384                   if (IS_ATAPI_DEVICE(dev_info)) {
14385                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
14386                               *lun_dip, MPTSAS_VARIANT, "atapi") !=
14387                               DDI_PROP_SUCCESS) {
14388                                   mptsas_log(mpt, CE_WARN,
14389                                       "mptsas unable to create "
14390                                       "property for device variant ");
14391                                   ndi_rtn = NDI_FAILURE;
14392                                   goto phys_create_done;
14393                           }
14394                   }

14396 phys_raid_lun:
14397                   /*
14398                    * if this is a SAS controller, and the target is a SATA
14399                    * drive, set the 'pm-capable' property for sd and if on
```



```
14400                    * an OPL platform, also check if this is an ATAPI
14401                    * device.
14402                    */
14403                   instance = ddi_get_instance(mpt->m_dip);
14404                   if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14405                       MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14406                           NDBG2(("mptsas%d: creating pm-capable property, "
14407                               "target %d", instance, target));

14409                           if ((ndi_prop_update_int(DDI_DEV_T_NONE,
14410                               *lun_dip, "pm-capable", 1)) !=
14411                               DDI_PROP_SUCCESS) {
14412                                   mptsas_log(mpt, CE_WARN, "mptsas "
14413                                       "failed to create pm-capable "
14414                                       "property, target %d", target);
14415                                   ndi_rtn = NDI_FAILURE;
14416                                   goto phys_create_done;
14417                           }

14419                   }

14421                   if ((inq->inq_dtype == 0) || (inq->inq_dtype == 5)) {
14422                           /*
14423                            * add 'obp-path' properties for devinfo
14424                            */
14425                           bzero(wwn_str, sizeof (wwn_str));
14426                           (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
14427                           component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
14428                           if (guid) {
14429                                   (void) snprintf(component, MAXPATHLEN,
14430                                       "disk@w%s,%x", wwn_str, lun);
14431                           } else {
14432                                   (void) snprintf(component, MAXPATHLEN,
14433                                       "disk@p%x,%x", phy, lun);
14434                           }
14435                           if (ddi_pathname_obp_set(*lun_dip, component)
14436                               != DDI_SUCCESS) {
14437                                   mptsas_log(mpt, CE_WARN, "mpt_sas driver "
14438                                       "unable to set obp-path for SAS "
14439                                       "object %s", component);
14440                                   ndi_rtn = NDI_FAILURE;
14441                                   goto phys_create_done;
14442                           }
14443                   }
14444                   /*
14445                    * Create the phy-num property for non-raid disk
14446                    */
14447                   if (ptgt->m_phymask != 0) {
14448                           if (ndi_prop_update_int(DDI_DEV_T_NONE,
14449                               *lun_dip, "phy-num", ptgt->m_phynum) !=
14450                               DDI_PROP_SUCCESS) {
14451                                   mptsas_log(mpt, CE_WARN, "mptsas driver "
14452                                       "failed to create phy-num property for "
14453                                       "target %d", target);
14454                                   ndi_rtn = NDI_FAILURE;
14455                                   goto phys_create_done;
14456                           }
14457                   }
14458 phys_create_done:
14459                   /*
14460                    * If props were setup ok, online the lun
14461                    */
14462                   if (ndi_rtn == NDI_SUCCESS) {
14463                           /*
14464                            * Try to online the new node
14465                            */
```

```
14466                           ndi_rtn = ndi_devi_online(*lun_dip, NDI_ONLINE_ATTACH);
14467                   }
15152                   if (ndi_rtn == NDI_SUCCESS) {
15153                           mutex_enter(&mpt->m_mutex);
15154                           if (mptsas_set_led_status(mpt, ptgt, 0) !=
15155                               DDI_SUCCESS) {
15156                                   NDBG14(("mptsas: clear LED for tgt %x "
15157                                       "failed", ptgt->m_slot_num));
15158                           }
15159                           mutex_exit(&mpt->m_mutex);
15160                   }

14469                   /*
14470                    * If success set rtn flag, else unwire alloc'd lun
14471                    */
14472                   if (ndi_rtn != NDI_SUCCESS) {
14473                           NDBG12(("mptsas driver unable to online "
14474                               "target %d lun %d", target, lun));
14475                           ndi_prop_remove_all(*lun_dip);
14476                           (void) ndi_devi_free(*lun_dip);
14477                           *lun_dip = NULL;
14478                   }
14479           }

14481           scsi_hba_nodename_compatible_free(nodename, compatible);

14483           if (wwn_str != NULL) {
14484                   kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14485           }
14486           if (component != NULL) {
14487                   kmem_free(component, MAXPATHLEN);
14488           }


14491           return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14492   }
```

_____unchanged_portion_omitted_

```
15092   mptsas_target_t *
15093   mptsas_tgt_alloc(mptsas_hash_table_t *hashtab, uint16_t devhdl, uint64_t wwid,
15094       uint32_t devinfo, mptsas_phymask_t phymask, uint8_t phynum)
15787       uint32_t devinfo, mptsas_phymask_t phymask, uint8_t phynum, mptsas_t *mpt)
15095   {
15096           mptsas_target_t *tmp_tgt = NULL;

15098           tmp_tgt = mptsas_hash_search(hashtab, wwid, phymask);
15099           if (tmp_tgt != NULL) {
15100                   NDBG20(("Hash item already exist"));
15101                   tmp_tgt->m_deviceinfo = devinfo;
15102                   tmp_tgt->m_devhdl = devhdl;
15103                   return (tmp_tgt);
15104           }
15105           tmp_tgt = kmem_zalloc(sizeof (struct mptsas_target), KM_SLEEP);
15106           if (tmp_tgt == NULL) {
15107                   cmn_err(CE_WARN, "Fatal, allocated tgt failed");
15108                   return (NULL);
15109           }
15110           tmp_tgt->m_devhdl = devhdl;
15111           tmp_tgt->m_sas_wwn = wwid;
15112           tmp_tgt->m_deviceinfo = devinfo;
15113           tmp_tgt->m_phymask = phymask;
15114           tmp_tgt->m_phynum = phynum;
15115           /* Initialized the tgt structure */
15116           tmp_tgt->m_qfull_retries = QFULL_RETRIES;
15117           tmp_tgt->m_qfull_retry_interval =
15118               drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
```



