```
**********************************************************
  416643 Tue Dec  4 16:29:49 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
re #6530 mpt_sas crash when more than 1 Initiator involved - ie HA
**********************************************************
```

```
/*
 * 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);

241 static void mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd);
```

```
242 static int mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
243     uint8_t *data, uint32_t request_size, uint32_t reply_size,
244     uint32_t data_size, uint32_t direction, uint8_t *dataout,
245     uint32_t dataout_size, short timeout, int mode);
246 static int mptsas_free_devhdl(mptsas_t *mpt, uint16_t devhdl);

248 static uint8_t mptsas_get_fw_diag_buffer_number(mptsas_t *mpt,
249     uint32_t unique_id);
250 static void mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd);
251 static int mptsas_post_fw_diag_buffer(mptsas_t *mpt,
252     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code);
253 static int mptsas_release_fw_diag_buffer(mptsas_t *mpt,
254     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
255     uint32_t diag_type);
256 static int mptsas_diag_register(mptsas_t *mpt,
257     mptsas_fw_diag_register_t *diag_register, uint32_t *return_code);
258 static int mptsas_diag_unregister(mptsas_t *mpt,
259     mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code);
260 static int mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
261     uint32_t *return_code);
262 static int mptsas_diag_read_buffer(mptsas_t *mpt,
263     mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
264     uint32_t *return_code, int ioctl_mode);
265 static int mptsas_diag_release(mptsas_t *mpt,
266     mptsas_fw_diag_release_t *diag_release, uint32_t *return_code);
267 static int mptsas_do_diag_action(mptsas_t *mpt, uint32_t action,
268     uint8_t *diag_action, uint32_t length, uint32_t *return_code,
269     int ioctl_mode);
270 static int mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *data,
271     int mode);

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

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

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

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

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

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

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

303 static mptsas_cmd_t *mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t);
304 static void mptsas_doneq_empty(mptsas_t *mpt);
305 static void mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg);
```



```
307 static mptsas_cmd_t *mptsas_waitq_rm(mptsas_t *mpt);
308 static void mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);
309 static mptsas_cmd_t *mptsas_tx_waitq_rm(mptsas_t *mpt);
310 static void mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);


313 static void mptsas_start_watch_reset_delay();
314 static void mptsas_setup_bus_reset_delay(mptsas_t *mpt);
315 static void mptsas_watch_reset_delay(void *arg);
316 static int mptsas_watch_reset_delay_subr(mptsas_t *mpt);

332 static int mptsas_outstanding_cmds_n(mptsas_t *mpt);
318 /*
319  * helper functions
320  */
321 static void mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);

323 static dev_info_t *mptsas_find_child(dev_info_t *pdip, char *name);
324 static dev_info_t *mptsas_find_child_phy(dev_info_t *pdip, uint8_t phy);
325 static dev_info_t *mptsas_find_child_addr(dev_info_t *pdip, uint64_t sasaddr,
326     int lun);
327 static mdi_pathinfo_t *mptsas_find_path_addr(dev_info_t *pdip, uint64_t sasaddr,
328     int lun);
329 static mdi_pathinfo_t *mptsas_find_path_phy(dev_info_t *pdip, uint8_t phy);
330 static dev_info_t *mptsas_find_smp_child(dev_info_t *pdip, char *str_wwn);

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

336 static mptsas_target_t *mptsas_phy_to_tgt(mptsas_t *mpt, int phymask,
337     uint8_t phy);
338 static mptsas_target_t *mptsas_wwid_to_ptgt(mptsas_t *mpt, int phymask,
339     uint64_t wwid);
340 static mptsas_smp_t *mptsas_wwid_to_psmp(mptsas_t *mpt, int phymask,
341     uint64_t wwid);

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

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

350 static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
351     uint32_t *status, uint8_t cmd);
352 static dev_info_t *mptsas_get_dip_from_dev(dev_t dev,
353     mptsas_phymask_t *phymask);
354 static mptsas_target_t *mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr,
355     mptsas_phymask_t phymask);
356 static int mptsas_set_led_status(mptsas_t *mpt, mptsas_target_t *ptgt,
357     uint32_t slotstatus);


360 /*
361  * Enumeration / DR functions
362  */
363 static void mptsas_config_all(dev_info_t *pdip);
364 static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
365     dev_info_t **lundip);
366 static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
367     dev_info_t **lundip);

369 static int mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt);
```

```
370 static int mptsas_offline_target(dev_info_t *pdip, char *name);

372 static int mptsas_config_raid(dev_info_t *pdip, uint16_t target,
373     dev_info_t **dip);

375 static int mptsas_config_luns(dev_info_t *pdip, mptsas_target_t *ptgt);
376 static int mptsas_probe_lun(dev_info_t *pdip, int lun,
377     dev_info_t **dip, mptsas_target_t *ptgt);

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

382 static int mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
383     char *guid, dev_info_t **dip, mptsas_target_t *ptgt, int lun);
384 static int mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
385     char *guid, dev_info_t **dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt,
386     int lun);

388 static void mptsas_offline_missed_luns(dev_info_t *pdip,
389     uint16_t *repluns, int lun_cnt, mptsas_target_t *ptgt);
390 static int mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,
391     mdi_pathinfo_t *rpip, uint_t flags);

393 static int mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn,
394     dev_info_t **smp_dip);
395 static int mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
396     uint_t flags);

398 static int mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data,
399     int mode, int *rval);
400 static int mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data,
401     int mode, int *rval);
402 static int mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data,
403     int mode, int *rval);
404 static void mptsas_record_event(void *args);
405 static int mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data,
406     int mode);

408 static void mptsas_hash_init(mptsas_hash_table_t *hashtab);
409 static void mptsas_hash_uninit(mptsas_hash_table_t *hashtab, size_t datalen);
410 static void mptsas_hash_add(mptsas_hash_table_t *hashtab, void *data);
411 static void * mptsas_hash_rem(mptsas_hash_table_t *hashtab, uint64_t key1,
412     mptsas_phymask_t key2);
413 static void * mptsas_hash_search(mptsas_hash_table_t *hashtab, uint64_t key1,
414     mptsas_phymask_t key2);
415 static void * mptsas_hash_traverse(mptsas_hash_table_t *hashtab, int pos);

417 mptsas_target_t *mptsas_tgt_alloc(mptsas_hash_table_t *, uint16_t, uint64_t,
418     uint32_t, mptsas_phymask_t, uint8_t);
434     uint32_t, mptsas_phymask_t, uint8_t, mptsas_t *);
419 static mptsas_smp_t *mptsas_smp_alloc(mptsas_hash_table_t *hashtab,
420     mptsas_smp_t *data);
421 static void mptsas_smp_free(mptsas_hash_table_t *hashtab, uint64_t wwid,
422     mptsas_phymask_t phymask);
423 static void mptsas_tgt_free(mptsas_hash_table_t *, uint64_t, mptsas_phymask_t);
424 static void * mptsas_search_by_devhdl(mptsas_hash_table_t *, uint16_t);
425 static int mptsas_online_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
426     dev_info_t **smp_dip);

428 /*
429  * Power management functions
430  */
431 static int mptsas_get_pci_cap(mptsas_t *mpt);
432 static int mptsas_init_pm(mptsas_t *mpt);

434 /*
```



```
435  * MPT MSI tunable:
436  *
437  * By default MSI is enabled on all supported platforms.
438  */
439 boolean_t mptsas_enable_msi = B_TRUE;
440 boolean_t mptsas_physical_bind_failed_page_83 = B_FALSE;

442 static int mptsas_register_intrs(mptsas_t *);
443 static void mptsas_unregister_intrs(mptsas_t *);
444 static int mptsas_add_intrs(mptsas_t *, int);
445 static void mptsas_rem_intrs(mptsas_t *);

447 /*
448  * FMA Prototypes
449  */
450 static void mptsas_fm_init(mptsas_t *mpt);
451 static void mptsas_fm_fini(mptsas_t *mpt);
452 static int mptsas_fm_error_cb(dev_info_t *, ddi_fm_error_t *, const void *);

454 extern pri_t minclsyspri, maxclsyspri;

456 /*
457  * This device is created by the SCSI pseudo nexus driver (SCSI vHCI).  It is
458  * under this device that the paths to a physical device are created when
459  * MPxIO is used.
460  */
461 extern dev_info_t       *scsi_vhci_dip;

463 /*
464  * Tunable timeout value for Inquiry VPD page 0x83
465  * By default the value is 30 seconds.
466  */
467 int mptsas_inq83_retry_timeout = 30;

469 /*
470  * This is used to allocate memory for message frame storage, not for
471  * data I/O DMA. All message frames must be stored in the first 4G of
472  * physical memory.
473  */
474 ddi_dma_attr_t mptsas_dma_attrs = {
475         DMA_ATTR_V0,    /* attribute layout version             */
476         0x0ull,         /* address low - should be 0 (longlong) */
477         0xffffffffull,  /* address high - 32-bit max range      */
478         0x00ffffffull,  /* count max - max DMA object size      */
479         4,              /* allocation alignment requirements    */
480         0x78,           /* burstsizes - binary encoded values   */
481         1,              /* minxfer - gran. of DMA engine        */
482         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
483         0xffffffffull,  /* max segment size (DMA boundary)      */
484         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
485         512,            /* granularity - device transfer size   */
486         0               /* flags, set to 0                      */
487 };
______unchanged_portion_omitted_

926 /*
927  * Notes:
928  *      Set up all device state and allocate data structures,
929  *      mutexes, condition variables, etc. for device operation.
930  *      Add interrupts needed.
931  *      Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
932  */
933 static int
934 mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
935 {
936         mptsas_t                *mpt = NULL;
```

```
        int                     instance, i, j;
        int                     doneq_thread_num;
        char                    intr_added = 0;
        char                    map_setup = 0;
        char                    config_setup = 0;
        char                    hba_attach_setup = 0;
        char                    smp_attach_setup = 0;
        char                    mutex_init_done = 0;
        char                    event_taskq_create = 0;
        char                    dr_taskq_create = 0;
        char                    doneq_thread_create = 0;
        scsi_hba_tran_t         *hba_tran;
        uint_t                  mem_bar = MEM_SPACE;
        int                     rval = DDI_FAILURE;

        /* CONSTCOND */
        ASSERT(NO_COMPETING_THREADS);

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

        switch (cmd) {
        case DDI_ATTACH:
                break;

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

                mpt = TRAN2MPT(hba_tran);

                if (!mpt) {
                        return (DDI_FAILURE);
                }

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

                /*
                 * raise power.
                 */
                if (mpt->m_options & MPTSAS_OPT_PM) {
                        mutex_exit(&mpt->m_mutex);
                        (void) pm_busy_component(dip, 0);
                        rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
                        if (rval == DDI_SUCCESS) {
                                mutex_enter(&mpt->m_mutex);
                        } else {
                                /*
                                 * The pm_raise_power() call above failed,
                                 * and that can only occur if we were unable
                                 * to reset the hardware.  This is probably
                                 * due to unhealty hardware, and because
                                 * important filesystems(such as the root
                                 * filesystem) could be on the attached disks,
                                 * it would not be a good idea to continue,
                                 * as we won't be entirely certain we are
                                 * writing correct data.  So we panic() here
                                 * to not only prevent possible data corruption,
                                 * but to give developers or end users a hope
                                 * of identifying and correcting any problems.
```



```
                                 */
                                fm_panic("mptsas could not reset hardware "
                                    "during resume");
                        }
                }

                mpt->m_suspended = 0;

                /*
                 * Reinitialize ioc
                 */
                mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
                if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
                        mutex_exit(&mpt->m_mutex);
                        if (mpt->m_options & MPTSAS_OPT_PM) {
                                (void) pm_idle_component(dip, 0);
                        }
                        fm_panic("mptsas init chip fail during resume");
                }
                /*
                 * mptsas_update_driver_data needs interrupts so enable them
                 * first.
                 */
                MPTSAS_ENABLE_INTR(mpt);
                mptsas_update_driver_data(mpt);

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

                mutex_exit(&mpt->m_mutex);

                /*
                 * Restart watch thread
                 */
                mutex_enter(&mptsas_global_mutex);
                if (mptsas_timeout_id == 0) {
                        mptsas_timeout_id = timeout(mptsas_watch, NULL,
                            mptsas_tick);
                        mptsas_timeouts_enabled = 1;
                }
                mutex_exit(&mptsas_global_mutex);

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

                return (DDI_SUCCESS);

        default:
                return (DDI_FAILURE);

        }

        instance = ddi_get_instance(dip);

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

        mpt = ddi_get_soft_state(mptsas_state, instance);
```

```
1070        if (mpt == NULL) {
1071                mptsas_log(NULL, CE_WARN,
1072                    "mptsas%d: cannot get soft state", instance);
1073                goto fail;
1074        }

1076        /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
1077        scsi_size_clean(dip);

1079        mpt->m_dip = dip;
1080        mpt->m_instance = instance;

1082        /* Make a per-instance copy of the structures */
1083        mpt->m_io_dma_attr = mptsas_dma_attrs64;
1084        mpt->m_msg_dma_attr = mptsas_dma_attrs;
1085        mpt->m_reg_acc_attr = mptsas_dev_attr;
1086        mpt->m_dev_acc_attr = mptsas_dev_attr;

1088        /*
1089         * Initialize FMA
1090         */
1091        mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
1092            DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
1093            DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
1094            DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);

1096        mptsas_fm_init(mpt);

1098        if (mptsas_alloc_handshake_msg(mpt,
1099            sizeof (Mpi2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
1100                mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
1101                goto fail;
1102        }

1104        /*
1105         * Setup configuration space
1106         */
1107        if (mptsas_config_space_init(mpt) == FALSE) {
1108                mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
1109                goto fail;
1110        }
1111        config_setup++;

1113        if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
1114            0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
1115                mptsas_log(mpt, CE_WARN, "map setup failed");
1116                goto fail;
1117        }
1118        map_setup++;

1120        /*
1121         * A taskq is created for dealing with the event handler
1122         */
1123        if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
1124            1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1125                mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
1126                goto fail;
1127        }
1128        event_taskq_create++;

1130        /*
1131         * A taskq is created for dealing with dr events
1132         */
1133        if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
1134            "mptsas_dr_taskq",
```



```
1135            1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1136                mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
1137                    "failed");
1138                goto fail;
1139        }
1140        dr_taskq_create++;

1142        mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1143            0, "mptsas_doneq_thread_threshold_prop", 10);
1144        mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1145            0, "mptsas_doneq_length_threshold_prop", 8);
1146        mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1147            0, "mptsas_doneq_thread_n_prop", 8);

1149        if (mpt->m_doneq_thread_n) {
1150                cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
1151                mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);

1153                mutex_enter(&mpt->m_doneq_mutex);
1154                mpt->m_doneq_thread_id =
1155                    kmem_zalloc(sizeof (mptsas_doneq_thread_list_t)
1156                    * mpt->m_doneq_thread_n, KM_SLEEP);

1158                for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1159                        cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
1160                            CV_DRIVER, NULL);
1161                        mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
1162                            MUTEX_DRIVER, NULL);
1163                        mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1164                        mpt->m_doneq_thread_id[j].flag |=
1165                            MPTSAS_DONEQ_THREAD_ACTIVE;
1166                        mpt->m_doneq_thread_id[j].arg.mpt = mpt;
1167                        mpt->m_doneq_thread_id[j].arg.t = j;
1168                        mpt->m_doneq_thread_id[j].threadp =
1169                            thread_create(NULL, 0, mptsas_doneq_thread,
1170                            &mpt->m_doneq_thread_id[j].arg,
1171                            0, &p0, TS_RUN, minclsyspri);
1172                        mpt->m_doneq_thread_id[j].donetail =
1173                            &mpt->m_doneq_thread_id[j].doneq;
1174                        mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1175                }
1176                mutex_exit(&mpt->m_doneq_mutex);
1177                doneq_thread_create++;
1178        }

1180        /* Initialize mutex used in interrupt handler */
1181        mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
1182            DDI_INTR_PRI(mpt->m_intr_pri));
1183        mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
1184        mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
1200        mutex_init(&mpt->m_intr_mutex, NULL, MUTEX_DRIVER,
1185            DDI_INTR_PRI(mpt->m_intr_pri));
1186        for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1187                mutex_init(&mpt->m_phy_info[i].smhba_info.phy_mutex,
1188                    NULL, MUTEX_DRIVER,
1189                    DDI_INTR_PRI(mpt->m_intr_pri));
1190        }

1192        cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
1193        cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
1194        cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
1195        cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
1196        cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
1197        mutex_init_done++;

1199        /*
```

```
1200          * Disable hardware interrupt since we're not ready to
1201          * handle it yet.
1202          */
1203         MPTSAS_DISABLE_INTR(mpt);
1204         if (mptsas_register_intrs(mpt) == FALSE)
1205                 goto fail;
1206         intr_added++;

1208         mutex_enter(&mpt->m_mutex);
1209         /*
1210          * Initialize power management component
1211          */
1212         if (mpt->m_options & MPTSAS_OPT_PM) {
1213                 if (mptsas_init_pm(mpt)) {
1214                         mutex_exit(&mpt->m_mutex);
1215                         mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
1216                             "failed");
1217                         goto fail;
1218                 }
1219         }

1221         /*
1222          * Initialize chip using Message Unit Reset, if allowed
1223          */
1224         mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1225         if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
1226                 mutex_exit(&mpt->m_mutex);
1227                 mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
1228                 goto fail;
1229         }

1231         /*
1232          * Fill in the phy_info structure and get the base WWID
1233          */
1234         if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
1235                 mptsas_log(mpt, CE_WARN,
1236                     "mptsas_get_manufacture_page5 failed!");
1237                 goto fail;
1238         }

1240         if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {
1241                 mptsas_log(mpt, CE_WARN,
1242                     "mptsas_get_sas_io_unit_page_hndshk failed!");
1243                 goto fail;
1244         }

1246         if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
1247                 mptsas_log(mpt, CE_WARN,
1248                     "mptsas_get_manufacture_page0 failed!");
1249                 goto fail;
1250         }

1252         mutex_exit(&mpt->m_mutex);

1254         /*
1255          * Register the iport for multiple port HBA
1256          */
1257         mptsas_iport_register(mpt);

1259         /*
1260          * initialize SCSI HBA transport structure
1261          */
1262         if (mptsas_hba_setup(mpt) == FALSE)
1263                 goto fail;
1264         hba_attach_setup++;
```



```
1266         if (mptsas_smp_setup(mpt) == FALSE)
1267                 goto fail;
1268         smp_attach_setup++;

1270         if (mptsas_cache_create(mpt) == FALSE)
1271                 goto fail;

1273         mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
1274             dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
1275         if (mpt->m_scsi_reset_delay == 0) {
1276                 mptsas_log(mpt, CE_NOTE,
1277                     "scsi_reset_delay of 0 is not recommended,"
1278                     " resetting to SCSI_DEFAULT_RESET_DELAY\n");
1279                 mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
1280         }

1282         /*
1283          * Initialize the wait and done FIFO queue
1284          */
1285         mpt->m_donetail = &mpt->m_doneq;
1286         mpt->m_waitqtail = &mpt->m_waitq;
1287         mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
1288         mpt->m_tx_draining = 0;

1290         /*
1291          * ioc cmd queue initialize
1292          */
1293         mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
1294         mpt->m_dev_handle = 0xFFFF;

1296         MPTSAS_ENABLE_INTR(mpt);

1298         /*
1299          * enable event notification
1300          */
1301         mutex_enter(&mpt->m_mutex);
1302         if (mptsas_ioc_enable_event_notification(mpt)) {
1303                 mutex_exit(&mpt->m_mutex);
1304                 goto fail;
1305         }
1306         mutex_exit(&mpt->m_mutex);

1308         /*
1309          * Initialize PHY info for smhba
1310          */
1311         if (mptsas_smhba_setup(mpt)) {
1312                 mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
1313                     "failed");
1314                 goto fail;
1315         }

1317         /* Check all dma handles allocated in attach */
1318         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
1319             != DDI_SUCCESS) ||
1320             (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
1321             != DDI_SUCCESS) ||
1322             (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
1323             != DDI_SUCCESS) ||
1324             (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
1325             != DDI_SUCCESS) ||
1326             (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
1327             != DDI_SUCCESS)) {
1328                 goto fail;
1329         }

1331         /* Check all acc handles allocated in attach */
```

```
1332         if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
1333             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
1334             != DDI_SUCCESS) ||
1335             (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
1336             != DDI_SUCCESS) ||
1337             (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
1338             != DDI_SUCCESS) ||
1339             (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
1340             != DDI_SUCCESS) ||
1341             (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
1342             != DDI_SUCCESS) ||
1343             (mptsas_check_acc_handle(mpt->m_config_handle)
1344             != DDI_SUCCESS)) {
1345                 goto fail;
1346         }

1348         /*
1349          * After this point, we are not going to fail the attach.
1350          */
1351         /*
1352          * used for mptsas_watch
1353          */
1354         mptsas_list_add(mpt);

1356         mutex_enter(&mptsas_global_mutex);
1357         if (mptsas_timeouts_enabled == 0) {
1358                 mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
1359                     dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);

1361                 mptsas_tick = mptsas_scsi_watchdog_tick *
1362                     drv_usectohz((clock_t)1000000);

1364                 mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
1365                 mptsas_timeouts_enabled = 1;
1366         }
1367         mutex_exit(&mptsas_global_mutex);

1369         /* Print message of HBA present */
1370         ddi_report_dev(dip);

1372         /* report idle status to pm framework */
1373         if (mpt->m_options & MPTSAS_OPT_PM) {
1374                 (void) pm_idle_component(dip, 0);
1375         }

1377         return (DDI_SUCCESS);

1379 fail:
1380         mptsas_log(mpt, CE_WARN, "attach failed");
1381         mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
1382         ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
1383         if (mpt) {
1384                 mutex_enter(&mptsas_global_mutex);

1386                 if (mptsas_timeout_id && (mptsas_head == NULL)) {
1387                         timeout_id_t tid = mptsas_timeout_id;
1388                         mptsas_timeouts_enabled = 0;
1389                         mptsas_timeout_id = 0;
1390                         mutex_exit(&mptsas_global_mutex);
1391                         (void) untimeout(tid);
1392                         mutex_enter(&mptsas_global_mutex);
1393                 }
1394                 mutex_exit(&mptsas_global_mutex);
1395                 /* deallocate in reverse order */
1396                 mptsas_cache_destroy(mpt);
```



```
1398                 if (smp_attach_setup) {
1399                         mptsas_smp_teardown(mpt);
1400                 }
1401                 if (hba_attach_setup) {
1402                         mptsas_hba_teardown(mpt);
1403                 }

1405                 if (mpt->m_active) {
1406                         mptsas_hash_uninit(&mpt->m_active->m_smptbl,
1407                             sizeof (mptsas_smp_t));
1408                         mptsas_hash_uninit(&mpt->m_active->m_tgttbl,
1409                             sizeof (mptsas_target_t));
1410                         mptsas_free_active_slots(mpt);
1411                 }
1412                 if (intr_added) {
1413                         mptsas_unregister_intrs(mpt);
1414                 }

1416                 if (doneq_thread_create) {
1417                         mutex_enter(&mpt->m_doneq_mutex);
1418                         doneq_thread_num = mpt->m_doneq_thread_n;
1419                         for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1420                                 mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1421                                 mpt->m_doneq_thread_id[j].flag &=
1422                                     (~MPTSAS_DONEQ_THREAD_ACTIVE);
1423                                 cv_signal(&mpt->m_doneq_thread_id[j].cv);
1424                                 mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1425                         }
1426                         while (mpt->m_doneq_thread_n) {
1427                                 cv_wait(&mpt->m_doneq_thread_cv,
1428                                     &mpt->m_doneq_mutex);
1429                         }
1430                         for (j = 0; j < doneq_thread_num; j++) {
1431                                 cv_destroy(&mpt->m_doneq_thread_id[j].cv);
1432                                 mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
1433                         }
1434                         kmem_free(mpt->m_doneq_thread_id,
1435                             sizeof (mptsas_doneq_thread_list_t)
1436                             * doneq_thread_num);
1437                         mutex_exit(&mpt->m_doneq_mutex);
1438                         cv_destroy(&mpt->m_doneq_thread_cv);
1439                         mutex_destroy(&mpt->m_doneq_mutex);
1440                 }
1441                 if (event_taskq_create) {
1442                         ddi_taskq_destroy(mpt->m_event_taskq);
1443                 }
1444                 if (dr_taskq_create) {
1445                         ddi_taskq_destroy(mpt->m_dr_taskq);
1446                 }
1447                 if (mutex_init_done) {
1448                         mutex_destroy(&mpt->m_tx_waitq_mutex);
1462                         mutex_destroy(&mpt->m_intr_mutex);
1449                         mutex_destroy(&mpt->m_passthru_mutex);
1450                         mutex_destroy(&mpt->m_mutex);
1451                         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1452                                 mutex_destroy(
1453                                     &mpt->m_phy_info[i].smhba_info.phy_mutex);
1454                         }
1455                         cv_destroy(&mpt->m_cv);
1456                         cv_destroy(&mpt->m_passthru_cv);
1457                         cv_destroy(&mpt->m_fw_cv);
1458                         cv_destroy(&mpt->m_config_cv);
1459                         cv_destroy(&mpt->m_fw_diag_cv);
1460                 }

1462                 if (map_setup) {
```

```
1463                        mptsas_cfg_fini(mpt);
1464                }
1465                if (config_setup) {
1466                        mptsas_config_space_fini(mpt);
1467                }
1468                mptsas_free_handshake_msg(mpt);
1469                mptsas_hba_fini(mpt);

1471                mptsas_fm_fini(mpt);
1472                ddi_soft_state_free(mptsas_state, instance);
1473                ddi_prop_remove_all(dip);
1474        }
1475        return (DDI_FAILURE);
1476 }
```

