```
**********************************************************
    3456 Tue Jun 17 10:45:12 2014
new/usr/src/man/man7d/mpt_sas.7d
NEX-1889 upstream
**********************************************************
'\" te
.\" Copyright (c) 2009, Sun Microsystems, Inc. All Rights Reserved
.\" Copyright 2014, Nexenta Systems, Inc. All Rights Reserved
.\" The contents of this file are subject to the terms of the Common Development
.\" or http://www.opensolaris.org/os/licensing.  See the License for the specifi
.\" the following below this CDDL HEADER, with the fields enclosed by brackets "
.TH MPT_SAS 7D "Apr 24, 2014"
.TH MPT_SAS 7D "Jul 16, 2009"
.SH NAME
mpt_sas \- SAS-2/3 host bus adapter driver
mpt_sas \- SAS-2 host bus adapter driver
.SH SYNOPSIS
.sp
.in +2
.nf
scsi@unit-address
.fi
.in -2

.SH DESCRIPTION
.sp
.LP
The \fBmpt_sas\fR host bus adapter driver is a nexus driver that supports the
LSI SAS200x/2x08 and SAS300x/3x08 series of chips. These chips support SAS/SATA
interfaces, including tagged and untagged queuing, SATA 3G/SAS 3G/SAS 6G/SAS
12G.
LSI SAS200x/2108 series of chips. These chips support SAS/SATA interfaces,
including tagged and untagged queuing, SATA 3G/SAS 3G/SAS 6G.
.SS "Configuration"
.sp
.LP
The \fBmpt_sas\fR driver is configured by defining properties in
\fBmpt_sas.conf\fR. These properties override the global SCSI settings. The
\fBmpt_sas\fR driver supports one modifiable property:
.sp
.ne 2
.na
\fB\fBmpxio-disable\fR\fR
.ad
.sp .6
.RS 4n
Solaris I/O multipathing is enabled or disabled on SAS devices with the
\fBmpxio-disable\fR property. Specifying \fBmpxio-disable="no"\fR activates I/O
multipathing, while \fBmpxio-disable="yes"\fR disables I/O multipathing.
.sp
Solaris I/O multipathing can be enabled or disabled on a per port basis. Per
port settings override the global setting for the specified ports.
.sp
The following example shows how to disable multipathing on port 0 whose parent
is \fB/pci@0,0/pci8086,2940@1c/pci1000,72@0\fR:
.sp
.in +2
.nf
name="mpt_sas"    parent="/pci@0,0/pci8086,2940@1c/pci1000,72@0"
mpxio-disable="yes";
.fi
.in -2

.RE

.SH EXAMPLES
```



```
.LP
\fBExample 1 \fRUsing the \fBmpt_sas\fR Configuration File to Disable MPXIO
.sp
.LP
Create a file called \fB/kernel/drv/mpt_sas.conf\fR and add the following line:

.sp
.in +2
.nf
name="mpt_sas"    parent="/pci@0,0/pci8086,2940@1c/pci1000,72@0"
mpxio-disable="yes";
.fi
.in -2

.SH FILES
.sp
.ne 2
.na
\fB\fB/kernel/drv/mpt_sas\fR\fR
.ad
.sp .6
.RS 4n
32-bit ELF kernel module
.RE

.sp
.ne 2
.na
\fB\fB/kernel/drv/sparcv9/mpt_sas\fR\fR
.ad
.sp .6
.RS 4n
64-bit SPARC ELF kernel module
.RE

.sp
.ne 2
.na
\fB\fB/kernel/drv/amd64/mpt_sas\fR\fR
.ad
.sp .6
.RS 4n
64-bit x86 ELF kernel module
.RE

.sp
.ne 2
.na
\fB\fB/kernel/drv/mpt_sas.conf\fR\fR
.ad
.sp .6
.RS 4n
Optional configuration file
.RE

.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for a description of the following attributes:
.sp

.sp
.TS
box;
l | l
l | l .
```

```
124 ATTRIBUTE TYPE  ATTRIBUTE VALUE
125 _
126 Architecture    SPARC, x86
127 .TE

129 .SH SEE ALSO
130 .sp
131 .LP
132 \fBprtconf\fR(1M), \fBdriver.conf\fR(4), \fBpci\fR(4), \fBattributes\fR(5),
133 \fBscsi_abort\fR(9F), \fBscsi_device\fR(9S), \fBscsi_extended_sense\fR(9S),
134 \fBscsi_inquiry\fR(9S), \fBscsi_hba_attach_setup\fR(9F),
135 \fBscsi_ifgetcap\fR(9F), \fBscsi_ifsetcap\fR(9F), \fBscsi_pkt\fR(9S),
136 \fBscsi_reset\fR(9F), \fBscsi_sync_pkt\fR(9F), \fBscsi_transport\fR(9F),
```

```
**********************************************************
    2864 Tue Jun 17 10:45:12 2014
new/usr/src/pkg/manifests/driver-storage-mpt_sas.mf
NEX-1889 upstream
**********************************************************
```

```
#
# 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) 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2014, Nexenta Systems, Inc. All Rights Reserved
#

#
# The default for payload-bearing actions in this package is to appear in the
# global zone only.  See the include file for greater detail, as well as
# information about overriding the defaults.
#
<include global_zone_only_component>
set name=pkg.fmri value=pkg:/driver/storage/mpt_sas@$(PKGVERS)
set name=pkg.description value="LSI MPT SAS 2.0/2.5 Controller HBA Driver"
set name=pkg.summary value="LSI MPT SAS 2.0/2.5 Controller HBA Driver"
set name=pkg.description value="LSI MPT SAS 2.0 Controller HBA Driver"
set name=pkg.summary value="LSI MPT SAS 2.0 Controller HBA Driver"
set name=info.classification \
    value=org.opensolaris.category.2008:Drivers/Storage
set name=variant.arch value=$(ARCH)
dir path=kernel group=sys
dir path=kernel/drv group=sys
dir path=kernel/drv/$(ARCH64) group=sys
dir path=usr/share/man
dir path=usr/share/man/man7d
driver name=mpt_sas class=scsi-self-identifying \
    alias=pci1000,64 \
    alias=pci1000,70 \
    alias=pci1000,72 \
    alias=pci1000,76 \
    alias=pciex1000,64 \
    alias=pciex1000,65 \
    alias=pciex1000,6e \
    alias=pciex1000,70 \
    alias=pciex1000,72 \
    alias=pciex1000,74 \
    alias=pciex1000,76 \
    alias=pciex1000,77 \
    alias=pciex1000,7e \
    alias=pciex1000,80 \
    alias=pciex1000,81 \
```



```
    alias=pciex1000,82 \
    alias=pciex1000,83 \
    alias=pciex1000,84 \
    alias=pciex1000,85 \
    alias=pciex1000,86 \
    alias=pciex1000,87 \
    alias=pciex1000,90 \
    alias=pciex1000,91 \
    alias=pciex1000,92 \
    alias=pciex1000,93 \
    alias=pciex1000,94 \
    alias=pciex1000,95 \
    alias=pciex1000,96 \
    alias=pciex1000,97
    alias=pciex1000,87
file path=kernel/drv/$(ARCH64)/mpt_sas group=sys
$(i386_ONLY)file path=kernel/drv/mpt_sas group=sys
file path=kernel/drv/mpt_sas.conf group=sys \
    original_name=SUNWmptsas:kernel/drv/mpt_sas.conf preserve=true
file path=usr/share/man/man7d/mpt_sas.7d
legacy pkg=SUNWmptsas desc="LSI MPT SAS 2.0/2.5 Controller HBA Driver" \
    name="LSI MPT SAS 2.0/2.5 Controller HBA Driver"
legacy pkg=SUNWmptsas desc="LSI MPT SAS 2.0 Controller HBA Driver" \
    name="LSI MPT SAS 2.0 Controller HBA Driver"
license cr_Sun license=cr_Sun
license lic_CDDL license=lic_CDDL
```

```
**********************************************************
    1812 Tue Jun 17 10:45:12 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mpt_sas.conf
NEX-1889 upstream
**********************************************************
```

```
#
# 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 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#
#

#
# The mpt_sas driver, as a pHCI driver, must specify the vHCI class it
# belongs to(scsi_vhci).
#
ddi-vhci-class="scsi_vhci";
#
# I/O multipathing feature (MPxIO) can be enabled or disabled using
# mpxio-disable property. Setting mpxio-disable="no" will activate
# I/O multipathing; setting mpxio-disable="yes" disables the feature.
#
# Global mpxio-disable property:
#
# To globally enable MPxIO on all LSI MPT SAS 2.0/2.5 controllers set:
# To globally enable MPxIO on all LSI MPT SAS 2.0 controllers set:
# mpxio-disable="no";
#
# To globally disable MPxIO on all LSI MPT SAS 2.0/2.5 controllers set:
# To globally disable MPxIO on all LSI MPT SAS 2.0 controllers set:
# mpxio-disable="yes";
#
# You can also enable or disable MPxIO on a per HBA basis.
# Per HBA settings override the global setting for the specified HBAs.
# To disable MPxIO on a controller whose parent is /pci@7c0/pci@0/pci@9
# and the unit-address is "0" set:
# name="mpt_sas" parent="/pci@7c0/pci@0/pci@9" unit-address="0" mpxio-disable="y
#
mpxio-disable="no";
```

```
**********************************************************
  441881 Tue Jun 17 10:45:12 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas.c
NEX-1889 upstream
**********************************************************
```

```
/*
 * 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 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2014, Joyent, Inc. All rights reserved.
 * Copyright 2014 OmniTI Computer Consulting, Inc. All rights reserved.
 * Copyright (c) 2014, Tegile 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/model.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/scsi/adapters/mpt_sas/mptsas_hash.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 */

/*
 * Resource initilaization for hardware
 */
static void mptsas_setup_cmd_reg(mptsas_t *mpt);
static void mptsas_disable_bus_master(mptsas_t *mpt);
static void mptsas_hba_fini(mptsas_t *mpt);
static void mptsas_cfg_fini(mptsas_t *mptsas_blkp);
static int mptsas_hba_setup(mptsas_t *mpt);
static void mptsas_hba_teardown(mptsas_t *mpt);
static int mptsas_config_space_init(mptsas_t *mpt);
static void mptsas_config_space_fini(mptsas_t *mpt);
static void mptsas_iport_register(mptsas_t *mpt);
static int mptsas_smp_setup(mptsas_t *mpt);
static void mptsas_smp_teardown(mptsas_t *mpt);
static int mptsas_cache_create(mptsas_t *mpt);
static void mptsas_cache_destroy(mptsas_t *mpt);
static int mptsas_alloc_request_frames(mptsas_t *mpt);
static int mptsas_alloc_reply_frames(mptsas_t *mpt);
static int mptsas_alloc_free_queue(mptsas_t *mpt);
static int mptsas_alloc_post_queue(mptsas_t *mpt);
static void mptsas_alloc_reply_args(mptsas_t *mpt);
static int mptsas_alloc_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd);
static void mptsas_free_extra_sgl_frame(mptsas_t *mpt, mptsas_cmd_t *cmd);
static int mptsas_init_chip(mptsas_t *mpt, int first_time);

/*
 * SCSA function prototypes
 */
static int mptsas_scsi_start(struct scsi_address *ap, struct scsi_pkt *pkt);
static int mptsas_scsi_reset(struct scsi_address *ap, int level);
static int mptsas_scsi_abort(struct scsi_address *ap, struct scsi_pkt *pkt);
static int mptsas_scsi_getcap(struct scsi_address *ap, char *cap, int tgtonly);
static int mptsas_scsi_setcap(struct scsi_address *ap, char *cap, int value,
    int tgtonly);
static void mptsas_scsi_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt);
static struct scsi_pkt *mptsas_scsi_init_pkt(struct scsi_address *ap,
    struct scsi_pkt *pkt, struct buf *bp, int cmdlen, int statuslen,
        int tgtlen, int flags, int (*callback)(), caddr_t arg);
static void mptsas_scsi_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt);
static void mptsas_scsi_destroy_pkt(struct scsi_address *ap,
    struct scsi_pkt *pkt);
static int mptsas_scsi_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd);
static void mptsas_scsi_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran, struct scsi_device *sd);
static int mptsas_scsi_reset_notify(struct scsi_address *ap, int flag,
    void (*callback)(caddr_t), caddr_t arg);
static int mptsas_get_name(struct scsi_device *sd, char *name, int len);
static int mptsas_get_bus_addr(struct scsi_device *sd, char *name, int len);
static int mptsas_scsi_quiesce(dev_info_t *dip);
static int mptsas_scsi_unquiesce(dev_info_t *dip);
static int mptsas_bus_config(dev_info_t *pdip, uint_t flags,
    ddi_bus_config_op_t op, void *arg, dev_info_t **childp);

/*
 * SMP functions
 */
static int mptsas_smp_start(struct smp_pkt *smp_pkt);

/*
 * internal function prototypes.
 */
```



```
static void mptsas_list_add(mptsas_t *mpt);
static void mptsas_list_del(mptsas_t *mpt);

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

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

static void mptsas_ncmds_checkdrain(void *arg);

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

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

static void mptsas_handle_qfull(mptsas_t *mpt, mptsas_cmd_t *cmd);
static void mptsas_handle_event(void *args);
static int mptsas_handle_event_sync(void *args);
static void mptsas_handle_dr(void *args);
static void mptsas_handle_topo_change(mptsas_topo_change_list_t *topo_node,
    dev_info_t *pdip);

static void mptsas_restart_cmd(void *);

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

static uint_t mptsas_intr(caddr_t arg1, caddr_t arg2);
static void mptsas_process_intr(mptsas_t *mpt,
    pMpi2ReplyDescriptorsUnion_t reply_desc_union);
static void mptsas_handle_scsi_io_success(mptsas_t *mpt,
    pMpi2ReplyDescriptorsUnion_t reply_desc);
static void mptsas_handle_address_reply(mptsas_t *mpt,
    pMpi2ReplyDescriptorsUnion_t reply_desc);
static int mptsas_wait_intr(mptsas_t *mpt, int polltime);
static void mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd,
    uint32_t *control, pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl);

static void mptsas_watch(void *arg);
static void mptsas_watchsubr(mptsas_t *mpt);
static void mptsas_cmd_timeout(mptsas_t *mpt, mptsas_target_t *ptgt);

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

static uint8_t mptsas_get_fw_diag_buffer_number(mptsas_t *mpt,
    uint32_t unique_id);
static void mptsas_start_diag(mptsas_t *mpt, mptsas_cmd_t *cmd);
static int mptsas_post_fw_diag_buffer(mptsas_t *mpt,
    mptsas_fw_diagnostic_buffer_t *pBuffer, uint32_t *return_code);
static int mptsas_release_fw_diag_buffer(mptsas_t *mpt,
```

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

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

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

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

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

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

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

304 static void mptsas_deliver_doneq_thread(mptsas_t *mpt);
305 static void mptsas_doneq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
306 static void mptsas_doneq_mv(mptsas_t *mpt, uint64_t t);

308 static mptsas_cmd_t *mptsas_doneq_thread_rm(mptsas_t *mpt, uint64_t t);
309 static void mptsas_doneq_empty(mptsas_t *mpt);
310 static void mptsas_doneq_thread(mptsas_doneq_thread_arg_t *arg);

312 static mptsas_cmd_t *mptsas_waitq_rm(mptsas_t *mpt);
313 static void mptsas_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);
314 static mptsas_cmd_t *mptsas_tx_waitq_rm(mptsas_t *mpt);
315 static void mptsas_tx_waitq_delete(mptsas_t *mpt, mptsas_cmd_t *cmd);


318 static void mptsas_start_watch_reset_delay();
319 static void mptsas_setup_bus_reset_delay(mptsas_t *mpt);
320 static void mptsas_watch_reset_delay(void *arg);
321 static int mptsas_watch_reset_delay_subr(mptsas_t *mpt);

323 /*
324  * helper functions
```



```
325  */
326 static void mptsas_dump_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);

328 static dev_info_t *mptsas_find_child(dev_info_t *pdip, char *name);
329 static dev_info_t *mptsas_find_child_phy(dev_info_t *pdip, uint8_t phy);
330 static dev_info_t *mptsas_find_child_addr(dev_info_t *pdip, uint64_t sasaddr,
331     int lun);
332 static mdi_pathinfo_t *mptsas_find_path_addr(dev_info_t *pdip, uint64_t sasaddr,
333     int lun);
334 static mdi_pathinfo_t *mptsas_find_path_phy(dev_info_t *pdip, uint8_t phy);
335 static dev_info_t *mptsas_find_smp_child(dev_info_t *pdip, char *str_wwn);

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

341 static mptsas_target_t *mptsas_phy_to_tgt(mptsas_t *mpt,
342     mptsas_phymask_t phymask, uint8_t phy);
343 static mptsas_target_t *mptsas_wwid_to_ptgt(mptsas_t *mpt,
344     mptsas_phymask_t phymask, uint64_t wwid);
345 static mptsas_smp_t *mptsas_wwid_to_psmp(mptsas_t *mpt,
346     mptsas_phymask_t phymask, uint64_t wwid);

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

351 static int mptsas_get_target_device_info(mptsas_t *mpt, uint32_t page_address,
352     uint16_t *handle, mptsas_target_t **pptgt);
353 static void mptsas_update_phymask(mptsas_t *mpt);

355 static int mptsas_send_sep(mptsas_t *mpt, mptsas_target_t *ptgt,
356     uint32_t *status, uint8_t cmd);
357 static dev_info_t *mptsas_get_dip_from_dev(dev_t dev,
358     mptsas_phymask_t *phymask);
359 static mptsas_target_t *mptsas_addr_to_ptgt(mptsas_t *mpt, char *addr,
360     mptsas_phymask_t phymask);
361 static int mptsas_flush_led_status(mptsas_t *mpt, mptsas_target_t *ptgt);


364 /*
365  * Enumeration / DR functions
366  */
367 static void mptsas_config_all(dev_info_t *pdip);
368 static int mptsas_config_one_addr(dev_info_t *pdip, uint64_t sasaddr, int lun,
369     dev_info_t **lundip);
370 static int mptsas_config_one_phy(dev_info_t *pdip, uint8_t phy, int lun,
371     dev_info_t **lundip);

373 static int mptsas_config_target(dev_info_t *pdip, mptsas_target_t *ptgt);
374 static int mptsas_offline_target(dev_info_t *pdip, char *name);

376 static int mptsas_config_raid(dev_info_t *pdip, uint16_t target,
377     dev_info_t **dip);

379 static int mptsas_config_luns(dev_info_t *pdip, mptsas_target_t *ptgt);
380 static int mptsas_probe_lun(dev_info_t *pdip, int lun,
381     dev_info_t **dip, mptsas_target_t *ptgt);

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

386 static int mptsas_create_phys_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
387     char *guid, dev_info_t **dip, mptsas_target_t *ptgt, int lun);
388 static int mptsas_create_virt_lun(dev_info_t *pdip, struct scsi_inquiry *sd,
389     char *guid, dev_info_t **dip, mdi_pathinfo_t **pip, mptsas_target_t *ptgt,
390     int lun);
```

```
392 static void mptsas_offline_missed_luns(dev_info_t *pdip,
393     uint16_t *repluns, int lun_cnt, mptsas_target_t *ptgt);
394 static int mptsas_offline_lun(dev_info_t *pdip, dev_info_t *rdip,
395     mdi_pathinfo_t *rpip, uint_t flags);

397 static int mptsas_config_smp(dev_info_t *pdip, uint64_t sas_wwn,
398     dev_info_t **smp_dip);
399 static int mptsas_offline_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
400     uint_t flags);

402 static int mptsas_event_query(mptsas_t *mpt, mptsas_event_query_t *data,
403     int mode, int *rval);
404 static int mptsas_event_enable(mptsas_t *mpt, mptsas_event_enable_t *data,
405     int mode, int *rval);
406 static int mptsas_event_report(mptsas_t *mpt, mptsas_event_report_t *data,
407     int mode, int *rval);
408 static void mptsas_record_event(void *args);
409 static int mptsas_reg_access(mptsas_t *mpt, mptsas_reg_access_t *data,
410     int mode);

412 mptsas_target_t *mptsas_tgt_alloc(mptsas_t *, uint16_t, uint64_t,
413     uint32_t, mptsas_phymask_t, uint8_t);
414 static mptsas_smp_t *mptsas_smp_alloc(mptsas_t *, mptsas_smp_t *);
415 static int mptsas_online_smp(dev_info_t *pdip, mptsas_smp_t *smp_node,
416     dev_info_t **smp_dip);

418 /*
419  * Power management functions
420  */
421 static int mptsas_get_pci_cap(mptsas_t *mpt);
422 static int mptsas_init_pm(mptsas_t *mpt);

424 /*
425  * MPT MSI tunable:
426  *
427  * By default MSI is enabled on all supported platforms.
428  */
429 boolean_t mptsas_enable_msi = B_TRUE;
430 boolean_t mptsas_physical_bind_failed_page_83 = B_FALSE;

432 static int mptsas_register_intrs(mptsas_t *);
433 static void mptsas_unregister_intrs(mptsas_t *);
434 static int mptsas_add_intrs(mptsas_t *, int);
435 static void mptsas_rem_intrs(mptsas_t *);

437 /*
438  * FMA Prototypes
439  */
440 static void mptsas_fm_init(mptsas_t *mpt);
441 static void mptsas_fm_fini(mptsas_t *mpt);
442 static int mptsas_fm_error_cb(dev_info_t *, ddi_fm_error_t *, const void *);

444 extern pri_t minclsyspri, maxclsyspri;

446 /*
447  * This device is created by the SCSI pseudo nexus driver (SCSI vHCI).  It is
448  * under this device that the paths to a physical device are created when
449  * MPxIO is used.
450  */
451 extern dev_info_t       *scsi_vhci_dip;

453 /*
454  * Tunable timeout value for Inquiry VPD page 0x83
455  * By default the value is 30 seconds.
456  */
```



```
457 int mptsas_inq83_retry_timeout = 30;

459 /*
460  * This is used to allocate memory for message frame storage, not for
461  * data I/O DMA. All message frames must be stored in the first 4G of
462  * physical memory.
463  */
464 ddi_dma_attr_t mptsas_dma_attrs = {
465         DMA_ATTR_V0,    /* attribute layout version             */
466         0x0ull,         /* address low - should be 0 (longlong) */
467         0xffffffffull,  /* address high - 32-bit max range      */
468         0x00ffffffull,  /* count max - max DMA object size      */
469         4,              /* allocation alignment requirements    */
470         0x78,           /* burstsizes - binary encoded values   */
471         1,              /* minxfer - gran. of DMA engine        */
472         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
473         0xffffffffull,  /* max segment size (DMA boundary)      */
474         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
475         512,            /* granularity - device transfer size   */
476         0               /* flags, set to 0                      */
477 };

479 /*
480  * This is used for data I/O DMA memory allocation. (full 64-bit DMA
481  * physical addresses are supported.)
482  */
483 ddi_dma_attr_t mptsas_dma_attrs64 = {
484         DMA_ATTR_V0,    /* attribute layout version             */
485         0x0ull,         /* address low - should be 0 (longlong) */
486         0xffffffffffffffffull,  /* address high - 64-bit max    */
487         0x00ffffffull,  /* count max - max DMA object size      */
488         4,              /* allocation alignment requirements    */
489         0x78,           /* burstsizes - binary encoded values   */
490         1,              /* minxfer - gran. of DMA engine        */
491         0x00ffffffull,  /* maxxfer - gran. of DMA engine        */
492         0xffffffffull,  /* max segment size (DMA boundary)      */
493         MPTSAS_MAX_DMA_SEGS, /* scatter/gather list length      */
494         512,            /* granularity - device transfer size   */
495         0               /* flags, set to 0 */
494         DDI_DMA_RELAXED_ORDERING        /* flags, enable relaxed ordering */
496 };
_____unchanged_portion_omitted_
561 #define TARGET_PROP     "target"
562 #define LUN_PROP        "lun"
563 #define LUN64_PROP      "lun64"
564 #define SAS_PROP        "sas-mpt"
565 #define MDI_GUID        "wwn"
566 #define NDI_GUID        "guid"
567 #define MPTSAS_DEV_GONE "mptsas_dev_gone"

569 /*
570  * Local static data
571  */
572 #if defined(MPTSAS_DEBUG)
573 uint32_t mptsas_debug_flags = 0x0;
572 uint32_t mptsas_debug_flags = 0;
574 #endif  /* defined(MPTSAS_DEBUG) */
575 uint32_t mptsas_debug_resets = 0;

577 static kmutex_t         mptsas_global_mutex;
578 static void             *mptsas_state;          /* soft state ptr */
579 static krwlock_t        mptsas_global_rwlock;

581 static kmutex_t         mptsas_log_mutex;
582 static char             mptsas_log_buf[256];
583 _NOTE(MUTEX_PROTECTS_DATA(mptsas_log_mutex, mptsas_log_buf))
```

```
585 static mptsas_t *mptsas_head, *mptsas_tail;
586 static clock_t mptsas_scsi_watchdog_tick;
587 static clock_t mptsas_tick;
588 static timeout_id_t mptsas_reset_watch;
589 static timeout_id_t mptsas_timeout_id;
590 static int mptsas_timeouts_enabled = 0;

592 /*
593  * The only software retriction on switching msg buffers to 64 bit seems to
594  * be the Auto Request Sense interface. The high 32 bits for all such
595  * requests appear to be required to sit in the same 4G segment.
596  * See initialization of SenseBufferAddressHigh in mptsas_init.c, and
597  * the use of SenseBufferLowAddress in requests. Note that there is
598  * currently a dependency on scsi_alloc_consistent_buf() adhering to
599  * this requirement.
600  * There is also a question about improved performance over PCI/PCIX
601  * if transfers are within the first 4Gb.
602  */
603 static int mptsas_use_64bit_msgaddr = 0;

605 /*
606  * warlock directives
607  */
608 _NOTE(SCHEME_PROTECTS_DATA("unique per pkt", scsi_pkt \
609         mptsas_cmd NcrTableIndirect buf scsi_cdb scsi_status))
610 _NOTE(SCHEME_PROTECTS_DATA("unique per pkt", smp_pkt))
611 _NOTE(SCHEME_PROTECTS_DATA("stable data", scsi_device scsi_address))
612 _NOTE(SCHEME_PROTECTS_DATA("No Mutex Needed", mptsas_tgt_private))
613 _NOTE(SCHEME_PROTECTS_DATA("No Mutex Needed", scsi_hba_tran::tran_tgt_private))

615 /*
616  * SM - HBA statics
617  */
618 char    *mptsas_driver_rev = MPTSAS_MOD_STRING;

620 #ifdef MPTSAS_DEBUG
621 void debug_enter(char *);
622 #endif

624 /*
625  * Notes:
626  *      - scsi_hba_init(9F) initializes SCSI HBA modules
627  *      - must call scsi_hba_fini(9F) if modload() fails
628  */
629 int
630 _init(void)
631 {
632         int status;
633         /* CONSTCOND */
634         ASSERT(NO_COMPETING_THREADS);

636         NDBG0(("_init"));

638         status = ddi_soft_state_init(&mptsas_state, MPTSAS_SIZE,
639             MPTSAS_INITIAL_SOFT_SPACE);
640         if (status != 0) {
641                 return (status);
642         }

644         if ((status = scsi_hba_init(&modlinkage)) != 0) {
645                 ddi_soft_state_fini(&mptsas_state);
646                 return (status);
647         }

649         mutex_init(&mptsas_global_mutex, NULL, MUTEX_DRIVER, NULL);
```



```
650         rw_init(&mptsas_global_rwlock, NULL, RW_DRIVER, NULL);
651         mutex_init(&mptsas_log_mutex, NULL, MUTEX_DRIVER, NULL);

653         if ((status = mod_install(&modlinkage)) != 0) {
654                 mutex_destroy(&mptsas_log_mutex);
655                 rw_destroy(&mptsas_global_rwlock);
656                 mutex_destroy(&mptsas_global_mutex);
657                 ddi_soft_state_fini(&mptsas_state);
658                 scsi_hba_fini(&modlinkage);
659         }

