```
**********************************************************
    2449 Tue Nov  6 14:28:50 2012
new/usr/src/pkg/manifests/driver-storage-mr_sas.mf
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
#
# 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 2012 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/mr_sas@$(PKGVERS)
set name=pkg.description value="LSI MegaRAID SAS2.0 Controller HBA Driver"
set name=pkg.summary value="LSI MegaRAID SAS2.0 HBA Driver"
set name=info.classification \
    value=org.opensolaris.category.2008:Drivers/Storage
set name=variant.arch value=$(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
$(sparc_ONLY)driver name=mr_sas class=scsi-self-identifying \
    alias=pci1000,78 \
    alias=pci1000,79 \
    alias=pciex1000,5b \
    alias=pciex1000,5d \
    alias=pciex1000,78 \
    alias=pciex1000,79
$(i386_ONLY)driver name=mr_sas class=scsi-self-identifying \
    alias=pciex1000,5b \
    alias=pciex1000,5d \
    alias=pciex1000,78 \
    alias=pciex1000,79
file path=kernel/drv/$(ARCH64)/mr_sas group=sys
$(i386_ONLY)file path=kernel/drv/mr_sas group=sys
file path=kernel/drv/mr_sas.conf group=sys
file path=usr/share/man/man7d/mr_sas.7d
legacy pkg=SUNWmrsas desc="LSI MegaRAID SAS2.0 Controller HBA Driver" \
    name="LSI MegaRAID SAS2.0 HBA Driver"
```



```
license cr_Sun license=cr_Sun
license usr/src/uts/common/io/mr_sas/THIRDPARTYLICENSE \
    license=usr/src/uts/common/io/mr_sas/THIRDPARTYLICENSE
```

```
**********************************************************
   43013 Tue Nov  6 14:28:50 2012
new/usr/src/uts/common/Makefile.files
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
#
# 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) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2012 Nexenta Systems, Inc. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#

#
# This Makefile defines all file modules for the directory uts/common
# and its children. These are the source files which may be considered
# common to all SunOS systems.

i386_CORE_OBJS += \
                atomic.o        \
                avintr.o        \
                pic.o

sparc_CORE_OBJS +=

COMMON_CORE_OBJS +=             \
                beep.o          \
                bitset.o        \
                bp_map.o        \
                brand.o         \
                cpucaps.o       \
                cmt.o           \
                cmt_policy.o    \
                cpu.o           \
                cpu_event.o     \
                cpu_intr.o      \
                cpu_pm.o        \
                cpupart.o       \
                cap_util.o      \
                disp.o          \
                group.o         \
                kstat_fr.o      \
                iscsiboot_prop.o        \
                lgrp.o          \
                lgrp_topo.o     \
                mmapobj.o       \
                mutex.o         \
```



```
                page_lock.o     \
                page_retire.o   \
                panic.o         \
                param.o         \
                pg.o            \
                pghw.o          \
                putnext.o       \
                rctl_proc.o     \
                rwlock.o        \
                seg_kmem.o      \
                softint.o       \
                string.o        \
                strtol.o        \
                strtoul.o       \
                strtoll.o       \
                strtoull.o      \
                thread_intr.o   \
                vm_page.o       \
                vm_pagelist.o   \
                zlib_obj.o      \
                clock_tick.o

CORE_OBJS +=    $(COMMON_CORE_OBJS) $($(MACH)_CORE_OBJS)

ZLIB_OBJS =     zutil.o zmod.o zmod_subr.o \
                adler32.o crc32.o deflate.o inffast.o \
                inflate.o inftrees.o trees.o

GENUNIX_OBJS += \
                access.o        \
                acl.o           \
                acl_common.o    \
                adjtime.o       \
                alarm.o         \
                aio_subr.o      \
                auditsys.o      \
                audit_core.o    \
                audit_zone.o    \
                audit_memory.o  \
                autoconf.o      \
                avl.o           \
                bdev_dsort.o    \
                bio.o           \
                bitmap.o        \
                blabel.o        \
                brandsys.o      \
                bz2blocksort.o  \
                bz2compress.o   \
                bz2decompress.o \
                bz2randtable.o  \
                bz2bzlib.o      \
                bz2crctable.o   \
                bz2huffman.o    \
                callb.o         \
                callout.o       \
                chdir.o         \
                chmod.o         \
                chown.o         \
                cladm.o         \
                class.o         \
                clock.o         \
                clock_highres.o \
                clock_realtime.o\
                close.o         \
                compress.o      \
                condvar.o       \
```

```
128                 conf.o          \
129                 console.o       \
130                 contract.o      \
131                 copyops.o       \
132                 core.o          \
133                 corectl.o       \
134                 cred.o          \
135                 cs_stubs.o      \
136                 dacf.o          \
137                 dacf_clnt.o     \
138                 damap.o \
139                 cyclic.o        \
140                 ddi.o           \
141                 ddifm.o         \
142                 ddi_hp_impl.o   \
143                 ddi_hp_ndi.o    \
144                 ddi_intr.o      \
145                 ddi_intr_impl.o \
146                 ddi_intr_irm.o  \
147                 ddi_nodeid.o    \
148                 ddi_timer.o     \
149                 devcfg.o        \
150                 devcache.o      \
151                 device.o        \
152                 devid.o         \
153                 devid_cache.o   \
154                 devid_scsi.o    \
155                 devid_smp.o     \
156                 devpolicy.o     \
157                 disp_lock.o     \
158                 dnlc.o          \
159                 driver.o        \
160                 dumpsubr.o      \
161                 driver_lyr.o    \
162                 dtrace_subr.o   \
163                 errorq.o        \
164                 etheraddr.o     \
165                 evchannels.o    \
166                 exacct.o        \
167                 exacct_core.o   \
168                 exec.o          \
169                 exit.o          \
170                 fbio.o          \
171                 fcntl.o         \
172                 fdbuffer.o      \
173                 fdsync.o        \
174                 fem.o           \
175                 ffs.o           \
176                 fio.o           \
177                 flock.o         \
178                 fm.o            \
179                 fork.o          \
180                 vpm.o           \
181                 fs_reparse.o    \
182                 fs_subr.o       \
183                 fsflush.o       \
184                 ftrace.o        \
185                 getcwd.o        \
186                 getdents.o      \
187                 getloadavg.o    \
188                 getpagesizes.o  \
189                 getpid.o        \
190                 gfs.o           \
191                 rusagesys.o     \
192                 gid.o           \
193                 groups.o        \
```



```
194                 grow.o          \
195                 hat_refmod.o    \
196                 id32.o          \
197                 id_space.o      \
198                 inet_ntop.o     \
199                 instance.o      \
200                 ioctl.o         \
201                 ip_cksum.o      \
202                 issetugid.o     \
203                 ippconf.o       \
204                 kcpc.o          \
205                 kdi.o           \
206                 kiconv.o        \
207                 klpd.o          \
208                 kmem.o          \
209                 ksyms_snapshot.o        \
210                 l_strplumb.o    \
211                 labelsys.o      \
212                 link.o          \
213                 list.o          \
214                 lockstat_subr.o \
215                 log_sysevent.o  \
216                 logsubr.o       \
217                 lookup.o        \
218                 lseek.o         \
219                 ltos.o          \
220                 lwp.o           \
221                 lwp_create.o    \
222                 lwp_info.o      \
223                 lwp_self.o      \
224                 lwp_sobj.o      \
225                 lwp_timer.o     \
226                 lwpsys.o        \
227                 main.o          \
228                 mmapobjsys.o    \
229                 memcntl.o       \
230                 memstr.o        \
231                 lgrpsys.o       \
232                 mkdir.o         \
233                 mknod.o         \
234                 mount.o         \
235                 move.o          \
236                 msacct.o        \
237                 multidata.o     \
238                 nbmlock.o       \
239                 ndifm.o         \
240                 nice.o          \
241                 netstack.o      \
242                 ntptime.o       \
243                 nvpair.o        \
244                 nvpair_alloc_system.o   \
245                 nvpair_alloc_fixed.o    \
246                 fnvpair.o       \
247                 octet.o         \
248                 open.o          \
249                 p_online.o      \
250                 pathconf.o      \
251                 pathname.o      \
252                 pause.o         \
253                 serializer.o    \
254                 pci_intr_lib.o  \
255                 pci_cap.o       \
256                 pcifm.o         \
257                 pgrp.o          \
258                 pgrpsys.o       \
259                 pid.o           \
```

```
260                 pkp_hash.o      \
261                 policy.o        \
262                 poll.o          \
263                 pool.o          \
264                 pool_pset.o     \
265                 port_subr.o     \
266                 ppriv.o         \
267                 printf.o        \
268                 priocntl.o      \
269                 priv.o          \
270                 priv_const.o    \
271                 proc.o          \
272                 procset.o       \
273                 processor_bind.o        \
274                 processor_info.o        \
275                 profil.o        \
276                 project.o       \
277                 qsort.o         \
278                 rctl.o          \
279                 rctlsys.o       \
280                 readlink.o      \
281                 refstr.o        \
282                 rename.o        \
283                 resolvepath.o   \
284                 retire_store.o  \
285                 process.o       \
286                 rlimit.o        \
287                 rmap.o          \
288                 rw.o            \
289                 rwstlock.o      \
290                 sad_conf.o      \
291                 sid.o           \
292                 sidsys.o        \
293                 sched.o         \
294                 schedctl.o      \
295                 sctp_crc32.o    \
296                 seg_dev.o       \
297                 seg_kp.o        \
298                 seg_kpm.o       \
299                 seg_map.o       \
300                 seg_vn.o        \
301                 seg_spt.o       \
302                 semaphore.o     \
303                 sendfile.o      \