*unchanged_portion_omitted_*

```
1677 static int
1678 mptsas_do_detach(dev_info_t *dip)
1679 {
1680        mptsas_t        *mpt;
1681        scsi_hba_tran_t *tran;
1682        int             circ = 0;
1683        int             circ1 = 0;
1684        mdi_pathinfo_t  *pip = NULL;
1685        int             i;
1686        int             doneq_thread_num = 0;

1688        NDBG0(("mptsas_do_detach: dip=0x%p", (void *)dip));

1690        if ((tran = ndi_flavorv_get(dip, SCSA_FLAVOR_SCSI_DEVICE)) == NULL)
1691                return (DDI_FAILURE);

1693        mpt = TRAN2MPT(tran);
1694        if (!mpt) {
1695                return (DDI_FAILURE);
1696        }
1697        /*
1698         * Still have pathinfo child, should not detach mpt driver
1699         */
1700        if (scsi_hba_iport_unit_address(dip)) {
1701                if (mpt->m_mpxio_enable) {
1702                        /*
1703                         * MPxIO enabled for the iport
1704                         */
1705                        ndi_devi_enter(scsi_vhci_dip, &circ1);
1706                        ndi_devi_enter(dip, &circ);
1707                        while (pip = mdi_get_next_client_path(dip, NULL)) {
1708                                if (mdi_pi_free(pip, 0) == MDI_SUCCESS) {
1709                                        continue;
1710                                }
1711                                ndi_devi_exit(dip, circ);
1712                                ndi_devi_exit(scsi_vhci_dip, circ1);
1713                                NDBG12(("detach failed because of "
1714                                    "outstanding path info"));
1715                                return (DDI_FAILURE);
1716                        }
1717                        ndi_devi_exit(dip, circ);
1718                        ndi_devi_exit(scsi_vhci_dip, circ1);
1719                        (void) mdi_phci_unregister(dip, 0);
1720                }

1722                ddi_prop_remove_all(dip);

1724                return (DDI_SUCCESS);
1725        }
```



```
1727        /* Make sure power level is D0 before accessing registers */
1728        if (mpt->m_options & MPTSAS_OPT_PM) {
1729                (void) pm_busy_component(dip, 0);
1730                if (mpt->m_power_level != PM_LEVEL_D0) {
1731                        if (pm_raise_power(dip, 0, PM_LEVEL_D0) !=
1732                            DDI_SUCCESS) {
1733                                mptsas_log(mpt, CE_WARN,
1734                                    "mptsas%d: Raise power request failed.",
1735                                    mpt->m_instance);
1736                                (void) pm_idle_component(dip, 0);
1737                                return (DDI_FAILURE);
1738                        }
1739                }
1740        }

1742        /*
1743         * Send RAID action system shutdown to sync IR.  After action, send a
1744         * Message Unit Reset. Since after that DMA resource will be freed,
1745         * set ioc to READY state will avoid HBA initiated DMA operation.
1746         */
1747        mutex_enter(&mpt->m_mutex);
1748        MPTSAS_DISABLE_INTR(mpt);
1749        mptsas_raid_action_system_shutdown(mpt);
1750        mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1751        (void) mptsas_ioc_reset(mpt, FALSE);
1752        mutex_exit(&mpt->m_mutex);
1753        mptsas_rem_intrs(mpt);
1754        ddi_taskq_destroy(mpt->m_event_taskq);
1755        ddi_taskq_destroy(mpt->m_dr_taskq);

1757        if (mpt->m_doneq_thread_n) {
1758                mutex_enter(&mpt->m_doneq_mutex);
1759                doneq_thread_num = mpt->m_doneq_thread_n;
1760                for (i = 0; i < mpt->m_doneq_thread_n; i++) {
1761                        mutex_enter(&mpt->m_doneq_thread_id[i].mutex);
1762                        mpt->m_doneq_thread_id[i].flag &=
1763                            (~MPTSAS_DONEQ_THREAD_ACTIVE);
1764                        cv_signal(&mpt->m_doneq_thread_id[i].cv);
1765                        mutex_exit(&mpt->m_doneq_thread_id[i].mutex);
1766                }
1767                while (mpt->m_doneq_thread_n) {
1768                        cv_wait(&mpt->m_doneq_thread_cv,
1769                            &mpt->m_doneq_mutex);
1770                }
1771                for (i = 0;  i < doneq_thread_num; i++) {
1772                        cv_destroy(&mpt->m_doneq_thread_id[i].cv);
1773                        mutex_destroy(&mpt->m_doneq_thread_id[i].mutex);
1774                }
1775                kmem_free(mpt->m_doneq_thread_id,
1776                    sizeof (mptsas_doneq_thread_list_t)
1777                    * doneq_thread_num);
1778                mutex_exit(&mpt->m_doneq_mutex);
1779                cv_destroy(&mpt->m_doneq_thread_cv);
1780                mutex_destroy(&mpt->m_doneq_mutex);
1781        }

1783        scsi_hba_reset_notify_tear_down(mpt->m_reset_notify_listf);

1785        mptsas_list_del(mpt);

1787        /*
1788         * Cancel timeout threads for this mpt
1789         */
1790        mutex_enter(&mpt->m_mutex);
1791        if (mpt->m_quiesce_timeid) {
1792                timeout_id_t tid = mpt->m_quiesce_timeid;
```

```
1793                mpt->m_quiesce_timeid = 0;
1794                mutex_exit(&mpt->m_mutex);
1795                (void) untimeout(tid);
1796                mutex_enter(&mpt->m_mutex);
1797        }

1799        if (mpt->m_restart_cmd_timeid) {
1800                timeout_id_t tid = mpt->m_restart_cmd_timeid;
1801                mpt->m_restart_cmd_timeid = 0;
1802                mutex_exit(&mpt->m_mutex);
1803                (void) untimeout(tid);
1804                mutex_enter(&mpt->m_mutex);
1805        }

1807        mutex_exit(&mpt->m_mutex);

1809        /*
1810         * last mpt? ... if active, CANCEL watch threads.
1811         */
1812        mutex_enter(&mptsas_global_mutex);
1813        if (mptsas_head == NULL) {
1814                timeout_id_t tid;
1815                /*
1816                 * Clear mptsas_timeouts_enable so that the watch thread
1817                 * gets restarted on DDI_ATTACH
1818                 */
1819                mptsas_timeouts_enabled = 0;
1820                if (mptsas_timeout_id) {
1821                        tid = mptsas_timeout_id;
1822                        mptsas_timeout_id = 0;
1823                        mutex_exit(&mptsas_global_mutex);
1824                        (void) untimeout(tid);
1825                        mutex_enter(&mptsas_global_mutex);
1826                }
1827                if (mptsas_reset_watch) {
1828                        tid = mptsas_reset_watch;
1829                        mptsas_reset_watch = 0;
1830                        mutex_exit(&mptsas_global_mutex);
1831                        (void) untimeout(tid);
1832                        mutex_enter(&mptsas_global_mutex);
1833                }
1834        }
1835        mutex_exit(&mptsas_global_mutex);

1837        /*
1838         * Delete Phy stats
1839         */
1840        mptsas_destroy_phy_stats(mpt);

1842        /*
1843         * Delete nt_active.
1844         */
1845        mutex_enter(&mpt->m_mutex);
1846        mptsas_hash_uninit(&mpt->m_active->m_tgttbl, sizeof (mptsas_target_t));
1847        mptsas_hash_uninit(&mpt->m_active->m_smptbl, sizeof (mptsas_smp_t));
1848        mptsas_free_active_slots(mpt);
1849        mutex_exit(&mpt->m_mutex);

1851        /* deallocate everything that was allocated in mptsas_attach */
1852        mptsas_cache_destroy(mpt);

1854        mptsas_hba_fini(mpt);
1855        mptsas_cfg_fini(mpt);

1857        /* Lower the power informing PM Framework */
1858        if (mpt->m_options & MPTSAS_OPT_PM) {
```



```
1859                if (pm_lower_power(dip, 0, PM_LEVEL_D3) != DDI_SUCCESS)
1860                        mptsas_log(mpt, CE_WARN,
1861                            "!mptsas%d: Lower power request failed "
1862                            "during detach, ignoring.",
1863                            mpt->m_instance);
1864        }

1866        mutex_destroy(&mpt->m_tx_waitq_mutex);
1880        mutex_destroy(&mpt->m_intr_mutex);
1867        mutex_destroy(&mpt->m_passthru_mutex);
1868        mutex_destroy(&mpt->m_mutex);
1869        for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1870                mutex_destroy(&mpt->m_phy_info[i].smhba_info.phy_mutex);
1871        }
1872        cv_destroy(&mpt->m_cv);
1873        cv_destroy(&mpt->m_passthru_cv);
1874        cv_destroy(&mpt->m_fw_cv);
1875        cv_destroy(&mpt->m_config_cv);
1876        cv_destroy(&mpt->m_fw_diag_cv);


1879        mptsas_smp_teardown(mpt);
1880        mptsas_hba_teardown(mpt);

1882        mptsas_config_space_fini(mpt);

1884        mptsas_free_handshake_msg(mpt);

1886        mptsas_fm_fini(mpt);
1887        ddi_soft_state_free(mptsas_state, ddi_get_instance(dip));
1888        ddi_prop_remove_all(dip);

1890        return (DDI_SUCCESS);
1891 }
______unchanged_portion_omitted_

2174 static int
2175 mptsas_power(dev_info_t *dip, int component, int level)
2176 {
2177 #ifndef __lock_lint
2178         _NOTE(ARGUNUSED(component))
2179 #endif
2180         mptsas_t        *mpt;
2181         int             rval = DDI_SUCCESS;
2182         int             polls = 0;
2183         uint32_t        ioc_status;

2185         if (scsi_hba_iport_unit_address(dip) != 0)
2186                 return (DDI_SUCCESS);

2188         mpt = ddi_get_soft_state(mptsas_state, ddi_get_instance(dip));
2189         if (mpt == NULL) {
2190                 return (DDI_FAILURE);
2191         }

2193         mutex_enter(&mpt->m_mutex);

2195         /*
2196          * If the device is busy, don't lower its power level
2197          */
2198         if (mpt->m_busy && (mpt->m_power_level > level)) {
2199                 mutex_exit(&mpt->m_mutex);
2200                 return (DDI_FAILURE);
2201         }
2202         switch (level) {
2203         case PM_LEVEL_D0:
```

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

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
```

```
 * without interrupts (a FLAG_NOINTR command).
 */
int
mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
{
        int     rval = TRUE;

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

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

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

        /*
         * Wait, using drv_usecwait(), long enough for the command to
         * reasonably return from the target if the target isn't
         * "dead".  A polled command may well be sent from scsi_poll, and
         * there are retries built in to scsi_poll if the transport
         * accepted the packet (TRAN_ACCEPT).  scsi_poll waits 1 second
         * and retries the transport up to scsi_poll_busycnt times
         * (currently 60) if
         * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
         * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
         *
         * limit the waiting to avoid a hang in the event that the
         * cmd never gets started but we are still receiving interrupts
         */
        while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
                if (mptsas_wait_intr(mpt, polltime) == FALSE) {
                        NDBG5(("mptsas_poll: command incomplete"));
                        rval = FALSE;
                        break;
                }
        }

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

        if (rval == FALSE) {

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



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

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

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

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

        }
        mptsas_fma_check(mpt, poll_cmd);
        NDBG5(("mptsas_poll: done"));
        return (rval);
}

/*
 * Used for polling cmds and TM function
 */
static int
mptsas_wait_intr(mptsas_t *mpt, int polltime)
{
        int                             cnt;
        pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
        Mpi2ReplyDescriptorsUnion_t     reply_desc_union_v;
        uint32_t                        int_mask;
        uint8_t reply_type;

        NDBG5(("mptsas_wait_intr"));

        mpt->m_polled_intr = 1;

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

        /*
         * Keep polling for at least (polltime * 1000) seconds
         */
        for (cnt = 0; cnt < polltime; cnt++) {
                mutex_enter(&mpt->m_intr_mutex);
                (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
                    DDI_DMA_SYNC_FORCPU);

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

                if (ddi_get32(mpt->m_acc_post_queue_hdl,
                    &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
                    ddi_get32(mpt->m_acc_post_queue_hdl,
                    &reply_desc_union->Words.High) == 0xFFFFFFFF) {
                        mutex_exit(&mpt->m_intr_mutex);
                        drv_usecwait(1000);
                        continue;
                }

                reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
                    &reply_desc_union->Default.ReplyFlags);
                reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
                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                  }
4829                  if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
4830                      (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
4831                          mptsas_log(mpt, CE_WARN, "No memory available"
4832                          "for dispatch taskq");
4833                          /*
4834                           * Return the reply frame to the free queue.
4835                           */
4836                          ddi_put32(mpt->m_acc_free_queue_hdl,
4837                              &((uint32_t *)(void *)
4838                              mpt->m_free_queue)[mpt->m_free_index], reply_addr);
4839                          (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4840                              DDI_DMA_SYNC_FORDEV);
4841                          if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4842                                  mpt->m_free_index = 0;
4843                          }

4845                          ddi_put32(mpt->m_datap,
4846                              &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
4847                  }
4848                  return;
4849          case MPI2_FUNCTION_DIAG_BUFFER_POST:
4850                  /*
4851                   * If SMID is 0, this implies that the reply is due to a
```

```
4852                  * release function with a status that the buffer has been
4853                  * released.  Set the buffer flags accordingly.
4854                  */
4855                 if (SMID == 0) {
4856                         iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
4857                             &reply->IOCStatus);
4858                         buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
4859                             &(((pMpi2DiagBufferPostReply_t)reply)->BufferType));
4860                         if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
4861                                 pBuffer =
4862                                     &mpt->m_fw_diag_buffer_list[buffer_type];
4863                                 pBuffer->valid_data = TRUE;
4864                                 pBuffer->owned_by_firmware = FALSE;
4865                                 pBuffer->immediate = FALSE;
4866                         }
4867                 } else {
4868                         /*
4869                          * Normal handling of diag post reply with SMID.
4870                          */
4871                         cmd = slots->m_slot[SMID];

4873                         /*
4874                          * print warning and return if the slot is empty
4875                          */
4876                         if (cmd == NULL) {
4877                                 mptsas_log(mpt, CE_WARN, "?NULL command for "
4878                                     "address reply in slot %d", SMID);
4879                                 return;
4880                         }
4881                         cmd->cmd_rfm = reply_addr;
4882                         cmd->cmd_flags |= CFLAG_FINISHED;
4883                         cv_broadcast(&mpt->m_fw_diag_cv);
4884                 }
4885                 return;
4886         default:
4887                 mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
4888                 break;
4889         }

4891         /*
4892          * Return the reply frame to the free queue.
4893          */
4894         ddi_put32(mpt->m_acc_free_queue_hdl,
4895             &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
4896             reply_addr);
4897         (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4898             DDI_DMA_SYNC_FORDEV);
4899         if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4900                 mpt->m_free_index = 0;
4901         }
4902         ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
4903             mpt->m_free_index);

4905         if (cmd->cmd_flags & CFLAG_FW_CMD)
4906                 return;

4908         if (cmd->cmd_flags & CFLAG_RETRY) {
4909                 /*
4910                  * The target returned QFULL or busy, do not add tihs
4911                  * pkt to the doneq since the hba will retry
4912                  * this cmd.
4913                  *
4914                  * The pkt has already been resubmitted in
4915                  * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4916                  * Remove this cmd_flag here.
4917                  */
```



```
4918                 cmd->cmd_flags &= ~CFLAG_RETRY;
4919         } else {
4920                 mptsas_doneq_add(mpt, cmd);
4921         }
4922 }

4924 static void
4925 mptsas_check_scsi_io_error(mptsas_t *mpt, pMpi2SCSIIOReply_t reply,
4926     mptsas_cmd_t *cmd)
4927 {
4928         uint8_t                 scsi_status, scsi_state;
4929         uint16_t                ioc_status;
4930         uint32_t                xferred, sensecount, responsedata, loginfo = 0;
4931         struct scsi_pkt         *pkt;
4932         struct scsi_arq_status  *arqstat;
4933         struct buf              *bp;
4934         mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
4935         uint8_t                 *sensedata = NULL;

4937         if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
4938             (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
4939                 bp = cmd->cmd_ext_arq_buf;
4940         } else {
4941                 bp = cmd->cmd_arq_buf;
4942         }

4944         scsi_status = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIStatus);
4945         ioc_status = ddi_get16(mpt->m_acc_reply_frame_hdl, &reply->IOCStatus);
4946         scsi_state = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->SCSIState);
4947         xferred = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->TransferCount);
4948         sensecount = ddi_get32(mpt->m_acc_reply_frame_hdl, &reply->SenseCount);
4949         responsedata = ddi_get32(mpt->m_acc_reply_frame_hdl,
4950             &reply->ResponseInfo);

4952         if (ioc_status & MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
4953                 loginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
4954                     &reply->IOCLogInfo);
4955                 mptsas_log(mpt, CE_NOTE,
4956                     "?Log info 0x%x received for target %d.\n"
4957                     "\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
4958                     loginfo, Tgt(cmd), scsi_status, ioc_status,
4959                     scsi_state);
4960         }

4962         NDBG31(("\t\tscsi_status=0x%x, ioc_status=0x%x, scsi_state=0x%x",
4963             scsi_status, ioc_status, scsi_state));

4965         pkt = CMD2PKT(cmd);
4966         *(pkt->pkt_scbp) = scsi_status;

4968         if (loginfo == 0x31170000) {
4969                 /*
4970                  * if loginfo PL_LOGINFO_CODE_IO_DEVICE_MISSING_DELAY_RETRY
4971                  * 0x31170000 comes, that means the device missing delay
4972                  * is in progressing, the command need retry later.
4973                  */
4974                 *(pkt->pkt_scbp) = STATUS_BUSY;
4975                 return;
4976         }

4978         if ((scsi_state & MPI2_SCSI_STATE_NO_SCSI_STATUS) &&
4979             ((ioc_status & MPI2_IOCSTATUS_MASK) ==
4980             MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE)) {
4981                 pkt->pkt_reason = CMD_INCOMPLETE;
4982                 pkt->pkt_state |= STATE_GOT_BUS;
5115                 mutex_enter(&ptgt->m_tgt_intr_mutex);
```

```
4983                if (ptgt->m_reset_delay == 0) {
4984                        mptsas_set_throttle(mpt, ptgt,
4985                            DRAIN_THROTTLE);
4986                }
5120                mutex_exit(&ptgt->m_tgt_intr_mutex);
4987                return;
4988        }

4990        if (scsi_state & MPI2_SCSI_STATE_RESPONSE_INFO_VALID) {
4991                responsedata &= 0x000000FF;
4992                if (responsedata & MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF) {
4993                        mptsas_log(mpt, CE_NOTE, "Do not support the TLR\n");
4994                        pkt->pkt_reason = CMD_TLR_OFF;
4995                        return;
4996                }
4997        }


5000        switch (scsi_status) {
5001        case MPI2_SCSI_STATUS_CHECK_CONDITION:
5002                pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5003                arqstat = (void*)(pkt->pkt_scbp);
5004                arqstat->sts_rqpkt_status = *((struct scsi_status *)
5005                    (pkt->pkt_scbp));
5006                pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET |
5007                    STATE_SENT_CMD | STATE_GOT_STATUS | STATE_ARQ_DONE);
5008                if (cmd->cmd_flags & CFLAG_XARQ) {
5009                        pkt->pkt_state |= STATE_XARQ_DONE;
5010                }
5011                if (pkt->pkt_resid != cmd->cmd_dmacount) {
5012                        pkt->pkt_state |= STATE_XFERRED_DATA;
5013                }
5014                arqstat->sts_rqpkt_reason = pkt->pkt_reason;
5015                arqstat->sts_rqpkt_state  = pkt->pkt_state;
5016                arqstat->sts_rqpkt_state |= STATE_XFERRED_DATA;
5017                arqstat->sts_rqpkt_statistics = pkt->pkt_statistics;
5018                sensedata = (uint8_t *)&arqstat->sts_sensedata;

5020                bcopy((uchar_t *)bp->b_un.b_addr, sensedata,
5021                    ((cmd->cmd_rqslen >= sensecount) ? sensecount :
5022                    cmd->cmd_rqslen));
5023                arqstat->sts_rqpkt_resid = (cmd->cmd_rqslen - sensecount);
5024                cmd->cmd_flags |= CFLAG_CMDARQ;
5025                /*
5026                 * Set proper status for pkt if autosense was valid
5027                 */
5028                if (scsi_state & MPI2_SCSI_STATE_AUTOSENSE_VALID) {
5029                        struct scsi_status zero_status = { 0 };
5030                        arqstat->sts_rqpkt_status = zero_status;
5031                }

5033                /*
5034                 * ASC=0x47 is parity error
5035                 * ASC=0x48 is initiator detected error received
5036                 */
5037                if ((scsi_sense_key(sensedata) == KEY_ABORTED_COMMAND) &&
5038                    ((scsi_sense_asc(sensedata) == 0x47) ||
5039                    (scsi_sense_asc(sensedata) == 0x48))) {
5040                        mptsas_log(mpt, CE_NOTE, "Aborted_command!");
5041                }

5043                /*
5044                 * ASC/ASCQ=0x3F/0x0E means report_luns data changed
5045                 * ASC/ASCQ=0x25/0x00 means invalid lun
5046                 */
5047                if (((scsi_sense_key(sensedata) == KEY_UNIT_ATTENTION) &&
```



```
5048                    (scsi_sense_asc(sensedata) == 0x3F) &&
5049                    (scsi_sense_ascq(sensedata) == 0x0E)) ||
5050                    ((scsi_sense_key(sensedata) == KEY_ILLEGAL_REQUEST) &&
5051                    (scsi_sense_asc(sensedata) == 0x25) &&
5052                    (scsi_sense_ascq(sensedata) == 0x00))) {
5053                        mptsas_topo_change_list_t *topo_node = NULL;

5055                        topo_node = kmem_zalloc(
5056                            sizeof (mptsas_topo_change_list_t),
5057                            KM_NOSLEEP);
5058                        if (topo_node == NULL) {
5059                                mptsas_log(mpt, CE_NOTE, "No memory"
5060                                    "resource for handle SAS dynamic"
5061                                    "reconfigure.\n");
5062                                break;
5063                        }
5064                        topo_node->mpt = mpt;
5065                        topo_node->event = MPTSAS_DR_EVENT_RECONFIG_TARGET;
5066                        topo_node->un.phymask = ptgt->m_phymask;
5067                        topo_node->devhdl = ptgt->m_devhdl;
5068                        topo_node->object = (void *)ptgt;
5069                        topo_node->flags = MPTSAS_TOPO_FLAG_LUN_ASSOCIATED;

5071                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
5072                            mptsas_handle_dr,
5073                            (void *)topo_node,
5074                            DDI_NOSLEEP)) != DDI_SUCCESS) {
5075                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq"
5076                                    "for handle SAS dynamic reconfigure"
5077                                    "failed. \n");
5078                        }
5079                }
5080                break;
5081        case MPI2_SCSI_STATUS_GOOD:
5082                switch (ioc_status & MPI2_IOCSTATUS_MASK) {
5083                case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
5084                        pkt->pkt_reason = CMD_DEV_GONE;
5085                        pkt->pkt_state |= STATE_GOT_BUS;
5220                        mutex_enter(&ptgt->m_tgt_intr_mutex);
5086                        if (ptgt->m_reset_delay == 0) {
5087                                mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5088                        }
5224                        mutex_exit(&ptgt->m_tgt_intr_mutex);
5089                        NDBG31(("lost disk for target%d, command:%x",
5090                            Tgt(cmd), pkt->pkt_cdbp[0]));
5091                        break;
5092                case MPI2_IOCSTATUS_SCSI_DATA_OVERRUN:
5093                        NDBG31(("data overrun: xferred=%d", xferred));
5094                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5095                        pkt->pkt_reason = CMD_DATA_OVR;
5096                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5097                            | STATE_SENT_CMD | STATE_GOT_STATUS
5098                            | STATE_XFERRED_DATA);
5099                        pkt->pkt_resid = 0;
5100                        break;
5101                case MPI2_IOCSTATUS_SCSI_RESIDUAL_MISMATCH:
5102                case MPI2_IOCSTATUS_SCSI_DATA_UNDERRUN:
5103                        NDBG31(("data underrun: xferred=%d", xferred));
5104                        NDBG31(("dmacount=%d", cmd->cmd_dmacount));
5105                        pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET
5106                            | STATE_SENT_CMD | STATE_GOT_STATUS);
5107                        pkt->pkt_resid = (cmd->cmd_dmacount - xferred);
5108                        if (pkt->pkt_resid != cmd->cmd_dmacount) {
5109                                pkt->pkt_state |= STATE_XFERRED_DATA;
5110                        }
5111                        break;
```

```
5112                 case MPI2_IOCSTATUS_SCSI_TASK_TERMINATED:
5113                         mptsas_set_pkt_reason(mpt,
5114                             cmd, CMD_RESET, STAT_BUS_RESET);
5115                         break;
5116                 case MPI2_IOCSTATUS_SCSI_IOC_TERMINATED:
5117                 case MPI2_IOCSTATUS_SCSI_EXT_TERMINATED:
5118                         mptsas_set_pkt_reason(mpt,
5119                             cmd, CMD_RESET, STAT_DEV_RESET);
5120                         break;
5121                 case MPI2_IOCSTATUS_SCSI_IO_DATA_ERROR:
5122                 case MPI2_IOCSTATUS_SCSI_PROTOCOL_ERROR:
5123                         pkt->pkt_state |= (STATE_GOT_BUS | STATE_GOT_TARGET);
5124                         mptsas_set_pkt_reason(mpt,
5125                             cmd, CMD_TERMINATED, STAT_TERMINATED);
5126                         break;
5127                 case MPI2_IOCSTATUS_INSUFFICIENT_RESOURCES:
5128                 case MPI2_IOCSTATUS_BUSY:
5129                         /*
5130                          * set throttles to drain
5131                          */
5132                         ptgt = (mptsas_target_t *)mptsas_hash_traverse(
5133                             &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
5134                         while (ptgt != NULL) {
5271                                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5135                                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5273                                 mutex_exit(&ptgt->m_tgt_intr_mutex);

5137                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
5138                                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
5139                         }

5141                         /*
5142                          * retry command
5143                          */
5144                         cmd->cmd_flags |= CFLAG_RETRY;
5145                         cmd->cmd_pkt_flags |= FLAG_HEAD;

5285                         mutex_exit(&mpt->m_mutex);
5147                         (void) mptsas_accept_pkt(mpt, cmd);
5287                         mutex_enter(&mpt->m_mutex);
5148                         break;
5149                 default:
5150                         mptsas_log(mpt, CE_WARN,
5151                             "unknown ioc_status = %x\n", ioc_status);
5152                         mptsas_log(mpt, CE_CONT, "scsi_state = %x, transfer "
5153                             "count = %x, scsi_status = %x", scsi_state,
5154                             xferred, scsi_status);
5155                         break;
5156                 }
5157                 break;
5158         case MPI2_SCSI_STATUS_TASK_SET_FULL:
5159                 mptsas_handle_qfull(mpt, cmd);
5160                 break;
5161         case MPI2_SCSI_STATUS_BUSY:
5162                 NDBG31(("scsi_status busy received"));
5163                 break;
5164         case MPI2_SCSI_STATUS_RESERVATION_CONFLICT:
5165                 NDBG31(("scsi_status reservation conflict received"));
5166                 break;
5167         default:
5168                 mptsas_log(mpt, CE_WARN, "scsi_status=%x, ioc_status=%x\n",
5169                     scsi_status, ioc_status);
5170                 mptsas_log(mpt, CE_WARN,
5171                     "mptsas_process_intr: invalid scsi status\n");
5172                 break;
5173         }
```