661         return (status);
662 }
_____unchanged_portion_omitted_

1025 /*
1026  * Notes:
1027  *      Set up all device state and allocate data structures,
1028  *      mutexes, condition variables, etc. for device operation.
1029  *      Add interrupts needed.
1030  *      Return DDI_SUCCESS if device is ready, else return DDI_FAILURE.
1031  */
1032 static int
1033 mptsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
1034 {
1035         mptsas_t                *mpt = NULL;
1036         int                     instance, i, j;
1037         int                     doneq_thread_num;
1038         char                    intr_added = 0;
1039         char                    map_setup = 0;
1040         char                    config_setup = 0;
1041         char                    hba_attach_setup = 0;
1042         char                    smp_attach_setup = 0;
1043         char                    mutex_init_done = 0;
1044         char                    event_taskq_create = 0;
1045         char                    dr_taskq_create = 0;
1046         char                    doneq_thread_create = 0;
1047         scsi_hba_tran_t         *hba_tran;
1048         uint_t                  mem_bar = MEM_SPACE;
1049         int                     rval = DDI_FAILURE;

1051         /* CONSTCOND */
1052         ASSERT(NO_COMPETING_THREADS);

1054         if (scsi_hba_iport_unit_address(dip)) {
1055                 return (mptsas_iport_attach(dip, cmd));
1056         }

1058         switch (cmd) {
1059         case DDI_ATTACH:
1060                 break;

1062         case DDI_RESUME:
1063                 if ((hba_tran = ddi_get_driver_private(dip)) == NULL)
1064                         return (DDI_FAILURE);

1066                 mpt = TRAN2MPT(hba_tran);

1068                 if (!mpt) {
1069                         return (DDI_FAILURE);
1070                 }

1072                 /*
1073                  * Reset hardware and softc to "no outstanding commands"
1074                  * Note that a check condition can result on first command
1075                  * to a target.
```

```
1076                 */
1077                mutex_enter(&mpt->m_mutex);

1079                /*
1080                 * raise power.
1081                 */
1082                if (mpt->m_options & MPTSAS_OPT_PM) {
1083                        mutex_exit(&mpt->m_mutex);
1084                        (void) pm_busy_component(dip, 0);
1085                        rval = pm_power_has_changed(dip, 0, PM_LEVEL_D0);
1086                        if (rval == DDI_SUCCESS) {
1087                                mutex_enter(&mpt->m_mutex);
1088                        } else {
1089                                /*
1090                                 * The pm_raise_power() call above failed,
1091                                 * and that can only occur if we were unable
1092                                 * to reset the hardware.  This is probably
1093                                 * due to unhealty hardware, and because
1094                                 * important filesystems(such as the root
1095                                 * filesystem) could be on the attached disks,
1096                                 * it would not be a good idea to continue,
1097                                 * as we won't be entirely certain we are
1098                                 * writing correct data.  So we panic() here
1099                                 * to not only prevent possible data corruption,
1100                                 * but to give developers or end users a hope
1101                                 * of identifying and correcting any problems.
1102                                 */
1103                                fm_panic("mptsas could not reset hardware "
1104                                    "during resume");
1105                        }
1106                }

1108                mpt->m_suspended = 0;

1110                /*
1111                 * Reinitialize ioc
1112                 */
1113                mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1114                if (mptsas_init_chip(mpt, FALSE) == DDI_FAILURE) {
1115                        mutex_exit(&mpt->m_mutex);
1116                        if (mpt->m_options & MPTSAS_OPT_PM) {
1117                                (void) pm_idle_component(dip, 0);
1118                        }
1119                        fm_panic("mptsas init chip fail during resume");
1120                }
1121                /*
1122                 * mptsas_update_driver_data needs interrupts so enable them
1123                 * first.
1124                 */
1125                MPTSAS_ENABLE_INTR(mpt);
1126                mptsas_update_driver_data(mpt);

1128                /* start requests, if possible */
1129                mptsas_restart_hba(mpt);

1131                mutex_exit(&mpt->m_mutex);

1133                /*
1134                 * Restart watch thread
1135                 */
1136                mutex_enter(&mptsas_global_mutex);
1137                if (mptsas_timeout_id == 0) {
1138                        mptsas_timeout_id = timeout(mptsas_watch, NULL,
1139                            mptsas_tick);
1140                        mptsas_timeouts_enabled = 1;
1141                }
```



```
1142                mutex_exit(&mptsas_global_mutex);

1144                /* report idle status to pm framework */
1145                if (mpt->m_options & MPTSAS_OPT_PM) {
1146                        (void) pm_idle_component(dip, 0);
1147                }

1149                return (DDI_SUCCESS);

1151        default:
1152                return (DDI_FAILURE);

1154        }

1156        instance = ddi_get_instance(dip);

1158        /*
1159         * Allocate softc information.
1160         */
1161        if (ddi_soft_state_zalloc(mptsas_state, instance) != DDI_SUCCESS) {
1162                mptsas_log(NULL, CE_WARN,
1163                    "mptsas%d: cannot allocate soft state", instance);
1164                goto fail;
1165        }

1167        mpt = ddi_get_soft_state(mptsas_state, instance);

1169        if (mpt == NULL) {
1170                mptsas_log(NULL, CE_WARN,
1171                    "mptsas%d: cannot get soft state", instance);
1172                goto fail;
1173        }

1175        /* Indicate that we are 'sizeof (scsi_*(9S))' clean. */
1176        scsi_size_clean(dip);

1178        mpt->m_dip = dip;
1179        mpt->m_instance = instance;

1181        /* Make a per-instance copy of the structures */
1182        mpt->m_io_dma_attr = mptsas_dma_attrs64;
1183        if (mptsas_use_64bit_msgaddr) {
1184                mpt->m_msg_dma_attr = mptsas_dma_attrs64;
1185        } else {
1186                mpt->m_msg_dma_attr = mptsas_dma_attrs;
1187        }
1188        mpt->m_reg_acc_attr = mptsas_dev_attr;
1189        mpt->m_dev_acc_attr = mptsas_dev_attr;

1191        /*
1192         * Initialize FMA
1193         */
1194        mpt->m_fm_capabilities = ddi_getprop(DDI_DEV_T_ANY, mpt->m_dip,
1195            DDI_PROP_CANSLEEP | DDI_PROP_DONTPASS, "fm-capable",
1196            DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
1197            DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);

1199        mptsas_fm_init(mpt);

1201        if (mptsas_alloc_handshake_msg(mpt,
1202            sizeof (Mpi2SCSITaskManagementRequest_t)) == DDI_FAILURE) {
1203                mptsas_log(mpt, CE_WARN, "cannot initialize handshake msg.");
1204                goto fail;
1205        }

1207        /*
```

```
1208          * Setup configuration space
1209          */
1210         if (mptsas_config_space_init(mpt) == FALSE) {
1211                 mptsas_log(mpt, CE_WARN, "mptsas_config_space_init failed");
1212                 goto fail;
1213         }
1214         config_setup++;

1216         if (ddi_regs_map_setup(dip, mem_bar, (caddr_t *)&mpt->m_reg,
1217             0, 0, &mpt->m_reg_acc_attr, &mpt->m_datap) != DDI_SUCCESS) {
1218                 mptsas_log(mpt, CE_WARN, "map setup failed");
1219                 goto fail;
1220         }
1221         map_setup++;

1223         /*
1224          * A taskq is created for dealing with the event handler
1225          */
1226         if ((mpt->m_event_taskq = ddi_taskq_create(dip, "mptsas_event_taskq",
1227             1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1228                 mptsas_log(mpt, CE_NOTE, "ddi_taskq_create failed");
1229                 goto fail;
1230         }
1231         event_taskq_create++;

1233         /*
1234          * A taskq is created for dealing with dr events
1235          */
1236         if ((mpt->m_dr_taskq = ddi_taskq_create(dip,
1237             "mptsas_dr_taskq",
1238             1, TASKQ_DEFAULTPRI, 0)) == NULL) {
1239                 mptsas_log(mpt, CE_NOTE, "ddi_taskq_create for discovery "
1240                     "failed");
1241                 goto fail;
1242         }
1243         dr_taskq_create++;

1245         mpt->m_doneq_thread_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1246             0, "mptsas_doneq_thread_threshold_prop", 10);
1247         mpt->m_doneq_length_threshold = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1248             0, "mptsas_doneq_length_threshold_prop", 8);
1249         mpt->m_doneq_thread_n = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1250             0, "mptsas_doneq_thread_n_prop", 8);

1252         if (mpt->m_doneq_thread_n) {
1253                 cv_init(&mpt->m_doneq_thread_cv, NULL, CV_DRIVER, NULL);
1254                 mutex_init(&mpt->m_doneq_mutex, NULL, MUTEX_DRIVER, NULL);

1256                 mutex_enter(&mpt->m_doneq_mutex);
1257                 mpt->m_doneq_thread_id =
1258                     kmem_zalloc(sizeof (mptsas_doneq_thread_list_t)
1259                     * mpt->m_doneq_thread_n, KM_SLEEP);

1261                 for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1262                         cv_init(&mpt->m_doneq_thread_id[j].cv, NULL,
1263                             CV_DRIVER, NULL);
1264                         mutex_init(&mpt->m_doneq_thread_id[j].mutex, NULL,
1265                             MUTEX_DRIVER, NULL);
1266                         mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1267                         mpt->m_doneq_thread_id[j].flag |=
1268                             MPTSAS_DONEQ_THREAD_ACTIVE;
1269                         mpt->m_doneq_thread_id[j].arg.mpt = mpt;
1270                         mpt->m_doneq_thread_id[j].arg.t = j;
1271                         mpt->m_doneq_thread_id[j].threadp =
1272                             thread_create(NULL, 0, mptsas_doneq_thread,
1273                             &mpt->m_doneq_thread_id[j].arg,
```



```
1274                             0, &p0, TS_RUN, minclsyspri);
1275                         mpt->m_doneq_thread_id[j].donetail =
1276                             &mpt->m_doneq_thread_id[j].doneq;
1277                         mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
1278                 }
1279                 mutex_exit(&mpt->m_doneq_mutex);
1280                 doneq_thread_create++;
1281         }

1283         /*
1284          * Disable hardware interrupt since we're not ready to
1285          * handle it yet.
1286          */
1287         MPTSAS_DISABLE_INTR(mpt);
1288         if (mptsas_register_intrs(mpt) == FALSE)
1289                 goto fail;
1290         intr_added++;

1292         /* Initialize mutex used in interrupt handler */
1293         mutex_init(&mpt->m_mutex, NULL, MUTEX_DRIVER,
1294             DDI_INTR_PRI(mpt->m_intr_pri));
1295         mutex_init(&mpt->m_passthru_mutex, NULL, MUTEX_DRIVER, NULL);
1296         mutex_init(&mpt->m_tx_waitq_mutex, NULL, MUTEX_DRIVER,
1297             DDI_INTR_PRI(mpt->m_intr_pri));
1298         for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1299                 mutex_init(&mpt->m_phy_info[i].smhba_info.phy_mutex,
1300                     NULL, MUTEX_DRIVER,
1301                     DDI_INTR_PRI(mpt->m_intr_pri));
1302         }

1304         cv_init(&mpt->m_cv, NULL, CV_DRIVER, NULL);
1305         cv_init(&mpt->m_passthru_cv, NULL, CV_DRIVER, NULL);
1306         cv_init(&mpt->m_fw_cv, NULL, CV_DRIVER, NULL);
1307         cv_init(&mpt->m_config_cv, NULL, CV_DRIVER, NULL);
1308         cv_init(&mpt->m_fw_diag_cv, NULL, CV_DRIVER, NULL);
1309         mutex_init_done++;

1283         /*
1284          * Disable hardware interrupt since we're not ready to
1285          * handle it yet.
1286          */
1287         MPTSAS_DISABLE_INTR(mpt);
1288         if (mptsas_register_intrs(mpt) == FALSE)
1289                 goto fail;
1290         intr_added++;

1311         mutex_enter(&mpt->m_mutex);
1312         /*
1313          * Initialize power management component
1314          */
1315         if (mpt->m_options & MPTSAS_OPT_PM) {
1316                 if (mptsas_init_pm(mpt)) {
1317                         mutex_exit(&mpt->m_mutex);
1318                         mptsas_log(mpt, CE_WARN, "mptsas pm initialization "
1319                             "failed");
1320                         goto fail;
1321                 }
1322         }

1324         /*
1325          * Initialize chip using Message Unit Reset, if allowed
1326          */
1327         mpt->m_softstate |= MPTSAS_SS_MSG_UNIT_RESET;
1328         if (mptsas_init_chip(mpt, TRUE) == DDI_FAILURE) {
1329                 mutex_exit(&mpt->m_mutex);
1330                 mptsas_log(mpt, CE_WARN, "mptsas chip initialization failed");
```

```
1331                goto fail;
1332        }
1333
1334        /*
1335         * Fill in the phy_info structure and get the base WWID
1336         */
1337        if (mptsas_get_manufacture_page5(mpt) == DDI_FAILURE) {
1338                mptsas_log(mpt, CE_WARN,
1339                    "mptsas_get_manufacture_page5 failed!");
1340                goto fail;
1341        }
1342
1343        if (mptsas_get_sas_io_unit_page_hndshk(mpt)) {
1344                mptsas_log(mpt, CE_WARN,
1345                    "mptsas_get_sas_io_unit_page_hndshk failed!");
1346                goto fail;
1347        }
1348
1349        if (mptsas_get_manufacture_page0(mpt) == DDI_FAILURE) {
1350                mptsas_log(mpt, CE_WARN,
1351                    "mptsas_get_manufacture_page0 failed!");
1352                goto fail;
1353        }
1354
1355        mutex_exit(&mpt->m_mutex);
1356
1357        /*
1358         * Register the iport for multiple port HBA
1359         */
1360        mptsas_iport_register(mpt);
1361
1362        /*
1363         * initialize SCSI HBA transport structure
1364         */
1365        if (mptsas_hba_setup(mpt) == FALSE)
1366                goto fail;
1367        hba_attach_setup++;
1368
1369        if (mptsas_smp_setup(mpt) == FALSE)
1370                goto fail;
1371        smp_attach_setup++;
1372
1373        if (mptsas_cache_create(mpt) == FALSE)
1374                goto fail;
1375
1376        mpt->m_scsi_reset_delay = ddi_prop_get_int(DDI_DEV_T_ANY,
1377            dip, 0, "scsi-reset-delay", SCSI_DEFAULT_RESET_DELAY);
1378        if (mpt->m_scsi_reset_delay == 0) {
1379                mptsas_log(mpt, CE_NOTE,
1380                    "scsi_reset_delay of 0 is not recommended,"
1381                    " resetting to SCSI_DEFAULT_RESET_DELAY\n");
1382                mpt->m_scsi_reset_delay = SCSI_DEFAULT_RESET_DELAY;
1383        }
1384
1385        /*
1386         * Initialize the wait and done FIFO queue
1387         */
1388        mpt->m_donetail = &mpt->m_doneq;
1389        mpt->m_waitqtail = &mpt->m_waitq;
1390        mpt->m_tx_waitqtail = &mpt->m_tx_waitq;
1391        mpt->m_tx_draining = 0;
1392
1393        /*
1394         * ioc cmd queue initialize
1395         */
1396        mpt->m_ioc_event_cmdtail = &mpt->m_ioc_event_cmdq;
```



```
1397        mpt->m_dev_handle = 0xFFFF;
1398
1399        MPTSAS_ENABLE_INTR(mpt);
1400
1401        /*
1402         * enable event notification
1403         */
1404        mutex_enter(&mpt->m_mutex);
1405        if (mptsas_ioc_enable_event_notification(mpt)) {
1406                mutex_exit(&mpt->m_mutex);
1407                goto fail;
1408        }
1409        mutex_exit(&mpt->m_mutex);
1410
1411        /*
1412         * Initialize PHY info for smhba
1413         */
1414        if (mptsas_smhba_setup(mpt)) {
1415                mptsas_log(mpt, CE_WARN, "mptsas phy initialization "
1416                    "failed");
1417                goto fail;
1418        }
1419
1420        /* Check all dma handles allocated in attach */
1421        if ((mptsas_check_dma_handle(mpt->m_dma_req_frame_hdl)
1422            != DDI_SUCCESS) ||
1423            (mptsas_check_dma_handle(mpt->m_dma_reply_frame_hdl)
1424            != DDI_SUCCESS) ||
1425            (mptsas_check_dma_handle(mpt->m_dma_free_queue_hdl)
1426            != DDI_SUCCESS) ||
1427            (mptsas_check_dma_handle(mpt->m_dma_post_queue_hdl)
1428            != DDI_SUCCESS) ||
1429            (mptsas_check_dma_handle(mpt->m_hshk_dma_hdl)
1430            != DDI_SUCCESS)) {
1431                goto fail;
1432        }
1433
1434        /* Check all acc handles allocated in attach */
1435        if ((mptsas_check_acc_handle(mpt->m_datap) != DDI_SUCCESS) ||
1436            (mptsas_check_acc_handle(mpt->m_acc_req_frame_hdl)
1437            != DDI_SUCCESS) ||
1438            (mptsas_check_acc_handle(mpt->m_acc_reply_frame_hdl)
1439            != DDI_SUCCESS) ||
1440            (mptsas_check_acc_handle(mpt->m_acc_free_queue_hdl)
1441            != DDI_SUCCESS) ||
1442            (mptsas_check_acc_handle(mpt->m_acc_post_queue_hdl)
1443            != DDI_SUCCESS) ||
1444            (mptsas_check_acc_handle(mpt->m_hshk_acc_hdl)
1445            != DDI_SUCCESS) ||
1446            (mptsas_check_acc_handle(mpt->m_config_handle)
1447            != DDI_SUCCESS)) {
1448                goto fail;
1449        }
1450
1451        /*
1452         * After this point, we are not going to fail the attach.
1453         */
1454        /*
1455         * used for mptsas_watch
1456         */
1457        mptsas_list_add(mpt);
1458
1459        mutex_enter(&mptsas_global_mutex);
1460        if (mptsas_timeouts_enabled == 0) {
1461                mptsas_scsi_watchdog_tick = ddi_prop_get_int(DDI_DEV_T_ANY,
1462                    dip, 0, "scsi-watchdog-tick", DEFAULT_WD_TICK);
```

```
1464                mptsas_tick = mptsas_scsi_watchdog_tick *
1465                    drv_usectohz((clock_t)1000000);

1467                mptsas_timeout_id = timeout(mptsas_watch, NULL, mptsas_tick);
1468                mptsas_timeouts_enabled = 1;
1469        }
1470        mutex_exit(&mptsas_global_mutex);

1472        /* Print message of HBA present */
1473        ddi_report_dev(dip);

1475        /* report idle status to pm framework */
1476        if (mpt->m_options & MPTSAS_OPT_PM) {
1477                (void) pm_idle_component(dip, 0);
1478        }

1480        return (DDI_SUCCESS);

1482 fail:
1483        mptsas_log(mpt, CE_WARN, "attach failed");
1484        mptsas_fm_ereport(mpt, DDI_FM_DEVICE_NO_RESPONSE);
1485        ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_LOST);
1486        if (mpt) {
1487                mutex_enter(&mptsas_global_mutex);

1489                if (mptsas_timeout_id && (mptsas_head == NULL)) {
1490                        timeout_id_t tid = mptsas_timeout_id;
1491                        mptsas_timeouts_enabled = 0;
1492                        mptsas_timeout_id = 0;
1493                        mutex_exit(&mptsas_global_mutex);
1494                        (void) untimeout(tid);
1495                        mutex_enter(&mptsas_global_mutex);
1496                }
1497                mutex_exit(&mptsas_global_mutex);
1498                /* deallocate in reverse order */
1499                mptsas_cache_destroy(mpt);

1501                if (smp_attach_setup) {
1502                        mptsas_smp_teardown(mpt);
1503                }
1504                if (hba_attach_setup) {
1505                        mptsas_hba_teardown(mpt);
1506                }

1508                if (mpt->m_targets)
1509                        refhash_destroy(mpt->m_targets);
1510                if (mpt->m_smp_targets)
1511                        refhash_destroy(mpt->m_smp_targets);

1513                if (mpt->m_active) {
1514                        mptsas_free_active_slots(mpt);
1515                }
1516                if (intr_added) {
1517                        mptsas_unregister_intrs(mpt);
1518                }

1520                if (doneq_thread_create) {
1521                        mutex_enter(&mpt->m_doneq_mutex);
1522                        doneq_thread_num = mpt->m_doneq_thread_n;
1523                        for (j = 0; j < mpt->m_doneq_thread_n; j++) {
1524                                mutex_enter(&mpt->m_doneq_thread_id[j].mutex);
1525                                mpt->m_doneq_thread_id[j].flag &=
1526                                    (~MPTSAS_DONEQ_THREAD_ACTIVE);
1527                                cv_signal(&mpt->m_doneq_thread_id[j].cv);
1528                                mutex_exit(&mpt->m_doneq_thread_id[j].mutex);
```



```
1529                        }
1530                        while (mpt->m_doneq_thread_n) {
1531                                cv_wait(&mpt->m_doneq_thread_cv,
1532                                    &mpt->m_doneq_mutex);
1533                        }
1534                        for (j = 0; j < doneq_thread_num; j++) {
1535                                cv_destroy(&mpt->m_doneq_thread_id[j].cv);
1536                                mutex_destroy(&mpt->m_doneq_thread_id[j].mutex);
1537                        }
1538                        kmem_free(mpt->m_doneq_thread_id,
1539                            sizeof (mptsas_doneq_thread_list_t)
1540                            * doneq_thread_num);
1541                        mutex_exit(&mpt->m_doneq_mutex);
1542                        cv_destroy(&mpt->m_doneq_thread_cv);
1543                        mutex_destroy(&mpt->m_doneq_mutex);
1544                }
1545                if (event_taskq_create) {
1546                        ddi_taskq_destroy(mpt->m_event_taskq);
1547                }
1548                if (dr_taskq_create) {
1549                        ddi_taskq_destroy(mpt->m_dr_taskq);
1550                }
1551                if (mutex_init_done) {
1552                        mutex_destroy(&mpt->m_tx_waitq_mutex);
1553                        mutex_destroy(&mpt->m_passthru_mutex);
1554                        mutex_destroy(&mpt->m_mutex);
1555                        for (i = 0; i < MPTSAS_MAX_PHYS; i++) {
1556                                mutex_destroy(
1557                                    &mpt->m_phy_info[i].smhba_info.phy_mutex);
1558                        }
1559                        cv_destroy(&mpt->m_cv);
1560                        cv_destroy(&mpt->m_passthru_cv);
1561                        cv_destroy(&mpt->m_fw_cv);
1562                        cv_destroy(&mpt->m_config_cv);
1563                        cv_destroy(&mpt->m_fw_diag_cv);
1564                }

1566                if (map_setup) {
1567                        mptsas_cfg_fini(mpt);
1568                }
1569                if (config_setup) {
1570                        mptsas_config_space_fini(mpt);
1571                }
1572                mptsas_free_handshake_msg(mpt);
1573                mptsas_hba_fini(mpt);

1575                mptsas_fm_fini(mpt);
1576                ddi_soft_state_free(mptsas_state, instance);
1577                ddi_prop_remove_all(dip);
1578        }
1579        return (DDI_FAILURE);
1580 }
_____unchanged_portion_omitted_

2233 static int
2234 mptsas_cache_create(mptsas_t *mpt)
2235 {
2236        int instance = mpt->m_instance;
2237        char buf[64];

2239        /*
2240         * create kmem cache for packets
2241         */
2242        (void) sprintf(buf, "mptsas%d_cache", instance);
2243        mpt->m_kmem_cache = kmem_cache_create(buf,
2244            sizeof (struct mptsas_cmd) + scsi_pkt_size(), 16,
```

```
2225             sizeof (struct mptsas_cmd) + scsi_pkt_size(), 8,
2245             mptsas_kmem_cache_constructor, mptsas_kmem_cache_destructor,
2246             NULL, (void *)mpt, NULL, 0);

2248         if (mpt->m_kmem_cache == NULL) {
2249                 mptsas_log(mpt, CE_WARN, "creating kmem cache failed");
2250                 return (FALSE);
2251         }

2253         /*
2254          * create kmem cache for extra SGL frames if SGL cannot
2255          * be accomodated into main request frame.
2256          */
2257         (void) sprintf(buf, "mptsas%d_cache_frames", instance);
2258         mpt->m_cache_frames = kmem_cache_create(buf,
2259             sizeof (mptsas_cache_frames_t), 16,
2240             sizeof (mptsas_cache_frames_t), 8,
2260             mptsas_cache_frames_constructor, mptsas_cache_frames_destructor,
2261             NULL, (void *)mpt, NULL, 0);

2263         if (mpt->m_cache_frames == NULL) {
2264                 mptsas_log(mpt, CE_WARN, "creating cache for frames failed");
2265                 return (FALSE);
2266         }

2268         return (TRUE);
2269 }
______unchanged_portion_omitted_

3942 static int
3943 mptsas_cache_frames_constructor(void *buf, void *cdrarg, int kmflags)
3944 {
3945         mptsas_cache_frames_t   *p = buf;
3946         mptsas_t                *mpt = cdrarg;
3947         ddi_dma_attr_t          frame_dma_attr;
3948         size_t                  mem_size, alloc_len;
3949         ddi_dma_cookie_t        cookie;
3950         uint_t                  ncookie;
3951         int (*callback)(caddr_t) = (kmflags == KM_SLEEP)
3952             ? DDI_DMA_SLEEP: DDI_DMA_DONTWAIT;

3954         frame_dma_attr = mpt->m_msg_dma_attr;
3955         frame_dma_attr.dma_attr_align = 0x10;
3956         frame_dma_attr.dma_attr_sgllen = 1;

3958         if (ddi_dma_alloc_handle(mpt->m_dip, &frame_dma_attr, callback, NULL,
3959             &p->m_dma_hdl) != DDI_SUCCESS) {
3960                 mptsas_log(mpt, CE_WARN, "Unable to allocate dma handle for"
3961                     " extra SGL.");
3962                 return (DDI_FAILURE);
3963         }

3965         mem_size = (mpt->m_max_request_frames - 1) * mpt->m_req_frame_size;

3967         if (ddi_dma_mem_alloc(p->m_dma_hdl, mem_size, &mpt->m_dev_acc_attr,
3968             DDI_DMA_CONSISTENT, callback, NULL, (caddr_t *)&p->m_frames_addr,
3969             &alloc_len, &p->m_acc_hdl) != DDI_SUCCESS) {
3970                 ddi_dma_free_handle(&p->m_dma_hdl);
3971                 p->m_dma_hdl = NULL;
3972                 mptsas_log(mpt, CE_WARN, "Unable to allocate dma memory for"
3973                     " extra SGL.");
3974                 return (DDI_FAILURE);
3975         }

3977         if (ddi_dma_addr_bind_handle(p->m_dma_hdl, NULL, p->m_frames_addr,
3978             alloc_len, DDI_DMA_RDWR | DDI_DMA_CONSISTENT, callback, NULL,
```



```
3979             &cookie, &ncookie) != DDI_DMA_MAPPED) {
3980                 (void) ddi_dma_mem_free(&p->m_acc_hdl);
3981                 ddi_dma_free_handle(&p->m_dma_hdl);
3982                 p->m_dma_hdl = NULL;
3983                 mptsas_log(mpt, CE_WARN, "Unable to bind DMA resources for"
3984                     " extra SGL");
3985                 return (DDI_FAILURE);
3986         }

3988         /*
3989          * Store the SGL memory address.  This chip uses this
3990          * address to dma to and from the driver.  The second
3991          * address is the address mpt uses to fill in the SGL.
3992          */
3993         p->m_phys_addr = cookie.dmac_laddress;
3974         p->m_phys_addr = cookie.dmac_address;

3995         return (DDI_SUCCESS);
3996 }
______unchanged_portion_omitted_

4199 static void
4200 mptsas_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
4201                 ddi_acc_handle_t acc_hdl, uint_t cookiec,
4202                 uint32_t end_flags)
4181 mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd, uint32_t *control,
4182         pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4203 {
4204         pMpi2SGESimple64_t      sge;
4184         uint_t                  cookiec;
4205         mptti_t                 *dmap;
4206         uint32_t                flags;
4187         pMpi2SGESimple64_t      sge;
4188         pMpi2SGEChain64_t       sgechain;
4189         ASSERT(cmd->cmd_flags & CFLAG_DMAVALID);

4208         dmap = cmd->cmd_sg;
4191         /*
4192          * Save the number of entries in the DMA
4193          * Scatter/Gather list
4194          */
4195         cookiec = cmd->cmd_cookiec;

