```
**********************************************************
  411925 Tue Dec  4 16:30:09 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
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
  25  */

  27 /*
  28  * Copyright (c) 2000 to 2010, LSI Corporation.
  29  * All rights reserved.
  30  *
  31  * Redistribution and use in source and binary forms of all code within
  32  * this file that is exclusively owned by LSI, with or without
  33  * modification, is permitted provided that, in addition to the CDDL 1.0
  34  * License requirements, the following conditions are met:
  35  *
  36  *    Neither the name of the author nor the names of its contributors may be
  37  *    used to endorse or promote products derived from this software without
  38  *    specific prior written permission.
  39  *
  40  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  41  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  42  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  43  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
  44  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
  45  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
  46  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
  47  * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
  48  * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  49  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
  50  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
  51  * DAMAGE.
  52  */

  54 /*
  55  * mptsas - This is a driver based on LSI Logic's MPT2.0 interface.
  56  *
  57  */

  59 #if defined(lint) || defined(DEBUG)
  60 #define MPTSAS_DEBUG
```



```
  61 #endif

  63 /*
  64  * standard header files.
  65  */
  66 #include <sys/note.h>
  67 #include <sys/scsi/scsi.h>
  68 #include <sys/pci.h>
  69 #include <sys/file.h>
  70 #include <sys/policy.h>
  71 #include <sys/sysevent.h>
  72 #include <sys/sysevent/eventdefs.h>
  73 #include <sys/sysevent/dr.h>
  74 #include <sys/sata/sata_defs.h>
  75 #include <sys/scsi/generic/sas.h>
  76 #include <sys/scsi/impl/scsi_sas.h>

  78 #pragma pack(1)
  79 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_type.h>
  80 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2.h>
  81 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>
  82 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_init.h>
  83 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_ioc.h>
  84 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_sas.h>
  85 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>
  86 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_raid.h>
  87 #pragma pack()

  89 /*
  90  * private header files.
  91  *
  92  */
  93 #include <sys/scsi/impl/scsi_reset_notify.h>
  94 #include <sys/scsi/adapters/mpt_sas/mptsas_var.h>
  95 #include <sys/scsi/adapters/mpt_sas/mptsas_ioctl.h>
  96 #include <sys/scsi/adapters/mpt_sas/mptsas_smhba.h>
  97 #include <sys/raidioctl.h>

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

 101 /*
 102  * FMA header files
 103  */
 104 #include <sys/ddifm.h>
 105 #include <sys/fm/protocol.h>
 106 #include <sys/fm/util.h>
 107 #include <sys/fm/io/ddi.h>

 109 /*
 110  * autoconfiguration data and routines.
 111  */
 112 static int mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
 113 static int mptsas_detach(dev_info_t *devi, ddi_detach_cmd_t cmd);
 114 static int mptsas_power(dev_info_t *dip, int component, int level);

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

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

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

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

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

192 static int mptsas_quiesce_bus(mptsas_t *mpt);
```



```
193 static int mptsas_unquiesce_bus(mptsas_t *mpt);

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

198 static void mptsas_ncmds_checkdrain(void *arg);

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

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

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

218 static void mptsas_restart_cmd(void *);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

323 static dev_info_t *mptsas_find_child(dev_info_t *pdip, char *name);
324 static dev_info_t *mptsas_find_child_phy(dev_info_t *pdip, uint8_t phy);
```



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

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

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

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

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

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


356 /*
357  * Enumeration / DR functions
358  */
359 static void mptsas_config_all(dev_info_t *pdip);
360 static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
361     dev_info_t **lundip);
362 static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
363     dev_info_t **lundip);

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

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

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

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

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

384 static void mptsas_offline_missed_luns(dev_info_t *pdip,
385     uint16_t *repluns, int lun_cnt, mptsas_target_t *ptgt);
386 static int mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,
```

```
387     mdi_pathinfo_t *rpip, uint_t flags);

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

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

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

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

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

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

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

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

450 extern pri_t minclsyspri, maxclsyspri;

