```
**********************************************************
  414487 Fri Aug 31 14:55:20 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
re #7364 rb2201 "hddisco" hangs after unplugging both cables from JBOD (and NMS
re #8346 rb2639 KT disk failures
re #8346 rb2639 KT disk failures
re #10443 rb3479 3.1.3 crash: BAD TRAP: type=e (#pf Page fault)
**********************************************************
```

```
/*
 * 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/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>

/*
 * autoconfiguration data and routines.
 */
static int mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
static int mptsas_power(dev_info_t *dip, int component, int level);

/*
 * cb_ops function
 */
static int mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
        cred_t *credp, int *rval);
#ifdef __sparc
static int mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd);
#else  /* __sparc */
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);
229 static void mptsas_handle_scsi_io_success(mptsas_t *mpt,
230     pMpi2ReplyDescriptorsUnion_t reply_desc);
231 static void mptsas_handle_address_reply(mptsas_t *mpt,
232     pMpi2ReplyDescriptorsUnion_t reply_desc);
233 static int mptsas_wait_intr(mptsas_t *mpt, int polltime);
234 static void mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd,
235     uint32_t *control, pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl);

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

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

249 static uint8_t mptsas_get_fw_diag_buffer_number(mptsas_t *mpt,
250     uint32_t unique_id);
251 static void mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd);
252 static int mptsas_post_fw_diag_buffer(mptsas_t *mpt,
253     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code);
254 static int mptsas_release_fw_diag_buffer(mptsas_t *mpt,
255     mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code,
256     uint32_t diag_type);
```

```
257 static int mptsas_diag_register(mptsas_t *mpt,
258     mptsas_fw_diag_register_t *diag_register, uint32_t *return_code);
259 static int mptsas_diag_unregister(mptsas_t *mpt,
260     mptsas_fw_diag_unregister_t *diag_unregister, uint32_t *return_code);
261 static int mptsas_diag_query(mptsas_t *mpt, mptsas_fw_diag_query_t *diag_query,
262     uint32_t *return_code);
263 static int mptsas_diag_read_buffer(mptsas_t *mpt,
264     mptsas_diag_read_buffer_t *diag_read_buffer, uint8_t *ioctl_buf,
265     uint32_t *return_code, int ioctl_mode);
266 static int mptsas_diag_release(mptsas_t *mpt,
267     mptsas_fw_diag_release_t *diag_release, uint32_t *return_code);
268 static int mptsas_do_diag_action(mptsas_t *mpt, uint32_t action,
269     uint8_t *diag_action, uint32_t length, uint32_t *return_code,
270     int ioctl_mode);
271 static int mptsas_diag_action(mptsas_t *mpt, mptsas_diag_action_t *data,
272     int mode);

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

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

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

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

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

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

300 static void mptsas_deliver_doneq_thread(mptsas_t *mpt);
301 static void mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
302 static void mptsas_doneq_mv(mptsas_t *mpt, uint64_t t);

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

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


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

319 /*
320  * helper functions
321  */
322 static void mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);
```



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

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

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

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

347 static int mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
348     uint16_t *handle, mptsas_target_t **pptgt);
349 static void mptsas_update_phymask(mptsas_t *mpt);

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


357 /*
358  * Enumeration / DR functions
359  */
360 static void mptsas_config_all(dev_info_t *pdip);
361 static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
362     dev_info_t **lundip);
363 static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
364     dev_info_t **lundip);

366 static int mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt);
367 static int mptsas_offline_target(dev_info_t *pdip, char *name);

369 static int mptsas_config_raid(dev_info_t *pdip, uint16_t target,
370     dev_info_t **dip);

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

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

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

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

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

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

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

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

425 /*
426  * Power management functions
427  */
428 static int mptsas_get_pci_cap(mptsas_t *mpt);
429 static int mptsas_init_pm(mptsas_t *mpt);

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

439 static int mptsas_register_intrs(mptsas_t *);
440 static void mptsas_unregister_intrs(mptsas_t *);
441 static int mptsas_add_intrs(mptsas_t *, int);
442 static void mptsas_rem_intrs(mptsas_t *);