```
15119           tmp_tgt->m_t_throttle = MAX_THROTTLE;
15813           mutex_init(&tmp_tgt->m_tgt_intr_mutex, NULL, MUTEX_DRIVER,
15814               DDI_INTR_PRI(mpt->m_intr_pri));

15121           mptsas_hash_add(hashtab, tmp_tgt);

15123           return (tmp_tgt);
15124   }

15126   static void
15127   mptsas_tgt_free(mptsas_hash_table_t *hashtab, uint64_t wwid,
15128       mptsas_phymask_t phymask)
15129   {
15130           mptsas_target_t *tmp_tgt;
15131           tmp_tgt = mptsas_hash_rem(hashtab, wwid, phymask);
15132           if (tmp_tgt == NULL) {
15133                   cmn_err(CE_WARN, "Tgt not found, nothing to free");
15134           } else {
15830                   mutex_destroy(&tmp_tgt->m_tgt_intr_mutex);
15135                   kmem_free(tmp_tgt, sizeof (struct mptsas_target));
15136           }
15137   }
```

_____unchanged_portion_omitted_

```
16037   static mptsas_target_t *
16038   mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr, mptsas_phymask_t phymask)
16039   {
16040           uint8_t                 phynum;
16041           uint64_t                wwn;
16042           int                     lun;
16043           mptsas_target_t         *ptgt = NULL;

16045           if (mptsas_parse_address(addr, &wwn, &phynum, &lun) != DDI_SUCCESS) {
16046                   return (NULL);
16047           }
16048           if (addr[0] == 'w') {
16049                   ptgt = mptsas_wwid_to_ptgt(mpt, (int)phymask, wwn);
16050           } else {
16051                   ptgt = mptsas_phy_to_tgt(mpt, (int)phymask, phynum);
16052           }
16053           return (ptgt);
16054   }

16056   #ifdef MPTSAS_GET_LED
16057   static int
16058   mptsas_get_led_status(mptsas_t *mpt, mptsas_target_t *ptgt,
16059       uint32_t *slotstatus)
16060   {
16061           return (mptsas_send_sep(mpt, ptgt, slotstatus,
16062               MPI2_SEP_REQ_ACTION_READ_STATUS));
16063   }
16064   #endif
16065   static int
16066   mptsas_set_led_status(mptsas_t *mpt, mptsas_target_t *ptgt, uint32_t slotstatus)
16067   {
16068           NDBG14(("mptsas_ioctl: set LED status %x for slot %x",
16069               slotstatus, ptgt->m_slot_num));
16070           return (mptsas_send_sep(mpt, ptgt, &slotstatus,
16071               MPI2_SEP_REQ_ACTION_WRITE_STATUS));
16072   }
16073   /*
16074    *  send sep request, use enclosure/slot addressing
16075    */
16076   static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
16077       uint32_t *status, uint8_t act)
16078   {
16079           Mpi2SepRequest_t        req;
```

```
16080         Mpi2SepReply_t          rep;
16081         int                     ret;

16083         ASSERT(mutex_owned(&mpt->m_mutex));

16085         bzero(&req, sizeof (req));
16086         bzero(&rep, sizeof (rep));

16088         /* Do nothing for RAID volumes */
16089         if (ptgt->m_phymask == 0) {
16090                 NDBG14(("mptsas_send_sep: Skip RAID volumes"));
16091                 return (DDI_FAILURE);
16092         }

16094         req.Function = MPI2_FUNCTION_SCSI_ENCLOSURE_PROCESSOR;
16095         req.Action = act;
16096         req.Flags = MPI2_SEP_REQ_FLAGS_ENCLOSURE_SLOT_ADDRESS;
16097         req.EnclosureHandle = LE_16(ptgt->m_enclosure);
16098         req.Slot = LE_16(ptgt->m_slot_num);
16099         if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16100                 req.SlotStatus = LE_32(*status);
16101         }
16102         ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
16103             sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
16104         if (ret != 0) {
16105                 mptsas_log(mpt, CE_NOTE, "mptsas_send_sep: passthru SEP "
16106                     "Processor Request message error %d", ret);
16107                 return (DDI_FAILURE);
16108         }
16109         /* do passthrough success, check the ioc status */
16110         if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
16111                 if ((LE_16(rep.IOCStatus) & MPI2_IOCSTATUS_MASK) ==
16112                     MPI2_IOCSTATUS_INVALID_FIELD) {
16113                         mptsas_log(mpt, CE_NOTE, "send sep act %x: Not "
16114                             "supported action, loginfo %x", act,
16115                             LE_32(rep.IOCLogInfo));
16116                         return (DDI_FAILURE);
16117                 }
16118                 mptsas_log(mpt, CE_NOTE, "send_sep act %x: ioc "
16119                     "status:%x", act, LE_16(rep.IOCStatus));
16120                 return (DDI_FAILURE);
16121         }
16122         if (act != MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
16123                 *status = LE_32(rep.SlotStatus);
16124         }

16126         return (DDI_SUCCESS);
16127 }

15342 int
15343 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
15344     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
15345     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
15346 {
15347         ddi_dma_cookie_t        new_cookie;
15348         size_t                  alloc_len;
15349         uint_t                  ncookie;

15351         if (cookiep == NULL)
15352                 cookiep = &new_cookie;

15354         if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
15355             NULL, dma_hdp) != DDI_SUCCESS) {
15356                 dma_hdp = NULL;
15357                 return (FALSE);
15358         }
```



```
15360         if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
15361             DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
15362             acc_hdp) != DDI_SUCCESS) {
15363                 ddi_dma_free_handle(dma_hdp);
15364                 dma_hdp = NULL;
15365                 return (FALSE);
15366         }