```
5174 }
______unchanged_portion_omitted_


5263 /*
5264  * mpt interrupt handler.
5265  */
5266 static uint_t
5267 mptsas_intr(caddr_t arg1, caddr_t arg2)
5268 {
5269         mptsas_t                        *mpt = (void *)arg1;
5270         pMpi2ReplyDescriptorsUnion_t    reply_desc_union;
5271         uchar_t                         did_reply = FALSE;
5411         int                             i = 0, j;
5412         uint8_t                         reply_type;
5413         uint16_t                        SMID;

5273         NDBG1(("mptsas_intr: arg1 0x%p arg2 0x%p", (void *)arg1, (void *)arg2));

5275         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);
```

```
5277         /*
5278          * If interrupts are shared by two channels then check whether this
5279          * interrupt is genuinely for this channel by making sure first the
5280          * chip is in high power state.
5281          */
5282         if ((mpt->m_options & MPTSAS_OPT_PM) &&
5283             (mpt->m_power_level != PM_LEVEL_D0)) {
5284                 mutex_exit(&mpt->m_mutex);
5602                 mutex_exit(&mpt->m_intr_mutex);
5285                 return (DDI_INTR_UNCLAIMED);
5286         }

5288         /*
5289          * If polling, interrupt was triggered by some shared interrupt because
5290          * IOC interrupts are disabled during polling, so polling routine will
5291          * handle any replies.  Considering this, if polling is happening,
5292          * return with interrupt unclaimed.
5293          */
5294         if (mpt->m_polled_intr) {
5295                 mutex_exit(&mpt->m_mutex);
5613                 mutex_exit(&mpt->m_intr_mutex);
5296                 mptsas_log(mpt, CE_WARN, "mpt_sas: Unclaimed interrupt");
5297                 return (DDI_INTR_UNCLAIMED);
5298         }

5300         /*
5301          * Read the istat register.
5302          */
5303         if ((INTPENDING(mpt)) != 0) {
5304                 /*
5305                  * read fifo until empty.
5306                  */
5307 #ifndef __lock_lint
5308                 _NOTE(CONSTCOND)
5309 #endif
5310                 while (TRUE) {
5311                         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5312                             DDI_DMA_SYNC_FORCPU);
5313                         reply_desc_union = (pMpi2ReplyDescriptorsUnion_t)
5314                             MPTSAS_GET_NEXT_REPLY(mpt, mpt->m_post_index);

5316                         if (ddi_get32(mpt->m_acc_post_queue_hdl,
5317                             &reply_desc_union->Words.Low) == 0xFFFFFFFF ||
5318                             ddi_get32(mpt->m_acc_post_queue_hdl,
5319                             &reply_desc_union->Words.High) == 0xFFFFFFFF) {
5320                                 break;
5321                         }

5323                         /*
5324                          * The reply is valid, process it according to its
5325                          * type.  Also, set a flag for updating the reply index
5326                          * after they've all been processed.
5327                          */
5328                         did_reply = TRUE;

5330                         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);

5332                         /*
5333                          * Increment post index and roll over if needed.
5334                          */
5335                         if (++mpt->m_post_index == mpt->m_post_queue_depth) {
5336                                 mpt->m_post_index = 0;
5337                         }
5689                         if (i >= MPI_ADDRESS_COALSCE_MAX)
5690                                 break;
5338                 }

5340                 /*
5341                  * Update the global reply index if at least one reply was
5342                  * processed.
5343                  */
5344                 if (did_reply) {
5345                         ddi_put32(mpt->m_datap,
5346                             &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);
5347                 }

5715                 }
5348         } else {
5349                 mutex_exit(&mpt->m_mutex);
5717                 mutex_exit(&mpt->m_intr_mutex);
5350                 return (DDI_INTR_UNCLAIMED);
5351         }
5352         NDBG1(("mptsas_intr complete"));
5721         mutex_exit(&mpt->m_intr_mutex);
```

```
5354         /*
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         /*
5355          * If no helper threads are created, process the doneq in ISR. If
5356          * helpers are created, use the doneq length as a metric to measure the
5357          * load on the interrupt CPU. If it is long enough, which indicates the
5358          * load is heavy, then we deliver the IO completions to the helpers.
5359          * This measurement has some limitations, although it is simple and
5360          * straightforward and works well for most of the cases at present.
5361          */
5362         if (!mpt->m_doneq_thread_n ||
5363             (mpt->m_doneq_len <= mpt->m_doneq_length_threshold)) {
5746         if (!mpt->m_doneq_thread_n) {
5364                 mptsas_doneq_empty(mpt);
5365         } 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) {
5366                 mptsas_deliver_doneq_thread(mpt);
5756                 } else {
5757                         mptsas_doneq_empty(mpt);
5367         }
5759         }

5369         /*
5370          * If there are queued cmd, start them now.
5371          */
5764         mutex_enter(&mpt->m_intr_mutex);
5372         if (mpt->m_waitq != NULL) {
5373                 mptsas_restart_waitq(mpt);
5374         }

5766                 mutex_exit(&mpt->m_intr_mutex);
5767                 mutex_enter(&mpt->m_mutex);
5768                 mptsas_restart_hba(mpt);
5376         mutex_exit(&mpt->m_mutex);
5377         return (DDI_INTR_CLAIMED);
5771         }
5772         mutex_exit(&mpt->m_intr_mutex);
5773         return (DDI_INTR_CLAIMED);
5378 }

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  */
5380 static void
```



```
5381 mptsas_process_intr(mptsas_t *mpt,
5382     pMpi2ReplyDescriptorsUnion_t reply_desc_union)
5383 {
5384         uint8_t reply_type;

5386         ASSERT(mutex_owned(&mpt->m_mutex));

5388         /*
5389          * The reply is valid, process it according to its
5390          * type.  Also, set a flag for updated the reply index
5391          * after they've all been processed.
5392          */
5393         reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5394             &reply_desc_union->Default.ReplyFlags);
5395         reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
5794         reply_type = reply_desc_union->Default.ReplyFlags;
5396         if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS) {
5397                 mptsas_handle_scsi_io_success(mpt, reply_desc_union);
5398         } else if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
5399                 mptsas_handle_address_reply(mpt, reply_desc_union);
5400         } else {
5401                 mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
5402                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5403         }

5405         /*
5406          * Clear the reply descriptor for re-use and increment
5407          * index.
5408          */
5409         ddi_put64(mpt->m_acc_post_queue_hdl,
5410             &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
5411             0xFFFFFFFFFFFFFFFF);
5412         (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5413             DDI_DMA_SYNC_FORDEV);
5414 }

5416 /*
5417  * handle qfull condition
5418  */
5419 static void
5420 mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd)
5421 {
5422         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

5424         if ((++cmd->cmd_qfull_retries > ptgt->m_qfull_retries) ||
5425             (ptgt->m_qfull_retries == 0)) {
5426                 /*
5427                  * We have exhausted the retries on QFULL, or,
5428                  * the target driver has indicated that it
5429                  * wants to handle QFULL itself by setting
5430                  * qfull-retries capability to 0. In either case
5431                  * we want the target driver's QFULL handling
5432                  * to kick in. We do this by having pkt_reason
5433                  * as CMD_CMPLT and pkt_scbp as STATUS_QFULL.
5434                  */
5824                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5435                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
5826                 mutex_exit(&ptgt->m_tgt_intr_mutex);
5436         } else {
5828                 mutex_enter(&ptgt->m_tgt_intr_mutex);
5437                 if (ptgt->m_reset_delay == 0) {
5438                         ptgt->m_t_throttle =
5439                             max((ptgt->m_t_ncmds - 2), 0);
5440                 }
5833                 mutex_exit(&ptgt->m_tgt_intr_mutex);
```

```
5442                cmd->cmd_pkt_flags |= FLAG_HEAD;
5443                cmd->cmd_flags &= ~(CFLAG_TRANFLAG);
5444                cmd->cmd_flags |= CFLAG_RETRY;

5839                mutex_exit(&mpt->m_mutex);
5446                (void) mptsas_accept_pkt(mpt, cmd);
5841                mutex_enter(&mpt->m_mutex);

5448                /*
5449                 * when target gives queue full status with no commands
5450                 * outstanding (m_t_ncmds == 0), throttle is set to 0
5451                 * (HOLD_THROTTLE), and the queue full handling start
5452                 * (see psarc/1994/313); if there are commands outstanding,
5453                 * throttle is set to (m_t_ncmds - 2)
5454                 */
5850                mutex_enter(&ptgt->m_tgt_intr_mutex);
5455                if (ptgt->m_t_throttle == HOLD_THROTTLE) {
5456                        /*
5457                         * By setting throttle to QFULL_THROTTLE, we
5458                         * avoid submitting new commands and in
5459                         * mptsas_restart_cmd find out slots which need
5460                         * their throttles to be cleared.
5461                         */
5462                        mptsas_set_throttle(mpt, ptgt, QFULL_THROTTLE);
5463                        if (mpt->m_restart_cmd_timeid == 0) {
5464                                mpt->m_restart_cmd_timeid =
5465                                    timeout(mptsas_restart_cmd, mpt,
5466                                    ptgt->m_qfull_retry_interval);
5467                        }
5468                }
5865                mutex_exit(&ptgt->m_tgt_intr_mutex);
5469        }
5470 }
```

*_____unchanged_portion_omitted_*

```
5790 static void
5791 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
5792     dev_info_t *parent)
5793 {
5794        mptsas_target_t *ptgt = NULL;
5795        mptsas_smp_t    *psmp = NULL;
5796        mptsas_t        *mpt = (void *)topo_node->mpt;
5797        uint16_t        devhdl;
5798        uint16_t        attached_devhdl;
5799        uint64_t        sas_wwn = 0;
5800        int             rval = 0;
5801        uint32_t        page_address;
5802        uint8_t         phy, flags;
5803        char            *addr = NULL;
5804        dev_info_t      *lundip;
5805        int             circ = 0, circ1 = 0;
5806        char            attached_wwnstr[MPTSAS_WWN_STRLEN];

5808        NDBG20(("mptsas%d handle_topo_change enter", mpt->m_instance));

5810        ASSERT(mutex_owned(&mpt->m_mutex));

5812        switch (topo_node->event) {
5813        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5814        {
5815                char *phy_mask_name;
5816                mptsas_phymask_t phymask = 0;

5818                if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5819                        /*
5820                         * Get latest RAID info.
```



```
5821                         */
5822                        (void) mptsas_get_raid_info(mpt);
5823                        ptgt = mptsas_search_by_devhdl(
5824                            &mpt->m_active->m_tgttbl, topo_node->devhdl);
5825                        if (ptgt == NULL)
5826                                break;
5827                } else {
5828                        ptgt = (void *)topo_node->object;
5829                }

5831                if (ptgt == NULL) {
5832                        /*
5833                         * If a Phys Disk was deleted, RAID info needs to be
5834                         * updated to reflect the new topology.
5835                         */
5836                        (void) mptsas_get_raid_info(mpt);

5838                        /*
5839                         * Get sas device page 0 by DevHandle to make sure if
5840                         * SSP/SATA end device exist.
5841                         */
5842                        page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
5843                            MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
5844                            topo_node->devhdl;

5846                        rval = mptsas_get_target_device_info(mpt, page_address,
5847                            &devhdl, &ptgt);
5848                        if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
5849                                mptsas_log(mpt, CE_NOTE,
5850                                    "mptsas_handle_topo_change: target %d is "
5851                                    "not a SAS/SATA device. \n",
5852                                    topo_node->devhdl);
5853                        } else if (rval == DEV_INFO_FAIL_ALLOC) {
5854                                mptsas_log(mpt, CE_NOTE,
5855                                    "mptsas_handle_topo_change: could not "
5856                                    "allocate memory. \n");
5857                        }
5858                        /*
5859                         * If rval is DEV_INFO_PHYS_DISK than there is nothing
5860                         * else to do, just leave.
5861                         */
5862                        if (rval != DEV_INFO_SUCCESS) {
5863                                return;
5864                        }
5865                }

5867                ASSERT(ptgt->m_devhdl == topo_node->devhdl);

5869                mutex_exit(&mpt->m_mutex);
5870                flags = topo_node->flags;

5872                if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
5873                        phymask = ptgt->m_phymask;
5874                        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
5875                        (void) sprintf(phy_mask_name, "%x", phymask);
5876                        parent = scsi_hba_iport_find(mpt->m_dip,
5877                            phy_mask_name);
5878                        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5879                        if (parent == NULL) {
5880                                mptsas_log(mpt, CE_WARN, "Failed to find a "
5881                                    "iport for PD, should not happen!");
5882                                mutex_enter(&mpt->m_mutex);
5883                                break;
5884                        }
5885                }
```

```
5887                if (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5888                        ndi_devi_enter(parent, &circ1);
5889                        (void) mptsas_config_raid(parent, topo_node->devhdl,
5890                            &lundip);
5891                        ndi_devi_exit(parent, circ1);
5892                } else {
5893                        /*
5894                         * hold nexus for bus configure
5895                         */
5896                        ndi_devi_enter(scsi_vhci_dip, &circ);
5897                        ndi_devi_enter(parent, &circ1);
5898                        rval = mptsas_config_target(parent, ptgt);
5899                        /*
5900                         * release nexus for bus configure
5901                         */
5902                        ndi_devi_exit(parent, circ1);
5903                        ndi_devi_exit(scsi_vhci_dip, circ);

5905                        /*
5906                         * Add parent's props for SMHBA support
5907                         */
5908                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5909                                bzero(attached_wwnstr,
5910                                    sizeof (attached_wwnstr));
5911                                (void) sprintf(attached_wwnstr, "w%016"PRIx64,
5912                                    ptgt->m_sas_wwn);
5913                                if (ddi_prop_update_string(DDI_DEV_T_NONE,
5914                                    parent,
5915                                    SCSI_ADDR_PROP_ATTACHED_PORT,
5916                                    attached_wwnstr)
5917                                    != DDI_PROP_SUCCESS) {
5918                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5919                                            parent,
5920                                            SCSI_ADDR_PROP_ATTACHED_PORT);
5921                                        mptsas_log(mpt, CE_WARN, "Failed to"
5922                                            "attached-port props");
5923                                        return;
5924                                }
5925                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5926                                    MPTSAS_NUM_PHYS, 1) !=
5927                                    DDI_PROP_SUCCESS) {
5928                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5929                                            parent, MPTSAS_NUM_PHYS);
5930                                        mptsas_log(mpt, CE_WARN, "Failed to"
5931                                            " create num-phys props");
5932                                        return;
5933                                }

5935                                /*
5936                                 * Update PHY info for smhba
5937                                 */
5938                                mutex_enter(&mpt->m_mutex);
5939                                if (mptsas_smhba_phy_init(mpt)) {
5940                                        mutex_exit(&mpt->m_mutex);
5941                                        mptsas_log(mpt, CE_WARN, "mptsas phy"
5942                                            " update failed");
5943                                        return;
5944                                }
5945                                mutex_exit(&mpt->m_mutex);
5946                                mptsas_smhba_set_phy_props(mpt,
5947                                    ddi_get_name_addr(parent), parent,
5948                                    1, &attached_devhdl);
5949                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5950                                    MPTSAS_VIRTUAL_PORT, 0) !=
5951                                    DDI_PROP_SUCCESS) {
5952                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
```



```
5953                                            parent, MPTSAS_VIRTUAL_PORT);
5954                                        mptsas_log(mpt, CE_WARN,
5955                                            "mptsas virtual-port"
5956                                            "port prop update failed");
5957                                        return;
5958                                }
5959                        }
5960                }
5961                mutex_enter(&mpt->m_mutex);

5963                NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
5964                    "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
5965                    ptgt->m_phymask));
5966                break;
5967        }
5968        case MPTSAS_DR_EVENT_OFFLINE_TARGET:
5969        {
5970                mptsas_hash_table_t *tgttbl = &mpt->m_active->m_tgttbl;
5971                devhdl = topo_node->devhdl;
5972                ptgt = mptsas_search_by_devhdl(tgttbl, devhdl);
5973                if (ptgt == NULL)
5974                        break;

5976                sas_wwn = ptgt->m_sas_wwn;
5977                phy = ptgt->m_phynum;

5979                addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

5981                if (sas_wwn) {
5982                        (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
5983                } else {
5984                        (void) sprintf(addr, "p%x", phy);
5985                }
5986                ASSERT(ptgt->m_devhdl == devhdl);

5988                if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
5989                    (topo_node->flags ==
5990                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
5991                        /*
5992                         * Get latest RAID info if RAID volume status changes
5993                         * or Phys Disk status changes
5994                         */
5995                        (void) mptsas_get_raid_info(mpt);
5996                }
5997                /*
5998                 * Abort all outstanding command on the device
5999                 */
6000                rval = mptsas_do_scsi_reset(mpt, devhdl);
6001                if (rval) {
6002                        NDBG20(("mptsas%d handle_topo_change to reset target "
6003                            "before offline devhdl:%x, phymask:%x, rval:%x",
6004                            mpt->m_instance, ptgt->m_devhdl, ptgt->m_phymask,
6005                            rval));
6006                }

6008                mutex_exit(&mpt->m_mutex);

6010                ndi_devi_enter(scsi_vhci_dip, &circ);
6011                ndi_devi_enter(parent, &circ1);
6012                rval = mptsas_offline_target(parent, addr);
6013                ndi_devi_exit(parent, circ1);
6014                ndi_devi_exit(scsi_vhci_dip, circ);
6015                NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6016                    "phymask:%x, rval:%x", mpt->m_instance,
6017                    ptgt->m_devhdl, ptgt->m_phymask, rval));
```

```
6019                kmem_free(addr, SCSI_MAXNAMELEN);

6021                /*
6022                 * Clear parent's props for SMHBA support
6023                 */
6024                flags = topo_node->flags;
6025                if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6026                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6027                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6028                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6029                            DDI_PROP_SUCCESS) {
6030                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6031                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6032                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6033                                    "prop update failed");
6034                                break;
6035                        }
6036                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6037                            MPTSAS_NUM_PHYS, 0) !=
6038                            DDI_PROP_SUCCESS) {
6039                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6040                                    MPTSAS_NUM_PHYS);
6041                                mptsas_log(mpt, CE_WARN, "mptsas num phys "
6042                                    "prop update failed");
6043                                break;
6044                        }
6045                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6046                            MPTSAS_VIRTUAL_PORT, 1) !=
6047                            DDI_PROP_SUCCESS) {
6048                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6049                                    MPTSAS_VIRTUAL_PORT);
6050                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6051                                    "prop update failed");
6052                                break;
6053                        }
6054                }

6056                mutex_enter(&mpt->m_mutex);
6057                if (mptsas_set_led_status(mpt, ptgt, 0) != DDI_SUCCESS) {
6058                        NDBG14(("mptsas: clear LED for tgt %x failed",
6059                            ptgt->m_slot_num));
6060                }
6061                if (rval == DDI_SUCCESS) {
6062                        mptsas_tgt_free(&mpt->m_active->m_tgttbl,
6063                            ptgt->m_sas_wwn, ptgt->m_phymask);
6064                        ptgt = NULL;
6065                } else {
6066                        /*
6067                         * clean DR_INTRANSITION flag to allow I/O down to
6068                         * PHCI driver since failover finished.
6069                         * Invalidate the devhdl
6070                         */
6468                        mutex_enter(&ptgt->m_tgt_intr_mutex);
6071                        ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6072                        ptgt->m_tgt_unconfigured = 0;
6073                        mutex_enter(&mpt->m_tx_waitq_mutex);
6074                        ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
6075                        mutex_exit(&mpt->m_tx_waitq_mutex);
6472                        mutex_exit(&ptgt->m_tgt_intr_mutex);
6076                }

6078                /*
6079                 * Send SAS IO Unit Control to free the dev handle
6080                 */
6081                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6082                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
```



```
6083                        rval = mptsas_free_devhdl(mpt, devhdl);

6085                        NDBG20(("mptsas%d handle_topo_change to remove "
6086                            "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6087                            rval));
6088                }

6090                break;
6091        }
6092        case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
6093        {
6094                devhdl = topo_node->devhdl;
6095                /*
6096                 * If this is the remove handle event, do a reset first.
6097                 */
6098                if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6099                        rval = mptsas_do_scsi_reset(mpt, devhdl);
6100                        if (rval) {
6101                                NDBG20(("mpt%d reset target before remove "
6102                                    "devhdl:%x, rval:%x", mpt->m_instance,
6103                                    devhdl, rval));
6104                        }
6105                }

6107                /*
6108                 * Send SAS IO Unit Control to free the dev handle
6109                 */
6110                rval = mptsas_free_devhdl(mpt, devhdl);
6111                NDBG20(("mptsas%d handle_topo_change to remove "
6112                    "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6113                    rval));
6114                break;
6115        }
6116        case MPTSAS_DR_EVENT_RECONFIG_SMP:
6117        {
6118                mptsas_smp_t smp;
6119                dev_info_t *smpdip;
6120                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;

6122                devhdl = topo_node->devhdl;

6124                page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
6125                    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
6126                rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
6127                if (rval != DDI_SUCCESS) {
6128                        mptsas_log(mpt, CE_WARN, "failed to online smp, "
6129                            "handle %x", devhdl);
6130                        return;
6131                }

6133                psmp = mptsas_smp_alloc(smptbl, &smp);
6134                if (psmp == NULL) {
6135                        return;
6136                }

6138                mutex_exit(&mpt->m_mutex);
6139                ndi_devi_enter(parent, &circ1);
6140                (void) mptsas_online_smp(parent, psmp, &smpdip);
6141                ndi_devi_exit(parent, circ1);

6143                mutex_enter(&mpt->m_mutex);
6144                break;
6145        }
6146        case MPTSAS_DR_EVENT_OFFLINE_SMP:
6147        {
6148                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;
```

```
6149                devhdl = topo_node->devhdl;
6150                uint32_t dev_info;

6152                psmp = mptsas_search_by_devhdl(smptbl, devhdl);
6153                if (psmp == NULL)
6154                        break;
6155                /*
6156                 * The mptsas_smp_t data is released only if the dip is offlined
6157                 * successfully.
6158                 */
6159                mutex_exit(&mpt->m_mutex);

6161                ndi_devi_enter(parent, &circ1);
6162                rval = mptsas_offline_smp(parent, psmp, NDI_DEVI_REMOVE);
6163                ndi_devi_exit(parent, circ1);

6165                dev_info = psmp->m_deviceinfo;
6166                if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
6167                    DEVINFO_DIRECT_ATTACHED) {
6168                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6169                            MPTSAS_VIRTUAL_PORT, 1) !=
6170                            DDI_PROP_SUCCESS) {
6171                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6172                                    MPTSAS_VIRTUAL_PORT);
6173                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6174                                    "prop update failed");
6175                                return;
6176                        }
6177                        /*
6178                         * Check whether the smp connected to the iport,
6179                         */
6180                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6181                            MPTSAS_NUM_PHYS, 0) !=
6182                            DDI_PROP_SUCCESS) {
6183                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6184                                    MPTSAS_NUM_PHYS);
6185                                mptsas_log(mpt, CE_WARN, "mptsas num phys"
6186                                    "prop update failed");
6187                                return;
6188                        }
6189                        /*
6190                         * Clear parent's attached-port props
6191                         */
6192                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6193                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6194                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6195                            DDI_PROP_SUCCESS) {
6196                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6197                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6198                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6199                                    "prop update failed");
6200                                return;
6201                        }
6202                }

6204                mutex_enter(&mpt->m_mutex);
6205                NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6206                    "rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
6207                if (rval == DDI_SUCCESS) {
6208                        mptsas_smp_free(smptbl, psmp->m_sasaddr,
6209                            psmp->m_phymask);
6210                } else {
6211                        psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
6212                }

6214                bzero(attached_wwnstr, sizeof (attached_wwnstr));
```



```
6216                        break;
6217                }
6218                default:
6219                        return;
6220                }
6221        }
______unchanged_portion_omitted_

6307    #define SMP_RESET_IN_PROGRESS MPI2_EVENT_SAS_TOPO_LR_SMP_RESET_IN_PROGRESS
6308    /*
6309     * handle sync events from ioc in interrupt
6310     * return value:
6311     * DDI_SUCCESS: The event is handled by this func
6312     * DDI_FAILURE: Event is not handled
6313     */
6314    static int
6315    mptsas_handle_event_sync(void *args)
6316    {
6317            m_replyh_arg_t                  *replyh_arg;
6318            pMpi2EventNotificationReply_t   eventreply;
6319            uint32_t                        event, rfm;
6320            mptsas_t                        *mpt;
6321            uint_t                          iocstatus;

6323            replyh_arg = (m_replyh_arg_t *)args;
6324            rfm = replyh_arg->rfm;
6325            mpt = replyh_arg->mpt;

6327            ASSERT(mutex_owned(&mpt->m_mutex));

6329            eventreply = (pMpi2EventNotificationReply_t)
6330                (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
6331            event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

6333            if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
6334                &eventreply->IOCStatus)) {
6335                    if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
6336                            mptsas_log(mpt, CE_WARN,
6337                                "!mptsas_handle_event_sync: IOCStatus=0x%x, "
6338                                "IOCLogInfo=0x%x", iocstatus,
6339                                ddi_get32(mpt->m_acc_reply_frame_hdl,
6340                                &eventreply->IOCLogInfo));
6341                    } else {
6342                            mptsas_log(mpt, CE_WARN,
6343                                "mptsas_handle_event_sync: IOCStatus=0x%x, "
6344                                "IOCLogInfo=0x%x", iocstatus,
6345                                ddi_get32(mpt->m_acc_reply_frame_hdl,
6346                                &eventreply->IOCLogInfo));
6347                    }
6348            }

6350            /*
6351             * figure out what kind of event we got and handle accordingly
6352             */
6353            switch (event) {
6354            case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
6355            {
6356                    pMpi2EventDataSasTopologyChangeList_t   sas_topo_change_list;
6357                    uint8_t                         num_entries, expstatus, phy;
6358                    uint8_t                         phystatus, physport, state, i;
6359                    uint8_t                         start_phy_num, link_rate;
6360                    uint16_t                        dev_handle, reason_code;
6361                    uint16_t                        enc_handle, expd_handle;
6362                    char                            string[80], curr[80], prev[80];
6363                    mptsas_topo_change_list_t       *topo_head = NULL;
```