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



```
451 extern pri_t minclsyspri, maxclsyspri;

453 /*
454  * This device is created by the SCSI pseudo nexus driver (SCSI vHCI).  It is
455  * under this device that the paths to a physical device are created when
456  * MPxIO is used.
457  */
458 extern dev_info_t       *scsi_vhci_dip;

460 /*
461  * Tunable timeout value for Inquiry VPD page 0x83
462  * By default the value is 30 seconds.
463  */
464 int mptsas_inq83_retry_timeout = 30;
465 /*
466  * Maximum number of command timeouts (0 - 255) considered acceptable.
467  */
468 int mptsas_timeout_threshold = 2;
469 /*
470  * Timeouts exceeding threshold within this period are considered excessive.
471  */
472 int mptsas_timeout_interval = 30;

474 /*
475  * This is used to allocate memory for message frame storage, not for
476  * data I/O DMA. All message frames must be stored in the first 4G of
477  * physical memory.
478  */
479 ddi_dma_attr_t mptsas_dma_attrs = {
480         DMA_ATTR_V0,    /* attribute layout version             */
481         0x0ull,         /* address low - should be 0 (longlong) */
482         0xffffffffull,  /* address high - 32-bit max range      */
483         0x00ffffffull,  /* count max - max DMA object size      */
484         4,              /* allocation alignment requirements    */
485         0x78,           /* burstsizes - binary encoded values   */
486         1,              /* minxfer - gran. of DMA engine        */
487         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
488         0xffffffffull,  /* max segment size (DMA boundary)      */
489         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
490         512,            /* granularity - device transfer size   */
491         0               /* flags, set to 0                      */
492 };
_____unchanged_portion_omitted_

2623 static void
2624 mptsas_alloc_reply_args(mptsas_t *mpt)
2625 {
2626         if (mpt->m_replyh_args == NULL) {
2621         if (mpt->m_replyh_args != NULL) {
2622                 kmem_free(mpt->m_replyh_args, sizeof (m_replyh_arg_t)
2623                     * mpt->m_max_replies);
2624                 mpt->m_replyh_args = NULL;
2625         }
2627                 mpt->m_replyh_args = kmem_zalloc(sizeof (m_replyh_arg_t) *
2628                     mpt->m_max_replies, KM_SLEEP);
2629         }
2630 }
_____unchanged_portion_omitted_

4704 static void
4705 mptsas_handle_address_reply(mptsas_t *mpt,
4706     pMpi2ReplyDescriptorsUnion_t reply_desc)
4707 {
4708         pMpi2AddressReplyDescriptor_t   address_reply;
4709         pMPI2DefaultReply_t             reply;
4710         mptsas_fw_diagnostic_buffer_t   *pBuffer;
```

```
4711        uint32_t                        reply_addr;
4712        uint16_t                        SMID, iocstatus;
4713        mptsas_slots_t                  *slots = mpt->m_active;
4714        mptsas_cmd_t                    *cmd = NULL;
4715        uint8_t                         function, buffer_type;
4716        m_replyh_arg_t                  *args;
4717        int                             reply_frame_no;

4719        ASSERT(mutex_owned(&mpt->m_mutex));

4721        address_reply = (pMpi2AddressReplyDescriptor_t)reply_desc;
4722        reply_addr = ddi_get32(mpt->m_acc_post_queue_hdl,
4723            &address_reply->ReplyFrameAddress);
4724        SMID = ddi_get16(mpt->m_acc_post_queue_hdl, &address_reply->SMID);

4726        /*
4727         * If reply frame is not in the proper range we should ignore this
4728         * message and exit the interrupt handler.
4729         */
4730        if ((reply_addr < mpt->m_reply_frame_dma_addr) ||
4731            (reply_addr >= (mpt->m_reply_frame_dma_addr +
4732            (mpt->m_reply_frame_size * mpt->m_max_replies))) ||
4733            ((reply_addr - mpt->m_reply_frame_dma_addr) %
4734            mpt->m_reply_frame_size != 0)) {
4735                mptsas_log(mpt, CE_WARN, "?Received invalid reply frame "
4736                    "address 0x%x\n", reply_addr);
4737                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
4738                return;
4739        }

4741        (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
4742            DDI_DMA_SYNC_FORCPU);
4743        reply = (pMPI2DefaultReply_t)(mpt->m_reply_frame + (reply_addr -
4744            mpt->m_reply_frame_dma_addr));
4745        function = ddi_get8(mpt->m_acc_reply_frame_hdl, &reply->Function);

4747        /*
4748         * don't get slot information and command for events since these values
4749         * don't exist
4750         */
4751        if ((function != MPI2_FUNCTION_EVENT_NOTIFICATION) &&
4752            (function != MPI2_FUNCTION_DIAG_BUFFER_POST)) {
4753                /*
4754                 * This could be a TM reply, which use the last allocated SMID,
4755                 * so allow for that.
4756                 */
4757                if ((SMID == 0) || (SMID > (slots->m_n_slots + 1))) {
4758                        mptsas_log(mpt, CE_WARN, "?Received invalid SMID of "
4759                            "%d\n", SMID);
4760                        ddi_fm_service_impact(mpt->m_dip,
4761                            DDI_SERVICE_UNAFFECTED);
4762                        return;
4763                }

4765                cmd = slots->m_slot[SMID];

4767                /*
4768                 * print warning and return if the slot is empty
4769                 */
4770                if (cmd == NULL) {
4771                        mptsas_log(mpt, CE_WARN, "?NULL command for address "
4772                            "reply in slot %d", SMID);
4773                        return;
4774                }
4775                if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
4776                    (cmd->cmd_flags & CFLAG_CONFIG) ||
```



```
4777                    (cmd->cmd_flags & CFLAG_FW_DIAG)) {
4778                        cmd->cmd_rfm = reply_addr;
4779                        cmd->cmd_flags |= CFLAG_FINISHED;
4780                        cv_broadcast(&mpt->m_passthru_cv);
4781                        cv_broadcast(&mpt->m_config_cv);
4782                        cv_broadcast(&mpt->m_fw_diag_cv);
4783                        return;
4784                } else if (!(cmd->cmd_flags & CFLAG_FW_CMD)) {
4785                        mptsas_remove_cmd(mpt, cmd);
4786                }
4787                NDBG31(("\t\tmptsas_process_intr: slot=%d", SMID));
4788        }
4789        /*
4790         * Depending on the function, we need to handle
4791         * the reply frame (and cmd) differently.
4792         */
4793        switch (function) {
4794        case MPI2_FUNCTION_SCSI_IO_REQUEST:
4795                mptsas_check_scsi_io_error(mpt, (pMpi2SCSIIOReply_t)reply, cmd);
4796                break;
4797        case MPI2_FUNCTION_SCSI_TASK_MGMT:
4798                cmd->cmd_rfm = reply_addr;
4799                mptsas_check_task_mgt(mpt, (pMpi2SCSIManagementReply_t)reply,
4800                    cmd);
4801                break;
4802        case MPI2_FUNCTION_FW_DOWNLOAD:
4803                cmd->cmd_flags |= CFLAG_FINISHED;
4804                cv_signal(&mpt->m_fw_cv);
4805                break;
4806        case MPI2_FUNCTION_EVENT_NOTIFICATION:
4807                reply_frame_no = (reply_addr - mpt->m_reply_frame_dma_addr) /
4808                    mpt->m_reply_frame_size;
4809                args = &mpt->m_replyh_args[reply_frame_no];
4810                args->mpt = (void *)mpt;
4811                args->rfm = reply_addr;

4813                /*
4814                 * Record the event if its type is enabled in
4815                 * this mpt instance by ioctl.
4816                 */
4817                mptsas_record_event(args);

4819                /*
4820                 * Handle time critical events
4821                 * NOT_RESPONDING/ADDED only now
4822                 */
4823                if (mptsas_handle_event_sync(args) == DDI_SUCCESS) {
4824                        /*
4825                         * Would not return main process,
4826                         * just let taskq resolve ack action
4827                         * and ack would be sent in taskq thread
4828                         */
4829                        NDBG20(("send mptsas_handle_event_sync success"));
4830                }

4832                if (mpt->m_in_reset) {
4833                        NDBG20(("dropping event received during reset"));
4834                        return;
4835                }

4837                if ((ddi_taskq_dispatch(mpt->m_event_taskq, mptsas_handle_event,
4838                    (void *)args, DDI_NOSLEEP)) != DDI_SUCCESS) {
4839                        mptsas_log(mpt, CE_WARN, "No memory available"
4840                        "for dispatch taskq");
4841                        /*
4842                         * Return the reply frame to the free queue.
```

```
4843                         */
4844                        ddi_put32(mpt->m_acc_free_queue_hdl,
4845                            &((uint32_t *)(void *)
4846                            mpt->m_free_queue)[mpt->m_free_index], reply_addr);
4847                        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4848                            DDI_DMA_SYNC_FORDEV);
4849                        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4850                                mpt->m_free_index = 0;
4851                        }
4853                        ddi_put32(mpt->m_datap,
4854                            &mpt->m_reg->ReplyFreeHostIndex, mpt->m_free_index);
4855                }
4856                return;
4857        case MPI2_FUNCTION_DIAG_BUFFER_POST:
4858                /*
4859                 * If SMID is 0, this implies that the reply is due to a
4860                 * release function with a status that the buffer has been
4861                 * released.  Set the buffer flags accordingly.
4862                 */
4863                if (SMID == 0) {
4864                        iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
4865                            &reply->IOCStatus);
4866                        buffer_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
4867                            &(((pMpi2DiagBufferPostReply_t)reply)->BufferType));
4868                        if (iocstatus == MPI2_IOCSTATUS_DIAGNOSTIC_RELEASED) {
4869                                pBuffer =
4870                                    &mpt->m_fw_diag_buffer_list[buffer_type];
4871                                pBuffer->valid_data = TRUE;
4872                                pBuffer->owned_by_firmware = FALSE;
4873                                pBuffer->immediate = FALSE;
4874                        }
4875                } else {
4876                        /*
4877                         * Normal handling of diag post reply with SMID.
4878                         */
4879                        cmd = slots->m_slot[SMID];
4881                        /*
4882                         * print warning and return if the slot is empty
4883                         */
4884                        if (cmd == NULL) {
4885                                mptsas_log(mpt, CE_WARN, "?NULL command for "
4886                                    "address reply in slot %d", SMID);
4887                                return;
4888                        }
4889                        cmd->cmd_rfm = reply_addr;
4890                        cmd->cmd_flags |= CFLAG_FINISHED;
4891                        cv_broadcast(&mpt->m_fw_diag_cv);
4892                }
4893                return;
4894        default:
4895                mptsas_log(mpt, CE_WARN, "Unknown function 0x%x ", function);
4896                break;
4897        }
4899        /*
4900         * Return the reply frame to the free queue.
4901         */
4902        ddi_put32(mpt->m_acc_free_queue_hdl,
4903            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
4904            reply_addr);
4905        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
4906            DDI_DMA_SYNC_FORDEV);
4907        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
4908                mpt->m_free_index = 0;
```



```
4909        }
4910        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
4911            mpt->m_free_index);
4913        if (cmd->cmd_flags & CFLAG_FW_CMD)
4914                return;
4916        if (cmd->cmd_flags & CFLAG_RETRY) {
4917                /*
4918                 * The target returned QFULL or busy, do not add tihs
4919                 * pkt to the doneq since the hba will retry
4920                 * this cmd.
4921                 *
4922                 * The pkt has already been resubmitted in
4923                 * mptsas_handle_qfull() or in mptsas_check_scsi_io_error().
4924                 * Remove this cmd_flag here.
4925                 */
4926                cmd->cmd_flags &= ~CFLAG_RETRY;
4927        } else {
4928                mptsas_doneq_add(mpt, cmd);
4929        }
4930 }
_____unchanged_portion_omitted_
5608 /*
5609  * mptsas_handle_dr is a task handler for DR, the DR action includes:
5610  * 1. Directly attched Device Added/Removed.
5611  * 2. Expander Device Added/Removed.
5612  * 3. Indirectly Attached Device Added/Expander.
5613  * 4. LUNs of a existing device status change.
5614  * 5. RAID volume created/deleted.
5615  * 6. Member of RAID volume is released because of RAID deletion.
5616  * 7. Physical disks are removed because of RAID creation.
5617  */
5618 static void
5619 mptsas_handle_dr(void *args) {
5620        mptsas_topo_change_list_t       *topo_node = NULL;
5621        mptsas_topo_change_list_t       *save_node = NULL;
5622        mptsas_t                        *mpt;
5623        dev_info_t                      *parent = NULL;
5624        mptsas_phymask_t                phymask = 0;
5625        char                            *phy_mask_name;
5626        uint8_t                         flags = 0, physport = 0xff;
5627        uint8_t                         port_update = 0;
5628        uint_t                          event;
5630        topo_node = (mptsas_topo_change_list_t *)args;
5632        mpt = topo_node->mpt;
5633        event = topo_node->event;
5634        flags = topo_node->flags;
5636        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
5638        NDBG20(("mptsas%d handle_dr enter", mpt->m_instance));
5640        switch (event) {
5641        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5642                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5643                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE) ||
5644                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
5645                        /*
5646                         * Direct attached or expander attached device added
5647                         * into system or a Phys Disk that is being unhidden.
5648                         */
5649                        port_update = 1;
```

```
5650                }
5651                break;
5652        case MPTSAS_DR_EVENT_RECONFIG_SMP:
5653                /*
5654                 * New expander added into system, it must be the head
5655                 * of topo_change_list_t
5656                 */
5657                port_update = 1;
5658                break;
5659        default:
5660                port_update = 0;
5661                break;
5662        }
5663        /*
5664         * All cases port_update == 1 may cause initiator port form change
5665         */
5666        mutex_enter(&mpt->m_mutex);
5667        if (mpt->m_port_chng && port_update) {
5668                /*
5669                 * mpt->m_port_chng flag indicates some PHYs of initiator
5670                 * port have changed to online. So when expander added or
5671                 * directly attached device online event come, we force to
5672                 * update port information by issueing SAS IO Unit Page and
5673                 * update PHYMASKs.
5674                 */
5675                (void) mptsas_update_phymask(mpt);
5676                mpt->m_port_chng = 0;

5678        }
5679        mutex_exit(&mpt->m_mutex);
5680        while (topo_node) {
5681                phymask = 0;
5682                if (parent == NULL) {
5683                        physport = topo_node->un.physport;
5684                        event = topo_node->event;
5685                        flags = topo_node->flags;
5686                        if (event & (MPTSAS_DR_EVENT_OFFLINE_TARGET |
5687                            MPTSAS_DR_EVENT_OFFLINE_SMP)) {
5688                                /*
5689                                 * For all offline events, phymask is known
5690                                 */
5691                                phymask = topo_node->un.phymask;
5692                                goto find_parent;
5693                        }
5694                        if (event & MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
5695                                goto handle_topo_change;
5696                        }
5697                        if (flags & MPTSAS_TOPO_FLAG_LUN_ASSOCIATED) {
5698                                phymask = topo_node->un.phymask;
5699                                goto find_parent;
5700                        }

5702                        if ((flags ==
5703                            MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) &&
5704                            (event == MPTSAS_DR_EVENT_RECONFIG_TARGET)) {
5705                                /*
5706                                 * There is no any field in IR_CONFIG_CHANGE
5707                                 * event indicate physport/phynum, let's get
5708                                 * parent after SAS Device Page0 request.
5709                                 */
5710                                goto handle_topo_change;
5711                        }

5713                        mutex_enter(&mpt->m_mutex);
5714                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5715                                /*
```



```
5716                                 * If the direct attached device added or a
5717                                 * phys disk is being unhidden, argument
5718                                 * physport actually is PHY#, so we have to get
5719                                 * phymask according PHY#.
5720                                 */
5721                                physport = mpt->m_phy_info[physport].port_num;
5722                        }

5724                        /*
5725                         * Translate physport to phymask so that we can search
5726                         * parent dip.
5727                         */
5728                        phymask = mptsas_physport_to_phymask(mpt,
5729                            physport);
5730                        mutex_exit(&mpt->m_mutex);

5732 find_parent:
5733                        bzero(phy_mask_name, MPTSAS_MAX_PHYS);
5734                        /*
5735                         * For RAID topology change node, write the iport name
5736                         * as v0.
5737                         */
5738                        if (flags & MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5739                                (void) sprintf(phy_mask_name, "v0");
5740                        } else {
5741                                /*
5742                                 * phymask can bo 0 if the drive has been
5743                                 * pulled by the time an add event is
5744                                 * processed.  If phymask is 0, just skip this
5745                                 * event and continue.
5746                                 */
5747                                if (phymask == 0) {
5748                                        mutex_enter(&mpt->m_mutex);
5749                                        save_node = topo_node;
5750                                        topo_node = topo_node->next;
5751                                        ASSERT(save_node);
5752                                        kmem_free(save_node,
5753                                            sizeof (mptsas_topo_change_list_t));
5754                                        mutex_exit(&mpt->m_mutex);

5756                                        parent = NULL;
5757                                        continue;
5758                                }
5759                                (void) sprintf(phy_mask_name, "%x", phymask);
5760                        }
5761                        parent = scsi_hba_iport_find(mpt->m_dip,
5762                            phy_mask_name);
5763                        if (parent == NULL) {
5764                                mptsas_log(mpt, CE_WARN, "Failed to find an "
5765                                    "iport, should not happen!");
5766                                goto out;
5767                        }

5769                }
5770                ASSERT(parent);
5771 handle_topo_change:

5773                mutex_enter(&mpt->m_mutex);
5774                /*
5775                 * If HBA is being reset, don't perform operations depending
5776                 * on the IOC. We must free the topo list, however.
5777                 */
5778                if (!mpt->m_in_reset)

5779                        mptsas_handle_topo_change(topo_node, parent);
5780                else
```

```
5781                        NDBG20(("skipping topo change received during reset"));
5782                save_node = topo_node;
5783                topo_node = topo_node->next;
5784                ASSERT(save_node);
5785                kmem_free(save_node, sizeof (mptsas_topo_change_list_t));
5786                mutex_exit(&mpt->m_mutex);

5788                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
5789                    (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) ||
5790                    (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED)) {
5791                        /*
5792                         * If direct attached device associated, make sure
5793                         * reset the parent before start the next one. But
5794                         * all devices associated with expander shares the
5795                         * parent.  Also, reset parent if this is for RAID.
5796                         */
5797                        parent = NULL;
5798                }
5799        }
5800 out:
5801        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5802 }

5804 static void
5805 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
5806     dev_info_t *parent)
5807 {
5808        mptsas_target_t *ptgt = NULL;
5809        mptsas_smp_t    *psmp = NULL;
5810        mptsas_t        *mpt = (void *)topo_node->mpt;
5811        uint16_t        devhdl;
5812        uint16_t        attached_devhdl;
5813        uint64_t        sas_wwn = 0;
5814        int             rval = 0;
5815        uint32_t        page_address;
5816        uint8_t         phy, flags;
5817        char            *addr = NULL;
5818        dev_info_t      *lundip;
5819        int             circ = 0, circ1 = 0;
5820        char            attached_wwnstr[MPTSAS_WWN_STRLEN];

5822        NDBG20(("mptsas%d handle_topo_change enter", mpt->m_instance));

5824        ASSERT(mutex_owned(&mpt->m_mutex));

5826        switch (topo_node->event) {
5827        case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5828        {
5829                char *phy_mask_name;
5830                mptsas_phymask_t phymask = 0;

5832                if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5833                        /*
5834                         * Get latest RAID info.
5835                         */
5836                        (void) mptsas_get_raid_info(mpt);
5837                        ptgt = mptsas_search_by_devhdl(
5838                            &mpt->m_active->m_tgttbl, topo_node->devhdl);
5839                        if (ptgt == NULL)
5840                                break;
5841                } else {
5842                        ptgt = (void *)topo_node->object;
5843                }

5845                if (ptgt == NULL) {
5846                        /*
```



```
5847                         * If a Phys Disk was deleted, RAID info needs to be
5848                         * updated to reflect the new topology.
5849                         */
5850                        (void) mptsas_get_raid_info(mpt);

5852                        /*
5853                         * Get sas device page 0 by DevHandle to make sure if
5854                         * SSP/SATA end device exist.
5855                         */
5856                        page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
5857                            MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
5858                            topo_node->devhdl;

5860                        rval = mptsas_get_target_device_info(mpt, page_address,
5861                            &devhdl, &ptgt);
5862                        if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
5863                                mptsas_log(mpt, CE_NOTE,
5864                                    "mptsas_handle_topo_change: target %d is "
5865                                    "not a SAS/SATA device. \n",
5866                                    topo_node->devhdl);
5867                        } else if (rval == DEV_INFO_FAIL_ALLOC) {
5868                                mptsas_log(mpt, CE_NOTE,
5869                                    "mptsas_handle_topo_change: could not "
5870                                    "allocate memory. \n");
5871                        }
5872                        /*
5873                         * If rval is DEV_INFO_PHYS_DISK than there is nothing
5874                         * else to do, just leave.
5875                         */
5876                        if (rval != DEV_INFO_SUCCESS) {
5877                                return;
5878                        }
5879                }

5881                ASSERT(ptgt->m_devhdl == topo_node->devhdl);

5883                mutex_exit(&mpt->m_mutex);
5884                flags = topo_node->flags;

5886                if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
5887                        phymask = ptgt->m_phymask;
5888                        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
5889                        (void) sprintf(phy_mask_name, "%x", phymask);
5890                        parent = scsi_hba_iport_find(mpt->m_dip,
5891                            phy_mask_name);
5892                        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5893                        if (parent == NULL) {
5894                                mptsas_log(mpt, CE_WARN, "Failed to find a "
5895                                    "iport for PD, should not happen!");
5896                                mutex_enter(&mpt->m_mutex);
5897                                break;
5898                        }
5899                }

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

```
5913                        /*
5914                         * release nexus for bus configure
5915                         */
5916                        ndi_devi_exit(parent, circ1);
5917                        ndi_devi_exit(scsi_vhci_dip, circ);

5919                        /*
5920                         * Add parent's props for SMHBA support
5921                         */
5922                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5923                                bzero(attached_wwnstr,
5924                                    sizeof (attached_wwnstr));
5925                                (void) sprintf(attached_wwnstr, "w%016"PRIx64,
5926                                    ptgt->m_sas_wwn);
5927                                if (ddi_prop_update_string(DDI_DEV_T_NONE,
5928                                    parent,
5929                                    SCSI_ADDR_PROP_ATTACHED_PORT,
5930                                    attached_wwnstr)
5931                                    != DDI_PROP_SUCCESS) {
5932                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5933                                            parent,
5934                                            SCSI_ADDR_PROP_ATTACHED_PORT);
5935                                        mptsas_log(mpt, CE_WARN, "Failed to"
5936                                            "attached-port props");
5937                                        return;
5938                                }
5939                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5940                                    MPTSAS_NUM_PHYS, 1) !=
5941                                    DDI_PROP_SUCCESS) {
5942                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5943                                            parent, MPTSAS_NUM_PHYS);
5944                                        mptsas_log(mpt, CE_WARN, "Failed to"
5945                                            " create num-phys props");
5946                                        return;
5947                                }

5949                                /*
5950                                 * Update PHY info for smhba
5951                                 */
5952                                mutex_enter(&mpt->m_mutex);
5953                                if (mptsas_smhba_phy_init(mpt)) {
5954                                        mutex_exit(&mpt->m_mutex);
5955                                        mptsas_log(mpt, CE_WARN, "mptsas phy"
5956                                            " update failed");
5957                                        return;
5958                                }
5959                                mutex_exit(&mpt->m_mutex);
5960                                mptsas_smhba_set_phy_props(mpt,
5961                                    ddi_get_name_addr(parent), parent,
5962                                    1, &attached_devhdl);
5963                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5964                                    MPTSAS_VIRTUAL_PORT, 0) !=
5965                                    DDI_PROP_SUCCESS) {
5966                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5967                                            parent, MPTSAS_VIRTUAL_PORT);
5968                                        mptsas_log(mpt, CE_WARN,
5969                                            "mptsas virtual-port"
5970                                            "port prop update failed");
5971                                        return;
5972                                }
5973                        }
5974                }
5975                mutex_enter(&mpt->m_mutex);

5977                NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
5978                    "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
```



```
5979                        ptgt->m_phymask));
5980                break;
5981        }
5982        case MPTSAS_DR_EVENT_OFFLINE_TARGET:
5983        {
5984                mptsas_hash_table_t *tgttbl = &mpt->m_active->m_tgttbl;
5985                devhdl = topo_node->devhdl;
5986                ptgt = mptsas_search_by_devhdl(tgttbl, devhdl);
5987                if (ptgt == NULL)
5988                        break;

5990                sas_wwn = ptgt->m_sas_wwn;
5991                phy = ptgt->m_phynum;