15368         if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
15369             (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
15370             cookiep, &ncookie) != DDI_DMA_MAPPED) {
15371                 (void) ddi_dma_mem_free(acc_hdp);
15372                 ddi_dma_free_handle(dma_hdp);
15373                 dma_hdp = NULL;
15374                 return (FALSE);
15375         }

15377         return (TRUE);
15378 }

15380 void
15381 mptsas_dma_addr_destroy(ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp)
15382 {
15383         if (*dma_hdp == NULL)
15384                 return;

15386         (void) ddi_dma_unbind_handle(*dma_hdp);
15387         (void) ddi_dma_mem_free(acc_hdp);
15388         ddi_dma_free_handle(dma_hdp);
15389         dma_hdp = NULL;
16177 }

16179 static int
16180 mptsas_outstanding_cmds_n(mptsas_t *mpt)
16181 {
16182         int n = 0, i;
16183         for (i = 0; i < mpt->m_slot_freeq_pair_n; i++) {
16184                 mutex_enter(&mpt->m_slot_freeq_pairp[i].
16185                     m_slot_allocq.s.m_fq_mutex);
16186                 mutex_enter(&mpt->m_slot_freeq_pairp[i].
16187                     m_slot_releq.s.m_fq_mutex);
16188                 n += (mpt->m_slot_freeq_pairp[i].m_slot_allocq.s.m_fq_n_init -
16189                     mpt->m_slot_freeq_pairp[i].m_slot_allocq.s.m_fq_n -
16190                     mpt->m_slot_freeq_pairp[i].m_slot_releq.s.m_fq_n);
16191                 mutex_exit(&mpt->m_slot_freeq_pairp[i].
16192                     m_slot_releq.s.m_fq_mutex);
16193                 mutex_exit(&mpt->m_slot_freeq_pairp[i].
16194                     m_slot_allocq.s.m_fq_mutex);
16195         }
16196         if (mpt->m_max_requests - 2 < n)
16197                 panic("mptsas: free slot allocq and releq crazy");
16198         return (n);
15390 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   79763 Tue Dec  4 12:30:15 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
XXXX Nexenta fixes for mpt_sas(7d)
**********************************************************
______unchanged_portion_omitted_

 200 void
 201 mptsas_start_config_page_access(mptsas_t *mpt, mptsas_cmd_t *cmd)
 202 {
 203         pMpi2ConfigRequest_t    request;
 204         pMpi2SGESimple64_t      sge;
 205         struct scsi_pkt         *pkt = cmd->cmd_pkt;
 206         mptsas_config_request_t *config = pkt->pkt_ha_private;
 207         uint8_t                 direction;
 208         uint32_t                length, flagslength, request_desc_low;

 210         ASSERT(mutex_owned(&mpt->m_mutex));

 212         /*
 213          * Point to the correct message and clear it as well as the global
 214          * config page memory.
 215          */
 216         request = (pMpi2ConfigRequest_t)(mpt->m_req_frame +
 217             (mpt->m_req_frame_size * cmd->cmd_slot));
 218         bzero(request, mpt->m_req_frame_size);

 220         /*
 221          * Form the request message.
 222          */
 223         ddi_put8(mpt->m_acc_req_frame_hdl, &request->Function,
 224             MPI2_FUNCTION_CONFIG);
 225         ddi_put8(mpt->m_acc_req_frame_hdl, &request->Action, config->action);
 226         direction = MPI2_SGE_FLAGS_IOC_TO_HOST;
 227         length = 0;
 228         sge = (pMpi2SGESimple64_t)&request->PageBufferSGE;
 229         if (config->action == MPI2_CONFIG_ACTION_PAGE_HEADER) {
 230                 if (config->page_type > MPI2_CONFIG_PAGETYPE_MASK) {
 231                         ddi_put8(mpt->m_acc_req_frame_hdl,
 232                             &request->Header.PageType,
 233                             MPI2_CONFIG_PAGETYPE_EXTENDED);
 234                         ddi_put8(mpt->m_acc_req_frame_hdl,
 235                             &request->ExtPageType, config->page_type);
 236                 } else {
 237                         ddi_put8(mpt->m_acc_req_frame_hdl,
 238                             &request->Header.PageType, config->page_type);
 239                 }
 240         } else {
 241                 ddi_put8(mpt->m_acc_req_frame_hdl, &request->ExtPageType,
 242                     config->ext_page_type);
 243                 ddi_put16(mpt->m_acc_req_frame_hdl, &request->ExtPageLength,
 244                     config->ext_page_length);
 245                 ddi_put8(mpt->m_acc_req_frame_hdl, &request->Header.PageType,
 246                     config->page_type);
 247                 ddi_put8(mpt->m_acc_req_frame_hdl, &request->Header.PageLength,
 248                     config->page_length);
 249                 ddi_put8(mpt->m_acc_req_frame_hdl,
 250                     &request->Header.PageVersion, config->page_version);
 251                 if ((config->page_type & MPI2_CONFIG_PAGETYPE_MASK) ==
 252                     MPI2_CONFIG_PAGETYPE_EXTENDED) {
 253                         length = config->ext_page_length * 4;
 254                 } else {
 255                         length = config->page_length * 4;
 256                 }

 258                 if (config->action == MPI2_CONFIG_ACTION_PAGE_WRITE_NVRAM) {
```



```
 259                         direction = MPI2_SGE_FLAGS_HOST_TO_IOC;
 260                 }
 261                 ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.Low,
 262                     (uint32_t)cmd->cmd_dma_addr);
 263                 ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.High,
 264                     (uint32_t)(cmd->cmd_dma_addr >> 32));
 265         }
 266         ddi_put8(mpt->m_acc_req_frame_hdl, &request->Header.PageNumber,
 267             config->page_number);
 268         ddi_put32(mpt->m_acc_req_frame_hdl, &request->PageAddress,
 269             config->page_address);
 270         flagslength = ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
 271             MPI2_SGE_FLAGS_END_OF_BUFFER |
 272             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
 273             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
 274             MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
 275             direction |
 276             MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
 277         flagslength |= length;
 278         ddi_put32(mpt->m_acc_req_frame_hdl, &sge->FlagsLength, flagslength);