4197         NDBG1(("mptsas_sge_setup: cookiec=%d", cookiec));

4199         /*
4200          * Set read/write bit in control.
4201          */
4202         if (cmd->cmd_flags & CFLAG_DMASEND) {
4203                 *control |= MPI2_SCSIIO_CONTROL_WRITE;
4204         } else {
4205                 *control |= MPI2_SCSIIO_CONTROL_READ;
4206         }

4208         ddi_put32(acc_hdl, &frame->DataLength, cmd->cmd_dmacount);

4210         /*
4211          * We have 2 cases here.  First where we can fit all the
4212          * SG elements into the main frame, and the case
4213          * where we can't.
4214          * If we have more cookies than we can attach to a frame
4215          * we will need to use a chain element to point
4216          * a location of memory where the rest of the S/G
4217          * elements reside.
4218          */
4219         if (cookiec <= MPTSAS_MAX_FRAME_SGES64(mpt)) {
```

```
4220                dmap = cmd->cmd_sg;
4210        sge = (pMpi2SGESimple64_t)(&frame->SGL);
4211        while (cookiec--) {
4212                ddi_put32(acc_hdl, &sge->Address.Low,
4213                    dmap->addr.address64.Low);
4214                ddi_put32(acc_hdl, &sge->Address.High,
4215                    dmap->addr.address64.High);
4216                ddi_put32(acc_hdl, &sge->FlagsLength, dmap->count);
4223                        ddi_put32(acc_hdl,
4224                            &sge->Address.Low, dmap->addr.address64.Low);
4225                        ddi_put32(acc_hdl,
4226                            &sge->Address.High, dmap->addr.address64.High);
4227                        ddi_put32(acc_hdl, &sge->FlagsLength,
4228                            dmap->count);
4217                flags = ddi_get32(acc_hdl, &sge->FlagsLength);
4218                flags |= ((uint32_t)
4219                    (MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4220                    MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4221                    MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4222                    MPI2_SGE_FLAGS_SHIFT);

4224                /*
4225                 * If this is the last cookie, we set the flags
4226                 * to indicate so
4227                 */
4228                if (cookiec == 0) {
4229                        flags |= end_flags;
4241                                flags |=
4242                                    ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT
4243                                    | MPI2_SGE_FLAGS_END_OF_BUFFER
4244                                    | MPI2_SGE_FLAGS_END_OF_LIST) <<
4245                                    MPI2_SGE_FLAGS_SHIFT);
4230                }
4231                if (cmd->cmd_flags & CFLAG_DMASEND) {
4232                        flags |= (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4233                            MPI2_SGE_FLAGS_SHIFT);
4234                } else {
4235                        flags |= (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4236                            MPI2_SGE_FLAGS_SHIFT);
4237                }
4238                ddi_put32(acc_hdl, &sge->FlagsLength, flags);
4239                dmap++;
4240                sge++;
4241        }
4242 }

4244 static void
4245 mptsas_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4246     pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4247 {
4248        pMpi2SGESimple64_t      sge;
4249        pMpi2SGEChain64_t       sgechain;
4250        uint64_t                nframe_phys_addr;
4251        uint_t                  cookiec;
4252        mptti_t                 *dmap;
4253        uint32_t                flags;
4254        int                     i, j, k, l, frames, sgemax;
4255        int                     temp, maxframe_sges;
4256        uint8_t                 chainflags;
4257        uint16_t                chainlength;
4258        mptsas_cache_frames_t   *p;

4260        cookiec = cmd->cmd_cookiec;

4258        } else {
4262        /*
```



```
4263         * Hereby we start to deal with multiple frames.
4264         * The process is as follows:
4265         * 1. Determine how many frames are needed for SGL element
4266         *    storage; Note that all frames are stored in contiguous
4267         *    memory space and in 64-bit DMA mode each element is
4268         *    3 double-words (12 bytes) long.
4269         * 2. Fill up the main frame. We need to do this separately
4270         *    since it contains the SCSI IO request header and needs
4271         *    dedicated processing. Note that the last 4 double-words
4272         *    of the SCSI IO header is for SGL element storage
4273         *    (MPI2_SGE_IO_UNION).
4274         * 3. Fill the chain element in the main frame, so the DMA
4275         *    engine can use the following frames.
4276         * 4. Enter a loop to fill the remaining frames. Note that the
4277         *    last frame contains no chain element.  The remaining
4278         *    frames go into the mpt SGL buffer allocated on the fly,
4279         *    not immediately following the main message frame, as in
4280         *    Gen1.
4281         * Some restrictions:
4282         * 1. For 64-bit DMA, the simple element and chain element
4283         *    are both of 3 double-words (12 bytes) in size, even
4284         *    though all frames are stored in the first 4G of mem
4285         *    range and the higher 32-bits of the address are always 0.
4286         * 2. On some controllers (like the 1064/1068), a frame can
4287         *    hold SGL elements with the last 1 or 2 double-words
4288         *    (4 or 8 bytes) un-used. On these controllers, we should
4289         *    recognize that there's not enough room for another SGL
4290         *    element and move the sge pointer to the next frame.
4291         */
4289                int             i, j, k, l, frames, sgemax;
4290                int             temp;
4291                uint8_t         chainflags;
4292                uint16_t        chainlength;
4293                mptsas_cache_frames_t *p;

4293        /*
4294         * Sgemax is the number of SGE's that will fit
4295         * each extra frame and frames is total
4296         * number of frames we'll need.  1 sge entry per
4297         * frame is reseverd for the chain element thus the -1 below.
4298         */
4299        sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_SGE_SIMPLE64)) - 1);
4300        maxframe_sges = MPTSAS_MAX_FRAME_SGES64(mpt);
4301        temp = (cookiec - (maxframe_sges - 1)) / sgemax;
4301                sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_SGE_SIMPLE64))
4302                    - 1);
4303                temp = (cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) / sgemax;

4303        /*
4304         * A little check to see if we need to round up the number
4305         * of frames we need
4306         */
4307        if ((cookiec - (maxframe_sges - 1)) - (temp * sgemax) > 1) {
4309                if ((cookiec - (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) - (temp *
4310                    sgemax) > 1) {
4308                frames = (temp + 1);
4309        } else {
4310                frames = temp;
4311        }
4312        dmap = cmd->cmd_sg;
4313        sge = (pMpi2SGESimple64_t)(&frame->SGL);

4315        /*
4316         * First fill in the main frame
4317         */
4318        j = maxframe_sges - 1;
```

```
4319         mptsas_sge_mainframe(cmd, frame, acc_hdl, j,
4320             ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4321             MPI2_SGE_FLAGS_SHIFT));
4322         dmap += j;
4323         sge += j;
4324         j++;
4321                 for (j = 1; j < MPTSAS_MAX_FRAME_SGES64(mpt); j++) {
4322                         ddi_put32(acc_hdl, &sge->Address.Low,
4323                             dmap->addr.address64.Low);
4324                         ddi_put32(acc_hdl, &sge->Address.High,
4325                             dmap->addr.address64.High);
4326                         ddi_put32(acc_hdl, &sge->FlagsLength, dmap->count);
4327                         flags = ddi_get32(acc_hdl, &sge->FlagsLength);
4328                         flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4329                             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4330                             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4331                             MPI2_SGE_FLAGS_SHIFT);

4326         /*
4334                          * If this is the last SGE of this frame
4335                          * we set the end of list flag
4336                          */
4337                         if (j == (MPTSAS_MAX_FRAME_SGES64(mpt) - 1)) {
4338                                 flags |= ((uint32_t)
4339                                     (MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4340                                     MPI2_SGE_FLAGS_SHIFT);
4341                         }
4342                         if (cmd->cmd_flags & CFLAG_DMASEND) {
4343                                 flags |=
4344                                     (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4345                                     MPI2_SGE_FLAGS_SHIFT);
4346                         } else {
4347                                 flags |=
4348                                     (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4349                                     MPI2_SGE_FLAGS_SHIFT);
4350                         }
4351                         ddi_put32(acc_hdl, &sge->FlagsLength, flags);
4352                         dmap++;
4353                         sge++;
4354                 }

4356                 /*
4327          * Fill in the chain element in the main frame.
4328          * About calculation on ChainOffset:
4329          * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4330          *    in the end reserved for SGL element storage
4331          *    (MPI2_SGE_IO_UNION); we should count it in our
4332          *    calculation.  See its definition in the header file.
4333          * 2. Constant j is the counter of the current SGL element
4334          *    that will be processed, and (j - 1) is the number of
4335          *    SGL elements that have been processed (stored in the
4336          *    main frame).
4337          * 3. ChainOffset value should be in units of double-words (4
4338          *    bytes) so the last value should be divided by 4.
4339          */
4340         ddi_put8(acc_hdl, &frame->ChainOffset,
4341             (sizeof (MPI2_SCSI_IO_REQUEST) -
4342             sizeof (MPI2_SGE_IO_UNION) +
4343             (j - 1) * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
4344         sgechain = (pMpi2SGEChain64_t)sge;
4345         chainflags = (MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4346             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4347             MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4348         ddi_put8(acc_hdl, &sgechain->Flags, chainflags);

4350         /*
```



```
4351          * The size of the next frame is the accurate size of space
4352          * (in bytes) used to store the SGL elements. j is the counter
4353          * of SGL elements. (j - 1) is the number of SGL elements that
4354          * have been processed (stored in frames).
4355          */
4356         if (frames >= 2) {
4357                 chainlength = mpt->m_req_frame_size /
4358                     sizeof (MPI2_SGE_SIMPLE64) *
4359                     sizeof (MPI2_SGE_SIMPLE64);
4360         } else {
4361                 chainlength = ((cookiec - (j - 1)) *
4362                     sizeof (MPI2_SGE_SIMPLE64));
4363         }

4365         p = cmd->cmd_extra_frames;

4367         ddi_put16(acc_hdl, &sgechain->Length, chainlength);
4368         ddi_put32(acc_hdl, &sgechain->Address.Low,
4369             (p->m_phys_addr&0xffffffffull));
4370         ddi_put32(acc_hdl, &sgechain->Address.High, p->m_phys_addr>>32);
4399             p->m_phys_addr);
4400                 /* SGL is allocated in the first 4G mem range */
4401                 ddi_put32(acc_hdl, &sgechain->Address.High, 0);

4372         /*
4373          * If there are more than 2 frames left we have to
4374          * fill in the next chain offset to the location of
4375          * the chain element in the next frame.
4376          * sgemax is the number of simple elements in an extra
4377          * frame. Note that the value NextChainOffset should be
4378          * in double-words (4 bytes).
4379          */
4380         if (frames >= 2) {
4381                 ddi_put8(acc_hdl, &sgechain->NextChainOffset,
4382                     (sgemax * sizeof (MPI2_SGE_SIMPLE64)) >> 2);
4383         } else {
4384                 ddi_put8(acc_hdl, &sgechain->NextChainOffset, 0);
4385         }

4387         /*
4388          * Jump to next frame;
4389          * Starting here, chain buffers go into the per command SGL.
4390          * This buffer is allocated when chain buffers are needed.
4391          */
4392         sge = (pMpi2SGESimple64_t)p->m_frames_addr;
4393         i = cookiec;

4395         /*
4396          * Start filling in frames with SGE's.  If we
4397          * reach the end of frame and still have SGE's
4398          * to fill we need to add a chain element and
4399          * use another frame.  j will be our counter
4400          * for what cookie we are at and i will be
4401          * the total cookiec. k is the current frame
4402          */
4403         for (k = 1; k <= frames; k++) {
4404                 for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {

4406                         /*
4407                          * If we have reached the end of frame
4408                          * and we have more SGE's to fill in
4409                          * we have to fill the final entry
4410                          * with a chain element and then
4411                          * continue to the next frame
4412                          */
4413                         if ((l == (sgemax + 1)) && (k != frames)) {
```

```
4414                            sgechain = (pMpi2SGEChain64_t)sge;
4415                            j--;
4416                            chainflags = (
4417                                MPI2_SGE_FLAGS_CHAIN_ELEMENT |
4418                                MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4419                                MPI2_SGE_FLAGS_64_BIT_ADDRESSING);
4420                            ddi_put8(p->m_acc_hdl,
4421                                &sgechain->Flags, chainflags);
4422                            /*
4423                             * k is the frame counter and (k + 1)
4424                             * is the number of the next frame.
4425                             * Note that frames are in contiguous
4426                             * memory space.
4427                             */
4428                            nframe_phys_addr = p->m_phys_addr +
4429                                (mpt->m_req_frame_size * k);
4430                            ddi_put32(p->m_acc_hdl,
4431                                &sgechain->Address.Low,
4432                                nframe_phys_addr&0xffffffffull);
4461                                    (p->m_phys_addr +
4462                                    (mpt->m_req_frame_size * k)));
4433                            ddi_put32(p->m_acc_hdl,
4434                                &sgechain->Address.High,
4435                                nframe_phys_addr>>32);
4464                                    &sgechain->Address.High, 0);

4437                            /*
4438                             * If there are more than 2 frames left
4439                             * we have to next chain offset to
4440                             * the location of the chain element
4441                             * in the next frame and fill in the
4442                             * length of the next chain
4443                             */
4444                            if ((frames - k) >= 2) {
4445                                  ddi_put8(p->m_acc_hdl,
4446                                      &sgechain->NextChainOffset,
4447                                      (sgemax *
4448                                      sizeof (MPI2_SGE_SIMPLE64))
4449                                      >> 2);
4450                                  ddi_put16(p->m_acc_hdl,
4451                                      &sgechain->Length,
4452                                      mpt->m_req_frame_size /
4453                                      sizeof (MPI2_SGE_SIMPLE64) *
4454                                      sizeof (MPI2_SGE_SIMPLE64));
4455                            } else {
4456                                  /*
4457                                   * This is the last frame. Set
4458                                   * the NextChainOffset to 0 and
4459                                   * Length is the total size of
4460                                   * all remaining simple elements
4461                                   */
4462                                  ddi_put8(p->m_acc_hdl,
4463                                      &sgechain->NextChainOffset,
4464                                      0);
4465                                  ddi_put16(p->m_acc_hdl,
4466                                      &sgechain->Length,
4467                                      (cookiec - j) *
4468                                      sizeof (MPI2_SGE_SIMPLE64));
4469                            }

4471                            /* Jump to the next frame */
4472                            sge = (pMpi2SGESimple64_t)
4473                                ((char *)p->m_frames_addr +
4474                                (int)mpt->m_req_frame_size * k);

4476                            continue;
```



```
4477                            }

4479                            ddi_put32(p->m_acc_hdl,
4480                                &sge->Address.Low,
4481                                dmap->addr.address64.Low);
4482                            ddi_put32(p->m_acc_hdl,
4483                                &sge->Address.High,
4484                                dmap->addr.address64.High);
4485                            ddi_put32(p->m_acc_hdl,
4486                                &sge->FlagsLength, dmap->count);
4487                            flags = ddi_get32(p->m_acc_hdl,
4488                                &sge->FlagsLength);
4489                            flags |= ((uint32_t)(
4490                                MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
4491                                MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
4492                                MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
4493                                MPI2_SGE_FLAGS_SHIFT);

4495                            /*
4496                             * If we are at the end of the frame and
4497                             * there is another frame to fill in
4498                             * we set the last simple element as last
4499                             * element
4500                             */
4501                            if ((l == sgemax) && (k != frames)) {
4502                                  flags |= ((uint32_t)
4503                                      (MPI2_SGE_FLAGS_LAST_ELEMENT) <<
4504                                      MPI2_SGE_FLAGS_SHIFT);
4505                            }

4507                            /*
4508                             * If this is the final cookie we
4509                             * indicate it by setting the flags
4510                             */
4511                            if (j == i) {
4512                                  flags |= ((uint32_t)
4513                                      (MPI2_SGE_FLAGS_LAST_ELEMENT |
4514                                      MPI2_SGE_FLAGS_END_OF_BUFFER |
4515                                      MPI2_SGE_FLAGS_END_OF_LIST) <<
4516                                      MPI2_SGE_FLAGS_SHIFT);
4517                            }
4518                            if (cmd->cmd_flags & CFLAG_DMASEND) {
4519                                  flags |=
4520                                      (MPI2_SGE_FLAGS_HOST_TO_IOC <<
4521                                      MPI2_SGE_FLAGS_SHIFT);
4522                            } else {
4523                                  flags |=
4524                                      (MPI2_SGE_FLAGS_IOC_TO_HOST <<
4525                                      MPI2_SGE_FLAGS_SHIFT);
4526                            }
4527                            ddi_put32(p->m_acc_hdl,
4528                                &sge->FlagsLength, flags);
4529                            dmap++;
4530                            sge++;
4531                      }
4532                }

4534                /*
4535                 * Sync DMA with the chain buffers that were just created
4536                 */
4537                (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4538        }

4540        static void
4541        mptsas_ieee_sge_mainframe(mptsas_cmd_t *cmd, pMpi2SCSIIORequest_t frame,
4542            ddi_acc_handle_t acc_hdl, uint_t cookiec,
```

```
4543     uint8_t end_flag)
4544 {
4545         pMpi2IeeeSgeSimple64_t  ieeesge;
4546         mptti_t                 *dmap;
4547         uint8_t                 flags;

4549         dmap = cmd->cmd_sg;

4551         NDBG1(("mptsas_ieee_sge_mainframe: cookiec=%d, %s", cookiec,
4552             cmd->cmd_flags & CFLAG_DMASEND?"Out":"In"));

4554         ieeesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);
4555         while (cookiec--) {
4556                 ddi_put32(acc_hdl, &ieeesge->Address.Low,
4557                     dmap->addr.address64.Low);
4558                 ddi_put32(acc_hdl, &ieeesge->Address.High,
4559                     dmap->addr.address64.High);
4560                 ddi_put32(acc_hdl, &ieeesge->Length, dmap->count);
4561                 NDBG1(("mptsas_ieee_sge_mainframe: len=%d", dmap->count));
4562                 flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4563                     MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);

4565                 /*
4566                  * If this is the last cookie, we set the flags
4567                  * to indicate so
4568                  */
4569                 if (cookiec == 0) {
4570                         flags |= end_flag;
4571                 }

4573                 /*
4574                  * XXX: Hmmm, what about the direction based on
4575                  * cmd->cmd_flags & CFLAG_DMASEND?
4576                  */
4577                 ddi_put8(acc_hdl, &ieeesge->Flags, flags);
4578                 dmap++;
4579                 ieeesge++;
4580         }
4581 }

4583 static void
4584 mptsas_ieee_sge_chain(mptsas_t *mpt, mptsas_cmd_t *cmd,
4585     pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4586 {
4587         pMpi2IeeeSgeSimple64_t  ieeesge;
4588         pMpi25IeeeSgeChain64_t  ieeesgechain;
4589         uint64_t                nframe_phys_addr;
4590         uint_t                  cookiec;
4591         mptti_t                 *dmap;
4592         uint8_t                 flags;
4593         int                     i, j, k, l, frames, sgemax;
4594         int                     temp, maxframe_sges;
4595         uint8_t                 chainflags;
4596         uint32_t                chainlength;
4597         mptsas_cache_frames_t   *p;

4599         cookiec = cmd->cmd_cookiec;

4601         NDBG1(("mptsas_ieee_sge_chain: cookiec=%d", cookiec));

4603         /*
4604          * Hereby we start to deal with multiple frames.
4605          * The process is as follows:
4606          * 1. Determine how many frames are needed for SGL element
4607          *    storage; Note that all frames are stored in contiguous
4608          *    memory space and in 64-bit DMA mode each element is
```



```
4609          *    4 double-words (16 bytes) long.
4610          * 2. Fill up the main frame. We need to do this separately
4611          *    since it contains the SCSI IO request header and needs
4612          *    dedicated processing. Note that the last 4 double-words
4613          *    of the SCSI IO header is for SGL element storage
4614          *    (MPI2_SGE_IO_UNION).
4615          * 3. Fill the chain element in the main frame, so the DMA
4616          *    engine can use the following frames.
4617          * 4. Enter a loop to fill the remaining frames. Note that the
4618          *    last frame contains no chain element.  The remaining
4619          *    frames go into the mpt SGL buffer allocated on the fly,
4620          *    not immediately following the main message frame, as in
4621          *    Gen1.
4622          * Some restrictions:
4623          * 1. For 64-bit DMA, the simple element and chain element
4624          *    are both of 4 double-words (16 bytes) in size, even
4625          *    though all frames are stored in the first 4G of mem
4626          *    range and the higher 32-bits of the address are always 0.
4627          * 2. On some controllers (like the 1064/1068), a frame can
4628          *    hold SGL elements with the last 1 or 2 double-words
4629          *    (4 or 8 bytes) un-used. On these controllers, we should
4630          *    recognize that there's not enough room for another SGL
4631          *    element and move the sge pointer to the next frame.
4632          */

4634         /*
4635          * Sgemax is the number of SGE's that will fit
4636          * each extra frame and frames is total
4637          * number of frames we'll need.  1 sge entry per
4638          * frame is reseverd for the chain element thus the -1 below.
4639          */
4640         sgemax = ((mpt->m_req_frame_size / sizeof (MPI2_IEEE_SGE_SIMPLE64))
4641             - 1);
4642         maxframe_sges = MPTSAS_MAX_FRAME_SGES64(mpt);
4643         temp = (cookiec - (maxframe_sges - 1)) / sgemax;

4645         /*
4646          * A little check to see if we need to round up the number
4647          * of frames we need
4648          */
4649         if ((cookiec - (maxframe_sges - 1)) - (temp * sgemax) > 1) {
4650                 frames = (temp + 1);
4651         } else {
4652                 frames = temp;
4653         }
4654         NDBG1(("mptsas_ieee_sge_chain: temp=%d, frames=%d", temp, frames));
4655         dmap = cmd->cmd_sg;
4656         ieeesge = (pMpi2IeeeSgeSimple64_t)(&frame->SGL);

4658         /*
4659          * First fill in the main frame
4660          */
4661         j = maxframe_sges - 1;
4662         mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl, j, 0);
4663         dmap += j;
4664         ieeesge += j;
4665         j++;

4667         /*
4668          * Fill in the chain element in the main frame.
4669          * About calculation on ChainOffset:
4670          * 1. Struct msg_scsi_io_request has 4 double-words (16 bytes)
4671          *    in the end reserved for SGL element storage
4672          *    (MPI2_SGE_IO_UNION); we should count it in our
4673          *    calculation.  See its definition in the header file.
4674          * 2. Constant j is the counter of the current SGL element
```

```
4675          *    that will be processed, and (j - 1) is the number of
4676          *    SGL elements that have been processed (stored in the
4677          *    main frame).
4678          * 3. ChainOffset value should be in units of quad-words (16
4679          *    bytes) so the last value should be divided by 16.
4680          */
4681         ddi_put8(acc_hdl, &frame->ChainOffset,
4682             (sizeof (MPI2_SCSI_IO_REQUEST) -
4683             sizeof (MPI2_SGE_IO_UNION) +
4684             (j - 1) * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4685         ieeesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
4686         chainflags = (MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4687             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4688         ddi_put8(acc_hdl, &ieeesgechain->Flags, chainflags);

4690         /*
4691          * The size of the next frame is the accurate size of space
4692          * (in bytes) used to store the SGL elements. j is the counter
4693          * of SGL elements. (j - 1) is the number of SGL elements that
4694          * have been processed (stored in frames).
4695          */
4696         if (frames >= 2) {
4697                 chainlength = mpt->m_req_frame_size /
4698                     sizeof (MPI2_IEEE_SGE_SIMPLE64) *
4699                     sizeof (MPI2_IEEE_SGE_SIMPLE64);
4700         } else {
4701                 chainlength = ((cookiec - (j - 1)) *
4702                     sizeof (MPI2_IEEE_SGE_SIMPLE64));
4703         }

4705         p = cmd->cmd_extra_frames;

4707         ddi_put32(acc_hdl, &ieeesgechain->Length, chainlength);
4708         ddi_put32(acc_hdl, &ieeesgechain->Address.Low,
4709             p->m_phys_addr&0xffffffffull);
4710         ddi_put32(acc_hdl, &ieeesgechain->Address.High, p->m_phys_addr>>32);

4712         /*
4713          * If there are more than 2 frames left we have to
4714          * fill in the next chain offset to the location of
4715          * the chain element in the next frame.
4716          * sgemax is the number of simple elements in an extra
4717          * frame. Note that the value NextChainOffset should be
4718          * in double-words (4 bytes).
4719          */
4720         if (frames >= 2) {
4721                 ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset,
4722                     (sgemax * sizeof (MPI2_IEEE_SGE_SIMPLE64)) >> 4);
4723         } else {
4724                 ddi_put8(acc_hdl, &ieeesgechain->NextChainOffset, 0);
4725         }

4727         /*
4728          * Jump to next frame;
4729          * Starting here, chain buffers go into the per command SGL.
4730          * This buffer is allocated when chain buffers are needed.
4731          */
4732         ieeesge = (pMpi2IeeeSgeSimple64_t)p->m_frames_addr;
4733         i = cookiec;

4735         /*
4736          * Start filling in frames with SGE's.  If we
4737          * reach the end of frame and still have SGE's
4738          * to fill we need to add a chain element and
4739          * use another frame.  j will be our counter
4740          * for what cookie we are at and i will be
```



```
4741          * the total cookiec. k is the current frame
4742          */
4743         for (k = 1; k <= frames; k++) {
4744                 for (l = 1; (l <= (sgemax + 1)) && (j <= i); j++, l++) {

4746                         /*
4747                          * If we have reached the end of frame
4748                          * and we have more SGE's to fill in
4749                          * we have to fill the final entry
4750                          * with a chain element and then
4751                          * continue to the next frame
4752                          */
4753                         if ((l == (sgemax + 1)) && (k != frames)) {
4754                                 ieeesgechain = (pMpi25IeeeSgeChain64_t)ieeesge;
4755                                 j--;
4756                                 chainflags =
4757                                     MPI2_IEEE_SGE_FLAGS_CHAIN_ELEMENT |
4758                                     MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR;
4759                                 ddi_put8(p->m_acc_hdl,
4760                                     &ieeesgechain->Flags, chainflags);
4761                                 /*
4762                                  * k is the frame counter and (k + 1)
4763                                  * is the number of the next frame.
4764                                  * Note that frames are in contiguous
4765                                  * memory space.
4766                                  */
4767                                 nframe_phys_addr = p->m_phys_addr +
4768                                     (mpt->m_req_frame_size * k);
4769                                 ddi_put32(p->m_acc_hdl,
4770                                     &ieeesgechain->Address.Low,
4771                                     nframe_phys_addr&0xffffffffull);
4772                                 ddi_put32(p->m_acc_hdl,
4773                                     &ieeesgechain->Address.High,
4774                                     nframe_phys_addr>>32);

4776                                 /*
4777                                  * If there are more than 2 frames left
4778                                  * we have to next chain offset to
4779                                  * the location of the chain element
4780                                  * in the next frame and fill in the
4781                                  * length of the next chain
4782                                  */
4783                                 if ((frames - k) >= 2) {
4784                                         ddi_put8(p->m_acc_hdl,
4785                                             &ieeesgechain->NextChainOffset,
4786                                             (sgemax *
4787                                             sizeof (MPI2_IEEE_SGE_SIMPLE64))
4788                                             >> 4);
4789                                         ddi_put32(p->m_acc_hdl,
4790                                             &ieeesgechain->Length,
4791                                             mpt->m_req_frame_size /
4792                                             sizeof (MPI2_IEEE_SGE_SIMPLE64) *
4793                                             sizeof (MPI2_IEEE_SGE_SIMPLE64));
4794                                 } else {
4795                                         /*
4796                                          * This is the last frame. Set
4797                                          * the NextChainOffset to 0 and
4798                                          * Length is the total size of
4799                                          * all remaining simple elements
4800                                          */
4801                                         ddi_put8(p->m_acc_hdl,
4802                                             &ieeesgechain->NextChainOffset,
4803                                             0);
4804                                         ddi_put32(p->m_acc_hdl,
4805                                             &ieeesgechain->Length,
4806                                             (cookiec - j) *
```