```
6364                mptsas_topo_change_list_t       *topo_tail = NULL;
6365                mptsas_topo_change_list_t       *topo_node = NULL;
6366                mptsas_target_t                 *ptgt;
6367                mptsas_smp_t                    *psmp;
6368                mptsas_hash_table_t             *tgttbl, *smptbl;
6369                uint8_t                         flags = 0, exp_flag;
6370                smhba_info_t                    *pSmhba = NULL;

6372                NDBG20(("mptsas_handle_event_sync: SAS topology change"));

6374                tgttbl = &mpt->m_active->m_tgttbl;
6375                smptbl = &mpt->m_active->m_smptbl;

6377                sas_topo_change_list = (pMpi2EventDataSasTopologyChangeList_t)
6378                    eventreply->EventData;

6380                enc_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6381                    &sas_topo_change_list->EnclosureHandle);
6382                expd_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6383                    &sas_topo_change_list->ExpanderDevHandle);
6384                num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
6385                    &sas_topo_change_list->NumEntries);
6386                start_phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
6387                    &sas_topo_change_list->StartPhyNum);
6388                expstatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
6389                    &sas_topo_change_list->ExpStatus);
6390                physport = ddi_get8(mpt->m_acc_reply_frame_hdl,
6391                    &sas_topo_change_list->PhysicalPort);

6393                string[0] = 0;
6394                if (expd_handle) {
6395                        flags = MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED;
6396                        switch (expstatus) {
6397                        case MPI2_EVENT_SAS_TOPO_ES_ADDED:
6398                                (void) sprintf(string, " added");
6399                                /*
6400                                 * New expander device added
6401                                 */
6402                                mpt->m_port_chng = 1;
6403                                topo_node = kmem_zalloc(
6404                                    sizeof (mptsas_topo_change_list_t),
6405                                    KM_SLEEP);
6406                                topo_node->mpt = mpt;
6407                                topo_node->event = MPTSAS_DR_EVENT_RECONFIG_SMP;
6408                                topo_node->un.physport = physport;
6409                                topo_node->devhdl = expd_handle;
6410                                topo_node->flags = flags;
6411                                topo_node->object = NULL;
6412                                if (topo_head == NULL) {
6413                                        topo_head = topo_tail = topo_node;
6414                                } else {
6415                                        topo_tail->next = topo_node;
6416                                        topo_tail = topo_node;
6417                                }
6418                                break;
6419                        case MPI2_EVENT_SAS_TOPO_ES_NOT_RESPONDING:
6420                                (void) sprintf(string, " not responding, "
6421                                    "removed");
6422                                psmp = mptsas_search_by_devhdl(smptbl,
6423                                    expd_handle);
6424                                if (psmp == NULL)
6425                                        break;

6427                                topo_node = kmem_zalloc(
6428                                    sizeof (mptsas_topo_change_list_t),
6429                                    KM_SLEEP);
```



```
6430                                topo_node->mpt = mpt;
6431                                topo_node->un.phymask = psmp->m_phymask;
6432                                topo_node->event = MPTSAS_DR_EVENT_OFFLINE_SMP;
6433                                topo_node->devhdl = expd_handle;
6434                                topo_node->flags = flags;
6435                                topo_node->object = NULL;
6436                                if (topo_head == NULL) {
6437                                        topo_head = topo_tail = topo_node;
6438                                } else {
6439                                        topo_tail->next = topo_node;
6440                                        topo_tail = topo_node;
6441                                }
6442                                break;
6443                        case MPI2_EVENT_SAS_TOPO_ES_RESPONDING:
6444                                break;
6445                        case MPI2_EVENT_SAS_TOPO_ES_DELAY_NOT_RESPONDING:
6446                                (void) sprintf(string, " not responding, "
6447                                    "delaying removal");
6448                                break;
6449                        default:
6450                                break;
6451                        }
6452                } else {
6453                        flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
6454                }

6456                NDBG20(("SAS TOPOLOGY CHANGE for enclosure %x expander %x%s\n",
6457                    enc_handle, expd_handle, string));
6458                for (i = 0; i < num_entries; i++) {
6459                        phy = i + start_phy_num;
6460                        phystatus = ddi_get8(mpt->m_acc_reply_frame_hdl,
6461                            &sas_topo_change_list->PHY[i].PhyStatus);
6462                        dev_handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6463                            &sas_topo_change_list->PHY[i].AttachedDevHandle);
6464                        reason_code = phystatus & MPI2_EVENT_SAS_TOPO_RC_MASK;
6465                        /*
6466                         * Filter out processing of Phy Vacant Status unless
6467                         * the reason code is "Not Responding".  Process all
6468                         * other combinations of Phy Status and Reason Codes.
6469                         */
6470                        if ((phystatus &
6471                            MPI2_EVENT_SAS_TOPO_PHYSTATUS_VACANT) &&
6472                            (reason_code !=
6473                            MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING)) {
6474                                continue;
6475                        }
6476                        curr[0] = 0;
6477                        prev[0] = 0;
6478                        string[0] = 0;
6479                        switch (reason_code) {
6480                        case MPI2_EVENT_SAS_TOPO_RC_TARG_ADDED:
6481                        {
6482                                NDBG20(("mptsas%d phy %d physical_port %d "
6483                                    "dev_handle %d added", mpt->m_instance, phy,
6484                                    physport, dev_handle));
6485                                link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
6486                                    &sas_topo_change_list->PHY[i].LinkRate);
6487                                state = (link_rate &
6488                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
6489                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
6490                                switch (state) {
6491                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
6492                                        (void) sprintf(curr, "is disabled");
6493                                        break;
6494                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6495                                        (void) sprintf(curr, "is offline, "
```

```
6496                                        "failed speed negotiation");
6497                                break;
6498                        case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6499                                (void) sprintf(curr, "SATA OOB "
6500                                    "complete");
6501                                break;
6502                        case SMP_RESET_IN_PROGRESS:
6503                                (void) sprintf(curr, "SMP reset in "
6504                                    "progress");
6505                                break;
6506                        case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6507                                (void) sprintf(curr, "is online at "
6508                                    "1.5 Gbps");
6509                                break;
6510                        case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6511                                (void) sprintf(curr, "is online at 3.0 "
6512                                    "Gbps");
6513                                break;
6514                        case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6515                                (void) sprintf(curr, "is online at 6.0 "
6516                                    "Gbps");
6517                                break;
6518                        default:
6519                                (void) sprintf(curr, "state is "
6520                                    "unknown");
6521                                break;
6522                        }
6523                        /*
6524                         * New target device added into the system.
6525                         * Set association flag according to if an
6526                         * expander is used or not.
6527                         */
6528                        exp_flag =
6529                            MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
6530                        if (flags ==
6531                            MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
6532                                flags = exp_flag;
6533                        }
6534                        topo_node = kmem_zalloc(
6535                            sizeof (mptsas_topo_change_list_t),
6536                            KM_SLEEP);
6537                        topo_node->mpt = mpt;
6538                        topo_node->event =
6539                            MPTSAS_DR_EVENT_RECONFIG_TARGET;
6540                        if (expd_handle == 0) {
6541                                /*
6542                                 * Per MPI 2, if expander dev handle
6543                                 * is 0, it's a directly attached
6544                                 * device. So driver use PHY to decide
6545                                 * which iport is associated
6546                                 */
6547                                physport = phy;
6548                                mpt->m_port_chng = 1;
6549                        }
6550                        topo_node->un.physport = physport;
6551                        topo_node->devhdl = dev_handle;
6552                        topo_node->flags = flags;
6553                        topo_node->object = NULL;
6554                        if (topo_head == NULL) {
6555                                topo_head = topo_tail = topo_node;
6556                        } else {
6557                                topo_tail->next = topo_node;
6558                                topo_tail = topo_node;
6559                        }
6560                        break;
6561                }
```



```
6562                case MPI2_EVENT_SAS_TOPO_RC_TARG_NOT_RESPONDING:
6563                {
6564                        NDBG20(("mptsas%d phy %d physical_port %d "
6565                            "dev_handle %d removed", mpt->m_instance,
6566                            phy, physport, dev_handle));
6567                        /*
6568                         * Set association flag according to if an
6569                         * expander is used or not.
6570                         */
6571                        exp_flag =
6572                            MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
6573                        if (flags ==
6574                            MPTSAS_TOPO_FLAG_EXPANDER_ASSOCIATED) {
6575                                flags = exp_flag;
6576                        }
6577                        /*
6578                         * Target device is removed from the system
6579                         * Before the device is really offline from
6580                         * from system.
6581                         */
6582                        ptgt = mptsas_search_by_devhdl(tgttbl,
6583                            dev_handle);
6584                        /*
6585                         * If ptgt is NULL here, it means that the
6586                         * DevHandle is not in the hash table.  This is
6587                         * reasonable sometimes.  For example, if a
6588                         * disk was pulled, then added, then pulled
6589                         * again, the disk will not have been put into
6590                         * the hash table because the add event will
6591                         * have an invalid phymask.  BUT, this does not
6592                         * mean that the DevHandle is invalid.  The
6593                         * controller will still have a valid DevHandle
6594                         * that must be removed.  To do this, use the
6595                         * MPTSAS_TOPO_FLAG_REMOVE_HANDLE event.
6596                         */
6597                        if (ptgt == NULL) {
6598                                topo_node = kmem_zalloc(
6599                                    sizeof (mptsas_topo_change_list_t),
6600                                    KM_SLEEP);
6601                                topo_node->mpt = mpt;
6602                                topo_node->un.phymask = 0;
6603                                topo_node->event =
6604                                    MPTSAS_TOPO_FLAG_REMOVE_HANDLE;
6605                                topo_node->devhdl = dev_handle;
6606                                topo_node->flags = flags;
6607                                topo_node->object = NULL;
6608                                if (topo_head == NULL) {
6609                                        topo_head = topo_tail =
6610                                            topo_node;
6611                                } else {
6612                                        topo_tail->next = topo_node;
6613                                        topo_tail = topo_node;
6614                                }
6615                                break;
6616                        }

6618                        /*
6619                         * Update DR flag immediately avoid I/O failure
6620                         * before failover finish. Pay attention to the
6621                         * mutex protect, we need grab m_tx_waitq_mutex
6622                         * during set m_dr_flag because we won't add
6623                         * the following command into waitq, instead,
6624                         * we need return TRAN_BUSY in the tran_start
6625                         * 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().
6626                                 */
6627                                mutex_enter(&mpt->m_tx_waitq_mutex);
7023                                mutex_enter(&ptgt->m_tgt_intr_mutex);
6628                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6629                                mutex_exit(&mpt->m_tx_waitq_mutex);
7025                                mutex_exit(&ptgt->m_tgt_intr_mutex);

6631                                topo_node = kmem_zalloc(
6632                                    sizeof (mptsas_topo_change_list_t),
6633                                    KM_SLEEP);
6634                                topo_node->mpt = mpt;
6635                                topo_node->un.phymask = ptgt->m_phymask;
6636                                topo_node->event =
6637                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
6638                                topo_node->devhdl = dev_handle;
6639                                topo_node->flags = flags;
6640                                topo_node->object = NULL;
6641                                if (topo_head == NULL) {
6642                                        topo_head = topo_tail = topo_node;
6643                                } else {
6644                                        topo_tail->next = topo_node;
6645                                        topo_tail = topo_node;
6646                                }
6647                                break;
6648                        }
6649                        case MPI2_EVENT_SAS_TOPO_RC_PHY_CHANGED:
6650                                link_rate = ddi_get8(mpt->m_acc_reply_frame_hdl,
6651                                    &sas_topo_change_list->PHY[i].LinkRate);
6652                                state = (link_rate &
6653                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_MASK) >>
6654                                    MPI2_EVENT_SAS_TOPO_LR_CURRENT_SHIFT;
6655                                pSmhba = &mpt->m_phy_info[i].smhba_info;
6656                                pSmhba->negotiated_link_rate = state;
6657                                switch (state) {
6658                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
6659                                        (void) sprintf(curr, "is disabled");
6660                                        mptsas_smhba_log_sysevent(mpt,
6661                                            ESC_SAS_PHY_EVENT,
6662                                            SAS_PHY_REMOVE,
6663                                            &mpt->m_phy_info[i].smhba_info);
6664                                        mpt->m_phy_info[i].smhba_info.
6665                                            negotiated_link_rate
6666                                            = 0x1;
6667                                        break;
6668                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6669                                        (void) sprintf(curr, "is offline, "
6670                                            "failed speed negotiation");
6671                                        mptsas_smhba_log_sysevent(mpt,
6672                                            ESC_SAS_PHY_EVENT,
6673                                            SAS_PHY_OFFLINE,
6674                                            &mpt->m_phy_info[i].smhba_info);
6675                                        break;
6676                                case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6677                                        (void) sprintf(curr, "SATA OOB "
6678                                            "complete");
6679                                        break;
6680                                case SMP_RESET_IN_PROGRESS:
6681                                        (void) sprintf(curr, "SMP reset in "
6682                                            "progress");
6683                                        break;
6684                                case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6685                                        (void) sprintf(curr, "is online at "
6686                                            "1.5 Gbps");
6687                                        if ((expd_handle == 0) &&
```



```
6688                                            (enc_handle == 1)) {
6689                                                mpt->m_port_chng = 1;
6690                                        }
6691                                        mptsas_smhba_log_sysevent(mpt,
6692                                            ESC_SAS_PHY_EVENT,
6693                                            SAS_PHY_ONLINE,
6694                                            &mpt->m_phy_info[i].smhba_info);
6695                                        break;
6696                                case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6697                                        (void) sprintf(curr, "is online at 3.0 "
6698                                            "Gbps");
6699                                        if ((expd_handle == 0) &&
6700                                            (enc_handle == 1)) {
6701                                                mpt->m_port_chng = 1;
6702                                        }
6703                                        mptsas_smhba_log_sysevent(mpt,
6704                                            ESC_SAS_PHY_EVENT,
6705                                            SAS_PHY_ONLINE,
6706                                            &mpt->m_phy_info[i].smhba_info);
6707                                        break;
6708                                case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
6709                                        (void) sprintf(curr, "is online at "
6710                                            "6.0 Gbps");
6711                                        if ((expd_handle == 0) &&
6712                                            (enc_handle == 1)) {
6713                                                mpt->m_port_chng = 1;
6714                                        }
6715                                        mptsas_smhba_log_sysevent(mpt,
6716                                            ESC_SAS_PHY_EVENT,
6717                                            SAS_PHY_ONLINE,
6718                                            &mpt->m_phy_info[i].smhba_info);
6719                                        break;
6720                                default:
6721                                        (void) sprintf(curr, "state is "
6722                                            "unknown");
6723                                        break;
6724                                }

6726                                state = (link_rate &
6727                                    MPI2_EVENT_SAS_TOPO_LR_PREV_MASK) >>
6728                                    MPI2_EVENT_SAS_TOPO_LR_PREV_SHIFT;
6729                                switch (state) {
6730                                case MPI2_EVENT_SAS_TOPO_LR_PHY_DISABLED:
6731                                        (void) sprintf(prev, ", was disabled");
6732                                        break;
6733                                case MPI2_EVENT_SAS_TOPO_LR_NEGOTIATION_FAILED:
6734                                        (void) sprintf(prev, ", was offline, "
6735                                            "failed speed negotiation");
6736                                        break;
6737                                case MPI2_EVENT_SAS_TOPO_LR_SATA_OOB_COMPLETE:
6738                                        (void) sprintf(prev, ", was SATA OOB "
6739                                            "complete");
6740                                        break;
6741                                case SMP_RESET_IN_PROGRESS:
6742                                        (void) sprintf(prev, ", was SMP reset "
6743                                            "in progress");
6744                                        break;
6745                                case MPI2_EVENT_SAS_TOPO_LR_RATE_1_5:
6746                                        (void) sprintf(prev, ", was online at "
6747                                            "1.5 Gbps");
6748                                        break;
6749                                case MPI2_EVENT_SAS_TOPO_LR_RATE_3_0:
6750                                        (void) sprintf(prev, ", was online at "
6751                                            "3.0 Gbps");
6752                                        break;
6753                                case MPI2_EVENT_SAS_TOPO_LR_RATE_6_0:
```

```
6754                                    (void) sprintf(prev, ", was online at "
6755                                        "6.0 Gbps");
6756                                    break;
6757                            default:
6758                            break;
6759                            }
6760                            (void) sprintf(&string[strlen(string)], "link "
6761                                "changed, ");
6762                            break;
6763                    case MPI2_EVENT_SAS_TOPO_RC_NO_CHANGE:
6764                            continue;
6765                    case MPI2_EVENT_SAS_TOPO_RC_DELAY_NOT_RESPONDING:
6766                            (void) sprintf(&string[strlen(string)],
6767                                "target not responding, delaying "
6768                                "removal");
6769                            break;
6770                    }
6771                    NDBG20(("mptsas%d phy %d DevHandle %x, %s%s%s\n",
6772                        mpt->m_instance, phy, dev_handle, string, curr,
6773                        prev));
6774            }
6775            if (topo_head != NULL) {
6776                    /*
6777                     * Launch DR taskq to handle topology change
6778                     */
6779                    if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
6780                        mptsas_handle_dr, (void *)topo_head,
6781                        DDI_NOSLEEP)) != DDI_SUCCESS) {
6782                            mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
6783                                "for handle SAS DR event failed. \n");
6784                    }
6785            }
6786            break;
6787    }
6788    case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
6789    {
6790            Mpi2EventDataIrConfigChangeList_t       *irChangeList;
6791            mptsas_topo_change_list_t               *topo_head = NULL;
6792            mptsas_topo_change_list_t               *topo_tail = NULL;
6793            mptsas_topo_change_list_t               *topo_node = NULL;
6794            mptsas_target_t                         *ptgt;
6795            mptsas_hash_table_t                     *tgttbl;
6796            uint8_t                                 num_entries, i, reason;
6797            uint16_t                                volhandle, diskhandle;

6799            irChangeList = (pMpi2EventDataIrConfigChangeList_t)
6800                eventreply->EventData;
6801            num_entries = ddi_get8(mpt->m_acc_reply_frame_hdl,
6802                &irChangeList->NumElements);

6804            tgttbl = &mpt->m_active->m_tgttbl;

6806            NDBG20(("mptsas%d IR_CONFIGURATION_CHANGE_LIST event received",
6807                mpt->m_instance));

6809            for (i = 0; i < num_entries; i++) {
6810                    reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
6811                        &irChangeList->ConfigElement[i].ReasonCode);
6812                    volhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6813                        &irChangeList->ConfigElement[i].VolDevHandle);
6814                    diskhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
6815                        &irChangeList->ConfigElement[i].PhysDiskDevHandle);

6817                    switch (reason) {
6818                    case MPI2_EVENT_IR_CHANGE_RC_ADDED:
6819                    case MPI2_EVENT_IR_CHANGE_RC_VOLUME_CREATED:
```



```
6820                    {
6821                            NDBG20(("mptsas %d volume added\n",
6822                                mpt->m_instance));

6824                            topo_node = kmem_zalloc(
6825                                sizeof (mptsas_topo_change_list_t),
6826                                KM_SLEEP);

6828                            topo_node->mpt = mpt;
6829                            topo_node->event =
6830                                MPTSAS_DR_EVENT_RECONFIG_TARGET;
6831                            topo_node->un.physport = 0xff;
6832                            topo_node->devhdl = volhandle;
6833                            topo_node->flags =
6834                                MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
6835                            topo_node->object = NULL;
6836                            if (topo_head == NULL) {
6837                                    topo_head = topo_tail = topo_node;
6838                            } else {
6839                                    topo_tail->next = topo_node;
6840                                    topo_tail = topo_node;
6841                            }
6842                            break;
6843                    }
6844                    case MPI2_EVENT_IR_CHANGE_RC_REMOVED:
6845                    case MPI2_EVENT_IR_CHANGE_RC_VOLUME_DELETED:
6846                    {
6847                            NDBG20(("mptsas %d volume deleted\n",
6848                                mpt->m_instance));
6849                            ptgt = mptsas_search_by_devhdl(tgttbl,
6850                                volhandle);
6851                            if (ptgt == NULL)
6852                                    break;

6854                            /*
6855                             * Clear any flags related to volume
6856                             */
6857                            (void) mptsas_delete_volume(mpt, volhandle);

6859                            /*
6860                             * Update DR flag immediately avoid I/O failure
6861                             */
6862                            mutex_enter(&mpt->m_tx_waitq_mutex);
7258                            mutex_enter(&ptgt->m_tgt_intr_mutex);
6863                            ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6864                            mutex_exit(&mpt->m_tx_waitq_mutex);
7260                            mutex_exit(&ptgt->m_tgt_intr_mutex);

6866                            topo_node = kmem_zalloc(
6867                                sizeof (mptsas_topo_change_list_t),
6868                                KM_SLEEP);
6869                            topo_node->mpt = mpt;
6870                            topo_node->un.phymask = ptgt->m_phymask;
6871                            topo_node->event =
6872                                MPTSAS_DR_EVENT_OFFLINE_TARGET;
6873                            topo_node->devhdl = volhandle;
6874                            topo_node->flags =
6875                                MPTSAS_TOPO_FLAG_RAID_ASSOCIATED;
6876                            topo_node->object = (void *)ptgt;
6877                            if (topo_head == NULL) {
6878                                    topo_head = topo_tail = topo_node;
6879                            } else {
6880                                    topo_tail->next = topo_node;
6881                                    topo_tail = topo_node;
6882                            }
6883                            break;
```

```
6884                        }
6885                        case MPI2_EVENT_IR_CHANGE_RC_PD_CREATED:
6886                        case MPI2_EVENT_IR_CHANGE_RC_HIDE:
6887                        {
6888                                ptgt = mptsas_search_by_devhdl(tgttbl,
6889                                    diskhandle);
6890                                if (ptgt == NULL)
6891                                        break;

6893                                /*
6894                                 * Update DR flag immediately avoid I/O failure
6895                                 */
6896                                mutex_enter(&mpt->m_tx_waitq_mutex);
7292                                mutex_enter(&ptgt->m_tgt_intr_mutex);
6897                                ptgt->m_dr_flag = MPTSAS_DR_INTRANSITION;
6898                                mutex_exit(&mpt->m_tx_waitq_mutex);
7294                                mutex_exit(&ptgt->m_tgt_intr_mutex);

6900                                topo_node = kmem_zalloc(
6901                                    sizeof (mptsas_topo_change_list_t),
6902                                    KM_SLEEP);
6903                                topo_node->mpt = mpt;
6904                                topo_node->un.phymask = ptgt->m_phymask;
6905                                topo_node->event =
6906                                    MPTSAS_DR_EVENT_OFFLINE_TARGET;
6907                                topo_node->devhdl = diskhandle;
6908                                topo_node->flags =
6909                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
6910                                topo_node->object = (void *)ptgt;
6911                                if (topo_head == NULL) {
6912                                        topo_head = topo_tail = topo_node;
6913                                } else {
6914                                        topo_tail->next = topo_node;
6915                                        topo_tail = topo_node;
6916                                }
6917                                break;
6918                        }
6919                        case MPI2_EVENT_IR_CHANGE_RC_UNHIDE:
6920                        case MPI2_EVENT_IR_CHANGE_RC_PD_DELETED:
6921                        {
6922                                /*
6923                                 * The physical drive is released by a IR
6924                                 * volume. But we cannot get the the physport
6925                                 * or phynum from the event data, so we only
6926                                 * can get the physport/phynum after SAS
6927                                 * Device Page0 request for the devhdl.
6928                                 */
6929                                topo_node = kmem_zalloc(
6930                                    sizeof (mptsas_topo_change_list_t),
6931                                    KM_SLEEP);
6932                                topo_node->mpt = mpt;
6933                                topo_node->un.phymask = 0;
6934                                topo_node->event =
6935                                    MPTSAS_DR_EVENT_RECONFIG_TARGET;
6936                                topo_node->devhdl = diskhandle;
6937                                topo_node->flags =
6938                                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED;
6939                                topo_node->object = NULL;
6940                                mpt->m_port_chng = 1;
6941                                if (topo_head == NULL) {
6942                                        topo_head = topo_tail = topo_node;
6943                                } else {
6944                                        topo_tail->next = topo_node;
6945                                        topo_tail = topo_node;
6946                                }
6947                                break;
```



```
6948                        }
6949                        default:
6950                                break;
6951                        }
6952                }

6954                if (topo_head != NULL) {
6955                        /*
6956                         * Launch DR taskq to handle topology change
6957                         */
6958                        if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
6959                            mptsas_handle_dr, (void *)topo_head,
6960                            DDI_NOSLEEP)) != DDI_SUCCESS) {
6961                                mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
6962                                    "for handle SAS DR event failed. \n");
6963                        }
6964                }
6965                break;
6966        }
6967        default:
6968                return (DDI_FAILURE);
6969        }

6971        return (DDI_SUCCESS);
6972 }
_____unchanged_portion_omitted_

7537 /*
7538  * invoked from timeout() to restart qfull cmds with throttle == 0
7539  */
7540 static void
7541 mptsas_restart_cmd(void *arg)
7542 {
7543        mptsas_t        *mpt = arg;
7544        mptsas_target_t *ptgt = NULL;

7546        mutex_enter(&mpt->m_mutex);

7548        mpt->m_restart_cmd_timeid = 0;

7550        ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
7551            MPTSAS_HASH_FIRST);
7552        while (ptgt != NULL) {
7949                mutex_enter(&ptgt->m_tgt_intr_mutex);
7553                if (ptgt->m_reset_delay == 0) {
7554                        if (ptgt->m_t_throttle == QFULL_THROTTLE) {
7555                                mptsas_set_throttle(mpt, ptgt,
7556                                    MAX_THROTTLE);
7557                        }
7558                }
7956                mutex_exit(&ptgt->m_tgt_intr_mutex);

7560                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
7561                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
7562        }
7563        mptsas_restart_hba(mpt);
7564        mutex_exit(&mpt->m_mutex);
7565 }

7965 /*
7966  * mptsas_remove_cmd0 is similar to mptsas_remove_cmd except that it is called
7967  * where m_intr_mutex has already been held.
7968  */
7567 void
7568 mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
7569 {
```

```
7972         ASSERT(mutex_owned(&mpt->m_mutex));