452 /*
```



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

459 /*
460  * Tunable timeout value for Inquiry VPD page 0x83
461  * By default the value is 30 seconds.
462  */
463 int mptsas_inq83_retry_timeout = 30;

465 /*
466  * This is used to allocate memory for message frame storage, not for
467  * data I/O DMA. All message frames must be stored in the first 4G of
468  * physical memory.
469  */
470 ddi_dma_attr_t mptsas_dma_attrs = {
471         DMA_ATTR_V0,    /* attribute layout version             */
472         0x0ull,         /* address low - should be 0 (longlong) */
473         0xffffffffull,  /* address high - 32-bit max range      */
474         0x00ffffffull,  /* count max - max DMA object size      */
475         4,              /* allocation alignment requirements    */
476         0x78,           /* burstsizes - binary encoded values   */
477         1,              /* minxfer - gran. of DMA engine        */
478         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
479         0xffffffffull,  /* max segment size (DMA boundary)      */
480         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
481         512,            /* granularity - device transfer size   */
482         0               /* flags, set to 0                      */
483 };
_____unchanged_portion_omitted_

5786 static void
5787 mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
5788     dev_info_t *parent)
5789 {
5790         mptsas_target_t *ptgt = NULL;
5791         mptsas_smp_t    *psmp = NULL;
5792         mptsas_t        *mpt = (void *)topo_node->mpt;
5793         uint16_t        devhdl;
5794         uint16_t        attached_devhdl;
5795         uint64_t        sas_wwn = 0;
5796         int             rval = 0;
5797         uint32_t        page_address;
5798         uint8_t         phy, flags;
5799         char            *addr = NULL;
5800         dev_info_t      *lundip;
5801         int             circ = 0, circ1 = 0;
5802         char            attached_wwnstr[MPTSAS_WWN_STRLEN];

5804         NDBG20(("mptsas%d handle_topo_change enter", mpt->m_instance));

5806         ASSERT(mutex_owned(&mpt->m_mutex));

5808         switch (topo_node->event) {
5809         case MPTSAS_DR_EVENT_RECONFIG_TARGET:
5810         {
5811                 char *phy_mask_name;
5812                 mptsas_phymask_t phymask = 0;

5814                 if (topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5815                         /*
5816                          * Get latest RAID info.
5817                          */
5818                         (void) mptsas_get_raid_info(mpt);
```

```
5819                        ptgt = mptsas_search_by_devhdl(
5820                            &mpt->m_active->m_tgttbl, topo_node->devhdl);
5821                        if (ptgt == NULL)
5822                                break;
5823                } else {
5824                        ptgt = (void *)topo_node->object;
5825                }

5827                if (ptgt == NULL) {
5828                        /*
5829                         * If a Phys Disk was deleted, RAID info needs to be
5830                         * updated to reflect the new topology.
5831                         */
5832                        (void) mptsas_get_raid_info(mpt);

5834                        /*
5835                         * Get sas device page 0 by DevHandle to make sure if
5836                         * SSP/SATA end device exist.
5837                         */
5838                        page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
5839                            MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
5840                            topo_node->devhdl;

5842                        rval = mptsas_get_target_device_info(mpt, page_address,
5843                            &devhdl, &ptgt);
5844                        if (rval == DEV_INFO_WRONG_DEVICE_TYPE) {
5845                                mptsas_log(mpt, CE_NOTE,
5846                                    "mptsas_handle_topo_change: target %d is "
5847                                    "not a SAS/SATA device. \n",
5848                                    topo_node->devhdl);
5849                        } else if (rval == DEV_INFO_FAIL_ALLOC) {
5850                                mptsas_log(mpt, CE_NOTE,
5851                                    "mptsas_handle_topo_change: could not "
5852                                    "allocate memory. \n");
5853                        }
5854                        /*
5855                         * If rval is DEV_INFO_PHYS_DISK than there is nothing
5856                         * else to do, just leave.
5857                         */
5858                        if (rval != DEV_INFO_SUCCESS) {
5859                                return;
5860                        }
5861                }

5863                ASSERT(ptgt->m_devhdl == topo_node->devhdl);

5865                mutex_exit(&mpt->m_mutex);
5866                flags = topo_node->flags;

5868                if (flags == MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED) {
5869                        phymask = ptgt->m_phymask;
5870                        phy_mask_name = kmem_zalloc(MPTSAS_MAX_PHYS, KM_SLEEP);
5871                        (void) sprintf(phy_mask_name, "%x", phymask);
5872                        parent = scsi_hba_iport_find(mpt->m_dip,
5873                            phy_mask_name);
5874                        kmem_free(phy_mask_name, MPTSAS_MAX_PHYS);
5875                        if (parent == NULL) {
5876                                mptsas_log(mpt, CE_WARN, "Failed to find a "
5877                                    "iport for PD, should not happen!");
5878                                mutex_enter(&mpt->m_mutex);
5879                                break;
5880                        }
5881                }

5883                if (flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) {
5884                        ndi_devi_enter(parent, &circ1);
```



```
5885                        (void) mptsas_config_raid(parent, topo_node->devhdl,
5886                            &lundip);
5887                        ndi_devi_exit(parent, circ1);
5888                } else {
5889                        /*
5890                         * hold nexus for bus configure
5891                         */
5892                        ndi_devi_enter(scsi_vhci_dip, &circ);
5893                        ndi_devi_enter(parent, &circ1);
5894                        rval = mptsas_config_target(parent, ptgt);
5895                        /*
5896                         * release nexus for bus configure
5897                         */
5898                        ndi_devi_exit(parent, circ1);
5899                        ndi_devi_exit(scsi_vhci_dip, circ);

5901                        /*
5902                         * Add parent's props for SMHBA support
5903                         */
5904                        if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
5905                                bzero(attached_wwnstr,
5906                                    sizeof (attached_wwnstr));
5907                                (void) sprintf(attached_wwnstr, "w%016"PRIx64,
5908                                    ptgt->m_sas_wwn);
5909                                if (ddi_prop_update_string(DDI_DEV_T_NONE,
5910                                    parent,
5911                                    SCSI_ADDR_PROP_ATTACHED_PORT,
5912                                    attached_wwnstr)
5913                                    != DDI_PROP_SUCCESS) {
5914                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5915                                            parent,
5916                                            SCSI_ADDR_PROP_ATTACHED_PORT);
5917                                        mptsas_log(mpt, CE_WARN, "Failed to"
5918                                            "attached-port props");
5919                                        return;
5920                                }
5921                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5922                                    MPTSAS_NUM_PHYS, 1) !=
5923                                    DDI_PROP_SUCCESS) {
5924                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5925                                            parent, MPTSAS_NUM_PHYS);
5926                                        mptsas_log(mpt, CE_WARN, "Failed to"
5927                                            " create num-phys props");
5928                                        return;
5929                                }

5931                                /*
5932                                 * Update PHY info for smhba
5933                                 */
5934                                mutex_enter(&mpt->m_mutex);
5935                                if (mptsas_smhba_phy_init(mpt)) {
5936                                        mutex_exit(&mpt->m_mutex);
5937                                        mptsas_log(mpt, CE_WARN, "mptsas phy"
5938                                            " update failed");
5939                                        return;
5940                                }
5941                                mutex_exit(&mpt->m_mutex);
5942                                mptsas_smhba_set_phy_props(mpt,
5943                                    ddi_get_name_addr(parent), parent,
5944                                    1, &attached_devhdl);
5945                                if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
5946                                    MPTSAS_VIRTUAL_PORT, 0) !=
5947                                    DDI_PROP_SUCCESS) {
5948                                        (void) ddi_prop_remove(DDI_DEV_T_NONE,
5949                                            parent, MPTSAS_VIRTUAL_PORT);
5950                                        mptsas_log(mpt, CE_WARN,
```

```
5951                                            "mptsas virtual-port"
5952                                            "port prop update failed");
5953                                    return;
5954                            }
5955                    }
5956            }
5957            mutex_enter(&mpt->m_mutex);

5959            NDBG20(("mptsas%d handle_topo_change to online devhdl:%x, "
5960                "phymask:%x.", mpt->m_instance, ptgt->m_devhdl,
5961                ptgt->m_phymask));
5962            break;
5963    }
5964    case MPTSAS_DR_EVENT_OFFLINE_TARGET:
5965    {
5966            mptsas_hash_table_t *tgttbl = &mpt->m_active->m_tgttbl;
5967            devhdl = topo_node->devhdl;
5968            ptgt = mptsas_search_by_devhdl(tgttbl, devhdl);
5969            if (ptgt == NULL)
5970                    break;

5972            sas_wwn = ptgt->m_sas_wwn;
5973            phy = ptgt->m_phynum;

5975            addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);

5977            if (sas_wwn) {