5993                addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

5995                if (sas_wwn) {
5996                        (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
5997                } else {
5998                        (void) sprintf(addr, "p%x", phy);
5999                }
6000                ASSERT(ptgt->m_devhdl == devhdl);

6002                if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
6003                    (topo_node->flags ==
6004                    MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
6005                        /*
6006                         * Get latest RAID info if RAID volume status changes
6007                         * or Phys Disk status changes
6008                         */
6009                        (void) mptsas_get_raid_info(mpt);
6010                }
6011                /*
6012                 * Abort all outstanding command on the device
6013                 */
6014                rval = mptsas_do_scsi_reset(mpt, devhdl);
6015                if (rval) {
6016                        NDBG20(("mptsas%d handle_topo_change to reset target "
6017                            "before offline devhdl:%x, phymask:%x, rval:%x",
6018                            mpt->m_instance, ptgt->m_devhdl, ptgt->m_phymask,
6019                            rval));
6020                }

6022                mutex_exit(&mpt->m_mutex);

6024                ndi_devi_enter(scsi_vhci_dip, &circ);
6025                ndi_devi_enter(parent, &circ1);
6026                rval = mptsas_offline_target(parent, addr);
6027                ndi_devi_exit(parent, circ1);
6028                ndi_devi_exit(scsi_vhci_dip, circ);
6029                NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6030                    "phymask:%x, rval:%x", mpt->m_instance,
6031                    ptgt->m_devhdl, ptgt->m_phymask, rval));

6033                kmem_free(addr, SCSI_MAXNAMELEN);

6035                /*
6036                 * Clear parent's props for SMHBA support
6037                 */
6038                flags = topo_node->flags;
6039                if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6040                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6041                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6042                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6043                            DDI_PROP_SUCCESS) {
6044                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
```

```
6045                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6046                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6047                                    "prop update failed");
6048                                break;
6049                        }
6050                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6051                            MPTSAS_NUM_PHYS, 0) !=
6052                            DDI_PROP_SUCCESS) {
6053                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6054                                    MPTSAS_NUM_PHYS);
6055                                mptsas_log(mpt, CE_WARN, "mptsas num phys "
6056                                    "prop update failed");
6057                                break;
6058                        }
6059                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6060                            MPTSAS_VIRTUAL_PORT, 1) !=
6061                            DDI_PROP_SUCCESS) {
6062                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6063                                    MPTSAS_VIRTUAL_PORT);
6064                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6065                                    "prop update failed");
6066                                break;
6067                        }
6068                }

6070                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                }
6071                if (rval == DDI_SUCCESS) {
6072                        mptsas_tgt_free(&mpt->m_active->m_tgttbl,
6073                            ptgt->m_sas_wwn, ptgt->m_phymask);
6074                        ptgt = NULL;
6075                } else {
6076                        /*
6077                         * clean DR_INTRANSITION flag to allow I/O down to
6078                         * PHCI driver since failover finished.
6079                         * Invalidate the devhdl
6080                         */
6081                        ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6082                        ptgt->m_tgt_unconfigured = 0;
6083                        mutex_enter(&mpt->m_tx_waitq_mutex);
6084                        ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
6085                        mutex_exit(&mpt->m_tx_waitq_mutex);
6086                }

6088                /*
6089                 * Send SAS IO Unit Control to free the dev handle
6090                 */
6091                if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6092                    (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
6093                        rval = mptsas_free_devhdl(mpt, devhdl);

6095                        NDBG20(("mptsas%d handle_topo_change to remove "
6096                            "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6097                            rval));
6098                }

6100                break;
6101        }
6102        case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
6103        {
6104                devhdl = topo_node->devhdl;
6105                /*
6106                 * If this is the remove handle event, do a reset first.
```



```
6107                 */
6108                if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6109                        rval = mptsas_do_scsi_reset(mpt, devhdl);
6110                        if (rval) {
6111                                NDBG20(("mpt%d reset target before remove "
6112                                    "devhdl:%x, rval:%x", mpt->m_instance,
6113                                    devhdl, rval));
6114                        }
6115                }

6117                /*
6118                 * Send SAS IO Unit Control to free the dev handle
6119                 */
6120                rval = mptsas_free_devhdl(mpt, devhdl);
6121                NDBG20(("mptsas%d handle_topo_change to remove "
6122                    "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6123                    rval));
6124                break;
6125        }
6126        case MPTSAS_DR_EVENT_RECONFIG_SMP:
6127        {
6128                mptsas_smp_t smp;
6129                dev_info_t *smpdip;
6130                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;

6132                devhdl = topo_node->devhdl;

6134                page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
6135                    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
6136                rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
6137                if (rval != DDI_SUCCESS) {
6138                        mptsas_log(mpt, CE_WARN, "failed to online smp, "
6139                            "handle %x", devhdl);
6140                        return;
6141                }

6143                psmp = mptsas_smp_alloc(smptbl, &smp);
6144                if (psmp == NULL) {
6145                        return;
6146                }

6148                mutex_exit(&mpt->m_mutex);
6149                ndi_devi_enter(parent, &circ1);
6150                (void) mptsas_online_smp(parent, psmp, &smpdip);
6151                ndi_devi_exit(parent, circ1);

6153                mutex_enter(&mpt->m_mutex);
6154                break;
6155        }
6156        case MPTSAS_DR_EVENT_OFFLINE_SMP:
6157        {
6158                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;
6159                devhdl = topo_node->devhdl;
6160                uint32_t dev_info;

6162                psmp = mptsas_search_by_devhdl(smptbl, devhdl);
6163                if (psmp == NULL)
6164                        break;
6165                /*
6166                 * The mptsas_smp_t data is released only if the dip is offlined
6167                 * successfully.
6168                 */
6169                mutex_exit(&mpt->m_mutex);

6171                ndi_devi_enter(parent, &circ1);
6172                rval = mptsas_offline_smp(parent, psmp, NDI_DEVI_REMOVE);
```

```
6173                ndi_devi_exit(parent, circ1);

6175                dev_info = psmp->m_deviceinfo;
6176                if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
6177                    DEVINFO_DIRECT_ATTACHED) {
6178                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6179                            MPTSAS_VIRTUAL_PORT, 1) !=
6180                            DDI_PROP_SUCCESS) {
6181                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6182                                    MPTSAS_VIRTUAL_PORT);
6183                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6184                                    "prop update failed");
6185                                return;
6186                        }
6187                        /*
6188                         * Check whether the smp connected to the iport,
6189                         */
6190                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6191                            MPTSAS_NUM_PHYS, 0) !=
6192                            DDI_PROP_SUCCESS) {
6193                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6194                                    MPTSAS_NUM_PHYS);
6195                                mptsas_log(mpt, CE_WARN, "mptsas num phys"
6196                                    "prop update failed");
6197                                return;
6198                        }
6199                        /*
6200                         * Clear parent's attached-port props
6201                         */
6202                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6203                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6204                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6205                            DDI_PROP_SUCCESS) {
6206                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6207                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6208                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6209                                    "prop update failed");
6210                                return;
6211                        }
6212                }

6214                mutex_enter(&mpt->m_mutex);
6215                NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6216                    "rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
6217                if (rval == DDI_SUCCESS) {
6218                        mptsas_smp_free(smptbl, psmp->m_sasaddr,
6219                            psmp->m_phymask);
6220                } else {
6221                        psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
6222                }

6224                bzero(attached_wwnstr, sizeof (attached_wwnstr));

6226                break;
6227        }
6228        default:
6229                return;
6230        }
6231 }
```

*unchanged_portion_omitted_*

```
6984 /*
6985  * handle events from ioc
6986  */
6987 static void
6988 mptsas_handle_event(void *args)
```



```
6989 {
6990        m_replyh_arg_t                  *replyh_arg;
6991        pMpi2EventNotificationReply_t   eventreply;
6992        uint32_t                        event, iocloginfo, rfm;
6993        uint32_t                        status;
6994        uint8_t                         port;
6995        mptsas_t                        *mpt;
6996        uint_t                          iocstatus;

6998        replyh_arg = (m_replyh_arg_t *)args;
6999        rfm = replyh_arg->rfm;
7000        mpt = replyh_arg->mpt;

7002        mutex_enter(&mpt->m_mutex);
7003        /*
7004         * If HBA is being reset, drop incoming event.
7005         */
7006        if (mpt->m_in_reset) {
7007                NDBG20(("dropping event received prior to reset"));
7008                mutex_exit(&mpt->m_mutex);
7009                return;
7010        }

7012        eventreply = (pMpi2EventNotificationReply_t)
7013            (mpt->m_reply_frame + (rfm - mpt->m_reply_frame_dma_addr));
7014        event = ddi_get16(mpt->m_acc_reply_frame_hdl, &eventreply->Event);

7016        if (iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
7017            &eventreply->IOCStatus)) {
7018                if (iocstatus == MPI2_IOCSTATUS_FLAG_LOG_INFO_AVAILABLE) {
7019                        mptsas_log(mpt, CE_WARN,
7020                            "!mptsas_handle_event: IOCStatus=0x%x, "
7021                            "IOCLogInfo=0x%x", iocstatus,
7022                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7023                            &eventreply->IOCLogInfo));
7024                } else {
7025                        mptsas_log(mpt, CE_WARN,
7026                            "mptsas_handle_event: IOCStatus=0x%x, "
7027                            "IOCLogInfo=0x%x", iocstatus,
7028                            ddi_get32(mpt->m_acc_reply_frame_hdl,
7029                            &eventreply->IOCLogInfo));
7030                }
7031        }

7033        /*
7034         * figure out what kind of event we got and handle accordingly
7035         */
7036        switch (event) {
7037        case MPI2_EVENT_LOG_ENTRY_ADDED:
7038                break;
7039        case MPI2_EVENT_LOG_DATA:
7040                iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7041                    &eventreply->IOCLogInfo);
7042                NDBG20(("mptsas %d log info %x received.\n", mpt->m_instance,
7043                    iocloginfo));
7044                break;
7045        case MPI2_EVENT_STATE_CHANGE:
7046                NDBG20(("mptsas%d state change.", mpt->m_instance));
7047                break;
7048        case MPI2_EVENT_HARD_RESET_RECEIVED:
7049                NDBG20(("mptsas%d event change.", mpt->m_instance));
7050                break;
7051        case MPI2_EVENT_SAS_DISCOVERY:
7052        {
7053                MPI2_EVENT_DATA_SAS_DISCOVERY   *sasdiscovery;
7054                char                            string[80];
```

```
7055                uint8_t                         rc;

7057                sasdiscovery =
7058                    (pMpi2EventDataSasDiscovery_t)eventreply->EventData;

7060                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7061                    &sasdiscovery->ReasonCode);
7062                port = ddi_get8(mpt->m_acc_reply_frame_hdl,
7063                    &sasdiscovery->PhysicalPort);
7064                status = ddi_get32(mpt->m_acc_reply_frame_hdl,
7065                    &sasdiscovery->DiscoveryStatus);

7067                string[0] = 0;
7068                switch (rc) {
7069                case MPI2_EVENT_SAS_DISC_RC_STARTED:
7070                        (void) sprintf(string, "STARTING");
7071                        break;
7072                case MPI2_EVENT_SAS_DISC_RC_COMPLETED:
7073                        (void) sprintf(string, "COMPLETED");
7074                        break;
7075                default:
7076                        (void) sprintf(string, "UNKNOWN");
7077                        break;
7078                }

7080                NDBG20(("SAS DISCOVERY is %s for port %d, status %x", string,
7081                    port, status));

7083                break;
7084        }
7085        case MPI2_EVENT_EVENT_CHANGE:
7086                NDBG20(("mptsas%d event change.", mpt->m_instance));
7087                break;
7088        case MPI2_EVENT_TASK_SET_FULL:
7089        {
7090                pMpi2EventDataTaskSetFull_t     taskfull;

7092                taskfull = (pMpi2EventDataTaskSetFull_t)eventreply->EventData;

7094                NDBG20(("TASK_SET_FULL received for mptsas%d, depth %d\n",
7095                    mpt->m_instance,  ddi_get16(mpt->m_acc_reply_frame_hdl,
7096                    &taskfull->CurrentDepth)));
7097                break;
7098        }
7099        case MPI2_EVENT_SAS_TOPOLOGY_CHANGE_LIST:
7100        {
7101                /*
7102                 * SAS TOPOLOGY CHANGE LIST Event has already been handled
7103                 * in mptsas_handle_event_sync() of interrupt context
7104                 */
7105                break;
7106        }
7107        case MPI2_EVENT_SAS_ENCL_DEVICE_STATUS_CHANGE:
7108        {
7109                pMpi2EventDataSasEnclDevStatusChange_t  encstatus;
7110                uint8_t                                 rc;
7111                char                                    string[80];

7113                encstatus = (pMpi2EventDataSasEnclDevStatusChange_t)
7114                    eventreply->EventData;

7116                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7117                    &encstatus->ReasonCode);
7118                switch (rc) {
7119                case MPI2_EVENT_SAS_ENCL_RC_ADDED:
7120                        (void) sprintf(string, "added");
```



```
7121                        break;
7122                case MPI2_EVENT_SAS_ENCL_RC_NOT_RESPONDING:
7123                        (void) sprintf(string, ", not responding");
7124                        break;
7125                default:
7126                break;
7127                }
7128                NDBG20(("mptsas%d ENCLOSURE STATUS CHANGE for enclosure %x%s\n",
7129                    mpt->m_instance, ddi_get16(mpt->m_acc_reply_frame_hdl,
7130                    &encstatus->EnclosureHandle), string));
7131                break;
7132        }

7134        /*
7135         * MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE is handled by
7136         * mptsas_handle_event_sync,in here just send ack message.
7137         */
7138        case MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE:
7139        {
7140                pMpi2EventDataSasDeviceStatusChange_t   statuschange;
7141                uint8_t                                 rc;
7142                uint16_t                                devhdl;
7143                uint64_t                                wwn = 0;
7144                uint32_t                                wwn_lo, wwn_hi;

7146                statuschange = (pMpi2EventDataSasDeviceStatusChange_t)
7147                    eventreply->EventData;
7148                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7149                    &statuschange->ReasonCode);
7150                wwn_lo = ddi_get32(mpt->m_acc_reply_frame_hdl,
7151                    (uint32_t *)(void *)&statuschange->SASAddress);
7152                wwn_hi = ddi_get32(mpt->m_acc_reply_frame_hdl,
7153                    (uint32_t *)(void *)&statuschange->SASAddress + 1);
7154                wwn = ((uint64_t)wwn_hi << 32) | wwn_lo;
7155                devhdl =  ddi_get16(mpt->m_acc_reply_frame_hdl,
7156                    &statuschange->DevHandle);

