```
**********************************************************
  413826 Tue Dec  4 16:30:45 2012
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
re #8346 rb2639 KT disk failures (fix lint/cstyle)
**********************************************************
```

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

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

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

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

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



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

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

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

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

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

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

/*
 * cb_ops function
 */
static int mptsas_ioctl(dev_t dev, int cmd, intptr_t data, int mode,
        cred_t *credp, int *rval);
#ifdef __sparc
static int mptsas_reset(dev_info_t *devi, ddi_reset_cmd_t cmd);
#else  /* __sparc */
static int mptsas_quiesce(dev_info_t *devi);
#endif  /* __sparc */

/*
```

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

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

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

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

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



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

 198 static void mptsas_ncmds_checkdrain(void *arg);

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

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

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

 218 static void mptsas_restart_cmd(void *);

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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



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

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

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

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

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

351 static dev_info_t *mptsas_get_dip_from_dev(dev_t dev,
352     mptsas_phymask_t *phymask);
353 static mptsas_target_t *mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr,
354     mptsas_phymask_t phymask);


355 /*
356  * Enumeration / DR functions
357  */
358 static void mptsas_config_all(dev_info_t *pdip);
359 static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
360     dev_info_t **lundip);
361 static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
362     dev_info_t **lundip);

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

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

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

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

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

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

388 static int mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn,
389     dev_info_t **smp_dip);
```

```
390 static int mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
391     uint_t flags);

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

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

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

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

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

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

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

449 extern pri_t minclsyspri, maxclsyspri;

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



```
456 extern dev_info_t       *scsi_vhci_dip;

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

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

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

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

8522         /*
8523          * Make sure the I/O Controller has flushed all cmds
8524          * that are associated with this target for a target reset
8525          * and target/lun for abort task set.
8526          * Account for TM requests, which use the last SMID.
8527          */
8528         for (slot = 0; slot <= mpt->m_active->m_n_slots; slot++) {
8529                 if ((cmd = slots->m_slot[slot]) == NULL)
8530                         continue;
8531                 reason = CMD_RESET;
8532                 stat = STAT_DEV_RESET;
8533                 switch (tasktype) {
```

```
8534                 case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8535                         if (Tgt(cmd) == target) {
8536                                 if (cmd->cmd_tgt_addr->m_timeout < 0) {
8537                                         /*
8538                                          * When timeout requested, propagate
8539                                          * proper reason and statistics to
8540                                          * target drivers.
8541                                          */
8542                                         reason = CMD_TIMEOUT;
8543                                         stat |= STAT_TIMEOUT;
8544                                 }
8547
8545                                 NDBG25(("mptsas_flush_target discovered non-"
8546                                     "NULL cmd in slot %d, tasktype 0x%x", slot,
8547                                     tasktype));
8548                                 mptsas_dump_cmd(mpt, cmd);
8549                                 mptsas_remove_cmd(mpt, cmd);
8550                                 mptsas_set_pkt_reason(mpt, cmd, reason, stat);
8551                                 mptsas_doneq_add(mpt, cmd);
8552                         }
8553                         break;
8554                 case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8555                         reason = CMD_ABORTED;
8556                         stat = STAT_ABORTED;
8557                         /*FALLTHROUGH*/
8558                 case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8559                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {

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

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

8581         reason = CMD_RESET;
8582         stat = STAT_DEV_RESET;

8584         switch (tasktype) {
8585         case MPI2_SCSITASKMGMT_TASKTYPE_TARGET_RESET:
8586                 while (cmd != NULL) {
8587                         next_cmd = cmd->cmd_linkp;
8588                         if (Tgt(cmd) == target) {
8589                                 mptsas_waitq_delete(mpt, cmd);
8590                                 mptsas_set_pkt_reason(mpt, cmd,
8591                                     reason, stat);
8592                                 mptsas_doneq_add(mpt, cmd);
8593                         }
8594                         cmd = next_cmd;
8595                 }
8596                 mutex_enter(&mpt->m_tx_waitq_mutex);
8597                 cmd = mpt->m_tx_waitq;
8598                 while (cmd != NULL) {
```