 280         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
 281             DDI_DMA_SYNC_FORDEV);
 282         request_desc_low = (cmd->cmd_slot << 16) +
 283             MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
 284         cmd->cmd_rfm = NULL;
 285         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
 285         MPTSAS_START_CMD(mpt, request_desc_low, 0);
 286         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
 287             DDI_SUCCESS) ||
 288             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
 289             DDI_SUCCESS)) {
 290                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
 291         }
 292 }
______unchanged_portion_omitted_

1202 int
1203 mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
1204     uint8_t type, int mode)
1205 {

1207         /*
1208          * In order to avoid allocating variables on the stack,
1209          * we make use of the pre-existing mptsas_cmd_t and
1210          * scsi_pkt which are included in the mptsas_t which
1211          * is passed to this routine.
1212          */

1214         ddi_dma_attr_t          flsh_dma_attrs;
1215         ddi_dma_cookie_t        flsh_cookie;
1216         ddi_dma_handle_t        flsh_dma_handle;
1217         ddi_acc_handle_t        flsh_accessp;
1218         caddr_t                 memp, flsh_memp;
1219         uint32_t                flagslength;
1220         pMpi2FWDownloadRequest  fwdownload;
1221         pMpi2FWDownloadTCSGE_t  tcsge;
1222         pMpi2SGESimple64_t      sge;
1223         mptsas_cmd_t            *cmd;
1224         struct scsi_pkt         *pkt;
1225         int                     i;
1226         int                     rvalue = 0;
1227         uint32_t                request_desc_low;

1229         if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
1230                 mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): allocation "
```

```
1231                    "failed. event ack command pool is full\n");
1232                return (rvalue);
1233        }

1235        bzero((caddr_t)cmd, sizeof (*cmd));
1236        bzero((caddr_t)pkt, scsi_pkt_size());
1237        cmd->ioc_cmd_slot = (uint32_t)rvalue;

1239        /*
1240         * dynamically create a customized dma attribute structure
1241         * that describes the flash file.
1242         */
1243        flsh_dma_attrs = mpt->m_msg_dma_attr;
1244        flsh_dma_attrs.dma_attr_sgllen = 1;

1246        if (mptsas_dma_addr_create(mpt, flsh_dma_attrs, &flsh_dma_handle,
1247            &flsh_accessp, &flsh_memp, size, &flsh_cookie) == FALSE) {
1248                mptsas_log(mpt, CE_WARN,
1249                    "(unable to allocate dma resource.");
1250                mptsas_return_to_pool(mpt, cmd);
1251                return (-1);
1252        }

1254        bzero(flsh_memp, size);

1256        for (i = 0; i < size; i++) {
1257                (void) ddi_copyin(ptrbuffer + i, flsh_memp + i, 1, mode);
1258        }
1259        (void) ddi_dma_sync(flsh_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);

1261        /*
1262         * form a cmd/pkt to store the fw download message
1263         */
1264        pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
1265        pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
1266        pkt->pkt_ha_private     = (opaque_t)cmd;
1267        pkt->pkt_flags          = FLAG_HEAD;
1268        pkt->pkt_time           = 60;
1269        cmd->cmd_pkt            = pkt;
1270        cmd->cmd_scblen         = 1;
1271        cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_CMD;

1273        /*
1274         * Save the command in a slot
1275         */
1276        if (mptsas_save_cmd(mpt, cmd) == FALSE) {
1277                mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);
1278                mptsas_return_to_pool(mpt, cmd);
1279                return (-1);
1280        }

1282        /*
1283         * Fill in fw download message
1284         */
1285        ASSERT(cmd->cmd_slot != 0);
1286        memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
1287        bzero(memp, mpt->m_req_frame_size);
1288        fwdownload = (void *)memp;
1289        ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->Function,
1290            MPI2_FUNCTION_FW_DOWNLOAD);
1291        ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->ImageType, type);
1292        ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->MsgFlags,
1293            MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1294        ddi_put32(mpt->m_acc_req_frame_hdl, &fwdownload->TotalImageSize, size);

1296        tcsge = (pMpi2FWDownloadTCSGE_t)&fwdownload->SGL;
```



```
1297        ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->ContextSize, 0);
1298        ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->DetailsLength, 12);
1299        ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->Flags, 0);
1300        ddi_put32(mpt->m_acc_req_frame_hdl, &tcsge->ImageOffset, 0);
1301        ddi_put32(mpt->m_acc_req_frame_hdl, &tcsge->ImageSize, size);

1303        sge = (pMpi2SGESimple64_t)(tcsge + 1);
1304        flagslength = size;
1305        flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
1306            MPI2_SGE_FLAGS_END_OF_BUFFER |
1307            MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
1308            MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
1309            MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
1310            MPI2_SGE_FLAGS_HOST_TO_IOC |
1311            MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
1312        ddi_put32(mpt->m_acc_req_frame_hdl, &sge->FlagsLength, flagslength);
1313        ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.Low,
1314            flsh_cookie.dmac_address);
1315        ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.High,
1316            (uint32_t)(flsh_cookie.dmac_laddress >> 32));

1318        /*
1319         * Start command
1320         */
1321        (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1322            DDI_DMA_SYNC_FORDEV);
1323        request_desc_low = (cmd->cmd_slot << 16) +
1324            MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
1325        cmd->cmd_rfm = NULL;
1327        mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
1326        MPTSAS_START_CMD(mpt, request_desc_low, 0);

1328        rvalue = 0;
1329        (void) cv_reltimedwait(&mpt->m_fw_cv, &mpt->m_mutex,
1330            drv_usectohz(60 * MICROSEC), TR_CLOCK_TICK);
1331        if (!(cmd->cmd_flags & CFLAG_FINISHED)) {
1332                mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1333                if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
1334                        mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1335                }
1336                rvalue = -1;
1337        }
1338        mptsas_remove_cmd(mpt, cmd);
1339        mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);