7158                NDBG13(("MPI2_EVENT_SAS_DEVICE_STATUS_CHANGE wwn is %"PRIx64,
7159                    wwn));

7161                switch (rc) {
7162                case MPI2_EVENT_SAS_DEV_STAT_RC_SMART_DATA:
7163                        NDBG20(("SMART data received, ASC/ASCQ = %02x/%02x",
7164                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7165                            &statuschange->ASC),
7166                            ddi_get8(mpt->m_acc_reply_frame_hdl,
7167                            &statuschange->ASCQ)));
7168                        break;

7170                case MPI2_EVENT_SAS_DEV_STAT_RC_UNSUPPORTED:
7171                        NDBG20(("Device not supported"));
7172                        break;

7174                case MPI2_EVENT_SAS_DEV_STAT_RC_INTERNAL_DEVICE_RESET:
7175                        NDBG20(("IOC internally generated the Target Reset "
7176                            "for devhdl:%x", devhdl));
7177                        break;

7179                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_INTERNAL_DEV_RESET:
7180                        NDBG20(("IOC's internally generated Target Reset "
7181                            "completed for devhdl:%x", devhdl));
7182                        break;

7184                case MPI2_EVENT_SAS_DEV_STAT_RC_TASK_ABORT_INTERNAL:
7185                        NDBG20(("IOC internally generated Abort Task"));
7186                        break;
```

```
7188                case MPI2_EVENT_SAS_DEV_STAT_RC_CMP_TASK_ABORT_INTERNAL:
7189                        NDBG20(("IOC's internally generated Abort Task "
7190                            "completed"));
7191                        break;

7193                case MPI2_EVENT_SAS_DEV_STAT_RC_ABORT_TASK_SET_INTERNAL:
7194                        NDBG20(("IOC internally generated Abort Task Set"));
7195                        break;

7197                case MPI2_EVENT_SAS_DEV_STAT_RC_CLEAR_TASK_SET_INTERNAL:
7198                        NDBG20(("IOC internally generated Clear Task Set"));
7199                        break;

7201                case MPI2_EVENT_SAS_DEV_STAT_RC_QUERY_TASK_INTERNAL:
7202                        NDBG20(("IOC internally generated Query Task"));
7203                        break;

7205                case MPI2_EVENT_SAS_DEV_STAT_RC_ASYNC_NOTIFICATION:
7206                        NDBG20(("Device sent an Asynchronous Notification"));
7207                        break;

7209                default:
7210                        break;
7211                }
7212                break;
7213        }
7214        case MPI2_EVENT_IR_CONFIGURATION_CHANGE_LIST:
7215        {
7216                /*
7217                 * IR TOPOLOGY CHANGE LIST Event has already been handled
7218                 * in mpt_handle_event_sync() of interrupt context
7219                 */
7220                break;
7221        }
7222        case MPI2_EVENT_IR_OPERATION_STATUS:
7223        {
7224                Mpi2EventDataIrOperationStatus_t        *irOpStatus;
7225                char                                    reason_str[80];
7226                uint8_t                                 rc, percent;
7227                uint16_t                                handle;

7229                irOpStatus = (pMpi2EventDataIrOperationStatus_t)
7230                    eventreply->EventData;
7231                rc = ddi_get8(mpt->m_acc_reply_frame_hdl,
7232                    &irOpStatus->RAIDOperation);
7233                percent = ddi_get8(mpt->m_acc_reply_frame_hdl,
7234                    &irOpStatus->PercentComplete);
7235                handle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7236                    &irOpStatus->VolDevHandle);

7238                switch (rc) {
7239                        case MPI2_EVENT_IR_RAIDOP_RESYNC:
7240                                (void) sprintf(reason_str, "resync");
7241                                break;
7242                        case MPI2_EVENT_IR_RAIDOP_ONLINE_CAP_EXPANSION:
7243                                (void) sprintf(reason_str, "online capacity "
7244                                    "expansion");
7245                                break;
7246                        case MPI2_EVENT_IR_RAIDOP_CONSISTENCY_CHECK:
7247                                (void) sprintf(reason_str, "consistency check");
7248                                break;
7249                        default:
7250                                (void) sprintf(reason_str, "unknown reason %x",
7251                                    rc);
7252                }
```



```
7254                NDBG20(("mptsas%d raid operational status: (%s)"
7255                    "\thandle(0x%04x), percent complete(%d)\n",
7256                    mpt->m_instance, reason_str, handle, percent));
7257                break;
7258        }
7259        case MPI2_EVENT_SAS_BROADCAST_PRIMITIVE:
7260        {
7261                pMpi2EventDataSasBroadcastPrimitive_t   sas_broadcast;
7262                uint8_t                                 phy_num;
7263                uint8_t                                 primitive;

7265                sas_broadcast = (pMpi2EventDataSasBroadcastPrimitive_t)
7266                    eventreply->EventData;

7268                phy_num = ddi_get8(mpt->m_acc_reply_frame_hdl,
7269                    &sas_broadcast->PhyNum);
7270                primitive = ddi_get8(mpt->m_acc_reply_frame_hdl,
7271                    &sas_broadcast->Primitive);

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

```
7319                            SAS_PORT_BROADCAST_D27_4,
7320                            &mpt->m_phy_info[phy_num].smhba_info);
7321                    break;
7322            default:
7323                    NDBG20(("mptsas%d: unknown BROADCAST PRIMITIVE"
7324                        " %x received",
7325                        mpt->m_instance, primitive));
7326                    break;
7327            }
7328            NDBG20(("mptsas%d sas broadcast primitive: "
7329                "\tprimitive(0x%04x), phy(%d) complete\n",
7330                mpt->m_instance, primitive, phy_num));
7331            break;
7332        }
7333        case MPI2_EVENT_IR_VOLUME:
7334        {
7335            Mpi2EventDataIrVolume_t         *irVolume;
7336            uint16_t                        devhandle;
7337            uint32_t                        state;
7338            int                             config, vol;
7339            mptsas_slots_t                  *slots = mpt->m_active;
7340            uint8_t                         found = FALSE;

7342            irVolume = (pMpi2EventDataIrVolume_t)eventreply->EventData;
7343            state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7344                &irVolume->NewValue);
7345            devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7346                &irVolume->VolDevHandle);

7348            NDBG20(("EVENT_IR_VOLUME event is received"));

7350            /*
7351             * Get latest RAID info and then find the DevHandle for this
7352             * event in the configuration.  If the DevHandle is not found
7353             * just exit the event.
7354             */
7355            (void) mptsas_get_raid_info(mpt);
7356            for (config = 0; (config < slots->m_num_raid_configs) &&
7357                (!found); config++) {
7358                for (vol = 0; vol < MPTSAS_MAX_RAIDVOLS; vol++) {
7359                    if (slots->m_raidconfig[config].m_raidvol[vol].
7360                        m_raidhandle == devhandle) {
7361                        found = TRUE;
7362                        break;
7363                    }
7364                }
7365            }
7366            if (!found) {
7367                break;
7368            }

7370            switch (irVolume->ReasonCode) {
7371            case MPI2_EVENT_IR_VOLUME_RC_SETTINGS_CHANGED:
7372            {
7373                uint32_t i;
7374                slots->m_raidconfig[config].m_raidvol[vol].m_settings =
7375                    state;

7377                i = state & MPI2_RAIDVOL0_SETTING_MASK_WRITE_CACHING;
7378                mptsas_log(mpt, CE_NOTE, " Volume %d settings changed"
7379                    ", auto-config of hot-swap drives is %s"
7380                    ", write caching is %s"
7381                    ", hot-spare pool mask is %02x\n",
7382                    vol, state &
7383                    MPI2_RAIDVOL0_SETTING_AUTO_CONFIG_HSWAP_DISABLE
7384                    ? "disabled" : "enabled",
```



```
7385                            i == MPI2_RAIDVOL0_SETTING_UNCHANGED
7386                            ? "controlled by member disks" :
7387                            i == MPI2_RAIDVOL0_SETTING_DISABLE_WRITE_CACHING
7388                            ? "disabled" :
7389                            i == MPI2_RAIDVOL0_SETTING_ENABLE_WRITE_CACHING
7390                            ? "enabled" :
7391                            "incorrectly set",
7392                            (state >> 16) & 0xff);
7393                                break;
7394                }
7395                case MPI2_EVENT_IR_VOLUME_RC_STATE_CHANGED:
7396                {
7397                        slots->m_raidconfig[config].m_raidvol[vol].m_state =
7398                            (uint8_t)state;

7400                        mptsas_log(mpt, CE_NOTE,
7401                            "Volume %d is now %s\n", vol,
7402                            state == MPI2_RAID_VOL_STATE_OPTIMAL
7403                            ? "optimal" :
7404                            state == MPI2_RAID_VOL_STATE_DEGRADED
7405                            ? "degraded" :
7406                            state == MPI2_RAID_VOL_STATE_ONLINE
7407                            ? "online" :
7408                            state == MPI2_RAID_VOL_STATE_INITIALIZING
7409                            ? "initializing" :
7410                            state == MPI2_RAID_VOL_STATE_FAILED
7411                            ? "failed" :
7412                            state == MPI2_RAID_VOL_STATE_MISSING
7413                            ? "missing" :
7414                            "state unknown");
7415                        break;
7416                }
7417                case MPI2_EVENT_IR_VOLUME_RC_STATUS_FLAGS_CHANGED:
7418                {
7419                        slots->m_raidconfig[config].m_raidvol[vol].
7420                            m_statusflags = state;

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

```
7451                }
7452                break;
7453        }
7454        case MPI2_EVENT_IR_PHYSICAL_DISK:
7455        {
7456                Mpi2EventDataIrPhysicalDisk_t   *irPhysDisk;
7457                uint16_t                        devhandle, enchandle, slot;
7458                uint32_t                        status, state;
7459                uint8_t                         physdisknum, reason;

7461                irPhysDisk = (Mpi2EventDataIrPhysicalDisk_t *)
7462                    eventreply->EventData;
7463                physdisknum = ddi_get8(mpt->m_acc_reply_frame_hdl,
7464                    &irPhysDisk->PhysDiskNum);
7465                devhandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7466                    &irPhysDisk->PhysDiskDevHandle);
7467                enchandle = ddi_get16(mpt->m_acc_reply_frame_hdl,
7468                    &irPhysDisk->EnclosureHandle);
7469                slot = ddi_get16(mpt->m_acc_reply_frame_hdl,
7470                    &irPhysDisk->Slot);
7471                state = ddi_get32(mpt->m_acc_reply_frame_hdl,
7472                    &irPhysDisk->NewValue);
7473                reason = ddi_get8(mpt->m_acc_reply_frame_hdl,
7474                    &irPhysDisk->ReasonCode);

7476                NDBG20(("EVENT_IR_PHYSICAL_DISK event is received"));

7478                switch (reason) {
7479                case MPI2_EVENT_IR_PHYSDISK_RC_SETTINGS_CHANGED:
7480                        mptsas_log(mpt, CE_NOTE,
7481                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7482                            "for enclosure with handle 0x%x is now in hot "
7483                            "spare pool %d",
7484                            physdisknum, devhandle, slot, enchandle,
7485                            (state >> 16) & 0xff);
7486                        break;

7488                case MPI2_EVENT_IR_PHYSDISK_RC_STATUS_FLAGS_CHANGED:
7489                        status = state;
7490                        mptsas_log(mpt, CE_NOTE,
7491                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7492                            "for enclosure with handle 0x%x is now "
7493                            "%s%s%s%s%s\n", physdisknum, devhandle, slot,
7494                            enchandle,
7495                            status & MPI2_PHYSDISK0_STATUS_FLAG_INACTIVE_VOLUME
7496                            ? ", inactive" : ", active",
7497                            status & MPI2_PHYSDISK0_STATUS_FLAG_OUT_OF_SYNC
7498                            ? ", out of sync" : "",
7499                            status & MPI2_PHYSDISK0_STATUS_FLAG_QUIESCED
7500                            ? ", quiesced" : "",
7501                            status &
7502                            MPI2_PHYSDISK0_STATUS_FLAG_WRITE_CACHE_ENABLED
7503                            ? ", write cache enabled" : "",
7504                            status & MPI2_PHYSDISK0_STATUS_FLAG_OCE_TARGET
7505                            ? ", capacity expansion target" : "");
7506                        break;

7508                case MPI2_EVENT_IR_PHYSDISK_RC_STATE_CHANGED:
7509                        mptsas_log(mpt, CE_NOTE,
7510                            " PhysDiskNum %d with DevHandle 0x%x in slot %d "
7511                            "for enclosure with handle 0x%x is now %s\n",
7512                            physdisknum, devhandle, slot, enchandle,
7513                            state == MPI2_RAID_PD_STATE_OPTIMAL
7514                            ? "optimal" :
7515                            state == MPI2_RAID_PD_STATE_REBUILDING
7516                            ? "rebuilding" :
```



```
7517                            state == MPI2_RAID_PD_STATE_DEGRADED
7518                            ? "degraded" :
7519                            state == MPI2_RAID_PD_STATE_HOT_SPARE
7520                            ? "a hot spare" :
7521                            state == MPI2_RAID_PD_STATE_ONLINE
7522                            ? "online" :
7523                            state == MPI2_RAID_PD_STATE_OFFLINE
7524                            ? "offline" :
7525                            state == MPI2_RAID_PD_STATE_NOT_COMPATIBLE
7526                            ? "not compatible" :
7527                            state == MPI2_RAID_PD_STATE_NOT_CONFIGURED
7528                            ? "not configured" :
7529                            "state unknown");
7530                        break;
7531                }
7532                break;
7533        }
7534        default:
7535                NDBG20(("mptsas%d: unknown event %x received",
7536                    mpt->m_instance, event));
7537                break;
7538        }

7540        /*
7541         * Return the reply frame to the free queue.
7542         */
7543        ddi_put32(mpt->m_acc_free_queue_hdl,
7544            &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index], rfm);
7545        (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
7546            DDI_DMA_SYNC_FORDEV);
7547        if (++mpt->m_free_index == mpt->m_free_queue_depth) {
7548                mpt->m_free_index = 0;
7549        }
7550        ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
7551            mpt->m_free_index);
7552        mutex_exit(&mpt->m_mutex);
7553 }
```

*unchanged_portion_omitted_*