5978                    (void) sprintf(addr, "w%016"PRIx64, sas_wwn);
5979            } else {
5980                    (void) sprintf(addr, "p%x", phy);
5981            }
5982            ASSERT(ptgt->m_devhdl == devhdl);

5984            if ((topo_node->flags == MPTSAS_TOPO_FLAG_RAID_ASSOCIATED) ||
5985                (topo_node->flags ==
5986                MPTSAS_TOPO_FLAG_RAID_PHYSDRV_ASSOCIATED)) {
5987                    /*
5988                     * Get latest RAID info if RAID volume status changes
5989                     * or Phys Disk status changes
5990                     */
5991                    (void) mptsas_get_raid_info(mpt);
5992            }
5993            /*
5994             * Abort all outstanding command on the device
5995             */
5996            rval = mptsas_do_scsi_reset(mpt, devhdl);
5997            if (rval) {
5998                    NDBG20(("mptsas%d handle_topo_change to reset target "
5999                        "before offline devhdl:%x, phymask:%x, rval:%x",
6000                        mpt->m_instance, ptgt->m_devhdl, ptgt->m_phymask,
6001                        rval));
6002            }

6004            mutex_exit(&mpt->m_mutex);

6006            ndi_devi_enter(scsi_vhci_dip, &circ);
6007            ndi_devi_enter(parent, &circ1);
6008            rval = mptsas_offline_target(parent, addr);
6009            ndi_devi_exit(parent, circ1);
6010            ndi_devi_exit(scsi_vhci_dip, circ);
6011            NDBG20(("mptsas%d handle_topo_change to offline devhdl:%x, "
6012                "phymask:%x, rval:%x", mpt->m_instance,
6013                ptgt->m_devhdl, ptgt->m_phymask, rval));

6015            kmem_free(addr, SCSI_MAXNAMELEN);
```



```
6017            /*
6018             * Clear parent's props for SMHBA support
6019             */
6020            flags = topo_node->flags;
6021            if (flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) {
6022                    bzero(attached_wwnstr, sizeof (attached_wwnstr));
6023                    if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6024                        SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6025                        DDI_PROP_SUCCESS) {
6026                            (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6027                                SCSI_ADDR_PROP_ATTACHED_PORT);
6028                            mptsas_log(mpt, CE_WARN, "mptsas attached port "
6029                                "prop update failed");
6030                            break;
6031                    }
6032                    if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6033                        MPTSAS_NUM_PHYS, 0) !=
6034                        DDI_PROP_SUCCESS) {
6035                            (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6036                                MPTSAS_NUM_PHYS);
6037                            mptsas_log(mpt, CE_WARN, "mptsas num phys "
6038                                "prop update failed");
6039                            break;
6040                    }
6041                    if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6042                        MPTSAS_VIRTUAL_PORT, 1) !=
6043                        DDI_PROP_SUCCESS) {
6044                            (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6045                                MPTSAS_VIRTUAL_PORT);
6046                            mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6047                                "prop update failed");
6048                            break;
6049                    }
6050            }

6052            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            }
6053            if (rval == DDI_SUCCESS) {
6054                    mptsas_tgt_free(&mpt->m_active->m_tgttbl,
6055                        ptgt->m_sas_wwn, ptgt->m_phymask);
6056                    ptgt = NULL;
6057            } else {
6058                    /*
6059                     * clean DR_INTRANSITION flag to allow I/O down to
6060                     * PHCI driver since failover finished.
6061                     * Invalidate the devhdl
6062                     */
6063                    ptgt->m_devhdl = MPTSAS_INVALID_DEVHDL;
6064                    ptgt->m_tgt_unconfigured = 0;
6065                    mutex_enter(&mpt->m_tx_waitq_mutex);
6066                    ptgt->m_dr_flag = MPTSAS_DR_INACTIVE;
6067                    mutex_exit(&mpt->m_tx_waitq_mutex);
6068            }

6070            /*
6071             * Send SAS IO Unit Control to free the dev handle
6072             */
6073            if ((flags == MPTSAS_TOPO_FLAG_DIRECT_ATTACHED_DEVICE) ||
6074                (flags == MPTSAS_TOPO_FLAG_EXPANDER_ATTACHED_DEVICE)) {
6075                    rval = mptsas_free_devhdl(mpt, devhdl);

6077                    NDBG20(("mptsas%d handle_topo_change to remove "
6078                        "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
```

```
6079                                rval));
6080                }

6082                break;
6083        }
6084        case MPTSAS_TOPO_FLAG_REMOVE_HANDLE:
6085        {
6086                devhdl = topo_node->devhdl;
6087                /*
6088                 * If this is the remove handle event, do a reset first.
6089                 */
6090                if (topo_node->event == MPTSAS_TOPO_FLAG_REMOVE_HANDLE) {
6091                        rval = mptsas_do_scsi_reset(mpt, devhdl);
6092                        if (rval) {
6093                                NDBG20(("mpt%d reset target before remove "
6094                                    "devhdl:%x, rval:%x", mpt->m_instance,
6095                                    devhdl, rval));
6096                        }
6097                }

6099                /*
6100                 * Send SAS IO Unit Control to free the dev handle
6101                 */
6102                rval = mptsas_free_devhdl(mpt, devhdl);
6103                NDBG20(("mptsas%d handle_topo_change to remove "
6104                    "devhdl:%x, rval:%x", mpt->m_instance, devhdl,
6105                    rval));
6106                break;
6107        }
6108        case MPTSAS_DR_EVENT_RECONFIG_SMP:
6109        {
6110                mptsas_smp_t smp;
6111                dev_info_t *smpdip;
6112                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;

6114                devhdl = topo_node->devhdl;

6116                page_address = (MPI2_SAS_EXPAND_PGAD_FORM_HNDL &
6117                    MPI2_SAS_EXPAND_PGAD_FORM_MASK) | (uint32_t)devhdl;
6118                rval = mptsas_get_sas_expander_page0(mpt, page_address, &smp);
6119                if (rval != DDI_SUCCESS) {
6120                        mptsas_log(mpt, CE_WARN, "failed to online smp, "
6121                            "handle %x", devhdl);
6122                        return;
6123                }

6125                psmp = mptsas_smp_alloc(smptbl, &smp);
6126                if (psmp == NULL) {
6127                        return;
6128                }

6130                mutex_exit(&mpt->m_mutex);
6131                ndi_devi_enter(parent, &circ1);
6132                (void) mptsas_online_smp(parent, psmp, &smpdip);
6133                ndi_devi_exit(parent, circ1);

6135                mutex_enter(&mpt->m_mutex);
6136                break;
6137        }
6138        case MPTSAS_DR_EVENT_OFFLINE_SMP:
6139        {
6140                mptsas_hash_table_t *smptbl = &mpt->m_active->m_smptbl;
6141                devhdl = topo_node->devhdl;
6142                uint32_t dev_info;

6144                psmp = mptsas_search_by_devhdl(smptbl, devhdl);
```



```
6145                if (psmp == NULL)
6146                        break;
6147                /*
6148                 * The mptsas_smp_t data is released only if the dip is offlined
6149                 * successfully.
6150                 */
6151                mutex_exit(&mpt->m_mutex);

6153                ndi_devi_enter(parent, &circ1);
6154                rval = mptsas_offline_smp(parent, psmp, NDI_DEVI_REMOVE);
6155                ndi_devi_exit(parent, circ1);

6157                dev_info = psmp->m_deviceinfo;
6158                if ((dev_info & DEVINFO_DIRECT_ATTACHED) ==
6159                    DEVINFO_DIRECT_ATTACHED) {
6160                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6161                            MPTSAS_VIRTUAL_PORT, 1) !=
6162                            DDI_PROP_SUCCESS) {
6163                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6164                                    MPTSAS_VIRTUAL_PORT);
6165                                mptsas_log(mpt, CE_WARN, "mptsas virtual port "
6166                                    "prop update failed");
6167                                return;
6168                        }
6169                        /*
6170                         * Check whether the smp connected to the iport,
6171                         */
6172                        if (ddi_prop_update_int(DDI_DEV_T_NONE, parent,
6173                            MPTSAS_NUM_PHYS, 0) !=
6174                            DDI_PROP_SUCCESS) {
6175                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6176                                    MPTSAS_NUM_PHYS);
6177                                mptsas_log(mpt, CE_WARN, "mptsas num phys"
6178                                    "prop update failed");
6179                                return;
6180                        }
6181                        /*
6182                         * Clear parent's attached-port props
6183                         */
6184                        bzero(attached_wwnstr, sizeof (attached_wwnstr));
6185                        if (ddi_prop_update_string(DDI_DEV_T_NONE, parent,
6186                            SCSI_ADDR_PROP_ATTACHED_PORT, attached_wwnstr) !=
6187                            DDI_PROP_SUCCESS) {
6188                                (void) ddi_prop_remove(DDI_DEV_T_NONE, parent,
6189                                    SCSI_ADDR_PROP_ATTACHED_PORT);
6190                                mptsas_log(mpt, CE_WARN, "mptsas attached port "
6191                                    "prop update failed");
6192                                return;
6193                        }
6194                }

6196                mutex_enter(&mpt->m_mutex);
6197                NDBG20(("mptsas%d handle_topo_change to remove devhdl:%x, "
6198                    "rval:%x", mpt->m_instance, psmp->m_devhdl, rval));
6199                if (rval == DDI_SUCCESS) {
6200                        mptsas_smp_free(smptbl, psmp->m_sasaddr,
6201                            psmp->m_phymask);
6202                } else {
6203                        psmp->m_devhdl = MPTSAS_INVALID_DEVHDL;
6204                }

6206                bzero(attached_wwnstr, sizeof (attached_wwnstr));

6208                break;
6209        }
6210        default:
```

```
6211                 return;
6212         }
6213 }
```