```
4807                                            sizeof (MPI2_IEEE_SGE_SIMPLE64));
4808                                }
4809
4810                                /* Jump to the next frame */
4811                                ieeesge = (pMpi2IeeeSgeSimple64_t)
4812                                    ((char *)p->m_frames_addr +
4813                                    (int)mpt->m_req_frame_size * k);
4814
4815                                continue;
4816                        }
4817
4818                        ddi_put32(p->m_acc_hdl,
4819                            &ieeesge->Address.Low,
4820                            dmap->addr.address64.Low);
4821                        ddi_put32(p->m_acc_hdl,
4822                            &ieeesge->Address.High,
4823                            dmap->addr.address64.High);
4824                        ddi_put32(p->m_acc_hdl,
4825                            &ieeesge->Length, dmap->count);
4826                        flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
4827                            MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
4828
4829                        /*
4830                         * If we are at the end of the frame and
4831                         * there is another frame to fill in
4832                         * do we need to do anything?
4833                         * if ((l == sgemax) && (k != frames)) {
4834                         * }
4835                         */
4836
4837                        /*
4838                         * If this is the final cookie set end of list.
4839                         */
4840                        if (j == i) {
4841                                flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
4842                        }
4843
4844                        ddi_put8(p->m_acc_hdl, &ieeesge->Flags, flags);
4845                        dmap++;
4846                        ieeesge++;
4847                }
4848        }
4849
4850        /*
4851         * Sync DMA with the chain buffers that were just created
4852         */
4853        (void) ddi_dma_sync(p->m_dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
4854 }
4855
4856 static void
4857 mptsas_sge_setup(mptsas_t *mpt, mptsas_cmd_t *cmd, uint32_t *control,
4858     pMpi2SCSIIORequest_t frame, ddi_acc_handle_t acc_hdl)
4859 {
4860        ASSERT(cmd->cmd_flags & CFLAG_DMAVALID);
4861
4862        NDBG1(("mptsas_sge_setup: cookiec=%d", cmd->cmd_cookiec));
4863
4864        /*
4865         * Set read/write bit in control.
4866         */
4867        if (cmd->cmd_flags & CFLAG_DMASEND) {
4868                *control |= MPI2_SCSIIO_CONTROL_WRITE;
4869        } else {
4870                *control |= MPI2_SCSIIO_CONTROL_READ;
4871        }
```



```
4873        ddi_put32(acc_hdl, &frame->DataLength, cmd->cmd_dmacount);
4874
4875        /*
4876         * We have 4 cases here.  First where we can fit all the
4877         * SG elements into the main frame, and the case
4878         * where we can't. The SG element is also different when using
4879         * MPI2.5 interface.
4880         * If we have more cookies than we can attach to a frame
4881         * we will need to use a chain element to point
4882         * a location of memory where the rest of the S/G
4883         * elements reside.
4884         */
4885        if (cmd->cmd_cookiec <= MPTSAS_MAX_FRAME_SGES64(mpt)) {
4886                if (mpt->m_MPI25) {
4887                        mptsas_ieee_sge_mainframe(cmd, frame, acc_hdl,
4888                            cmd->cmd_cookiec,
4889                            MPI25_IEEE_SGE_FLAGS_END_OF_LIST);
4890                } else {
4891                        mptsas_sge_mainframe(cmd, frame, acc_hdl,
4892                            cmd->cmd_cookiec,
4893                            ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT
4894                            | MPI2_SGE_FLAGS_END_OF_BUFFER
4895                            | MPI2_SGE_FLAGS_END_OF_LIST) <<
4896                            MPI2_SGE_FLAGS_SHIFT));
4897                }
4898        } else {
4899                if (mpt->m_MPI25) {
4900                        mptsas_ieee_sge_chain(mpt, cmd, frame, acc_hdl);
4901                } else {
4902                        mptsas_sge_chain(mpt, cmd, frame, acc_hdl);
4903                }
4904        }
4905 }
4906
4907 /*
4908  * Interrupt handling
4909  * Utility routine.  Poll for status of a command sent to HBA
4910  * without interrupts (a FLAG_NOINTR command).
4911  */
4912 int
4913 mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime)
4914 {
4915        int     rval = TRUE;
4916
4917        NDBG5(("mptsas_poll: cmd=0x%p", (void *)poll_cmd));
4918
4919        if ((poll_cmd->cmd_flags & CFLAG_TM_CMD) == 0) {
4920                mptsas_restart_hba(mpt);
4921        }
4922
4923        /*
4924         * Wait, using drv_usecwait(), long enough for the command to
4925         * reasonably return from the target if the target isn't
4926         * "dead".  A polled command may well be sent from scsi_poll, and
4927         * there are retries built in to scsi_poll if the transport
4928         * accepted the packet (TRAN_ACCEPT).  scsi_poll waits 1 second
4929         * and retries the transport up to scsi_poll_busycnt times
4930         * (currently 60) if
4931         * 1. pkt_reason is CMD_INCOMPLETE and pkt_state is 0, or
4932         * 2. pkt_reason is CMD_CMPLT and *pkt_scbp has STATUS_BUSY
4933         *
4934         * limit the waiting to avoid a hang in the event that the
4935         * cmd never gets started but we are still receiving interrupts
4936         */
4937        while (!(poll_cmd->cmd_flags & CFLAG_FINISHED)) {
4938                if (mptsas_wait_intr(mpt, polltime) == FALSE) {
```

```
4939                        NDBG5(("mptsas_poll: command incomplete"));
4940                        rval = FALSE;
4941                        break;
4942                }
4943        }

4945        if (rval == FALSE) {

4947                /*
4948                 * this isn't supposed to happen, the hba must be wedged
4949                 * Mark this cmd as a timeout.
4950                 */
4951                mptsas_set_pkt_reason(mpt, poll_cmd, CMD_TIMEOUT,
4952                    (STAT_TIMEOUT|STAT_ABORTED));

4954                if (poll_cmd->cmd_queued == FALSE) {

4956                        NDBG5(("mptsas_poll: not on waitq"));

4958                        poll_cmd->cmd_pkt->pkt_state |=
4959                            (STATE_GOT_BUS|STATE_GOT_TARGET|STATE_SENT_CMD);
4960                } else {

4962                        /* find and remove it from the waitq */
4963                        NDBG5(("mptsas_poll: delete from waitq"));
4964                        mptsas_waitq_delete(mpt, poll_cmd);
4965                }

4967        }
4968        mptsas_fma_check(mpt, poll_cmd);
4969        NDBG5(("mptsas_poll: done"));
4970        return (rval);
4971 }
```

*_____unchanged_portion_omitted_*

```
5815 static void
5816 mptsas_process_intr(mptsas_t *mpt,
5817     pMpi2ReplyDescriptorsUnion_t reply_desc_union)
5818 {
5819        uint8_t reply_type;

5821        ASSERT(mutex_owned(&mpt->m_mutex));

5823        /*
5824         * The reply is valid, process it according to its
5825         * type.  Also, set a flag for updated the reply index
5826         * after they've all been processed.
5827         */
5828        reply_type = ddi_get8(mpt->m_acc_post_queue_hdl,
5829            &reply_desc_union->Default.ReplyFlags);
5830        reply_type &= MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;
5831        if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS ||
5832            reply_type == MPI25_RPY_DESCRIPT_FLAGS_FAST_PATH_SCSI_IO_SUCCESS) {
5494        if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS) {
5833                mptsas_handle_scsi_io_success(mpt, reply_desc_union);
5834        } else if (reply_type == MPI2_RPY_DESCRIPT_FLAGS_ADDRESS_REPLY) {
5835                mptsas_handle_address_reply(mpt, reply_desc_union);
5836        } else {
5837                mptsas_log(mpt, CE_WARN, "?Bad reply type %x", reply_type);
5838                ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
5839        }

5841        /*
5842         * Clear the reply descriptor for re-use and increment
5843         * index.
5844         */
```



```
5845        ddi_put64(mpt->m_acc_post_queue_hdl,
5846            &((uint64_t *)(void *)mpt->m_post_queue)[mpt->m_post_index],
5847            0xFFFFFFFFFFFFFFFF);
5848        (void) ddi_dma_sync(mpt->m_dma_post_queue_hdl, 0, 0,
5849            DDI_DMA_SYNC_FORDEV);
5850 }
```

*_____unchanged_portion_omitted_*

```
9629 #ifdef MPTSAS_DEBUG
9630 /*PRINTFLIKE1*/
9631 void
9632 mptsas_printf(char *fmt, ...)
9633 {
9634        dev_info_t      *dev = 0;
9635        va_list         ap;

9637        mutex_enter(&mptsas_log_mutex);

9639        va_start(ap, fmt);
9640        (void) vsprintf(mptsas_log_buf, fmt, ap);
9641        va_end(ap);

9643 #ifdef PROM_PRINTF
9644        prom_printf("%s:\t%s\n", mptsas_label, mptsas_log_buf);
9645 #else
9646        scsi_log(dev, mptsas_label, CE_CONT, "!%s\n", mptsas_log_buf);
9308        scsi_log(dev, mptsas_label, SCSI_DEBUG, "%s\n", mptsas_log_buf);
9647 #endif
9648        mutex_exit(&mptsas_log_mutex);
9649 }
```

*_____unchanged_portion_omitted_*

```
10007 static void
10008 mptsas_passthru_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10009     pMpi2SGESimple64_t sgep)
9670 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
10010 {
10011        uint32_t                sge_flags;
10012        uint32_t                data_size, dataout_size;
9672        caddr_t                 memp;
9673        pMPI2RequestHeader_t    request_hdrp;
9674        struct scsi_pkt         *pkt = cmd->cmd_pkt;
9675        mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
9676        uint32_t                request_size, data_size, dataout_size;
9677        uint32_t                direction;
10013        ddi_dma_cookie_t        data_cookie;
10014        ddi_dma_cookie_t        dataout_cookie;
9680        uint32_t                request_desc_low, request_desc_high = 0;
9681        uint32_t                i, sense_bufp;
9682        uint8_t                 desc_type;
9683        uint8_t                 *request, function;
9684        ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
9685        ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

9687        desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

9689        request = pt->request;
9690        direction = pt->direction;
9691        request_size = pt->request_size;
10016        data_size = pt->data_size;
10017        dataout_size = pt->dataout_size;
10018        data_cookie = pt->data_cookie;
10019        dataout_cookie = pt->dataout_cookie;

9697        /*
9698         * Store the passthrough message in memory location
```

```
9699          * corresponding to our slot number
9700          */
9701         memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
9702         request_hdrp = (pMPI2RequestHeader_t)memp;
9703         bzero(memp, mpt->m_req_frame_size);

9705         for (i = 0; i < request_size; i++) {
9706                 bcopy(request + i, memp + i, 1);
9707         }

9709         if (data_size || dataout_size) {
9710                 pMpi2SGESimple64_t      sgep;
9711                 uint32_t                sge_flags;

9713                 sgep = (pMpi2SGESimple64_t)((uint8_t *)request_hdrp +
9714                     request_size);
10021         if (dataout_size) {

10022                 sge_flags = dataout_size |
10023                     ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10024                     MPI2_SGE_FLAGS_END_OF_BUFFER |
10025                     MPI2_SGE_FLAGS_HOST_TO_IOC |
10026                     MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10027                     MPI2_SGE_FLAGS_SHIFT);
10028                 ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
10029                 ddi_put32(acc_hdl, &sgep->Address.Low,
10030                     (uint32_t)(dataout_cookie.dmac_laddress & 0xffffffffull));
9725                         (uint32_t)(dataout_cookie.dmac_laddress &
9726                         0xffffffffull));
10031                 ddi_put32(acc_hdl, &sgep->Address.High,
10032                     (uint32_t)(dataout_cookie.dmac_laddress >> 32));
9728                         (uint32_t)(dataout_cookie.dmac_laddress
9729                         >> 32));
10033                 sgep++;
10034         }
10035         sge_flags = data_size;
10036         sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
10037             MPI2_SGE_FLAGS_LAST_ELEMENT |
10038             MPI2_SGE_FLAGS_END_OF_BUFFER |
10039             MPI2_SGE_FLAGS_END_OF_LIST |
10040             MPI2_SGE_FLAGS_64_BIT_ADDRESSING) <<
10041             MPI2_SGE_FLAGS_SHIFT);
10042         if (pt->direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
9739                 if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10043                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_HOST_TO_IOC) <<
10044                     MPI2_SGE_FLAGS_SHIFT);
10045         } else {
10046                 sge_flags |= ((uint32_t)(MPI2_SGE_FLAGS_IOC_TO_HOST) <<
10047                     MPI2_SGE_FLAGS_SHIFT);
10048         }
10049         ddi_put32(acc_hdl, &sgep->FlagsLength, sge_flags);
9746                 ddi_put32(acc_hdl, &sgep->FlagsLength,
9747                     sge_flags);
10050         ddi_put32(acc_hdl, &sgep->Address.Low,
10051             (uint32_t)(data_cookie.dmac_laddress & 0xffffffffull));
9749                     (uint32_t)(data_cookie.dmac_laddress &
9750                     0xffffffffull));
10052         ddi_put32(acc_hdl, &sgep->Address.High,
10053             (uint32_t)(data_cookie.dmac_laddress >> 32));
10054 }

10056 static void
10057 mptsas_passthru_ieee_sge(ddi_acc_handle_t acc_hdl, mptsas_pt_request_t *pt,
10058     pMpi2IeeeSgeSimple64_t ieeesgep)
10059 {
10060         uint8_t                 sge_flags;
```



```
10061         uint32_t                data_size, dataout_size;
10062         ddi_dma_cookie_t        data_cookie;
10063         ddi_dma_cookie_t        dataout_cookie;

10065         data_size = pt->data_size;
10066         dataout_size = pt->dataout_size;
10067         data_cookie = pt->data_cookie;
10068         dataout_cookie = pt->dataout_cookie;

10070         sge_flags = (MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
10071             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR);
10072         if (dataout_size) {
10073                 ddi_put32(acc_hdl, &ieeesgep->Length, dataout_size);
10074                 ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10075                     (uint32_t)(dataout_cookie.dmac_laddress &
10076                     0xffffffffull));
10077                 ddi_put32(acc_hdl, &ieeesgep->Address.High,
10078                     (uint32_t)(dataout_cookie.dmac_laddress >> 32));
10079                 ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
10080                 ieeesgep++;
10081         }
10082         sge_flags |= MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
10083         ddi_put32(acc_hdl, &ieeesgep->Length, data_size);
10084         ddi_put32(acc_hdl, &ieeesgep->Address.Low,
10085             (uint32_t)(data_cookie.dmac_laddress & 0xffffffffull));
10086         ddi_put32(acc_hdl, &ieeesgep->Address.High,
10087             (uint32_t)(data_cookie.dmac_laddress >> 32));
10088         ddi_put8(acc_hdl, &ieeesgep->Flags, sge_flags);
10089 }

10091 static void
10092 mptsas_start_passthru(mptsas_t *mpt, mptsas_cmd_t *cmd)
10093 {
10094         caddr_t                 memp;
10095         pMPI2RequestHeader_t    request_hdrp;
10096         struct scsi_pkt         *pkt = cmd->cmd_pkt;
10097         mptsas_pt_request_t     *pt = pkt->pkt_ha_private;
10098         uint32_t                request_size;
10099         uint32_t                request_desc_low, request_desc_high = 0;
10100         uint64_t                sense_bufp;
10101         uint8_t                 desc_type;
10102         uint8_t                 *request, function;
10103         ddi_dma_handle_t        dma_hdl = mpt->m_dma_req_frame_hdl;
10104         ddi_acc_handle_t        acc_hdl = mpt->m_acc_req_frame_hdl;

10106         desc_type = MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;

10108         request = pt->request;
10109         request_size = pt->request_size;

10111         /*
10112          * Store the passthrough message in memory location
10113          * corresponding to our slot number
10114          */
10115         memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
10116         request_hdrp = (pMPI2RequestHeader_t)memp;
10117         bzero(memp, mpt->m_req_frame_size);

10119         bcopy(request, memp, request_size);

10121         NDBG15(("mptsas_start_passthru: Func 0x%x, MsgFlags 0x%x, "
10122             "size=%d, in %d, out %d", request_hdrp->Function,
10123             request_hdrp->MsgFlags, request_size,
10124             pt->data_size, pt->dataout_size));

10126         /*
```

```
10127          * Add an SGE, even if the length is zero.
10128          */
10129         if (mpt->m_MPI25 && pt->simple == 0) {
10130                 mptsas_passthru_ieee_sge(acc_hdl, pt,
10131                     (pMpi2IeeeSgeSimple64_t)
10132                     ((uint8_t *)request_hdrp + pt->sgl_offset));
10133         } else {
10134                 mptsas_passthru_sge(acc_hdl, pt,
10135                     (pMpi2SGESimple64_t)
10136                     ((uint8_t *)request_hdrp + pt->sgl_offset));
10137         }

10139         function = request_hdrp->Function;
10140         if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10141             (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10142                 pMpi2SCSIIORequest_t    scsi_io_req;

10144                 NDBG15(("mptsas_start_passthru: Is SCSI IO Req"));
10145                 scsi_io_req = (pMpi2SCSIIORequest_t)request_hdrp;
10146                 /*
10147                  * Put SGE for data and data_out buffer at the end of
10148                  * scsi_io_request message header.(64 bytes in total)
10149                  * Following above SGEs, the residual space will be
10150                  * used by sense data.
10151                  */
10152                 ddi_put8(acc_hdl,
10153                     &scsi_io_req->SenseBufferLength,
10154                     (uint8_t)(request_size - 64));

10156                 sense_bufp = (uint32_t)(mpt->m_req_frame_dma_addr +
10157                     (mpt->m_req_frame_size * cmd->cmd_slot) & 0xffffffffull);
9771                sense_bufp = mpt->m_req_frame_dma_addr +
9772                    (mpt->m_req_frame_size * cmd->cmd_slot);
10158                 sense_bufp += 64;
10159                 ddi_put32(acc_hdl,
10160                     &scsi_io_req->SenseBufferLowAddress, sense_bufp);

10162                 /*
10163                  * Set SGLOffset0 value
10164                  */
10165                 ddi_put8(acc_hdl, &scsi_io_req->SGLOffset0,
10166                     offsetof(MPI2_SCSI_IO_REQUEST, SGL) / 4);

10168                 /*
10169                  * Setup descriptor info.  RAID passthrough must use the
10170                  * default request descriptor which is already set, so if this
10171                  * is a SCSI IO request, change the descriptor to SCSI IO.
10172                  */
10173                 if (function == MPI2_FUNCTION_SCSI_IO_REQUEST) {
10174                         desc_type = MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO;
10175                         request_desc_high = (ddi_get16(acc_hdl,
10176                             &scsi_io_req->DevHandle) << 16);
10177                 }
10178         }

10180         /*
10181          * We must wait till the message has been completed before
10182          * beginning the next message so we wait for this one to
10183          * finish.
10184          */
10185         (void) ddi_dma_sync(dma_hdl, 0, 0, DDI_DMA_SYNC_FORDEV);
10186         request_desc_low = (cmd->cmd_slot << 16) + desc_type;
10187         cmd->cmd_rfm = NULL;
10188         MPTSAS_START_CMD(mpt, request_desc_low, request_desc_high);
10189         if ((mptsas_check_dma_handle(dma_hdl) != DDI_SUCCESS) ||
10190             (mptsas_check_acc_handle(acc_hdl) != DDI_SUCCESS)) {
```



```
10191                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
10192         }
10193 }

10195 typedef void (mptsas_pre_f)(mptsas_t *, mptsas_pt_request_t *);
10196 static mptsas_pre_f     mpi_pre_ioc_facts;
10197 static mptsas_pre_f     mpi_pre_port_facts;
10198 static mptsas_pre_f     mpi_pre_fw_download;
10199 static mptsas_pre_f     mpi_pre_fw_25_download;
10200 static mptsas_pre_f     mpi_pre_fw_upload;
10201 static mptsas_pre_f     mpi_pre_fw_25_upload;
10202 static mptsas_pre_f     mpi_pre_sata_passthrough;
10203 static mptsas_pre_f     mpi_pre_smp_passthrough;
10204 static mptsas_pre_f     mpi_pre_config;
10205 static mptsas_pre_f     mpi_pre_sas_io_unit_control;
10206 static mptsas_pre_f     mpi_pre_scsi_io_req;

10208 /*
10209  * Prepare the pt for a SAS2 FW_DOWNLOAD request.
10210  */
10211 static void
10212 mpi_pre_fw_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
10213 {
10214         pMpi2FWDownloadTCSGE_t tcsge;
10215         pMpi2FWDownloadRequest req;

10217         /*
10218          * If SAS3, call separate function.
10219          */
10220         if (mpt->m_MPI25) {
10221                 mpi_pre_fw_25_download(mpt, pt);
10222                 return;
10223         }

10225         /*
10226          * User requests should come in with the Transaction
10227          * context element where the SGL will go. Putting the
10228          * SGL after that seems to work, but don't really know
10229          * why. Other drivers tend to create an extra SGL and
10230          * refer to the TCE through that.
10231          */
10232         req = (pMpi2FWDownloadRequest)pt->request;
10233         tcsge = (pMpi2FWDownloadTCSGE_t)&req->SGL;
10234         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10235             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10236                 mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10237         }

10239         pt->sgl_offset = offsetof(MPI2_FW_DOWNLOAD_REQUEST, SGL) +
10240             sizeof (*tcsge);
10241         if (pt->request_size != pt->sgl_offset)
10242                 NDBG15(("mpi_pre_fw_download(): Incorrect req size, "
10243                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10244                     (int)pt->request_size, (int)pt->sgl_offset,
10245                     (int)pt->dataout_size));
10246         if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10247                 NDBG15(("mpi_pre_fw_download(): Incorrect rep size, "
10248                     "0x%x, should be 0x%x", pt->data_size,
10249                     (int)sizeof (MPI2_FW_DOWNLOAD_REPLY)));
10250 }

10252 /*
10253  * Prepare the pt for a SAS3 FW_DOWNLOAD request.
10254  */
10255 static void
10256 mpi_pre_fw_25_download(mptsas_t *mpt, mptsas_pt_request_t *pt)
```

```
10257 {
10258         pMpi2FWDownloadTCSGE_t tcsge;
10259         pMpi2FWDownloadRequest req2;
10260         pMpi25FWDownloadRequest req25;

10262         /*
10263          * User requests should come in with the Transaction
10264          * context element where the SGL will go. The new firmware
10265          * Doesn't use TCE and has space in the main request for
10266          * this information. So move to the right place.
10267          */
10268         req2 = (pMpi2FWDownloadRequest)pt->request;
10269         req25 = (pMpi25FWDownloadRequest)pt->request;
10270         tcsge = (pMpi2FWDownloadTCSGE_t)&req2->SGL;
10271         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10272             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10273                 mptsas_log(mpt, CE_WARN, "FW Download tce invalid!");
10274         }
10275         req25->ImageOffset = tcsge->ImageOffset;
10276         req25->ImageSize = tcsge->ImageSize;

10278         pt->sgl_offset = offsetof(MPI25_FW_DOWNLOAD_REQUEST, SGL);
10279         if (pt->request_size != pt->sgl_offset)
10280                 NDBG15(("mpi_pre_fw_25_download(): Incorrect req size, "
10281                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10282                     pt->request_size, pt->sgl_offset,
10283                     pt->dataout_size));
10284         if (pt->data_size < sizeof (MPI2_FW_DOWNLOAD_REPLY))
10285                 NDBG15(("mpi_pre_fw_25_download(): Incorrect rep size, "
10286                     "0x%x, should be 0x%x", pt->data_size,
10287                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10288 }

10290 /*
10291  * Prepare the pt for a SAS2 FW_UPLOAD request.
10292  */
10293 static void
10294 mpi_pre_fw_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10295 {
10296         pMpi2FWUploadTCSGE_t tcsge;
10297         pMpi2FWUploadRequest_t req;

10299         /*
10300          * If SAS3, call separate function.
10301          */
10302         if (mpt->m_MPI25) {
10303                 mpi_pre_fw_25_upload(mpt, pt);
10304                 return;
10305         }

10307         /*
10308          * User requests should come in with the Transaction
10309          * context element where the SGL will go. Putting the
10310          * SGL after that seems to work, but don't really know
10311          * why. Other drivers tend to create an extra SGL and
10312          * refer to the TCE through that.
10313          */
10314         req = (pMpi2FWUploadRequest_t)pt->request;
10315         tcsge = (pMpi2FWUploadTCSGE_t)&req->SGL;
10316         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10317             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10318                 mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10319         }

10321         pt->sgl_offset = offsetof(MPI2_FW_UPLOAD_REQUEST, SGL) +
10322             sizeof (*tcsge);
```



```
10323         if (pt->request_size != pt->sgl_offset)
10324                 NDBG15(("mpi_pre_fw_upload(): Incorrect req size, "
10325                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10326                     pt->request_size, pt->sgl_offset,
10327                     pt->dataout_size));
10328         if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10329                 NDBG15(("mpi_pre_fw_upload(): Incorrect rep size, "
10330                     "0x%x, should be 0x%x", pt->data_size,
10331                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10332 }

10334 /*
10335  * Prepare the pt a SAS3 FW_UPLOAD request.
10336  */
10337 static void
10338 mpi_pre_fw_25_upload(mptsas_t *mpt, mptsas_pt_request_t *pt)
10339 {
10340         pMpi2FWUploadTCSGE_t tcsge;
10341         pMpi2FWUploadRequest_t req2;
10342         pMpi25FWUploadRequest_t req25;

10344         /*
10345          * User requests should come in with the Transaction
10346          * context element where the SGL will go. The new firmware
10347          * Doesn't use TCE and has space in the main request for
10348          * this information. So move to the right place.
10349          */
10350         req2 = (pMpi2FWUploadRequest_t)pt->request;
10351         req25 = (pMpi25FWUploadRequest_t)pt->request;
10352         tcsge = (pMpi2FWUploadTCSGE_t)&req2->SGL;
10353         if (tcsge->ContextSize != 0 || tcsge->DetailsLength != 12 ||
10354             tcsge->Flags != MPI2_SGE_FLAGS_TRANSACTION_ELEMENT) {
10355                 mptsas_log(mpt, CE_WARN, "FW Upload tce invalid!");
10356         }
10357         req25->ImageOffset = tcsge->ImageOffset;
10358         req25->ImageSize = tcsge->ImageSize;

10360         pt->sgl_offset = offsetof(MPI25_FW_UPLOAD_REQUEST, SGL);
10361         if (pt->request_size != pt->sgl_offset)
10362                 NDBG15(("mpi_pre_fw_25_upload(): Incorrect req size, "
10363                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10364                     pt->request_size, pt->sgl_offset,
10365                     pt->dataout_size));
10366         if (pt->data_size < sizeof (MPI2_FW_UPLOAD_REPLY))
10367                 NDBG15(("mpi_pre_fw_25_upload(): Incorrect rep size, "
10368                     "0x%x, should be 0x%x", pt->data_size,
10369                     (int)sizeof (MPI2_FW_UPLOAD_REPLY)));
10370 }

10372 /*
10373  * Prepare the pt for an IOC_FACTS request.
10374  */
10375 static void
10376 mpi_pre_ioc_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10377 {
10378 #ifndef __lock_lint
10379         _NOTE(ARGUNUSED(mpt))
10380 #endif
10381         if (pt->request_size != sizeof (MPI2_IOC_FACTS_REQUEST))
10382                 NDBG15(("mpi_pre_ioc_facts(): Incorrect req size, "
10383                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10384                     pt->request_size,
10385                     (int)sizeof (MPI2_IOC_FACTS_REQUEST),
10386                     pt->dataout_size));
10387         if (pt->data_size != sizeof (MPI2_IOC_FACTS_REPLY))
10388                 NDBG15(("mpi_pre_ioc_facts(): Incorrect rep size, "
```

```
10389                     "0x%x, should be 0x%x", pt->data_size,
10390                     (int)sizeof (MPI2_IOC_FACTS_REPLY)));
10391         pt->sgl_offset = (uint16_t)pt->request_size;
10392 }

10394 /*
10395  * Prepare the pt for a PORT_FACTS request.
10396  */
10397 static void
10398 mpi_pre_port_facts(mptsas_t *mpt, mptsas_pt_request_t *pt)
10399 {
10400 #ifndef __lock_lint
10401         _NOTE(ARGUNUSED(mpt))
10402 #endif
10403         if (pt->request_size != sizeof (MPI2_PORT_FACTS_REQUEST))
10404                 NDBG15(("mpi_pre_port_facts(): Incorrect req size, "
10405                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10406                     pt->request_size,
10407                     (int)sizeof (MPI2_PORT_FACTS_REQUEST),
10408                     pt->dataout_size));
10409         if (pt->data_size != sizeof (MPI2_PORT_FACTS_REPLY))
10410                 NDBG15(("mpi_pre_port_facts(): Incorrect rep size, "
10411                     "0x%x, should be 0x%x", pt->data_size,
10412                     (int)sizeof (MPI2_PORT_FACTS_REPLY)));
10413         pt->sgl_offset = (uint16_t)pt->request_size;
10414 }