```
8507 /*
8508  * Clean up from a device reset.
8509  * For the case of target reset, this function clears the waitq of all
8510  * commands for a particular target.   For the case of abort task set, this
8511  * function clears the waitq of all commonds for a particular target/lun.
8512  */
8513 static void
8514 mptsas_flush_target(mptsas_t *mpt, ushort_t target, int lun, uint8_t tasktype)
8515 {
8516        mptsas_slots_t  *slots = mpt->m_active;
8517        mptsas_cmd_t    *cmd, *next_cmd;
8518        int             slot;
8519        uchar_t         reason;
8520        uint_t          stat;

8522        NDBG25(("mptsas_flush_target: target=%d lun=%d", target, lun));

8524        /*
8525         * Make sure the I/O Controller has flushed all cmds
8526         * that are associated with this target for a target reset
8527         * and target/lun for abort task set.
8528         * Account for TM requests, which use the last SMID.
8529         */
8530        for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8531                if ((cmd = slots->m_slot[slot]) == NULL)
8532                        continue;
8533                reason = CMD_RESET;
```

```
8534                 stat = STAT_DEV_RESET;
8535                 switch (tasktype) {
8536                 case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8537                         if (Tgt(cmd) == target) {
8538                                 if (cmd->cmd_tgt_addr->m_timeout < 0) {
8539                                         /*
8540                                          * When timeout requested, propagate
8541                                          * proper reason and statistics to
8542                                          * target drivers.
8543                                          */
8544                                         reason = CMD_TIMEOUT;
8545                                         stat |= STAT_TIMEOUT;
8546                                 }
8547 
8548                                 NDBG25(("mptsas_flush_target discovered non-"
8549                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8550                                     tasktype));
8551                                 mptsas_dump_cmd(mpt, cmd);
8552                                 mptsas_remove_cmd(mpt, cmd);
8553                                 mptsas_set_pkt_reason(mpt, cmd, reason, stat);
8554                                 mptsas_doneq_add(mpt, cmd);
8555                         }
8556                         break;
8557                 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8558                         reason = CMD_ABORTED;
8559                         stat = STAT_ABORTED;
8560                         /*FALLTHROUGH*/
8561                 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8562                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8563 
8564                                 NDBG25(("mptsas_flush_target discovered non-"
8565                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8566                                     tasktype));
8567                                 mptsas_dump_cmd(mpt, cmd);
8568                                 mptsas_remove_cmd(mpt, cmd);
8569                                 mptsas_set_pkt_reason(mpt, cmd, reason,
8570                                     stat);
8571                                 mptsas_doneq_add(mpt, cmd);
8572                         }
8573                         break;
8574                 default:
8575                         break;
8576                 }
8577         }
8578 
8579         /*
8580          * Flush the waitq and tx_waitq of this target's cmds
8581          */
8582         cmd = mpt->m_waitq;
8583 
8584         reason = CMD_RESET;
8585         stat = STAT_DEV_RESET;
8586 
8587         switch (tasktype) {
8588         case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8589                 while (cmd != NULL) {
8590                         next_cmd = cmd->cmd_linkp;
8591                         if (Tgt(cmd) == target) {
8592                                 mptsas_waitq_delete(mpt, cmd);
8593                                 mptsas_set_pkt_reason(mpt, cmd,
8594                                     reason, stat);
8595                                 mptsas_doneq_add(mpt, cmd);
8596                         }
8597                         cmd = next_cmd;
8598                 }
8599                 mutex_enter(&mpt->m_tx_waitq_mutex);
```



```
8600                 cmd = mpt->m_tx_waitq;
8601                 while (cmd != NULL) {
8602                         next_cmd = cmd->cmd_linkp;
8603                         if (Tgt(cmd) == target) {
8604                                 mptsas_tx_waitq_delete(mpt, cmd);
8605                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8606                                 mptsas_set_pkt_reason(mpt, cmd,
8607                                     reason, stat);
8608                                 mptsas_doneq_add(mpt, cmd);
8609                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8610                         }
8611                         cmd = next_cmd;
8612                 }
8613                 mutex_exit(&mpt->m_tx_waitq_mutex);
8614                 break;
8615         case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8616                 reason = CMD_ABORTED;
8617                 stat =  STAT_ABORTED;
8618                 /*FALLTHROUGH*/
8619         case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8620                 while (cmd != NULL) {
8621                         next_cmd = cmd->cmd_linkp;
8622                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8623                                 mptsas_waitq_delete(mpt, cmd);
8624                                 mptsas_set_pkt_reason(mpt, cmd,
8625                                     reason, stat);
8626                                 mptsas_doneq_add(mpt, cmd);
8627                         }
8628                         cmd = next_cmd;
8629                 }
8630                 mutex_enter(&mpt->m_tx_waitq_mutex);
8631                 cmd = mpt->m_tx_waitq;
8632                 while (cmd != NULL) {
8633                         next_cmd = cmd->cmd_linkp;
8634                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8635                                 mptsas_tx_waitq_delete(mpt, cmd);
8636                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8637                                 mptsas_set_pkt_reason(mpt, cmd,
8638                                     reason, stat);
8639                                 mptsas_doneq_add(mpt, cmd);
8640                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8641                         }
8642                         cmd = next_cmd;
8643                 }
8644                 mutex_exit(&mpt->m_tx_waitq_mutex);
8645                 break;
8646         default:
8647                 mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
8648                     tasktype);
8649                 break;
8650         }
8651 }
8652 
8653 /*
8654  * Clean up hba state, abort all outstanding command and commands in waitq
8655  * reset timeout of all targets.
8656  */
8657 static void
8658 mptsas_flush_hba(mptsas_t *mpt)
8659 {
8660         mptsas_slots_t  *slots = mpt->m_active;
8661         mptsas_cmd_t    *cmd;
8662         int             slot;
8663 
8664         NDBG25(("mptsas_flush_hba"));
```

```
8666         /*
8667          * The I/O Controller should have already sent back
8668          * all commands via the scsi I/O reply frame.  Make
8669          * sure all commands have been flushed.
8670          * Account for TM request, which use the last SMID.
8671          */
8672         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8673                 if ((cmd = slots->m_slot[slot]) == NULL)
8674                         continue;

8676                 if (cmd->cmd_flags & CFLAG_CMDIOC) {
8677                         /*
8678                          * Need to make sure to tell everyone that might be
8679                          * waiting on this command that it's going to fail.  If
8680                          * we get here, this command will never timeout because
8681                          * the active command table is going to be re-allocated,
8682                          * so there will be nothing to check against a time out.
8683                          * Instead, mark the command as failed due to reset.
8684                          */
8685                         mptsas_set_pkt_reason(mpt, cmd, CMD_RESET,
8686                             STAT_BUS_RESET);
8687                         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8688                             (cmd->cmd_flags & CFLAG_CONFIG) ||
8689                             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8690                                 cmd->cmd_flags |= CFLAG_FINISHED;
8691                                 cv_broadcast(&mpt->m_passthru_cv);
8692                                 cv_broadcast(&mpt->m_config_cv);
8693                                 cv_broadcast(&mpt->m_fw_diag_cv);
8694                         }
8695                         continue;
8696                 }

8698                 NDBG25(("mptsas_flush_hba discovered non-NULL cmd in slot %d",
8699                     slot));
8700                 mptsas_dump_cmd(mpt, cmd);

8702                 mptsas_remove_cmd(mpt, cmd);
8703                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8704                 mptsas_doneq_add(mpt, cmd);
8705         }

8707         /*
8708          * Flush the waitq.
8709          */
8710         while ((cmd = mptsas_waitq_rm(mpt)) != NULL) {
8711                 mptsas_set_pkt_reason(mpt, cmd, CMD_RESET, STAT_BUS_RESET);
8712                 if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
8713                     (cmd->cmd_flags & CFLAG_CONFIG) ||
8714                     (cmd->cmd_flags & CFLAG_FW_DIAG)) {
8715                         cmd->cmd_flags |= CFLAG_FINISHED;
8716                         cv_broadcast(&mpt->m_passthru_cv);
8717                         cv_broadcast(&mpt->m_config_cv);
8718                         cv_broadcast(&mpt->m_fw_diag_cv);
8719                 } else {
8720                         mptsas_doneq_add(mpt, cmd);
8721                 }
8722         }

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



```
8732                 mutex_enter(&mpt->m_tx_waitq_mutex);
8733         }
8734         mutex_exit(&mpt->m_tx_waitq_mutex);

8736         /*
8737          * Drain the taskqs prior to reallocating resources.
8738          */
8739         mutex_exit(&mpt->m_mutex);
8740         ddi_taskq_wait(mpt->m_event_taskq);
8741         ddi_taskq_wait(mpt->m_dr_taskq);
8742         mutex_enter(&mpt->m_mutex);
8743 }
```

*_____unchanged_portion_omitted_*

```
9342 static void
9343 mptsas_watchsubr(mptsas_t *mpt)
9344 {
9345         int             i;
9346         mptsas_cmd_t    *cmd;
9347         mptsas_target_t *ptgt = NULL;

9349         NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

9351 #ifdef MPTSAS_TEST
9352         if (mptsas_enable_untagged) {
9353                 mptsas_test_untagged++;
9354         }
9355 #endif

9357         /*
9358          * Check for commands stuck in active slot
9359          * Account for TM requests, which use the last SMID.
9360          */
9361         for (i = 0; i <= mpt->m_active->m_n_slots; i++) {
9362                 if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9363                         if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9364                                 cmd->cmd_active_timeout -=
9365                                     mptsas_scsi_watchdog_tick;
9366                                 if (cmd->cmd_active_timeout <= 0) {
9367                                         /*
9368                                          * There seems to be a command stuck
9369                                          * in the active slot.  Drain throttle.
9370                                          */
9371                                         mptsas_set_throttle(mpt,
9372                                             cmd->cmd_tgt_addr,
9373                                             DRAIN_THROTTLE);
9374                                 }
9375                         }
9376                         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
9377                             (cmd->cmd_flags & CFLAG_CONFIG) ||
9378                             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
9379                                 cmd->cmd_active_timeout -=
9380                                     mptsas_scsi_watchdog_tick;
9381                                 if (cmd->cmd_active_timeout <= 0) {
9382                                         /*
9383                                          * passthrough command timeout
9384                                          */
9385                                         cmd->cmd_flags |= (CFLAG_FINISHED |
9386                                             CFLAG_TIMEOUT);
9387                                         cv_broadcast(&mpt->m_passthru_cv);
9388                                         cv_broadcast(&mpt->m_config_cv);
9389                                         cv_broadcast(&mpt->m_fw_diag_cv);
9390                                 }
9391                         }
9392                 }
9393         }
```

```
9395         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9396             MPTSAS_HASH_FIRST);
9397         while (ptgt != NULL) {
9398                 /*
9399                  * If we were draining due to a qfull condition,
9400                  * go back to full throttle.
9401                  */
9402                 if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9403                     (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9404                     (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9405                         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9406                         mptsas_restart_hba(mpt);
9407                 }
9408
9409                 if ((ptgt->m_t_ncmds > 0) &&
9410                     (ptgt->m_timebase)) {
9411
9412                         if (ptgt->m_timebase <=
9413                             mptsas_scsi_watchdog_tick) {
9414                                 ptgt->m_timebase +=
9415                                     mptsas_scsi_watchdog_tick;
9416                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9417                                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9418                                 continue;
9419                         }
9420
9421                         ptgt->m_timeout -= mptsas_scsi_watchdog_tick;
9422
9423                         if (ptgt->m_timeout_count > 0) {
9424                                 ptgt->m_timeout_interval +=
9425                                     mptsas_scsi_watchdog_tick;
9426                         }
9427                         if (ptgt->m_timeout_interval > mptsas_timeout_interval)
9428                                 ptgt->m_timeout_interval = 0;
9429                                 ptgt->m_timeout_count = 0;
9430                         }
9431
9432                         if (ptgt->m_timeout < 0) {
9433                                 ptgt->m_timeout_count++;
9434                                 if (ptgt->m_timeout_count >
9435                                     mptsas_timeout_threshold) {
9436                                         ptgt->m_timeout_count = 0;
9437                                         mptsas_kill_target(mpt, ptgt);
9438                                 } else {
9439                                         mptsas_cmd_timeout(mpt, ptgt->m_devhdl);
9440                                 }
9441                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9442                                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9443                                 continue;
9444                         }
9445
9446                         if ((ptgt->m_timeout) <=
9447                             mptsas_scsi_watchdog_tick) {
9448                                 NDBG23(("pending timeout"));
9449                                 mptsas_set_throttle(mpt, ptgt,
9450                                     DRAIN_THROTTLE);
9451                         }
9452                 }
9453
9454                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9455                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9456         }
9457 }
```

*unchanged_portion_omitted_*


```
9481 /*
9482  * target causing too many timeouts
9483  */
9484 static void
9485 mptsas_kill_target(mptsas_t *mpt, mptsas_target_t *ptgt)
9486 {
9487         mptsas_topo_change_list_t       *topo_node = NULL;
9488
9489         NDBG29(("mptsas_tgt_kill: target=%d", ptgt->m_devhdl));
9490         mptsas_log(mpt, CE_WARN, "timeout threshold exceeded for "
9491             "Target %d", ptgt->m_devhdl);
9492
9493         topo_node = kmem_zalloc(sizeof (mptsas_topo_change_list_t), KM_SLEEP);
9494         topo_node->mpt = mpt;
9495         topo_node->un.phymask = ptgt->m_phymask;
9496         topo_node->event = MPTSAS_DR_EVENT_OFFLINE_TARGET;
9497         topo_node->devhdl = ptgt->m_devhdl;
9498         if (ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
9499                 topo_node->flags = MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE;
9500         else
9501                 topo_node->flags = MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE;
9502         topo_node->object = NULL;
9503
9504         /*
9505          * Launch DR taskq to fake topology change
9506          */
9507         if ((ddi_taskq_dispatch(mpt->m_dr_taskq,
9508             mptsas_handle_dr, (void *)topo_node,
9509             DDI_NOSLEEP)) != DDI_SUCCESS) {
9510                 mptsas_log(mpt, CE_NOTE, "mptsas start taskq "
9511                     "for fake offline event failed. \n");
9512         }
9513 }
9514
9515 /*
9516  * Device / Hotplug control
9517  */
9518 static int
9519 mptsas_scsi_quiesce(dev_info_t *dip)
9520 {
9521         mptsas_t        *mpt;
9522         scsi_hba_tran_t *tran;
9523
9524         tran = ddi_get_driver_private(dip);
9525         if (tran == NULL || (mpt = TRAN2MPT(tran)) == NULL)
9526                 return (-1);
9527
9528         return (mptsas_quiesce_bus(mpt));
9529 }
```