1341        return (rvalue);
1342 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
   22061 Tue Dec  4 12:30:16 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_raid.c
XXXX Nexenta fixes for mpt_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) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (c) 2000 to 2010, LSI Corporation.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms of all code within
 * this file that is exclusively owned by LSI, with or without
 * modification, is permitted provided that, in addition to the CDDL 1.0
 * License requirements, the following conditions are met:
 *
 *    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.
 */

/*
 * mptsas_raid - This file contains all the RAID related functions for the
 * MPT interface.
 */

#if defined(lint) || defined(DEBUG)
#define MPTSAS_DEBUG
#endif
```



```
#define MPI_RAID_VOL_PAGE_0_PHYSDISK_MAX        2

/*
 * standard header files
 */
#include <sys/note.h>
#include <sys/scsi/scsi.h>
#include <sys/byteorder.h>
#include <sys/raidioctl.h>

#pragma pack(1)

#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_type.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_init.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_ioc.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_raid.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>

#pragma pack()

/*
 * private header files.
 */
#include <sys/scsi/adapters/mpt_sas/mptsas_var.h>

static int mptsas_get_raid_wwid(mptsas_t *mpt, mptsas_raidvol_t *raidvol);

extern int mptsas_check_dma_handle(ddi_dma_handle_t handle);
extern int mptsas_check_acc_handle(ddi_acc_handle_t handle);
extern mptsas_target_t *mptsas_tgt_alloc(mptsas_hash_table_t *, uint16_t,
    uint64_t, uint32_t, mptsas_phymask_t, uint8_t);
    uint64_t, uint32_t, mptsas_phymask_t, uint8_t, mptsas_t *);

static int
mptsas_raidconf_page_0_cb(mptsas_t *mpt, caddr_t page_memp,
    ddi_acc_handle_t accessp, uint16_t iocstatus, uint32_t iocloginfo,
    va_list ap)
{
#ifndef __lock_lint
        _NOTE(ARGUNUSED(ap))
#endif
        pMpi2RaidConfigurationPage0_t   raidconfig_page0;
        pMpi2RaidConfig0ConfigElement_t element;
        uint32_t *configgnum;
        int rval = DDI_SUCCESS, i;
        uint8_t numelements, vol, disk;
        uint16_t elementtype, voldevhandle;
        uint16_t etype_vol, etype_pd, etype_hs;
        uint16_t etype_oce;
        mptsas_slots_t *slots = mpt->m_active;
        m_raidconfig_t *raidconfig;
        uint64_t raidwwn;
        uint32_t native;
        mptsas_target_t *ptgt;
        uint32_t configindex;

        if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
                return (DDI_FAILURE);
        }

        if (iocstatus != MPI2_IOCSTATUS_SUCCESS) {
                mptsas_log(mpt, CE_WARN, "mptsas_get_raid_conf_page0 "
                    "config: IOCStatus=0x%x, IOCLogInfo=0x%x",
                    iocstatus, iocloginfo);
```

```
127                 rval = DDI_FAILURE;
128                 return (rval);
129         }
130         configNum = va_arg(ap,  uint32_t *);
131         configindex = va_arg(ap, uint32_t);
132         raidconfig_page0 = (pMpi2RaidConfigurationPage0_t)page_memp;
133         /*
134          * Get all RAID configurations.
135          */
136         etype_vol = MPI2_RAIDCONFIG0_EFLAGS_VOLUME_ELEMENT;
137         etype_pd = MPI2_RAIDCONFIG0_EFLAGS_VOL_PHYS_DISK_ELEMENT;
138         etype_hs = MPI2_RAIDCONFIG0_EFLAGS_HOT_SPARE_ELEMENT;
139         etype_oce = MPI2_RAIDCONFIG0_EFLAGS_OCE_ELEMENT;
140         /*
141          * Set up page address for next time through.
142          */
143         *configNum =  ddi_get8(accessp,
144             &raidconfig_page0->ConfigNum);

146         /*
147          * Point to the right config in the structure.
148          * Increment the number of valid RAID configs.
149          */
150         raidconfig = &slots->m_raidconfig[configindex];
151         slots->m_num_raid_configs++;

153         /*
154          * Set the native flag if this is not a foreign
155          * configuration.
156          */
157         native = ddi_get32(accessp, &raidconfig_page0->Flags);
158         if (native & MPI2_RAIDCONFIG0_FLAG_FOREIGN_CONFIG) {
159                 native = FALSE;
160         } else {
161                 native = TRUE;
162         }
163         raidconfig->m_native = (uint8_t)native;

165         /*
166          * Get volume information for the volumes in the
167          * config.
168          */
169         numelements = ddi_get8(accessp, &raidconfig_page0->NumElements);
170         vol = 0;
171         disk = 0;
172         element = (pMpi2RaidConfig0ConfigElement_t)
173             &raidconfig_page0->ConfigElement;

175         for (i = 0; ((i < numelements) && native); i++, element++) {
176                 /*
177                  * Get the element type.  Could be Volume,
178                  * PhysDisk, Hot Spare, or Online Capacity
179                  * Expansion PhysDisk.
180                  */
181                 elementtype = ddi_get16(accessp, &element->ElementFlags);
182                 elementtype &= MPI2_RAIDCONFIG0_EFLAGS_MASK_ELEMENT_TYPE;

184                 /*
185                  * For volumes, get the RAID settings and the
186                  * WWID.
187                  */
188                 if (elementtype == etype_vol) {
189                         voldevhandle = ddi_get16(accessp,
190                             &element->VolDevHandle);
191                         raidconfig->m_raidvol[vol].m_israid = 1;
192                         raidconfig->m_raidvol[vol].
```



```
193                             m_raidhandle = voldevhandle;
194                         /*
195                          * Get the settings for the raid
196                          * volume.  This includes the
197                          * DevHandles for the disks making up
198                          * the raid volume.
199                          */
200                         if (mptsas_get_raid_settings(mpt,
201                             &raidconfig->m_raidvol[vol]))
202                                 continue;