10416 /*
10417  * Prepare pt for a SATA_PASSTHROUGH request.
10418  */
10419 static void
10420 mpi_pre_sata_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10421 {
10422 #ifndef __lock_lint
10423         _NOTE(ARGUNUSED(mpt))
10424 #endif
10425         pt->sgl_offset = offsetof(MPI2_SATA_PASSTHROUGH_REQUEST, SGL);
10426         if (pt->request_size != pt->sgl_offset)
10427                 NDBG15(("mpi_pre_sata_passthrough(): Incorrect req size, "
10428                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10429                     pt->request_size, pt->sgl_offset,
10430                     pt->dataout_size));
10431         if (pt->data_size != sizeof (MPI2_SATA_PASSTHROUGH_REPLY))
10432                 NDBG15(("mpi_pre_sata_passthrough(): Incorrect rep size, "
10433                     "0x%x, should be 0x%x", pt->data_size,
10434                     (int)sizeof (MPI2_SATA_PASSTHROUGH_REPLY)));
10435 }

10437 static void
10438 mpi_pre_smp_passthrough(mptsas_t *mpt, mptsas_pt_request_t *pt)
10439 {
10440 #ifndef __lock_lint
10441         _NOTE(ARGUNUSED(mpt))
10442 #endif
10443         pt->sgl_offset = offsetof(MPI2_SMP_PASSTHROUGH_REQUEST, SGL);
10444         if (pt->request_size != pt->sgl_offset)
10445                 NDBG15(("mpi_pre_smp_passthrough(): Incorrect req size, "
10446                     "0x%x, should be 0x%x, dataoutsz 0x%x",
10447                     pt->request_size, pt->sgl_offset,
10448                     pt->dataout_size));
10449         if (pt->data_size != sizeof (MPI2_SMP_PASSTHROUGH_REPLY))
10450                 NDBG15(("mpi_pre_smp_passthrough(): Incorrect rep size, "
10451                     "0x%x, should be 0x%x", pt->data_size,
10452                     (int)sizeof (MPI2_SMP_PASSTHROUGH_REPLY)));
10453 }
```



```
10455 /*
10456  * Prepare pt for a CONFIG request.
10457  */
10458 static void
10459 mpi_pre_config(mptsas_t *mpt, mptsas_pt_request_t *pt)
10460 {
10461 #ifndef __lock_lint
10462         _NOTE(ARGUNUSED(mpt))
10463 #endif
10464         pt->sgl_offset = offsetof(MPI2_CONFIG_REQUEST, PageBufferSGE);
10465         if (pt->request_size != pt->sgl_offset)
10466                 NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10467                     "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10468                     pt->sgl_offset, pt->dataout_size));
10469         if (pt->data_size != sizeof (MPI2_CONFIG_REPLY))
10470                 NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10471                     "should be 0x%x", pt->data_size,
10472                     (int)sizeof (MPI2_CONFIG_REPLY)));
10473         pt->simple = 1;
10474 }

10476 /*
10477  * Prepare pt for a SCSI_IO_REQ request.
10478  */
10479 static void
10480 mpi_pre_scsi_io_req(mptsas_t *mpt, mptsas_pt_request_t *pt)
10481 {
10482 #ifndef __lock_lint
10483         _NOTE(ARGUNUSED(mpt))
10484 #endif
10485         pt->sgl_offset = offsetof(MPI2_SCSI_IO_REQUEST, SGL);
10486         if (pt->request_size != pt->sgl_offset)
10487                 NDBG15(("mpi_pre_config(): Incorrect req size, 0x%x, "
10488                     "should be 0x%x, dataoutsz 0x%x", pt->request_size,
10489                     pt->sgl_offset,
10490                     pt->dataout_size));
10491         if (pt->data_size != sizeof (MPI2_SCSI_IO_REPLY))
10492                 NDBG15(("mpi_pre_config(): Incorrect rep size, 0x%x, "
10493                     "should be 0x%x", pt->data_size,
10494                     (int)sizeof (MPI2_SCSI_IO_REPLY)));
10495 }

10497 /*
10498  * Prepare the mptsas_cmd for a SAS_IO_UNIT_CONTROL request.
10499  */
10500 static void
10501 mpi_pre_sas_io_unit_control(mptsas_t *mpt, mptsas_pt_request_t *pt)
10502 {
10503 #ifndef __lock_lint
10504         _NOTE(ARGUNUSED(mpt))
10505 #endif
10506         pt->sgl_offset = (uint16_t)pt->request_size;
10507 }

10509 /*
10510  * A set of functions to prepare an mptsas_cmd for the various
10511  * supported requests.
10512  */
10513 static struct mptsas_func {
10514         U8              Function;
10515         char            *Name;
10516         mptsas_pre_f    *f_pre;
10517 } mptsas_func_list[] = {
10518         { MPI2_FUNCTION_IOC_FACTS, "IOC_FACTS",         mpi_pre_ioc_facts },
10519         { MPI2_FUNCTION_PORT_FACTS, "PORT_FACTS",       mpi_pre_port_facts },
10520         { MPI2_FUNCTION_FW_DOWNLOAD, "FW_DOWNLOAD",     mpi_pre_fw_download },
```

```
10521         { MPI2_FUNCTION_FW_UPLOAD, "FW_UPLOAD",          mpi_pre_fw_upload },
10522         { MPI2_FUNCTION_SATA_PASSTHROUGH, "SATA_PASSTHROUGH",
10523             mpi_pre_sata_passthrough },
10524         { MPI2_FUNCTION_SMP_PASSTHROUGH, "SMP_PASSTHROUGH",
10525             mpi_pre_smp_passthrough},
10526         { MPI2_FUNCTION_SCSI_IO_REQUEST, "SCSI_IO_REQUEST",
10527             mpi_pre_scsi_io_req},
10528         { MPI2_FUNCTION_CONFIG, "CONFIG",                mpi_pre_config},
10529         { MPI2_FUNCTION_SAS_IO_UNIT_CONTROL, "SAS_IO_UNIT_CONTROL",
10530             mpi_pre_sas_io_unit_control },
10531         { 0xFF, NULL,                               NULL } /* list end */
10532 };

10534 static void
10535 mptsas_prep_sgl_offset(mptsas_t *mpt, mptsas_pt_request_t *pt)
10536 {
10537         pMPI2RequestHeader_t    hdr;
10538         struct mptsas_func      *f;

10540         hdr = (pMPI2RequestHeader_t)pt->request;

10542         for (f = mptsas_func_list; f->f_pre != NULL; f++) {
10543                 if (hdr->Function == f->Function) {
10544                         f->f_pre(mpt, pt);
10545                         NDBG15(("mptsas_prep_sgl_offset: Function %s,"
10546                             " sgl_offset 0x%x", f->Name,
10547                             pt->sgl_offset));
10548                         return;
10549                 }
10550         }
10551         NDBG15(("mptsas_prep_sgl_offset: Unknown Function 0x%02x,"
10552             " returning req_size 0x%x for sgl_offset",
10553             hdr->Function, pt->request_size));
10554         pt->sgl_offset = (uint16_t)pt->request_size;
10555 }


10558 static int
10559 mptsas_do_passthru(mptsas_t *mpt, uint8_t *request, uint8_t *reply,
10560     uint8_t *data, uint32_t request_size, uint32_t reply_size,
10561     uint32_t data_size, uint8_t direction, uint8_t *dataout,
9815     uint32_t data_size, uint32_t direction, uint8_t *dataout,
10562     uint32_t dataout_size, short timeout, int mode)
10563 {
10564         mptsas_pt_request_t             pt;
10565         mptsas_dma_alloc_state_t        data_dma_state;
10566         mptsas_dma_alloc_state_t        dataout_dma_state;
10567         caddr_t                         memp;
10568         mptsas_cmd_t                    *cmd = NULL;
10569         struct scsi_pkt                 *pkt;
10570         uint32_t                        reply_len = 0, sense_len = 0;
10571         pMPI2RequestHeader_t            request_hdrp;
10572         pMPI2RequestHeader_t            request_msg;
10573         pMPI2DefaultReply_t             reply_msg;
10574         Mpi2SCSIIOReply_t               rep_msg;
10575         int                             i, status = 0, pt_flags = 0, rv = 0;
10576         int                             rvalue;
10577         uint8_t                         function;

10579         ASSERT(mutex_owned(&mpt->m_mutex));

10581         reply_msg = (pMPI2DefaultReply_t)(&rep_msg);
10582         bzero(reply_msg, sizeof (MPI2_DEFAULT_REPLY));
10583         request_msg = kmem_zalloc(request_size, KM_SLEEP);

10585         mutex_exit(&mpt->m_mutex);
```



```
10586         /*
10587          * copy in the request buffer since it could be used by
10588          * another thread when the pt request into waitq
10589          */
10590         if (ddi_copyin(request, request_msg, request_size, mode)) {
10591                 mutex_enter(&mpt->m_mutex);
10592                 status = EFAULT;
10593                 mptsas_log(mpt, CE_WARN, "failed to copy request data");
10594                 goto out;
10595         }
10596         mutex_enter(&mpt->m_mutex);

10598         function = request_msg->Function;
10599         if (function == MPI2_FUNCTION_SCSI_TASK_MGMT) {
10600                 pMpi2SCSITaskManagementRequest_t        task;
10601                 task = (pMpi2SCSITaskManagementRequest_t)request_msg;
10602                 mptsas_setup_bus_reset_delay(mpt);
10603                 rv = mptsas_ioc_task_management(mpt, task->TaskType,
10604                     task->DevHandle, (int)task->LUN[1], reply, reply_size,
10605                     mode);

10607                 if (rv != TRUE) {
10608                         status = EIO;
10609                         mptsas_log(mpt, CE_WARN, "task management failed");
10610                 }
10611                 goto out;
10612         }

10614         if (data_size != 0) {
10615                 data_dma_state.size = data_size;
10616                 if (mptsas_dma_alloc(mpt, &data_dma_state) != DDI_SUCCESS) {
10617                         status = ENOMEM;
10618                         mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10619                             "resource");
10620                         goto out;
10621                 }
10622                 pt_flags |= MPTSAS_DATA_ALLOCATED;
10623                 if (direction == MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10624                         mutex_exit(&mpt->m_mutex);
10625                         for (i = 0; i < data_size; i++) {
10626                                 if (ddi_copyin(data + i, (uint8_t *)
10627                                     data_dma_state.memp + i, 1, mode)) {
10628                                         mutex_enter(&mpt->m_mutex);
10629                                         status = EFAULT;
10630                                         mptsas_log(mpt, CE_WARN, "failed to "
10631                                             "copy read data");
10632                                         goto out;
10633                                 }
10634                         }
10635                         mutex_enter(&mpt->m_mutex);
10636                 }
10637         }
10638         else
10639                 bzero(&data_dma_state, sizeof (data_dma_state));

10641         if (dataout_size != 0) {
10642                 dataout_dma_state.size = dataout_size;
10643                 if (mptsas_dma_alloc(mpt, &dataout_dma_state) != DDI_SUCCESS) {
10644                         status = ENOMEM;
10645                         mptsas_log(mpt, CE_WARN, "failed to alloc DMA "
10646                             "resource");
10647                         goto out;
10648                 }
10649                 pt_flags |= MPTSAS_DATAOUT_ALLOCATED;
10650                 mutex_exit(&mpt->m_mutex);
10651                 for (i = 0; i < dataout_size; i++) {
```

```
10652                         if (ddi_copyin(dataout + i, (uint8_t *)
10653                             dataout_dma_state.memp + i, 1, mode)) {
10654                                 mutex_enter(&mpt->m_mutex);
10655                                 mptsas_log(mpt, CE_WARN, "failed to copy out"
10656                                     " data");
10657                                 status = EFAULT;
10658                                 goto out;
10659                         }
10660                 }
10661                 mutex_enter(&mpt->m_mutex);
10662         }
10663         else
10664                 bzero(&dataout_dma_state, sizeof (dataout_dma_state));

10666         if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
10667                 status = EAGAIN;
10668                 mptsas_log(mpt, CE_NOTE, "event ack command pool is full");
10669                 goto out;
10670         }
10671         pt_flags |= MPTSAS_REQUEST_POOL_CMD;

10673         bzero((caddr_t)cmd, sizeof (*cmd));
10674         bzero((caddr_t)pkt, scsi_pkt_size());
10675         bzero((caddr_t)&pt, sizeof (pt));

10677         cmd->ioc_cmd_slot = (uint32_t)(rvalue);

10679         pt.request = (uint8_t *)request_msg;
10680         pt.direction = direction;
10681         pt.simple = 0;
10682         pt.request_size = request_size;
10683         pt.data_size = data_size;
10684         pt.dataout_size = dataout_size;
10685         pt.data_cookie = data_dma_state.cookie;
10686         pt.dataout_cookie = dataout_dma_state.cookie;
10687         mptsas_prep_sgl_offset(mpt, &pt);

10689         /*
10690          * Form a blank cmd/pkt to store the acknowledgement message
10691          */
10692         pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
10693         pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
10694         pkt->pkt_ha_private     = (opaque_t)&pt;
10695         pkt->pkt_flags          = FLAG_HEAD;
10696         pkt->pkt_time           = timeout;
10697         cmd->cmd_pkt            = pkt;
10698         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_PASSTHRU;

10700         /*
10701          * Save the command in a slot
10702          */
10703         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
10704                 /*
10705                  * Once passthru command get slot, set cmd_flags
10706                  * CFLAG_PREPARED.
10707                  */
10708                 cmd->cmd_flags |= CFLAG_PREPARED;
10709                 mptsas_start_passthru(mpt, cmd);
10710         } else {
10711                 mptsas_waitq_add(mpt, cmd);
10712         }

10714         while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
10715                 cv_wait(&mpt->m_passthru_cv, &mpt->m_mutex);
10716         }
```



```
10718         if (cmd->cmd_flags & CFLAG_PREPARED) {
10719                 memp = mpt->m_req_frame + (mpt->m_req_frame_size *
10720                     cmd->cmd_slot);
10721                 request_hdrp = (pMPI2RequestHeader_t)memp;
10722         }

10724         if (cmd->cmd_flags & CFLAG_TIMEOUT) {
10725                 status = ETIMEDOUT;
10726                 mptsas_log(mpt, CE_WARN, "passthrough command timeout");
10727                 pt_flags |= MPTSAS_CMD_TIMEOUT;
10728                 goto out;
10729         }

10731         if (cmd->cmd_rfm) {
10732                 /*
10733                  * cmd_rfm is zero means the command reply is a CONTEXT
10734                  * reply and no PCI Write to post the free reply SMFA
10735                  * because no reply message frame is used.
10736                  * cmd_rfm is non-zero means the reply is a ADDRESS
10737                  * reply and reply message frame is used.
10738                  */
10739                 pt_flags |= MPTSAS_ADDRESS_REPLY;
10740                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
10741                     DDI_DMA_SYNC_FORCPU);
10742                 reply_msg = (pMPI2DefaultReply_t)
10743                     (mpt->m_reply_frame + (cmd->cmd_rfm -
10744                     mpt->m_reply_frame_dma_addr));
10745         }

10747         mptsas_fma_check(mpt, cmd);
10748         if (pkt->pkt_reason == CMD_TRAN_ERR) {
10749                 status = EAGAIN;
10750                 mptsas_log(mpt, CE_WARN, "passthru fma error");
10751                 goto out;
10752         }
10753         if (pkt->pkt_reason == CMD_RESET) {
10754                 status = EAGAIN;
10755                 mptsas_log(mpt, CE_WARN, "ioc reset abort passthru");
10756                 goto out;
10757         }

10759         if (pkt->pkt_reason == CMD_INCOMPLETE) {
10760                 status = EIO;
10761                 mptsas_log(mpt, CE_WARN, "passthrough command incomplete");
10762                 goto out;
10763         }

10765         mutex_exit(&mpt->m_mutex);
10766         if (cmd->cmd_flags & CFLAG_PREPARED) {
10767                 function = request_hdrp->Function;
10768                 if ((function == MPI2_FUNCTION_SCSI_IO_REQUEST) ||
10769                     (function == MPI2_FUNCTION_RAID_SCSI_IO_PASSTHROUGH)) {
10770                         reply_len = sizeof (MPI2_SCSI_IO_REPLY);
10771                         sense_len = reply_size - reply_len;
10772                 } else {
10773                         reply_len = reply_size;
10774                         sense_len = 0;
10775                 }

10777                 for (i = 0; i < reply_len; i++) {
10778                         if (ddi_copyout((uint8_t *)reply_msg + i, reply + i, 1,
10779                             mode)) {
10780                                 mutex_enter(&mpt->m_mutex);
10781                                 status = EFAULT;
10782                                 mptsas_log(mpt, CE_WARN, "failed to copy out "
10783                                     "reply data");
```

```
10784                                   goto out;
10785                           }
10786                   }
10787                   for (i = 0; i < sense_len; i++) {
10788                           if (ddi_copyout((uint8_t *)request_hdrp + 64 + i,
10789                               reply + reply_len + i, 1, mode)) {
10790                                   mutex_enter(&mpt->m_mutex);
10791                                   status = EFAULT;
10792                                   mptsas_log(mpt, CE_WARN, "failed to copy out "
10793                                       "sense data");
10794                                   goto out;
10795                           }
10796                   }
10797           }

10799           if (data_size) {
10800                   if (direction != MPTSAS_PASS_THRU_DIRECTION_WRITE) {
10801                           (void) ddi_dma_sync(data_dma_state.handle, 0, 0,
10802                               DDI_DMA_SYNC_FORCPU);
10803                           for (i = 0; i < data_size; i++) {
10804                                   if (ddi_copyout((uint8_t *)(
10805                                       data_dma_state.memp + i), data + i,  1,
10806                                       mode)) {
10807                                           mutex_enter(&mpt->m_mutex);
10808                                           status = EFAULT;
10809                                           mptsas_log(mpt, CE_WARN, "failed to "
10810                                               "copy out the reply data");
10811                                           goto out;
10812                                   }
10813                           }
10814                   }
10815           }
10816           mutex_enter(&mpt->m_mutex);
10817   out:
10818           /*
10819            * Put the reply frame back on the free queue, increment the free
10820            * index, and write the new index to the free index register.  But only
10821            * if this reply is an ADDRESS reply.
10822            */
10823           if (pt_flags & MPTSAS_ADDRESS_REPLY) {
10824                   ddi_put32(mpt->m_acc_free_queue_hdl,
10825                       &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
10826                       cmd->cmd_rfm);
10827                   (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
10828                       DDI_DMA_SYNC_FORDEV);
10829                   if (++mpt->m_free_index == mpt->m_free_queue_depth) {
10830                           mpt->m_free_index = 0;
10831                   }
10832                   ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
10833                       mpt->m_free_index);
10834           }
10835           if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
10836                   mptsas_remove_cmd(mpt, cmd);
10837                   pt_flags &= (~MPTSAS_REQUEST_POOL_CMD);
10838           }
10839           if (pt_flags & MPTSAS_REQUEST_POOL_CMD)
10840                   mptsas_return_to_pool(mpt, cmd);
10841           if (pt_flags & MPTSAS_DATA_ALLOCATED) {
10842                   if (mptsas_check_dma_handle(data_dma_state.handle) !=
10843                       DDI_SUCCESS) {
10844                           ddi_fm_service_impact(mpt->m_dip,
10845                               DDI_SERVICE_UNAFFECTED);
10846                           status = EFAULT;
10847                   }
10848                   mptsas_dma_free(&data_dma_state);
10849           }
```



```
10850           if (pt_flags & MPTSAS_DATAOUT_ALLOCATED) {
10851                   if (mptsas_check_dma_handle(dataout_dma_state.handle) !=
10852                       DDI_SUCCESS) {
10853                           ddi_fm_service_impact(mpt->m_dip,
10854                               DDI_SERVICE_UNAFFECTED);
10855                           status = EFAULT;
10856                   }
10857                   mptsas_dma_free(&dataout_dma_state);
10858           }
10859           if (pt_flags & MPTSAS_CMD_TIMEOUT) {
10860                   if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
10861                           mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
10862                   }
10863           }
10864           if (request_msg)
10865                   kmem_free(request_msg, request_size);

10867           return (status);
10868   }

10870   static int
10871   mptsas_pass_thru(mptsas_t *mpt, mptsas_pass_thru_t *data, int mode)
10872   {
10873           /*
10874            * If timeout is 0, set timeout to default of 60 seconds.
10875            */
10876           if (data->Timeout == 0) {
10877                   data->Timeout = MPTSAS_PASS_THRU_TIME_DEFAULT;
10878           }

10880           if (((data->DataSize == 0) &&
10881               (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_NONE)) ||
10882               ((data->DataSize != 0) &&
10883               ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_READ) ||
10884               (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_WRITE) ||
10885               ((data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) &&
10886               (data->DataOutSize != 0))))) {
10887                   if (data->DataDirection == MPTSAS_PASS_THRU_DIRECTION_BOTH) {
10888                           data->DataDirection = MPTSAS_PASS_THRU_DIRECTION_READ;
10889                   } else {
10890                           data->DataOutSize = 0;
10891                   }
10892                   /*
10893                    * Send passthru request messages
10894                    */
10895                   return (mptsas_do_passthru(mpt,
10896                       (uint8_t *)((uintptr_t)data->PtrRequest),
10897                       (uint8_t *)((uintptr_t)data->PtrReply),
10898                       (uint8_t *)((uintptr_t)data->PtrData),
10899                       data->RequestSize, data->ReplySize,
10900                       data->DataSize, (uint8_t)data->DataDirection,
10148                      data->DataSize, data->DataDirection,
10901                       (uint8_t *)((uintptr_t)data->PtrDataOut),
10902                       data->DataOutSize, data->Timeout, mode));
10903           } else {
10904                   return (EINVAL);
10905           }
10906   }
_____unchanged_portion_omitted_

11979   static void
11980   mptsas_read_adapter_data(mptsas_t *mpt, mptsas_adapter_data_t *adapter_data)
11981   {
11982           char    *driver_verstr = MPTSAS_MOD_STRING;

11984           mptsas_lookup_pci_data(mpt, adapter_data);
```

```
11985           adapter_data->AdapterType = mpt->m_MPI25 ?
11986               MPTIOCTL_ADAPTER_TYPE_SAS3 :
11987               MPTIOCTL_ADAPTER_TYPE_SAS2;
11233           adapter_data->AdapterType = MPTIOCTL_ADAPTER_TYPE_SAS2;
11988           adapter_data->PCIDeviceHwId = (uint32_t)mpt->m_devid;
11989           adapter_data->PCIDeviceHwRev = (uint32_t)mpt->m_revid;
11990           adapter_data->SubSystemId = (uint32_t)mpt->m_ssid;
11991           adapter_data->SubsystemVendorId = (uint32_t)mpt->m_svid;
11992           (void) strcpy((char *)&adapter_data->DriverVersion[0], driver_verstr);
11993           adapter_data->BiosVersion = 0;
11994           (void) mptsas_get_bios_page3(mpt, &adapter_data->BiosVersion);
11995   }
______unchanged_portion_omitted_

15733   /* smp transport routine */
15734   static int mptsas_smp_start(struct smp_pkt *smp_pkt)
15735   {
15736           uint64_t                        wwn;
15737           Mpi2SmpPassthroughRequest_t     req;
15738           Mpi2SmpPassthroughReply_t       rep;
15739           uint8_t                         direction = 0;
14985           uint32_t                        direction = 0;
15740           mptsas_t                        *mpt;
15741           int                             ret;
15742           uint64_t                        tmp64;

15744           mpt = (mptsas_t *)smp_pkt->smp_pkt_address->
15745               smp_a_hba_tran->smp_tran_hba_private;

15747           bcopy(smp_pkt->smp_pkt_address->smp_a_wwn, &wwn, SAS_WWN_BYTE_SIZE);
15748           /*
15749            * Need to compose a SMP request message
15750            * and call mptsas_do_passthru() function
15751            */
15752           bzero(&req, sizeof (req));
15753           bzero(&rep, sizeof (rep));
15754           req.PassthroughFlags = 0;
15755           req.PhysicalPort = 0xff;
15756           req.ChainOffset = 0;
15757           req.Function = MPI2_FUNCTION_SMP_PASSTHROUGH;

15759           if ((smp_pkt->smp_pkt_reqsize & 0xffff0000ul) != 0) {
15760                   smp_pkt->smp_pkt_reason = ERANGE;
15761                   return (DDI_FAILURE);
15762           }
15763           req.RequestDataLength = LE_16((uint16_t)(smp_pkt->smp_pkt_reqsize - 4));

15765           req.MsgFlags = 0;
15766           tmp64 = LE_64(wwn);
15767           bcopy(&tmp64, &req.SASAddress, SAS_WWN_BYTE_SIZE);
15768           if (smp_pkt->smp_pkt_rspsize > 0) {
15769                   direction |= MPTSAS_PASS_THRU_DIRECTION_READ;
15770           }
15771           if (smp_pkt->smp_pkt_reqsize > 0) {
15772                   direction |= MPTSAS_PASS_THRU_DIRECTION_WRITE;
15773           }

15775           mutex_enter(&mpt->m_mutex);
15776           ret = mptsas_do_passthru(mpt, (uint8_t *)&req, (uint8_t *)&rep,
15777               (uint8_t *)smp_pkt->smp_pkt_rsp,
15778               offsetof(Mpi2SmpPassthroughRequest_t, SGL), sizeof (rep),
15779               smp_pkt->smp_pkt_rspsize - 4, direction,
15780               (uint8_t *)smp_pkt->smp_pkt_req, smp_pkt->smp_pkt_reqsize - 4,
15781               smp_pkt->smp_pkt_timeout, FKIOCTL);
15782           mutex_exit(&mpt->m_mutex);
15783           if (ret != 0) {
```



```
15784                   cmn_err(CE_WARN, "smp_start do passthru error %d", ret);
15785                   smp_pkt->smp_pkt_reason = (uchar_t)(ret);
15786                   return (DDI_FAILURE);
15787           }
15788           /* do passthrough success, check the smp status */
15789           if (LE_16(rep.IOCStatus) != MPI2_IOCSTATUS_SUCCESS) {
15790                   switch (LE_16(rep.IOCStatus)) {
15791                   case MPI2_IOCSTATUS_SCSI_DEVICE_NOT_THERE:
15792                           smp_pkt->smp_pkt_reason = ENODEV;
15793                           break;
15794                   case MPI2_IOCSTATUS_SAS_SMP_DATA_OVERRUN:
15795                           smp_pkt->smp_pkt_reason = EOVERFLOW;
15796                           break;
15797                   case MPI2_IOCSTATUS_SAS_SMP_REQUEST_FAILED:
15798                           smp_pkt->smp_pkt_reason = EIO;
15799                           break;
15800                   default:
15801                           mptsas_log(mpt, CE_NOTE, "smp_start: get unknown ioc"
15802                               "status:%x", LE_16(rep.IOCStatus));
15803                           smp_pkt->smp_pkt_reason = EIO;
15804                           break;
15805                   }
15806                   return (DDI_FAILURE);
15807           }
15808           if (rep.SASStatus != MPI2_SASSTATUS_SUCCESS) {
15809                   mptsas_log(mpt, CE_NOTE, "smp_start: get error SAS status:%x",
15810                       rep.SASStatus);
15811                   smp_pkt->smp_pkt_reason = EIO;
15812                   return (DDI_FAILURE);
15813           }

15815           return (DDI_SUCCESS);
15816   }
______unchanged_portion_omitted_
```

```
**********************************************************
   83458 Tue Jun 17 10:45:14 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_impl.c
NEX-1889 upstream
**********************************************************
```

```
   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  * Copyright 2014 OmniTI Computer Consulting, Inc. All rights reserved.
  26  * Copyright (c) 2014, Tegile Systems Inc. All rights reserved.
  27  */