304                 session.o       \
305                 share.o         \
306                 shuttle.o       \
307                 sig.o           \
308                 sigaction.o     \
309                 sigaltstack.o   \
310                 signotify.o     \
311                 sigpending.o    \
312                 sigprocmask.o   \
313                 sigqueue.o      \
314                 sigsendset.o    \
315                 sigsuspend.o    \
316                 sigtimedwait.o  \
317                 sleepq.o        \
318                 sock_conf.o     \
319                 space.o         \
320                 sscanf.o        \
321                 stat.o          \
322                 statfs.o        \
323                 statvfs.o       \
324                 stol.o          \
325                 str_conf.o      \
```



```
326                 strcalls.o      \
327                 stream.o        \
328                 streamio.o      \
329                 strext.o        \
330                 strsubr.o       \
331                 strsun.o        \
332                 subr.o          \
333                 sunddi.o        \
334                 sunmdi.o        \
335                 sunndi.o        \
336                 sunpci.o        \
337                 sunpm.o         \
338                 sundlpi.o       \
339                 suntpi.o        \
340                 swap_subr.o     \
341                 swap_vnops.o    \
342                 symlink.o       \
343                 sync.o          \
344                 sysclass.o      \
345                 sysconfig.o     \
346                 sysent.o        \
347                 sysfs.o         \
348                 systeminfo.o    \
349                 task.o          \
350                 taskq.o         \
351                 tasksys.o       \
352                 time.o          \
353                 timer.o         \
354                 times.o         \
355                 timers.o        \
356                 thread.o        \
357                 tlabel.o        \
358                 tnf_res.o       \
359                 turnstile.o     \
360                 tty_common.o    \
361                 u8_textprep.o   \
362                 uadmin.o        \
363                 uconv.o         \
364                 ucredsys.o      \
365                 uid.o           \
366                 umask.o         \
367                 umount.o        \
368                 uname.o         \
369                 unix_bb.o       \
370                 unlink.o        \
371                 urw.o           \
372                 utime.o         \
373                 utssys.o        \
374                 uucopy.o        \
375                 vfs.o           \
376                 vfs_conf.o      \
377                 vmem.o          \
378                 vm_anon.o       \
379                 vm_as.o         \
380                 vm_meter.o      \
381                 vm_pageout.o    \
382                 vm_pvn.o        \
383                 vm_rm.o         \
384                 vm_seg.o        \
385                 vm_subr.o       \
386                 vm_swap.o       \
387                 vm_usage.o      \
388                 vnode.o         \
389                 vuid_queue.o    \
390                 vuid_store.o    \
391                 waitq.o         \
```

```
392                 watchpoint.o    \
393                 yield.o         \
394                 scsi_confdata.o \
395                 xattr.o         \
396                 xattr_common.o  \
397                 xdr_mblk.o      \
398                 xdr_mem.o       \
399                 xdr.o           \
400                 xdr_array.o     \
401                 xdr_refer.o     \
402                 xhat.o          \
403                 zone.o

405 #
406 #       Stubs for the stand-alone linker/loader
407 #
408 sparc_GENSTUBS_OBJS =   \
409         kobj_stubs.o

411 i386_GENSTUBS_OBJS =

413 COMMON_GENSTUBS_OBJS =

415 GENSTUBS_OBJS += $(COMMON_GENSTUBS_OBJS) $($(MACH)_GENSTUBS_OBJS)

417 #
418 #       DTrace and DTrace Providers
419 #
420 DTRACE_OBJS += dtrace.o dtrace_isa.o dtrace_asm.o

422 SDT_OBJS += sdt_subr.o

424 PROFILE_OBJS += profile.o

426 SYSTRACE_OBJS += systrace.o

428 LOCKSTAT_OBJS += lockstat.o

430 FASTTRAP_OBJS += fasttrap.o fasttrap_isa.o

432 DCPC_OBJS += dcpc.o

434 #
435 #       Driver (pseudo-driver) Modules
436 #
437 IPP_OBJS +=     ippctl.o

439 AUDIO_OBJS += audio_client.o audio_ddi.o audio_engine.o \
440             audio_fltdata.o audio_format.o audio_ctrl.o \
441             audio_grc3.o audio_output.o audio_input.o \
442             audio_oss.o audio_sun.o

444 AUDIOEMU10K_OBJS += audioemu10k.o

446 AUDIOENS_OBJS += audioens.o

448 AUDIOVIA823X_OBJS += audiovia823x.o

450 AUDIOVIA97_OBJS += audiovia97.o

452 AUDIO1575_OBJS += audio1575.o

454 AUDIO810_OBJS += audio810.o

456 AUDIOCMI_OBJS += audiocmi.o
```



```
458 AUDIOCMIHD_OBJS += audiocmihd.o

460 AUDIOHD_OBJS += audiohd.o

462 AUDIOIXP_OBJS += audioixp.o

464 AUDIOLS_OBJS += audiols.o

466 AUDIOP16X_OBJS += audiop16x.o

468 AUDIOPCI_OBJS += audiopci.o

470 AUDIOSOLO_OBJS += audiosolo.o

472 AUDIOTS_OBJS += audiots.o

474 AC97_OBJS += ac97.o ac97_ad.o ac97_alc.o ac97_cmi.o

476 BLKDEV_OBJS += blkdev.o

478 CARDBUS_OBJS += cardbus.o cardbus_hp.o cardbus_cfg.o

480 CONSKBD_OBJS += conskbd.o

482 CONSMS_OBJS +=  consms.o

484 OLDPTY_OBJS +=  tty_ptyconf.o

486 PTC_OBJS +=     tty_pty.o

488 PTSL_OBJS +=    tty_pts.o

490 PTM_OBJS +=     ptm.o

492 MII_OBJS +=     mii.o mii_cicada.o mii_natsemi.o mii_intel.o mii_qualsemi.o \
493                 mii_marvell.o mii_realtek.o mii_other.o

495 PTS_OBJS +=     pts.o

497 PTY_OBJS +=     ptms_conf.o

499 SAD_OBJS +=     sad.o

501 MD4_OBJS +=     md4.o md4_mod.o

503 MD5_OBJS +=     md5.o md5_mod.o

505 SHA1_OBJS +=    sha1.o sha1_mod.o

507 SHA2_OBJS +=    sha2.o sha2_mod.o

509 IPGPC_OBJS +=   classifierddi.o classifier.o filters.o trie.o table.o \
510                 ba_table.o

512 DSCPMK_OBJS +=  dscpmk.o dscpmkddi.o

514 DLCOSMK_OBJS += dlcosmk.o dlcosmkddi.o

516 FLOWACCT_OBJS +=        flowacctddi.o flowacct.o

518 TOKENMT_OBJS += tokenmt.o tokenmtddi.o

520 TSWTCL_OBJS +=  tswtcl.o tswtclddi.o

522 ARP_OBJS +=     arpddi.o
```

```
524 ICMP_OBJS +=    icmpddi.o

526 ICMP6_OBJS +=   icmp6ddi.o

528 RTS_OBJS +=     rtsddi.o

530 IP_ICMP_OBJS =  icmp.o icmp_opt_data.o
531 IP_RTS_OBJS =   rts.o rts_opt_data.o
532 IP_TCP_OBJS =   tcp.o tcp_fusion.o tcp_opt_data.o tcp_sack.o tcp_stats.o \
533                 tcp_misc.o tcp_timers.o tcp_time_wait.o tcp_tpi.o tcp_output.o \
534                 tcp_input.o tcp_socket.o tcp_bind.o tcp_cluster.o tcp_tunables.o
535 IP_UDP_OBJS =   udp.o udp_opt_data.o udp_tunables.o udp_stats.o
536 IP_SCTP_OBJS =  sctp.o sctp_opt_data.o sctp_output.o \
537                 sctp_init.o sctp_input.o sctp_cookie.o \
538                 sctp_conn.o sctp_error.o sctp_snmp.o \
539                 sctp_tunables.o sctp_shutdown.o sctp_common.o \
540                 sctp_timer.o sctp_heartbeat.o sctp_hash.o \
541                 sctp_bind.o sctp_notify.o sctp_asconf.o \
542                 sctp_addr.o tn_ipopt.o tnet.o ip_netinfo.o \
543                 sctp_misc.o
544 IP_ILB_OBJS =   ilb.o ilb_nat.o ilb_conn.o ilb_alg_hash.o ilb_alg_rr.o

546 IP_OBJS +=      igmp.o ipmp.o ip.o ip6.o ip6_asp.o ip6_if.o ip6_ire.o \
547                 ip6_rts.o ip_if.o ip_ire.o ip_listutils.o ip_mroute.o \
548                 ip_multi.o ip2mac.o ip_ndp.o ip_rts.o ip_srcid.o \
549                 ipddi.o ipdrop.o mi.o nd.o tunables.o optcom.o snmpcom.o \
550                 ipsec_loader.o spd.o ipclassifier.o inet_common.o ip_squeue.o \
551                 squeue.o ip_sadb.o ip_ftable.o proto_set.o radix.o ip_dummy.o \
552                 ip_helper_stream.o ip_tunables.o \
553                 ip_output.o ip_input.o ip6_input.o ip6_output.o ip_arp.o \
554                 conn_opt.o ip_attr.o ip_dce.o \
555                 $(IP_ICMP_OBJS) \
556                 $(IP_RTS_OBJS) \
557                 $(IP_TCP_OBJS) \
558                 $(IP_UDP_OBJS) \
559                 $(IP_SCTP_OBJS) \
560                 $(IP_ILB_OBJS)

562 IP6_OBJS +=     ip6ddi.o

564 HOOK_OBJS +=    hook.o

566 NETI_OBJS +=    neti_impl.o neti_mod.o neti_stack.o

568 KEYSOCK_OBJS += keysockddi.o keysock.o keysock_opt_data.o

570 IPNET_OBJS +=   ipnet.o ipnet_bpf.o

572 SPDSOCK_OBJS += spdsockddi.o spdsock.o spdsock_opt_data.o

574 IPSECESP_OBJS += ipsecespddi.o ipsecesp.o

576 IPSECAH_OBJS += ipsecahddi.o ipsecah.o sadb.o

578 SPPP_OBJS +=    sppp.o sppp_dlpi.o sppp_mod.o s_common.o

580 SPPPTUN_OBJS += sppptun.o sppptun_mod.o

582 SPPPASYN_OBJS += spppasyn.o spppasyn_mod.o

584 SPPPCOMP_OBJS += spppcomp.o spppcomp_mod.o deflate.o bsd-comp.o vjcompress.o \
585                 zlib.o

587 TCP_OBJS +=     tcpddi.o

589 TCP6_OBJS +=    tcp6ddi.o
```



```
591 NCA_OBJS +=     ncaddi.o

593 SDP_SOCK_MOD_OBJS += sockmod_sdp.o socksdp.o socksdpsubr.o

595 SCTP_SOCK_MOD_OBJS += sockmod_sctp.o socksctp.o socksctpsubr.o

597 PFP_SOCK_MOD_OBJS += sockmod_pfp.o

599 RDS_SOCK_MOD_OBJS += sockmod_rds.o

601 RDS_OBJS +=     rdsddi.o rdssubr.o rds_opt.o rds_ioctl.o

603 RDSIB_OBJS +=   rdsib.o rdsib_ib.o rdsib_cm.o rdsib_ep.o rdsib_buf.o \
604                 rdsib_debug.o rdsib_sc.o

606 RDSV3_OBJS +=   af_rds.o rdsv3_ddi.o bind.o loop.o threads.o connection.o \
607                 transport.o cong.o sysctl.o message.o rds_recv.o send.o \
608                 stats.o info.o page.o rdma_transport.o ib_ring.o ib_rdma.o \
609                 ib_recv.o ib.o ib_send.o ib_sysctl.o ib_stats.o ib_cm.o \
610                 rdsv3_sc.o rdsv3_debug.o rdsv3_impl.o rdma.o rdsv3_af_thr.o

612 ISER_OBJS +=    iser.o iser_cm.o iser_cq.o iser_ib.o iser_idm.o \
613                 iser_resource.o iser_xfer.o

615 UDP_OBJS +=     udpddi.o

617 UDP6_OBJS +=    udp6ddi.o

619 SY_OBJS +=      gentty.o

621 TCO_OBJS +=     ticots.o

623 TCOO_OBJS +=    ticotsord.o

625 TCL_OBJS +=     ticlts.o

627 TL_OBJS +=      tl.o

629 DUMP_OBJS +=    dump.o

631 BPF_OBJS +=     bpf.o bpf_filter.o bpf_mod.o bpf_dlt.o bpf_mac.o

633 CLONE_OBJS +=   clone.o

635 CN_OBJS +=      cons.o

637 DLD_OBJS +=     dld_drv.o dld_proto.o dld_str.o dld_flow.o

639 DLS_OBJS +=     dls.o dls_link.o dls_mod.o dls_stat.o dls_mgmt.o

641 GLD_OBJS +=     gld.o gldutil.o

643 MAC_OBJS +=     mac.o  mac_bcast.o mac_client.o mac_datapath_setup.o mac_flow.o
644                 mac_hio.o mac_mod.o mac_ndd.o mac_provider.o mac_sched.o \
645                 mac_protect.o mac_soft_ring.o mac_stat.o mac_util.o

647 MAC_6TO4_OBJS +=        mac_6to4.o

649 MAC_ETHER_OBJS +=       mac_ether.o

651 MAC_IPV4_OBJS +=        mac_ipv4.o

653 MAC_IPV6_OBJS +=        mac_ipv6.o

655 MAC_WIFI_OBJS +=        mac_wifi.o
```

```
MAC_IB_OBJS +=          mac_ib.o

IPTUN_OBJS +=   iptun_dev.o iptun_ctl.o iptun.o

AGGR_OBJS +=    aggr_dev.o aggr_ctl.o aggr_grp.o aggr_port.o \
                aggr_send.o aggr_recv.o aggr_lacp.o

SOFTMAC_OBJS += softmac_main.o softmac_ctl.o softmac_capab.o \
                softmac_dev.o softmac_stat.o softmac_pkt.o softmac_fp.o

NET80211_OBJS += net80211.o net80211_proto.o net80211_input.o \
                 net80211_output.o net80211_node.o net80211_crypto.o \
                 net80211_crypto_none.o net80211_crypto_wep.o net80211_ioctl.o \
                 net80211_crypto_tkip.o net80211_crypto_ccmp.o  \
                 net80211_ht.o

VNIC_OBJS +=    vnic_ctl.o vnic_dev.o

SIMNET_OBJS +=  simnet.o

IB_OBJS +=      ibnex.o ibnex_ioctl.o ibnex_hca.o

IBCM_OBJS +=    ibcm_impl.o ibcm_sm.o ibcm_ti.o ibcm_utils.o ibcm_path.o \
                ibcm_arp.o ibcm_arp_link.o

IBDM_OBJS +=    ibdm.o

IBDMA_OBJS +=   ibdma.o

IBMF_OBJS +=    ibmf.o ibmf_impl.o ibmf_dr.o ibmf_wqe.o ibmf_ud_dest.o ibmf_mod.