*unchanged_portion_omitted_*

```
11355 static int
11356 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
11357     int *rval)
11358 {
11359         int                     status = 0;
11360         mptsas_t                *mpt;
11361         mptsas_update_flash_t   flashdata;
11362         mptsas_pass_thru_t      passthru_data;
11363         mptsas_adapter_data_t   adapter_data;
11364         mptsas_pci_info_t       pci_info;
11365         int                     copylen;
11366
11367         int                     iport_flag = 0;
11368         dev_info_t              *dip = NULL;
11369         mptsas_phymask_t        phymask = 0;
```

```
11370         struct devctl_iocdata   *dcp = NULL;
11371         uint32_t                slotstatus = 0;
11372         char                    *addr = NULL;
11373         mptsas_target_t         *ptgt = NULL;

11375         *rval = MPTIOCTL_STATUS_GOOD;
11376         if (secpolicy_sys_config(credp, B_FALSE) != 0) {
11377                 return (EPERM);
11378         }

11380         mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
11381         if (mpt == NULL) {
11382                 /*
11383                  * Called from iport node, get the states
11384                  */
11385                 iport_flag = 1;
11386                 dip = mptsas_get_dip_from_dev(dev, &phymask);
11387                 if (dip == NULL) {
11388                         return (ENXIO);
11389                 }
11390                 mpt = DIP2MPT(dip);
11391         }
11392         /* Make sure power level is D0 before accessing registers */
11393         mutex_enter(&mpt->m_mutex);
11394         if (mpt->m_options & MPTSAS_OPT_PM) {
11395                 (void) pm_busy_component(mpt->m_dip, 0);
11396                 if (mpt->m_power_level != PM_LEVEL_D0) {
11397                         mutex_exit(&mpt->m_mutex);
11398                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
11399                             DDI_SUCCESS) {
11400                                 mptsas_log(mpt, CE_WARN,
11401                                     "mptsas%d: mptsas_ioctl: Raise power "
11402                                     "request failed.", mpt->m_instance);
11403                                 (void) pm_idle_component(mpt->m_dip, 0);
11404                                 return (ENXIO);
11405                         }
11406                 } else {
11407                         mutex_exit(&mpt->m_mutex);
11408                 }
11409         } else {
11410                 mutex_exit(&mpt->m_mutex);
11411         }

11413         if (iport_flag) {
11414                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
11329                 if (status != 0) {
11415                         goto out;
11416         }
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         }
11417         switch (cmd) {
11418                 case MPTIOCTL_UPDATE_FLASH:
11419                         if (ddi_copyin((void *)data, &flashdata,
11420                                 sizeof (struct mptsas_update_flash), mode)) {
11421                                 status = EFAULT;
11422                                 break;
11423                         }

11425                         mutex_enter(&mpt->m_mutex);
11426                         if (mptsas_update_flash(mpt,
11427                             (caddr_t)(long)flashdata.PtrBuffer,
11428                             flashdata.ImageSize, flashdata.ImageType, mode)) {
11429                                 status = EFAULT;
11430                         }

11432                         /*
11433                          * Reset the chip to start using the new
11434                          * firmware.  Reset if failed also.
11435                          */
11436                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11437                         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
11438                                 status = EFAULT;
11439                         }
11440                         mutex_exit(&mpt->m_mutex);
11441                         break;
11442                 case MPTIOCTL_PASS_THRU:
11443                         /*
```

```
11444                          * The user has requested to pass through a command to
11445                          * be executed by the MPT firmware.  Call our routine
11446                          * which does this.  Only allow one passthru IOCTL at
11447                          * one time. Other threads will block on
11448                          * m_passthru_mutex, which is of adaptive variant.
11449                          */
11450                         if (ddi_copyin((void *)data, &passthru_data,
11451                             sizeof (mptsas_pass_thru_t), mode)) {
11452                                 status = EFAULT;
11453                                 break;
11454                         }
11455                         mutex_enter(&mpt->m_passthru_mutex);
11456                         mutex_enter(&mpt->m_mutex);
11457                         status = mptsas_pass_thru(mpt, &passthru_data, mode);
11458                         mutex_exit(&mpt->m_mutex);
11459                         mutex_exit(&mpt->m_passthru_mutex);

11461                         break;
11462                 case MPTIOCTL_GET_ADAPTER_DATA:
11463                         /*
11464                          * The user has requested to read adapter data.  Call
11465                          * our routine which does this.
11466                          */
11467                         bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
11468                         if (ddi_copyin((void *)data, (void *)&adapter_data,
11469                             sizeof (mptsas_adapter_data_t), mode)) {
11470                                 status = EFAULT;
11471                                 break;
11472                         }
11473                         if (adapter_data.StructureLength >=
11474                             sizeof (mptsas_adapter_data_t)) {
11475                                 adapter_data.StructureLength = (uint32_t)
11476                                     sizeof (mptsas_adapter_data_t);
11477                                 copylen = sizeof (mptsas_adapter_data_t);
11478                                 mutex_enter(&mpt->m_mutex);
11479                                 mptsas_read_adapter_data(mpt, &adapter_data);
11480                                 mutex_exit(&mpt->m_mutex);
11481                         } else {
11482                                 adapter_data.StructureLength = (uint32_t)
11483                                     sizeof (mptsas_adapter_data_t);
11484                                 copylen = sizeof (adapter_data.StructureLength);
11485                                 *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
11486                         }
11487                         if (ddi_copyout((void *)(&adapter_data), (void *)data,
11488                             copylen, mode) != 0) {
11489                                 status = EFAULT;
11490                         }
11491                         break;
11492                 case MPTIOCTL_GET_PCI_INFO:
11493                         /*
11494                          * The user has requested to read pci info.  Call
11495                          * our routine which does this.
11496                          */
11497                         bzero(&pci_info, sizeof (mptsas_pci_info_t));
11498                         mutex_enter(&mpt->m_mutex);
11499                         mptsas_read_pci_info(mpt, &pci_info);
11500                         mutex_exit(&mpt->m_mutex);
11501                         if (ddi_copyout((void *)(&pci_info), (void *)data,
11502                             sizeof (mptsas_pci_info_t), mode) != 0) {
11503                                 status = EFAULT;
11504                         }
11505                         break;
11506                 case MPTIOCTL_RESET_ADAPTER:
11507                         mutex_enter(&mpt->m_mutex);
11508                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11509                         if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
```



```
11510                                 mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
11511                                     "failed");
11512                                 status = EFAULT;
11513                         }
11514                         mutex_exit(&mpt->m_mutex);
11515                         break;
11516                 case MPTIOCTL_DIAG_ACTION:
11517                         /*
11518                          * The user has done a diag buffer action.  Call our
11519                          * routine which does this.  Only allow one diag action
11520                          * at one time.
11521                          */
11522                         mutex_enter(&mpt->m_mutex);
11523                         if (mpt->m_diag_action_in_progress) {
11524                                 mutex_exit(&mpt->m_mutex);
11525                                 status = EBUSY;
11526                                 goto out;
11527                         }
11528                         mpt->m_diag_action_in_progress = 1;
11529                         status = mptsas_diag_action(mpt,
11530                             (mptsas_diag_action_t *)data, mode);
11531                         mpt->m_diag_action_in_progress = 0;
11532                         mutex_exit(&mpt->m_mutex);
11533                         break;
11534                 case MPTIOCTL_EVENT_QUERY:
11535                         /*
11536                          * The user has done an event query. Call our routine
11537                          * which does this.
11538                          */
11539                         status = mptsas_event_query(mpt,
11540                             (mptsas_event_query_t *)data, mode, rval);
11541                         break;
11542                 case MPTIOCTL_EVENT_ENABLE:
11543                         /*
11544                          * The user has done an event enable. Call our routine
11545                          * which does this.
11546                          */
11547                         status = mptsas_event_enable(mpt,
11548                             (mptsas_event_enable_t *)data, mode, rval);
11549                         break;
11550                 case MPTIOCTL_EVENT_REPORT:
11551                         /*
11552                          * The user has done an event report. Call our routine
11553                          * which does this.
11554                          */
11555                         status = mptsas_event_report(mpt,
11556                             (mptsas_event_report_t *)data, mode, rval);
11557                         break;
11558                 case MPTIOCTL_REG_ACCESS:
11559                         /*
11560                          * The user has requested register access.  Call our
11561                          * routine which does this.
11562                          */
11563                         status = mptsas_reg_access(mpt,
11564                             (mptsas_reg_access_t *)data, mode);
11565                         break;
11566                 default:
11567                         status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
11568                             rval);
11569                         break;
11570         }

11572 out:
11573         return (status);
11574 }
```