*unchanged_portion_omitted_*

```
11261 static int
11262 mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode, cred_t *credp,
11263     int *rval)
11264 {
11265         int                     status = 0;
11266         mptsas_t                *mpt;
11267         mptsas_update_flash_t   flashdata;
11268         mptsas_pass_thru_t      passthru_data;
11269         mptsas_adapter_data_t   adapter_data;
11270         mptsas_pci_info_t       pci_info;
11271         int                     copylen;

11273         int                     iport_flag = 0;
11274         dev_info_t              *dip = NULL;
11275         mptsas_phymask_t        phymask = 0;
11276         struct devctl_iocdata   *dcp = NULL;
11277         uint32_t                slotstatus = 0;
11278         char                    *addr = NULL;
11279         mptsas_target_t         *ptgt = NULL;

11281         *rval = MPTIOCTL_STATUS_GOOD;
11282         if (secpolicy_sys_config(credp, B_FALSE) != 0) {
11283                 return (EPERM);
11284         }

11286         mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
11287         if (mpt == NULL) {
11288                 /*
11289                  * Called from iport node, get the states
11290                  */
11291                 iport_flag = 1;
11292                 dip = mptsas_get_dip_from_dev(dev, &phymask);
11293                 if (dip == NULL) {
11294                         return (ENXIO);
11295                 }
11296                 mpt = DIP2MPT(dip);
11297         }
11298         /* Make sure power level is D0 before accessing registers */
11299         mutex_enter(&mpt->m_mutex);
11300         if (mpt->m_options & MPTSAS_OPT_PM) {
11301                 (void) pm_busy_component(mpt->m_dip, 0);
11302                 if (mpt->m_power_level != PM_LEVEL_D0) {
11303                         mutex_exit(&mpt->m_mutex);
11304                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
11305                             DDI_SUCCESS) {
11306                                 mptsas_log(mpt, CE_WARN,
11307                                     "mptsas%d: mptsas_ioctl: Raise power "
11308                                     "request failed.", mpt->m_instance);
11309                                 (void) pm_idle_component(mpt->m_dip, 0);
11310                                 return (ENXIO);
11311                         }
11312                 } else {
11313                         mutex_exit(&mpt->m_mutex);
11314                 }
11315         } else {
11316                 mutex_exit(&mpt->m_mutex);
11317         }

11319         if (iport_flag) {
11320                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
11329                 if (status != 0) {
```



```
11321                         goto out;
11322                 }
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         }
11323         switch (cmd) {
11324                 case MPTIOCTL_UPDATE_FLASH:
11325                         if (ddi_copyin((void *)data, &flashdata,
11326                                 sizeof (struct mptsas_update_flash), mode)) {
11327                                 status = EFAULT;
11328                                 break;
11329                         }
```

```
11331                       mutex_enter(&mpt->m_mutex);
11332                       if (mptsas_update_flash(mpt,
11333                           (caddr_t)(long)flashdata.PtrBuffer,
11334                           flashdata.ImageSize, flashdata.ImageType, mode)) {
11335                               status = EFAULT;
11336                       }
11337
11338                       /*
11339                        * Reset the chip to start using the new
11340                        * firmware.  Reset if failed also.
11341                        */
11342                       mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11343                       if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
11344                               status = EFAULT;
11345                       }
11346                       mutex_exit(&mpt->m_mutex);
11347                       break;
11348               case MPTIOCTL_PASS_THRU:
11349                       /*
11350                        * The user has requested to pass through a command to
11351                        * be executed by the MPT firmware.  Call our routine
11352                        * which does this.  Only allow one passthru IOCTL at
11353                        * one time. Other threads will block on
11354                        * m_passthru_mutex, which is of adaptive variant.
11355                        */
11356                       if (ddi_copyin((void *)data, &passthru_data,
11357                           sizeof (mptsas_pass_thru_t), mode)) {
11358                               status = EFAULT;
11359                               break;
11360                       }
11361                       mutex_enter(&mpt->m_passthru_mutex);
11362                       mutex_enter(&mpt->m_mutex);
11363                       status = mptsas_pass_thru(mpt, &passthru_data, mode);
11364                       mutex_exit(&mpt->m_mutex);
11365                       mutex_exit(&mpt->m_passthru_mutex);
11366
11367                       break;
11368               case MPTIOCTL_GET_ADAPTER_DATA:
11369                       /*
11370                        * The user has requested to read adapter data.  Call
11371                        * our routine which does this.
11372                        */
11373                       bzero(&adapter_data, sizeof (mptsas_adapter_data_t));
11374                       if (ddi_copyin((void *)data, (void *)&adapter_data,
11375                           sizeof (mptsas_adapter_data_t), mode)) {
11376                               status = EFAULT;
11377                               break;
11378                       }
11379                       if (adapter_data.StructureLength >=
11380                           sizeof (mptsas_adapter_data_t)) {
11381                               adapter_data.StructureLength = (uint32_t)
11382                                   sizeof (mptsas_adapter_data_t);
11383                               copylen = sizeof (mptsas_adapter_data_t);
11384                               mutex_enter(&mpt->m_mutex);
11385                               mptsas_read_adapter_data(mpt, &adapter_data);
11386                               mutex_exit(&mpt->m_mutex);
11387                       } else {
11388                               adapter_data.StructureLength = (uint32_t)
11389                                   sizeof (mptsas_adapter_data_t);
11390                               copylen = sizeof (adapter_data.StructureLength);
11391                               *rval = MPTIOCTL_STATUS_LEN_TOO_SHORT;
11392                       }
11393                       if (ddi_copyout((void *)(&adapter_data), (void *)data,
11394                           copylen, mode) != 0) {
11395                               status = EFAULT;
```



```
11396                       }
11397                       break;
11398               case MPTIOCTL_GET_PCI_INFO:
11399                       /*
11400                        * The user has requested to read pci info.  Call
11401                        * our routine which does this.
11402                        */
11403                       bzero(&pci_info, sizeof (mptsas_pci_info_t));
11404                       mutex_enter(&mpt->m_mutex);
11405                       mptsas_read_pci_info(mpt, &pci_info);
11406                       mutex_exit(&mpt->m_mutex);
11407                       if (ddi_copyout((void *)(&pci_info), (void *)data,
11408                           sizeof (mptsas_pci_info_t), mode) != 0) {
11409                               status = EFAULT;
11410                       }
11411                       break;
11412               case MPTIOCTL_RESET_ADAPTER:
11413                       mutex_enter(&mpt->m_mutex);
11414                       mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11415                       if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
11416                               mptsas_log(mpt, CE_WARN, "reset adapter IOCTL "
11417                                   "failed");
11418                               status = EFAULT;
11419                       }
11420                       mutex_exit(&mpt->m_mutex);
11421                       break;
11422               case MPTIOCTL_DIAG_ACTION:
11423                       /*
11424                        * The user has done a diag buffer action.  Call our
11425                        * routine which does this.  Only allow one diag action
11426                        * at one time.
11427                        */
11428                       mutex_enter(&mpt->m_mutex);
11429                       if (mpt->m_diag_action_in_progress) {
11430                               mutex_exit(&mpt->m_mutex);
11431                               return (EBUSY);
11432                       }
11433                       mpt->m_diag_action_in_progress = 1;
11434                       status = mptsas_diag_action(mpt,
11435                           (mptsas_diag_action_t *)data, mode);
11436                       mpt->m_diag_action_in_progress = 0;
11437                       mutex_exit(&mpt->m_mutex);
11438                       break;
11439               case MPTIOCTL_EVENT_QUERY:
11440                       /*
11441                        * The user has done an event query. Call our routine
11442                        * which does this.
11443                        */
11444                       status = mptsas_event_query(mpt,
11445                           (mptsas_event_query_t *)data, mode, rval);
11446                       break;
11447               case MPTIOCTL_EVENT_ENABLE:
11448                       /*
11449                        * The user has done an event enable. Call our routine
11450                        * which does this.
11451                        */
11452                       status = mptsas_event_enable(mpt,
11453                           (mptsas_event_enable_t *)data, mode, rval);
11454                       break;
11455               case MPTIOCTL_EVENT_REPORT:
11456                       /*
11457                        * The user has done an event report. Call our routine
11458                        * which does this.
11459                        */
11460                       status = mptsas_event_report(mpt,
11461                           (mptsas_event_report_t *)data, mode, rval);
```

```
11462                           break;
11463                   case MPTIOCTL_REG_ACCESS:
11464                           /*
11465                            * The user has requested register access.  Call our