                ibmf_send.o ibmf_recv.o ibmf_handlers.o ibmf_trans.o \
                ibmf_timers.o ibmf_msg.o ibmf_utils.o ibmf_rmpp.o \
                ibmf_saa.o ibmf_saa_impl.o ibmf_saa_utils.o ibmf_saa_events.o

IBTL_OBJS +=    ibtl_impl.o ibtl_util.o ibtl_mem.o ibtl_handlers.o ibtl_qp.o \
                ibtl_cq.o ibtl_wr.o ibtl_hca.o ibtl_chan.o ibtl_cm.o \
                ibtl_mcg.o ibtl_ibnex.o ibtl_srq.o ibtl_part.o

TAVOR_OBJS +=   tavor.o tavor_agents.o tavor_cfg.o tavor_ci.o tavor_cmd.o \
                tavor_cq.o tavor_event.o tavor_ioctl.o tavor_misc.o \
                tavor_mr.o tavor_qp.o tavor_qpmod.o tavor_rsrc.o \
                tavor_srq.o tavor_stats.o tavor_umap.o tavor_wr.o

HERMON_OBJS +=  hermon.o hermon_agents.o hermon_cfg.o hermon_ci.o hermon_cmd.o \
                hermon_cq.o hermon_event.o hermon_ioctl.o hermon_misc.o \
                hermon_mr.o hermon_qp.o hermon_qpmod.o hermon_rsrc.o \
                hermon_srq.o hermon_stats.o hermon_umap.o hermon_wr.o \
                hermon_fcoib.o hermon_fm.o

DAPLT_OBJS +=   daplt.o

SOL_OFS_OBJS += sol_cma.o sol_ib_cma.o sol_uobj.o \
                sol_ofs_debug_util.o sol_ofs_gen_util.o \
                sol_kverbs.o

SOL_UCMA_OBJS +=        sol_ucma.o

SOL_UVERBS_OBJS +=      sol_uverbs.o sol_uverbs_comp.o sol_uverbs_event.o \
                        sol_uverbs_hca.o sol_uverbs_qp.o

SOL_UMAD_OBJS += sol_umad.o

KSTAT_OBJS +=   kstat.o

KSYMS_OBJS +=   ksyms.o
```



```
INSTANCE_OBJS += inst_sync.o

IWSCN_OBJS +=   iwscons.o

LOFI_OBJS +=    lofi.o LzmaDec.o

FSSNAP_OBJS +=  fssnap.o

FSSNAPIF_OBJS += fssnap_if.o

MM_OBJS +=      mem.o

PHYSMEM_OBJS += physmem.o

OPTIONS_OBJS += options.o

WINLOCK_OBJS += winlockio.o

PM_OBJS +=      pm.o
SRN_OBJS +=     srn.o

PSEUDO_OBJS +=  pseudonex.o

RAMDISK_OBJS += ramdisk.o

LLC1_OBJS += llc1.o

USBKBM_OBJS += usbkbm.o

USBWCM_OBJS += usbwcm.o

BOFI_OBJS += bofi.o

HID_OBJS += hid.o

HWA_RC_OBJS += hwarc.o

USBSKEL_OBJS += usbskel.o

USBVC_OBJS += usbvc.o usbvc_v4l2.o

HIDPARSER_OBJS += hidparser.o

USB_AC_OBJS += usb_ac.o

USB_AS_OBJS += usb_as.o

USB_AH_OBJS += usb_ah.o

USBMS_OBJS += usbms.o

USBPRN_OBJS += usbprn.o

UGEN_OBJS += ugen.o

USBSER_OBJS += usbser.o usbser_rseq.o

USBSACM_OBJS += usbsacm.o

USBSER_KEYSPAN_OBJS += usbser_keyspan.o keyspan_dsd.o keyspan_pipe.o

USBS49_FW_OBJS += keyspan_49fw.o

USBSPRL_OBJS += usbser_pl2303.o pl2303_dsd.o
```

```
788 WUSB_CA_OBJS += wusb_ca.o

790 USBFTDI_OBJS += usbser_uftdi.o uftdi_dsd.o

792 USBECM_OBJS += usbecm.o

794 WC_OBJS += wscons.o vcons.o

796 VCONS_CONF_OBJS += vcons_conf.o

798 SCSI_OBJS +=    scsi_capabilities.o scsi_confsubr.o scsi_control.o \
799                 scsi_data.o scsi_fm.o scsi_hba.o scsi_reset_notify.o \
800                 scsi_resource.o scsi_subr.o scsi_transport.o scsi_watch.o \
801                 smp_transport.o

803 SCSI_VHCI_OBJS +=               scsi_vhci.o mpapi_impl.o scsi_vhci_tpgs.o

805 SCSI_VHCI_F_SYM_OBJS +=         sym.o

807 SCSI_VHCI_F_TPGS_OBJS +=        tpgs.o

809 SCSI_VHCI_F_ASYM_SUN_OBJS +=    asym_sun.o

811 SCSI_VHCI_F_SYM_HDS_OBJS +=     sym_hds.o

813 SCSI_VHCI_F_TAPE_OBJS +=        tape.o

815 SCSI_VHCI_F_TPGS_TAPE_OBJS +=   tpgs_tape.o

817 SGEN_OBJS +=    sgen.o

819 SMP_OBJS +=     smp.o

821 SATA_OBJS +=    sata.o

823 USBA_OBJS +=    hcdi.o  usba.o  usbai.o hubdi.o parser.o genconsole.o \
824                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
825                 usba_devdb.o usba10_calls.o usba_ugen.o whcdi.o wa.o
826 USBA_WITHOUT_WUSB_OBJS +=       hcdi.o  usba.o  usbai.o hubdi.o parser.o gencons
827                 usbai_pipe_mgmt.o usbai_req.o usbai_util.o usbai_register.o \
828                 usba_devdb.o usba10_calls.o usba_ugen.o

830 USBA10_OBJS +=  usba10.o

832 RSM_OBJS +=     rsm.o   rsmka_pathmanager.o     rsmka_util.o

834 RSMOPS_OBJS +=  rsmops.o

836 S1394_OBJS +=   t1394.o t1394_errmsg.o s1394.o s1394_addr.o s1394_asynch.o \
837                 s1394_bus_reset.o s1394_cmp.o s1394_csr.o s1394_dev_disc.o \
838                 s1394_fa.o s1394_fcp.o \
839                 s1394_hotplug.o s1394_isoch.o s1394_misc.o h1394.o nx1394.o

841 HCI1394_OBJS += hci1394.o hci1394_async.o hci1394_attach.o hci1394_buf.o \
842                 hci1394_csr.o hci1394_detach.o hci1394_extern.o \
843                 hci1394_ioctl.o hci1394_isoch.o hci1394_isr.o \
844                 hci1394_ixl_comp.o hci1394_ixl_isr.o hci1394_ixl_misc.o \
845                 hci1394_ixl_update.o hci1394_misc.o hci1394_ohci.o \
846                 hci1394_q.o hci1394_s1394if.o hci1394_tlabel.o \
847                 hci1394_tlist.o hci1394_vendor.o

849 AV1394_OBJS +=  av1394.o av1394_as.o av1394_async.o av1394_cfgrom.o \
850                 av1394_cmp.o av1394_fcp.o av1394_isoch.o av1394_isoch_chan.o \
851                 av1394_isoch_recv.o av1394_isoch_xmit.o av1394_list.o \
852                 av1394_queue.o
```



```
854 DCAM1394_OBJS += dcam.o dcam_frame.o dcam_param.o dcam_reg.o \
855                 dcam_ring_buff.o

857 SCSA1394_OBJS += hba.o sbp2_driver.o sbp2_bus.o

859 SBP2_OBJS += cfgrom.o sbp2.o

861 PMODEM_OBJS += pmodem.o pmodem_cis.o cis.o cis_callout.o cis_handlers.o cis_para

863 DSW_OBJS +=     dsw.o dsw_dev.o ii_tree.o

865 NCALL_OBJS +=   ncall.o \
866                 ncall_stub.o

868 RDC_OBJS +=     rdc.o \
869                 rdc_dev.o \
870                 rdc_io.o \
871                 rdc_clnt.o \
872                 rdc_prot_xdr.o \
873                 rdc_svc.o \
874                 rdc_bitmap.o \
875                 rdc_health.o \
876                 rdc_subr.o \
877                 rdc_diskq.o

879 RDCSRV_OBJS +=  rdcsrv.o

881 RDCSTUB_OBJS += rdc_stub.o

883 SDBC_OBJS +=    sd_bcache.o \
884                 sd_bio.o \
885                 sd_conf.o \
886                 sd_ft.o \
887                 sd_hash.o \
888                 sd_io.o \
889                 sd_misc.o \
890                 sd_pcu.o \
891                 sd_tdaemon.o \
892                 sd_trace.o \
893                 sd_iob_impl0.o \
894                 sd_iob_impl1.o \
895                 sd_iob_impl2.o \
896                 sd_iob_impl3.o \
897                 sd_iob_impl4.o \
898                 sd_iob_impl5.o \
899                 sd_iob_impl6.o \
900                 sd_iob_impl7.o \
901                 safestore.o \
902                 safestore_ram.o

904 NSCTL_OBJS +=   nsctl.o \
905                 nsc_cache.o \
906                 nsc_disk.o \
907                 nsc_dev.o \
908                 nsc_freeze.o \
909                 nsc_gen.o \
910                 nsc_mem.o \
911                 nsc_ncallio.o \
912                 nsc_power.o \
913                 nsc_resv.o \
914                 nsc_rmspin.o \
915                 nsc_solaris.o \
916                 nsc_trap.o \
917                 nsc_list.o
918 UNISTAT_OBJS += spuni.o \
919                 spcs_s_k.o
```

```
NSKERN_OBJS +=  nsc_ddi.o \
                nsc_proc.o \
                nsc_raw.o \
                nsc_thread.o \
                nskernd.o

SV_OBJS +=      sv.o

PMCS_OBJS += pmcs_attach.o pmcs_ds.o pmcs_intr.o pmcs_nvram.o pmcs_sata.o \
                pmcs_scsa.o pmcs_smhba.o pmcs_subr.o pmcs_fwlog.o

PMCS8001FW_C_OBJS +=    pmcs_fw_hdr.o
PMCS8001FW_OBJS +=              $(PMCS8001FW_C_OBJS) SPCBoot.o ila.o firmware.o

#
#       Build up defines and paths.

ST_OBJS +=      st.o    st_conf.o

EMLXS_OBJS +=   emlxs_clock.o emlxs_dfc.o emlxs_dhchap.o emlxs_diag.o \
                emlxs_download.o emlxs_dump.o emlxs_els.o emlxs_event.o \
                emlxs_fcf.o emlxs_fcp.o emlxs_fct.o emlxs_hba.o emlxs_ip.o \
                emlxs_mbox.o emlxs_mem.o emlxs_msg.o emlxs_node.o \
                emlxs_pkt.o emlxs_sli3.o emlxs_sli4.o emlxs_solaris.o \
                emlxs_thread.o

EMLXS_FW_OBJS +=        emlxs_fw.o

OCE_OBJS +=     oce_buf.o oce_fm.o oce_gld.o oce_hw.o oce_intr.o oce_main.o \
                oce_mbx.o oce_mq.o oce_queue.o oce_rx.o oce_stat.o oce_tx.o \
                oce_utils.o

FCT_OBJS += discovery.o fct.o

QLT_OBJS += 2400.o 2500.o 8100.o qlt.o qlt_dma.o

SRPT_OBJS += srpt_mod.o srpt_ch.o srpt_cm.o srpt_ioc.o srpt_stp.o

FCOE_OBJS += fcoe.o fcoe_eth.o fcoe_fc.o

FCOET_OBJS += fcoet.o fcoet_eth.o fcoet_fc.o

FCOEI_OBJS += fcoei.o fcoei_eth.o fcoei_lv.o

ISCSIT_SHARED_OBJS += \
                iscsit_common.o

ISCSIT_OBJS +=  $(ISCSIT_SHARED_OBJS) \
                iscsit.o iscsit_tgt.o iscsit_sess.o iscsit_login.o \
                iscsit_text.o iscsit_isns.o iscsit_radiusauth.o \
                iscsit_radiuspacket.o iscsit_auth.o iscsit_authclient.o

PPPT_OBJS +=    alua_ic_if.o pppt.o pppt_msg.o pppt_tgt.o

STMF_OBJS += lun_map.o stmf.o

STMF_SBD_OBJS += sbd.o sbd_scsi.o sbd_pgr.o sbd_zvol.o

SYSMSG_OBJS +=  sysmsg.o

SES_OBJS +=     ses.o ses_sen.o ses_safte.o ses_ses.o

TNF_OBJS +=     tnf_buf.o       tnf_trace.o     tnf_writer.o    trace_init.o \
                trace_funcs.o   tnf_probe.o     tnf.o
```



```
LOGINDMUX_OBJS += logindmux.o

DEVINFO_OBJS += devinfo.o

DEVPOLL_OBJS += devpoll.o

DEVPOOL_OBJS += devpool.o

I8042_OBJS +=   i8042.o

KB8042_OBJS +=  \
                at_keyprocess.o \
                kb8042.o        \
                kb8042_keytables.o

MOUSE8042_OBJS += mouse8042.o

FDC_OBJS +=     fdc.o

ASY_OBJS +=     asy.o

ECPP_OBJS +=    ecpp.o

VUIDM3P_OBJS += vuidmice.o vuidm3p.o

VUIDM4P_OBJS += vuidmice.o vuidm4p.o

VUIDM5P_OBJS += vuidmice.o vuidm5p.o

VUIDPS2_OBJS += vuidmice.o vuidps2.o

HPCSVC_OBJS += hpcsvc.o

PCIE_MISC_OBJS += pcie.o pcie_fault.o pcie_hp.o pciehpc.o pcishpc.o pcie_pwr.o p

PCIHPNEXUS_OBJS += pcihp.o

OPENEEPR_OBJS += openprom.o

RANDOM_OBJS += random.o

PSHOT_OBJS += pshot.o

GEN_DRV_OBJS += gen_drv.o

TCLIENT_OBJS += tclient.o

TPHCI_OBJS += tphci.o

TVHCI_OBJS += tvhci.o

EMUL64_OBJS += emul64.o emul64_bsd.o

FCP_OBJS += fcp.o

FCIP_OBJS += fcip.o

FCSM_OBJS += fcsm.o

FCTL_OBJS += fctl.o

FP_OBJS += fp.o

QLC_OBJS += ql_api.o ql_debug.o ql_hba_fru.o ql_init.o ql_iocb.o ql_ioctl.o \
        ql_isr.o ql_mbx.o ql_nx.o ql_xioctl.o ql_fw_table.o
```

```
1052 QLC_FW_2200_OBJS += ql_fw_2200.o

1054 QLC_FW_2300_OBJS += ql_fw_2300.o

1056 QLC_FW_2400_OBJS += ql_fw_2400.o

1058 QLC_FW_2500_OBJS += ql_fw_2500.o

1060 QLC_FW_6322_OBJS += ql_fw_6322.o

1062 QLC_FW_8100_OBJS += ql_fw_8100.o

1064 QLGE_OBJS += qlge.o qlge_dbg.o qlge_flash.o qlge_fm.o qlge_gld.o qlge_mpi.o

1066 ZCONS_OBJS += zcons.o

1068 NV_SATA_OBJS += nv_sata.o

1070 SI3124_OBJS += si3124.o

1072 AHCI_OBJS += ahci.o

1074 PCIIDE_OBJS += pci-ide.o

1076 PCEPP_OBJS += pcepp.o

1078 CPC_OBJS += cpc.o

1080 CPUID_OBJS += cpuid_drv.o

1082 SYSEVENT_OBJS += sysevent.o

1084 BL_OBJS += bl.o

1086 DRM_OBJS += drm_sunmod.o drm_kstat.o drm_agpsupport.o \
1087             drm_auth.o drm_bufs.o drm_context.o drm_dma.o \
1088             drm_drawable.o drm_drv.o drm_fops.o drm_ioctl.o drm_irq.o \
1089             drm_lock.o drm_memory.o drm_msg.o drm_pci.o drm_scatter.o \
1090             drm_cache.o drm_gem.o drm_mm.o ati_pcigart.o

1092 FM_OBJS += devfm.o devfm_machdep.o

1094 RTLS_OBJS +=    rtls.o