_____***unchanged_portion_omitted_***

```
13809 static int
13810 mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *inq, char *guid,
13811     dev_info_t **lun_dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt, int lun)
13812 {
13813         int                     target;
13814         char                    *nodename = NULL;
13815         char                    **compatible = NULL;
13816         int                     ncompatible     = 0;
13817         int                     mdi_rtn = MDI_FAILURE;
13818         int                     rval = DDI_FAILURE;
13819         char                    *old_guid = NULL;
13820         mptsas_t                *mpt = DIP2MPT(pdip);
13821         char                    *lun_addr = NULL;
13822         char                    *wwn_str = NULL;
13823         char                    *attached_wwn_str = NULL;
13824         char                    *component = NULL;
13825         uint8_t                 phy = 0xFF;
13826         uint64_t                sas_wwn;
13827         int64_t                 lun64 = 0;
13828         uint32_t                devinfo;
13829         uint16_t                dev_hdl;
13830         uint16_t                pdev_hdl;
13831         uint64_t                dev_sas_wwn;
13832         uint64_t                pdev_sas_wwn;
13833         uint32_t                pdev_info;
13834         uint8_t                 physport;
13835         uint8_t                 phy_id;
13836         uint32_t                page_address;
13837         uint16_t                bay_num, enclosure;
13838         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
13839         uint32_t                dev_info;

13841         mutex_enter(&mpt->m_mutex);
13842         target = ptgt->m_devhdl;
13843         sas_wwn = ptgt->m_sas_wwn;
13844         devinfo = ptgt->m_deviceinfo;
13845         phy = ptgt->m_phynum;
13846         mutex_exit(&mpt->m_mutex);

13848         if (sas_wwn) {
13849                 *pip = mptsas_find_path_addr(pdip, sas_wwn, lun);
13850         } else {
13851                 *pip = mptsas_find_path_phy(pdip, phy);
13852         }

13854         if (*pip != NULL) {
13855                 *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
13856                 ASSERT(*lun_dip != NULL);
13857                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, *lun_dip,
13858                     (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM),
13859                     MDI_CLIENT_GUID_PROP, &old_guid) == DDI_SUCCESS) {
13860                         if (strncmp(guid, old_guid, strlen(guid)) == 0) {
13861                                 /*
13862                                  * Same path back online again.
13863                                  */
13864                                 (void) ddi_prop_free(old_guid);
13865                                 if ((!MDI_PI_IS_ONLINE(*pip)) &&
13866                                     (!MDI_PI_IS_STANDBY(*pip)) &&
13867                                     (ptgt->m_tgt_unconfigured == 0)) {
13868                                         rval = mdi_pi_online(*pip, 0);
13841                                         mutex_enter(&mpt->m_mutex);
13842                                         (void) mptsas_set_led_status(mpt, ptgt,
13843                                             0);
13844                                         mutex_exit(&mpt->m_mutex);
13869                                 } else {
```



```
13870                                         rval = DDI_SUCCESS;
13871                                 }
13872                                 if (rval != DDI_SUCCESS) {
13873                                         mptsas_log(mpt, CE_WARN, "path:target: "
13874                                             "%x, lun:%x online failed!", target,
13875                                             lun);
13876                                         *pip = NULL;
13877                                         *lun_dip = NULL;
13878                                 }
13879                                 return (rval);
13880                         } else {
13881                                 /*
13882                                  * The GUID of the LUN has changed which maybe
13883                                  * because customer mapped another volume to the
13884                                  * same LUN.
13885                                  */
13886                                 mptsas_log(mpt, CE_WARN, "The GUID of the "
13887                                     "target:%x, lun:%x was changed, maybe "
13888                                     "because someone mapped another volume "
13889                                     "to the same LUN", target, lun);
13890                                 (void) ddi_prop_free(old_guid);
13891                                 if (!MDI_PI_IS_OFFLINE(*pip)) {
13892                                         rval = mdi_pi_offline(*pip, 0);
13893                                         if (rval != MDI_SUCCESS) {
13894                                                 mptsas_log(mpt, CE_WARN, "path:"
13895                                                     "target:%x, lun:%x offline "
13896                                                     "failed!", target, lun);
13897                                                 *pip = NULL;
13898                                                 *lun_dip = NULL;
13899                                                 return (DDI_FAILURE);
13900                                         }
13901                                 }
13902                                 if (mdi_pi_free(*pip, 0) != MDI_SUCCESS) {
13903                                         mptsas_log(mpt, CE_WARN, "path:target:"
13904                                             "%x, lun:%x free failed!", target,
13905                                             lun);
13906                                         *pip = NULL;
13907                                         *lun_dip = NULL;
13908                                         return (DDI_FAILURE);
13909                                 }
13910                         }
13911                 } else {
13912                         mptsas_log(mpt, CE_WARN, "Can't get client-guid "
13913                             "property for path:target:%x, lun:%x", target, lun);
13914                         *pip = NULL;
13915                         *lun_dip = NULL;
13916                         return (DDI_FAILURE);
13917                 }
13918         }
13919         scsi_hba_nodename_compatible_get(inq, NULL,
13920             inq->inq_dtype, NULL, &nodename, &compatible, &ncompatible);

13922         /*
13923          * if nodename can't be determined then print a message and skip it
13924          */
13925         if (nodename == NULL) {
13926                 mptsas_log(mpt, CE_WARN, "mptsas driver found no compatible "
13927                     "driver for target%d lun %d dtype:0x%02x", target, lun,
13928                     inq->inq_dtype);
13929                 return (DDI_FAILURE);
13930         }

13932         wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
13933         /* The property is needed by MPAPI */
13934         (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
```

```
13936         lun_addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13937         if (guid) {
13938                 (void) sprintf(lun_addr, "w%s,%x", wwn_str, lun);
13939                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
13940         } else {
13941                 (void) sprintf(lun_addr, "p%x,%x", phy, lun);
13942                 (void) sprintf(wwn_str, "p%x", phy);
13943         }

13945         mdi_rtn = mdi_pi_alloc_compatible(pdip, nodename,
13946             guid, lun_addr, compatible, ncompatible,
13947             0, pip);
13948         if (mdi_rtn == MDI_SUCCESS) {

13950                 if (mdi_prop_update_string(*pip, MDI_GUID,
13951                     guid) != DDI_SUCCESS) {
13952                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13953                             "create prop for target %d lun %d (MDI_GUID)",
13954                             target, lun);
13955                         mdi_rtn = MDI_FAILURE;
13956                         goto virt_create_done;
13957                 }

13959                 if (mdi_prop_update_int(*pip, LUN_PROP,
13960                     lun) != DDI_SUCCESS) {
13961                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13962                             "create prop for target %d lun %d (LUN_PROP)",
13963                             target, lun);
13964                         mdi_rtn = MDI_FAILURE;
13965                         goto virt_create_done;
13966                 }
13967                 lun64 = (int64_t)lun;
13968                 if (mdi_prop_update_int64(*pip, LUN64_PROP,
13969                     lun64) != DDI_SUCCESS) {
13970                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13971                             "create prop for target %d (LUN64_PROP)",
13972                             target);
13973                         mdi_rtn = MDI_FAILURE;
13974                         goto virt_create_done;
13975                 }
13976                 if (mdi_prop_update_string_array(*pip, "compatible",
13977                     compatible, ncompatible) !=
13978                     DDI_PROP_SUCCESS) {
13979                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13980                             "create prop for target %d lun %d (COMPATIBLE)",
13981                             target, lun);
13982                         mdi_rtn = MDI_FAILURE;
13983                         goto virt_create_done;
13984                 }
13985                 if (sas_wwn && (mdi_prop_update_string(*pip,
13986                     SCSI_ADDR_PROP_TARGET_PORT, wwn_str) != DDI_PROP_SUCCESS)) {
13987                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13988                             "create prop for target %d lun %d "
13989                             "(target-port)", target, lun);
13990                         mdi_rtn = MDI_FAILURE;
13991                         goto virt_create_done;
13992                 } else if ((sas_wwn == 0) && (mdi_prop_update_int(*pip,
13993                     "sata-phy", phy) != DDI_PROP_SUCCESS)) {
13994                         /*
13995                          * Direct attached SATA device without DeviceName
13996                          */
13997                         mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13998                             "create prop for SAS target %d lun %d "
13999                             "(sata-phy)", target, lun);
14000                         mdi_rtn = MDI_FAILURE;
14001                         goto virt_create_done;
```



```
14002                 }
14003                 mutex_enter(&mpt->m_mutex);

14005                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14006                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14007                     (uint32_t)ptgt->m_devhdl;
14008                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14009                     &dev_hdl, &dev_sas_wwn, &dev_info, &physport,
14010                     &phy_id, &pdev_hdl, &bay_num, &enclosure);
14011                 if (rval != DDI_SUCCESS) {
14012                         mutex_exit(&mpt->m_mutex);
14013                         mptsas_log(mpt, CE_WARN, "mptsas unable to get "
14014                             "parent device for handle %d", page_address);
14015                         mdi_rtn = MDI_FAILURE;
14016                         goto virt_create_done;
14017                 }

14019                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14020                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
14021                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14022                     &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
14023                     &phy_id, &pdev_hdl, &bay_num, &enclosure);
14024                 if (rval != DDI_SUCCESS) {
14025                         mutex_exit(&mpt->m_mutex);
14026                         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
14027                             "device info for handle %d", page_address);
14028                         mdi_rtn = MDI_FAILURE;
14029                         goto virt_create_done;
14030                 }

14032                 mutex_exit(&mpt->m_mutex);

14034                 /*
14035                  * If this device direct attached to the controller
14036                  * set the attached-port to the base wwid
14037                  */
14038                 if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
14039                     != DEVINFO_DIRECT_ATTACHED) {
14040                         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14041                             pdev_sas_wwn);
14042                 } else {
14043                         /*
14044                          * Update the iport's attached-port to guid
14045                          */
14046                         if (sas_wwn == 0) {
14047                                 (void) sprintf(wwn_str, "p%x", phy);
14048                         } else {
14049                                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14050                         }
14051                         if (ddi_prop_update_string(DDI_DEV_T_NONE,
14052                             pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
14053                             DDI_PROP_SUCCESS) {
14054                                 mptsas_log(mpt, CE_WARN,
14055                                     "mptsas unable to create "
14056                                     "property for iport target-port"
14057                                     " %s (sas_wwn)",
14058                                     wwn_str);
14059                                 mdi_rtn = MDI_FAILURE;
14060                                 goto virt_create_done;
14061                         }

14063                         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14064                             mpt->un.m_base_wwid);
14065                 }

14067                 if (mdi_prop_update_string(*pip,
```

```
14068                       SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
14069                       DDI_PROP_SUCCESS) {
14070                           mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14071                               "property for iport attached-port %s (sas_wwn)",
14072                               attached_wwn_str);
14073                           mdi_rtn = MDI_FAILURE;
14074                           goto virt_create_done;
14075                   }


14078                   if (inq->inq_dtype == 0) {
14079                           component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
14080                           /*
14081                            * set obp path for pathinfo
14082                            */
14083                           (void) snprintf(component, MAXPATHLEN,
14084                               "disk@%s", lun_addr);

14086                           if (mdi_pi_pathname_obp_set(*pip, component) !=
14087                               DDI_SUCCESS) {
14088                                   mptsas_log(mpt, CE_WARN, "mpt_sas driver "
14089                                       "unable to set obp-path for object %s",
14090                                       component);
14091                                   mdi_rtn = MDI_FAILURE;
14092                                   goto virt_create_done;
14093                           }
14094                   }

14096                   *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
14097                   if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14098                       MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14099                           if ((ndi_prop_update_int(DDI_DEV_T_NONE, *lun_dip,
14100                               "pm-capable", 1)) !=
14101                               DDI_PROP_SUCCESS) {
14102                                   mptsas_log(mpt, CE_WARN, "mptsas driver"
14103                                       "failed to create pm-capable "
14104                                       "property, target %d", target);
14105                                   mdi_rtn = MDI_FAILURE;
14106                                   goto virt_create_done;
14107                           }
14108                   }
14109                   /*
14110                    * Create the phy-num property
14111                    */
14112                   if (mdi_prop_update_int(*pip, "phy-num",
14113                       ptgt->m_phynum) != DDI_SUCCESS) {
14114                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
14115                               "create phy-num property for target %d lun %d",
14116                               target, lun);
14117                           mdi_rtn = MDI_FAILURE;
14118                           goto virt_create_done;
14119                   }
14120                   NDBG20(("new path:%s onlining,", MDI_PI(*pip)->pi_addr));
14121                   mdi_rtn = mdi_pi_online(*pip, 0);
14098                   if (mdi_rtn == MDI_SUCCESS) {
14099                           mutex_enter(&mpt->m_mutex);
14100                           if (mptsas_set_led_status(mpt, ptgt, 0) !=
14101                               DDI_SUCCESS) {
14102                                   NDBG14(("mptsas: clear LED for slot %x "
14103                                       "failed", ptgt->m_slot_num));
14104                           }
14105                           mutex_exit(&mpt->m_mutex);
14106                   }
14122                   if (mdi_rtn == MDI_NOT_SUPPORTED) {
14123                           mdi_rtn = MDI_FAILURE;
14124                   }
```



```
14125 virt_create_done:
14126                   if (*pip && mdi_rtn != MDI_SUCCESS) {
14127                           (void) mdi_pi_free(*pip, 0);
14128                           *pip = NULL;
14129                           *lun_dip = NULL;
14130                   }
14131           }

14133           scsi_hba_nodename_compatible_free(nodename, compatible);
14134           if (lun_addr != NULL) {
14135                   kmem_free(lun_addr, SCSI_MAXNAMELEN);
14136           }
14137           if (wwn_str != NULL) {
14138                   kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14139           }
14140           if (component != NULL) {
14141                   kmem_free(component, MAXPATHLEN);
14142           }

14144           return ((mdi_rtn == MDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14145 }

14147 static int
14148 mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *inq,
14149     char *guid, dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14150 {
14151           int                     target;
14152           int                     rval;
14153           int                     ndi_rtn = NDI_FAILURE;
14154           uint64_t                be_sas_wwn;
14155           char                    *nodename = NULL;
14156           char                    **compatible = NULL;
14157           int                     ncompatible = 0;
14158           int                     instance = 0;
14159           mptsas_t                *mpt = DIP2MPT(pdip);
14160           char                    *wwn_str = NULL;
14161           char                    *component = NULL;
14162           char                    *attached_wwn_str = NULL;
14163           uint8_t                 phy = 0xFF;
14164           uint64_t                sas_wwn;
14165           uint32_t                devinfo;
14166           uint16_t                dev_hdl;
14167           uint16_t                pdev_hdl;
14168           uint64_t                pdev_sas_wwn;
14169           uint64_t                dev_sas_wwn;
14170           uint32_t                pdev_info;
14171           uint8_t                 physport;
14172           uint8_t                 phy_id;
14173           uint32_t                page_address;
14174           uint16_t                bay_num, enclosure;
14175           char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
14176           uint32_t                dev_info;
14177           int64_t                 lun64 = 0;

14179           mutex_enter(&mpt->m_mutex);
14180           target = ptgt->m_devhdl;
14181           sas_wwn = ptgt->m_sas_wwn;
14182           devinfo = ptgt->m_deviceinfo;
14183           phy = ptgt->m_phynum;
14184           mutex_exit(&mpt->m_mutex);

14186           /*
14187            * generate compatible property with binding-set "mpt"
14188            */
14189           scsi_hba_nodename_compatible_get(inq, NULL, inq->inq_dtype, NULL,
14190               &nodename, &compatible, &ncompatible);
```

```
14192         /*
14193          * if nodename can't be determined then print a message and skip it
14194          */
14195         if (nodename == NULL) {
14196                 mptsas_log(mpt, CE_WARN, "mptsas found no compatible driver "
14197                     "for target %d lun %d", target, lun);
14198                 return (DDI_FAILURE);
14199         }

14201         ndi_rtn = ndi_devi_alloc(pdip, nodename,
14202             DEVI_SID_NODEID, lun_dip);

14204         /*
14205          * if lun alloc success, set props
14206          */
14207         if (ndi_rtn == NDI_SUCCESS) {

14209                 if (ndi_prop_update_int(DDI_DEV_T_NONE,
14210                     *lun_dip, LUN_PROP, lun) !=
14211                     DDI_PROP_SUCCESS) {
14212                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14213                             "property for target %d lun %d (LUN_PROP)",
14214                             target, lun);
14215                         ndi_rtn = NDI_FAILURE;
14216                         goto phys_create_done;
14217                 }

14219                 lun64 = (int64_t)lun;
14220                 if (ndi_prop_update_int64(DDI_DEV_T_NONE,
14221                     *lun_dip, LUN64_PROP, lun64) !=
14222                     DDI_PROP_SUCCESS) {
14223                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14224                             "property for target %d lun64 %d (LUN64_PROP)",
14225                             target, lun);
14226                         ndi_rtn = NDI_FAILURE;
14227                         goto phys_create_done;
14228                 }
14229                 if (ndi_prop_update_string_array(DDI_DEV_T_NONE,
14230                     *lun_dip, "compatible", compatible, ncompatible)
14231                     != DDI_PROP_SUCCESS) {
14232                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14233                             "property for target %d lun %d (COMPATIBLE)",
14234                             target, lun);
14235                         ndi_rtn = NDI_FAILURE;
14236                         goto phys_create_done;
14237                 }

14239                 /*
14240                  * We need the SAS WWN for non-multipath devices, so
14241                  * we'll use the same property as that multipathing
14242                  * devices need to present for MPAPI. If we don't have
14243                  * a WWN (e.g. parallel SCSI), don't create the prop.
14244                  */
14245                 wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
14246                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14247                 if (sas_wwn && ndi_prop_update_string(DDI_DEV_T_NONE,
14248                     *lun_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str)
14249                     != DDI_PROP_SUCCESS) {
14250                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14251                             "create property for SAS target %d lun %d "
14252                             "(target-port)", target, lun);
14253                         ndi_rtn = NDI_FAILURE;
14254                         goto phys_create_done;
14255                 }
```



```
14257                 be_sas_wwn = BE_64(sas_wwn);
14258                 if (sas_wwn && ndi_prop_update_byte_array(
14259                     DDI_DEV_T_NONE, *lun_dip, "port-wwn",
14260                     (uchar_t *)&be_sas_wwn, 8) != DDI_PROP_SUCCESS) {
14261                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14262                             "create property for SAS target %d lun %d "
14263                             "(port-wwn)", target, lun);
14264                         ndi_rtn = NDI_FAILURE;
14265                         goto phys_create_done;
14266                 } else if ((sas_wwn == 0) && (ndi_prop_update_int(
14267                     DDI_DEV_T_NONE, *lun_dip, "sata-phy", phy) !=
14268                     DDI_PROP_SUCCESS)) {
14269                         /*
14270                          * Direct attached SATA device without DeviceName
14271                          */
14272                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14273                             "create property for SAS target %d lun %d "
14274                             "(sata-phy)", target, lun);
14275                         ndi_rtn = NDI_FAILURE;
14276                         goto phys_create_done;
14277                 }

14279                 if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
14280                     *lun_dip, SAS_PROP) != DDI_PROP_SUCCESS) {
14281                         mptsas_log(mpt, CE_WARN, "mptsas unable to"
14282                             "create property for SAS target %d lun %d"
14283                             " (SAS_PROP)", target, lun);
14284                         ndi_rtn = NDI_FAILURE;
14285                         goto phys_create_done;
14286                 }
14287                 if (guid && (ndi_prop_update_string(DDI_DEV_T_NONE,
14288                     *lun_dip, NDI_GUID, guid) != DDI_SUCCESS)) {
14289                         mptsas_log(mpt, CE_WARN, "mptsas unable "
14290                             "to create guid property for target %d "
14291                             "lun %d", target, lun);
14292                         ndi_rtn = NDI_FAILURE;
14293                         goto phys_create_done;
14294                 }

14296                 /*
14297                  * The following code is to set properties for SM-HBA support,
14298                  * it doesn't apply to RAID volumes
14299                  */
14300                 if (ptgt->m_phymask == 0)
14301                         goto phys_raid_lun;

14303                 mutex_enter(&mpt->m_mutex);

14305                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14306                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14307                     (uint32_t)ptgt->m_devhdl;
14308                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14309                     &dev_hdl, &dev_sas_wwn, &dev_info,
14310                     &physport, &phy_id, &pdev_hdl,
14311                     &bay_num, &enclosure);
14312                 if (rval != DDI_SUCCESS) {
14313                         mutex_exit(&mpt->m_mutex);
14314                         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
14315                             "parent device for handle %d.", page_address);
14316                         ndi_rtn = NDI_FAILURE;
14317                         goto phys_create_done;
14318                 }

14320                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14321                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
14322                 rval = mptsas_get_sas_device_page0(mpt, page_address,
```

```
14323                       &dev_hdl, &pdev_sas_wwn, &pdev_info,
14324                       &physport, &phy_id, &pdev_hdl, &bay_num, &enclosure);
14325                   if (rval != DDI_SUCCESS) {
14326                           mutex_exit(&mpt->m_mutex);
14327                           mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14328                               "device for handle %d.", page_address);
14329                           ndi_rtn = NDI_FAILURE;
14330                           goto phys_create_done;
14331                   }

14333                   mutex_exit(&mpt->m_mutex);

14335                   /*
14336                    * If this device direct attached to the controller
14337                    * set the attached-port to the base wwid
14338                    */
14339                   if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
14340                       != DEVINFO_DIRECT_ATTACHED) {
14341                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14342                               pdev_sas_wwn);
14343                   } else {
14344                           /*
14345                            * Update the iport's attached-port to guid
14346                            */
14347                           if (sas_wwn == 0) {
14348                                   (void) sprintf(wwn_str, "p%x", phy);
14349                           } else {
14350                                   (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14351                           }
14352                           if (ddi_prop_update_string(DDI_DEV_T_NONE,
14353                               pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
14354                               DDI_PROP_SUCCESS) {
14355                                   mptsas_log(mpt, CE_WARN,
14356                                       "mptsas unable to create "
14357                                       "property for iport target-port"
14358                                       " %s (sas_wwn)",
14359                                       wwn_str);
14360                                   ndi_rtn = NDI_FAILURE;
14361                                   goto phys_create_done;
14362                           }

14364                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14365                               mpt->un.m_base_wwid);
14366                   }

14368                   if (ndi_prop_update_string(DDI_DEV_T_NONE,
14369                       *lun_dip, SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
14370                       DDI_PROP_SUCCESS) {
14371                           mptsas_log(mpt, CE_WARN,
14372                               "mptsas unable to create "
14373                               "property for iport attached-port %s (sas_wwn)",
14374                               attached_wwn_str);
14375                           ndi_rtn = NDI_FAILURE;
14376                           goto phys_create_done;
14377                   }

14379                   if (IS_SATA_DEVICE(dev_info)) {
14380                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
14381                               *lun_dip, MPTSAS_VARIANT, "sata") !=
14382                               DDI_PROP_SUCCESS) {
14383                                   mptsas_log(mpt, CE_WARN,
14384                                       "mptsas unable to create "
14385                                       "property for device variant ");
14386                                   ndi_rtn = NDI_FAILURE;
14387                                   goto phys_create_done;
14388                           }
```



```
14389                   }

14391                   if (IS_ATAPI_DEVICE(dev_info)) {
14392                           if (ndi_prop_update_string(DDI_DEV_T_NONE,
14393                               *lun_dip, MPTSAS_VARIANT, "atapi") !=
14394                               DDI_PROP_SUCCESS) {
14395                                   mptsas_log(mpt, CE_WARN,
14396                                       "mptsas unable to create "
14397                                       "property for device variant ");
14398                                   ndi_rtn = NDI_FAILURE;
14399                                   goto phys_create_done;
14400                           }
14401                   }

14403 phys_raid_lun:
14404                   /*
14405                    * if this is a SAS controller, and the target is a SATA
14406                    * drive, set the 'pm-capable' property for sd and if on
14407                    * an OPL platform, also check if this is an ATAPI
14408                    * device.
14409                    */
14410                   instance = ddi_get_instance(mpt->m_dip);
14411                   if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14412                       MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14413                           NDBG2(("mptsas%d: creating pm-capable property, "
14414                               "target %d", instance, target));

14416                           if ((ndi_prop_update_int(DDI_DEV_T_NONE,
14417                               *lun_dip, "pm-capable", 1)) !=
14418                               DDI_PROP_SUCCESS) {
14419                                   mptsas_log(mpt, CE_WARN, "mptsas "
14420                                       "failed to create pm-capable "
14421                                       "property, target %d", target);
14422                                   ndi_rtn = NDI_FAILURE;
14423                                   goto phys_create_done;
14424                           }

14426                   }

14428                   if ((inq->inq_dtype == 0) || (inq->inq_dtype == 5)) {
14429                           /*
14430                            * add 'obp-path' properties for devinfo
14431                            */
14432                           bzero(wwn_str, sizeof (wwn_str));
14433                           (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
14434                           component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
14435                           if (guid) {
14436                                   (void) snprintf(component, MAXPATHLEN,
14437                                       "disk@w%s,%x", wwn_str, lun);
14438                           } else {
14439                                   (void) snprintf(component, MAXPATHLEN,
14440                                       "disk@p%x,%x", phy, lun);
14441                           }
14442                           if (ddi_pathname_obp_set(*lun_dip, component)
14443                               != DDI_SUCCESS) {
14444                                   mptsas_log(mpt, CE_WARN, "mpt_sas driver "
14445                                       "unable to set obp-path for SAS "
14446                                       "object %s", component);
14447                                   ndi_rtn = NDI_FAILURE;
14448                                   goto phys_create_done;
14449                           }
14450                   }
14451                   /*
14452                    * Create the phy-num property for non-raid disk
14453                    */
14454                   if (ptgt->m_phymask != 0) {
```

```
14455                         if (ndi_prop_update_int(DDI_DEV_T_NONE,
14456                             *lun_dip, "phy-num", ptgt->m_phynum) !=
14457                             DDI_PROP_SUCCESS) {
14458                                 mptsas_log(mpt, CE_WARN, "mptsas driver "
14459                                     "failed to create phy-num property for "
14460                                     "target %d", target);
14461                                 ndi_rtn = NDI_FAILURE;
14462                                 goto phys_create_done;
14463                         }
14464                 }
14465 phys_create_done:
14466                 /*
14467                  * If props were setup ok, online the lun
14468                  */
14469                 if (ndi_rtn == NDI_SUCCESS) {
14470                         /*
14471                          * Try to online the new node
14472                          */
14473                         ndi_rtn = ndi_devi_online(*lun_dip, NDI_ONLINE_ATTACH);
14474                 }
14460                 if (ndi_rtn == NDI_SUCCESS) {
14461                         mutex_enter(&mpt->m_mutex);
14462                         if (mptsas_set_led_status(mpt, ptgt, 0) !=
14463                             DDI_SUCCESS) {
14464                                 NDBG14(("mptsas: clear LED for tgt %x "
14465                                     "failed", ptgt->m_slot_num));
14466                         }
14467                         mutex_exit(&mpt->m_mutex);
14468                 }

14476                 /*
14477                  * If success set rtn flag, else unwire alloc'd lun
14478                  */
14479                 if (ndi_rtn != NDI_SUCCESS) {
14480                         NDBG12(("mptsas driver unable to online "
14481                             "target %d lun %d", target, lun));
14482                         ndi_prop_remove_all(*lun_dip);
14483                         (void) ndi_devi_free(*lun_dip);
14484                         *lun_dip = NULL;
14485                 }
14486         }

14488         scsi_hba_nodename_compatible_free(nodename, compatible);

14490         if (wwn_str != NULL) {
14491                 kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14492         }
14493         if (component != NULL) {
14494                 kmem_free(component, MAXPATHLEN);
14495         }


14498         return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14499 }
```