```
8599                         next_cmd = cmd->cmd_linkp;
8600                         if (Tgt(cmd) == target) {
8601                                 mptsas_tx_waitq_delete(mpt, cmd);
8602                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8603                                 mptsas_set_pkt_reason(mpt, cmd,
8604                                     reason, stat);
8605                                 mptsas_doneq_add(mpt, cmd);
8606                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8607                         }
8608                         cmd = next_cmd;
8609                 }
8610                 mutex_exit(&mpt->m_tx_waitq_mutex);
8611                 break;
8612         case MPI2_SCSITASKMGMT_TASKTYPE_ABRT_TASK_SET:
8613                 reason = CMD_ABORTED;
8614                 stat =  STAT_ABORTED;
8615                 /*FALLTHROUGH*/
8616         case MPI2_SCSITASKMGMT_TASKTYPE_LOGICAL_UNIT_RESET:
8617                 while (cmd != NULL) {
8618                         next_cmd = cmd->cmd_linkp;
8619                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8620                                 mptsas_waitq_delete(mpt, cmd);
8621                                 mptsas_set_pkt_reason(mpt, cmd,
8622                                     reason, stat);
8623                                 mptsas_doneq_add(mpt, cmd);
8624                         }
8625                         cmd = next_cmd;
8626                 }
8627                 mutex_enter(&mpt->m_tx_waitq_mutex);
8628                 cmd = mpt->m_tx_waitq;
8629                 while (cmd != NULL) {
8630                         next_cmd = cmd->cmd_linkp;
8631                         if ((Tgt(cmd) == target) && (Lun(cmd) == lun)) {
8632                                 mptsas_tx_waitq_delete(mpt, cmd);
8633                                 mutex_exit(&mpt->m_tx_waitq_mutex);
8634                                 mptsas_set_pkt_reason(mpt, cmd,
8635                                     reason, stat);
8636                                 mptsas_doneq_add(mpt, cmd);
8637                                 mutex_enter(&mpt->m_tx_waitq_mutex);
8638                         }
8639                         cmd = next_cmd;
8640                 }
8641                 mutex_exit(&mpt->m_tx_waitq_mutex);
8642                 break;
8643         default:
8644                 mptsas_log(mpt, CE_WARN, "Unknown task management type %d.",
8645                     tasktype);
8646                 break;
8647         }
8648 }
```

_____unchanged_portion_omitted_

```
9339 static void
9340 mptsas_watchsubr(mptsas_t *mpt)
9341 {
9342         int             i;
9343         mptsas_cmd_t    *cmd;
9344         mptsas_target_t *ptgt = NULL;

9346         NDBG30(("mptsas_watchsubr: mpt=0x%p", (void *)mpt));

9348 #ifdef MPTSAS_TEST
9349         if (mptsas_enable_untagged) {
9350                 mptsas_test_untagged++;
9351         }
9352 #endif
```

```
9354         /*
9355          * Check for commands stuck in active slot
9356          * Account for TM requests, which use the last SMID.
9357          */
9358         for (i = 0; i <= mpt->m_active->m_n_slots; i++) {
9359                 if ((cmd = mpt->m_active->m_slot[i]) != NULL) {
9360                         if ((cmd->cmd_flags & CFLAG_CMDIOC) == 0) {
9361                                 cmd->cmd_active_timeout -=
9362                                     mptsas_scsi_watchdog_tick;
9363                                 if (cmd->cmd_active_timeout <= 0) {
9364                                         /*
9365                                          * There seems to be a command stuck
9366                                          * in the active slot.  Drain throttle.
9367                                          */
9368                                         mptsas_set_throttle(mpt,
9369                                             cmd->cmd_tgt_addr,
9370                                             DRAIN_THROTTLE);
9371                                 }
9372                         }
9373                         if ((cmd->cmd_flags & CFLAG_PASSTHRU) ||
9374                             (cmd->cmd_flags & CFLAG_CONFIG) ||
9375                             (cmd->cmd_flags & CFLAG_FW_DIAG)) {
9376                                 cmd->cmd_active_timeout -=
9377                                     mptsas_scsi_watchdog_tick;
9378                                 if (cmd->cmd_active_timeout <= 0) {
9379                                         /*
9380                                          * passthrough command timeout
9381                                          */
9382                                         cmd->cmd_flags |= (CFLAG_FINISHED |
9383                                             CFLAG_TIMEOUT);
9384                                         cv_broadcast(&mpt->m_passthru_cv);
9385                                         cv_broadcast(&mpt->m_config_cv);
9386                                         cv_broadcast(&mpt->m_fw_diag_cv);
9387                                 }
9388                         }
9389                 }
9390         }

9392         ptgt = (mptsas_target_t *)mptsas_hash_traverse(&mpt->m_active->m_tgttbl,
9393             MPTSAS_HASH_FIRST);
9394         while (ptgt != NULL) {
9395                 /*
9396                  * If we were draining due to a qfull condition,
9397                  * go back to full throttle.
9398                  */
9399                 if ((ptgt->m_t_throttle < MAX_THROTTLE) &&
9400                     (ptgt->m_t_throttle > HOLD_THROTTLE) &&
9401                     (ptgt->m_t_ncmds < ptgt->m_t_throttle)) {
9402                         mptsas_set_throttle(mpt, ptgt, MAX_THROTTLE);
9403                         mptsas_restart_hba(mpt);
9404                 }