1096 #
1097 #                       exec modules
1098 #
1099 AOUTEXEC_OBJS +=aout.o

1101 ELFEXEC_OBJS += elf.o elf_notes.o old_notes.o

1103 INTPEXEC_OBJS +=intp.o

1105 SHBINEXEC_OBJS +=shbin.o

1107 JAVAEXEC_OBJS +=java.o

1109 #
1110 #                       file system modules
1111 #
1112 AUTOFS_OBJS +=  auto_vfsops.o auto_vnops.o auto_subr.o auto_xdr.o auto_sys.o

1114 CACHEFS_OBJS += cachefs_cnode.o         cachefs_cod.o \
1115                 cachefs_dir.o           cachefs_dlog.o  cachefs_filegrp.o \
1116                 cachefs_fscache.o       cachefs_ioctl.o cachefs_log.o \
1117                 cachefs_module.o \
```



```
1118                 cachefs_noopc.o         cachefs_resource.o \
1119                 cachefs_strict.o \
1120                 cachefs_subr.o          cachefs_vfsops.o \
1121                 cachefs_vnops.o

1123 DCFS_OBJS +=    dc_vnops.o

1125 DEVFS_OBJS +=   devfs_subr.o    devfs_vfsops.o  devfs_vnops.o

1127 DEV_OBJS  +=    sdev_subr.o     sdev_vfsops.o   sdev_vnops.o    \
1128                 sdev_ptsops.o   sdev_zvolops.o  sdev_comm.o     \
1129                 sdev_profile.o  sdev_ncache.o   sdev_netops.o   \
1130                 sdev_ipnetops.o \
1131                 sdev_vtops.o

1133 CTFS_OBJS +=    ctfs_all.o ctfs_cdir.o ctfs_ctl.o ctfs_event.o \
1134                 ctfs_latest.o ctfs_root.o ctfs_sym.o ctfs_tdir.o ctfs_tmpl.o

1136 OBJFS_OBJS +=   objfs_vfs.o     objfs_root.o    objfs_common.o \
1137                 objfs_odir.o    objfs_data.o

1139 FDFS_OBJS +=    fdops.o

1141 FIFO_OBJS +=    fifosubr.o      fifovnops.o

1143 PIPE_OBJS +=    pipe.o

1145 HSFS_OBJS +=    hsfs_node.o     hsfs_subr.o     hsfs_vfsops.o   hsfs_vnops.o \
1146                 hsfs_susp.o     hsfs_rrip.o     hsfs_susp_subr.o

1148 LOFS_OBJS +=    lofs_subr.o     lofs_vfsops.o   lofs_vnops.o

1150 NAMEFS_OBJS +=  namevfs.o       namevno.o

1152 NFS_OBJS +=     nfs_client.o    nfs_common.o    nfs_dump.o \
1153                 nfs_subr.o      nfs_vfsops.o    nfs_vnops.o \
1154                 nfs_xdr.o       nfs_sys.o       nfs_strerror.o \
1155                 nfs3_vfsops.o   nfs3_vnops.o    nfs3_xdr.o \
1156                 nfs_acl_vnops.o nfs_acl_xdr.o   nfs4_vfsops.o \
1157                 nfs4_vnops.o    nfs4_xdr.o      nfs4_idmap.o \
1158                 nfs4_shadow.o   nfs4_subr.o \
1159                 nfs4_attr.o     nfs4_rnode.o    nfs4_client.o \
1160                 nfs4_acache.o   nfs4_common.o   nfs4_client_state.o \
1161                 nfs4_callback.o nfs4_recovery.o nfs4_client_secinfo.o \
1162                 nfs4_client_debug.o     nfs_stats.o \
1163                 nfs4_acl.o      nfs4_stub_vnops.o       nfs_cmd.o

1165 NFSSRV_OBJS +=  nfs_server.o    nfs_srv.o       nfs3_srv.o \
1166                 nfs_acl_srv.o   nfs_auth.o      nfs_auth_xdr.o \
1167                 nfs_export.o    nfs_log.o       nfs_log_xdr.o \
1168                 nfs4_srv.o      nfs4_state.o    nfs4_srv_attr.o \
1169                 nfs4_srv_ns.o   nfs4_db.o       nfs4_srv_deleg.o \
1170                 nfs4_deleg_ops.o nfs4_srv_readdir.o nfs4_dispatch.o

1172 SMBSRV_SHARED_OBJS += \
1173                 smb_inet.o \
1174                 smb_match.o \
1175                 smb_msgbuf.o \
1176                 smb_oem.o \
1177                 smb_string.o \
1178                 smb_utf8.o \
1179                 smb_door_legacy.o \
1180                 smb_xdr.o \
1181                 smb_token.o \
1182                 smb_token_xdr.o \
1183                 smb_sid.o \
```

```
1184                smb_native.o \
1185                smb_netbios_util.o

1187 SMBSRV_OBJS +=  $(SMBSRV_SHARED_OBJS)                  \
1188                smb_acl.o                               \
1189                smb_alloc.o                             \
1190                smb_close.o                             \
1191                smb_common_open.o                       \
1192                smb_common_transact.o                   \
1193                smb_create.o                            \
1194                smb_delete.o                            \
1195                smb_directory.o                         \
1196                smb_dispatch.o                          \
1197                smb_echo.o                              \
1198                smb_fem.o                               \
1199                smb_find.o                              \
1200                smb_flush.o                             \
1201                smb_fsinfo.o                            \
1202                smb_fsops.o                             \
1203                smb_init.o                              \
1204                smb_kdoor.o                             \
1205                smb_kshare.o                            \
1206                smb_kutil.o                             \
1207                smb_lock.o                              \
1208                smb_lock_byte_range.o                   \
1209                smb_locking_andx.o                      \
1210                smb_logoff_andx.o                       \
1211                smb_mangle_name.o                       \
1212                smb_mbuf_marshaling.o                   \
1213                smb_mbuf_util.o                         \
1214                smb_negotiate.o                         \
1215                smb_net.o                               \
1216                smb_node.o                              \
1217                smb_nt_cancel.o                         \
1218                smb_nt_create_andx.o                    \
1219                smb_nt_transact_create.o                \
1220                smb_nt_transact_ioctl.o                 \
1221                smb_nt_transact_notify_change.o         \
1222                smb_nt_transact_quota.o                 \
1223                smb_nt_transact_security.o              \
1224                smb_odir.o                              \
1225                smb_ofile.o                             \
1226                smb_open_andx.o                         \
1227                smb_opipe.o                             \
1228                smb_oplock.o                            \
1229                smb_pathname.o                          \
1230                smb_print.o                             \
1231                smb_process_exit.o                      \
1232                smb_query_fileinfo.o                    \
1233                smb_read.o                              \
1234                smb_rename.o                            \
1235                smb_sd.o                                \
1236                smb_seek.o                              \
1237                smb_server.o                            \
1238                smb_session.o                           \
1239                smb_session_setup_andx.o                \
1240                smb_set_fileinfo.o                      \
1241                smb_signing.o                           \
1242                smb_tree.o                              \
1243                smb_trans2_create_directory.o           \
1244                smb_trans2_dfs.o                        \
1245                smb_trans2_find.o                       \
1246                smb_tree_connect.o                      \
1247                smb_unlock_byte_range.o                 \
1248                smb_user.o                              \
1249                smb_vfs.o                               \
```



```
1250                smb_vops.o                              \
1251                smb_vss.o                               \
1252                smb_write.o                             \
1253                smb_write_raw.o

1255 PCFS_OBJS +=    pc_alloc.o      pc_dir.o        pc_node.o       pc_subr.o \
1256                 pc_vfsops.o     pc_vnops.o

1258 PROC_OBJS +=    prcontrol.o     prioctl.o       prsubr.o        prusrio.o \
1259                 prvfsops.o      prvnops.o

1261 MNTFS_OBJS +=   mntvfsops.o     mntvnops.o

1263 SHAREFS_OBJS += sharetab.o      sharefs_vfsops.o        sharefs_vnops.o

1265 SPEC_OBJS +=    specsubr.o      specvfsops.o    specvnops.o

1267 SOCK_OBJS +=    socksubr.o      sockvfsops.o    sockparams.o    \
1268                 socksyscalls.o  socktpi.o       sockstr.o \
1269                 sockcommon_vnops.o      sockcommon_subr.o \
1270                 sockcommon_sops.o       sockcommon.o    \
1271                 sock_notsupp.o  socknotify.o \
1272                 nl7c.o          nl7curi.o       nl7chttp.o      nl7clogd.o \
1273                 nl7cnca.o       sodirect.o      sockfilter.o

1275 TMPFS_OBJS +=   tmp_dir.o       tmp_subr.o      tmp_tnode.o     tmp_vfsops.o \
1276                 tmp_vnops.o

1278 UDFS_OBJS +=    udf_alloc.o     udf_bmap.o      udf_dir.o       \
1279                 udf_inode.o     udf_subr.o      udf_vfsops.o    \
1280                 udf_vnops.o

1282 UFS_OBJS +=     ufs_alloc.o     ufs_bmap.o      ufs_dir.o       ufs_xattr.o \
1283                 ufs_inode.o     ufs_subr.o      ufs_tables.o    ufs_vfsops.o \
1284                 ufs_vnops.o     quota.o         quotacalls.o    quota_ufs.o \
1285                 ufs_filio.o     ufs_lockfs.o    ufs_thread.o    ufs_trans.o \
1286                 ufs_acl.o       ufs_panic.o     ufs_directio.o  ufs_log.o \
1287                 ufs_extvnops.o  ufs_snap.o      lufs.o          lufs_thread.o \
1288                 lufs_log.o      lufs_map.o      lufs_top.o      lufs_debug.o
1289 VSCAN_OBJS +=   vscan_drv.o     vscan_svc.o vscan_door.o

1291 NSMB_OBJS +=    smb_conn.o      smb_dev.o       smb_iod.o       smb_pass.o \
1292                 smb_rq.o        smb_sign.o      smb_smb.o       smb_subrs.o \
1293                 smb_time.o      smb_tran.o      smb_trantcp.o   smb_usr.o \
1294                 subr_mchain.o

1296 SMBFS_COMMON_OBJS += smbfs_ntacl.o
1297 SMBFS_OBJS +=    smbfs_vfsops.o  smbfs_vnops.o   smbfs_node.o    \
1298                 smbfs_acl.o     smbfs_client.o  smbfs_smb.o     \
1299                 smbfs_subr.o    smbfs_subr2.o   \
1300                 smbfs_rwlock.o  smbfs_xattr.o   \
1301                 $(SMBFS_COMMON_OBJS)


1304 #
1305 #                       LVM modules
1306 #
1307 MD_OBJS += md.o md_error.o md_ioctl.o md_mddb.o md_names.o \
1308         md_med.o md_rename.o md_subr.o

1310 MD_COMMON_OBJS = md_convert.o md_crc.o md_revchk.o

1312 MD_DERIVED_OBJS = metamed_xdr.o meta_basic_xdr.o

1314 SOFTPART_OBJS += sp.o sp_ioctl.o
```

```
1316 STRIPE_OBJS += stripe.o stripe_ioctl.o

1318 HOTSPARES_OBJS += hotspares.o

1320 RAID_OBJS += raid.o raid_ioctl.o raid_replay.o raid_resync.o raid_hotspare.o

1322 MIRROR_OBJS += mirror.o mirror_ioctl.o mirror_resync.o

1324 NOTIFY_OBJS += md_notify.o

1326 TRANS_OBJS += mdtrans.o trans_ioctl.o trans_log.o

1328 ZFS_COMMON_OBJS +=              \
1329         arc.o                   \
1330         bplist.o                \
1331         bpobj.o                 \
1332         bptree.o                \
1333         dbuf.o                  \
1334         ddt.o                   \