_____unchanged_portion_omitted_

```
15361 #ifdef MPTSAS_GET_LED
15362 static int
15363 mptsas_get_led_status(mptsas_t *mpt, mptsas_target_t *ptgt,
15364     uint32_t *slotstatus)
15365 {
15366         return (mptsas_send_sep(mpt, ptgt, slotstatus,
15367             MPI2_SEP_REQ_ACTION_READ_STATUS));
15368 }
15369 #endif
15370 static int
```



```
15371 mptsas_set_led_status(mptsas_t *mpt, mptsas_target_t *ptgt, uint32_t slotstatus)
15372 {
15373         NDBG14(("mptsas_ioctl: set LED status %x for slot %x",
15374             slotstatus, ptgt->m_slot_num));
15375         return (mptsas_send_sep(mpt, ptgt, &slotstatus,
15376             MPI2_SEP_REQ_ACTION_WRITE_STATUS));
15377 }
15378 /*
15379  *  send sep request, use enclosure/slot addressing
15380  */
15381 static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
15382     uint32_t *status, uint8_t act)
15383 {
15384         Mpi2SepRequest_t        req;
15385         Mpi2SepReply_t          rep;
15386         int                     ret;

15388         ASSERT(mutex_owned(&mpt->m_mutex));

15390         bzero(&req, sizeof (req));
15391         bzero(&rep, sizeof (rep));

15393         /* Do nothing for RAID volumes */
15394         if (ptgt->m_phymask == 0) {
15395                 NDBG14(("mptsas_send_sep: Skip RAID volumes"));
15396                 return (DDI_FAILURE);
15397         }

15399         req.Function = MPI2_FUNCTION_SCSI_ENCLOSURE_PROCESSOR;
15400         req.Action = act;
15401         req.Flags = MPI2_SEP_REQ_FLAGS_ENCLOSURE_SLOT_ADDRESS;
15402         req.EnclosureHandle = LE_16(ptgt->m_enclosure);
15403         req.Slot = LE_16(ptgt->m_slot_num);
15404         if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
15405                 req.SlotStatus = LE_32(*status);
15406         }
15407         ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
15408             sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
15409         if (ret != 0) {
15410                 mptsas_log(mpt, CE_NOTE, "mptsas_send_sep: passthru SEP "
15411                     "Processor Request message error %d", ret);
15412                 return (DDI_FAILURE);
15413         }
15414         /* do passthrough success, check the ioc status */
15415         if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
15416                 if ((LE_16(rep.IOCStatus) & MPI2_IOCSTATUS_MASK) ==
15417                     MPI2_IOCSTATUS_INVALID_FIELD) {
15418                         mptsas_log(mpt, CE_NOTE, "send sep act %x: Not "
15419                             "supported action, loginfo %x", act,
15420                             LE_32(rep.IOCLogInfo));
15421                         return (DDI_FAILURE);
15422                 }
15423                 mptsas_log(mpt, CE_NOTE, "send_sep act %x: ioc "
15424                     "status:%x", act, LE_16(rep.IOCStatus));
15425                 return (DDI_FAILURE);
15426         }
15427         if (act != MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
15428                 *status = LE_32(rep.SlotStatus);
15429         }

15431         return (DDI_SUCCESS);
15432 }

15367 int
15368 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
15369     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
```

```
15370     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
15371 {
15372         ddi_dma_cookie_t        new_cookie;
15373         size_t                  alloc_len;
15374         uint_t                  ncookie;

15376         if (cookiep == NULL)
15377                 cookiep = &new_cookie;

15379         if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
15380             NULL, dma_hdp) != DDI_SUCCESS) {
15381                 dma_hdp = NULL;
15382                 return (FALSE);
15383         }

15385         if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
15386             DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
15387             acc_hdp) != DDI_SUCCESS) {
15388                 ddi_dma_free_handle(dma_hdp);
15389                 dma_hdp = NULL;
15390                 return (FALSE);
15391         }

15393         if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
15394             (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
15395             cookiep, &ncookie) != DDI_DMA_MAPPED) {
15396                 (void) ddi_dma_mem_free(acc_hdp);
15397                 ddi_dma_free_handle(dma_hdp);
15398                 dma_hdp = NULL;
15399                 return (FALSE);
15400         }

15402         return (TRUE);
15403 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   43458 Fri Aug 31 14:55:23 2012
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
re #8346 rb2639 KT disk failures
**********************************************************
______unchanged_portion_omitted_

 156 /*
 157  * preferred pkt_private length in 64-bit quantities
 158  */
 159 #ifdef  _LP64
 160 #define PKT_PRIV_SIZE   2
 161 #define PKT_PRIV_LEN    16      /* in bytes */
 162 #else /* _ILP32 */
 163 #define PKT_PRIV_SIZE   1
 164 #define PKT_PRIV_LEN    8       /* in bytes */
 165 #endif

 167 #define PKT2CMD(pkt)    ((struct mptsas_cmd *)((pkt)->pkt_ha_private))
 168 #define CMD2PKT(cmdp)   ((struct scsi_pkt *)((cmdp)->cmd_pkt))
 169 #define EXTCMDS_STATUS_SIZE (sizeof (struct scsi_arq_status))

 171 /*
 172  * get offset of item in structure
 173  */
 174 #define MPTSAS_GET_ITEM_OFF(type, member) ((size_t)(&((type *)0)->member))

 176 /*
 177  * WWID provided by LSI firmware is generated by firmware but the WWID is not
 178  * IEEE NAA standard format, OBP has no chance to distinguish format of unit
 179  * address. According LSI's confirmation, the top nibble of RAID WWID is
 180  * meanless, so the consensus between Solaris and OBP is to replace top nibble
 181  * of WWID provided by LSI to "3" always to hint OBP that this is a RAID WWID
 182  * format unit address.
 183  */
 184 #define MPTSAS_RAID_WWID(wwid) \
 185         ((wwid & 0x0FFFFFFFFFFFFFFF) | 0x3000000000000000)

 187 typedef struct mptsas_target {
 188                 uint64_t                m_sas_wwn;      /* hash key1 */
 189                 mptsas_phymask_t        m_phymask;      /* hash key2 */
 190                 /*
 191                  * m_dr_flag is a flag for DR, make sure the member
 192                  * take the place of dr_flag of mptsas_hash_data.
 193                  */
 194                 uint8_t                 m_dr_flag;      /* dr_flag */
 195                 uint16_t                m_devhdl;
 196                 uint32_t                m_deviceinfo;
 197                 uint8_t                 m_phynum;
 198                 uint32_t                m_dups;
 199                 int32_t                 m_timeout;
 200                 int32_t                 m_timebase;
 201                 int32_t                 m_t_throttle;
 202                 int32_t                 m_t_ncmds;
 203                 int32_t                 m_reset_delay;
 204                 int32_t                 m_t_nwait;

 206                 uint16_t                m_qfull_retry_interval;
 207                 uint8_t                 m_qfull_retries;
 208                 uint16_t                m_enclosure;
 209                 uint16_t                m_slot_num;
 210                 uint32_t                m_tgt_unconfigured;
 211                 uint32_t                m_timeout_interval;
 212                 uint8_t                 m_timeout_count;

 214 } mptsas_target_t;
______unchanged_portion_omitted_
```