7974         /*
7975          * With new fine-grained lock mechanism, the outstanding cmd is only
7976          * linked to m_active before the dma is triggerred(MPTSAS_START_CMD)
7977          * to send it. that is, mptsas_save_cmd() doesn't link the outstanding
7978          * cmd now. So when mptsas_remove_cmd is called, a mptsas_save_cmd must
7979          * have been called, but the cmd may have not been linked.
7980          * For mptsas_remove_cmd0, the cmd must have been linked.
7981          * In order to keep the same semantic, we link the cmd to the
7982          * outstanding cmd list.
7983          */
7984         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;

7986         mutex_enter(&mpt->m_intr_mutex);
7987         mptsas_remove_cmd0(mpt, cmd);
7988         mutex_exit(&mpt->m_intr_mutex);
7989 }

7991 static inline void
7992 mptsas_remove_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd)
7993 {
7570         int             slot;
7571         mptsas_slots_t  *slots = mpt->m_active;
7572         int             t;
7573         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
7998         mptsas_slot_free_e_t    *pe;

7575         ASSERT(cmd != NULL);
7576         ASSERT(cmd->cmd_queued == FALSE);

7578         /*
7579          * Task Management cmds are removed in their own routines.  Also,
7580          * we don't want to modify timeout based on TM cmds.
7581          */
7582         if (cmd->cmd_flags & CFLAG_TM_CMD) {
7583                 return;
7584         }

7586         t = Tgt(cmd);
7587         slot = cmd->cmd_slot;
8013         pe = mpt->m_slot_free_ae + slot - 1;
8014         ASSERT(cmd == slots->m_slot[slot]);
8015         ASSERT((slot > 0) && slot < (mpt->m_max_requests - 1));

7589         /*
7590          * remove the cmd.
7591          */
7592         if (cmd == slots->m_slot[slot]) {
7593                 NDBG31(("mptsas_remove_cmd: removing cmd=0x%p", (void *)cmd));
8020         mutex_enter(&mpt->m_slot_freeq_pairp[pe->cpuid].
8021             m_slot_releq.s.m_fq_mutex);
8022         NDBG31(("mptsas_remove_cmd0: removing cmd=0x%p", (void *)cmd));
7594                 slots->m_slot[slot] = NULL;
7595                 mpt->m_ncmds--;
8024         ASSERT(pe->slot == slot);
8025         list_insert_tail(&mpt->m_slot_freeq_pairp[pe->cpuid].
8026             m_slot_releq.s.m_fq_list, pe);
8027         mpt->m_slot_freeq_pairp[pe->cpuid].m_slot_releq.s.m_fq_n++;
8028         ASSERT(mpt->m_slot_freeq_pairp[pe->cpuid].
8029             m_slot_releq.s.m_fq_n <= mpt->m_max_requests - 2);
8030         mutex_exit(&mpt->m_slot_freeq_pairp[pe->cpuid].
8031             m_slot_releq.s.m_fq_mutex);

7597                 /*
7598                  * only decrement per target ncmds if command
```



```
7599                  * has a target associated with it.
7600                  */
7601                 if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
8038                 mutex_enter(&ptgt->m_tgt_intr_mutex);
7602                         ptgt->m_t_ncmds--;
7603                         /*
7604                          * reset throttle if we just ran an untagged command
7605                          * to a tagged target
7606                          */
7607                         if ((ptgt->m_t_ncmds == 0) &&
7608                             ((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0)) {
7609                                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
7610                         }
8048                 mutex_exit(&ptgt->m_tgt_intr_mutex);
7611                 }

7613         }

7615         /*
7616          * This is all we need to do for ioc commands.
8053          * The ioc cmds would never be handled in fastpath in ISR, so we make
8054          * sure the mptsas_return_to_pool() would always be called with
8055          * m_mutex protected.
7617          */
7618         if (cmd->cmd_flags & CFLAG_CMDIOC) {
8058                 ASSERT(mutex_owned(&mpt->m_mutex));
7619                 mptsas_return_to_pool(mpt, cmd);
7620                 return;
7621         }

7623         /*
7624          * Figure out what to set tag Q timeout for...
7625          *
7626          * Optimize: If we have duplicate's of same timeout
7627          * we're using, then we'll use it again until we run
7628          * out of duplicates.  This should be the normal case
7629          * for block and raw I/O.
7630          * If no duplicates, we have to scan through tag que and
7631          * find the longest timeout value and use it.  This is
7632          * going to take a while...
7633          * Add 1 to m_n_slots to account for TM request.
7634          */
8075         mutex_enter(&ptgt->m_tgt_intr_mutex);
7635         if (cmd->cmd_pkt->pkt_time == ptgt->m_timebase) {
7636                 if (--(ptgt->m_dups) == 0) {
7637                         if (ptgt->m_t_ncmds) {
7638                                 mptsas_cmd_t *ssp;
7639                                 uint_t n = 0;
7640                                 ushort_t nslots = (slots->m_n_slots + 1);
7641                                 ushort_t i;
7642                                 /*
7643                                  * This crude check assumes we don't do
7644                                  * this too often which seems reasonable
7645                                  * for block and raw I/O.
7646                                  */
7647                                 for (i = 0; i < nslots; i++) {
7648                                         ssp = slots->m_slot[i];
7649                                         if (ssp && (Tgt(ssp) == t) &&
7650                                             (ssp->cmd_pkt->pkt_time > n)) {
7651                                                 n = ssp->cmd_pkt->pkt_time;
7652                                                 ptgt->m_dups = 1;
7653                                         } else if (ssp && (Tgt(ssp) == t) &&
7654                                             (ssp->cmd_pkt->pkt_time == n)) {
7655                                                 ptgt->m_dups++;
7656                                         }
7657                                 }
```

```
7658                                ptgt->m_timebase = n;
7659                        } else {
7660                                ptgt->m_dups = 0;
7661                                ptgt->m_timebase = 0;
7662                        }
7663                }
7664        }
7665        ptgt->m_timeout = ptgt->m_timebase;

7667        ASSERT(cmd != slots->m_slot[cmd->cmd_slot]);
8109        mutex_exit(&ptgt->m_tgt_intr_mutex);
7668 }

7670 /*
7671  * accept all cmds on the tx_waitq if any and then
7672  * start a fresh request from the top of the device queue.
7673  *
7674  * since there are always cmds queued on the tx_waitq, and rare cmds on
7675  * the instance waitq, so this function should not be invoked in the ISR,
7676  * the mptsas_restart_waitq() is invoked in the ISR instead. otherwise, the
7677  * burden belongs to the IO dispatch CPUs is moved the interrupt CPU.
7678  */
7679 static void
7680 mptsas_restart_hba(mptsas_t *mpt)
7681 {
7682        ASSERT(mutex_owned(&mpt->m_mutex));

7684        mutex_enter(&mpt->m_tx_waitq_mutex);
7685        if (mpt->m_tx_waitq) {
7686                mptsas_accept_tx_waitq(mpt);
7687        }
7688        mutex_exit(&mpt->m_tx_waitq_mutex);
7689        mptsas_restart_waitq(mpt);
7690 }

7692 /*
7693  * start a fresh request from the top of the device queue
7694  */
7695 static void
7696 mptsas_restart_waitq(mptsas_t *mpt)
7697 {
7698        mptsas_cmd_t    *cmd, *next_cmd;
7699        mptsas_target_t *ptgt = NULL;

7701        NDBG1(("mptsas_restart_waitq: mpt=0x%p", (void *)mpt));
8121        NDBG1(("mptsas_restart_hba: mpt=0x%p", (void *)mpt));

7703        ASSERT(mutex_owned(&mpt->m_mutex));

7705        /*
7706         * If there is a reset delay, don't start any cmds.  Otherwise, start
7707         * as many cmds as possible.
7708         * Since SMID 0 is reserved and the TM slot is reserved, the actual max
7709         * commands is m_max_requests - 2.
7710         */
7711        cmd = mpt->m_waitq;

7713        while (cmd != NULL) {
7714                next_cmd = cmd->cmd_linkp;
7715                if (cmd->cmd_flags & CFLAG_PASSTHRU) {
7716                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7717                                /*
7718                                 * passthru command get slot need
7719                                 * set CFLAG_PREPARED.
7720                                 */
7721                                cmd->cmd_flags |= CFLAG_PREPARED;
```



```
7722                                mptsas_waitq_delete(mpt, cmd);
7723                                mptsas_start_passthru(mpt, cmd);
7724                        }
7725                        cmd = next_cmd;
7726                        continue;
7727                }
7728                if (cmd->cmd_flags & CFLAG_CONFIG) {
7729                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7730                                /*
7731                                 * Send the config page request and delete it
7732                                 * from the waitq.
7733                                 */
7734                                cmd->cmd_flags |= CFLAG_PREPARED;
7735                                mptsas_waitq_delete(mpt, cmd);
7736                                mptsas_start_config_page_access(mpt, cmd);
7737                        }
7738                        cmd = next_cmd;
7739                        continue;
7740                }
7741                if (cmd->cmd_flags & CFLAG_FW_DIAG) {
7742                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7743                                /*
7744                                 * Send the FW Diag request and delete if from
7745                                 * the waitq.
7746                                 */
7747                                cmd->cmd_flags |= CFLAG_PREPARED;
7748                                mptsas_waitq_delete(mpt, cmd);
7749                                mptsas_start_diag(mpt, cmd);
7750                        }
7751                        cmd = next_cmd;
7752                        continue;
7753                }

7755                ptgt = cmd->cmd_tgt_addr;
7756                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) &&
7757                    (ptgt->m_t_ncmds == 0)) {
7758                        mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
7759                }
7760                if ((mpt->m_ncmds <= (mpt->m_max_requests - 2)) &&
7761                    (ptgt && (ptgt->m_reset_delay == 0)) &&
7762                    (ptgt && (ptgt->m_t_ncmds <
7763                    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);
7764                        if (mptsas_save_cmd(mpt, cmd) == TRUE) {
7765                                mptsas_waitq_delete(mpt, cmd);
7766                                (void) mptsas_start_cmd(mpt, cmd);
7767                        }
8191                                goto out;
7768                }
8193                        mutex_exit(&ptgt->m_tgt_intr_mutex);
8194                        mutex_exit(&mpt->m_intr_mutex);
8195                }
8196 out:
7769                cmd = next_cmd;
7770        }
7771 }

7772 /*
7773  * Cmds are queued if tran_start() doesn't get the m_mutexlock(no wait).
```

```
 * Accept all those queued cmds before new cmd is accept so that the
 * cmds are sent in order.
 * mpt tag type lookup
 */
static void
mptsas_accept_tx_waitq(mptsas_t *mpt)
static char mptsas_tag_lookup[] =
        {0, MSG_HEAD_QTAG, MSG_ORDERED_QTAG, 0, MSG_SIMPLE_QTAG};

/*
 * mptsas_start_cmd0 is similar to mptsas_start_cmd, except that, it is called
 * without ANY mutex protected, while, mptsas_start_cmd is called with m_mutex
 * protected.
 *
 * the relevant field in ptgt should be protected by m_tgt_intr_mutex in both
 * functions.
 *
 * before the cmds are linked on the slot for monitor as outstanding cmds, they
 * are accessed as slab objects, so slab framework ensures the exclusive access,
 * and no other mutex is requireed. Linking for monitor and the trigger of dma
 * must be done exclusively.
 */
static int
mptsas_start_cmd0(mptsas_t *mpt, mptsas_cmd_t *cmd)
{
        mptsas_cmd_t *cmd;
        struct scsi_pkt         *pkt = CMD2PKT(cmd);
        uint32_t                control = 0;
        int                     n;
        caddr_t                 mem;
        pMpi2SCSIIORequest_t    io_request;
        ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
        ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;
        mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
        uint16_t                SMID, io_flags = 0;
        uint32_t                request_desc_low, request_desc_high;

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

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

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

        /*
         * if a non-tagged cmd is submitted to an active tagged target
```



```
         * then drain before submitting this cmd; SCSI-2 allows RQSENSE
         * to be untagged
         */
        mutex_enter(&ptgt->m_tgt_intr_mutex);
        if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
            (ptgt->m_t_ncmds > 1) &&
            ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
            (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
                if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
                        NDBG23(("target=%d, untagged cmd, start draining\n",
                            ptgt->m_devhdl));

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

                        mutex_enter(&mpt->m_mutex);
                        mptsas_remove_cmd(mpt, cmd);
                        cmd->cmd_pkt_flags |= FLAG_HEAD;
                        mptsas_waitq_add(mpt, cmd);
                        mutex_exit(&mpt->m_mutex);
                        return (DDI_FAILURE);
                }
                mutex_exit(&ptgt->m_tgt_intr_mutex);
                return (DDI_FAILURE);
        }
        mutex_exit(&ptgt->m_tgt_intr_mutex);

        /*
         * Set correct tag bits.
         */
        if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
                switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
                    FLAG_TAGMASK) >> 12)]) {
                case MSG_SIMPLE_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
                        break;
                case MSG_HEAD_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
                        break;
                case MSG_ORDERED_QTAG:
                        control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
                        break;
                default:
                        mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
                        break;
                }
                if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL) {
                        mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
        } else {
                if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
                        ptgt->m_t_throttle = 1;
                }
                cmd->cmd_linkp = NULL;
                mutex_exit(&mpt->m_tx_waitq_mutex);
                if (mptsas_accept_pkt(mpt, cmd) != TRAN_ACCEPT)
                        cmn_err(CE_WARN, "mpt: mptsas_accept_tx_waitq: failed "
                            "to accept cmd on queue\n");
                mutex_enter(&mpt->m_tx_waitq_mutex);
                control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
        }
}

        if (cmd->cmd_pkt_flags & FLAG_TLR) {
                control |= MPI2_SCSIIO_CONTROL_TLR_ON;
```

```
8310         }

7810 /*
7811  * 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
7812  */
7813 static char mptsas_tag_lookup[] =
7814         {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 }

7816 static int
7817 mptsas_start_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd)
7818 {
7819         struct scsi_pkt         *pkt = CMD2PKT(cmd);
7820         uint32_t                control = 0;
7821         int                     n;
7822         caddr_t                 mem;
7823         pMpi2SCSIIORequest_t    io_request;
7824         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
7825         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;
7826         mptsas_target_t         *ptgt = cmd->cmd_tgt_addr;
7827         uint16_t                SMID, io_flags = 0;
7828         uint32_t                request_desc_low, request_desc_high;

7830         NDBG1(("mptsas_start_cmd: cmd=0x%p", (void *)cmd));

7832         /*
7833          * Set SMID and increment index.  Rollover to 1 instead of 0 if index
7834          * is at the max.  0 is an invalid SMID, so we call the first index 1.
7835          */
7836         SMID = cmd->cmd_slot;

7838         /*
```

```
7839          * It is possible for back to back device reset to
7840          * happen before the reset delay has expired.  That's
7841          * ok, just let the device reset go out on the bus.
7842          */
7843         if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
7844                 ASSERT(ptgt->m_reset_delay == 0);
7845         }
7847         /*
7848          * if a non-tagged cmd is submitted to an active tagged target
7849          * then drain before submitting this cmd; SCSI-2 allows RQSENSE
7850          * to be untagged
7851          */
8451         mutex_enter(&ptgt->m_tgt_intr_mutex);
7852         if (((cmd->cmd_pkt_flags & FLAG_TAGMASK) == 0) &&
7853             (ptgt->m_t_ncmds > 1) &&
7854             ((cmd->cmd_flags & CFLAG_TM_CMD) == 0) &&
7855             (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE)) {
7856                 if ((cmd->cmd_pkt_flags & FLAG_NOINTR) == 0) {
7857                         NDBG23(("target=%d, untagged cmd, start draining\n",
7858                             ptgt->m_devhdl));
7860                         if (ptgt->m_reset_delay == 0) {
7861                                 mptsas_set_throttle(mpt, ptgt, DRAIN_THROTTLE);
7862                         }
8463                         mutex_exit(&ptgt->m_tgt_intr_mutex);
7864                         mptsas_remove_cmd(mpt, cmd);
7865                         cmd->cmd_pkt_flags |= FLAG_HEAD;
7866                         mptsas_waitq_add(mpt, cmd);
8468                         return (DDI_FAILURE);
7867                 }
8470                 mutex_exit(&ptgt->m_tgt_intr_mutex);
7868                 return (DDI_FAILURE);
7869         }
8473         mutex_exit(&ptgt->m_tgt_intr_mutex);
7871         /*
7872          * Set correct tag bits.
7873          */
7874         if (cmd->cmd_pkt_flags & FLAG_TAGMASK) {
7875                 switch (mptsas_tag_lookup[((cmd->cmd_pkt_flags &
7876                     FLAG_TAGMASK) >> 12)]) {
7877                 case MSG_SIMPLE_QTAG:
7878                         control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
7879                         break;
7880                 case MSG_HEAD_QTAG:
7881                         control |= MPI2_SCSIIO_CONTROL_HEADOFQ;
7882                         break;
7883                 case MSG_ORDERED_QTAG:
7884                         control |= MPI2_SCSIIO_CONTROL_ORDEREDQ;
7885                         break;
7886                 default:
7887                         mptsas_log(mpt, CE_WARN, "mpt: Invalid tag type\n");
7888                         break;
7889                 }
7890         } else {
7891                 if (*(cmd->cmd_pkt->pkt_cdbp) != SCMD_REQUEST_SENSE) {
7892                         ptgt->m_t_throttle = 1;
7893                 }
7894                 control |= MPI2_SCSIIO_CONTROL_SIMPLEQ;
7895         }
7897         if (cmd->cmd_pkt_flags & FLAG_TLR) {
7898                 control |= MPI2_SCSIIO_CONTROL_TLR_ON;
7899         }
```



```
7901         mem = mpt->m_req_frame + (mpt->m_req_frame_size * SMID);
7902         io_request = (pMpi2SCSIIORequest_t)mem;
7904         bzero(io_request, sizeof (Mpi2SCSIIORequest_t));
7905         ddi_put8(acc_hdl, &io_request->SGLOffset0, offsetof
7906             (MPI2_SCSI_IO_REQUEST, SGL) / 4);
7907         mptsas_init_std_hdr(acc_hdl, io_request, ptgt->m_devhdl, Lun(cmd), 0,
7908             MPI2_FUNCTION_SCSI_IO_REQUEST);
7910         (void) ddi_rep_put8(acc_hdl, (uint8_t *)pkt->pkt_cdbp,
7911             io_request->CDB.CDB32, cmd->cmd_cdblen, DDI_DEV_AUTOINCR);
7913         io_flags = cmd->cmd_cdblen;
7914         ddi_put16(acc_hdl, &io_request->IoFlags, io_flags);
7915         /*
7916          * setup the Scatter/Gather DMA list for this request
7917          */
7918         if (cmd->cmd_cookiec > 0) {
7919                 mptsas_sge_setup(mpt, cmd, &control, io_request, acc_hdl);
7920         } else {
7921                 ddi_put32(acc_hdl, &io_request->SGL.MpiSimple.FlagsLength,
7922                     ((uint32_t)MPI2_SGE_FLAGS_LAST_ELEMENT |
7923                     MPI2_SGE_FLAGS_END_OF_BUFFER |
7924                     MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
7925                     MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
7926         }
7928         /*
7929          * save ARQ information
7930          */
7931         ddi_put8(acc_hdl, &io_request->SenseBufferLength, cmd->cmd_rqslen);
7932         if ((cmd->cmd_flags & (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) ==
7933             (CFLAG_SCBEXTERN | CFLAG_EXTARQBUFVALID)) {
7934                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
7935                     cmd->cmd_ext_arqcookie.dmac_address);
7936         } else {
7937                 ddi_put32(acc_hdl, &io_request->SenseBufferLowAddress,
7938                     cmd->cmd_arqcookie.dmac_address);
7939         }
7941         ddi_put32(acc_hdl, &io_request->Control, control);
7943         NDBG31(("starting message=0x%p, with cmd=0x%p",
7944             (void *)(uintptr_t)mpt->m_req_frame_dma_addr, (void *)cmd));
7946         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
7948         /*
7949          * Build request descriptor and write it to the request desc post reg.
7950          */
7951         request_desc_low = (SMID << 16) + MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
7952         request_desc_high = ptgt->m_devhdl << 16;
8558         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
7953         MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
7955         /*
7956          * Start timeout.
7957          */
8564         mutex_enter(&ptgt->m_tgt_intr_mutex);
7958 #ifdef MPTSAS_TEST
7959         /*
7960          * Temporarily set timebase = 0;  needed for
7961          * timeout torture test.
7962          */
```

```
7963         if (mptsas_test_timeouts) {
7964                 ptgt->m_timebase = 0;
7965         }
7966 #endif
7967         n = pkt->pkt_time - ptgt->m_timebase;

7969         if (n == 0) {
7970                 (ptgt->m_dups)++;
7971                 ptgt->m_timeout = ptgt->m_timebase;
7972         } else if (n > 0) {
7973                 ptgt->m_timeout =
7974                     ptgt->m_timebase = pkt->pkt_time;
7975                 ptgt->m_dups = 1;
7976         } else if (n < 0) {
7977                 ptgt->m_timeout = ptgt->m_timebase;
7978         }
7979 #ifdef MPTSAS_TEST
7980         /*
7981          * Set back to a number higher than
7982          * mptsas_scsi_watchdog_tick
7983          * so timeouts will happen in mptsas_watchsubr
7984          */
7985         if (mptsas_test_timeouts) {
7986                 ptgt->m_timebase = 60;
7987         }
7988 #endif
8596         mutex_exit(&ptgt->m_tgt_intr_mutex);

7990         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
7991             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
7992                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
7993                 return (DDI_FAILURE);
7994         }
7995         return (DDI_SUCCESS);
7996 }
```

_____***unchanged_portion_omitted_***

```
8034 /*
8035  * move the current global doneq to the doneq of thead[t]
8643  * move the current global doneq to the doneq of thread[t]
8036  */
8037 static void
8038 mptsas_doneq_mv(mptsas_t *mpt, uint64_t t)
8039 {
8040         mptsas_cmd_t                    *cmd;
8041         mptsas_doneq_thread_list_t      *item = &mpt->m_doneq_thread_id[t];

8043         ASSERT(mutex_owned(&item->mutex));
8652         mutex_enter(&mpt->m_intr_mutex);
8044         while ((cmd = mpt->m_doneq) != NULL) {
8045                 if ((mpt->m_doneq = cmd->cmd_linkp) == NULL) {
8046                         mpt->m_donetail = &mpt->m_doneq;
8047                 }
8048                 cmd->cmd_linkp = NULL;
8049                 *item->donetail = cmd;
8050                 item->donetail = &cmd->cmd_linkp;
8051                 mpt->m_doneq_len--;
8052                 item->len++;
8053         }
8663         mutex_exit(&mpt->m_intr_mutex);
8054 }
```

_____***unchanged_portion_omitted_***

```
8127 /*
8128  * These routines manipulate the queue of commands that
8129  * are waiting for their completion routines to be called.
```



```
8130  * The queue is usually in FIFO order but on an MP system
8131  * it's possible for the completion routines to get out
8132  * of order. If that's a problem you need to add a global
8133  * mutex around the code that calls the completion routine
8134  * 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.
8135  */
8136 static void
8137 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)
8138 {
8139         struct scsi_pkt *pkt = CMD2PKT(cmd);

8141         NDBG31(("mptsas_doneq_add: cmd=0x%p", (void *)cmd));
8746         NDBG31(("mptsas_doneq_add0: cmd=0x%p", (void *)cmd));

8143         ASSERT((cmd->cmd_flags & CFLAG_COMPLETED) == 0);
8144         cmd->cmd_linkp = NULL;
8145         cmd->cmd_flags |= CFLAG_FINISHED;
8146         cmd->cmd_flags &= ~CFLAG_IN_TRANSPORT;

8148         mptsas_fma_check(mpt, cmd);

8150         /*
8151          * only add scsi pkts that have completion routines to
8152          * the doneq.  no intr cmds do not have callbacks.
8153          */
8154         if (pkt && (pkt->pkt_comp)) {
8155                 *mpt->m_donetail = cmd;
8156                 mpt->m_donetail = &cmd->cmd_linkp;
8157                 mpt->m_doneq_len++;
8158         }
8159 }

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 }

8161 static mptsas_cmd_t *
8162 mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t)
8163 {
8164         mptsas_cmd_t                    *cmd;
8165         mptsas_doneq_thread_list_t      *item = &mpt->m_doneq_thread_id[t];

8167         /* pop one off the done queue */
8168         if ((cmd = item->doneq) != NULL) {
8169                 /* if the queue is now empty fix the tail pointer */
```

```
8170                 NDBG31(("mptsas_doneq_thread_rm: cmd=0x%p", (void *)cmd));
8171                 if ((item->doneq = cmd->cmd_linkp) == NULL) {
8172                         item->donetail = &item->doneq;
8173                 }
8174                 cmd->cmd_linkp = NULL;
8175                 item->len--;
8176         }
8177         return (cmd);
8178 }

8180 static void
8181 mptsas_doneq_empty(mptsas_t *mpt)
8182 {
8807         mutex_enter(&mpt->m_intr_mutex);
8183         if (mpt->m_doneq && !mpt->m_in_callback) {
8184                 mptsas_cmd_t    *cmd, *next;
8185                 struct scsi_pkt *pkt;

8187                 mpt->m_in_callback = 1;
8188                 cmd = mpt->m_doneq;
8189                 mpt->m_doneq = NULL;
8190                 mpt->m_donetail = &mpt->m_doneq;
8191                 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)
8193                 mutex_exit(&mpt->m_mutex);
8194                 /*
8195                  * run the completion routines of all the
8196                  * completed commands
8197                  */
8198                 while (cmd != NULL) {
8199                         next = cmd->cmd_linkp;
8200                         cmd->cmd_linkp = NULL;
8201                         /* run this command's completion routine */
8202                         cmd->cmd_flags |= CFLAG_COMPLETED;
8203                         pkt = CMD2PKT(cmd);
8204                         mptsas_pkt_comp(pkt, cmd);
8205                         cmd = next;
8206                 }
8839                 if ((curthread->t_flag & T_INTR_THREAD) == 0)
8207                 mutex_enter(&mpt->m_mutex);
8208                 mpt->m_in_callback = 0;
8842                 return;
8209         }
8844         mutex_exit(&mpt->m_intr_mutex);
8210 }