1335         ddt_zap.o               \
1336         dmu.o                   \
1337         dmu_diff.o              \
1338         dmu_send.o              \
1339         dmu_object.o            \
1340         dmu_objset.o            \
1341         dmu_traverse.o          \
1342         dmu_tx.o                \
1343         dnode.o                 \
1344         dnode_sync.o            \
1345         dsl_dir.o               \
1346         dsl_dataset.o           \
1347         dsl_deadlist.o          \
1348         dsl_pool.o              \
1349         dsl_synctask.o          \
1350         dmu_zfetch.o            \
1351         dsl_deleg.o             \
1352         dsl_prop.o              \
1353         dsl_scan.o              \
1354         zfeature.o              \
1355         gzip.o                  \
1356         lzjb.o                  \
1357         metaslab.o              \
1358         refcount.o              \
1359         sa.o                    \
1360         sha256.o                \
1361         spa.o                   \
1362         spa_config.o            \
1363         spa_errlog.o            \
1364         spa_history.o           \
1365         spa_misc.o              \
1366         space_map.o             \
1367         txg.o                   \
1368         uberblock.o             \
1369         unique.o                \
1370         vdev.o                  \
1371         vdev_cache.o            \
1372         vdev_file.o             \
1373         vdev_label.o            \
1374         vdev_mirror.o           \
1375         vdev_missing.o          \
1376         vdev_queue.o            \
1377         vdev_raidz.o            \
1378         vdev_root.o             \
1379         zap.o                   \
1380         zap_leaf.o              \
1381         zap_micro.o             \
```



```
1382         zfs_byteswap.o          \
1383         zfs_debug.o             \
1384         zfs_fm.o                \
1385         zfs_fuid.o              \
1386         zfs_sa.o                \
1387         zfs_znode.o             \
1388         zil.o                   \
1389         zio.o                   \
1390         zio_checksum.o          \
1391         zio_compress.o          \
1392         zio_inject.o            \
1393         zle.o                   \
1394         zrlock.o

1396 ZFS_SHARED_OBJS +=              \
1397         zfeature_common.o       \
1398         zfs_comutil.o           \
1399         zfs_deleg.o             \
1400         zfs_fletcher.o          \
1401         zfs_namecheck.o         \
1402         zfs_prop.o              \
1403         zpool_prop.o            \
1404         zprop_common.o

1406 ZFS_OBJS +=                     \
1407         $(ZFS_COMMON_OBJS)      \
1408         $(ZFS_SHARED_OBJS)      \
1409         vdev_disk.o             \
1410         zfs_acl.o               \
1411         zfs_ctldir.o            \
1412         zfs_dir.o               \
1413         zfs_ioctl.o             \
1414         zfs_log.o               \
1415         zfs_onexit.o            \
1416         zfs_replay.o            \
1417         zfs_rlock.o             \
1418         rrwlock.o               \
1419         zfs_vfsops.o            \
1420         zfs_vnops.o             \
1421         zvol.o

1423 ZUT_OBJS +=                     \
1424         zut.o

1426 #
1427 #                       streams modules
1428 #
1429 BUFMOD_OBJS     +=      bufmod.o

1431 CONNLD_OBJS +=  connld.o

1433 DEDUMP_OBJS +=  dedump.o

1435 DRCOMPAT_OBJS +=        drcompat.o

1437 LDLINUX_OBJS += ldlinux.o

1439 LDTERM_OBJS +=  ldterm.o uwidth.o

1441 PCKT_OBJS +=    pckt.o

1443 PFMOD_OBJS +=   pfmod.o

1445 PTEM_OBJS +=    ptem.o

1447 REDIRMOD_OBJS += strredirm.o
```

```
1449 TIMOD_OBJS +=   timod.o

1451 TIRDWR_OBJS +=  tirdwr.o

1453 TTCOMPAT_OBJS +=ttcompat.o

1455 LOG_OBJS +=     log.o

1457 PIPEMOD_OBJS += pipemod.o

1459 RPCMOD_OBJS +=  rpcmod.o        clnt_cots.o     clnt_clts.o \
1460                 clnt_gen.o      clnt_perr.o     mt_rpcinit.o    rpc_calmsg.o \
1461                 rpc_prot.o      rpc_sztypes.o   rpc_subr.o      rpcb_prot.o \
1462                 svc.o           svc_clts.o      svc_gen.o       svc_cots.o \
1463                 rpcsys.o        xdr_sizeof.o    clnt_rdma.o     svc_rdma.o \
1464                 xdr_rdma.o      rdma_subr.o     xdrrdma_sizeof.o

1466 TLIMOD_OBJS +=  tlimod.o        t_kalloc.o      t_kbind.o       t_kclose.o \
1467                 t_kconnect.o    t_kfree.o       t_kgtstate.o    t_kopen.o \
1468                 t_krcvudat.o    t_ksndudat.o    t_kspoll.o      t_kunbind.o \
1469                 t_kutil.o

1471 RLMOD_OBJS += rlmod.o

1473 TELMOD_OBJS += telmod.o

1475 CRYPTMOD_OBJS += cryptmod.o

1477 KB_OBJS +=      kbd.o           keytables.o

1479 #
1480 #                       ID mapping module
1481 #
1482 IDMAP_OBJS +=   idmap_mod.o     idmap_kapi.o    idmap_xdr.o     idmap_cache.o

1484 #
1485 #                       scheduling class modules
1486 #
1487 SDC_OBJS +=             sysdc.o

1489 RT_OBJS +=              rt.o
1490 RT_DPTBL_OBJS +=        rt_dptbl.o

1492 TS_OBJS +=              ts.o
1493 TS_DPTBL_OBJS +=        ts_dptbl.o

1495 IA_OBJS +=              ia.o

1497 FSS_OBJS +=             fss.o

1499 FX_OBJS +=              fx.o
1500 FX_DPTBL_OBJS +=        fx_dptbl.o

1502 #
1503 #                       Inter-Process Communication (IPC) modules
1504 #
1505 IPC_OBJS +=     ipc.o

1507 IPCMSG_OBJS +=  msg.o

1509 IPCSEM_OBJS +=  sem.o

1511 IPCSHM_OBJS +=  shm.o

1513 #
```



```
1514 #                       bignum module
1515 #
1516 COMMON_BIGNUM_OBJS += bignum_mod.o bignumimpl.o

1518 BIGNUM_OBJS += $(COMMON_BIGNUM_OBJS) $(BIGNUM_PSR_OBJS)

1520 #
1521 #                       kernel cryptographic framework
1522 #
1523 KCF_OBJS +=     kcf.o kcf_callprov.o kcf_cbufcall.o kcf_cipher.o kcf_crypto.o \
1524                 kcf_cryptoadm.o kcf_ctxops.o kcf_digest.o kcf_dual.o \
1525                 kcf_keys.o kcf_mac.o kcf_mech_tabs.o kcf_miscapi.o \
1526                 kcf_object.o kcf_policy.o kcf_prov_lib.o kcf_prov_tabs.o \
1527                 kcf_sched.o kcf_session.o kcf_sign.o kcf_spi.o kcf_verify.o \
1528                 kcf_random.o modes.o ecb.o cbc.o ctr.o ccm.o gcm.o \
1529                 fips_random.o

1531 CRYPTOADM_OBJS += cryptoadm.o

1533 CRYPTO_OBJS +=  crypto.o

1535 DPROV_OBJS +=   dprov.o

1537 DCA_OBJS +=     dca.o dca_3des.o dca_debug.o dca_dsa.o dca_kstat.o dca_rng.o \
1538                 dca_rsa.o

1540 AESPROV_OBJS += aes.o aes_impl.o aes_modes.o

1542 ARCFOURPROV_OBJS += arcfour.o arcfour_crypt.o

1544 BLOWFISHPROV_OBJS += blowfish.o blowfish_impl.o

1546 ECCPROV_OBJS += ecc.o ec.o ec2_163.o ec2_mont.o ecdecode.o ecl_mult.o \
1547                 ecp_384.o ecp_jac.o ec2_193.o ecl.o ecp_192.o ecp_521.o \
1548                 ecp_jm.o ec2_233.o ecl_curve.o ecp_224.o ecp_aff.o \
1549                 ecp_mont.o ec2_aff.o ec_naf.o ecl_gf.o ecp_256.o mp_gf2m.o \
1550                 mpi.o mplogic.o mpmontg.o mpprime.o oid.o \
1551                 secitem.o ec2_test.o ecp_test.o

1553 RSAPROV_OBJS += rsa.o rsa_impl.o pkcs1.o

1555 SWRANDPROV_OBJS += swrand.o

1557 #
1558 #                       kernel SSL
1559 #
1560 KSSL_OBJS +=    kssl.o kssl_ioctl.o

1562 KSSL_SOCKFIL_MOD_OBJS += ksslfilter.o ksslapi.o ksslrec.o

1564 #
1565 #                       misc. modules
1566 #

1568 C2AUDIT_OBJS += adr.o audit.o audit_event.o audit_io.o \
1569                 audit_path.o audit_start.o audit_syscalls.o audit_token.o \
1570                 audit_mem.o

1572 PCIC_OBJS +=    pcic.o

1574 RPCSEC_OBJS +=  secmod.o        sec_clnt.o      sec_svc.o       sec_gen.o \
1575                 auth_des.o      auth_kern.o     auth_none.o     auth_loopb.o\
1576                 authdesprt.o    authdesubr.o    authu_prot.o \
1577                 key_call.o      key_prot.o      svc_authu.o     svcauthdes.o

1579 RPCSEC_GSS_OBJS +=      rpcsec_gssmod.o rpcsec_gss.o rpcsec_gss_misc.o \
```

```
1580                 rpcsec_gss_utils.o svc_rpcsec_gss.o

1582 CONSCONFIG_OBJS += consconfig.o

1584 CONSCONFIG_DACF_OBJS  += consconfig_dacf.o consplat.o

1586 TEM_OBJS += tem.o tem_safe.o 6x10.o 7x14.o 12x22.o

1588 KBTRANS_OBJS +=                          \
1589                 kbtrans.o                \
1590                 kbtrans_keytables.o      \
1591                 kbtrans_polled.o         \
1592                 kbtrans_streams.o        \
1593                 usb_keytables.o

1595 KGSSD_OBJS +=   gssd_clnt_stubs.o gssd_handle.o gssd_prot.o \
1596                 gss_display_name.o gss_release_name.o gss_import_name.o \
1597                 gss_release_buffer.o gss_release_oid_set.o gen_oids.o gssdmod.o

1599 KGSSD_DERIVED_OBJS = gssd_xdr.o

1601 KGSS_DUMMY_OBJS += dmech.o

1603 KSOCKET_OBJS += ksocket.o ksocket_mod.o

1605 CRYPTO= cksumtypes.o decrypt.o encrypt.o encrypt_length.o etypes.o \
1606         nfold.o verify_checksum.o prng.o block_size.o make_checksum.o\
1607         checksum_length.o hmac.o default_state.o mandatory_sumtype.o

1609 # crypto/des
1610 CRYPTO_DES= f_cbc.o f_cksum.o f_parity.o weak_key.o d3_cbc.o ef_crypto.o

1612 CRYPTO_DK= checksum.o derive.o dk_decrypt.o dk_encrypt.o

1614 CRYPTO_ARCFOUR= k5_arcfour.o

1616 # crypto/enc_provider
1617 CRYPTO_ENC= des.o des3.o arcfour_provider.o aes_provider.o

1619 # crypto/hash_provider
1620 CRYPTO_HASH= hash_kef_generic.o hash_kmd5.o hash_crc32.o hash_ksha1.o

1622 # crypto/keyhash_provider
1623 CRYPTO_KEYHASH= descbc.o k5_kmd5des.o k_hmac_md5.o

1625 # crypto/crc32
1626 CRYPTO_CRC32= crc32.o

1628 # crypto/old
1629 CRYPTO_OLD= old_decrypt.o old_encrypt.o

1631 # crypto/raw
1632 CRYPTO_RAW= raw_decrypt.o raw_encrypt.o

1634 K5_KRB= kfree.o copy_key.o \
1635         parse.o init_ctx.o \
1636         ser_adata.o ser_addr.o \
1637         ser_auth.o ser_cksum.o \
1638         ser_key.o ser_princ.o \
1639         serialize.o unparse.o \
1640         ser_actx.o

1642 K5_OS=  timeofday.o toffset.o \
1643         init_os_ctx.o c_ustime.o

1645 SEAL=
```



```
1646 # EXPORT DELETE START
1647 SEAL=   seal.o unseal.o
1648 # EXPORT DELETE END

1650 MECH=   delete_sec_context.o \
1651         import_sec_context.o \
1652         gssapi_krb5.o \
1653         k5seal.o k5unseal.o k5sealv3.o \
1654         ser_sctx.o \
1655         sign.o \
1656         util_crypt.o  \
1657         util_validate.o  util_ordering.o  \
1658         util_seqnum.o util_set.o util_seed.o \
1659         wrap_size_limit.o verify.o



1663 MECH_GEN= util_token.o


1666 KGSS_KRB5_OBJS += krb5mech.o \
1667         $(MECH) $(SEAL) $(MECH_GEN) \
1668         $(CRYPTO) $(CRYPTO_DES) $(CRYPTO_DK) $(CRYPTO_ARCFOUR) \
1669         $(CRYPTO_ENC) $(CRYPTO_HASH) \
1670         $(CRYPTO_KEYHASH) $(CRYPTO_CRC32) \
1671         $(CRYPTO_OLD) \
1672         $(CRYPTO_RAW) $(K5_KRB) $(K5_OS)

1674 DES_OBJS +=     des_crypt.o des_impl.o des_ks.o des_soft.o

1676 DLBOOT_OBJS +=  bootparam_xdr.o nfs_dlinet.o scan.o

1678 KRTLD_OBJS +=   kobj_bootflags.o getoptstr.o \
1679                 kobj.o kobj_kdi.o kobj_lm.o kobj_subr.o

1681 MOD_OBJS +=     modctl.o modsubr.o modsysfile.o modconf.o modhash.o