11466                            * routine which does this.
11467                            */
11468                           status = mptsas_reg_access(mpt,
11469                               (mptsas_reg_access_t *)data, mode);
11470                           break;
11471                   default:
11472                           status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
11473                               rval);
11474                           break;
11475           }

11477 out:
11478           return (status);
11479 }
______unchanged_portion_omitted_

13714 static int
13715 mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *inq, char *guid,
13716     dev_info_t **lun_dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt, int lun)
13717 {
13718           int                     target;
13719           char                    *nodename = NULL;
13720           char                    **compatible = NULL;
13721           int                     ncompatible     = 0;
13722           int                     mdi_rtn = MDI_FAILURE;
13723           int                     rval = DDI_FAILURE;
13724           char                    *old_guid = NULL;
13725           mptsas_t                *mpt = DIP2MPT(pdip);
13726           char                    *lun_addr = NULL;
13727           char                    *wwn_str = NULL;
13728           char                    *attached_wwn_str = NULL;
13729           char                    *component = NULL;
13730           uint8_t                 phy = 0xFF;
13731           uint64_t                sas_wwn;
13732           int64_t                 lun64 = 0;
13733           uint32_t                devinfo;
13734           uint16_t                dev_hdl;
13735           uint16_t                pdev_hdl;
13736           uint64_t                dev_sas_wwn;
13737           uint64_t                pdev_sas_wwn;
13738           uint32_t                pdev_info;
13739           uint8_t                 physport;
13740           uint8_t                 phy_id;
13741           uint32_t                page_address;
13742           uint16_t                bay_num, enclosure;
13743           char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
13744           uint32_t                dev_info;

13746           mutex_enter(&mpt->m_mutex);
13747           target = ptgt->m_devhdl;
13748           sas_wwn = ptgt->m_sas_wwn;
13749           devinfo = ptgt->m_deviceinfo;
13750           phy = ptgt->m_phynum;
13751           mutex_exit(&mpt->m_mutex);

13753           if (sas_wwn) {
13754                   *pip = mptsas_find_path_addr(pdip, sas_wwn, lun);
13755           } else {
13756                   *pip = mptsas_find_path_phy(pdip, phy);
13757           }

13759           if (*pip != NULL) {
```



```
13760                   *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
13761                   ASSERT(*lun_dip != NULL);
13762                   if (ddi_prop_lookup_string(DDI_DEV_T_ANY, *lun_dip,
13763                       (DDI_PROP_DONTPASS | DDI_PROP_NOTPROM),
13764                       MDI_CLIENT_GUID_PROP, &old_guid) == DDI_SUCCESS) {
13765                           if (strncmp(guid, old_guid, strlen(guid)) == 0) {
13766                                   /*
13767                                    * Same path back online again.
13768                                    */
13769                                   (void) ddi_prop_free(old_guid);
13770                                   if ((!MDI_PI_IS_ONLINE(*pip)) &&
13771                                       (!MDI_PI_IS_STANDBY(*pip)) &&
13772                                       (ptgt->m_tgt_unconfigured == 0)) {
13773                                           rval = mdi_pi_online(*pip, 0);
13840                                           mutex_enter(&mpt->m_mutex);
13841                                           (void) mptsas_set_led_status(mpt, ptgt,
13842                                               0);
13843                                           mutex_exit(&mpt->m_mutex);
13774                                   } else {
13775                                           rval = DDI_SUCCESS;
13776                                   }
13777                                   if (rval != DDI_SUCCESS) {
13778                                           mptsas_log(mpt, CE_WARN, "path:target: "
13779                                               "%x, lun:%x online failed!", target,
13780                                               lun);
13781                                           *pip = NULL;
13782                                           *lun_dip = NULL;
13783                                   }
13784                                   return (rval);
13785                           } else {
13786                                   /*
13787                                    * The GUID of the LUN has changed which maybe
13788                                    * because customer mapped another volume to the
13789                                    * same LUN.
13790                                    */
13791                                   mptsas_log(mpt, CE_WARN, "The GUID of the "
13792                                       "target:%x, lun:%x was changed, maybe "
13793                                       "because someone mapped another volume "
13794                                       "to the same LUN", target, lun);
13795                                   (void) ddi_prop_free(old_guid);
13796                                   if (!MDI_PI_IS_OFFLINE(*pip)) {
13797                                           rval = mdi_pi_offline(*pip, 0);
13798                                           if (rval != MDI_SUCCESS) {
13799                                                   mptsas_log(mpt, CE_WARN, "path:"
13800                                                       "target:%x, lun:%x offline "
13801                                                       "failed!", target, lun);
13802                                                   *pip = NULL;
13803                                                   *lun_dip = NULL;
13804                                                   return (DDI_FAILURE);
13805                                           }
13806                                   }
13807                                   if (mdi_pi_free(*pip, 0) != MDI_SUCCESS) {
13808                                           mptsas_log(mpt, CE_WARN, "path:target:"
13809                                               "%x, lun:%x free failed!", target,
13810                                               lun);
13811                                           *pip = NULL;
13812                                           *lun_dip = NULL;
13813                                           return (DDI_FAILURE);
13814                                   }
13815                           }
13816                   } else {
13817                           mptsas_log(mpt, CE_WARN, "Can't get client-guid "
13818                               "property for path:target:%x, lun:%x", target, lun);
13819                           *pip = NULL;
13820                           *lun_dip = NULL;
13821                           return (DDI_FAILURE);
```

```
13822                   }
13823           }
13824           scsi_hba_nodename_compatible_get(inq, NULL,
13825               inq->inq_dtype, NULL, &nodename, &compatible, &ncompatible);

13827           /*
13828            * if nodename can't be determined then print a message and skip it
13829            */
13830           if (nodename == NULL) {
13831                   mptsas_log(mpt, CE_WARN, "mptsas driver found no compatible "
13832                       "driver for target%d lun %d dtype:0x%02x", target, lun,
13833                       inq->inq_dtype);
13834                   return (DDI_FAILURE);
13835           }

13837           wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
13838           /* The property is needed by MPAPI */
13839           (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);

13841           lun_addr = kmem_zalloc(SCSI_MAXNAMELEN, KM_SLEEP);
13842           if (guid) {
13843                   (void) sprintf(lun_addr, "w%s,%x", wwn_str, lun);
13844                   (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
13845           } else {
13846                   (void) sprintf(lun_addr, "p%x,%x", phy, lun);
13847                   (void) sprintf(wwn_str, "p%x", phy);
13848           }

13850           mdi_rtn = mdi_pi_alloc_compatible(pdip, nodename,
13851               guid, lun_addr, compatible, ncompatible,
13852               0, pip);
13853           if (mdi_rtn == MDI_SUCCESS) {

13855                   if (mdi_prop_update_string(*pip, MDI_GUID,
13856                       guid) != DDI_SUCCESS) {
13857                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13858                               "create prop for target %d lun %d (MDI_GUID)",
13859                               target, lun);
13860                           mdi_rtn = MDI_FAILURE;
13861                           goto virt_create_done;
13862                   }

13864                   if (mdi_prop_update_int(*pip, LUN_PROP,
13865                       lun) != DDI_SUCCESS) {
13866                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13867                               "create prop for target %d lun %d (LUN_PROP)",
13868                               target, lun);
13869                           mdi_rtn = MDI_FAILURE;
13870                           goto virt_create_done;
13871                   }
13872                   lun64 = (int64_t)lun;
13873                   if (mdi_prop_update_int64(*pip, LUN64_PROP,
13874                       lun64) != DDI_SUCCESS) {
13875                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13876                               "create prop for target %d (LUN64_PROP)",
13877                               target);
13878                           mdi_rtn = MDI_FAILURE;
13879                           goto virt_create_done;
13880                   }
13881                   if (mdi_prop_update_string_array(*pip, "compatible",
13882                       compatible, ncompatible) !=
13883                       DDI_PROP_SUCCESS) {
13884                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13885                               "create prop for target %d lun %d (COMPATIBLE)",
13886                               target, lun);
13887                           mdi_rtn = MDI_FAILURE;
```



```
13888                           goto virt_create_done;
13889                   }
13890                   if (sas_wwn && (mdi_prop_update_string(*pip,
13891                       SCSI_ADDR_PROP_TARGET_PORT, wwn_str) != DDI_PROP_SUCCESS)) {