8212 /*
8213  * These routines manipulate the target's queue of pending requests
8214  */
8215 void
8216 mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd)
8217 {
8218         NDBG7(("mptsas_waitq_add: cmd=0x%p", (void *)cmd));
8219         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;
8220         cmd->cmd_queued = TRUE;
8221         if (ptgt)
8222                 ptgt->m_t_nwait++;
8223         if (cmd->cmd_pkt_flags & FLAG_HEAD) {
8859                 mutex_enter(&mpt->m_intr_mutex);
```



```
8224                 if ((cmd->cmd_linkp = mpt->m_waitq) == NULL) {
8225                         mpt->m_waitqtail = &cmd->cmd_linkp;
8226                 }
8227                 mpt->m_waitq = cmd;
8864                 mutex_exit(&mpt->m_intr_mutex);
8228         } else {
8229                 cmd->cmd_linkp = NULL;
8230                 *(mpt->m_waitqtail) = cmd;
8231                 mpt->m_waitqtail = &cmd->cmd_linkp;
8232         }
8233 }

8235 static mptsas_cmd_t *
8236 mptsas_waitq_rm(mptsas_t *mpt)
8237 {
8238         mptsas_cmd_t    *cmd;
8239         mptsas_target_t *ptgt;
8240         NDBG7(("mptsas_waitq_rm"));

8879         mutex_enter(&mpt->m_intr_mutex);
8242         MPTSAS_WAITQ_RM(mpt, cmd);
8881         mutex_exit(&mpt->m_intr_mutex);

8244         NDBG7(("mptsas_waitq_rm: cmd=0x%p", (void *)cmd));
8245         if (cmd) {
8246                 ptgt = cmd->cmd_tgt_addr;
8247                 if (ptgt) {
8248                         ptgt->m_t_nwait--;
8249                         ASSERT(ptgt->m_t_nwait >= 0);
8250                 }
8251         }
8252         return (cmd);
8253 }

8255 /*
8256  * remove specified cmd from the middle of the wait queue.
8257  */
8258 static void
8259 mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8260 {
8261         mptsas_cmd_t    *prevp = mpt->m_waitq;
8262         mptsas_target_t *ptgt = cmd->cmd_tgt_addr;

8264         NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8265             (void *)mpt, (void *)cmd));
8266         if (ptgt) {
8267                 ptgt->m_t_nwait--;
8268                 ASSERT(ptgt->m_t_nwait >= 0);
8269         }

8271         if (prevp == cmd) {
8911                 mutex_enter(&mpt->m_intr_mutex);
8272                 if ((mpt->m_waitq = cmd->cmd_linkp) == NULL)
8273                         mpt->m_waitqtail = &mpt->m_waitq;
8914                 mutex_exit(&mpt->m_intr_mutex);

8275                 cmd->cmd_linkp = NULL;
8276                 cmd->cmd_queued = FALSE;
8277                 NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8278                     (void *)mpt, (void *)cmd));
8279                 return;
8280         }

8282         while (prevp != NULL) {
8283                 if (prevp->cmd_linkp == cmd) {
8284                         if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
```

```
8285                                mpt->m_waitqtail = &prevp->cmd_linkp;

8287                        cmd->cmd_linkp = NULL;
8288                        cmd->cmd_queued = FALSE;
8289                        NDBG7(("mptsas_waitq_delete: mpt=0x%p cmd=0x%p",
8290                            (void *)mpt, (void *)cmd));
8291                        return;
8292                }
8293                prevp = prevp->cmd_linkp;
8294        }
8295        cmn_err(CE_PANIC, "mpt: mptsas_waitq_delete: queue botch");
8296 }

8298 static mptsas_cmd_t *
8299 mptsas_tx_waitq_rm(mptsas_t *mpt)
8300 {
8301        mptsas_cmd_t *cmd;
8302        NDBG7(("mptsas_tx_waitq_rm"));

8304        MPTSAS_TX_WAITQ_RM(mpt, cmd);

8306        NDBG7(("mptsas_tx_waitq_rm: cmd=0x%p", (void *)cmd));

8308        return (cmd);
8309 }

8311 /*
8312  * remove specified cmd from the middle of the tx_waitq.
8313  */
8314 static void
8315 mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd)
8316 {
8317        mptsas_cmd_t *prevp = mpt->m_tx_waitq;

8319        NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8320            (void *)mpt, (void *)cmd));

8322        if (prevp == cmd) {
8323                if ((mpt->m_tx_waitq = cmd->cmd_linkp) == NULL)
8324                        mpt->m_tx_waitqtail = &mpt->m_tx_waitq;

8326                cmd->cmd_linkp = NULL;
8327                cmd->cmd_queued = FALSE;
8328                NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8329                    (void *)mpt, (void *)cmd));
8330                return;
8331        }

8333        while (prevp != NULL) {
8334                if (prevp->cmd_linkp == cmd) {
8335                        if ((prevp->cmd_linkp = cmd->cmd_linkp) == NULL)
8336                                mpt->m_tx_waitqtail = &prevp->cmd_linkp;

8338                        cmd->cmd_linkp = NULL;
8339                        cmd->cmd_queued = FALSE;
8340                        NDBG7(("mptsas_tx_waitq_delete: mpt=0x%p cmd=0x%p",
8341                            (void *)mpt, (void *)cmd));
8342                        return;
8343                }
8344                prevp = prevp->cmd_linkp;
8345        }
8346        cmn_err(CE_PANIC, "mpt: mptsas_tx_waitq_delete: queue botch");
8347 }

8349 /*
8350  * device and bus reset handling
```



```
8351  *
8352  * Notes:
8353  *      - RESET_ALL:    reset the controller
8354  *      - RESET_TARGET: reset the target specified in scsi_address
8355  */
8356 static int
8357 mptsas_scsi_reset(struct scsi_address *ap, int level)
8358 {
8359        mptsas_t               *mpt = ADDR2MPT(ap);
8360        int                    rval;
8361        mptsas_tgt_private_t    *tgt_private;
8362        mptsas_target_t         *ptgt = NULL;

8364        tgt_private = (mptsas_tgt_private_t *)ap->a_hba_tran->tran_tgt_private;
8365        ptgt = tgt_private->t_private;
8366        if (ptgt == NULL) {
8367                return (FALSE);
8368        }
8369        NDBG22(("mptsas_scsi_reset: target=%d level=%d", ptgt->m_devhdl,
8370            level));

8372        mutex_enter(&mpt->m_mutex);
8373        /*
8374         * if we are not in panic set up a reset delay for this target
8375         */
8376        if (!ddi_in_panic()) {
8377                mptsas_setup_bus_reset_delay(mpt);
8378        } else {
8379                drv_usecwait(mpt->m_scsi_reset_delay * 1000);
8380        }
8381        rval = mptsas_do_scsi_reset(mpt, ptgt->m_devhdl);
8382        mutex_exit(&mpt->m_mutex);

8384        /*
8385         * The transport layer expect to only see TRUE and
8386         * FALSE. Therefore, we will adjust the return value
8387         * if mptsas_do_scsi_reset returns FAILED.
8388         */
8389        if (rval == FAILED)
8390                rval = FALSE;
8391        return (rval);
8392 }
______unchanged_portion_omitted_

8489 /*
8490  * Clean up from a device reset.
8491  * For the case of target reset, this function clears the waitq of all
8492  * commands for a particular target.   For the case of abort task set, this
8493  * function clears the waitq of all commonds for a particular target/lun.
8494  */
8495 static void
8496 mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
8497 {
8498        mptsas_slots_t  *slots = mpt->m_active;
8499        mptsas_cmd_t    *cmd, *next_cmd;
8500        int             slot;
8501        uchar_t         reason;
8502        uint_t          stat;

8504        NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));

8506        /*
8507         * Make sure the I/O Controller has flushed all cmds
8508         * that are associated with this target for a target reset
8509         * and target/lun for abort task set.
8510         * Account for TM requests, which use the last SMID.
```

```
8511          */
9102         mutex_enter(&mpt->m_intr_mutex);
8512         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8513                 if ((cmd = slots->m_slot[slot]) == NULL)
9104                 if ((cmd = slots->m_slot[slot]) == NULL) {
8514                         continue;
9106                 }
8515                 reason = CMD_RESET;
8516                 stat = STAT_DEV_RESET;
8517                 switch (tasktype) {
8518                 case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8519                         if (Tgt(cmd) == target) {
8520                                 NDBG25(("mptsas_flush_target discovered non-"
8521                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8522                                     tasktype));
8523                                 mptsas_dump_cmd(mpt, cmd);
8524                                 mptsas_remove_cmd(mpt, cmd);
9116                                 mptsas_remove_cmd0(mpt, cmd);
8525                                 mptsas_set_pkt_reason(mpt, cmd, reason, stat);
8526                                 mptsas_doneq_add(mpt, cmd);
9118                                 mptsas_doneq_add0(mpt, cmd);
8527                         }
8528                         break;
8529                 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8530                         reason = CMD_ABORTED;
8531                         stat = STAT_ABORTED;
8532                         /*FALLTHROUGH*/
8533                 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8534                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {

8536                                 NDBG25(("mptsas_flush_target discovered non-"
8537                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8538                                     tasktype));
8539                                 mptsas_dump_cmd(mpt, cmd);
8540                                 mptsas_remove_cmd(mpt, cmd);
9132                                 mptsas_remove_cmd0(mpt, cmd);
8541                                 mptsas_set_pkt_reason(mpt, cmd, reason,
8542                                     stat);
8543                                 mptsas_doneq_add(mpt, cmd);
9135                                 mptsas_doneq_add0(mpt, cmd);
8544                         }
8545                         break;
8546                 default:
8547                         break;
8548                 }
8549         }
9142         mutex_exit(&mpt->m_intr_mutex);

8551         /*
8552          * Flush the waitq and tx_waitq of this target's cmds
9145          * Flush the waitq of this target's cmds
8553          */
8554         cmd = mpt->m_waitq;

8556         reason = CMD_RESET;
8557         stat = STAT_DEV_RESET;

8559         switch (tasktype) {
8560         case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8561                 while (cmd != NULL) {
8562                         next_cmd = cmd->cmd_linkp;
8563                         if (Tgt(cmd) == target) {
8564                                 mptsas_waitq_delete(mpt, cmd);
8565                                 mptsas_set_pkt_reason(mpt, cmd,
8566                                     reason, stat);
8567                                 mptsas_doneq_add(mpt, cmd);
```



```
8568                         }
8569                         cmd = next_cmd;
8570                 }
8571                 mutex_enter(&mpt->m_tx_waitq_mutex);
8572                 cmd = mpt->m_tx_waitq;
8573                 while (cmd != NULL) {
8574                         next_cmd = cmd->cmd_linkp;
8575                         if (Tgt(cmd) == target) {
8576                                 mptsas_tx_waitq_delete(mpt, cmd);
8577                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8578                                 mptsas_set_pkt_reason(mpt, cmd,
8579                                     reason, stat);
8580                                 mptsas_doneq_add(mpt, cmd);
8581                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8582                         }
8583                         cmd = next_cmd;
8584                 }
8585                 mutex_exit(&mpt->m_tx_waitq_mutex);
8586                 break;
8587         case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8588                 reason = CMD_ABORTED;
8589                 stat =  STAT_ABORTED;
8590                 /*FALLTHROUGH*/
8591         case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8592                 while (cmd != NULL) {
8593                         next_cmd = cmd->cmd_linkp;
8594                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8595                                 mptsas_waitq_delete(mpt, cmd);
8596                                 mptsas_set_pkt_reason(mpt, cmd,
8597                                     reason, stat);
8598                                 mptsas_doneq_add(mpt, cmd);
8599                         }
8600                         cmd = next_cmd;
8601                 }
8602                 mutex_enter(&mpt->m_tx_waitq_mutex);
8603                 cmd = mpt->m_tx_waitq;
8604                 while (cmd != NULL) {
8605                         next_cmd = cmd->cmd_linkp;
8606                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8607                                 mptsas_tx_waitq_delete(mpt, cmd);
8608                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8609                                 mptsas_set_pkt_reason(mpt, cmd,
8610                                     reason, stat);
8611                                 mptsas_doneq_add(mpt, cmd);
8612                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8613                         }
8614                         cmd = next_cmd;
8615                 }
8616                 mutex_exit(&mpt->m_tx_waitq_mutex);
8617                 break;
8618         default:
8619                 mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
8620                     tasktype);
8621                 break;
8622         }
8623 }

8625 /*
8626  * Clean up hba state, abort all outstanding command and commands in waitq
8627  * reset timeout of all targets.
8628  */
8629 static void
8630 mptsas_flush_hba(mptsas_t *mpt)
8631 {
8632         mptsas_slots_t  *slots = mpt->m_active;
8633         mptsas_cmd_t    *cmd;
```

```
8634         int             slot;

8636         NDBG25(("mptsas_flush_hba"));

8638         /*
8639          * The I/O Controller should have already sent back
8640          * all commands via the scsi I/O reply frame.  Make
8641          * sure all commands have been flushed.
8642          * Account for TM request, which use the last SMID.
8643          */
9207         mutex_enter(&mpt->m_intr_mutex);
8644         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8645                 if ((cmd = slots->m_slot[slot]) == NULL)
9209                 if ((cmd = slots->m_slot[slot]) == NULL) {
8646                         continue;
9211                 }

8648                 if (cmd->cmd_flags & CFLAG_CMDIOC) {
8649                         /*
8650                          * Need to make sure to tell everyone that might be
8651                          * waiting on this command that it's going to fail.  If
8652                          * we get here, this command will never timeout because
8653                          * the active command table is going to be re-allocated,
8654                          * so there will be nothing to check against a time out.
8655                          * Instead, mark the command as failed due to reset.
8656                          */
8657                         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
8658                             STAT_BUS_RESET);
8659                         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8660                             (cmd->cmd_flags & CFLAG_CONFIG) ||
8661                             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8662                                 cmd->cmd_flags |= CFLAG_FINISHED;
8663                                 cv_broadcast(&mpt->m_passthru_cv);
8664                                 cv_broadcast(&mpt->m_config_cv);
8665                                 cv_broadcast(&mpt->m_fw_diag_cv);
8666                         }
8667                         continue;
8668                 }

8670                 NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
8671                     slot));
8672                 mptsas_dump_cmd(mpt, cmd);

8674                 mptsas_remove_cmd(mpt, cmd);
9239                 mptsas_remove_cmd0(mpt, cmd);
8675                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8676                 mptsas_doneq_add(mpt, cmd);
9241                 mptsas_doneq_add0(mpt, cmd);
8677         }
9243         mutex_exit(&mpt->m_intr_mutex);

8679         /*
8680          * Flush the waitq.
8681          */
8682         while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
8683                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, 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                 } else {
8692                         mptsas_doneq_add(mpt, cmd);
8693                 }
```



```
8694         }

8696         /*
8697          * Flush the tx_waitq
8698          */
8699         mutex_enter(&mpt->m_tx_waitq_mutex);
8700         while ((cmd = mptsas_tx_waitq_rm(mpt)) != NULL) {
8701                 mutex_exit(&mpt->m_tx_waitq_mutex);
8702                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8703                 mptsas_doneq_add(mpt, cmd);
8704                 mutex_enter(&mpt->m_tx_waitq_mutex);
8705         }
8706         mutex_exit(&mpt->m_tx_waitq_mutex);
8707 }
______unchanged_portion_omitted_

8746 static void
8747 mptsas_setup_bus_reset_delay(mptsas_t *mpt)
8748 {
8749         mptsas_target_t *ptgt = NULL;

8751         NDBG22(("mptsas_setup_bus_reset_delay"));
8752         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
8753             MPTSAS_HASH_FIRST);
8754         while (ptgt != NULL) {
9309                 mutex_enter(&ptgt->m_tgt_intr_mutex);
8755                 mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
8756                 ptgt->m_reset_delay = mpt->m_scsi_reset_delay;
9312                 mutex_exit(&ptgt->m_tgt_intr_mutex);

8758                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
8759                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
8760         }

8762         mptsas_start_watch_reset_delay();
8763 }
______unchanged_portion_omitted_

8800 static int
8801 mptsas_watch_reset_delay_subr(mptsas_t *mpt)
8802 {
8803         int             done = 0;
8804         int             restart = 0;
8805         mptsas_target_t *ptgt = NULL;

8807         NDBG22(("mptsas_watch_reset_delay_subr: mpt=0x%p", (void *)mpt));

8809         ASSERT(mutex_owned(&mpt->m_mutex));

8811         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
8812             MPTSAS_HASH_FIRST);
8813         while (ptgt != NULL) {
9370                 mutex_enter(&ptgt->m_tgt_intr_mutex);
8814                 if (ptgt->m_reset_delay != 0) {
8815                         ptgt->m_reset_delay -=
8816                             MPTSAS_WATCH_RESET_DELAY_TICK;
8817                         if (ptgt->m_reset_delay <= 0) {
8818                                 ptgt->m_reset_delay = 0;
8819                                 mptsas_set_throttle(mpt, ptgt,
8820                                     MAX_THROTTLE);
8821                                 restart++;
8822                         } else {
8823                                 done = -1;
8824                         }
8825                 }
9383                 mutex_exit(&ptgt->m_tgt_intr_mutex);
```

```
8827                ptgt = (mptsas_target_t *)mptsas_hash_traverse(
8828                    &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
8829        }

8831        if (restart > 0) {
8832                mptsas_restart_hba(mpt);
8833        }
8834        return (done);
8835 }
______unchanged_portion_omitted_

8883 static int
8884 mptsas_do_scsi_abort(mptsas_t *mpt, int target, int lun, struct scsi_pkt *pkt)
8885 {
8886        mptsas_cmd_t    *sp = NULL;
8887        mptsas_slots_t  *slots = mpt->m_active;
8888        int             rval = FALSE;

8890        ASSERT(mutex_owned(&mpt->m_mutex));

8892        /*
8893         * Abort the command pkt on the target/lun in ap.  If pkt is
8894         * NULL, abort all outstanding commands on that target/lun.
8895         * If you can abort them, return 1, else return 0.
8896         * Each packet that's aborted should be sent back to the target
8897         * driver through the callback routine, with pkt_reason set to
8898         * CMD_ABORTED.
8899         *
8900         * abort cmd pkt on HBA hardware; clean out of outstanding
8901         * command lists, etc.
8902         */
8903        if (pkt != NULL) {
8904                /* abort the specified packet */
8905                sp = PKT2CMD(pkt);

8907                if (sp->cmd_queued) {
8908                        NDBG23(("mptsas_do_scsi_abort: queued sp=0x%p aborted",
8909                            (void *)sp));
8910                        mptsas_waitq_delete(mpt, sp);
8911                        mptsas_set_pkt_reason(mpt, sp, CMD_ABORTED,
8912                            STAT_ABORTED);
8913                        mptsas_doneq_add(mpt, sp);
8914                        rval = TRUE;
8915                        goto done;
8916                }

8918                /*
8919                 * Have mpt firmware abort this command
8920                 */

9479                mutex_enter(&mpt->m_intr_mutex);
8922                if (slots->m_slot[sp->cmd_slot] != NULL) {
9481                        mutex_exit(&mpt->m_intr_mutex);
8923                        rval = mptsas_ioc_task_management(mpt,
8924                            MPI2_SCSITASKMGMT_TASKTYPE_ABORT_TASK, target,
8925                            lun, NULL, 0, 0);

8927                        /*
8928                         * The transport layer expects only TRUE and FALSE.
8929                         * Therefore, if mptsas_ioc_task_management returns
8930                         * FAILED we will return FALSE.
8931                         */
8932                        if (rval == FAILED)
8933                                rval = FALSE;
8934                        goto done;
```



```
8935                }
9495                mutex_exit(&mpt->m_intr_mutex);
8936        }

8938        /*
8939         * If pkt is NULL then abort task set
8940         */
8941        rval = mptsas_ioc_task_management(mpt,
8942            MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET, target, lun, NULL, 0, 0);

8944        /*
8945         * The transport layer expects only TRUE and FALSE.
8946         * Therefore, if mptsas_ioc_task_management returns
8947         * FAILED we will return FALSE.
8948         */
8949        if (rval == FAILED)
8950                rval = FALSE;

8952 #ifdef MPTSAS_TEST
8953        if (rval && mptsas_test_stop) {
8954                debug_enter("mptsas_do_scsi_abort");
8955        }
8956 #endif

8958 done:
8959        mptsas_doneq_empty(mpt);
8960        return (rval);
8961 }
______unchanged_portion_omitted_

9038 /*
9039  * (*tran_setcap).  Set the capability named to the value given.
9040  */
9041 static int
9042 mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value, int tgtonly)
9043 {
9044        mptsas_t        *mpt = ADDR2MPT(ap);
9045        int             ckey;
9046        int             rval = FALSE;
9607        mptsas_target_t *ptgt;

9048        NDBG24(("mptsas_scsi_setcap: target=%d, cap=%s value=%x tgtonly=%x",
9049            ap->a_target, cap, value, tgtonly));

9051        if (!tgtonly) {
9052                return (rval);
9053        }

9055        mutex_enter(&mpt->m_mutex);

9057        if ((mptsas_scsi_capchk(cap, tgtonly, &ckey)) != TRUE) {
9058                mutex_exit(&mpt->m_mutex);
9059                return (UNDEFINED);
9060        }

9062        switch (ckey) {
9063        case SCSI_CAP_DMA_MAX:
9064        case SCSI_CAP_MSG_OUT:
9065        case SCSI_CAP_PARITY:
9066        case SCSI_CAP_INITIATOR_ID:
9067        case SCSI_CAP_LINKED_CMDS:
9068        case SCSI_CAP_UNTAGGED_QING:
9069        case SCSI_CAP_RESET_NOTIFICATION:
9070                /*
9071                 * None of these are settable via
9072                 * the capability interface.
```

```
9073                 */
9074                break;
9075        case SCSI_CAP_ARQ:
9076                /*
9077                 * We cannot turn off arq so return false if asked to
9078                 */
9079                if (value) {
9080                        rval = TRUE;
9081                } else {
9082                        rval = FALSE;
9083                }
9084                break;
9085        case SCSI_CAP_TAGGED_QING:
9086                mptsas_set_throttle(mpt, ((mptsas_tgt_private_t *)
9087                    (ap->a_hba_tran->tran_tgt_private))->t_private,
9088                    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);
9089                rval = TRUE;
9090                break;
9091        case SCSI_CAP_QFULL_RETRIES:
9092                ((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
9093                    t_private->m_qfull_retries = (uchar_t)value;
9094                rval = TRUE;
9095                break;
9096        case SCSI_CAP_QFULL_RETRY_INTERVAL:
9097                ((mptsas_tgt_private_t *)(ap->a_hba_tran->tran_tgt_private))->
9098                    t_private->m_qfull_retry_interval =
9099                    drv_usectohz(value * 1000);
9100                rval = TRUE;
9101                break;
9102        default:
9103                rval = UNDEFINED;
9104                break;
9105        }
9106        mutex_exit(&mpt->m_mutex);
9107        return (rval);
9108 }
```

*unchanged_portion_omitted_*

```
9126 static int
9127 mptsas_alloc_active_slots(mptsas_t *mpt, int flag)
9128 {
9129        mptsas_slots_t  *old_active = mpt->m_active;
9130        mptsas_slots_t  *new_active;
9131        size_t          size;
9132        int             rval = -1, i;
9695        int             rval = -1, nslot, i;
9696        mptsas_slot_free_e_t    *pe;

9134        /*
9135         * if there are active commands, then we cannot
9136         * change size of active slots array.
9137         */
9138        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        }

9140        size = MPTSAS_SLOTS_SIZE(mpt);
9141        new_active = kmem_zalloc(size, flag);
9142        if (new_active == NULL) {
```



```
9143                NDBG1(("new active alloc failed"));
9144                return (rval);
9145        }
9146        /*
9147         * Since SMID 0 is reserved and the TM slot is reserved, the
9148         * number of slots that can be used at any one time is
9149         * m_max_requests - 2.
9150         */
9151        new_active->m_n_slots = (mpt->m_max_requests - 2);
9714        new_active->m_n_slots = nslot = (mpt->m_max_requests - 2);
9152        new_active->m_size = size;
9153        new_active->m_tags = 1;

9154        if (old_active) {
9155                new_active->m_tgttbl = old_active->m_tgttbl;
9156                new_active->m_smptbl = old_active->m_smptbl;
9157                new_active->m_num_raid_configs =
9158                    old_active->m_num_raid_configs;
9159                for (i = 0; i < new_active->m_num_raid_configs; i++) {
9160                        new_active->m_raidconfig[i] =
9161                            old_active->m_raidconfig[i];
9162                }
9163                mptsas_free_active_slots(mpt);
9164        }

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].
9750                    m_slot_allocq.s.m_fq_mutex, NULL, MUTEX_DRIVER,
9751                    DDI_INTR_PRI(mpt->m_intr_pri));
9752                mutex_init(&mpt->m_slot_freeq_pairp[i].
9753                    m_slot_releq.s.m_fq_mutex, NULL, MUTEX_DRIVER,
9754                    DDI_INTR_PRI(mpt->m_intr_pri));
9755        }
9756        pe = mpt->m_slot_free_ae = kmem_zalloc(nslot *
9757            sizeof (mptsas_slot_free_e_t), KM_SLEEP);
9758        /*
9759         * An array of Mpi2ReplyDescriptorsUnion_t is defined here.
9760         * We are trying to eliminate the m_mutex in the context
9761         * reply code path in the ISR. Since the read of the
9762         * ReplyDescriptor and update/write of the ReplyIndex must
9763         * be atomic (since the poll thread may also update them at
9764         * the same time) so we first read out of the ReplyDescriptor
9765         * into this array and update the ReplyIndex register with a
9766         * separate mutex m_intr_mutex protected, and then release the
9767         * mutex and process all of them. the length of the array is
9768         * defined as max as 128(128*64=8k), which is
9769         * assumed as the maxmium depth of the interrupt coalese.
9770         */
```

```
9771          mpt->m_reply = kmem_zalloc(MPI_ADDRESS_COALSCE_MAX *
9772              sizeof (Mpi2ReplyDescriptorsUnion_t), KM_SLEEP);
9773          for (i = 0; i < nslot; i++, pe++) {
9774                  pe->slot = i + 1; /* SMID 0 is reserved */
9775                  pe->cpuid = i % mpt->m_slot_freeq_pair_n;
9776                  list_insert_tail(&mpt->m_slot_freeq_pairp
9777                      [i % mpt->m_slot_freeq_pair_n]
9778                      .m_slot_allocq.s.m_fq_list, pe);
9779                  mpt->m_slot_freeq_pairp[i % mpt->m_slot_freeq_pair_n]
9780                      .m_slot_allocq.s.m_fq_n++;
9781                  mpt->m_slot_freeq_pairp[i % mpt->m_slot_freeq_pair_n]
9782                      .m_slot_allocq.s.m_fq_n_init++;
9783          }