1683 STRPLUMB_OBJS += strplumb.o

1685 CPR_OBJS +=     cpr_driver.o cpr_dump.o \
1686                 cpr_main.o cpr_misc.o cpr_mod.o cpr_stat.o \
1687                 cpr_uthread.o

1689 PROF_OBJS +=    prf.o

1691 SE_OBJS += se_driver.o

1693 SYSACCT_OBJS += acct.o

1695 ACCTCTL_OBJS += acctctl.o

1697 EXACCTSYS_OBJS += exacctsys.o

1699 KAIO_OBJS += aio.o

1701 PCMCIA_OBJS += pcmcia.o cs.o cis.o cis_callout.o cis_handlers.o cis_params.o

1703 BUSRA_OBJS += busra.o

1705 PCS_OBJS += pcs.o

1707 PCAN_OBJS += pcan.o

1709 PCATA_OBJS += pcide.o pcdisk.o pclabel.o pcata.o

1711 PCSER_OBJS += pcser.o pcser_cis.o
```

```
1713 PCWL_OBJS += pcwl.o

1715 PSET_OBJS +=    pset.o

1717 OHCI_OBJS += ohci.o ohci_hub.o ohci_polled.o

1719 UHCI_OBJS += uhci.o uhciutil.o uhcitgt.o uhcihub.o uhcipolled.o

1721 EHCI_OBJS += ehci.o ehci_hub.o ehci_xfer.o ehci_intr.o ehci_util.o ehci_polled.o

1723 HUBD_OBJS += hubd.o

1725 USB_MID_OBJS += usb_mid.o

1727 USB_IA_OBJS += usb_ia.o

1729 UWBA_OBJS += uwba.o uwbai.o

1731 SCSA2USB_OBJS += scsa2usb.o usb_ms_bulkonly.o usb_ms_cbi.o

1733 HWAHC_OBJS += hwahc.o hwahc_util.o

1735 WUSB_DF_OBJS += wusb_df.o
1736 WUSB_FWMOD_OBJS += wusb_fwmod.o

1738 IPF_OBJS += ip_fil_solaris.o fil.o solaris.o ip_state.o ip_frag.o ip_nat.o \
1739             ip_proxy.o ip_auth.o ip_pool.o ip_htable.o ip_lookup.o \
1740             ip_log.o misc.o ip_compat.o ip_nat6.o drand48.o

1742 IBD_OBJS +=     ibd.o ibd_cm.o

1744 EIBNX_OBJS +=   enx_main.o enx_hdlrs.o enx_ibt.o enx_log.o enx_fip.o \
1745                 enx_misc.o enx_q.o enx_ctl.o

1747 EOIB_OBJS +=    eib_adm.o eib_chan.o eib_cmn.o eib_ctl.o eib_data.o \
1748                 eib_fip.o eib_ibt.o eib_log.o eib_mac.o eib_main.o \
1749                 eib_rsrc.o eib_svc.o eib_vnic.o

1751 DLPISTUB_OBJS += dlpistub.o

1753 SDP_OBJS +=     sdpddi.o

1755 TRILL_OBJS +=   trill.o

1757 CTF_OBJS += ctf_create.o ctf_decl.o ctf_error.o ctf_hash.o ctf_labels.o \
1758         ctf_lookup.o ctf_open.o ctf_types.o ctf_util.o ctf_subr.o ctf_mod.o

1760 SMBIOS_OBJS += smb_error.o smb_info.o smb_open.o smb_subr.o smb_dev.o

1762 RPCIB_OBJS += rpcib.o

1764 KMDB_OBJS += kdrv.o

1766 AFE_OBJS += afe.o

1768 BGE_OBJS += bge_main2.o bge_chip2.o bge_kstats.o bge_log.o bge_ndd.o \
1769                 bge_atomic.o bge_mii.o bge_send.o bge_recv2.o bge_mii_5906.o

1771 DMFE_OBJS += dmfe_log.o dmfe_main.o dmfe_mii.o

1773 EFE_OBJS += efe.o

1775 ELXL_OBJS += elxl.o

1777 HME_OBJS += hme.o
```



```
1779 IXGB_OBJS += ixgb.o ixgb_atomic.o ixgb_chip.o ixgb_gld.o ixgb_kstats.o \
1780                 ixgb_log.o ixgb_ndd.o ixgb_rx.o ixgb_tx.o ixgb_xmii.o

1782 NGE_OBJS += nge_main.o nge_atomic.o nge_chip.o nge_ndd.o nge_kstats.o \
1783                 nge_log.o nge_rx.o nge_tx.o nge_xmii.o

1785 PCN_OBJS += pcn.o

1787 RGE_OBJS += rge_main.o rge_chip.o rge_ndd.o rge_kstats.o rge_log.o rge_rxtx.o

1789 URTW_OBJS += urtw.o

1791 ARN_OBJS += arn_hw.o arn_eeprom.o arn_mac.o arn_calib.o arn_ani.o arn_phy.o arn_
1792                 arn_main.o arn_recv.o arn_xmit.o arn_rc.o

1794 ATH_OBJS += ath_aux.o ath_main.o ath_osdep.o ath_rate.o

1796 ATU_OBJS += atu.o

1798 IPW_OBJS += ipw2100_hw.o ipw2100.o

1800 IWI_OBJS += ipw2200_hw.o ipw2200.o

1802 IWH_OBJS += iwh.o

1804 IWK_OBJS += iwk2.o

1806 IWP_OBJS += iwp.o

1808 MWL_OBJS += mwl.o

1810 MWLFW_OBJS += mwlfw_mode.o

1812 WPI_OBJS += wpi.o

1814 RAL_OBJS += rt2560.o ral_rate.o

1816 RUM_OBJS += rum.o

1818 RWD_OBJS += rt2661.o

1820 RWN_OBJS += rt2860.o

1822 UATH_OBJS += uath.o

1824 UATHFW_OBJS += uathfw_mod.o

1826 URAL_OBJS += ural.o

1828 RTW_OBJS += rtw.o smc93cx6.o rtwphy.o rtwphyio.o

1830 ZYD_OBJS += zyd.o zyd_usb.o zyd_hw.o zyd_fw.o

1832 MXFE_OBJS += mxfe.o

1834 MPTSAS_OBJS += mptsas.o mptsas_impl.o mptsas_init.o mptsas_raid.o mptsas_smhba.o

1836 SFE_OBJS += sfe.o sfe_util.o

1838 BFE_OBJS += bfe.o

1840 BRIDGE_OBJS += bridge.o

1842 IDM_SHARED_OBJS += base64.o
```

```
1844 IDM_OBJS +=     $(IDM_SHARED_OBJS) \
1845                 idm.o idm_impl.o idm_text.o idm_conn_sm.o idm_so.o

1847 VR_OBJS += vr.o

1849 ATGE_OBJS += atge_main.o atge_l1e.o atge_mii.o atge_l1.o atge_l1c.o

1851 YGE_OBJS = yge.o

1853 #
1854 #       Build up defines and paths.
1855 #
1856 LINT_DEFS       += -Dunix

1858 #
1859 #       This duality can be removed when the native and target compilers
1860 #       are the same (or at least recognize the same command line syntax!)
1861 #       It is a bug in the current compilation system that the assember
1862 #       can't process the -Y I, flag.
1863 #
1864 NATIVE_INC_PATH += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common
1865 AS_INC_PATH     += $(INC_PATH) -I$(UTSBASE)/common
1866 INCLUDE_PATH    += $(INC_PATH) $(CCYFLAG)$(UTSBASE)/common

1868 PCIEB_OBJS += pcieb.o

1870 #       Chelsio N110 10G NIC driver module
1871 #
1872 CH_OBJS = ch.o glue.o pe.o sge.o

1874 CH_COM_OBJS =   ch_mac.o ch_subr.o cspi.o espi.o ixf1010.o mc3.o mc4.o mc5.o \
1875                 mv88e1xxx.o mv88x201x.o my3126.o pm3393.o tp.o ulp.o \
1876                 vsc7321.o vsc7326.o xpak.o

1878 #
1879 #       PCI strings file
1880 #
1881 PCI_STRING_OBJS = pci_strings.o

1883 NET_DACF_OBJS += net_dacf.o

1885 #
1886 #       Xframe 10G NIC driver module
1887 #
1888 XGE_OBJS = xge.o xgell.o

1890 XGE_HAL_OBJS =  xgehal-channel.o xgehal-fifo.o xgehal-ring.o  xgehal-config.o \
1891                 xgehal-driver.o  xgehal-mm.o xgehal-stats.o  xgehal-device.o \
1892                 xge-queue.o  xgehal-mgmt.o xgehal-mgmtaux.o

1894 #
1895 #       e1000g module
1896 #
1897 E1000G_OBJS +=  e1000_80003es2lan.o e1000_82540.o e1000_82541.o e1000_82542.o \
1898                 e1000_82543.o e1000_82571.o e1000_api.o e1000_ich8lan.o \
1899                 e1000_mac.o e1000_manage.o e1000_nvm.o e1000_osdep.o \
1900                 e1000_phy.o e1000g_debug.o e1000g_main.o e1000g_alloc.o \
1901                 e1000g_tx.o e1000g_rx.o e1000g_stat.o

1903 #
1904 #       Intel 82575 1G NIC driver module
1905 #
1906 IGB_OBJS =      igb_82575.o igb_api.o igb_mac.o igb_manage.o \
1907                 igb_nvm.o igb_osdep.o igb_phy.o igb_buf.o \
1908                 igb_debug.o igb_gld.o igb_log.o igb_main.o \
1909                 igb_rx.o igb_stat.o igb_tx.o
```



```
1911 #
1912 #       Intel Pro/100 NIC driver module
1913 #
1914 IPRB_OBJS =     iprb.o

1916 #
1917 #       Intel 10GbE PCIE NIC driver module
1918 #
1919 IXGBE_OBJS =    ixgbe_82598.o ixgbe_82599.o ixgbe_api.o         \
1920                 ixgbe_common.o ixgbe_phy.o                      \
1921                 ixgbe_buf.o ixgbe_debug.o ixgbe_gld.o           \
1922                 ixgbe_log.o ixgbe_main.o                        \
1923                 ixgbe_osdep.o ixgbe_rx.o ixgbe_stat.o           \
1924                 ixgbe_tx.o  ixgbe_x540.o ixgbe_mbx.o

1926 #
1927 #       NIU 10G/1G driver module
1928 #
1929 NXGE_OBJS =     nxge_mac.o nxge_ipp.o nxge_rxdma.o              \
1930                 nxge_txdma.o nxge_txc.o nxge_main.o             \
1931                 nxge_hw.o nxge_fzc.o nxge_virtual.o             \
1932                 nxge_send.o nxge_classify.o nxge_fflp.o         \
1933                 nxge_fflp_hash.o nxge_ndd.o nxge_kstats.o       \
1934                 nxge_zcp.o nxge_fm.o nxge_espc.o nxge_hv.o      \
1935                 nxge_hio.o nxge_hio_guest.o nxge_intr.o

1937 NXGE_NPI_OBJS = \
1938                 npi.o npi_mac.o npi_ipp.o                       \
1939                 npi_txdma.o npi_rxdma.o npi_txc.o               \
1940                 npi_zcp.o npi_espc.o npi_fflp.o                 \
1941                 npi_vir.o

1943 NXGE_HCALL_OBJS =       \
1944                 nxge_hcall.o

1946 #
1947 # Virtio modules
1948 #

1950 # Virtio core
1951 VIRTIO_OBJS = virtio.o

1953 # Virtio block driver
1954 VIOBLK_OBJS = vioblk.o

1956 #
1957 #       kiconv modules
1958 #
1959 KICONV_EMEA_OBJS += kiconv_emea.o

1961 KICONV_JA_OBJS += kiconv_ja.o

1963 KICONV_KO_OBJS += kiconv_cck_common.o kiconv_ko.o

1965 KICONV_SC_OBJS += kiconv_cck_common.o kiconv_sc.o

1967 KICONV_TC_OBJS += kiconv_cck_common.o kiconv_tc.o

1969 #
1970 #       AAC module
1971 #
1972 AAC_OBJS = aac.o aac_ioctl.o

1974 #
1975 #       sdcard modules
```

```
1976 #
1977 SDA_OBJS =      sda_cmd.o sda_host.o sda_init.o sda_mem.o sda_mod.o sda_slot.o
1978 SDHOST_OBJS =   sdhost.o

1980 #
1981 #       hxge 10G driver module
1982 #
1983 HXGE_OBJS =     hxge_main.o hxge_vmac.o hxge_send.o              \
1984                 hxge_txdma.o hxge_rxdma.o hxge_virtual.o         \
1985                 hxge_fm.o hxge_fzc.o hxge_hw.o hxge_kstats.o     \
1986                 hxge_ndd.o hxge_pfc.o                            \
1987                 hpi.o hpi_vmac.o hpi_rxdma.o hpi_txdma.o         \
1988                 hpi_vir.o hpi_pfc.o

1990 #
1991 #       MEGARAID_SAS module
1992 #
1993 MEGA_SAS_OBJS = megaraid_sas.o

1995 #
1996 #       MR_SAS module
1997 #
1998 MR_SAS_OBJS = ld_pd_map.o mr_sas.o mr_sas_tbolt.o mr_sas_list.o
1998 MR_SAS_OBJS = mr_sas.o

2000 #
2001 #       ISCSI_INITIATOR module
2002 #
2003 ISCSI_INITIATOR_OBJS =  chap.o iscsi_io.o iscsi_thread.o         \
2004                         iscsi_ioctl.o iscsid.o iscsi.o           \
2005                         iscsi_login.o isns_client.o iscsiAuthClient.o   \
2006                         iscsi_lun.o iscsiAuthClientGlue.o        \
2007                         iscsi_net.o nvfile.o iscsi_cmd.o         \
2008                         iscsi_queue.o persistent.o iscsi_conn.o \
2009                         iscsi_sess.o radius_auth.o iscsi_crc.o  \
2010                         iscsi_stats.o radius_packet.o iscsi_doorclt.o   \
2011                         iscsi_targetparam.o utils.o kifconf.o

2013 #
2014 #       ntxn 10Gb/1Gb NIC driver module
2015 #
2016 NTXN_OBJS =     unm_nic_init.o unm_gem.o unm_nic_hw.o unm_ndd.o \
2017                         unm_nic_main.o unm_nic_isr.o unm_nic_ctx.o niu.o

2019 #
2020 #       Myricom 10Gb NIC driver module
2021 #
2022 MYRI10GE_OBJS = myri10ge.o myri10ge_lro.o

2024 #       nulldriver module
2025 #
2026 NULLDRIVER_OBJS =       nulldriver.o

2028 TPM_OBJS =      tpm.o tpm_hcall.o
```

```
**********************************************************
   17098 Tue Nov  6 14:28:51 2012
new/usr/src/uts/common/io/mr_sas/fusion.h
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * fusion.h
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 */


#ifndef _FUSION_H_
#define _FUSION_H_

#define U64     uint64_t
#define U32     uint32_t
#define U16     uint16_t
#define U8      uint8_t
#define S8      char
#define S16     short
#define S32     int

/* MPI2 defines */
#define MPI2_REPLY_POST_HOST_INDEX_OFFSET       (0x6C)
#define MPI2_FUNCTION_IOC_INIT                  (0x02) /* IOC Init */
#define MPI2_WHOINIT_HOST_DRIVER                (0x04)
#define MPI2_VERSION_MAJOR                      (0x02)
#define MPI2_VERSION_MINOR                      (0x00)
#define MPI2_VERSION_MAJOR_MASK                 (0xFF00)
#define MPI2_VERSION_MAJOR_SHIFT                (8)
#define MPI2_VERSION_MINOR_MASK                 (0x00FF)