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

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

9418                         ptgt->m_timeout -= mptsas_scsi_watchdog_tick;
```



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

9430                         if (ptgt->m_timeout < 0) {
9431                                 ptgt->m_timeout_count++;
9432                                 if (ptgt->m_timeout_count >
9433                                     mptsas_timeout_threshold) {
9434                                         ptgt->m_timeout_count = 0;
9435                                         mptsas_kill_target(mpt, ptgt);
9436                                 } else {
9437                                         mptsas_cmd_timeout(mpt, ptgt->m_devhdl);
9438                                 }
9439                                 ptgt = (mptsas_target_t *)mptsas_hash_traverse(
9440                                     &mpt->m_active->m_tgttbl, MPTSAS_HASH_NEXT);
9441                                 continue;
9442                         }

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

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

_____unchanged_portion_omitted_

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

11365         int                     iport_flag = 0;
11366         dev_info_t              *dip = NULL;
11367         mptsas_phymask_t        phymask = 0;
11370         struct devctl_iocdata   *dcp = NULL;
11371         uint32_t                slotstatus = 0;
11372         char                    *addr = NULL;
11373         mptsas_target_t         *ptgt = NULL;

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

11374         mpt = ddi_get_soft_state(mptsas_state, MINOR2INST(getminor(dev)));
```

```
11375         if (mpt == NULL) {
11376                 /*
11377                  * Called from iport node, get the states
11378                  */
11379                 iport_flag = 1;
11380                 dip = mptsas_get_dip_from_dev(dev, &phymask);
11381                 if (dip == NULL) {
11382                         return (ENXIO);
11383                 }
11384                 mpt = DIP2MPT(dip);
11385         }
11386         /* Make sure power level is D0 before accessing registers */
11387         mutex_enter(&mpt->m_mutex);
11388         if (mpt->m_options & MPTSAS_OPT_PM) {
11389                 (void) pm_busy_component(mpt->m_dip, 0);
11390                 if (mpt->m_power_level != PM_LEVEL_D0) {
11391                         mutex_exit(&mpt->m_mutex);
11392                         if (pm_raise_power(mpt->m_dip, 0, PM_LEVEL_D0) !=
11393                             DDI_SUCCESS) {
11394                                 mptsas_log(mpt, CE_WARN,
11395                                     "mptsas%d: mptsas_ioctl: Raise power "
11396                                     "request failed.", mpt->m_instance);
11397                                 (void) pm_idle_component(mpt->m_dip, 0);
11398                                 return (ENXIO);
11399                         }
11400                 } else {
11401                         mutex_exit(&mpt->m_mutex);
11402                 }
11403         } else {
11404                 mutex_exit(&mpt->m_mutex);
11405         }

11407         if (iport_flag) {
11408                 status = scsi_hba_ioctl(dev, cmd, data, mode, credp, rval);
11409                 goto out;
11410         }
11411         switch (cmd) {
11412                 case MPTIOCTL_UPDATE_FLASH:
11413                         if (ddi_copyin((void *)data, &flashdata,
11414                                 sizeof (struct mptsas_update_flash), mode)) {
11415                                 status = EFAULT;
11416                                 break;
11417                         }

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

11426                         /*
11427                          * Reset the chip to start using the new
11428                          * firmware.  Reset if failed also.
11429                          */
11430                         mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
11431                         if (mptsas_restart_ioc(mpt) == DDI_FAILURE) {
11432                                 status = EFAULT;
11433                         }
11434                         mutex_exit(&mpt->m_mutex);
11435                         break;
11436                 case MPTIOCTL_PASS_THRU:
11437                         /*
11438                          * The user has requested to pass through a command to
11439                          * be executed by the MPT firmware.  Call our routine
11440                          * which does this.  Only allow one passthru IOCTL at
```



```
11441                          * one time. Other threads will block on
11442                          * m_passthru_mutex, which is of adaptive variant.
11443                          */
11444                         if (ddi_copyin((void *)data, &passthru_data,
11445                             sizeof (mptsas_pass_thru_t), mode)) {
11446                                 status = EFAULT;
11447                                 break;
11448                         }
11449                         mutex_enter(&mpt->m_passthru_mutex);
11450                         mutex_enter(&mpt->m_mutex);
11451                         status = mptsas_pass_thru(mpt, &passthru_data, mode);
11452                         mutex_exit(&mpt->m_mutex);
11453                         mutex_exit(&mpt->m_passthru_mutex);

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