9165          mpt->m_active = new_active;
9166          rval = 0;

9168          return (rval);
9169  }

9171  static void
9172  mptsas_free_active_slots(mptsas_t *mpt)
9173  {
9174          mptsas_slots_t  *active = mpt->m_active;
9175          size_t          size;
9796          mptsas_slot_free_e_t    *pe;
9797          int     i;

9177          if (active == NULL)
9178                  return;

9802          if (mpt->m_slot_freeq_pairp) {
9803                  for (i = 0; i < mpt->m_slot_freeq_pair_n; i++) {
9804                          while ((pe = list_head(&mpt->m_slot_freeq_pairp
9805                              [i].m_slot_allocq.s.m_fq_list)) != NULL) {
9806                                  list_remove(&mpt->m_slot_freeq_pairp[i]
9807                                      .m_slot_allocq.s.m_fq_list, pe);
9808                          }
9809                          list_destroy(&mpt->m_slot_freeq_pairp
9810                              [i].m_slot_allocq.s.m_fq_list);
9811                          while ((pe = list_head(&mpt->m_slot_freeq_pairp
9812                              [i].m_slot_releq.s.m_fq_list)) != NULL) {
9813                                  list_remove(&mpt->m_slot_freeq_pairp[i]
9814                                      .m_slot_releq.s.m_fq_list, pe);
9815                          }
9816                          list_destroy(&mpt->m_slot_freeq_pairp
9817                              [i].m_slot_releq.s.m_fq_list);
9818                          mutex_destroy(&mpt->m_slot_freeq_pairp
9819                              [i].m_slot_allocq.s.m_fq_mutex);
9820                          mutex_destroy(&mpt->m_slot_freeq_pairp
9821                              [i].m_slot_releq.s.m_fq_mutex);
9822                  }
9823                  kmem_free(mpt->m_slot_freeq_pairp, mpt->m_slot_freeq_pair_n *
9824                      sizeof (mptsas_slot_freeq_pair_t));
9825          }
9826          if (mpt->m_slot_free_ae)
9827                  kmem_free(mpt->m_slot_free_ae, mpt->m_active->m_n_slots *
9828                      sizeof (mptsas_slot_free_e_t));

9830          if (mpt->m_reply)
9831                  kmem_free(mpt->m_reply, MPI_ADDRESS_COALSCE_MAX *
9832                      sizeof (Mpi2ReplyDescriptorsUnion_t));

9179          size = active->m_size;
9180          kmem_free(active, size);
9181          mpt->m_active = NULL;
```



```
9182  }
______unchanged_portion_omitted_

9306  static void
9307  mptsas_watchsubr(mptsas_t *mpt)
9308  {
9309          int             i;
9310          mptsas_cmd_t    *cmd;
9311          mptsas_target_t *ptgt = NULL;

9313          NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

9315  #ifdef MPTSAS_TEST
9316          if (mptsas_enable_untagged) {
9317                  mptsas_test_untagged++;
9318          }
9319  #endif

9321          /*
9322           * Check for commands stuck in active slot
9323           * Account for TM requests, which use the last SMID.
9324           */
9980          mutex_enter(&mpt->m_intr_mutex);
9325          for (i = 0; i <= mpt->m_active->m_n_slots; i++) {
9326                  if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9327                          if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9328                                  cmd->cmd_active_timeout -=
9329                                      mptsas_scsi_watchdog_tick;
9330                                  if (cmd->cmd_active_timeout <= 0) {
9331                                          /*
9332                                           * There seems to be a command stuck
9333                                           * in the active slot.  Drain throttle.
9334                                           */
9335                                          mptsas_set_throttle(mpt,
9336                                              cmd->cmd_tgt_addr,
9991                                          ptgt = cmd->cmd_tgt_addr;
9992                                          mutex_enter(&ptgt->m_tgt_intr_mutex);
9993                                          mptsas_set_throttle(mpt, ptgt,
9337                                              DRAIN_THROTTLE);
9995                                          mutex_exit(&ptgt->m_tgt_intr_mutex);
9338                                  }
9339                          }
9340                          if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
9341                              (cmd->cmd_flags & CFLAG_CONFIG) ||
9342                              (cmd->cmd_flags & CFLAG_FW_DIAG)) {
9343                                  cmd->cmd_active_timeout -=
9344                                      mptsas_scsi_watchdog_tick;
9345                                  if (cmd->cmd_active_timeout <= 0) {
9346                                          /*
9347                                           * passthrough command timeout
9348                                           */
9349                                          cmd->cmd_flags |= (CFLAG_FINISHED |
9350                                              CFLAG_TIMEOUT);
9351                                          cv_broadcast(&mpt->m_passthru_cv);
9352                                          cv_broadcast(&mpt->m_config_cv);
9353                                          cv_broadcast(&mpt->m_fw_diag_cv);
9354                                  }
9355                          }
9356                  }
9357          }
10016         mutex_exit(&mpt->m_intr_mutex);

9359          ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9360              MPTSAS_HASH_FIRST);
9361          while (ptgt != NULL) {
9362                  /*
```

```
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                   /*
9363                    * If we were draining due to a qfull condition,
9364                    * go back to full throttle.
9365                    */
9366                   if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9367                       (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9368                       (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9369                           mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9370                           mptsas_restart_hba(mpt);
9371                   }

9373                   if ((ptgt->m_t_ncmds > 0) &&
9374                       (ptgt->m_timebase)) {

9376                           if (ptgt->m_timebase <=
9377                               mptsas_scsi_watchdog_tick) {
9378                                   ptgt->m_timebase +=
9379                                       mptsas_scsi_watchdog_tick;
10046                                   mutex_exit(&ptgt->m_tgt_intr_mutex);
9380                                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9381                                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9382                                   continue;
9383                           }

9385                           ptgt->m_timeout -= mptsas_scsi_watchdog_tick;

9387                           if (ptgt->m_timeout < 0) {
10055                                   mutex_exit(&ptgt->m_tgt_intr_mutex);
9388                                   mptsas_cmd_timeout(mpt, ptgt->m_devhdl);
9389                                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9390                                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9391                                   continue;
9392                           }

9394                           if ((ptgt->m_timeout) <=
9395                               mptsas_scsi_watchdog_tick) {
9396                                   NDBG23(("pending timeout"));
9397                                   mptsas_set_throttle(mpt, ptgt,
9398                                       DRAIN_THROTTLE);
9399                           }
9400                   }

10069                   mutex_exit(&ptgt->m_tgt_intr_mutex);
9402                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9403                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9404           }
9405 }
```

*unchanged_portion_omitted_*

```
9458 static int
9459 mptsas_quiesce_bus(mptsas_t *mpt)
9460 {
9461           mptsas_target_t *ptgt = NULL;

9463           NDBG28(("mptsas_quiesce_bus"));
9464           mutex_enter(&mpt->m_mutex);

9466           /* Set all the throttles to zero */
9467           ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9468               MPTSAS_HASH_FIRST);
```



```
9469           while (ptgt != NULL) {
10138                   mutex_enter(&ptgt->m_tgt_intr_mutex);
9470                   mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10140                   mutex_exit(&ptgt->m_tgt_intr_mutex);

9472                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9473                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9474           }

9476           /* If there are any outstanding commands in the queue */
9477           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);
9478                   mpt->m_softstate |= MPTSAS_SS_DRAINING;
9479                   mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
9480                       mpt, (MPTSAS_QUIESCE_TIMEOUT * drv_usectohz(1000000)));
9481                   if (cv_wait_sig(&mpt->m_cv, &mpt->m_mutex) == 0) {
9482                           /*
9483                            * Quiesce has been interrupted
9484                            */
9485                           mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
9486                           ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9487                               &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
9488                           while (ptgt != NULL) {
10161                                   mutex_enter(&ptgt->m_tgt_intr_mutex);
9489                                   mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10163                                   mutex_exit(&ptgt->m_tgt_intr_mutex);

9491                                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9492                                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9493                           }
9494                           mptsas_restart_hba(mpt);
9495                           if (mpt->m_quiesce_timeid != 0) {
9496                                   timeout_id_t tid = mpt->m_quiesce_timeid;
9497                                   mpt->m_quiesce_timeid = 0;
9498                                   mutex_exit(&mpt->m_mutex);
9499                                   (void) untimeout(tid);
9500                                   return (-1);
9501                           }
9502                           mutex_exit(&mpt->m_mutex);
9503                           return (-1);
9504                   } else {
9505                           /* Bus has been quiesced */
9506                           ASSERT(mpt->m_quiesce_timeid == 0);
9507                           mpt->m_softstate &= ~MPTSAS_SS_DRAINING;
9508                           mpt->m_softstate |= MPTSAS_SS_QUIESCED;
9509                           mutex_exit(&mpt->m_mutex);
9510                           return (0);
9511                   }
9512           }
10187           mutex_exit(&mpt->m_intr_mutex);
9513           /* Bus was not busy - QUIESCED */
9514           mutex_exit(&mpt->m_mutex);

9516           return (0);
9517 }

9519 static int
9520 mptsas_unquiesce_bus(mptsas_t *mpt)
9521 {
9522           mptsas_target_t *ptgt = NULL;

9524           NDBG28(("mptsas_unquiesce_bus"));
9525           mutex_enter(&mpt->m_mutex);
9526           mpt->m_softstate &= ~MPTSAS_SS_QUIESCED;
```

```
9527         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9528             MPTSAS_HASH_FIRST);
9529         while (ptgt != NULL) {
10205                 mutex_enter(&ptgt->m_tgt_intr_mutex);
9530                 mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
10207                 mutex_exit(&ptgt->m_tgt_intr_mutex);

9532                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9533                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9534         }
9535         mptsas_restart_hba(mpt);
9536         mutex_exit(&mpt->m_mutex);
9537         return (0);
9538 }

9540 static void
9541 mptsas_ncmds_checkdrain(void *arg)
9542 {
9543         mptsas_t        *mpt = arg;
9544         mptsas_target_t *ptgt = NULL;

9546         mutex_enter(&mpt->m_mutex);
9547         if (mpt->m_softstate & MPTSAS_SS_DRAINING) {
9548                 mpt->m_quiesce_timeid = 0;
9549                 if (mpt->m_ncmds == 0) {
9550                         /* Command queue has been drained */
9551                         cv_signal(&mpt->m_cv);
9552                 } else {
10226                  mutex_enter(&mpt->m_intr_mutex);
10227                  if (mptsas_outstanding_cmds_n(mpt)) {
10228                          mutex_exit(&mpt->m_intr_mutex);
9553                          /*
9554                           * The throttle may have been reset because
9555                           * of a SCSI bus reset
9556                           */
9557                          ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9558                              &mpt->m_active->m_tgttbl, MPTSAS_HASH_FIRST);
9559                          while (ptgt != NULL) {
10236                                  mutex_enter(&ptgt->m_tgt_intr_mutex);
9560                                  mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
10238                                  mutex_exit(&ptgt->m_tgt_intr_mutex);

9562                                  ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9563                                      &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9564                          }

9566                          mpt->m_quiesce_timeid = timeout(mptsas_ncmds_checkdrain,
9567                              mpt, (MPTSAS_QUIESCE_TIMEOUT *
9568                              drv_usectohz(1000000)));
10247                  } else {
10248                          mutex_exit(&mpt->m_intr_mutex);
10249                          /* Command queue has been drained */
10250                          cv_signal(&mpt->m_cv);
9569                 }
9570         }
9571         mutex_exit(&mpt->m_mutex);
9572 }
```

*_____unchanged_portion_omitted_*

```
9598 static void
9599 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
9600 {
9601         caddr_t                 memp;
9602         pMPI2RequestHeader_t    request_hdrp;
9603         struct scsi_pkt         *pkt = cmd->cmd_pkt;
9604         mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
```



```
9605         uint32_t                request_size, data_size, dataout_size;
9606         uint32_t                direction;
9607         ddi_dma_cookie_t        data_cookie;
9608         ddi_dma_cookie_t        dataout_cookie;
9609         uint32_t                request_desc_low, request_desc_high = 0;
9610         uint32_t                i, sense_bufp;
9611         uint8_t                 desc_type;
9612         uint8_t                 *request, function;
9613         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
9614         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

9616         desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

9618         request = pt->request;
9619         direction = pt->direction;
9620         request_size = pt->request_size;
9621         data_size = pt->data_size;
9622         dataout_size = pt->dataout_size;
9623         data_cookie = pt->data_cookie;
9624         dataout_cookie = pt->dataout_cookie;

9626         /*
9627          * Store the passthrough message in memory location
9628          * corresponding to our slot number
9629          */
9630         memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
9631         request_hdrp = (pMPI2RequestHeader_t)memp;
9632         bzero(memp, mpt->m_req_frame_size);

9634         for (i = 0; i < request_size; i++) {
9635                 bcopy(request + i, memp + i, 1);
9636         }

9638         if (data_size || dataout_size) {
9639                 pMpi2SGESimple64_t      sgep;
9640                 uint32_t                sge_flags;

9642                 sgep = (pMpi2SGESimple64_t)((uint8_t *)request_hdrp +
9643                     request_size);
9644                 if (dataout_size) {

9646                         sge_flags = dataout_size |
9647                             ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
9648                             MPI2_SGE_FLAGS_END_OF_BUFFER |
9649                             MPI2_SGE_FLAGS_HOST_TO_IOC |
9650                             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
9651                             MPI2_SGE_FLAGS_SHIFT);
9652                         ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
9653                         ddi_put32(acc_hdl, &sgep->Address.Low,
9654                             (uint32_t)(dataout_cookie.dmac_laddress &
9655                             0xffffffffull));
9656                         ddi_put32(acc_hdl, &sgep->Address.High,
9657                             (uint32_t)(dataout_cookie.dmac_laddress
9658                             >> 32));
9659                         sgep++;
9660                 }
9661                 sge_flags = data_size;
9662                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
9663                     MPI2_SGE_FLAGS_LAST_ELEMENT |
9664                     MPI2_SGE_FLAGS_END_OF_BUFFER |
9665                     MPI2_SGE_FLAGS_END_OF_LIST |
9666                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
9667                     MPI2_SGE_FLAGS_SHIFT);
9668                 if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
9669                         sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
9670                             MPI2_SGE_FLAGS_SHIFT);
```

```
9671                } else {
9672                        sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
9673                            MPI2_SGE_FLAGS_SHIFT);
9674                }
9675                ddi_put32(acc_hdl, &sgep->FlagsLength,
9676                    sge_flags);
9677                ddi_put32(acc_hdl, &sgep->Address.Low,
9678                    (uint32_t)(data_cookie.dmac_laddress &
9679                    0xffffffffull));
9680                ddi_put32(acc_hdl, &sgep->Address.High,
9681                    (uint32_t)(data_cookie.dmac_laddress >> 32));
9682        }

9684        function = request_hdrp->Function;
9685        if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
9686            (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
9687                pMpi2SCSIIORequest_t    scsi_io_req;

9689                scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;
9690                /*
9691                 * Put SGE for data and data_out buffer at the end of
9692                 * scsi_io_request message header.(64 bytes in total)
9693                 * Following above SGEs, the residual space will be
9694                 * used by sense data.
9695                 */
9696                ddi_put8(acc_hdl,
9697                    &scsi_io_req->SenseBufferLength,
9698                    (uint8_t)(request_size - 64));

9700                sense_bufp = mpt->m_req_frame_dma_addr +
9701                    (mpt->m_req_frame_size * cmd->cmd_slot);
9702                sense_bufp += 64;
9703                ddi_put32(acc_hdl,
9704                    &scsi_io_req->SenseBufferLowAddress, sense_bufp);

9706                /*
9707                 * Set SGLOffset0 value
9708                 */
9709                ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
9710                    offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);

9712                /*
9713                 * Setup descriptor info.  RAID passthrough must use the
9714                 * default request descriptor which is already set, so if this
9715                 * is a SCSI IO request, change the descriptor to SCSI IO.
9716                 */
9717                if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
9718                        desc_type = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
9719                        request_desc_high = (ddi_get16(acc_hdl,
9720                            &scsi_io_req->DevHandle) << 16);
9721                }
9722        }

9724        /*
9725         * We must wait till the message has been completed before
9726         * beginning the next message so we wait for this one to
9727         * finish.
9728         */
9729        (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
9730        request_desc_low = (cmd->cmd_slot << 16) + desc_type;
9731        cmd->cmd_rfm = NULL;
10414        mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
9732        MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
9733        if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
9734            (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
9735                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
```



```
9736        }
9737 }
_____unchanged_portion_omitted_

10099 static void
10100 mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd)
10101 {
10102        pMpi2DiagBufferPostRequest_t    pDiag_post_msg;
10103        pMpi2DiagReleaseRequest_t       pDiag_release_msg;
10104        struct scsi_pkt                 *pkt = cmd->cmd_pkt;
10105        mptsas_diag_request_t           *diag = pkt->pkt_ha_private;
10106        uint32_t                        request_desc_low, i;

10108        ASSERT(mutex_owned(&mpt->m_mutex));

10110        /*
10111         * Form the diag message depending on the post or release function.
10112         */
10113        if (diag->function == MPI2_FUNCTION_DIAG_BUFFER_POST) {
10114                pDiag_post_msg = (pMpi2DiagBufferPostRequest_t)
10115                    (mpt->m_req_frame + (mpt->m_req_frame_size *
10116                    cmd->cmd_slot));
10117                bzero(pDiag_post_msg, mpt->m_req_frame_size);
10118                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->Function,
10119                    diag->function);
10120                ddi_put8(mpt->m_acc_req_frame_hdl, &pDiag_post_msg->BufferType,
10121                    diag->pBuffer->buffer_type);
10122                ddi_put8(mpt->m_acc_req_frame_hdl,
10123                    &pDiag_post_msg->ExtendedType,
10124                    diag->pBuffer->extended_type);
10125                ddi_put32(mpt->m_acc_req_frame_hdl,
10126                    &pDiag_post_msg->BufferLength,
10127                    diag->pBuffer->buffer_data.size);
10128                for (i = 0; i < (sizeof (pDiag_post_msg->ProductSpecific) / 4);
10129                    i++) {
10130                        ddi_put32(mpt->m_acc_req_frame_hdl,
10131                            &pDiag_post_msg->ProductSpecific[i],
10132                            diag->pBuffer->product_specific[i]);
10133                }
10134                ddi_put32(mpt->m_acc_req_frame_hdl,
10135                    &pDiag_post_msg->BufferAddress.Low,
10136                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
10137                    & 0xffffffffull));
10138                ddi_put32(mpt->m_acc_req_frame_hdl,
10139                    &pDiag_post_msg->BufferAddress.High,
10140                    (uint32_t)(diag->pBuffer->buffer_data.cookie.dmac_laddress
10141                    >> 32));
10142        } else {
10143                pDiag_release_msg = (pMpi2DiagReleaseRequest_t)
10144                    (mpt->m_req_frame + (mpt->m_req_frame_size *
10145                    cmd->cmd_slot));
10146                bzero(pDiag_release_msg, mpt->m_req_frame_size);
10147                ddi_put8(mpt->m_acc_req_frame_hdl,
10148                    &pDiag_release_msg->Function, diag->function);
10149                ddi_put8(mpt->m_acc_req_frame_hdl,
10150                    &pDiag_release_msg->BufferType,
10151                    diag->pBuffer->buffer_type);
10152        }

10154        /*
10155         * Send the message
10156         */
10157        (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
10158            DDI_DMA_SYNC_FORDEV);
10159        request_desc_low = (cmd->cmd_slot << 16) +
10160            MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
```

```
10161         cmd->cmd_rfm = NULL;
10845         mpt->m_active->m_slot[cmd->cmd_slot] = cmd;
10162         MPTSAS_START_CMD(mpt, request_desc_low, 0);
10163         if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl) !=
10164             DDI_SUCCESS) ||
10165             (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl) !=
10166             DDI_SUCCESS)) {
10167                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10168         }
10169 }
```

*unchanged_portion_omitted_*

```
11269 static int
11270 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
11271     int *rval)
11272 {
11273         int                     status = 0;
11274         mptsas_t                *mpt;
11275         mptsas_update_flash_t   flashdata;
11276         mptsas_pass_thru_t      passthru_data;
11277         mptsas_adapter_data_t   adapter_data;
11278         mptsas_pci_info_t       pci_info;
11279         int                     copylen;

11281         int                     iport_flag = 0;
11282         dev_info_t              *dip = NULL;
11283         mptsas_phymask_t        phymask = 0;
11284         struct devctl_iocdata   *dcp = NULL;
11285         uint32_t                slotstatus = 0;
11286         char                    *addr = NULL;
11287         mptsas_target_t         *ptgt = NULL;

11289         *rval = MPTIOCTL_STATUS_GOOD;
11290         if (secpolicy_sys_config(credp, B_FALSE) != 0) {
11291                 return (EPERM);
11292         }

11294         mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
11295         if (mpt == NULL) {
11296                 /*
11297                  * Called from iport node, get the states
11298                  */
11299                 iport_flag = 1;
11300                 dip = mptsas_get_dip_from_dev(dev, &phymask);
11301                 if (dip == NULL) {
11302                         return (ENXIO);
11303                 }
11304                 mpt = DIP2MPT(dip);
11305         }
11306         /* Make sure power level is D0 before accessing registers */
11307         mutex_enter(&mpt->m_mutex);
11308         if (mpt->m_options & MPTSAS_OPT_PM) {
11309                 (void) pm_busy_component(mpt->m_dip, 0);
11310                 if (mpt->m_power_level != PM_LEVEL_D0) {
11311                         mutex_exit(&mpt->m_mutex);
11312                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
11313                             DDI_SUCCESS) {
11314                                 mptsas_log(mpt, CE_WARN,
11315                                     "mptsas%d: mptsas_ioctl: Raise power "
11316                                     "request failed.", mpt->m_instance);
11317                                 (void) pm_idle_component(mpt->m_dip, 0);
11318                                 return (ENXIO);
11319                         }
11320                 } else {
11321                         mutex_exit(&mpt->m_mutex);
11322                 }
```



```
11323         } else {
11324                 mutex_exit(&mpt->m_mutex);
11325         }

11327         if (iport_flag) {
11328                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
11329                 if (status != 0) {
11330                         goto out;
11331                 }
11332                 /*
11333                  * The following code control the OK2RM LED, it doesn't affect
11334                  * the ioctl return status.
11335                  */
11336                 if ((cmd == DEVCTL_DEVICE_ONLINE) ||
11337                     (cmd == DEVCTL_DEVICE_OFFLINE)) {
11338                         if (ndi_dc_allochdl((void *)data, &dcp) !=
11339                             NDI_SUCCESS) {
11340                                 goto out;
11341                         }
11342                         addr = ndi_dc_getaddr(dcp);
11343                         ptgt = mptsas_addr_to_ptgt(mpt, addr, phymask);
11344                         if (ptgt == NULL) {
11345                                 NDBG14(("mptsas_ioctl led control: tgt %s not "
11346                                     "found", addr));
11347                                 ndi_dc_freehdl(dcp);
11348                                 goto out;
11349                         }
11350                         mutex_enter(&mpt->m_mutex);
11351                         if (cmd == DEVCTL_DEVICE_ONLINE) {
11352                                 ptgt->m_tgt_unconfigured = 0;
11353                         } else if (cmd == DEVCTL_DEVICE_OFFLINE) {
11354                                 ptgt->m_tgt_unconfigured = 1;
11355                         }
11356                         slotstatus = 0;
11357 #ifdef MPTSAS_GET_LED
11358                         /*
11359                          * The get led status can't get a valid/reasonable
11360                          * state, so ignore the get led status, and write the
11361                          * required value directly
11362                          */
11363                         if (mptsas_get_led_status(mpt, ptgt, &slotstatus) !=
11364                             DDI_SUCCESS) {
11365                                 NDBG14(("mptsas_ioctl: get LED for tgt %s "
11366                                     "failed %x", addr, slotstatus));
11367                                 slotstatus = 0;
11368                         }
11369                         NDBG14(("mptsas_ioctl: LED status %x for %s",
11370                             slotstatus, addr));
11371 #endif
11372                         if (cmd == DEVCTL_DEVICE_OFFLINE) {
11373                                 slotstatus |=
11374                                     MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;
11375                         } else {
11376                                 slotstatus &=
11377                                     ~MPI2_SEP_REQ_SLOTSTATUS_REQUEST_REMOVE;
11378                         }
11379                         if (mptsas_set_led_status(mpt, ptgt, slotstatus) !=
11380                             DDI_SUCCESS) {
11381                                 NDBG14(("mptsas_ioctl: set LED for tgt %s "
11382                                     "failed %x", addr, slotstatus));
11383                         }
11384                         mutex_exit(&mpt->m_mutex);
11385                         ndi_dc_freehdl(dcp);
11386                 }
11387                 goto out;
11388         }
```

```
11389         switch (cmd) {
11390                 case MPTIOCTL_UPDATE_FLASH:
11391                         if (ddi_copyin((void *)data, &flashdata,
11392                                 sizeof (struct mptsas_update_flash), mode)) {
11393                                 status = EFAULT;
11394                                 break;
11395                         }

11397                         mutex_enter(&mpt->m_mutex);
11398                         if (mptsas_update_flash(mpt,
11399                             (caddr_t)(long)flashdata.PtrBuffer,
11400                             flashdata.ImageSize, flashdata.ImageType, mode)) {
11401                                 status = EFAULT;
11402                         }

11404                         /*
11405                          * Reset the chip to start using the new
11406                          * firmware.  Reset if failed also.
11407                          */
11408                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11409                         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
11410                                 status = EFAULT;
11411                         }
11412                         mutex_exit(&mpt->m_mutex);
11413                         break;
11414                 case MPTIOCTL_PASS_THRU:
11415                         /*
11416                          * The user has requested to pass through a command to
11417                          * be executed by the MPT firmware.  Call our routine
11418                          * which does this.  Only allow one passthru IOCTL at
11419                          * one time. Other threads will block on
11420                          * m_passthru_mutex, which is of adaptive variant.
11421                          */
11422                         if (ddi_copyin((void *)data, &passthru_data,
11423                             sizeof (mptsas_pass_thru_t), mode)) {
11424                                 status = EFAULT;
11425                                 break;
11426                         }
11427                         mutex_enter(&mpt->m_passthru_mutex);
11428                         mutex_enter(&mpt->m_mutex);
11429                         status = mptsas_pass_thru(mpt, &passthru_data, mode);
11430                         mutex_exit(&mpt->m_mutex);
11431                         mutex_exit(&mpt->m_passthru_mutex);