13892                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13893                               "create prop for target %d lun %d "
13894                               "(target-port)", target, lun);
13895                           mdi_rtn = MDI_FAILURE;
13896                           goto virt_create_done;
13897                   } else if ((sas_wwn == 0) && (mdi_prop_update_int(*pip,
13898                       "sata-phy", phy) != DDI_PROP_SUCCESS)) {
13899                           /*
13900                            * Direct attached SATA device without DeviceName
13901                            */
13902                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
13903                               "create prop for SAS target %d lun %d "
13904                               "(sata-phy)", target, lun);
13905                           mdi_rtn = MDI_FAILURE;
13906                           goto virt_create_done;
13907                   }
13908                   mutex_enter(&mpt->m_mutex);

13910                   page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
13911                       MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
13912                       (uint32_t)ptgt->m_devhdl;
13913                   rval = mptsas_get_sas_device_page0(mpt, page_address,
13914                       &dev_hdl, &dev_sas_wwn, &dev_info, &physport,
13915                       &phy_id, &pdev_hdl, &bay_num, &enclosure);
13916                   if (rval != DDI_SUCCESS) {
13917                           mutex_exit(&mpt->m_mutex);
13918                           mptsas_log(mpt, CE_WARN, "mptsas unable to get "
13919                               "parent device for handle %d", page_address);
13920                           mdi_rtn = MDI_FAILURE;
13921                           goto virt_create_done;
13922                   }

13924                   page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
13925                       MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
13926                   rval = mptsas_get_sas_device_page0(mpt, page_address,
13927                       &dev_hdl, &pdev_sas_wwn, &pdev_info, &physport,
13928                       &phy_id, &pdev_hdl, &bay_num, &enclosure);
13929                   if (rval != DDI_SUCCESS) {
13930                           mutex_exit(&mpt->m_mutex);
13931                           mptsas_log(mpt, CE_WARN, "mptsas unable to get"
13932                               "device info for handle %d", page_address);
13933                           mdi_rtn = MDI_FAILURE;
13934                           goto virt_create_done;
13935                   }

13937                   mutex_exit(&mpt->m_mutex);

13939                   /*
13940                    * If this device direct attached to the controller
13941                    * set the attached-port to the base wwid
13942                    */
13943                   if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
13944                       != DEVINFO_DIRECT_ATTACHED) {
13945                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
13946                               pdev_sas_wwn);
13947                   } else {
13948                           /*
13949                            * Update the iport's attached-port to guid
13950                            */
13951                           if (sas_wwn == 0) {
13952                                   (void) sprintf(wwn_str, "p%x", phy);
13953                           } else {
```

```
13954                                   (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
13955                           }
13956                           if (ddi_prop_update_string(DDI_DEV_T_NONE,
13957                               pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
13958                               DDI_PROP_SUCCESS) {
13959                                   mptsas_log(mpt, CE_WARN,
13960                                       "mptsas unable to create "
13961                                       "property for iport target-port"
13962                                       " %s (sas_wwn)",
13963                                       wwn_str);
13964                                   mdi_rtn = MDI_FAILURE;
13965                                   goto virt_create_done;
13966                           }

13968                           (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
13969                               mpt->un.m_base_wwid);
13970                   }

13972                   if (mdi_prop_update_string(*pip,
13973                       SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
13974                       DDI_PROP_SUCCESS) {
13975                           mptsas_log(mpt, CE_WARN, "mptsas unable to create "
13976                               "property for iport attached-port %s (sas_wwn)",
13977                               attached_wwn_str);
13978                           mdi_rtn = MDI_FAILURE;
13979                           goto virt_create_done;
13980                   }


13983                   if (inq->inq_dtype == 0) {
13984                           component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
13985                           /*
13986                            * set obp path for pathinfo
13987                            */
13988                           (void) snprintf(component, MAXPATHLEN,
13989                               "disk@%s", lun_addr);

13991                           if (mdi_pi_pathname_obp_set(*pip, component) !=
13992                               DDI_SUCCESS) {
13993                                   mptsas_log(mpt, CE_WARN, "mpt_sas driver "
13994                                       "unable to set obp-path for object %s",
13995                                       component);
13996                                   mdi_rtn = MDI_FAILURE;
13997                                   goto virt_create_done;
13998                           }
13999                   }

14001                   *lun_dip = MDI_PI(*pip)->pi_client->ct_dip;
14002                   if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14003                       MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14004                           if ((ndi_prop_update_int(DDI_DEV_T_NONE, *lun_dip,
14005                               "pm-capable", 1)) !=
14006                               DDI_PROP_SUCCESS) {
14007                                   mptsas_log(mpt, CE_WARN, "mptsas driver"
14008                                       "failed to create pm-capable "
14009                                       "property, target %d", target);
14010                                   mdi_rtn = MDI_FAILURE;
14011                                   goto virt_create_done;
14012                           }
14013                   }
14014                   /*
14015                    * Create the phy-num property
14016                    */
14017                   if (mdi_prop_update_int(*pip, "phy-num",
14018                       ptgt->m_phynum) != DDI_SUCCESS) {
14019                           mptsas_log(mpt, CE_WARN, "mptsas driver unable to "
```



```
14020                                   "create phy-num property for target %d lun %d",
14021                                   target, lun);
14022                           mdi_rtn = MDI_FAILURE;
14023                           goto virt_create_done;
14024                   }
14025                   NDBG20(("new path:%s onlining,", MDI_PI(*pip)->pi_addr));
14026                   mdi_rtn = mdi_pi_online(*pip, 0);
14097                   if (mdi_rtn == MDI_SUCCESS) {
14098                           mutex_enter(&mpt->m_mutex);
14099                           if (mptsas_set_led_status(mpt, ptgt, 0) !=
14100                               DDI_SUCCESS) {
14101                                   NDBG14(("mptsas: clear LED for slot %x "
14102                                       "failed", ptgt->m_slot_num));
14103                           }
14104                           mutex_exit(&mpt->m_mutex);
14105                   }
14027                   if (mdi_rtn == MDI_NOT_SUPPORTED) {
14028                           mdi_rtn = MDI_FAILURE;
14029                   }
14030 virt_create_done:
14031                   if (*pip && mdi_rtn != MDI_SUCCESS) {
14032                           (void) mdi_pi_free(*pip, 0);
14033                           *pip = NULL;
14034                           *lun_dip = NULL;
14035                   }
14036           }

14038           scsi_hba_nodename_compatible_free(nodename, compatible);
14039           if (lun_addr != NULL) {
14040                   kmem_free(lun_addr, SCSI_MAXNAMELEN);
14041           }
14042           if (wwn_str != NULL) {
14043                   kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14044           }
14045           if (component != NULL) {
14046                   kmem_free(component, MAXPATHLEN);
14047           }

14049           return ((mdi_rtn == MDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14050 }

14052 static int
14053 mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *inq,
14054     char *guid, dev_info_t **lun_dip, mptsas_target_t *ptgt, int lun)
14055 {
14056           int                     target;
14057           int                     rval;
14058           int                     ndi_rtn = NDI_FAILURE;
14059           uint64_t                be_sas_wwn;
14060           char                    *nodename = NULL;
14061           char                    **compatible = NULL;
14062           int                     ncompatible = 0;
14063           int                     instance = 0;
14064           mptsas_t                *mpt = DIP2MPT(pdip);
14065           char                    *wwn_str = NULL;
14066           char                    *component = NULL;
14067           char                    *attached_wwn_str = NULL;
14068           uint8_t                 phy = 0xFF;
14069           uint64_t                sas_wwn;
14070           uint32_t                devinfo;
14071           uint16_t                dev_hdl;
14072           uint16_t                pdev_hdl;
14073           uint64_t                pdev_sas_wwn;
14074           uint64_t                dev_sas_wwn;
14075           uint32_t                pdev_info;
14076           uint8_t                 physport;
```

```
14077         uint8_t                 phy_id;
14078         uint32_t                page_address;
14079         uint16_t                bay_num, enclosure;
14080         char                    pdev_wwn_str[MPTSAS_WWN_STRLEN];
14081         uint32_t                dev_info;
14082         int64_t                 lun64 = 0;

14084         mutex_enter(&mpt->m_mutex);