```
11507                         }
11508                         mutex_exit(&mpt->m_mutex);
11509                         break;
11510                 case MPTIOCTL_DIAG_ACTION:
11511                         /*
11512                          * The user has done a diag buffer action.  Call our
11513                          * routine which does this.  Only allow one diag action
11514                          * at one time.
11515                          */
11516                         mutex_enter(&mpt->m_mutex);
11517                         if (mpt->m_diag_action_in_progress) {
11518                                 mutex_exit(&mpt->m_mutex);
11519                                 return (EBUSY);
11520                         }
11521                         mpt->m_diag_action_in_progress = 1;
11522                         status = mptsas_diag_action(mpt,
11523                             (mptsas_diag_action_t *)data, mode);
11524                         mpt->m_diag_action_in_progress = 0;
11525                         mutex_exit(&mpt->m_mutex);
11526                         break;
11527                 case MPTIOCTL_EVENT_QUERY:
11528                         /*
11529                          * The user has done an event query. Call our routine
11530                          * which does this.
11531                          */
11532                         status = mptsas_event_query(mpt,
11533                             (mptsas_event_query_t *)data, mode, rval);
11534                         break;
11535                 case MPTIOCTL_EVENT_ENABLE:
11536                         /*
11537                          * The user has done an event enable. Call our routine
11538                          * which does this.
11539                          */
11540                         status = mptsas_event_enable(mpt,
11541                             (mptsas_event_enable_t *)data, mode, rval);
11542                         break;
11543                 case MPTIOCTL_EVENT_REPORT:
11544                         /*
11545                          * The user has done an event report. Call our routine
11546                          * which does this.
11547                          */
11548                         status = mptsas_event_report(mpt,
11549                             (mptsas_event_report_t *)data, mode, rval);
11550                         break;
11551                 case MPTIOCTL_REG_ACCESS:
11552                         /*
11553                          * The user has requested register access.  Call our
11554                          * routine which does this.
11555                          */
11556                         status = mptsas_reg_access(mpt,
11557                             (mptsas_reg_access_t *)data, mode);
11558                         break;
11559                 default:
11560                         status = scsi_hba_ioctl(dev, cmd, data, mode, credp,
11561                             rval);
11562                         break;
11563         }

11565 out:
11566         return (status);
11567 }
_____unchanged_portion_omitted_
15347 static mptsas_target_t *
15348 mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr, mptsas_phymask_t phymask)
15349 {
15350         uint8_t                 phynum;
```



```
15351         uint64_t                wwn;
15352         int                     lun;
15353         mptsas_target_t         *ptgt = NULL;

15355         if (mptsas_parse_address(addr, &wwn, &phynum, &lun) != DDI_SUCCESS) {
15356                 return (NULL);
15357         }
15358         if (addr[0] == 'w') {
15359                 ptgt = mptsas_wwid_to_ptgt(mpt, (int)phymask, wwn);
15360         } else {
15361                 ptgt = mptsas_phy_to_tgt(mpt, (int)phymask, phynum);
15362         }
15363         return (ptgt);
15364 }

15342 int
15343 mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
15344     ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
15345     uint32_t alloc_size, ddi_dma_cookie_t *cookiep)
15346 {
15347         ddi_dma_cookie_t        new_cookie;
15348         size_t                  alloc_len;
15349         uint_t                  ncookie;

15351         if (cookiep == NULL)
15352                 cookiep = &new_cookie;

15354         if (ddi_dma_alloc_handle(mpt->m_dip, &dma_attr, DDI_DMA_SLEEP,
15355             NULL, dma_hdp) != DDI_SUCCESS) {
15356                 dma_hdp = NULL;
15357                 return (FALSE);
15358         }

15360         if (ddi_dma_mem_alloc(*dma_hdp, alloc_size, &mpt->m_dev_acc_attr,
15361             DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, dma_memp, &alloc_len,
15362             acc_hdp) != DDI_SUCCESS) {
15363                 ddi_dma_free_handle(dma_hdp);
15364                 dma_hdp = NULL;
15365                 return (FALSE);
15366         }

15368         if (ddi_dma_addr_bind_handle(*dma_hdp, NULL, *dma_memp, alloc_len,
15369             (DDI_DMA_RDWR | DDI_DMA_CONSISTENT), DDI_DMA_SLEEP, NULL,
15370             cookiep, &ncookie) != DDI_DMA_MAPPED) {
15371                 (void) ddi_dma_mem_free(acc_hdp);
15372                 ddi_dma_free_handle(dma_hdp);
15373                 dma_hdp = NULL;
15374                 return (FALSE);
15375         }

15377         return (TRUE);
15378 }
_____unchanged_portion_omitted_
```