  29 /*
  30  * Copyright (c) 2000 to 2010, LSI Corporation.
  31  * All rights reserved.
  32  *
  33  * Redistribution and use in source and binary forms of all code within
  34  * this file that is exclusively owned by LSI, with or without
  35  * modification, is permitted provided that, in addition to the CDDL 1.0
  36  * License requirements, the following conditions are met:
  37  *
  38  *    Neither the name of the author nor the names of its contributors may be
  39  *    used to endorse or promote products derived from this software without
  40  *    specific prior written permission.
  41  *
  42  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  43  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  44  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  45  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
  46  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
  47  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
  48  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
  49  * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
  50  * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  51  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
  52  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
  53  * DAMAGE.
  54  */

  56 /*
  57  * mptsas_impl - This file contains all the basic functions for communicating
  58  * to MPT based hardware.
  59  */

  61 #if defined(lint) || defined(DEBUG)
```



```
  62 #define MPTSAS_DEBUG
  63 #endif

  65 /*
  66  * standard header files
  67  */
  68 #include <sys/note.h>
  69 #include <sys/scsi/scsi.h>
  70 #include <sys/pci.h>

  72 #pragma pack(1)
  73 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_type.h>
  74 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2.h>
  75 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>
  76 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_init.h>
  77 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_ioc.h>
  78 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_sas.h>
  79 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>
  80 #pragma pack()

  82 /*
  83  * private header files.
  84  */
  85 #include <sys/scsi/adapters/mpt_sas/mptsas_var.h>
  86 #include <sys/scsi/adapters/mpt_sas/mptsas_smhba.h>

  88 /*
  89  * FMA header files.
  90  */
  91 #include <sys/fm/io/ddi.h>

  93 #if defined(MPTSAS_DEBUG)
  94 extern uint32_t mptsas_debug_flags;
  95 #endif

  97 /*
  98  *  prototypes
  99  */
 100 static void mptsas_ioc_event_cmdq_add(mptsas_t *mpt, m_event_struct_t *cmd);
 101 static void mptsas_ioc_event_cmdq_delete(mptsas_t *mpt, m_event_struct_t *cmd);
 102 static m_event_struct_t *mptsas_ioc_event_find_by_cmd(mptsas_t *mpt,
 103     struct mptsas_cmd *cmd);

 105 /*
 106  * add ioc evnet cmd into the queue
 107  */
 108 static void
 109 mptsas_ioc_event_cmdq_add(mptsas_t *mpt, m_event_struct_t *cmd)
 110 {
 111         if ((cmd->m_event_linkp = mpt->m_ioc_event_cmdq) == NULL) {
 112                 mpt->m_ioc_event_cmdtail = &cmd->m_event_linkp;
 113                 mpt->m_ioc_event_cmdq = cmd;
 114         } else {
 115                 cmd->m_event_linkp = NULL;
 116                 *(mpt->m_ioc_event_cmdtail) = cmd;
 117                 mpt->m_ioc_event_cmdtail = &cmd->m_event_linkp;
 118         }
 119 }
_____unchanged_portion_omitted_

 296 int
 297 mptsas_access_config_page(mptsas_t *mpt, uint8_t action, uint8_t page_type,
 298     uint8_t page_number, uint32_t page_address, int (*callback) (mptsas_t *,
 299     caddr_t, ddi_acc_handle_t, uint16_t, uint32_t, va_list), ...)
 300 {
 301         va_list                 ap;
```

```
302         ddi_dma_attr_t          attrs;
303         ddi_dma_cookie_t        cookie;
304         ddi_acc_handle_t        accessp;
305         size_t                  len = 0;
306         mptsas_config_request_t config;
307         int                     rval = DDI_SUCCESS, config_flags = 0;
308         mptsas_cmd_t            *cmd;
309         struct scsi_pkt         *pkt;
310         pMpi2ConfigReply_t      reply;
311         uint16_t                iocstatus = 0;
312         uint32_t                iocloginfo;
313         caddr_t                 page_memp;
314         boolean_t               free_dma = B_FALSE;

316         va_start(ap, callback);
317         ASSERT(mutex_owned(&mpt->m_mutex));

319         /*
320          * Get a command from the pool.
321          */
322         if ((rval = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
323                 mptsas_log(mpt, CE_NOTE, "command pool is full for config "
324                     "page request");
325                 rval = DDI_FAILURE;
326                 goto page_done;
327         }
328         config_flags |= MPTSAS_REQUEST_POOL_CMD;

330         bzero((caddr_t)cmd, sizeof (*cmd));
331         bzero((caddr_t)pkt, scsi_pkt_size());
332         bzero((caddr_t)&config, sizeof (config));

334         /*
335          * Save the data for this request to be used in the call to start the
336          * config header request.
337          */
338         config.action = MPI2_CONFIG_ACTION_PAGE_HEADER;
339         config.page_type = page_type;
340         config.page_number = page_number;
341         config.page_address = page_address;

343         /*
344          * Form a blank cmd/pkt to store the acknowledgement message
345          */
346         pkt->pkt_ha_private     = (opaque_t)&config;
347         pkt->pkt_flags          = FLAG_HEAD;
348         pkt->pkt_time           = 60;
349         cmd->cmd_pkt            = pkt;
350         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_CONFIG;

352         /*
353          * Save the config header request message in a slot.
354          */
355         if (mptsas_save_cmd(mpt, cmd) == TRUE) {
356                 cmd->cmd_flags |= CFLAG_PREPARED;
357                 mptsas_start_config_page_access(mpt, cmd);
358         } else {
359                 mptsas_waitq_add(mpt, cmd);
360         }

362         /*
363          * If this is a request for a RAID info page, or any page called during
364          * the RAID info page request, poll because these config page requests
365          * are nested.  Poll to avoid data corruption due to one page's data
366          * overwriting the outer page request's data.  This can happen when
367          * the mutex is released in cv_wait.
```



```
368          */
369         if ((page_type == MPI2_CONFIG_EXTPAGETYPE_RAID_CONFIG) ||
370             (page_type == MPI2_CONFIG_PAGETYPE_RAID_VOLUME) ||
371             (page_type == MPI2_CONFIG_PAGETYPE_RAID_PHYSDISK)) {
372                 (void) mptsas_poll(mpt, cmd, pkt->pkt_time * 1000);
373         } else {
374                 while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
375                         cv_wait(&mpt->m_config_cv, &mpt->m_mutex);
376                 }
377         }

379         /*
380          * Check if the header request completed without timing out
381          */
382         if (cmd->cmd_flags & CFLAG_TIMEOUT) {
383                 mptsas_log(mpt, CE_WARN, "config header request timeout");
384                 rval = DDI_FAILURE;
385                 goto page_done;
386         }

388         /*
389          * cmd_rfm points to the reply message if a reply was given.  Check the
390          * IOCStatus to make sure everything went OK with the header request.
391          */
392         if (cmd->cmd_rfm) {
393                 config_flags |= MPTSAS_ADDRESS_REPLY;
394                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
395                     DDI_DMA_SYNC_FORCPU);
396                 reply = (pMpi2ConfigReply_t)(mpt->m_reply_frame + (cmd->cmd_rfm
397                     - mpt->m_reply_frame_dma_addr));
398                 config.page_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
399                     &reply->Header.PageType);
400                 config.page_number = ddi_get8(mpt->m_acc_reply_frame_hdl,
401                     &reply->Header.PageNumber);
402                 config.page_length = ddi_get8(mpt->m_acc_reply_frame_hdl,
403                     &reply->Header.PageLength);
404                 config.page_version = ddi_get8(mpt->m_acc_reply_frame_hdl,
405                     &reply->Header.PageVersion);
406                 config.ext_page_type = ddi_get8(mpt->m_acc_reply_frame_hdl,
407                     &reply->ExtPageType);
408                 config.ext_page_length = ddi_get16(mpt->m_acc_reply_frame_hdl,
409                     &reply->ExtPageLength);

411                 iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
412                     &reply->IOCStatus);
413                 iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
414                     &reply->IOCLogInfo);

416                 if (iocstatus) {
417                         NDBG13(("mptsas_access_config_page header: "
418                             "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
419                             iocloginfo));
420                         rval = DDI_FAILURE;
421                         goto page_done;
422                 }

424                 if ((config.page_type & MPI2_CONFIG_PAGETYPE_MASK) ==
425                     MPI2_CONFIG_PAGETYPE_EXTENDED)
426                         len = (config.ext_page_length * 4);
427                 else
428                         len = (config.page_length * 4);

430         }

432         if (pkt->pkt_reason == CMD_RESET) {
433                 mptsas_log(mpt, CE_WARN, "ioc reset abort config header "
```

```
434                     "request");
435             rval = DDI_FAILURE;
436             goto page_done;
437         }
438
439         /*
440          * Put the reply frame back on the free queue, increment the free
441          * index, and write the new index to the free index register.  But only
442          * if this reply is an ADDRESS reply.
443          */
444         if (config_flags & MPTSAS_ADDRESS_REPLY) {
445                 ddi_put32(mpt->m_acc_free_queue_hdl,
446                     &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
447                     cmd->cmd_rfm);
448                 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
449                     DDI_DMA_SYNC_FORDEV);
450                 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
451                         mpt->m_free_index = 0;
452                 }
453                 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
454                     mpt->m_free_index);
455                 config_flags &= (~MPTSAS_ADDRESS_REPLY);
456         }
457
458         /*
459          * Allocate DMA buffer here.  Store the info regarding this buffer in
460          * the cmd struct so that it can be used for this specific command and
461          * de-allocated after the command completes.  The size of the reply
462          * will not be larger than the reply frame size.
463          */
464         attrs = mpt->m_msg_dma_attr;
465         attrs.dma_attr_sgllen = 1;
466         attrs.dma_attr_granular = (uint32_t)len;
467
468         if (mptsas_dma_addr_create(mpt, attrs,
469             &cmd->cmd_dmahandle, &accessp, &page_memp,
470             len, &cookie) == FALSE) {
471                 rval = DDI_FAILURE;
472                 mptsas_log(mpt, CE_WARN,
473                     "mptsas_dma_addr_create(len=0x%x) failed", (int)len);
474                 goto page_done;
475         }
476         /* NOW we can safely call mptsas_dma_addr_destroy(). */
477         free_dma = B_TRUE;
478
479         cmd->cmd_dma_addr = cookie.dmac_laddress;
480         bzero(page_memp, len);
481
482         /*
483          * Save the data for this request to be used in the call to start the
484          * config page read
485          */
486         config.action = action;
487         config.page_address = page_address;
488
489         /*
490          * Re-use the cmd that was used to get the header.  Reset some of the
491          * values.
492          */
493         bzero((caddr_t)pkt, scsi_pkt_size());
494         pkt->pkt_ha_private     = (opaque_t)&config;
495         pkt->pkt_flags          = FLAG_HEAD;
496         pkt->pkt_time           = 60;
497         cmd->cmd_flags          = CFLAG_PREPARED | CFLAG_CMDIOC | CFLAG_CONFIG;
498
499         /*
```



```
500          * Send the config page request.  cmd is re-used from header request.
501          */
502         mptsas_start_config_page_access(mpt, cmd);
503
504         /*
505          * If this is a request for a RAID info page, or any page called during
506          * the RAID info page request, poll because these config page requests
507          * are nested.  Poll to avoid data corruption due to one page's data
508          * overwriting the outer page request's data.  This can happen when
509          * the mutex is released in cv_wait.
510          */
511         if ((page_type == MPI2_CONFIG_EXTPAGETYPE_RAID_CONFIG) ||
512             (page_type == MPI2_CONFIG_PAGETYPE_RAID_VOLUME) ||
513             (page_type == MPI2_CONFIG_PAGETYPE_RAID_PHYSDISK)) {
514                 (void) mptsas_poll(mpt, cmd, pkt->pkt_time * 1000);
515         } else {
516                 while ((cmd->cmd_flags & CFLAG_FINISHED) == 0) {
517                         cv_wait(&mpt->m_config_cv, &mpt->m_mutex);
518                 }
519         }
520
521         /*
522          * Check if the request completed without timing out
523          */
524         if (cmd->cmd_flags & CFLAG_TIMEOUT) {
525                 mptsas_log(mpt, CE_WARN, "config page request timeout");
526                 rval = DDI_FAILURE;
527                 goto page_done;
528         }
529
530         /*
531          * cmd_rfm points to the reply message if a reply was given.  The reply
532          * frame and the config page are returned from this function in the
533          * param list.
534          */
535         if (cmd->cmd_rfm) {
536                 config_flags |= MPTSAS_ADDRESS_REPLY;
537                 (void) ddi_dma_sync(mpt->m_dma_reply_frame_hdl, 0, 0,
538                     DDI_DMA_SYNC_FORCPU);
539                 (void) ddi_dma_sync(cmd->cmd_dmahandle, 0, 0,
540                     DDI_DMA_SYNC_FORCPU);
541                 reply = (pMpi2ConfigReply_t)(mpt->m_reply_frame + (cmd->cmd_rfm
542                     - mpt->m_reply_frame_dma_addr));
543                 iocstatus = ddi_get16(mpt->m_acc_reply_frame_hdl,
544                     &reply->IOCStatus);
545                 iocstatus = MPTSAS_IOCSTATUS(iocstatus);
546                 iocloginfo = ddi_get32(mpt->m_acc_reply_frame_hdl,
547                     &reply->IOCLogInfo);
548         }
549
550         if (callback(mpt, page_memp, accessp, iocstatus, iocloginfo, ap)) {
551                 rval = DDI_FAILURE;
552                 goto page_done;
553         }
554
555         mptsas_fma_check(mpt, cmd);
556         /*
557          * Check the DMA/ACC handles and then free the DMA buffer.
558          */
559         if ((mptsas_check_dma_handle(cmd->cmd_dmahandle) != DDI_SUCCESS) ||
560             (mptsas_check_acc_handle(accessp) != DDI_SUCCESS)) {
561                 ddi_fm_service_impact(mpt->m_dip, DDI_SERVICE_UNAFFECTED);
562                 rval = DDI_FAILURE;
563         }
564
565         if (pkt->pkt_reason == CMD_TRAN_ERR) {
```

```
 566                 mptsas_log(mpt, CE_WARN, "config fma error");
 567                 rval = DDI_FAILURE;
 568                 goto page_done;
 569         }
 570         if (pkt->pkt_reason == CMD_RESET) {
 571                 mptsas_log(mpt, CE_WARN, "ioc reset abort config request");
 572                 rval = DDI_FAILURE;
 573                 goto page_done;
 574         }

 576 page_done:
 577         va_end(ap);
 578         /*
 579          * Put the reply frame back on the free queue, increment the free
 580          * index, and write the new index to the free index register.  But only
 581          * if this reply is an ADDRESS reply.
 582          */
 583         if (config_flags & MPTSAS_ADDRESS_REPLY) {
 584                 ddi_put32(mpt->m_acc_free_queue_hdl,
 585                     &((uint32_t *)(void *)mpt->m_free_queue)[mpt->m_free_index],
 586                     cmd->cmd_rfm);
 587                 (void) ddi_dma_sync(mpt->m_dma_free_queue_hdl, 0, 0,
 588                     DDI_DMA_SYNC_FORDEV);
 589                 if (++mpt->m_free_index == mpt->m_free_queue_depth) {
 590                         mpt->m_free_index = 0;
 591                 }
 592                 ddi_put32(mpt->m_datap, &mpt->m_reg->ReplyFreeHostIndex,
 593                     mpt->m_free_index);
 594         }

 596         if (free_dma)
 597                 mptsas_dma_addr_destroy(&cmd->cmd_dmahandle, &accessp);

 599         if (cmd && (cmd->cmd_flags & CFLAG_PREPARED)) {
 600                 mptsas_remove_cmd(mpt, cmd);
 601                 config_flags &= (~MPTSAS_REQUEST_POOL_CMD);
 602         }
 603         if (config_flags & MPTSAS_REQUEST_POOL_CMD)
 604                 mptsas_return_to_pool(mpt, cmd);

 606         if (config_flags & MPTSAS_CMD_TIMEOUT) {
 607                 mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
 608                 if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
 609                         mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
 610                 }
 611         }

 613         return (rval);
 614 }
_____unchanged_portion_omitted_

1212 /*
1213  * Complete firmware download frame for v2.0 cards.
1214  */
1215 static void
1216 mptsas_uflash2(pMpi2FWDownloadRequest fwdownload,
1217     ddi_acc_handle_t acc_hdl, uint32_t size, uint8_t type,
1218     ddi_dma_cookie_t flsh_cookie)
1219 {
1220         pMpi2FWDownloadTCSGE_t  tcsge;
1221         pMpi2SGESimple64_t      sge;
1222         uint32_t                flagslength;

1224         ddi_put8(acc_hdl, &fwdownload->Function,
1225             MPI2_FUNCTION_FW_DOWNLOAD);
1226         ddi_put8(acc_hdl, &fwdownload->ImageType, type);
```



```
1227         ddi_put8(acc_hdl, &fwdownload->MsgFlags,
1228             MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1229         ddi_put32(acc_hdl, &fwdownload->TotalImageSize, size);

1231         tcsge = (pMpi2FWDownloadTCSGE_t)&fwdownload->SGL;
1232         ddi_put8(acc_hdl, &tcsge->ContextSize, 0);
1233         ddi_put8(acc_hdl, &tcsge->DetailsLength, 12);
1234         ddi_put8(acc_hdl, &tcsge->Flags, 0);
1235         ddi_put32(acc_hdl, &tcsge->ImageOffset, 0);
1236         ddi_put32(acc_hdl, &tcsge->ImageSize, size);

1238         sge = (pMpi2SGESimple64_t)(tcsge + 1);
1239         flagslength = size;
1240         flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
1241             MPI2_SGE_FLAGS_END_OF_BUFFER |
1242             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
1243             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
1244             MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
1245             MPI2_SGE_FLAGS_HOST_TO_IOC |
1246             MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
1247         ddi_put32(acc_hdl, &sge->FlagsLength, flagslength);
1248         ddi_put32(acc_hdl, &sge->Address.Low,
1249             flsh_cookie.dmac_address);
1250         ddi_put32(acc_hdl, &sge->Address.High,
1251             (uint32_t)(flsh_cookie.dmac_laddress >> 32));
1252 }

1254 /*
1255  * Complete firmware download frame for v2.0 cards.
1256  */
1257 static void
1258 mptsas_uflash25(pMpi25FWDownloadRequest fwdownload,
1259     ddi_acc_handle_t acc_hdl, uint32_t size, uint8_t type,
1260     ddi_dma_cookie_t flsh_cookie)
1261 {
1262         pMpi2IeeeSgeSimple64_t  sge;
1263         uint8_t                 flags;

1265         ddi_put8(acc_hdl, &fwdownload->Function,
1266             MPI2_FUNCTION_FW_DOWNLOAD);
1267         ddi_put8(acc_hdl, &fwdownload->ImageType, type);
1268         ddi_put8(acc_hdl, &fwdownload->MsgFlags,
1269             MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1270         ddi_put32(acc_hdl, &fwdownload->TotalImageSize, size);

1272         ddi_put32(acc_hdl, &fwdownload->ImageOffset, 0);
1273         ddi_put32(acc_hdl, &fwdownload->ImageSize, size);

1275         sge = (pMpi2IeeeSgeSimple64_t)&fwdownload->SGL;
1276         flags = MPI2_IEEE_SGE_FLAGS_SIMPLE_ELEMENT |
1277             MPI2_IEEE_SGE_FLAGS_SYSTEM_ADDR |
1278             MPI25_IEEE_SGE_FLAGS_END_OF_LIST;
1279         ddi_put8(acc_hdl, &sge->Flags, flags);
1280         ddi_put32(acc_hdl, &sge->Length, size);
1281         ddi_put32(acc_hdl, &sge->Address.Low,
1282             flsh_cookie.dmac_address);
1283         ddi_put32(acc_hdl, &sge->Address.High,
1284             (uint32_t)(flsh_cookie.dmac_laddress >> 32));
1285 }

1287 static int mptsas_enable_mpi25_flashupdate = 0;

1289 int
1290 mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
1291     uint8_t type, int mode)
1292 {
```

```
1294         /*
1295          * In order to avoid allocating variables on the stack,
1296          * we make use of the pre-existing mptsas_cmd_t and
1297          * scsi_pkt which are included in the mptsas_t which
1298          * is passed to this routine.
1299          */

1301         ddi_dma_attr_t          flsh_dma_attrs;
1302         ddi_dma_cookie_t        flsh_cookie;
1303         ddi_dma_handle_t        flsh_dma_handle;
1304         ddi_acc_handle_t        flsh_accessp;
1305         caddr_t                 memp, flsh_memp;
1226         uint32_t                flagslength;
1227         pMpi2FWDownloadRequest  fwdownload;
1228         pMpi2FWDownloadTCSGE_t  tcsge;
1229         pMpi2SGESimple64_t      sge;
1306         mptsas_cmd_t            *cmd;
1307         struct scsi_pkt         *pkt;
1308         int                     i;
1309         int                     rvalue = 0;
1310         uint32_t                request_desc_low;

1312         if (mpt->m_MPI25) {
1313                 /*
1314                  * The code is there but not tested yet.
1315                  * User has to know there are risks here.
1316                  */
1317                 mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): "
1318                     "Updating firmware through MPI 2.5 has not been "
1319                     "tested yet!\n"
1320                     "To enable set mptsas_enable_mpi25_flashupdate to 1\n");
1321                 if (!mptsas_enable_mpi25_flashupdate)
1322                         return (-1);
1323         }

1325         if ((rvalue = (mptsas_request_from_pool(mpt, &cmd, &pkt))) == -1) {
1326                 mptsas_log(mpt, CE_WARN, "mptsas_update_flash(): allocation "
1327                     "failed. event ack command pool is full\n");
1328                 return (rvalue);
1329         }

1331         bzero((caddr_t)cmd, sizeof (*cmd));
1332         bzero((caddr_t)pkt, scsi_pkt_size());
1333         cmd->ioc_cmd_slot = (uint32_t)rvalue;

1335         /*
1336          * dynamically create a customized dma attribute structure
1337          * that describes the flash file.
1338          */
1339         flsh_dma_attrs = mpt->m_msg_dma_attr;
1340         flsh_dma_attrs.dma_attr_sgllen = 1;

1342         if (mptsas_dma_addr_create(mpt, flsh_dma_attrs, &flsh_dma_handle,
1343             &flsh_accessp, &flsh_memp, size, &flsh_cookie) == FALSE) {
1344                 mptsas_log(mpt, CE_WARN,
1345                     "(unable to allocate dma resource.");
1346                 mptsas_return_to_pool(mpt, cmd);
1347                 return (-1);
1348         }

1350         bzero(flsh_memp, size);

1352         for (i = 0; i < size; i++) {
1353                 (void) ddi_copyin(ptrbuffer + i, flsh_memp + i, 1, mode);
1354         }
```



```
1355         (void) ddi_dma_sync(flsh_dma_handle, 0, 0, DDI_DMA_SYNC_FORDEV);

1357         /*
1358          * form a cmd/pkt to store the fw download message
1359          */
1360         pkt->pkt_cdbp           = (opaque_t)&cmd->cmd_cdb[0];
1361         pkt->pkt_scbp           = (opaque_t)&cmd->cmd_scb;
1362         pkt->pkt_ha_private     = (opaque_t)cmd;
1363         pkt->pkt_flags          = FLAG_HEAD;
1364         pkt->pkt_time           = 60;
1365         cmd->cmd_pkt            = pkt;
1366         cmd->cmd_scblen         = 1;
1367         cmd->cmd_flags          = CFLAG_CMDIOC | CFLAG_FW_CMD;

1369         /*
1370          * Save the command in a slot
1371          */
1372         if (mptsas_save_cmd(mpt, cmd) == FALSE) {
1373                 mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);
1374                 mptsas_return_to_pool(mpt, cmd);
1375                 return (-1);
1376         }

1378         /*
1379          * Fill in fw download message
1380          */
1381         ASSERT(cmd->cmd_slot != 0);
1382         memp = mpt->m_req_frame + (mpt->m_req_frame_size * cmd->cmd_slot);
1383         bzero(memp, mpt->m_req_frame_size);
1295         fwdownload = (void *)memp;
1296         ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->Function,
1297             MPI2_FUNCTION_FW_DOWNLOAD);
1298         ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->ImageType, type);
1299         ddi_put8(mpt->m_acc_req_frame_hdl, &fwdownload->MsgFlags,
1300             MPI2_FW_DOWNLOAD_MSGFLGS_LAST_SEGMENT);
1301         ddi_put32(mpt->m_acc_req_frame_hdl, &fwdownload->TotalImageSize, size);

1385         if (mpt->m_MPI25)
1386                 mptsas_uflash2((pMpi2FWDownloadRequest)memp,
1387                     mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);
1388         else
1389                 mptsas_uflash25((pMpi25FWDownloadRequest)memp,
1390                     mpt->m_acc_req_frame_hdl, size, type, flsh_cookie);
1303         tcsge = (pMpi2FWDownloadTCSGE_t)&fwdownload->SGL;
1304         ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->ContextSize, 0);
1305         ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->DetailsLength, 12);
1306         ddi_put8(mpt->m_acc_req_frame_hdl, &tcsge->Flags, 0);
1307         ddi_put32(mpt->m_acc_req_frame_hdl, &tcsge->ImageOffset, 0);
1308         ddi_put32(mpt->m_acc_req_frame_hdl, &tcsge->ImageSize, size);

1310         sge = (pMpi2SGESimple64_t)(tcsge + 1);
1311         flagslength = size;
1312         flagslength |= ((uint32_t)(MPI2_SGE_FLAGS_LAST_ELEMENT |
1313             MPI2_SGE_FLAGS_END_OF_BUFFER |
1314             MPI2_SGE_FLAGS_SIMPLE_ELEMENT |
1315             MPI2_SGE_FLAGS_SYSTEM_ADDRESS |
1316             MPI2_SGE_FLAGS_64_BIT_ADDRESSING |
1317             MPI2_SGE_FLAGS_HOST_TO_IOC |
1318             MPI2_SGE_FLAGS_END_OF_LIST) << MPI2_SGE_FLAGS_SHIFT);
1319         ddi_put32(mpt->m_acc_req_frame_hdl, &sge->FlagsLength, flagslength);
1320         ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.Low,
1321             flsh_cookie.dmac_address);
1322         ddi_put32(mpt->m_acc_req_frame_hdl, &sge->Address.High,
1323             (uint32_t)(flsh_cookie.dmac_laddress >> 32));

1392         /*
```

```
1393          * Start command
1394          */
1395         (void) ddi_dma_sync(mpt->m_dma_req_frame_hdl, 0, 0,
1396             DDI_DMA_SYNC_FORDEV);
1397         request_desc_low = (cmd->cmd_slot << 16) +
1398             MPI2_REQ_DESCRIPT_FLAGS_DEFAULT_TYPE;
1399         cmd->cmd_rfm = NULL;
1400         MPTSAS_START_CMD(mpt, request_desc_low, 0);

1402         rvalue = 0;
1403         (void) cv_reltimedwait(&mpt->m_fw_cv, &mpt->m_mutex,
1404             drv_usectohz(60 * MICROSEC), TR_CLOCK_TICK);
1405         if (!(cmd->cmd_flags & CFLAG_FINISHED)) {
1406                 mpt->m_softstate &= ~MPTSAS_SS_MSG_UNIT_RESET;
1407                 if ((mptsas_restart_ioc(mpt)) == DDI_FAILURE) {
1408                         mptsas_log(mpt, CE_WARN, "mptsas_restart_ioc failed");
1409                 }
1410                 rvalue = -1;
1411         }
1412         mptsas_remove_cmd(mpt, cmd);
1413         mptsas_dma_addr_destroy(&flsh_dma_handle, &flsh_accessp);

1415         return (rvalue);
1416 }

1418 static int
1419 mptsas_sasdevpage_0_cb(mptsas_t *mpt, caddr_t page_memp,
1420     ddi_acc_handle_t accessp, uint16_t iocstatus, uint32_t iocloginfo,
1421     va_list ap)
1422 {
1423 #ifndef __lock_lint
1424         _NOTE(ARGUNUSED(ap))
1425 #endif
1426         pMpi2SasDevicePage0_t   sasdevpage;
1427         int                     rval = DDI_SUCCESS, i;
1428         uint8_t                 *sas_addr = NULL;
1429         uint8_t                 tmp_sas_wwn[SAS_WWN_BYTE_SIZE];
1430         uint16_t                *devhdl, *bay_num, *enclosure;
1431         uint64_t                *sas_wwn;
1432         uint32_t                *dev_info;
1433         uint8_t                 *physport, *phynum;
1434         uint16_t                *pdevhdl, io_flags;
1367         uint16_t                *pdevhdl;
1435         uint32_t                page_address;