14085         target = ptgt->m_devhdl;
14086         sas_wwn = ptgt->m_sas_wwn;
14087         devinfo = ptgt->m_deviceinfo;
14088         phy = ptgt->m_phynum;
14089         mutex_exit(&mpt->m_mutex);

14091         /*
14092          * generate compatible property with binding-set "mpt"
14093          */
14094         scsi_hba_nodename_compatible_get(inq, NULL, inq->inq_dtype, NULL,
14095             &nodename, &compatible, &ncompatible);

14097         /*
14098          * if nodename can't be determined then print a message and skip it
14099          */
14100         if (nodename == NULL) {
14101                 mptsas_log(mpt, CE_WARN, "mptsas found no compatible driver "
14102                     "for target %d lun %d", target, lun);
14103                 return (DDI_FAILURE);
14104         }

14106         ndi_rtn = ndi_devi_alloc(pdip, nodename,
14107             DEVI_SID_NODEID, lun_dip);

14109         /*
14110          * if lun alloc success, set props
14111          */
14112         if (ndi_rtn == NDI_SUCCESS) {

14114                 if (ndi_prop_update_int(DDI_DEV_T_NONE,
14115                     *lun_dip, LUN_PROP, lun) !=
14116                     DDI_PROP_SUCCESS) {
14117                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14118                             "property for target %d lun %d (LUN_PROP)",
14119                             target, lun);
14120                         ndi_rtn = NDI_FAILURE;
14121                         goto phys_create_done;
14122                 }

14124                 lun64 = (int64_t)lun;
14125                 if (ndi_prop_update_int64(DDI_DEV_T_NONE,
14126                     *lun_dip, LUN64_PROP, lun64) !=
14127                     DDI_PROP_SUCCESS) {
14128                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14129                             "property for target %d lun64 %d (LUN64_PROP)",
14130                             target, lun);
14131                         ndi_rtn = NDI_FAILURE;
14132                         goto phys_create_done;
14133                 }
14134                 if (ndi_prop_update_string_array(DDI_DEV_T_NONE,
14135                     *lun_dip, "compatible", compatible, ncompatible)
14136                     != DDI_PROP_SUCCESS) {
14137                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14138                             "property for target %d lun %d (COMPATIBLE)",
14139                             target, lun);
14140                         ndi_rtn = NDI_FAILURE;
14141                         goto phys_create_done;
14142                 }
```



```
14144                 /*
14145                  * We need the SAS WWN for non-multipath devices, so
14146                  * we'll use the same property as that multipathing
14147                  * devices need to present for MPAPI. If we don't have
14148                  * a WWN (e.g. parallel SCSI), don't create the prop.
14149                  */
14150                 wwn_str = kmem_zalloc(MPTSAS_WWN_STRLEN, KM_SLEEP);
14151                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14152                 if (sas_wwn && ndi_prop_update_string(DDI_DEV_T_NONE,
14153                     *lun_dip, SCSI_ADDR_PROP_TARGET_PORT, wwn_str)
14154                     != DDI_PROP_SUCCESS) {
14155                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14156                             "create property for SAS target %d lun %d "
14157                             "(target-port)", target, lun);
14158                         ndi_rtn = NDI_FAILURE;
14159                         goto phys_create_done;
14160                 }

14162                 be_sas_wwn = BE_64(sas_wwn);
14163                 if (sas_wwn && ndi_prop_update_byte_array(
14164                     DDI_DEV_T_NONE, *lun_dip, "port-wwn",
14165                     (uchar_t *)&be_sas_wwn, 8) != DDI_PROP_SUCCESS) {
14166                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14167                             "create property for SAS target %d lun %d "
14168                             "(port-wwn)", target, lun);
14169                         ndi_rtn = NDI_FAILURE;
14170                         goto phys_create_done;
14171                 } else if ((sas_wwn == 0) && (ndi_prop_update_int(
14172                     DDI_DEV_T_NONE, *lun_dip, "sata-phy", phy) !=
14173                     DDI_PROP_SUCCESS)) {
14174                         /*
14175                          * Direct attached SATA device without DeviceName
14176                          */
14177                         mptsas_log(mpt, CE_WARN, "mptsas unable to "
14178                             "create property for SAS target %d lun %d "
14179                             "(sata-phy)", target, lun);
14180                         ndi_rtn = NDI_FAILURE;
14181                         goto phys_create_done;
14182                 }

14184                 if (ndi_prop_create_boolean(DDI_DEV_T_NONE,
14185                     *lun_dip, SAS_PROP) != DDI_PROP_SUCCESS) {
14186                         mptsas_log(mpt, CE_WARN, "mptsas unable to"
14187                             "create property for SAS target %d lun %d"
14188                             " (SAS_PROP)", target, lun);
14189                         ndi_rtn = NDI_FAILURE;
14190                         goto phys_create_done;
14191                 }
14192                 if (guid && (ndi_prop_update_string(DDI_DEV_T_NONE,
14193                     *lun_dip, NDI_GUID, guid) != DDI_SUCCESS)) {
14194                         mptsas_log(mpt, CE_WARN, "mptsas unable "
14195                             "to create guid property for target %d "
14196                             "lun %d", target, lun);
14197                         ndi_rtn = NDI_FAILURE;
14198                         goto phys_create_done;
14199                 }

14201                 /*
14202                  * The following code is to set properties for SM-HBA support,
14203                  * it doesn't apply to RAID volumes
14204                  */
14205                 if (ptgt->m_phymask == 0)
14206                         goto phys_raid_lun;

14208                 mutex_enter(&mpt->m_mutex);
```

```
14210                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14211                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) |
14212                     (uint32_t)ptgt->m_devhdl;
14213                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14214                     &dev_hdl, &dev_sas_wwn, &dev_info,
14215                     &physport, &phy_id, &pdev_hdl,
14216                     &bay_num, &enclosure);
14217                 if (rval != DDI_SUCCESS) {
14218                         mutex_exit(&mpt->m_mutex);
14219                         mptsas_log(mpt, CE_WARN, "mptsas unable to get"
14220                             "parent device for handle %d.", page_address);
14221                         ndi_rtn = NDI_FAILURE;
14222                         goto phys_create_done;
14223                 }

14225                 page_address = (MPI2_SAS_DEVICE_PGAD_FORM_HANDLE &
14226                     MPI2_SAS_DEVICE_PGAD_FORM_MASK) | (uint32_t)pdev_hdl;
14227                 rval = mptsas_get_sas_device_page0(mpt, page_address,
14228                     &dev_hdl, &pdev_sas_wwn, &pdev_info,
14229                     &physport, &phy_id, &pdev_hdl, &bay_num, &enclosure);
14230                 if (rval != DDI_SUCCESS) {
14231                         mutex_exit(&mpt->m_mutex);
14232                         mptsas_log(mpt, CE_WARN, "mptsas unable to create "
14233                             "device for handle %d.", page_address);
14234                         ndi_rtn = NDI_FAILURE;
14235                         goto phys_create_done;
14236                 }

14238                 mutex_exit(&mpt->m_mutex);

14240                 /*
14241                  * If this device direct attached to the controller
14242                  * set the attached-port to the base wwid
14243                  */
14244                 if ((ptgt->m_deviceinfo & DEVINFO_DIRECT_ATTACHED)
14245                     != DEVINFO_DIRECT_ATTACHED) {
14246                         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14247                             pdev_sas_wwn);
14248                 } else {
14249                         /*
14250                          * Update the iport's attached-port to guid
14251                          */
14252                         if (sas_wwn == 0) {
14253                                 (void) sprintf(wwn_str, "p%x", phy);
14254                         } else {
14255                                 (void) sprintf(wwn_str, "w%016"PRIx64, sas_wwn);
14256                         }
14257                         if (ddi_prop_update_string(DDI_DEV_T_NONE,
14258                             pdip, SCSI_ADDR_PROP_ATTACHED_PORT, wwn_str) !=
14259                             DDI_PROP_SUCCESS) {
14260                                 mptsas_log(mpt, CE_WARN,
14261                                     "mptsas unable to create "
14262                                     "property for iport target-port"
14263                                     " %s (sas_wwn)",
14264                                     wwn_str);
14265                                 ndi_rtn = NDI_FAILURE;
14266                                 goto phys_create_done;
14267                         }

14269                         (void) sprintf(pdev_wwn_str, "w%016"PRIx64,
14270                             mpt->un.m_base_wwid);
14271                 }

14273                 if (ndi_prop_update_string(DDI_DEV_T_NONE,
14274                     *lun_dip, SCSI_ADDR_PROP_ATTACHED_PORT, pdev_wwn_str) !=
```



```
14275                         DDI_PROP_SUCCESS) {
14276                             mptsas_log(mpt, CE_WARN,
14277                                 "mptsas unable to create "
14278                                 "property for iport attached-port %s (sas_wwn)",
14279                                 attached_wwn_str);
14280                             ndi_rtn = NDI_FAILURE;
14281                             goto phys_create_done;
14282                 }

14284                 if (IS_SATA_DEVICE(dev_info)) {
14285                         if (ndi_prop_update_string(DDI_DEV_T_NONE,
14286                             *lun_dip, MPTSAS_VARIANT, "sata") !=
14287                             DDI_PROP_SUCCESS) {
14288                                 mptsas_log(mpt, CE_WARN,
14289                                     "mptsas unable to create "
14290                                     "property for device variant ");
14291                                 ndi_rtn = NDI_FAILURE;
14292                                 goto phys_create_done;