204                         /*
205                          * Get the WWID of the RAID volume for
206                          * SAS HBA
207                          */
208                         if (mptsas_get_raid_wwid(mpt,
209                             &raidconfig->m_raidvol[vol]))
210                                 continue;

212                         raidwwn = raidconfig->m_raidvol[vol].
213                             m_raidwwid;

215                         /*
216                          * RAID uses phymask of 0.
217                          */
218                         ptgt = mptsas_tgt_alloc(&slots->m_tgttbl,
219                             voldevhandle, raidwwn, 0, 0, 0);
219                             voldevhandle, raidwwn, 0, 0, 0, mpt);

221                         raidconfig->m_raidvol[vol].m_raidtgt =
222                             ptgt;

224                         /*
225                          * Increment volume index within this
226                          * raid config.
227                          */
228                         vol++;
229                 } else if ((elementtype == etype_pd) ||
230                     (elementtype == etype_hs) ||
231                     (elementtype == etype_oce)) {
232                         /*
233                          * For all other element types, put
234                          * their DevHandles in the phys disk
235                          * list of the config.  These are all
236                          * some variation of a Phys Disk and
237                          * this list is used to keep these
238                          * disks from going online.
239                          */
240                         raidconfig->m_physdisk_devhdl[disk] = ddi_get16(accessp,
241                             &element->PhysDiskDevHandle);

243                         /*
244                          * Increment disk index within this
245                          * raid config.
246                          */
247                         disk++;
248                 }
249         }

251         return (rval);
252 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   43458 Tue Dec  4 12:30:16 2012
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
XXXX Nexenta fixes for mpt_sas(7d)
**********************************************************
______unchanged_portion_omitted_

 156 /*
 157  * preferred pkt_private length in 64-bit quantities
 158  */
 159 #ifdef  _LP64
 160 #define PKT_PRIV_SIZE   2
 161 #define PKT_PRIV_LEN    16      /* in bytes */
 162 #else /* _ILP32 */
 163 #define PKT_PRIV_SIZE   1
 164 #define PKT_PRIV_LEN    8       /* in bytes */
 165 #endif

 167 #define PKT2CMD(pkt)    ((struct mptsas_cmd *)((pkt)->pkt_ha_private))
 168 #define CMD2PKT(cmdp)   ((struct scsi_pkt *)((cmdp)->cmd_pkt))
 169 #define EXTCMDS_STATUS_SIZE (sizeof (struct scsi_arq_status))

 171 /*
 172  * get offset of item in structure
 173  */
 174 #define MPTSAS_GET_ITEM_OFF(type, member) ((size_t)(&((type *)0)->member))

 176 /*
 177  * WWID provided by LSI firmware is generated by firmware but the WWID is not
 178  * IEEE NAA standard format, OBP has no chance to distinguish format of unit
 179  * address. According LSI's confirmation, the top nibble of RAID WWID is
 180  * meanless, so the consensus between Solaris and OBP is to replace top nibble
 181  * of WWID provided by LSI to "3" always to hint OBP that this is a RAID WWID
 182  * format unit address.
 183  */
 184 #define MPTSAS_RAID_WWID(wwid) \
 185         ((wwid & 0x0FFFFFFFFFFFFFFF) | 0x3000000000000000)

 187 typedef struct mptsas_target {
 188                 uint64_t                m_sas_wwn;      /* hash key1 */
 189                 mptsas_phymask_t        m_phymask;      /* hash key2 */
 190                 /*
 191                  * m_dr_flag is a flag for DR, make sure the member
 192                  * take the place of dr_flag of mptsas_hash_data.
 193                  */
 194                 uint8_t                 m_dr_flag;      /* dr_flag */
 195                 uint16_t                m_devhdl;
 196                 uint32_t                m_deviceinfo;
 197                 uint8_t                 m_phynum;
 198                 uint32_t                m_dups;
 199                 int32_t                 m_timeout;
 200                 int32_t                 m_timebase;
 201                 int32_t                 m_t_throttle;
 202                 int32_t                 m_t_ncmds;
 203                 int32_t                 m_reset_delay;
 204                 int32_t                 m_t_nwait;

 206                 uint16_t                m_qfull_retry_interval;
 207                 uint8_t                 m_qfull_retries;
 208                 uint16_t                m_enclosure;
 209                 uint16_t                m_slot_num;
 210                 uint32_t                m_tgt_unconfigured;
 211                 uint32_t                m_timeout_interval;
 212                 uint8_t                 m_timeout_count;

 212                 /*
```



```
 213                  * For the common case, the elements in this structure are
 214                  * protected by the per hba instance mutex. In order to make
 215                  * the key code path in ISR lockless, a separate mutex is
 216                  * introdeced to protect those shown in ISR.
 217                  */
 218                 kmutex_t                m_tgt_intr_mutex;

 214 } mptsas_target_t;
______unchanged_portion_omitted_

 576 /*
 577  * The following defines are used in mptsas_set_init_mode to track the current
 578  * state as we progress through reprogramming the HBA from target mode into
 579  * initiator mode.
 580  */

 582 #define IOUC_READ_PAGE0         0x00000100
 583 #define IOUC_READ_PAGE1         0x00000200
 584 #define IOUC_WRITE_PAGE1        0x00000400
 585 #define IOUC_DONE               0x00000800
 586 #define DISCOVERY_IN_PROGRESS   MPI2_SASIOUNIT0_PORTFLAGS_DISCOVERY_IN_PROGRESS
 587 #define AUTO_PORT_CONFIGURATION MPI2_SASIOUNIT0_PORTFLAGS_AUTO_PORT_CONFIG

 589 /*
 590  * Last allocated slot is used for TM requests.  Since only m_max_requests
 591  * frames are allocated, the last SMID will be m_max_requests - 1.
 592  */
 593 #define MPTSAS_SLOTS_SIZE(mpt) \
 594         (sizeof (struct mptsas_slots) + (sizeof (struct mptsas_cmd *) * \
 595                 mpt->m_max_requests))
 596 #define MPTSAS_TM_SLOT(mpt)     (mpt->m_max_requests - 1)