11433                         break;
11434                 case MPTIOCTL_GET_ADAPTER_DATA:
11435                         /*
11436                          * The user has requested to read adapter data.  Call
11437                          * our routine which does this.
11438                          */
11439                         bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
11440                         if (ddi_copyin((void *)data, (void *)&adapter_data,
11441                             sizeof (mptsas_adapter_data_t), mode)) {
11442                                 status = EFAULT;
11443                                 break;
11444                         }
11445                         if (adapter_data.StructureLength >=
11446                             sizeof (mptsas_adapter_data_t)) {
11447                                 adapter_data.StructureLength = (uint32_t)
11448                                     sizeof (mptsas_adapter_data_t);
11449                                 copylen = sizeof (mptsas_adapter_data_t);
11450                                 mutex_enter(&mpt->m_mutex);
11451                                 mptsas_read_adapter_data(mpt, &adapter_data);
11452                                 mutex_exit(&mpt->m_mutex);
11453                         } else {
11454                                 adapter_data.StructureLength = (uint32_t)
```



```
11455                                     sizeof (mptsas_adapter_data_t);
11456                                 copylen = sizeof (adapter_data.StructureLength);
11457                                 *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
11458                         }
11459                         if (ddi_copyout((void *)(&adapter_data), (void *)data,
11460                             copylen, mode) != 0) {
11461                                 status = EFAULT;
11462                         }
11463                         break;
11464                 case MPTIOCTL_GET_PCI_INFO:
11465                         /*
11466                          * The user has requested to read pci info.  Call
11467                          * our routine which does this.
11468                          */
11469                         bzero(&pci_info, sizeof (mptsas_pci_info_t));
11470                         mutex_enter(&mpt->m_mutex);
11471                         mptsas_read_pci_info(mpt, &pci_info);
11472                         mutex_exit(&mpt->m_mutex);
11473                         if (ddi_copyout((void *)(&pci_info), (void *)data,
11474                             sizeof (mptsas_pci_info_t), mode) != 0) {
11475                                 status = EFAULT;
11476                         }
11477                         break;
11478                 case MPTIOCTL_RESET_ADAPTER:
11479                         mutex_enter(&mpt->m_mutex);
11480                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11481                         if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
11482                                 mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
11483                                     "failed");
11484                                 status = EFAULT;
11485                         }
11486                         mutex_exit(&mpt->m_mutex);
11487                         break;
11488                 case MPTIOCTL_DIAG_ACTION:
11489                         /*
11490                          * The user has done a diag buffer action.  Call our
11491                          * routine which does this.  Only allow one diag action
11492                          * at one time.
11493                          */
11494                         mutex_enter(&mpt->m_mutex);
11495                         if (mpt->m_diag_action_in_progress) {
11496                                 mutex_exit(&mpt->m_mutex);
11497                                 return (EBUSY);
11498                         }
11499                         mpt->m_diag_action_in_progress = 1;
11500                         status = mptsas_diag_action(mpt,
11501                             (mptsas_diag_action_t *)data, mode);
11502                         mpt->m_diag_action_in_progress = 0;
11503                         mutex_exit(&mpt->m_mutex);
11504                         break;
11505                 case MPTIOCTL_EVENT_QUERY:
11506                         /*
11507                          * The user has done an event query. Call our routine
11508                          * which does this.
11509                          */
11510                         status = mptsas_event_query(mpt,
11511                             (mptsas_event_query_t *)data, mode, rval);
11512                         break;
11513                 case MPTIOCTL_EVENT_ENABLE:
11514                         /*
11515                          * The user has done an event enable. Call our routine
11516                          * which does this.
11517                          */
11518                         status = mptsas_event_enable(mpt,
11519                             (mptsas_event_enable_t *)data, mode, rval);
11520                         break;
```

```
11521                 case MPTIOCTL_EVENT_REPORT:
11522                         /*
11523                          * The user has done an event report. Call our routine
11524                          * which does this.
11525                          */
11526                         status = mptsas_event_report(mpt,
11527                             (mptsas_event_report_t *)data, mode, rval);
11528                         break;
11529                 case MPTIOCTL_REG_ACCESS:
11530                         /*
11531                          * The user has requested register access.  Call our
11532                          * routine which does this.
11533                          */
11534                         status = mptsas_reg_access(mpt,
11535                             (mptsas_reg_access_t *)data, mode);
11536                         break;
11537                 default:
11538                         status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
11539                             rval);
11540                         break;
11541         }

11543 out:
12228           if (mpt->m_options & MPTSAS_OPT_PM)
12229                   (void) pm_idle_component(mpt->m_dip, 0);
11544           return (status);
11545 }

11547 int
11548 mptsas_restart_ioc(mptsas_t *mpt)
11549 {
11550           int             rval = DDI_SUCCESS;
11551           mptsas_target_t *ptgt = NULL;

11553           ASSERT(mutex_owned(&mpt->m_mutex));

11555           /*
11556            * Set a flag telling I/O path that we're processing a reset.  This is
11557            * needed because after the reset is complete, the hash table still
11558            * needs to be rebuilt.  If I/Os are started before the hash table is
11559            * rebuilt, I/O errors will occur.  This flag allows I/Os to be marked
11560            * so that they can be retried.
11561            */
11562           mpt->m_in_reset = TRUE;

11564           /*
11565            * Set all throttles to HOLD
11566            */
11567           ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
11568               MPTSAS_HASH_FIRST);
11569           while (ptgt != NULL) {
12256                   mutex_enter(&ptgt->m_tgt_intr_mutex);
11570                   mptsas_set_throttle(mpt, ptgt, HOLD_THROTTLE);
12258                   mutex_exit(&ptgt->m_tgt_intr_mutex);

11572                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
11573                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
11574           }

11576           /*
11577            * Disable interrupts
11578            */
11579           MPTSAS_DISABLE_INTR(mpt);

11581           /*
11582            * Abort all commands: outstanding commands, commands in waitq and
```



```
11583            * tx_waitq.
12270            * Abort all commands: outstanding commands, commands in waitq
11584            */
11585           mptsas_flush_hba(mpt);

11587           /*
11588            * Reinitialize the chip.
11589            */
11590           if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
11591                   rval = DDI_FAILURE;
11592           }

11594           /*
11595            * Enable interrupts again
11596            */
11597           MPTSAS_ENABLE_INTR(mpt);

11599           /*
11600            * If mptsas_init_chip was successful, update the driver data.
11601            */
11602           if (rval == DDI_SUCCESS) {
11603                   mptsas_update_driver_data(mpt);
11604           }

11606           /*
11607            * Reset the throttles
11608            */
11609           ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
11610               MPTSAS_HASH_FIRST);
11611           while (ptgt != NULL) {
12299                   mutex_enter(&ptgt->m_tgt_intr_mutex);
11612                   mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
12301                   mutex_exit(&ptgt->m_tgt_intr_mutex);

11614                   ptgt = (mptsas_target_t *)mptsas_hash_traverse(
11615                       &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
11616           }

11618           mptsas_doneq_empty(mpt);
11619           mptsas_restart_hba(mpt);

11621           if (rval != DDI_SUCCESS) {
11622                   mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
11623                   ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
11624           }

11626           /*
11627            * Clear the reset flag so that I/Os can continue.
11628            */
11629           mpt->m_in_reset = FALSE;

11631           return (rval);
11632 }
_____unchanged_portion_omitted_

11893 static int
11894 mptsas_init_pm(mptsas_t *mpt)
11895 {
11896           char            pmc_name[16];
11897           char            *pmc[] = {
11898                                   NULL,
11899                                   "0=Off (PCI D3 State)",
11900                                   "3=On (PCI D0 State)",
11901                                   NULL
11902                           };
11903           uint16_t        pmcsr_stat;
```

```
11905         if (mptsas_get_pci_cap(mpt) == FALSE) {
11906                 return (DDI_FAILURE);
11907         }
11908         /*
11909          * If PCI's capability does not support PM, then don't need
11910          * to registe the pm-components
11911          */
11912         if (!(mpt->m_options & MPTSAS_OPT_PM))
11913                 return (DDI_SUCCESS);
11914         /*
11915          * If power management is supported by this chip, create
11916          * pm-components property for the power management framework
11917          */
11918         (void) sprintf(pmc_name, "NAME=mptsas%d", mpt->m_instance);
11919         pmc[0] = pmc_name;
11920         if (ddi_prop_update_string_array(DDI_DEV_T_NONE, mpt->m_dip,
11921             "pm-components", pmc, 3) != DDI_PROP_SUCCESS) {
12611                 mutex_enter(&mpt->m_intr_mutex);
11922                 mpt->m_options &= ~MPTSAS_OPT_PM;
12613                 mutex_exit(&mpt->m_intr_mutex);
11923                 mptsas_log(mpt, CE_WARN,
11924                     "mptsas%d: pm-component property creation failed.",
11925                     mpt->m_instance);
11926                 return (DDI_FAILURE);
11927         }

11929         /*
11930          * Power on device.
11931          */
11932         (void) pm_busy_component(mpt->m_dip, 0);
11933         pmcsr_stat = pci_config_get16(mpt->m_config_handle,
11934             mpt->m_pmcsr_offset);
11935         if ((pmcsr_stat & PCI_PMCSR_STATE_MASK) != PCI_PMCSR_D0) {
11936                 mptsas_log(mpt, CE_WARN, "mptsas%d: Power up the device",
11937                     mpt->m_instance);
11938                 pci_config_put16(mpt->m_config_handle, mpt->m_pmcsr_offset,
11939                     PCI_PMCSR_D0);
11940         }
11941         if (pm_power_has_changed(mpt->m_dip, 0, PM_LEVEL_D0) != DDI_SUCCESS) {
11942                 mptsas_log(mpt, CE_WARN, "pm_power_has_changed failed");
11943                 return (DDI_FAILURE);
11944         }
12636         mutex_enter(&mpt->m_intr_mutex);
11945         mpt->m_power_level = PM_LEVEL_D0;
12638         mutex_exit(&mpt->m_intr_mutex);
11946         /*
11947          * Set pm idle delay.
11948          */
11949         mpt->m_pm_idle_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
11950             mpt->m_dip, 0, "mptsas-pm-idle-delay", MPTSAS_PM_IDLE_TIMEOUT);

11952         return (DDI_SUCCESS);
11953 }
______unchanged_portion_omitted_

12310 static int
12311 mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
12312     uint16_t *dev_handle, mptsas_target_t **pptgt)
12313 {
12314         int             rval;
12315         uint32_t        dev_info;
12316         uint64_t        sas_wwn;
12317         mptsas_phymask_t phymask;
12318         uint8_t         physport, phynum, config, disk;
12319         mptsas_slots_t  *slots = mpt->m_active;
```



```
12320         uint64_t        devicename;
12321         uint16_t        pdev_hdl;
12322         mptsas_target_t *tmp_tgt = NULL;
12323         uint16_t        bay_num, enclosure;

12325         ASSERT(*pptgt == NULL);

12327         rval = mptsas_get_sas_device_page0(mpt, page_address, dev_handle,
12328             &sas_wwn, &dev_info, &physport, &phynum, &pdev_hdl,
12329             &bay_num, &enclosure);
12330         if (rval != DDI_SUCCESS) {
12331                 rval = DEV_INFO_FAIL_PAGE0;
12332                 return (rval);
12333         }

12335         if ((dev_info & (MPI2_SAS_DEVICE_INFO_SSP_TARGET |
12336             MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
12337             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) == NULL) {
12338                 rval = DEV_INFO_WRONG_DEVICE_TYPE;
12339                 return (rval);
12340         }

12342         /*
12343          * Check if the dev handle is for a Phys Disk. If so, set return value
12344          * and exit.  Don't add Phys Disks to hash.
12345          */
12346         for (config = 0; config < slots->m_num_raid_configs; config++) {
12347                 for (disk = 0; disk < MPTSAS_MAX_DISKS_IN_CONFIG; disk++) {
12348                         if (*dev_handle == slots->m_raidconfig[config].
12349                             m_physdisk_devhdl[disk]) {
12350                                 rval = DEV_INFO_PHYS_DISK;
12351                                 return (rval);
12352                         }
12353                 }
12354         }

12356         /*
12357          * Get SATA Device Name from SAS device page0 for
12358          * sata device, if device name doesn't exist, set m_sas_wwn to
12359          * 0 for direct attached SATA. For the device behind the expander
12360          * we still can use STP address assigned by expander.
12361          */
12362         if (dev_info & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
12363             MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
12364                 mutex_exit(&mpt->m_mutex);
12365                 /* alloc a tmp_tgt to send the cmd */
12366                 tmp_tgt = kmem_zalloc(sizeof (struct mptsas_target),
12367                     KM_SLEEP);
12368                 tmp_tgt->m_devhdl = *dev_handle;
12369                 tmp_tgt->m_deviceinfo = dev_info;
12370                 tmp_tgt->m_qfull_retries = QFULL_RETRIES;
12371                 tmp_tgt->m_qfull_retry_interval =
12372                     drv_usectohz(QFULL_RETRY_INTERVAL * 1000);
12373                 tmp_tgt->m_t_throttle = MAX_THROTTLE;
12374                 devicename = mptsas_get_sata_guid(mpt, tmp_tgt, 0);
12375                 kmem_free(tmp_tgt, sizeof (struct mptsas_target));
12376                 mutex_enter(&mpt->m_mutex);
12377                 if (devicename != 0 && (((devicename >> 56) & 0xf0) == 0x50)) {
12378                         sas_wwn = devicename;
12379                 } else if (dev_info & MPI2_SAS_DEVICE_INFO_DIRECT_ATTACH) {
12380                         sas_wwn = 0;
12381                 }
12382         }

12384         phymask = mptsas_physport_to_phymask(mpt, physport);
12385         *pptgt = mptsas_tgt_alloc(&slots->m_tgttbl, *dev_handle, sas_wwn,
```

```
12386             dev_info, phymask, phynum);
13079             dev_info, phymask, phynum, mpt);
12387         if (*pptgt == NULL) {
12388                 mptsas_log(mpt, CE_WARN, "Failed to allocated target"
12389                     "structure!");
12390                 rval = DEV_INFO_FAIL_ALLOC;
12391                 return (rval);
12392         }
12393         (*pptgt)->m_enclosure = enclosure;
12394         (*pptgt)->m_slot_num = bay_num;
12395         return (DEV_INFO_SUCCESS);
12396 }
______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_
```



```
15471 void
15472 mptsas_dma_addr_destroy(ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp)
15473 {
15474         if (*dma_hdp == NULL)
15475                 return;

15477         (void) ddi_dma_unbind_handle(*dma_hdp);
15478         (void) ddi_dma_mem_free(acc_hdp);
15479         ddi_dma_free_handle(dma_hdp);
15480         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);
15481 }
______unchanged_portion_omitted_
```

```
**********************************************************
   79763 Tue Dec  4 16:29:52 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
re #6530 mpt_sas crash when more than 1 Initiator involved - ie HA
**********************************************************
______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 16:29:52 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_raid.c
re #6530 mpt_sas crash when more than 1 Initiator involved - ie HA
**********************************************************
```

```
/*
 * 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_***

```
**********************************************************
   43397 Tue Dec  4 16:29:53 2012
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
re #6530 mpt_sas crash when more than 1 Initiator involved - ie HA
**********************************************************
______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;

 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;

 212 } mptsas_target_t;
______unchanged_portion_omitted_

 574 /*
 575  * The following defines are used in mptsas_set_init_mode to track the current
 576  * state as we progress through reprogramming the HBA from target mode into
 577  * initiator mode.
 578  */

 580 #define IOUC_READ_PAGE0         0x00000100
 581 #define IOUC_READ_PAGE1         0x00000200
 582 #define IOUC_WRITE_PAGE1        0x00000400
 583 #define IOUC_DONE               0x00000800
 584 #define DISCOVERY_IN_PROGRESS   MPI2_SASIOUNIT0_PORTFLAGS_DISCOVERY_IN_PROGRESS
 585 #define AUTO_PORT_CONFIGURATION MPI2_SASIOUNIT0_PORTFLAGS_AUTO_PORT_CONFIG

 587 /*
 588  * Last allocated slot is used for TM requests.  Since only m_max_requests
 589  * frames are allocated, the last SMID will be m_max_requests - 1.
 590  */
 591 #define MPTSAS_SLOTS_SIZE(mpt) \
 592         (sizeof (struct mptsas_slots) + (sizeof (struct mptsas_cmd *) * \
 593                 mpt->m_max_requests))
 594 #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;

 596 /*
 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 /*
 597  * Macro for phy_flags
 598  */

 600 typedef struct smhba_info {
 601         kmutex_t        phy_mutex;
 602         uint8_t         phy_id;
 603         uint64_t        sas_addr;
 604         char            path[8];
 605         uint16_t        owner_devhdl;
```

```
606         uint16_t        attached_devhdl;
607         uint8_t         attached_phy_identify;
608         uint32_t        attached_phy_info;
609         uint8_t         programmed_link_rate;
610         uint8_t         hw_link_rate;
611         uint8_t         change_count;
612         uint32_t        phy_info;
613         uint8_t         negotiated_link_rate;
614         uint8_t         port_num;
615         kstat_t         *phy_stats;
616         uint32_t        invalid_dword_count;
617         uint32_t        running_disparity_error_count;
618         uint32_t        loss_of_dword_sync_count;
619         uint32_t        phy_reset_problem_count;
620         void            *mpt;
621 } smhba_info_t;
_____unchanged_portion_omitted_

653 typedef struct mptsas {
654         int             m_instance;

656         struct mptsas *m_next;

658         scsi_hba_tran_t         *m_tran;
659         smp_hba_tran_t          *m_smptran;
660         kmutex_t                m_mutex;
661         kmutex_t                m_passthru_mutex;
662         kcondvar_t              m_cv;
663         kcondvar_t              m_passthru_cv;
664         kcondvar_t              m_fw_cv;
665         kcondvar_t              m_config_cv;
666         kcondvar_t              m_fw_diag_cv;
667         dev_info_t              *m_dip;

669         /*
670          * soft state flags
671          */
672         uint_t          m_softstate;

674         struct mptsas_slots *m_active;  /* outstanding cmds */

676         mptsas_cmd_t    *m_waitq;       /* cmd queue for active request */
677         mptsas_cmd_t    **m_waitqtail;  /* wait queue tail ptr */

679         kmutex_t        m_tx_waitq_mutex;
680         mptsas_cmd_t    *m_tx_waitq;    /* TX cmd queue for active request */
681         mptsas_cmd_t    **m_tx_waitqtail;       /* tx_wait queue tail ptr */
682         int             m_tx_draining;  /* TX queue draining flag */

684         mptsas_cmd_t    *m_doneq;       /* queue of completed commands */
685         mptsas_cmd_t    **m_donetail;   /* queue tail ptr */

714         kmutex_t                m_passthru_mutex;
715         kcondvar_t              m_passthru_cv;
687         /*
688          * variables for helper threads (fan-out interrupts)
689          */
690         mptsas_doneq_thread_list_t      *m_doneq_thread_id;
691         uint32_t                m_doneq_thread_n;
692         uint32_t                m_doneq_thread_threshold;
693         uint32_t                m_doneq_length_threshold;
694         uint32_t                m_doneq_len;
695         kcondvar_t              m_doneq_thread_cv;
696         kmutex_t                m_doneq_mutex;

698         int             m_ncmds;        /* number of outstanding commands */
```



```
699         m_event_struct_t *m_ioc_event_cmdq;     /* cmd queue for ioc event */
700         m_event_struct_t **m_ioc_event_cmdtail; /* ioc cmd queue tail */

702         ddi_acc_handle_t m_datap;       /* operating regs data access handle */

704         struct _MPI2_SYSTEM_INTERFACE_REGS      *m_reg;

706         ushort_t        m_devid;        /* device id of chip. */
707         uchar_t         m_revid;        /* revision of chip. */
708         uint16_t        m_svid;         /* subsystem Vendor ID of chip */
709         uint16_t        m_ssid;         /* subsystem Device ID of chip */

711         uchar_t         m_sync_offset;  /* default offset for this chip. */

713         timeout_id_t    m_quiesce_timeid;

715         ddi_dma_handle_t m_dma_req_frame_hdl;
716         ddi_acc_handle_t m_acc_req_frame_hdl;
717         ddi_dma_handle_t m_dma_reply_frame_hdl;
718         ddi_acc_handle_t m_acc_reply_frame_hdl;
719         ddi_dma_handle_t m_dma_free_queue_hdl;
720         ddi_acc_handle_t m_acc_free_queue_hdl;
721         ddi_dma_handle_t m_dma_post_queue_hdl;
722         ddi_acc_handle_t m_acc_post_queue_hdl;

724         /*
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         /*
725          * list of reset notification requests
726          */
727         struct scsi_reset_notify_entry  *m_reset_notify_listf;

729         /*
730          * qfull handling
731          */
732         timeout_id_t    m_restart_cmd_timeid;

734         /*
735          * scsi reset delay per bus
736          */
737         uint_t          m_scsi_reset_delay;

739         int             m_pm_idle_delay;

741         uchar_t         m_polled_intr;  /* intr was polled. */
742         uchar_t         m_suspended;    /* true if driver is suspended */

744         struct kmem_cache *m_kmem_cache;
745         struct kmem_cache *m_cache_frames;

747         /*
748          * hba options.
749          */
750         uint_t          m_options;

752         int             m_in_callback;

754         int             m_power_level;  /* current power level */

756         int             m_busy;         /* power management busy state */
```

```
758         off_t           m_pmcsr_offset; /* PMCSR offset */

760         ddi_acc_handle_t m_config_handle;

762         ddi_dma_attr_t          m_io_dma_attr;  /* Used for data I/O */
763         ddi_dma_attr_t          m_msg_dma_attr; /* Used for message frames */
764         ddi_device_acc_attr_t   m_dev_acc_attr;
765         ddi_device_acc_attr_t   m_reg_acc_attr;

767         /*
768          * request/reply variables
769          */
770         caddr_t         m_req_frame;
771         uint64_t        m_req_frame_dma_addr;
772         caddr_t         m_reply_frame;
773         uint64_t        m_reply_frame_dma_addr;
774         caddr_t         m_free_queue;
775         uint64_t        m_free_queue_dma_addr;
776         caddr_t         m_post_queue;
777         uint64_t        m_post_queue_dma_addr;

779         m_replyh_arg_t *m_replyh_args;

781         uint16_t        m_max_requests;
782         uint16_t        m_req_frame_size;

784         /*
785          * Max frames per request reprted in IOC Facts
786          */
787         uint8_t         m_max_chain_depth;
788         /*
789          * Max frames per request which is used in reality. It's adjusted
790          * according DMA SG length attribute, and shall not exceed the
791          * m_max_chain_depth.
792          */
793         uint8_t         m_max_request_frames;

795         uint16_t        m_free_queue_depth;
796         uint16_t        m_post_queue_depth;
797         uint16_t        m_max_replies;
798         uint32_t        m_free_index;
799         uint32_t        m_post_index;
800         uint8_t         m_reply_frame_size;
801         uint32_t        m_ioc_capabilities;

803         /*
804          * indicates if the firmware was upload by the driver
805          * at boot time
806          */
807         ushort_t        m_fwupload;

809         uint16_t        m_productid;

811         /*
812          * per instance data structures for dma memory resources for
813          * MPI handshake protocol. only one handshake cmd can run at a time.
814          */
815         ddi_dma_handle_t        m_hshk_dma_hdl;

816         ddi_acc_handle_t        m_hshk_acc_hdl;

817         caddr_t                 m_hshk_memp;

818         size_t                  m_hshk_dma_size;

820         /* Firmware version on the card at boot time */
```



```
821         uint32_t                m_fwversion;

823         /* MSI specific fields */
824         ddi_intr_handle_t       *m_htable;      /* For array of interrupts */
825         int                     m_intr_type;    /* What type of interrupt */
826         int                     m_intr_cnt;     /* # of intrs count returned */
827         size_t                  m_intr_size;    /* Size of intr array */
828         uint_t                  m_intr_pri;     /* Interrupt priority   */
829         int                     m_intr_cap;     /* Interrupt capabilities */
830         ddi_taskq_t             *m_event_taskq;

832         /* SAS specific information */

834         union {
835                 uint64_t        m_base_wwid;    /* Base WWID */
836                 struct {
837 #ifdef _BIG_ENDIAN
838                         uint32_t        m_base_wwid_hi;
839                         uint32_t        m_base_wwid_lo;
840 #else
841                         uint32_t        m_base_wwid_lo;
842                         uint32_t        m_base_wwid_hi;
843 #endif
844                 } sasaddr;
845         } un;

847         uint8_t                 m_num_phys;             /* # of PHYs */
848         mptsas_phy_info_t       m_phy_info[MPTSAS_MAX_PHYS];
849         uint8_t                 m_port_chng;    /* initiator port changes */
850         MPI2_CONFIG_PAGE_MAN_0   m_MANU_page0;   /* Manufactor page 0 info */
851         MPI2_CONFIG_PAGE_MAN_1   m_MANU_page1;   /* Manufactor page 1 info */

853         /* FMA Capabilities */
854         int                     m_fm_capabilities;
855         ddi_taskq_t             *m_dr_taskq;
856         int                     m_mpxio_enable;
857         uint8_t                 m_done_traverse_dev;
858         uint8_t                 m_done_traverse_smp;
859         int                     m_diag_action_in_progress;
860         uint16_t                m_dev_handle;
861         uint16_t                m_smp_devhdl;

863         /*
864          * Event recording
865          */
866         uint8_t                 m_event_index;
867         uint32_t                m_event_number;
868         uint32_t                m_event_mask[4];
869         mptsas_event_entry_t    m_events[MPTSAS_EVENT_QUEUE_SIZE];

871         /*
872          * FW diag Buffer List
873          */
874         mptsas_fw_diagnostic_buffer_t
875                 m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_COUNT];

877         /*
878          * Event Replay flag (MUR support)
879          */
880         uint8_t                 m_event_replay;

882         /*
883          * IR Capable flag
884          */
885         uint8_t                 m_ir_capable;
```

```
 887         /*
 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         /*
 888          * Is HBA processing a diag reset?
 889          */
 890         uint8_t                 m_in_reset;

 892         /*
 893          * per instance cmd data structures for task management cmds
 894          */
 895         m_event_struct_t        m_event_task_mgmt;      /* must be last */
 896                                                         /* ... scsi_pkt_size */
 897 } mptsas_t;
```

_____unchanged_portion_omitted_