14293                         }
14294                 }

14296                 if (IS_ATAPI_DEVICE(dev_info)) {
14297                         if (ndi_prop_update_string(DDI_DEV_T_NONE,
14298                             *lun_dip, MPTSAS_VARIANT, "atapi") !=
14299                             DDI_PROP_SUCCESS) {
14300                                 mptsas_log(mpt, CE_WARN,
14301                                     "mptsas unable to create "
14302                                     "property for device variant ");
14303                                 ndi_rtn = NDI_FAILURE;
14304                                 goto phys_create_done;
14305                         }
14306                 }

14308 phys_raid_lun:
14309                 /*
14310                  * if this is a SAS controller, and the target is a SATA
14311                  * drive, set the 'pm-capable' property for sd and if on
14312                  * an OPL platform, also check if this is an ATAPI
14313                  * device.
14314                  */
14315                 instance = ddi_get_instance(mpt->m_dip);
14316                 if (devinfo & (MPI2_SAS_DEVICE_INFO_SATA_DEVICE |
14317                     MPI2_SAS_DEVICE_INFO_ATAPI_DEVICE)) {
14318                         NDBG2(("mptsas%d: creating pm-capable property, "
14319                             "target %d", instance, target));

14321                         if ((ndi_prop_update_int(DDI_DEV_T_NONE,
14322                             *lun_dip, "pm-capable", 1)) !=
14323                             DDI_PROP_SUCCESS) {
14324                                 mptsas_log(mpt, CE_WARN, "mptsas "
14325                                     "failed to create pm-capable "
14326                                     "property, target %d", target);
14327                                 ndi_rtn = NDI_FAILURE;
14328                                 goto phys_create_done;
14329                         }

14331                 }

14333                 if ((inq->inq_dtype == 0) || (inq->inq_dtype == 5)) {
14334                         /*
14335                          * add 'obp-path' properties for devinfo
14336                          */
14337                         bzero(wwn_str, sizeof (wwn_str));
14338                         (void) sprintf(wwn_str, "%016"PRIx64, sas_wwn);
14339                         component = kmem_zalloc(MAXPATHLEN, KM_SLEEP);
14340                         if (guid) {
```

```
14341                                   (void) snprintf(component, MAXPATHLEN,
14342                                       "disk@w%s,%x", wwn_str, lun);
14343                           } else {
14344                                   (void) snprintf(component, MAXPATHLEN,
14345                                       "disk@p%x,%x", phy, lun);
14346                           }
14347                           if (ddi_pathname_obp_set(*lun_dip, component)
14348                               != DDI_SUCCESS) {
14349                                   mptsas_log(mpt, CE_WARN, "mpt_sas driver "
14350                                       "unable to set obp-path for SAS "
14351                                       "object %s", component);
14352                                   ndi_rtn = NDI_FAILURE;
14353                                   goto phys_create_done;
14354                           }
14355                   }
14356                   /*
14357                    * Create the phy-num property for non-raid disk
14358                    */
14359                   if (ptgt->m_phymask != 0) {
14360                           if (ndi_prop_update_int(DDI_DEV_T_NONE,
14361                               *lun_dip, "phy-num", ptgt->m_phynum) !=
14362                               DDI_PROP_SUCCESS) {
14363                                   mptsas_log(mpt, CE_WARN, "mptsas driver "
14364                                       "failed to create phy-num property for "
14365                                       "target %d", target);
14366                                   ndi_rtn = NDI_FAILURE;
14367                                   goto phys_create_done;
14368                           }
14369                   }
14370 phys_create_done:
14371                   /*
14372                    * If props were setup ok, online the lun
14373                    */
14374                   if (ndi_rtn == NDI_SUCCESS) {
14375                           /*
14376                            * Try to online the new node
14377                            */
14378                           ndi_rtn = ndi_devi_online(*lun_dip, NDI_ONLINE_ATTACH);
14379                   }
14459                   if (ndi_rtn == NDI_SUCCESS) {
14460                           mutex_enter(&mpt->m_mutex);
14461                           if (mptsas_set_led_status(mpt, ptgt, 0) !=
14462                               DDI_SUCCESS) {
14463                                   NDBG14(("mptsas: clear LED for tgt %x "
14464                                       "failed", ptgt->m_slot_num));
14465                           }
14466                           mutex_exit(&mpt->m_mutex);
14467                   }

14381                   /*
14382                    * If success set rtn flag, else unwire alloc'd lun
14383                    */
14384                   if (ndi_rtn != NDI_SUCCESS) {
14385                           NDBG12(("mptsas driver unable to online "
14386                               "target %d lun %d", target, lun));
14387                           ndi_prop_remove_all(*lun_dip);
14388                           (void) ndi_devi_free(*lun_dip);
14389                           *lun_dip = NULL;
14390                   }
14391           }

14393           scsi_hba_nodename_compatible_free(nodename, compatible);

14395           if (wwn_str != NULL) {
14396                   kmem_free(wwn_str, MPTSAS_WWN_STRLEN);
14397           }
```



```
14398           if (component != NULL) {
14399                   kmem_free(component, MAXPATHLEN);
14400           }


14403           return ((ndi_rtn == NDI_SUCCESS) ? DDI_SUCCESS : DDI_FAILURE);
14404 }
______unchanged_portion_omitted_

15360 #ifdef MPTSAS_GET_LED
15361 static int
15362 mptsas_get_led_status(mptsas_t *mpt, mptsas_target_t *ptgt,
15363     uint32_t *slotstatus)
15364 {
15365           return (mptsas_send_sep(mpt, ptgt, slotstatus,
15366               MPI2_SEP_REQ_ACTION_READ_STATUS));
15367 }
15368 #endif
15369 static int
15370 mptsas_set_led_status(mptsas_t *mpt, mptsas_target_t *ptgt, uint32_t slotstatus)
15371 {
15372           NDBG14(("mptsas_ioctl: set LED status %x for slot %x",
15373               slotstatus, ptgt->m_slot_num));
15374           return (mptsas_send_sep(mpt, ptgt, &slotstatus,
15375               MPI2_SEP_REQ_ACTION_WRITE_STATUS));
15376 }
15377 /*
15378  *  send sep request, use enclosure/slot addressing
15379  */
15380 static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
15381     uint32_t *status, uint8_t act)
15382 {
15383           Mpi2SepRequest_t        req;
15384           Mpi2SepReply_t          rep;
15385           int                     ret;

15387           ASSERT(mutex_owned(&mpt->m_mutex));

15389           bzero(&req, sizeof (req));
15390           bzero(&rep, sizeof (rep));

15392           /* Do nothing for RAID volumes */
15393           if (ptgt->m_phymask == 0) {
15394                   NDBG14(("mptsas_send_sep: Skip RAID volumes"));
15395                   return (DDI_FAILURE);
15396           }

15398           req.Function = MPI2_FUNCTION_SCSI_ENCLOSURE_PROCESSOR;
15399           req.Action = act;
15400           req.Flags = MPI2_SEP_REQ_FLAGS_ENCLOSURE_SLOT_ADDRESS;
15401           req.EnclosureHandle = LE_16(ptgt->m_enclosure);
15402           req.Slot = LE_16(ptgt->m_slot_num);
15403           if (act == MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
15404                   req.SlotStatus = LE_32(*status);
15405           }
15406           ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep, NULL,
15407               sizeof (req), sizeof (rep), NULL, 0, NULL, 0, 60, FKIOCTL);
15408           if (ret != 0) {
15409                   mptsas_log(mpt, CE_NOTE, "mptsas_send_sep: passthru SEP "
15410                       "Processor Request message error %d", ret);
15411                   return (DDI_FAILURE);
15412           }
15413           /* do passthrough success, check the ioc status */
15414           if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
15415                   if ((LE_16(rep.IOCStatus) & MPI2_IOCSTATUS_MASK) ==
15416                       MPI2_IOCSTATUS_INVALID_FIELD) {
```

```
15417                           mptsas_log(mpt, CE_NOTE, "send sep act %x: Not "
15418                               "supported action, loginfo %x", act,
15419                               LE_32(rep.IOCLogInfo));
15420                           return (DDI_FAILURE);
15421                   }
15422                   mptsas_log(mpt, CE_NOTE, "send_sep act %x: ioc "
15423                       "status:%x", act, LE_16(rep.IOCStatus));
15424                   return (DDI_FAILURE);
15425           }
15426           if (act != MPI2_SEP_REQ_ACTION_WRITE_STATUS) {
15427                   *status = LE_32(rep.SlotStatus);
15428           }

15430           return (DDI_SUCCESS);
15431 }

15272 int
15273 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
15274     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
15275     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
15276 {
15277           ddi_dma_cookie_t        new_cookie;
15278           size_t                  alloc_len;
15279           uint_t                  ncookie;

15281           if (cookiep == NULL)
15282                   cookiep = &new_cookie;

15284           if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
15285               NULL, dma_hdp) != DDI_SUCCESS) {
15286                   dma_hdp = NULL;
15287                   return (FALSE);
15288           }

15290           if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
15291               DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
15292               acc_hdp) != DDI_SUCCESS) {
15293                   ddi_dma_free_handle(dma_hdp);
15294                   dma_hdp = NULL;
15295                   return (FALSE);
15296           }

15298           if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
15299               (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
15300               cookiep, &ncookie) != DDI_DMA_MAPPED) {
15301                   (void) ddi_dma_mem_free(acc_hdp);
15302                   ddi_dma_free_handle(dma_hdp);
15303                   dma_hdp = NULL;
15304                   return (FALSE);
15305           }

15307           return (TRUE);
15308 }
```

_____unchanged_portion_omitted_