 604 typedef struct mptsas_slot_free_e {
 605         processorid_t           cpuid;
 606         int                     slot;
 607         list_node_t             node;
 608 } mptsas_slot_free_e_t;

 598 /*
 611  * each of the allocq and releaseq in all CPU groups resides in separate
 612  * cacheline(64 bytes). Multiple mutex in the same cacheline is not good
 613  * for performance.
 614  */
 615 typedef union mptsas_slot_freeq {
 616         struct {
 617                 kmutex_t        m_fq_mutex;
 618                 list_t          m_fq_list;
 619                 int             m_fq_n;
 620                 int             m_fq_n_init;
 621         } s;
 622         char pad[64];
 623 } mptsas_slot_freeq_t;

 625 typedef struct mptsas_slot_freeq_pair {
 626         mptsas_slot_freeq_t     m_slot_allocq;
 627         mptsas_slot_freeq_t     m_slot_releq;
 628 } mptsas_slot_freeq_pair_t;

 630 /*
 599  * Macro for phy_flags
 600  */

 602 typedef struct smhba_info {
 603         kmutex_t        phy_mutex;
 604         uint8_t         phy_id;
 605         uint64_t        sas_addr;
```

```
606         char            path[8];
607         uint16_t        owner_devhdl;
608         uint16_t        attached_devhdl;
609         uint8_t         attached_phy_identify;
610         uint32_t        attached_phy_info;
611         uint8_t         programmed_link_rate;
612         uint8_t         hw_link_rate;
613         uint8_t         change_count;
614         uint32_t        phy_info;
615         uint8_t         negotiated_link_rate;
616         uint8_t         port_num;
617         kstat_t         *phy_stats;
618         uint32_t        invalid_dword_count;
619         uint32_t        running_disparity_error_count;
620         uint32_t        loss_of_dword_sync_count;
621         uint32_t        phy_reset_problem_count;
622         void            *mpt;
623 } smhba_info_t;
______unchanged_portion_omitted_

655 typedef struct mptsas {
656         int             m_instance;

658         struct mptsas *m_next;

660         scsi_hba_tran_t         *m_tran;
661         smp_hba_tran_t          *m_smptran;
662         kmutex_t                m_mutex;
663         kmutex_t                m_passthru_mutex;
664         kcondvar_t              m_cv;
665         kcondvar_t              m_passthru_cv;
666         kcondvar_t              m_fw_cv;
667         kcondvar_t              m_config_cv;
668         kcondvar_t              m_fw_diag_cv;
669         dev_info_t              *m_dip;

671         /*
672          * soft state flags
673          */
674         uint_t          m_softstate;

676         struct mptsas_slots *m_active;  /* outstanding cmds */

678         mptsas_cmd_t    *m_waitq;       /* cmd queue for active request */
679         mptsas_cmd_t    **m_waitqtail;  /* wait queue tail ptr */

681         kmutex_t        m_tx_waitq_mutex;
682         mptsas_cmd_t    *m_tx_waitq;    /* TX cmd queue for active request */
683         mptsas_cmd_t    **m_tx_waitqtail;       /* tx_wait queue tail ptr */
684         int             m_tx_draining;  /* TX queue draining flag */

686         mptsas_cmd_t    *m_doneq;       /* queue of completed commands */
687         mptsas_cmd_t    **m_donetail;   /* queue tail ptr */

714         kmutex_t                m_passthru_mutex;
715         kcondvar_t              m_passthru_cv;
689         /*
690          * variables for helper threads (fan-out interrupts)
691          */
692         mptsas_doneq_thread_list_t      *m_doneq_thread_id;
693         uint32_t                m_doneq_thread_n;
694         uint32_t                m_doneq_thread_threshold;
695         uint32_t                m_doneq_length_threshold;
696         uint32_t                m_doneq_len;
697         kcondvar_t              m_doneq_thread_cv;
698         kmutex_t                m_doneq_mutex;
```



```
700         int             m_ncmds;        /* number of outstanding commands */
701         m_event_struct_t *m_ioc_event_cmdq;     /* cmd queue for ioc event */
702         m_event_struct_t **m_ioc_event_cmdtail; /* ioc cmd queue tail */

704         ddi_acc_handle_t m_datap;       /* operating regs data access handle */

706         struct _MPI2_SYSTEM_INTERFACE_REGS      *m_reg;

708         ushort_t        m_devid;        /* device id of chip. */
709         uchar_t         m_revid;        /* revision of chip. */
710         uint16_t        m_svid;         /* subsystem Vendor ID of chip */
711         uint16_t        m_ssid;         /* subsystem Device ID of chip */

713         uchar_t         m_sync_offset;  /* default offset for this chip. */

715         timeout_id_t    m_quiesce_timeid;

717         ddi_dma_handle_t m_dma_req_frame_hdl;
718         ddi_acc_handle_t m_acc_req_frame_hdl;
719         ddi_dma_handle_t m_dma_reply_frame_hdl;
720         ddi_acc_handle_t m_acc_reply_frame_hdl;
721         ddi_dma_handle_t m_dma_free_queue_hdl;
722         ddi_acc_handle_t m_acc_free_queue_hdl;
723         ddi_dma_handle_t m_dma_post_queue_hdl;
724         ddi_acc_handle_t m_acc_post_queue_hdl;

726         /*
754          * Try the best to make the key code path in the ISR lockless.
755          * so avoid to use the per instance mutex m_mutex in the ISR. Introduce
756          * a separate mutex to protect the elements shown in ISR.
757          */
758         kmutex_t        m_intr_mutex;

760         /*
727          * list of reset notification requests
728          */
729         struct scsi_reset_notify_entry  *m_reset_notify_listf;

731         /*
732          * qfull handling
733          */
734         timeout_id_t    m_restart_cmd_timeid;

736         /*
737          * scsi reset delay per bus
738          */
739         uint_t          m_scsi_reset_delay;

741         int             m_pm_idle_delay;