1437         if ((iocstatus != MPI2_IOCSTATUS_SUCCESS) &&
1438             (iocstatus != MPI2_IOCSTATUS_CONFIG_INVALID_PAGE)) {
1439                 mptsas_log(mpt, CE_WARN, "mptsas_get_sas_device_page0 "
1440                     "header: IOCStatus=0x%x, IOCLogInfo=0x%x",
1441                     iocstatus, iocloginfo);
1442                 rval = DDI_FAILURE;
1443                 return (rval);
1444         }
1445         page_address = va_arg(ap, uint32_t);
1446         /*
1447          * The INVALID_PAGE status is normal if using GET_NEXT_HANDLE and there
1448          * are no more pages.  If everything is OK up to this point but the
1449          * status is INVALID_PAGE, change rval to FAILURE and quit.  Also,
1450          * signal that device traversal is complete.
1451          */
1452         if (iocstatus == MPI2_IOCSTATUS_CONFIG_INVALID_PAGE) {
1453                 if ((page_address & MPI2_SAS_DEVICE_PGAD_FORM_MASK) ==
1454                     MPI2_SAS_DEVICE_PGAD_FORM_GET_NEXT_HANDLE) {
1455                         mpt->m_done_traverse_dev = 1;
1456                 }
1457                 rval = DDI_FAILURE;
```



```
1458                 return (rval);
1459         }
1460         devhdl = va_arg(ap, uint16_t *);
1461         sas_wwn = va_arg(ap, uint64_t *);
1462         dev_info = va_arg(ap, uint32_t *);
1463         physport = va_arg(ap, uint8_t *);
1464         phynum = va_arg(ap, uint8_t *);
1465         pdevhdl = va_arg(ap, uint16_t *);
1466         bay_num = va_arg(ap, uint16_t *);
1467         enclosure = va_arg(ap, uint16_t *);


1470         sasdevpage = (pMpi2SasDevicePage0_t)page_memp;

1472         *dev_info = ddi_get32(accessp, &sasdevpage->DeviceInfo);
1473         *devhdl = ddi_get16(accessp, &sasdevpage->DevHandle);
1474         sas_addr = (uint8_t *)(&sasdevpage->SASAddress);
1475         for (i = 0; i < SAS_WWN_BYTE_SIZE; i++) {
1476                 tmp_sas_wwn[i] = ddi_get8(accessp, sas_addr + i);
1477         }
1478         bcopy(tmp_sas_wwn, sas_wwn, SAS_WWN_BYTE_SIZE);
1479         *sas_wwn = LE_64(*sas_wwn);
1480         *physport = ddi_get8(accessp, &sasdevpage->PhysicalPort);
1481         *phynum = ddi_get8(accessp, &sasdevpage->PhyNum);
1482         *pdevhdl = ddi_get16(accessp, &sasdevpage->ParentDevHandle);
1483         *bay_num = ddi_get16(accessp, &sasdevpage->Slot);
1484         *enclosure = ddi_get16(accessp, &sasdevpage->EnclosureHandle);

1486         /*
1487          * This is where we would check the flag
1488          * MPI25_SAS_DEVICE0_FLAGS_FAST_PATH_CAPABLE
1489          * and set something that will allow us to use fastpath during
1490          * target transfers.
1491          */
1492         io_flags = ddi_get16(accessp, &sasdevpage->Flags);
1493         if (io_flags & MPI25_SAS_DEVICE0_FLAGS_FAST_PATH_CAPABLE) {
1494                 /*
1495                  * uint8_t  *fast_path;
1496                  * fast_path = va_arg(ap, uint8_t *);
1497                  * *fast_path = MPI25_REQ_DESCRIPT_FLAGS_FAST_PATH_SCSI_IO;
1498                  *
1499                  * Need to change all calls to mptsas_get_sas_device_page0()
1500                  * to include another argument to collect this. And then what
1501                  * do we do with it?
1502                  * XXX For now print a message..
1503                  */
1504                 mptsas_log(mpt, CE_CONT,
1505                     "!Found MPI25_SAS_DEVICE0_FLAGS_FAST_"
1506                     "PATH_CAPABLE for %llx", (long long)*sas_wwn);
1507         }

1509         return (rval);
1510 }
______unchanged_portion_omitted_
```

```
**********************************************************
   19531 Tue Jun 17 10:45:14 2014
new/usr/src/uts/common/io/scsi/adapters/mpt_sas/mptsas_init.c
NEX-1889 upstream
**********************************************************
```

```
   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 2009 Sun Microsystems, Inc.  All rights reserved.
  24  * Use is subject to license terms.
  25  * Copyright (c) 2014, Tegile Systems Inc. All rights reserved.
  26  */

  28 /*
  29  * Copyright (c) 2000 to 2009, LSI Corporation.
  30  * All rights reserved.
  31  *
  32  * Redistribution and use in source and binary forms of all code within
  33  * this file that is exclusively owned by LSI, with or without
  34  * modification, is permitted provided that, in addition to the CDDL 1.0
  35  * License requirements, the following conditions are met:
  36  *
  37  *    Neither the name of the author nor the names of its contributors may be
  38  *    used to endorse or promote products derived from this software without
  39  *    specific prior written permission.
  40  *
  41  * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  42  * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  43  * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  44  * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
  45  * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
  46  * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
  47  * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
  48  * OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
  49  * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
  50  * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
  51  * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
  52  * DAMAGE.
  53  */

  55 /*
  56  * mptsas_init - This file contains all the functions used to initialize
  57  * MPT2.0 based hardware.
  58  */

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



```
  62 #endif

  64 /*
  65  * standard header files
  66  */
  67 #include <sys/note.h>
  68 #include <sys/scsi/scsi.h>

  70 #pragma pack(1)
  71 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_type.h>
  72 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2.h>
  73 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>
  74 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_init.h>
  75 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_ioc.h>
  76 #include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>
  77 #pragma pack()
  78 /*
  79  * private header files.
  80  */
  81 #include <sys/scsi/adapters/mpt_sas/mptsas_var.h>

  83 static int mptsas_ioc_do_get_facts(mptsas_t *mpt, caddr_t memp, int var,
  84         ddi_acc_handle_t accessp);
  85 static int mptsas_ioc_do_get_facts_reply(mptsas_t *mpt, caddr_t memp, int var,
  86         ddi_acc_handle_t accessp);
  87 static int mptsas_ioc_do_get_port_facts(mptsas_t *mpt, caddr_t memp, int var,
  88         ddi_acc_handle_t accessp);
  89 static int mptsas_ioc_do_get_port_facts_reply(mptsas_t *mpt, caddr_t memp,
  90     int var, ddi_acc_handle_t accessp);
  91 static int mptsas_ioc_do_enable_port(mptsas_t *mpt, caddr_t memp, int var,
  92         ddi_acc_handle_t accessp);
  93 static int mptsas_ioc_do_enable_port_reply(mptsas_t *mpt, caddr_t memp, int var,
  94         ddi_acc_handle_t accessp);
  95 static int mptsas_ioc_do_enable_event_notification(mptsas_t *mpt, caddr_t memp,
  96         int var, ddi_acc_handle_t accessp);
  97 static int mptsas_ioc_do_enable_event_notification_reply(mptsas_t *mpt,
  98     caddr_t memp, int var, ddi_acc_handle_t accessp);
  99 static int mptsas_do_ioc_init(mptsas_t *mpt, caddr_t memp, int var,
 100         ddi_acc_handle_t accessp);
 101 static int mptsas_do_ioc_init_reply(mptsas_t *mpt, caddr_t memp, int var,
 102         ddi_acc_handle_t accessp);

 104 static const char *
 105 mptsas_product_type_string(mptsas_t *mpt)
 106 {
 107         switch (mpt->m_productid & MPI2_FW_HEADER_PID_PROD_MASK) {

 109         case MPI2_FW_HEADER_PID_PROD_A:
 110                 return ("A");
 111         default:
 112                 return ("?");
 113         }
 114 }
_____unchanged_portion_omitted_

 163 static int
 164 mptsas_ioc_do_get_facts_reply(mptsas_t *mpt, caddr_t memp, int var,
 165                 ddi_acc_handle_t accessp)
 166 {
 167 #ifndef __lock_lint
 168         _NOTE(ARGUNUSED(var))
 169 #endif

 171         pMpi2IOCFactsReply_t    factsreply;
 172         int                     numbytes;
 173         uint_t                  iocstatus;
```

```
174         char                    buf[32];
175         uint16_t                numReplyFrames;
176         uint16_t                queueSize, queueDiff;
177         int                     simple_sge_main;
178         int                     simple_sge_next;
179         uint32_t                capabilities;
180         uint16_t                msgversion;

182         bzero(memp, sizeof (*factsreply));
183         factsreply = (void *)memp;
184         numbytes = sizeof (*factsreply);

186         /*
187          * get ioc facts reply message
188          */
189         if (mptsas_get_handshake_msg(mpt, memp, numbytes, accessp)) {
190                 return (DDI_FAILURE);
191         }

193         if (iocstatus = ddi_get16(accessp, &factsreply->IOCStatus)) {
194                 mptsas_log(mpt, CE_WARN, "mptsas_ioc_do_get_facts_reply: "
195                     "IOCStatus=0x%x, IOCLogInfo=0x%x", iocstatus,
196                     ddi_get32(accessp, &factsreply->IOCLogInfo));
197                 return (DDI_FAILURE);
198         }

200         /*
201          * store key values from reply to mpt structure
202          */
203         mpt->m_fwversion = ddi_get32(accessp, &factsreply->FWVersion.Word);
204         mpt->m_productid = ddi_get16(accessp, &factsreply->ProductID);


207         (void) sprintf(buf, "%u.%u.%u.%u",
208             ddi_get8(accessp, &factsreply->FWVersion.Struct.Major),
209             ddi_get8(accessp, &factsreply->FWVersion.Struct.Minor),
210             ddi_get8(accessp, &factsreply->FWVersion.Struct.Unit),
211             ddi_get8(accessp, &factsreply->FWVersion.Struct.Dev));
212         mptsas_log(mpt, CE_NOTE, "?mpt%d Firmware version v%s (%s)\n",
213             mpt->m_instance, buf, mptsas_product_type_string(mpt));
214         (void) ddi_prop_update_string(DDI_DEV_T_NONE, mpt->m_dip,
215             "firmware-version", buf);

217         /*
218          * Set up request info.
219          */
220         mpt->m_max_requests = ddi_get16(accessp,
221             &factsreply->RequestCredit) - 1;
222         mpt->m_req_frame_size = ddi_get16(accessp,
223             &factsreply->IOCRequestFrameSize) * 4;

225         /*
226          * Size of reply free queue should be the number of requests
227          * plus some additional for events (32).  Make sure number of
228          * reply frames is not a multiple of 16 so that the queue sizes
229          * are calculated correctly later to be a multiple of 16.
230          */
231         mpt->m_reply_frame_size = ddi_get8(accessp,
232             &factsreply->ReplyFrameSize) * 4;
233         numReplyFrames = mpt->m_max_requests + 32;
234         if (!(numReplyFrames % 16)) {
235                 numReplyFrames--;
236         }
237         mpt->m_max_replies = numReplyFrames;
238         queueSize = numReplyFrames;
239         queueSize += 16 - (queueSize % 16);
```



```
240         mpt->m_free_queue_depth = queueSize;

242         /*
243          * Size of reply descriptor post queue should be the number of
244          * request frames + the number of reply frames + 1 and needs to
245          * be a multiple of 16.  This size can be no larger than
246          * MaxReplyDescriptorPostQueueDepth from IOCFacts.  If the
247          * calculated queue size is larger than allowed, subtract a
248          * multiple of 16 from m_max_requests, m_max_replies, and
249          * m_reply_free_depth.
250          */
251         queueSize = mpt->m_max_requests + numReplyFrames + 1;
252         if (queueSize % 16) {
253                 queueSize += 16 - (queueSize % 16);
254         }
255         mpt->m_post_queue_depth = ddi_get16(accessp,
256             &factsreply->MaxReplyDescriptorPostQueueDepth);
257         if (queueSize > mpt->m_post_queue_depth) {
258                 queueDiff = queueSize - mpt->m_post_queue_depth;
259                 if (queueDiff % 16) {
260                         queueDiff += 16 - (queueDiff % 16);
261                 }
262                 mpt->m_max_requests -= queueDiff;
263                 mpt->m_max_replies -= queueDiff;
264                 mpt->m_free_queue_depth -= queueDiff;
265                 queueSize -= queueDiff;
266         }
267         mpt->m_post_queue_depth = queueSize;

269         /*
270          * Set up max chain depth.
271          */
272         mpt->m_max_chain_depth = ddi_get8(accessp,
273             &factsreply->MaxChainDepth);
274         mpt->m_ioc_capabilities = ddi_get32(accessp,
275             &factsreply->IOCCapabilities);

277         /*
278          * Set flag to check for SAS3 support.
279          */
280         msgversion = ddi_get16(accessp, &factsreply->MsgVersion);
281         if (msgversion == MPI2_VERSION_02_05) {
282                 mptsas_log(mpt, CE_NOTE, "?mpt_sas%d SAS 3 Supported\n",
283                     mpt->m_instance);
284                 mpt->m_MPI25 = TRUE;
285         } else {
286                 mptsas_log(mpt, CE_NOTE, "?mpt%d MPI Version 0x%x\n",
287                     mpt->m_instance, msgversion);
288         }

290         /*
291          * Calculate max frames per request based on DMA S/G length.
292          */
293         simple_sge_main = MPTSAS_MAX_FRAME_SGES64(mpt) - 1;
294         simple_sge_next = mpt->m_req_frame_size /
295             (mpt->m_MPI25 ? sizeof (MPI2_IEEE_SGE_SIMPLE64) :
296             sizeof (MPI2_SGE_SIMPLE64)) - 1;
280             sizeof (MPI2_SGE_SIMPLE64) - 1;

298         mpt->m_max_request_frames = (MPTSAS_MAX_DMA_SEGS -
299             simple_sge_main) / simple_sge_next + 1;
300         if (((MPTSAS_MAX_DMA_SEGS - simple_sge_main) %
301             simple_sge_next) > 1) {
302                 mpt->m_max_request_frames++;
303         }
```

```
305         /*
306          * Check if controller supports FW diag buffers and set flag to enable
307          * each type.
308          */
309         capabilities = ddi_get32(accessp, &factsreply->IOCCapabilities);
310         if (capabilities & MPI2_IOCFACTS_CAPABILITY_DIAG_TRACE_BUFFER) {
311                 mpt->m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_TRACE].enabled =
312                     TRUE;
313         }
314         if (capabilities & MPI2_IOCFACTS_CAPABILITY_SNAPSHOT_BUFFER) {
315                 mpt->m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_SNAPSHOT].
316                     enabled = TRUE;
317         }
318         if (capabilities & MPI2_IOCFACTS_CAPABILITY_EXTENDED_BUFFER) {
319                 mpt->m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_EXTENDED].
320                     enabled = TRUE;
321         }

323         /*
324          * Check if controller supports replaying events when issuing Message
325          * Unit Reset and set flag to enable MUR.
326          */
327         if (capabilities & MPI2_IOCFACTS_CAPABILITY_EVENT_REPLAY) {
328                 mpt->m_event_replay = TRUE;
329         }

331         /*
332          * Check if controller supports IR.
333          */
334         if (capabilities & MPI2_IOCFACTS_CAPABILITY_INTEGRATED_RAID) {
335                 mpt->m_ir_capable = TRUE;
336         }

338         return (DDI_SUCCESS);
339 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9514 Tue Jun 17 10:45:14 2014
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_ioctl.h
NEX-1889 upstream
**********************************************************
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright 2014 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.
 */

#ifndef _MPTSAS_IOCTL_H
#define _MPTSAS_IOCTL_H

#ifdef  __cplusplus
```



```
extern "C" {
#endif

#include <sys/types.h>

#define MPTIOCTL                        ('I' << 8)
#define MPTIOCTL_GET_ADAPTER_DATA       (MPTIOCTL | 1)
#define MPTIOCTL_UPDATE_FLASH           (MPTIOCTL | 2)
#define MPTIOCTL_RESET_ADAPTER          (MPTIOCTL | 3)
#define MPTIOCTL_PASS_THRU              (MPTIOCTL | 4)
#define MPTIOCTL_EVENT_QUERY            (MPTIOCTL | 5)
#define MPTIOCTL_EVENT_ENABLE           (MPTIOCTL | 6)
#define MPTIOCTL_EVENT_REPORT           (MPTIOCTL | 7)
#define MPTIOCTL_GET_PCI_INFO           (MPTIOCTL | 8)
#define MPTIOCTL_DIAG_ACTION            (MPTIOCTL | 9)
#define MPTIOCTL_REG_ACCESS             (MPTIOCTL | 10)
#define MPTIOCTL_GET_DISK_INFO          (MPTIOCTL | 11)
#define MPTIOCTL_LED_CONTROL            (MPTIOCTL | 12)

/*
 *  The following are our ioctl() return status values.  If everything went
 *  well, we return good status.  If the buffer length sent to us is too short
 *  we return a status to tell the user.
 */
#define MPTIOCTL_STATUS_GOOD            0
#define MPTIOCTL_STATUS_LEN_TOO_SHORT   1

typedef struct mptsas_pci_bits
{
        union {
                struct {
                        uint32_t        DeviceNumber    :5;
                        uint32_t        FunctionNumber  :3;
                        uint32_t        BusNumber       :24;
                } bits;
                uint32_t        AsDWORD;
        } u;
        uint32_t        PciSegmentId;
} mptsas_pci_bits_t;
/*
 *  The following is the MPTIOCTL_GET_ADAPTER_DATA data structure.  This data
 *  structure is setup so that we hopefully are properly aligned for both
 *  32-bit and 64-bit mode applications.
 *
 *  Adapter Type - Value = 4 = SCSI Protocol through SAS-2 adapter
 *                 Value = 6 = SCSI Protocol through SAS-3 adapter
 *
 *  MPI Port Number - The PCI Function number for this device
 *
 *  PCI Device HW Id - The PCI device number for this device
 *
 */
#define MPTIOCTL_ADAPTER_TYPE_SAS2      4
#define MPTIOCTL_ADAPTER_TYPE_SAS3      6

typedef struct mptsas_adapter_data
{
        uint32_t                StructureLength;
        uint32_t                AdapterType;
        uint32_t                MpiPortNumber;
        uint32_t                PCIDeviceHwId;
        uint32_t                PCIDeviceHwRev;
        uint32_t                SubSystemId;
        uint32_t                SubsystemVendorId;
        uint32_t                Reserved1;
        uint32_t                MpiFirmwareVersion;
```

```
128         uint32_t                BiosVersion;
129         uint8_t                 DriverVersion[32];
130         uint8_t                 Reserved2;
131         uint8_t                 ScsiId;
132         uint16_t                Reserved3;
133         mptsas_pci_bits_t       PciInformation;
134 } mptsas_adapter_data_t;
```

unchanged_portion_omitted_

```
**********************************************************
   45097 Tue Jun 17 10:45:14 2014
new/usr/src/uts/common/sys/scsi/adapters/mpt_sas/mptsas_var.h
NEX-1889 upstream
**********************************************************
```

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

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2014, Tegile Systems Inc. All rights reserved.
 * Copyright (c) 2013, Joyent, 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.
 */

#ifndef _SYS_SCSI_ADAPTERS_MPTVAR_H
#define _SYS_SCSI_ADAPTERS_MPTVAR_H

#include <sys/byteorder.h>
#include <sys/queue.h>
```



```
#include <sys/isa_defs.h>
#include <sys/sunmdi.h>
#include <sys/mdi_impldefs.h>
#include <sys/scsi/adapters/mpt_sas/mptsas_hash.h>
#include <sys/scsi/adapters/mpt_sas/mptsas_ioctl.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_tool.h>
#include <sys/scsi/adapters/mpt_sas/mpi/mpi2_cnfg.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Compile options
 */
#ifdef DEBUG
#define MPTSAS_DEBUG            /* turn on debugging code */
#endif  /* DEBUG */

#define MPTSAS_INITIAL_SOFT_SPACE       4

#define MAX_MPI_PORTS           16

/*
 * Note below macro definition and data type definition
 * are used for phy mask handling, it should be changed
 * simultaneously.
 */
#define MPTSAS_MAX_PHYS         16
typedef uint16_t                mptsas_phymask_t;

#define MPTSAS_INVALID_DEVHDL   0xffff
#define MPTSAS_SATA_GUID        "sata-guid"

/*
 * Hash table sizes for SMP targets (i.e., expanders) and ordinary SSP/STP
 * targets.  There's no need to go overboard here, as the ordinary paths for
 * I/O do not normally require hashed target lookups.  These should be good
 * enough and then some for any fabric within the hardware's capabilities.
 */
#define MPTSAS_SMP_BUCKET_COUNT         23
#define MPTSAS_TARGET_BUCKET_COUNT      97

/*
 * MPT HW defines
 */
#define MPTSAS_MAX_DISKS_IN_CONFIG      14
#define MPTSAS_MAX_DISKS_IN_VOL         10
#define MPTSAS_MAX_HOTSPARES            2
#define MPTSAS_MAX_RAIDVOLS             2
#define MPTSAS_MAX_RAIDCONFIGS          5

/*
 * 64-bit SAS WWN is displayed as 16 characters as HEX characters,
 * plus two means the prefix 'w' and end of the string '\0'.
 */
#define MPTSAS_WWN_STRLEN       (16 + 2)
#define MPTSAS_MAX_GUID_LEN     64

/*
 * DMA routine flags
 */
#define MPTSAS_DMA_HANDLE_ALLOCD        0x2
#define MPTSAS_DMA_MEMORY_ALLOCD        0x4
#define MPTSAS_DMA_HANDLE_BOUND 0x8
```

```
 127 /*
 128  * If the HBA supports DMA or bus-mastering, you may have your own
 129  * scatter-gather list for physically non-contiguous memory in one
 130  * I/O operation; if so, there's probably a size for that list.
 131  * It must be placed in the ddi_dma_lim_t structure, so that the system
 132  * DMA-support routines can use it to break up the I/O request, so we
 133  * define it here.
 134  */
 135 #if defined(__sparc)
 136 #define MPTSAS_MAX_DMA_SEGS     1
 137 #define MPTSAS_MAX_CMD_SEGS     1
 138 #else
 139 #define MPTSAS_MAX_DMA_SEGS     256
 140 #define MPTSAS_MAX_CMD_SEGS     257
 141 #endif
 142 #define MPTSAS_MAX_FRAME_SGES(mpt) \
 143         (((mpt->m_req_frame_size - (sizeof (MPI2_SCSI_IO_REQUEST))) / 8) + 1)

 145 /*
 146  * Calculating how many 64-bit DMA simple elements can be stored in the first
 145  * Caculating how many 64-bit DMA simple elements can be stored in the first
 147  * frame. Note that msg_scsi_io_request contains 2 double-words (8 bytes) for
 148  * element storage.  And 64-bit dma element is 3 double-words (12 bytes) in
 149  * size. IEEE 64-bit dma element used for SAS3 controllers is 4 double-words
 150  * (16 bytes).
 148  * size.
 151  */
 152 #define MPTSAS_MAX_FRAME_SGES64(mpt) \
 153         ((mpt->m_req_frame_size - \
 154         sizeof (MPI2_SCSI_IO_REQUEST) + sizeof (MPI2_SGE_IO_UNION)) / \
 155         (mpt->m_MPI25 ? sizeof (MPI2_IEEE_SGE_SIMPLE64) : \
 156         sizeof (MPI2_SGE_SIMPLE64)))
 152         (sizeof (MPI2_SCSI_IO_REQUEST)) + sizeof (MPI2_SGE_IO_UNION)) / 12)

 158 /*
 159  * Scatter-gather list structure defined by HBA hardware
 160  */
 161 typedef struct NcrTableIndirect {       /* Table Indirect entries */
 162         uint32_t count;         /* 24 bit count */
 163         union {
 164                 uint32_t address32;     /* 32 bit address */
 165                 struct {
 166                         uint32_t Low;
 167                         uint32_t High;
 168                 } address64;            /* 64 bit address */
 169         } addr;
 170 } mptti_t;
_____unchanged_portion_omitted_

 208 TAILQ_HEAD(mptsas_active_cmdq, mptsas_cmd);
 209 typedef struct mptsas_active_cmdq mptsas_active_cmdq_t;

 211 typedef struct mptsas_target {
 212                 mptsas_target_addr_t    m_addr;
 213                 refhash_link_t          m_link;
 214                 uint8_t                 m_dr_flag;
 215                 uint16_t                m_devhdl;
 216                 uint32_t                m_deviceinfo;
 217                 uint8_t                 m_phynum;
 218                 uint32_t                m_dups;
 219                 mptsas_active_cmdq_t    m_active_cmdq;
 220                 int32_t                 m_t_throttle;
 221                 int32_t                 m_t_ncmds;
 222                 int32_t                 m_reset_delay;
 223                 int32_t                 m_t_nwait;
```



```
 225                 uint16_t                m_qfull_retry_interval;
 226                 uint8_t                 m_qfull_retries;
 227                 uint16_t                m_enclosure;
 228                 uint16_t                m_slot_num;
 229                 uint32_t                m_tgt_unconfigured;
 230                 uint8_t                 m_led_status;
 231                 uint8_t                 m_scsi_req_desc_type;

 233 } mptsas_target_t;
_____unchanged_portion_omitted_

 248 typedef struct mptsas_cache_frames {
 249         ddi_dma_handle_t m_dma_hdl;
 250         ddi_acc_handle_t m_acc_hdl;
 251         caddr_t m_frames_addr;
 252         uint64_t m_phys_addr;
 247         uint32_t m_phys_addr;
 253 } mptsas_cache_frames_t;
_____unchanged_portion_omitted_

 370 /*
 371  * passthrough request structure
 372  */
 373 typedef struct mptsas_pt_request {
 374         uint8_t *request;
 375         uint32_t request_size;
 376         uint32_t data_size;
 377         uint32_t dataout_size;
 378         uint8_t direction;
 379         uint8_t simple;
 380         uint16_t sgl_offset;
 373         uint32_t direction;
 381         ddi_dma_cookie_t data_cookie;
 382         ddi_dma_cookie_t dataout_cookie;
 383 } mptsas_pt_request_t;
_____unchanged_portion_omitted_

 679 typedef struct mptsas {
 680         int             m_instance;

 682         struct mptsas *m_next;

 684         scsi_hba_tran_t         *m_tran;
 685         smp_hba_tran_t          *m_smptran;
 686         kmutex_t                m_mutex;
 687         kmutex_t                m_passthru_mutex;
 688         kcondvar_t              m_cv;
 689         kcondvar_t              m_passthru_cv;
 690         kcondvar_t              m_fw_cv;
 691         kcondvar_t              m_config_cv;
 692         kcondvar_t              m_fw_diag_cv;
 693         dev_info_t              *m_dip;

 695         /*
 696          * soft state flags
 697          */
 698         uint_t          m_softstate;

 700         refhash_t       *m_targets;
 701         refhash_t       *m_smp_targets;

 703         m_raidconfig_t  m_raidconfig[MPTSAS_MAX_RAIDCONFIGS];
 704         uint8_t         m_num_raid_configs;

 706         struct mptsas_slots *m_active;  /* outstanding cmds */
```

```
708         mptsas_cmd_t    *m_waitq;       /* cmd queue for active request */
709         mptsas_cmd_t    **m_waitqtail;  /* wait queue tail ptr */

711         kmutex_t        m_tx_waitq_mutex;
712         mptsas_cmd_t    *m_tx_waitq;    /* TX cmd queue for active request */
713         mptsas_cmd_t    **m_tx_waitqtail;       /* tx_wait queue tail ptr */
714         int             m_tx_draining;  /* TX queue draining flag */

716         mptsas_cmd_t    *m_doneq;       /* queue of completed commands */
717         mptsas_cmd_t    **m_donetail;   /* queue tail ptr */

719         /*
720          * variables for helper threads (fan-out interrupts)
721          */
722         mptsas_doneq_thread_list_t      *m_doneq_thread_id;
723         uint32_t                m_doneq_thread_n;
724         uint32_t                m_doneq_thread_threshold;
725         uint32_t                m_doneq_length_threshold;
726         uint32_t                m_doneq_len;
727         kcondvar_t              m_doneq_thread_cv;
728         kmutex_t                m_doneq_mutex;