743         uchar_t         m_polled_intr;  /* intr was polled. */
744         uchar_t         m_suspended;    /* true if driver is suspended */

746         struct kmem_cache *m_kmem_cache;
747         struct kmem_cache *m_cache_frames;

749         /*
750          * hba options.
751          */
752         uint_t          m_options;

754         int             m_in_callback;

756         int             m_power_level;  /* current power level */
```

```
758         int             m_busy;         /* power management busy state */

760         off_t           m_pmcsr_offset; /* PMCSR offset */

762         ddi_acc_handle_t m_config_handle;

764         ddi_dma_attr_t          m_io_dma_attr;  /* Used for data I/O */
765         ddi_dma_attr_t          m_msg_dma_attr; /* Used for message frames */
766         ddi_device_acc_attr_t   m_dev_acc_attr;
767         ddi_device_acc_attr_t   m_reg_acc_attr;

769         /*
770          * request/reply variables
771          */
772         caddr_t         m_req_frame;
773         uint64_t        m_req_frame_dma_addr;
774         caddr_t         m_reply_frame;
775         uint64_t        m_reply_frame_dma_addr;
776         caddr_t         m_free_queue;
777         uint64_t        m_free_queue_dma_addr;
778         caddr_t         m_post_queue;
779         uint64_t        m_post_queue_dma_addr;

781         m_replyh_arg_t *m_replyh_args;

783         uint16_t        m_max_requests;
784         uint16_t        m_req_frame_size;

786         /*
787          * Max frames per request reprted in IOC Facts
788          */
789         uint8_t         m_max_chain_depth;
790         /*
791          * Max frames per request which is used in reality. It's adjusted
792          * according DMA SG length attribute, and shall not exceed the
793          * m_max_chain_depth.
794          */
795         uint8_t         m_max_request_frames;

797         uint16_t        m_free_queue_depth;
798         uint16_t        m_post_queue_depth;
799         uint16_t        m_max_replies;
800         uint32_t        m_free_index;
801         uint32_t        m_post_index;
802         uint8_t         m_reply_frame_size;
803         uint32_t        m_ioc_capabilities;

805         /*
806          * indicates if the firmware was upload by the driver
807          * at boot time
808          */
809         ushort_t        m_fwupload;

811         uint16_t        m_productid;

813         /*
814          * per instance data structures for dma memory resources for
815          * MPI handshake protocol. only one handshake cmd can run at a time.
816          */
817         ddi_dma_handle_t        m_hshk_dma_hdl;

818         ddi_acc_handle_t        m_hshk_acc_hdl;

819         caddr_t                 m_hshk_memp;

820         size_t                  m_hshk_dma_size;
```



```
822         /* Firmware version on the card at boot time */
823         uint32_t                m_fwversion;

825         /* MSI specific fields */
826         ddi_intr_handle_t       *m_htable;      /* For array of interrupts */
827         int                     m_intr_type;    /* What type of interrupt */
828         int                     m_intr_cnt;     /* # of intrs count returned */
829         size_t                  m_intr_size;    /* Size of intr array */
830         uint_t                  m_intr_pri;     /* Interrupt priority   */
831         int                     m_intr_cap;     /* Interrupt capabilities */
832         ddi_taskq_t             *m_event_taskq;

834         /* SAS specific information */

836         union {
837                 uint64_t        m_base_wwid;    /* Base WWID */
838                 struct {
839 #ifdef _BIG_ENDIAN
840                         uint32_t        m_base_wwid_hi;
841                         uint32_t        m_base_wwid_lo;
842 #else
843                         uint32_t        m_base_wwid_lo;
844                         uint32_t        m_base_wwid_hi;
845 #endif
846                 } sasaddr;
847         } un;

849         uint8_t                 m_num_phys;             /* # of PHYs */
850         mptsas_phy_info_t       m_phy_info[MPTSAS_MAX_PHYS];
851         uint8_t                 m_port_chng;    /* initiator port changes */
852         MPI2_CONFIG_PAGE_MAN_0   m_MANU_page0;   /* Manufactor page 0 info */
853         MPI2_CONFIG_PAGE_MAN_1   m_MANU_page1;   /* Manufactor page 1 info */

855         /* FMA Capabilities */
856         int                     m_fm_capabilities;
857         ddi_taskq_t             *m_dr_taskq;
858         int                     m_mpxio_enable;
859         uint8_t                 m_done_traverse_dev;
860         uint8_t                 m_done_traverse_smp;
861         int                     m_diag_action_in_progress;
862         uint16_t                m_dev_handle;
863         uint16_t                m_smp_devhdl;

865         /*
866          * Event recording
867          */
868         uint8_t                 m_event_index;
869         uint32_t                m_event_number;
870         uint32_t                m_event_mask[4];
871         mptsas_event_entry_t    m_events[MPTSAS_EVENT_QUEUE_SIZE];

873         /*
874          * FW diag Buffer List
875          */
876         mptsas_fw_diagnostic_buffer_t
877                 m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_COUNT];

879         /*
880          * Event Replay flag (MUR support)
881          */
882         uint8_t                 m_event_replay;

884         /*
885          * IR Capable flag
886          */
```

```
 887         uint8_t                 m_ir_capable;

 889         /*
 927          * release and alloc queue for slot
 928          */
 929         int                     m_slot_freeq_pair_n;
 930         mptsas_slot_freeq_pair_t        *m_slot_freeq_pairp;
 931         mptsas_slot_free_e_t    *m_slot_free_ae;
 932 #define MPI_ADDRESS_COALSCE_MAX 128
 933         pMpi2ReplyDescriptorsUnion_t    m_reply;

 935         /*
 890          * Is HBA processing a diag reset?
 891          */
 892         uint8_t                 m_in_reset;

 894         /*
 895          * per instance cmd data structures for task management cmds
 896          */
 897         m_event_struct_t        m_event_task_mgmt;      /* must be last */
 898                                                         /* ... scsi_pkt_size */
 899 } mptsas_t;
```

_____***unchanged_portion_omitted_***