730         int             m_ncmds;        /* number of outstanding commands */
731         m_event_struct_t *m_ioc_event_cmdq;     /* cmd queue for ioc event */
732         m_event_struct_t **m_ioc_event_cmdtail; /* ioc cmd queue tail */

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

736         struct _MPI2_SYSTEM_INTERFACE_REGS      *m_reg;

738         ushort_t        m_devid;        /* device id of chip. */
739         uchar_t         m_revid;        /* revision of chip. */
740         uint16_t        m_svid;         /* subsystem Vendor ID of chip */
741         uint16_t        m_ssid;         /* subsystem Device ID of chip */

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

745         timeout_id_t    m_quiesce_timeid;

747         ddi_dma_handle_t m_dma_req_frame_hdl;
748         ddi_acc_handle_t m_acc_req_frame_hdl;
749         ddi_dma_handle_t m_dma_reply_frame_hdl;
750         ddi_acc_handle_t m_acc_reply_frame_hdl;
751         ddi_dma_handle_t m_dma_free_queue_hdl;
752         ddi_acc_handle_t m_acc_free_queue_hdl;
753         ddi_dma_handle_t m_dma_post_queue_hdl;
754         ddi_acc_handle_t m_acc_post_queue_hdl;

756         /*
757          * list of reset notification requests
758          */
759         struct scsi_reset_notify_entry  *m_reset_notify_listf;

761         /*
762          * qfull handling
763          */
764         timeout_id_t    m_restart_cmd_timeid;

766         /*
767          * scsi reset delay per bus
768          */
769         uint_t          m_scsi_reset_delay;

771         int             m_pm_idle_delay;

773         uchar_t         m_polled_intr;  /* intr was polled. */
```



```
774         uchar_t         m_suspended;    /* true if driver is suspended */

776         struct kmem_cache *m_kmem_cache;
777         struct kmem_cache *m_cache_frames;

779         /*
780          * hba options.
781          */
782         uint_t          m_options;

784         int             m_in_callback;

786         int             m_power_level;  /* current power level */

788         int             m_busy;         /* power management busy state */

790         off_t           m_pmcsr_offset; /* PMCSR offset */

792         ddi_acc_handle_t m_config_handle;

794         ddi_dma_attr_t          m_io_dma_attr;  /* Used for data I/O */
795         ddi_dma_attr_t          m_msg_dma_attr; /* Used for message frames */
796         ddi_device_acc_attr_t   m_dev_acc_attr;
797         ddi_device_acc_attr_t   m_reg_acc_attr;

799         /*
800          * request/reply variables
801          */
802         caddr_t         m_req_frame;
803         uint64_t        m_req_frame_dma_addr;
804         caddr_t         m_reply_frame;
805         uint64_t        m_reply_frame_dma_addr;
806         caddr_t         m_free_queue;
807         uint64_t        m_free_queue_dma_addr;
808         caddr_t         m_post_queue;
809         uint64_t        m_post_queue_dma_addr;

811         m_replyh_arg_t *m_replyh_args;

813         uint16_t        m_max_requests;
814         uint16_t        m_req_frame_size;

816         /*
817          * Max frames per request reprted in IOC Facts
818          */
819         uint8_t         m_max_chain_depth;
820         /*
821          * Max frames per request which is used in reality. It's adjusted
822          * according DMA SG length attribute, and shall not exceed the
823          * m_max_chain_depth.
824          */
825         uint8_t         m_max_request_frames;

827         uint16_t        m_free_queue_depth;
828         uint16_t        m_post_queue_depth;
829         uint16_t        m_max_replies;
830         uint32_t        m_free_index;
831         uint32_t        m_post_index;
832         uint8_t         m_reply_frame_size;
833         uint32_t        m_ioc_capabilities;

835         /*
836          * indicates if the firmware was upload by the driver
837          * at boot time
838          */
839         ushort_t        m_fwupload;
```

```
841         uint16_t        m_productid;

843         /*
844          * per instance data structures for dma memory resources for
845          * MPI handshake protocol. only one handshake cmd can run at a time.
846          */
847         ddi_dma_handle_t        m_hshk_dma_hdl;
848         ddi_acc_handle_t        m_hshk_acc_hdl;
849         caddr_t                 m_hshk_memp;
850         size_t                  m_hshk_dma_size;

852         /* Firmware version on the card at boot time */
853         uint32_t                m_fwversion;

855         /* MSI specific fields */
856         ddi_intr_handle_t       *m_htable;      /* For array of interrupts */
857         int                     m_intr_type;    /* What type of interrupt */
858         int                     m_intr_cnt;     /* # of intrs count returned */
859         size_t                  m_intr_size;    /* Size of intr array */
860         uint_t                  m_intr_pri;     /* Interrupt priority   */
861         int                     m_intr_cap;     /* Interrupt capabilities */
862         ddi_taskq_t             *m_event_taskq;

864         /* SAS specific information */

866         union {
867                 uint64_t        m_base_wwid;    /* Base WWID */
868                 struct {
869 #ifdef _BIG_ENDIAN
870                         uint32_t        m_base_wwid_hi;
871                         uint32_t        m_base_wwid_lo;
872 #else
873                         uint32_t        m_base_wwid_lo;
874                         uint32_t        m_base_wwid_hi;
875 #endif
876                 } sasaddr;
877         } un;

879         uint8_t                 m_num_phys;             /* # of PHYs */
880         mptsas_phy_info_t       m_phy_info[MPTSAS_MAX_PHYS];
881         uint8_t                 m_port_chng;    /* initiator port changes */
882         MPI2_CONFIG_PAGE_MAN_0   m_MANU_page0;   /* Manufactor page 0 info */
883         MPI2_CONFIG_PAGE_MAN_1   m_MANU_page1;   /* Manufactor page 1 info */

885         /* FMA Capabilities */
886         int                     m_fm_capabilities;
887         ddi_taskq_t             *m_dr_taskq;
888         int                     m_mpxio_enable;
889         uint8_t                 m_done_traverse_dev;
890         uint8_t                 m_done_traverse_smp;
891         int                     m_diag_action_in_progress;
892         uint16_t                m_dev_handle;
893         uint16_t                m_smp_devhdl;

895         /*
896          * Event recording
897          */
898         uint8_t                 m_event_index;
899         uint32_t                m_event_number;
900         uint32_t                m_event_mask[4];
901         mptsas_event_entry_t    m_events[MPTSAS_EVENT_QUEUE_SIZE];

903         /*
904          * FW diag Buffer List
905          */
```



```
906         mptsas_fw_diagnostic_buffer_t
907                 m_fw_diag_buffer_list[MPI2_DIAG_BUF_TYPE_COUNT];

909         /* GEN3 support */
910         uint8_t                 m_MPI25;

912         /*
913          * Event Replay flag (MUR support)
914          */
915         uint8_t                 m_event_replay;

917         /*
918          * IR Capable flag
919          */
920         uint8_t                 m_ir_capable;

922         /*
923          * Is HBA processing a diag reset?
924          */
925         uint8_t                 m_in_reset;

927         /*
928          * per instance cmd data structures for task management cmds
929          */
930         m_event_struct_t        m_event_task_mgmt;      /* must be last */
931                                                         /* ... scsi_pkt_size */
932 } mptsas_t;
```

*unchanged_portion_omitted_*

```
1049 /*
1050  * inq_dtype:
1051  * Bits 5 through 7 are the Peripheral Device Qualifier
1052  * 001b: device not connected to the LUN
1053  * Bits 0 through 4 are the Peripheral Device Type
1054  * 1fh: Unknown or no device type
1055  *
1056  * Although the inquiry may return success, the following value
1057  * means no valid LUN connected.
1058  */
1059 #define MPTSAS_VALID_LUN(sd_inq) \
1060         (((sd_inq->inq_dtype & 0xe0) != 0x20) && \
1061         ((sd_inq->inq_dtype & 0x1f) != 0x1f))

1063 /*
1064  * Default is to have 10 retries on receiving QFULL status and
1065  * each retry to be after 100 ms.
1066  */
1067 #define QFULL_RETRIES           10
1068 #define QFULL_RETRY_INTERVAL    100

1070 /*
1071  * Handy macros
1072  */
1073 #define Tgt(sp) ((sp)->cmd_pkt->pkt_address.a_target)
1074 #define Lun(sp) ((sp)->cmd_pkt->pkt_address.a_lun)

1076 #define IS_HEX_DIGIT(n) (((n) >= '0' && (n) <= '9') || \
1077         ((n) >= 'a' && (n) <= 'f') || ((n) >= 'A' && (n) <= 'F'))

1079 /*
1080  * poll time for mptsas_pollret() and mptsas_wait_intr()
1081  */
1082 #define MPTSAS_POLL_TIME        30000   /* 30 seconds */

1084 /*
1085  * default time for mptsas_do_passthru
```

```
 */
#define MPTSAS_PASS_THRU_TIME_DEFAULT   60      /* 60 seconds */

/*
 * macro to return the effective address of a given per-target field
 */
#define EFF_ADDR(start, offset)         ((start) + (offset))

#define SDEV2ADDR(devp)         (&((devp)->sd_address))
#define SDEV2TRAN(devp)         ((devp)->sd_address.a_hba_tran)
#define PKT2TRAN(pkt)           ((pkt)->pkt_address.a_hba_tran)
#define ADDR2TRAN(ap)           ((ap)->a_hba_tran)
#define DIP2TRAN(dip)           (ddi_get_driver_private(dip))


#define TRAN2MPT(hba)           ((mptsas_t *)(hba)->tran_hba_private)
#define DIP2MPT(dip)            (TRAN2MPT((scsi_hba_tran_t *)DIP2TRAN(dip)))
#define SDEV2MPT(sd)            (TRAN2MPT(SDEV2TRAN(sd)))
#define PKT2MPT(pkt)            (TRAN2MPT(PKT2TRAN(pkt)))

#define ADDR2MPT(ap)            (TRAN2MPT(ADDR2TRAN(ap)))

#define POLL_TIMEOUT            (2 * SCSI_POLL_TIMEOUT * 1000000)
#define SHORT_POLL_TIMEOUT      (1000000)       /* in usec, about 1 secs */
#define MPTSAS_QUIESCE_TIMEOUT  1               /* 1 sec */
#define MPTSAS_PM_IDLE_TIMEOUT  60              /* 60 seconds */

#define MPTSAS_GET_ISTAT(mpt)  (ddi_get32((mpt)->m_datap, \
                        &(mpt)->m_reg->HostInterruptStatus))

#define MPTSAS_SET_SIGP(P) \
                ClrSetBits(mpt->m_devaddr + NREG_ISTAT, 0, NB_ISTAT_SIGP)

#define MPTSAS_RESET_SIGP(P) (void) ddi_get8(mpt->m_datap, \
                        (uint8_t *)(mpt->m_devaddr + NREG_CTEST2))

#define MPTSAS_GET_INTCODE(P) (ddi_get32(mpt->m_datap, \
                        (uint32_t *)(mpt->m_devaddr + NREG_DSPS)))


#define MPTSAS_START_CMD(mpt, req_desc_lo, req_desc_hi) \
        ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostLow,\
            req_desc_lo);\
        ddi_put32(mpt->m_datap, &mpt->m_reg->RequestDescriptorPostHigh,\
            req_desc_hi);

#define INTPENDING(mpt) \
        (MPTSAS_GET_ISTAT(mpt) & MPI2_HIS_REPLY_DESCRIPTOR_INTERRUPT)

/*
 * Mask all interrupts to disable
 */
#define MPTSAS_DISABLE_INTR(mpt)        \
        ddi_put32((mpt)->m_datap, &(mpt)->m_reg->HostInterruptMask, \
            (MPI2_HIM_RIM | MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

/*
 * Mask Doorbell and Reset interrupts to enable reply desc int.
 */
#define MPTSAS_ENABLE_INTR(mpt) \
        ddi_put32(mpt->m_datap, &mpt->m_reg->HostInterruptMask, \
        (MPI2_HIM_DIM | MPI2_HIM_RESET_IRQ_MASK))

#define MPTSAS_GET_NEXT_REPLY(mpt, index)  \
        &((uint64_t *)(void *)mpt->m_post_queue)[index]
```



```
#define MPTSAS_GET_NEXT_FRAME(mpt, SMID) \
        (mpt->m_req_frame + (mpt->m_req_frame_size * SMID))

#define ClrSetBits32(hdl, reg, clr, set) \
        ddi_put32(hdl, (reg), \
            ((ddi_get32(mpt->m_datap, (reg)) & ~(clr)) | (set)))

#define ClrSetBits(reg, clr, set) \
        ddi_put8(mpt->m_datap, (uint8_t *)(reg), \
                ((ddi_get8(mpt->m_datap, (uint8_t *)(reg)) & ~(clr)) | (set)))

#define MPTSAS_WAITQ_RM(mpt, cmdp)      \
        if ((cmdp = mpt->m_waitq) != NULL) { \
                /* If the queue is now empty fix the tail pointer */    \
                if ((mpt->m_waitq = cmdp->cmd_linkp) == NULL) \
                        mpt->m_waitqtail = &mpt->m_waitq; \
                cmdp->cmd_linkp = NULL; \
                cmdp->cmd_queued = FALSE; \
        }

#define MPTSAS_TX_WAITQ_RM(mpt, cmdp)   \
        if ((cmdp = mpt->m_tx_waitq) != NULL) { \
                /* If the queue is now empty fix the tail pointer */    \
                if ((mpt->m_tx_waitq = cmdp->cmd_linkp) == NULL) \
                        mpt->m_tx_waitqtail = &mpt->m_tx_waitq; \
                cmdp->cmd_linkp = NULL; \
                cmdp->cmd_queued = FALSE; \
        }

/*
 * defaults for the global properties
 */
#define DEFAULT_SCSI_OPTIONS    SCSI_OPTIONS_DR
#define DEFAULT_TAG_AGE_LIMIT   2
#define DEFAULT_WD_TICK         1

/*
 * invalid hostid.
 */
#define MPTSAS_INVALID_HOSTID  -1

/*
 * Get/Set hostid from SCSI port configuration page
 */
#define MPTSAS_GET_HOST_ID(configuration) (configuration & 0xFF)
#define MPTSAS_SET_HOST_ID(hostid) (hostid | ((1 << hostid) << 16))

/*
 * Config space.
 */
#define MPTSAS_LATENCY_TIMER    0x40

/*
 * Offset to firmware version
 */
#define MPTSAS_FW_VERSION_OFFSET        9

/*
 * Offset and masks to get at the ProductId field
 */
#define MPTSAS_FW_PRODUCTID_OFFSET      8
#define MPTSAS_FW_PRODUCTID_MASK        0xFFFF0000
#define MPTSAS_FW_PRODUCTID_SHIFT       16

/*
 * Subsystem ID for HBAs.
```

```
 */
#define MPTSAS_HBA_SUBSYSTEM_ID    0x10C0
#define MPTSAS_RHEA_SUBSYSTEM_ID        0x10B0

/*
 * reset delay tick
 */
#define MPTSAS_WATCH_RESET_DELAY_TICK 50        /* specified in milli seconds */

/*
 * Ioc reset return values
 */
#define MPTSAS_RESET_FAIL       -1
#define MPTSAS_NO_RESET         0
#define MPTSAS_SUCCESS_HARDRESET        1
#define MPTSAS_SUCCESS_MUR      2

/*
 * throttle support.
 */
#define MAX_THROTTLE    32
#define HOLD_THROTTLE   0
#define DRAIN_THROTTLE  -1
#define QFULL_THROTTLE  -2

/*
 * Passthrough/config request flags
 */
#define MPTSAS_DATA_ALLOCATED           0x0001
#define MPTSAS_DATAOUT_ALLOCATED        0x0002
#define MPTSAS_REQUEST_POOL_CMD         0x0004
#define MPTSAS_ADDRESS_REPLY            0x0008
#define MPTSAS_CMD_TIMEOUT              0x0010

/*
 * response code tlr flag
 */
#define MPTSAS_SCSI_RESPONSE_CODE_TLR_OFF       0x02

/*
 * System Events
 */
#ifndef DDI_VENDOR_LSI
#define DDI_VENDOR_LSI  "LSI"
#endif  /* DDI_VENDOR_LSI */

/*
 * Shared functions
 */
int mptsas_save_cmd(struct mptsas *mpt, struct mptsas_cmd *cmd);
void mptsas_remove_cmd(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_waitq_add(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_log(struct mptsas *mpt, int level, char *fmt, ...);
int mptsas_poll(mptsas_t *mpt, mptsas_cmd_t *poll_cmd, int polltime);
int mptsas_do_dma(mptsas_t *mpt, uint32_t size, int var, int (*callback)());
int mptsas_send_config_request_msg(mptsas_t *mpt, uint8_t action,
        uint8_t pagetype, uint32_t pageaddress, uint8_t pagenumber,
        uint8_t pageversion, uint8_t pagelength, uint32_t
        SGEflagslength, uint32_t SGEaddress32);
int mptsas_send_extended_config_request_msg(mptsas_t *mpt, uint8_t action,
        uint8_t extpagetype, uint32_t pageaddress, uint8_t pagenumber,
        uint8_t pageversion, uint16_t extpagelength,
        uint32_t SGEflagslength, uint32_t SGEaddress32);
int mptsas_update_flash(mptsas_t *mpt, caddr_t ptrbuffer, uint32_t size,
        uint8_t type, int mode);
int mptsas_check_flash(mptsas_t *mpt, caddr_t origfile, uint32_t size,
```



```
        uint8_t type, int mode);
int mptsas_download_firmware();
int mptsas_can_download_firmware();
int mptsas_dma_alloc(mptsas_t *mpt, mptsas_dma_alloc_state_t *dma_statep);
void mptsas_dma_free(mptsas_dma_alloc_state_t *dma_statep);
mptsas_phymask_t mptsas_physport_to_phymask(mptsas_t *mpt, uint8_t physport);
void mptsas_fma_check(mptsas_t *mpt, mptsas_cmd_t *cmd);
int mptsas_check_acc_handle(ddi_acc_handle_t handle);
int mptsas_check_dma_handle(ddi_dma_handle_t handle);
void mptsas_fm_ereport(mptsas_t *mpt, char *detail);
int mptsas_dma_addr_create(mptsas_t *mpt, ddi_dma_attr_t dma_attr,
    ddi_dma_handle_t *dma_hdp, ddi_acc_handle_t *acc_hdp, caddr_t *dma_memp,
    uint32_t alloc_size, ddi_dma_cookie_t *cookiep);
void mptsas_dma_addr_destroy(ddi_dma_handle_t *, ddi_acc_handle_t *);

/*
 * impl functions
 */
int mptsas_ioc_wait_for_response(mptsas_t *mpt);
int mptsas_ioc_wait_for_doorbell(mptsas_t *mpt);
int mptsas_ioc_reset(mptsas_t *mpt, int);
int mptsas_send_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
    ddi_acc_handle_t accessp);
int mptsas_get_handshake_msg(mptsas_t *mpt, caddr_t memp, int numbytes,
    ddi_acc_handle_t accessp);
int mptsas_send_config_request_msg(mptsas_t *mpt, uint8_t action,
    uint8_t pagetype, uint32_t pageaddress, uint8_t pagenumber,
    uint8_t pageversion, uint8_t pagelength, uint32_t SGEflagslength,
    uint32_t SGEaddress32);
int mptsas_send_extended_config_request_msg(mptsas_t *mpt, uint8_t action,
    uint8_t extpagetype, uint32_t pageaddress, uint8_t pagenumber,
    uint8_t pageversion, uint16_t extpagelength,
    uint32_t SGEflagslength, uint32_t SGEaddress32);

int mptsas_request_from_pool(mptsas_t *mpt, mptsas_cmd_t **cmd,
    struct scsi_pkt **pkt);
void mptsas_return_to_pool(mptsas_t *mpt, mptsas_cmd_t *cmd);
void mptsas_destroy_ioc_event_cmd(mptsas_t *mpt);
void mptsas_start_config_page_access(mptsas_t *mpt, mptsas_cmd_t *cmd);
int mptsas_access_config_page(mptsas_t *mpt, uint8_t action, uint8_t page_type,
    uint8_t page_number, uint32_t page_address, int (*callback) (mptsas_t *,
    caddr_t, ddi_acc_handle_t, uint16_t, uint32_t, va_list), ...);

int mptsas_ioc_task_management(mptsas_t *mpt, int task_type,
    uint16_t dev_handle, int lun, uint8_t *reply, uint32_t reply_size,
    int mode);
int mptsas_send_event_ack(mptsas_t *mpt, uint32_t event, uint32_t eventcntx);
void mptsas_send_pending_event_ack(mptsas_t *mpt);
void mptsas_set_throttle(struct mptsas *mpt, mptsas_target_t *ptgt, int what);
int mptsas_restart_ioc(mptsas_t *mpt);
void mptsas_update_driver_data(struct mptsas *mpt);
uint64_t mptsas_get_sata_guid(mptsas_t *mpt, mptsas_target_t *ptgt, int lun);

/*
 * init functions
 */
int mptsas_ioc_get_facts(mptsas_t *mpt);
int mptsas_ioc_get_port_facts(mptsas_t *mpt, int port);
int mptsas_ioc_enable_port(mptsas_t *mpt);
int mptsas_ioc_enable_event_notification(mptsas_t *mpt);
int mptsas_ioc_init(mptsas_t *mpt);

/*
 * configuration pages operation
 */
int mptsas_get_sas_device_page0(mptsas_t *mpt, uint32_t page_address,
```

```
1350     uint16_t *dev_handle, uint64_t *sas_wwn, uint32_t *dev_info,
1351     uint8_t *physport, uint8_t *phynum, uint16_t *pdevhandle,
1352     uint16_t *slot_num, uint16_t *enclosure);
1353 int mptsas_get_sas_io_unit_page(mptsas_t *mpt);
1354 int mptsas_get_sas_io_unit_page_hndshk(mptsas_t *mpt);
1355 int mptsas_get_sas_expander_page0(mptsas_t *mpt, uint32_t page_address,
1356     mptsas_smp_t *info);
1357 int mptsas_set_ioc_params(mptsas_t *mpt);
1358 int mptsas_get_manufacture_page5(mptsas_t *mpt);
1359 int mptsas_get_sas_port_page0(mptsas_t *mpt, uint32_t page_address,
1360     uint64_t *sas_wwn, uint8_t *portwidth);
1361 int mptsas_get_bios_page3(mptsas_t *mpt,  uint32_t *bios_version);
1362 int
1363 mptsas_get_sas_phy_page0(mptsas_t *mpt, uint32_t page_address,
1364     smhba_info_t *info);
1365 int
1366 mptsas_get_sas_phy_page1(mptsas_t *mpt, uint32_t page_address,
1367     smhba_info_t *info);
1368 int
1369 mptsas_get_manufacture_page0(mptsas_t *mpt);
1370 void
1371 mptsas_create_phy_stats(mptsas_t *mpt, char *iport, dev_info_t *dip);
1372 void mptsas_destroy_phy_stats(mptsas_t *mpt);
1373 int mptsas_smhba_phy_init(mptsas_t *mpt);
1374 /*
1375  * RAID functions
1376  */
1377 int mptsas_get_raid_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol);
1378 int mptsas_get_raid_info(mptsas_t *mpt);
1379 int mptsas_get_physdisk_settings(mptsas_t *mpt, mptsas_raidvol_t *raidvol,
1380     uint8_t physdisknum);
1381 int mptsas_delete_volume(mptsas_t *mpt, uint16_t volid);
1382 void mptsas_raid_action_system_shutdown(mptsas_t *mpt);

1384 #define MPTSAS_IOCSTATUS(status) (status & MPI2_IOCSTATUS_MASK)
1385 /*
1386  * debugging.
1387  */
1388 #if defined(MPTSAS_DEBUG)

1390 void mptsas_printf(char *fmt, ...);

1392 #define MPTSAS_DBGPR(m, args)   \
1393         if (mptsas_debug_flags & (m)) \
1394                 mptsas_printf args
1395 #else   /* ! defined(MPTSAS_DEBUG) */
1396 #define MPTSAS_DBGPR(m, args)
1397 #endif  /* defined(MPTSAS_DEBUG) */

1399 #define NDBG0(args)     MPTSAS_DBGPR(0x01, args)        /* init */
1400 #define NDBG1(args)     MPTSAS_DBGPR(0x02, args)        /* normal running */
1401 #define NDBG2(args)     MPTSAS_DBGPR(0x04, args)        /* property handling */
1402 #define NDBG3(args)     MPTSAS_DBGPR(0x08, args)        /* pkt handling */

1404 #define NDBG4(args)     MPTSAS_DBGPR(0x10, args)        /* kmem alloc/free */
1405 #define NDBG5(args)     MPTSAS_DBGPR(0x20, args)        /* polled cmds */
1406 #define NDBG6(args)     MPTSAS_DBGPR(0x40, args)        /* interrupts */
1407 #define NDBG7(args)     MPTSAS_DBGPR(0x80, args)        /* queue handling */

1409 #define NDBG8(args)     MPTSAS_DBGPR(0x0100, args)      /* arq */
1410 #define NDBG9(args)     MPTSAS_DBGPR(0x0200, args)      /* Tagged Q'ing */
1411 #define NDBG10(args)    MPTSAS_DBGPR(0x0400, args)      /* halting chip */
1412 #define NDBG11(args)    MPTSAS_DBGPR(0x0800, args)      /* power management */

1414 #define NDBG12(args)    MPTSAS_DBGPR(0x1000, args)      /* enumeration */
1415 #define NDBG13(args)    MPTSAS_DBGPR(0x2000, args)      /* configuration page */
```



```
1416 #define NDBG14(args)    MPTSAS_DBGPR(0x4000, args)      /* LED control */
1417 #define NDBG15(args)    MPTSAS_DBGPR(0x8000, args)      /* Passthrough */
1407 #define NDBG15(args)    MPTSAS_DBGPR(0x8000, args)

1419 #define NDBG16(args)    MPTSAS_DBGPR(0x010000, args)
1420 #define NDBG17(args)    MPTSAS_DBGPR(0x020000, args)    /* scatter/gather */
1421 #define NDBG18(args)    MPTSAS_DBGPR(0x040000, args)
1422 #define NDBG19(args)    MPTSAS_DBGPR(0x080000, args)    /* handshaking */

1424 #define NDBG20(args)    MPTSAS_DBGPR(0x100000, args)    /* events */
1425 #define NDBG21(args)    MPTSAS_DBGPR(0x200000, args)    /* dma */
1426 #define NDBG22(args)    MPTSAS_DBGPR(0x400000, args)    /* reset */
1427 #define NDBG23(args)    MPTSAS_DBGPR(0x800000, args)    /* abort */

1429 #define NDBG24(args)    MPTSAS_DBGPR(0x1000000, args)   /* capabilities */
1430 #define NDBG25(args)    MPTSAS_DBGPR(0x2000000, args)   /* flushing */
1431 #define NDBG26(args)    MPTSAS_DBGPR(0x4000000, args)
1432 #define NDBG27(args)    MPTSAS_DBGPR(0x8000000, args)

1434 #define NDBG28(args)    MPTSAS_DBGPR(0x10000000, args)  /* hotplug */
1435 #define NDBG29(args)    MPTSAS_DBGPR(0x20000000, args)  /* timeouts */
1436 #define NDBG30(args)    MPTSAS_DBGPR(0x40000000, args)  /* mptsas_watch */
1437 #define NDBG31(args)    MPTSAS_DBGPR(0x80000000, args)  /* negotations */

1439 /*
1440  * auto request sense
1441  */
1442 #define RQ_MAKECOM_COMMON(pkt, flag, cmd) \
1443         (pkt)->pkt_flags = (flag), \
1444         ((union scsi_cdb *)(pkt)->pkt_cdbp)->scc_cmd = (cmd), \
1445         ((union scsi_cdb *)(pkt)->pkt_cdbp)->scc_lun = \
1446             (pkt)->pkt_address.a_lun

1448 #define RQ_MAKECOM_G0(pkt, flag, cmd, addr, cnt) \
1449         RQ_MAKECOM_COMMON((pkt), (flag), (cmd)), \
1450         FORMG0ADDR(((union scsi_cdb *)(pkt)->pkt_cdbp), (addr)), \
1451         FORMG0COUNT(((union scsi_cdb *)(pkt)->pkt_cdbp), (cnt))


1454 #ifdef  __cplusplus
1455 }
```

_____***unchanged_portion_omitted_***