#define MPI2_VERSION_MINOR_SHIFT                (0)
#define MPI2_VERSION    ((MPI2_VERSION_MAJOR << MPI2_VERSION_MAJOR_SHIFT) | \
                         MPI2_VERSION_MINOR)
#define MPI2_HEADER_VERSION_UNIT                (0x10)
#define MPI2_HEADER_VERSION_DEV                 (0x00)
#define MPI2_HEADER_VERSION_UNIT_MASK           (0xFF00)
#define MPI2_HEADER_VERSION_UNIT_SHIFT          (8)
#define MPI2_HEADER_VERSION_DEV_MASK            (0x00FF)
#define MPI2_HEADER_VERSION_DEV_SHIFT           (0)
#define MPI2_HEADER_VERSION                     ((MPI2_HEADER_VERSION_UNIT \
                                                << 8) | \
                                                MPI2_HEADER_VERSION_DEV)
#define MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR      (0x03)
#define MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG    (0x8000)
#define MPI2_SCSIIO_EEDPFLAGS_CHECK_REFTAG      (0x0400)
#define MPI2_SCSIIO_EEDPFLAGS_CHECK_REMOVE_OP   (0x0003)
#define MPI2_SCSIIO_EEDPFLAGS_CHECK_APPTAG      (0x0200)
#define MPI2_SCSIIO_EEDPFLAGS_CHECK_GUARD       (0x0100)
#define MPI2_SCSIIO_EEDPFLAGS_INSERT_OP         (0x0004)
#define MPI2_FUNCTION_SCSI_IO_REQUEST           (0x00) /* SCSI IO */
#define MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY   (0x06)
#define MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO         (0x00)
#define MPI2_SGE_FLAGS_64_BIT_ADDRESSING        (0x02)
#define MPI2_SCSIIO_CONTROL_WRITE               (0x01000000)
```



```
#define MPI2_SCSIIO_CONTROL_READ                (0x02000000)
#define MPI2_REQ_DESCRIPT_FLAGS_TYPE_MASK       (0x0E)
#define MPI2_RPY_DESCRIPT_FLAGS_UNUSED          (0x0F)
#define MPI2_RPY_DESCRIPT_FLAGS_SCSI_IO_SUCCESS (0x00)
#define MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK       (0x0F)
#define MPI2_WRSEQ_FLUSH_KEY_VALUE              (0x0)
#define MPI2_WRITE_SEQUENCE_OFFSET              (0x00000004)
#define MPI2_WRSEQ_1ST_KEY_VALUE                (0xF)
#define MPI2_WRSEQ_2ND_KEY_VALUE                (0x4)
#define MPI2_WRSEQ_3RD_KEY_VALUE                (0xB)
#define MPI2_WRSEQ_4TH_KEY_VALUE                (0x2)
#define MPI2_WRSEQ_5TH_KEY_VALUE                (0x7)
#define MPI2_WRSEQ_6TH_KEY_VALUE                (0xD)

/* Invader defines */
#define MPI2_TYPE_CUDA                                  0x2
#define MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH       0x4000
#define MR_RL_FLAGS_GRANT_DESTINATION_CPU0                      0x00
#define MR_RL_FLAGS_GRANT_DESTINATION_CPU1                      0x10
#define MR_RL_FLAGS_GRANT_DESTINATION_CUDA                      0x80
#define MR_RL_FLAGS_SEQ_NUM_ENABLE                                      0x8
#define MPI2_NSEG_FLAGS_SHIFT                                           4


#define MR_PD_INVALID                           0xFFFF
#define MAX_SPAN_DEPTH                          8
#define MAX_RAIDMAP_SPAN_DEPTH                  (MAX_SPAN_DEPTH)
#define MAX_ROW_SIZE                            32
#define MAX_RAIDMAP_ROW_SIZE                    (MAX_ROW_SIZE)
#define MAX_LOGICAL_DRIVES                      64
#define MAX_RAIDMAP_LOGICAL_DRIVES              (MAX_LOGICAL_DRIVES)
#define MAX_RAIDMAP_VIEWS                       (MAX_LOGICAL_DRIVES)
#define MAX_ARRAYS                              128
#define MAX_RAIDMAP_ARRAYS                      (MAX_ARRAYS)
#define MAX_PHYSICAL_DEVICES                    256
#define MAX_RAIDMAP_PHYSICAL_DEVICES            (MAX_PHYSICAL_DEVICES)

/* get the mapping information of LD */
#define MR_DCMD_LD_MAP_GET_INFO                 0x0300e101

#ifndef MPI2_POINTER
#define MPI2_POINTER    *
#endif

#pragma pack(1)

typedef struct _MPI25_IEEE_SGE_CHAIN64
{
        U64     Address;
        U32     Length;
        U16     Reserved1;
        U8      NextChainOffset;
        U8      Flags;
} MPI25_IEEE_SGE_CHAIN64, MPI2_POINTER PTR_MPI25_IEEE_SGE_CHAIN64,
    Mpi25IeeeSgeChain64_t, MPI2_POINTER pMpi25IeeeSgeChain64_t;

typedef struct _MPI2_SGE_SIMPLE_UNION
{
        U32     FlagsLength;
        union
        {
                U32     Address32;
                U64     Address64;
        } u1;
} MPI2_SGE_SIMPLE_UNION, MPI2_POINTER PTR_MPI2_SGE_SIMPLE_UNION,
    Mpi2SGESimpleUnion_t, MPI2_POINTER pMpi2SGESimpleUnion_t;
```

```
typedef struct
{
        U8      CDB[20];                        /* 0x00 */
        U32     PrimaryReferenceTag;            /* 0x14 */
        U16     PrimaryApplicationTag;          /* 0x18 */
        U16     PrimaryApplicationTagMask;      /* 0x1A */
        U32     TransferLength;                 /* 0x1C */
} MPI2_SCSI_IO_CDB_EEDP32, MPI2_POINTER PTR_MPI2_SCSI_IO_CDB_EEDP32,
    Mpi2ScsiIoCdbEedp32_t, MPI2_POINTER pMpi2ScsiIoCdbEedp32_t;

typedef struct _MPI2_SGE_CHAIN_UNION
{
        U16     Length;
        U8      NextChainOffset;
        U8      Flags;
        union
        {
                U32     Address32;
                U64     Address64;
        } u1;
} MPI2_SGE_CHAIN_UNION, MPI2_POINTER PTR_MPI2_SGE_CHAIN_UNION,
    Mpi2SGEChainUnion_t, MPI2_POINTER pMpi2SGEChainUnion_t;

typedef struct _MPI2_IEEE_SGE_SIMPLE32
{
        U32     Address;
        U32     FlagsLength;
} MPI2_IEEE_SGE_SIMPLE32, MPI2_POINTER PTR_MPI2_IEEE_SGE_SIMPLE32,
    Mpi2IeeeSgeSimple32_t, MPI2_POINTER pMpi2IeeeSgeSimple32_t;

typedef struct _MPI2_IEEE_SGE_SIMPLE64
{
        U64     Address;
        U32     Length;
        U16     Reserved1;
        U8      Reserved2;
        U8      Flags;
} MPI2_IEEE_SGE_SIMPLE64, MPI2_POINTER PTR_MPI2_IEEE_SGE_SIMPLE64,
    Mpi2IeeeSgeSimple64_t, MPI2_POINTER pMpi2IeeeSgeSimple64_t;

typedef union _MPI2_IEEE_SGE_SIMPLE_UNION
{
        MPI2_IEEE_SGE_SIMPLE32  Simple32;
        MPI2_IEEE_SGE_SIMPLE64  Simple64;
} MPI2_IEEE_SGE_SIMPLE_UNION, MPI2_POINTER PTR_MPI2_IEEE_SGE_SIMPLE_UNION,
    Mpi2IeeeSgeSimpleUnion_t, MPI2_POINTER pMpi2IeeeSgeSimpleUnion_t;

typedef MPI2_IEEE_SGE_SIMPLE32  MPI2_IEEE_SGE_CHAIN32;
typedef MPI2_IEEE_SGE_SIMPLE64  MPI2_IEEE_SGE_CHAIN64;

typedef union _MPI2_IEEE_SGE_CHAIN_UNION
{
        MPI2_IEEE_SGE_CHAIN32   Chain32;
        MPI2_IEEE_SGE_CHAIN64   Chain64;
} MPI2_IEEE_SGE_CHAIN_UNION, MPI2_POINTER PTR_MPI2_IEEE_SGE_CHAIN_UNION,
    Mpi2IeeeSgeChainUnion_t, MPI2_POINTER pMpi2IeeeSgeChainUnion_t;

typedef union _MPI2_SGE_IO_UNION
{
        MPI2_SGE_SIMPLE_UNION           MpiSimple;
        MPI2_SGE_CHAIN_UNION            MpiChain;
        MPI2_IEEE_SGE_SIMPLE_UNION      IeeeSimple;
        MPI2_IEEE_SGE_CHAIN_UNION       IeeeChain;
} MPI2_SGE_IO_UNION, MPI2_POINTER PTR_MPI2_SGE_IO_UNION,
    Mpi2SGEIOUnion_t, MPI2_POINTER pMpi2SGEIOUnion_t;
```



```
typedef union
{
        U8                              CDB32[32];
        MPI2_SCSI_IO_CDB_EEDP32         EEDP32;
        MPI2_SGE_SIMPLE_UNION           SGE;
} MPI2_SCSI_IO_CDB_UNION, MPI2_POINTER PTR_MPI2_SCSI_IO_CDB_UNION,
    Mpi2ScsiIoCdb_t, MPI2_POINTER pMpi2ScsiIoCdb_t;

/* Default Request Descriptor */
typedef struct _MPI2_DEFAULT_REQUEST_DESCRIPTOR
{
        U8              RequestFlags;                   /* 0x00 */
        U8              MSIxIndex;                      /* 0x01 */
        U16             SMID;                           /* 0x02 */
        U16             LMID;                           /* 0x04 */
        U16             DescriptorTypeDependent;        /* 0x06 */
} MPI2_DEFAULT_REQUEST_DESCRIPTOR,
    MPI2_POINTER PTR_MPI2_DEFAULT_REQUEST_DESCRIPTOR,
    Mpi2DefaultRequestDescriptor_t,
    MPI2_POINTER pMpi2DefaultRequestDescriptor_t;

/* High Priority Request Descriptor */
typedef struct _MPI2_HIGH_PRIORITY_REQUEST_DESCRIPTOR
{
        U8              RequestFlags;                   /* 0x00 */
        U8              MSIxIndex;                      /* 0x01 */
        U16             SMID;                           /* 0x02 */
        U16             LMID;                           /* 0x04 */
        U16             Reserved1;                      /* 0x06 */
} MPI2_HIGH_PRIORITY_REQUEST_DESCRIPTOR,
    MPI2_POINTER PTR_MPI2_HIGH_PRIORITY_REQUEST_DESCRIPTOR,
    Mpi2HighPriorityRequestDescriptor_t,
    MPI2_POINTER pMpi2HighPriorityRequestDescriptor_t;

/* SCSI IO Request Descriptor */
typedef struct _MPI2_SCSI_IO_REQUEST_DESCRIPTOR
{
        U8              RequestFlags;                   /* 0x00 */
        U8              MSIxIndex;                      /* 0x01 */
        U16             SMID;                           /* 0x02 */
        U16             LMID;                           /* 0x04 */
        U16             DevHandle;                      /* 0x06 */
} MPI2_SCSI_IO_REQUEST_DESCRIPTOR,
    MPI2_POINTER PTR_MPI2_SCSI_IO_REQUEST_DESCRIPTOR,
    Mpi2SCSIIORequestDescriptor_t,
    MPI2_POINTER pMpi2SCSIIORequestDescriptor_t;

/* SCSI Target Request Descriptor */
typedef struct _MPI2_SCSI_TARGET_REQUEST_DESCRIPTOR
{
        U8              RequestFlags;                   /* 0x00 */
        U8              MSIxIndex;                      /* 0x01 */
        U16             SMID;                           /* 0x02 */
        U16             LMID;                           /* 0x04 */
        U16             IoIndex;                        /* 0x06 */
} MPI2_SCSI_TARGET_REQUEST_DESCRIPTOR,
    MPI2_POINTER PTR_MPI2_SCSI_TARGET_REQUEST_DESCRIPTOR,
    Mpi2SCSITargetRequestDescriptor_t,
    MPI2_POINTER pMpi2SCSITargetRequestDescriptor_t;

/* RAID Accelerator Request Descriptor */
typedef struct _MPI2_RAID_ACCEL_REQUEST_DESCRIPTOR
{
        U8              RequestFlags;                   /* 0x00 */
        U8              MSIxIndex;                      /* 0x01 */
```

```
260         U16             SMID;                           /* 0x02 */
261         U16             LMID;                           /* 0x04 */
262         U16             Reserved;                       /* 0x06 */
263 } MPI2_RAID_ACCEL_REQUEST_DESCRIPTOR,
264     MPI2_POINTER PTR_MPI2_RAID_ACCEL_REQUEST_DESCRIPTOR,
265     Mpi2RAIDAcceleratorRequestDescriptor_t,
266     MPI2_POINTER pMpi2RAIDAcceleratorRequestDescriptor_t;

268 /* Default Reply Descriptor */
269 typedef struct _MPI2_DEFAULT_REPLY_DESCRIPTOR
270 {
271         U8              ReplyFlags;                     /* 0x00 */
272         U8              MSIxIndex;                      /* 0x01 */
273         U16             DescriptorTypeDependent1;       /* 0x02 */
274         U32             DescriptorTypeDependent2;       /* 0x04 */
275 } MPI2_DEFAULT_REPLY_DESCRIPTOR, MPI2_POINTER PTR_MPI2_DEFAULT_REPLY_DESCRIPTOR,
276     Mpi2DefaultReplyDescriptor_t, MPI2_POINTER pMpi2DefaultReplyDescriptor_t;

278 /* Address Reply Descriptor */
279 typedef struct _MPI2_ADDRESS_REPLY_DESCRIPTOR
280 {
281         U8              ReplyFlags;                     /* 0x00 */
282         U8              MSIxIndex;                      /* 0x01 */
283         U16             SMID;                           /* 0x02 */
284         U32             ReplyFrameAddress;              /* 0x04 */
285 } MPI2_ADDRESS_REPLY_DESCRIPTOR, MPI2_POINTER PTR_MPI2_ADDRESS_REPLY_DESCRIPTOR,
286     Mpi2AddressReplyDescriptor_t, MPI2_POINTER pMpi2AddressReplyDescriptor_t;

288 /* SCSI IO Success Reply Descriptor */
289 typedef struct _MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR
290 {
291         U8              ReplyFlags;                     /* 0x00 */
292         U8              MSIxIndex;                      /* 0x01 */
293         U16             SMID;                           /* 0x02 */
294         U16             TaskTag;                        /* 0x04 */
295         U16             Reserved1;                      /* 0x06 */
296 } MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR,
297     MPI2_POINTER PTR_MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR,
298     Mpi2SCSIIOSuccessReplyDescriptor_t,
299     MPI2_POINTER pMpi2SCSIIOSuccessReplyDescriptor_t;

301 /* TargetAssist Success Reply Descriptor */
302 typedef struct _MPI2_TARGETASSIST_SUCCESS_REPLY_DESCRIPTOR
303 {
304         U8              ReplyFlags;                     /* 0x00 */
305         U8              MSIxIndex;                      /* 0x01 */
306         U16             SMID;                           /* 0x02 */
307         U8              SequenceNumber;                 /* 0x04 */
308         U8              Reserved1;                      /* 0x05 */
309         U16             IoIndex;                        /* 0x06 */
310 } MPI2_TARGETASSIST_SUCCESS_REPLY_DESCRIPTOR,
311     MPI2_POINTER PTR_MPI2_TARGETASSIST_SUCCESS_REPLY_DESCRIPTOR,
312     Mpi2TargetAssistSuccessReplyDescriptor_t,
313     MPI2_POINTER pMpi2TargetAssistSuccessReplyDescriptor_t;

315 /* Target Command Buffer Reply Descriptor */
316 typedef struct _MPI2_TARGET_COMMAND_BUFFER_REPLY_DESCRIPTOR
317 {
318         U8              ReplyFlags;                     /* 0x00 */
319         U8              MSIxIndex;                      /* 0x01 */
320         U8              VP_ID;                          /* 0x02 */
321         U8              Flags;                          /* 0x03 */
322         U16             InitiatorDevHandle;             /* 0x04 */
323         U16             IoIndex;                        /* 0x06 */
324 } MPI2_TARGET_COMMAND_BUFFER_REPLY_DESCRIPTOR,
325     MPI2_POINTER PTR_MPI2_TARGET_COMMAND_BUFFER_REPLY_DESCRIPTOR,
```



```
326     Mpi2TargetCommandBufferReplyDescriptor_t,
327     MPI2_POINTER pMpi2TargetCommandBufferReplyDescriptor_t;

329 /* RAID Accelerator Success Reply Descriptor */
330 typedef struct _MPI2_RAID_ACCELERATOR_SUCCESS_REPLY_DESCRIPTOR
331 {
332         U8              ReplyFlags;                     /* 0x00 */
333         U8              MSIxIndex;                      /* 0x01 */
334         U16             SMID;                           /* 0x02 */
335         U32             Reserved;                       /* 0x04 */
336 } MPI2_RAID_ACCELERATOR_SUCCESS_REPLY_DESCRIPTOR,
337     MPI2_POINTER PTR_MPI2_RAID_ACCELERATOR_SUCCESS_REPLY_DESCRIPTOR,
338     Mpi2RAIDAcceleratorSuccessReplyDescriptor_t,
339     MPI2_POINTER pMpi2RAIDAcceleratorSuccessReplyDescriptor_t;

341 /* union of Reply Descriptors */
342 typedef union _MPI2_REPLY_DESCRIPTORS_UNION
343 {
344         MPI2_DEFAULT_REPLY_DESCRIPTOR                   Default;
345         MPI2_ADDRESS_REPLY_DESCRIPTOR                   AddressReply;
346         MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR           SCSIIOSuccess;
347         MPI2_TARGETASSIST_SUCCESS_REPLY_DESCRIPTOR      TargetAssistSuccess;
348         MPI2_TARGET_COMMAND_BUFFER_REPLY_DESCRIPTOR     TargetCommandBuffer;
349         MPI2_RAID_ACCELERATOR_SUCCESS_REPLY_DESCRIPTOR  RAIDAcceleratorSuccess;
350         U64                                             Words;
351 } MPI2_REPLY_DESCRIPTORS_UNION, MPI2_POINTER PTR_MPI2_REPLY_DESCRIPTORS_UNION,
352     Mpi2ReplyDescriptorsUnion_t, MPI2_POINTER pMpi2ReplyDescriptorsUnion_t;

354 /* IOCInit Request message */
355 typedef struct _MPI2_IOC_INIT_REQUEST
356 {
357         U8              WhoInit;                                /* 0x00 */
358         U8              Reserved1;                              /* 0x01 */
359         U8              ChainOffset;                            /* 0x02 */
360         U8              Function;                               /* 0x03 */
361         U16             Reserved2;                              /* 0x04 */
362         U8              Reserved3;                              /* 0x06 */
363         U8              MsgFlags;                               /* 0x07 */
364         U8              VP_ID;                                  /* 0x08 */
365         U8              VF_ID;                                  /* 0x09 */
366         U16             Reserved4;                              /* 0x0A */
367         U16             MsgVersion;                             /* 0x0C */
368         U16             HeaderVersion;                          /* 0x0E */
369         U32             Reserved5;                              /* 0x10 */
370         U16             Reserved6;                              /* 0x14 */
371         U8              Reserved7;                              /* 0x16 */
372         U8              HostMSIxVectors;                        /* 0x17 */
373         U16             Reserved8;                              /* 0x18 */
374         U16             SystemRequestFrameSize;                 /* 0x1A */
375         U16             ReplyDescriptorPostQueueDepth;          /* 0x1C */
376         U16             ReplyFreeQueueDepth;                    /* 0x1E */
377         U32             SenseBufferAddressHigh;                 /* 0x20 */
378         U32             SystemReplyAddressHigh;                 /* 0x24 */
379         U64             SystemRequestFrameBaseAddress;          /* 0x28 */
380         U64             ReplyDescriptorPostQueueAddress;        /* 0x30 */
381         U64             ReplyFreeQueueAddress;                  /* 0x38 */
382         U64             TimeStamp;                              /* 0x40 */
383 } MPI2_IOC_INIT_REQUEST, MPI2_POINTER PTR_MPI2_IOC_INIT_REQUEST,
384     Mpi2IOCInitRequest_t, MPI2_POINTER pMpi2IOCInitRequest_t;


387 typedef struct _MR_DEV_HANDLE_INFO {

389         /* Send bitmap of LDs that are idle with respect to FP */
390         U16             curDevHdl;
```

```
392         /* bitmap of valid device handles. */
393         U8              validHandles;
394         U8              reserved;
395         /* 0x04 dev handles for all the paths. */
396         U16             devHandle[2];
397 } MR_DEV_HANDLE_INFO;                           /* 0x08, Total Size */

399 typedef struct _MR_ARRAY_INFO {
400         U16     pd[MAX_RAIDMAP_ROW_SIZE];
401 } MR_ARRAY_INFO;                        /* 0x40, Total Size */

403 typedef struct _MR_QUAD_ELEMENT {
404         U64             logStart;                       /* 0x00 */
405         U64             logEnd;                         /* 0x08 */
406         U64             offsetInSpan;                   /* 0x10 */
407         U32             diff;                           /* 0x18 */
408         U32             reserved1;                      /* 0x1C */
409 } MR_QUAD_ELEMENT;                                      /* 0x20, Total size */

411 typedef struct _MR_SPAN_INFO {
412         U32             noElements;                     /* 0x00 */
413         U32             reserved1;                      /* 0x04 */
414         MR_QUAD_ELEMENT quads[MAX_RAIDMAP_SPAN_DEPTH];  /* 0x08 */
415 } MR_SPAN_INFO;                                         /* 0x108, Total size */

417 typedef struct _MR_LD_SPAN_ {                           /* SPAN structure */
418         /* 0x00, starting block number in array */
419         U64             startBlk;

421         /* 0x08, number of blocks */
422         U64             numBlks;

424         /* 0x10, array reference */
425         U16             arrayRef;

427         U8              reserved[6];    /* 0x12 */
428 } MR_LD_SPAN;                           /* 0x18, Total Size */

430 typedef struct _MR_SPAN_BLOCK_INFO {
431         /* number of rows/span */
432         U64             num_rows;

434         MR_LD_SPAN      span;                           /* 0x08 */
435         MR_SPAN_INFO    block_span_info;                /* 0x20 */
436 } MR_SPAN_BLOCK_INFO;                                   /* 0x128, Total Size */

438 typedef struct _MR_LD_RAID {
439         struct {
440                 U32     fpCapable       :1;
441                 U32     reserved5       :3;
442                 U32     ldPiMode        :4;
443                 U32     pdPiMode        :4;

445                 /* FDE or controller encryption (MR_LD_ENCRYPTION_TYPE) */
446                 U32     encryptionType  :8;

448                 U32     fpWriteCapable  :1;
449                 U32     fpReadCapable   :1;
450                 U32     fpWriteAcrossStripe:1;
451                 U32     fpReadAcrossStripe:1;
452                 U32     reserved4       :8;
453         } capability;                   /* 0x00 */
454         U32     reserved6;
455         U64     size;                   /* 0x08, LD size in blocks */
456         U8      spanDepth;              /* 0x10, Total Number of Spans */
457         U8      level;                  /* 0x11, RAID level */
```



```
458         /* 0x12, shift-count to get stripe size (0=512, 1=1K, 7=64K, etc.) */
459         U8      stripeShift;
460         U8      rowSize;                /* 0x13, number of disks in a row */
461         /* 0x14, number of data disks in a row */
462         U8      rowDataSize;
463         U8      writeMode;              /* 0x15, WRITE_THROUGH or WRITE_BACK */

465         /* 0x16, To differentiate between RAID1 and RAID1E */
466         U8      PRL;

468         U8      SRL;                    /* 0x17 */
469         U16     targetId;               /* 0x18, ld Target Id. */

471         /* 0x1a, state of ld, state corresponds to MR_LD_STATE */
472         U8      ldState;

474         /* 0x1b, Pre calculate region type requests based on MFC etc.. */
475         U8      regTypeReqOnWrite;

477         U8      modFactor;              /* 0x1c, same as rowSize */
478         /*
479          * 0x1d, region lock type used for read, valid only if
480          * regTypeOnReadIsValid=1
481          */
482         U8      regTypeReqOnRead;
483         U16     seqNum;                 /* 0x1e, LD sequence number */

485         struct {
486                 /* This LD requires sync command before completing */
487                 U32     ldSyncRequired:1;
488                 U32     reserved:31;
489         } flags;                        /* 0x20 */

491         U8      reserved3[0x5C];        /* 0x24 */
492 } MR_LD_RAID;                           /* 0x80, Total Size */

494 typedef struct _MR_LD_SPAN_MAP {
495         MR_LD_RAID              ldRaid; /* 0x00 */

497         /* 0x80, needed for GET_ARM() - R0/1/5 only. */
498         U8                      dataArmMap[MAX_RAIDMAP_ROW_SIZE];

500         MR_SPAN_BLOCK_INFO      spanBlock[MAX_RAIDMAP_SPAN_DEPTH]; /* 0xA0 */
501 } MR_LD_SPAN_MAP;       /* 0x9E0 */

503 typedef struct _MR_FW_RAID_MAP {
504         /* total size of this structure, including this field */
505         U32                     totalSize;
506         union {
507                 /* Simple method of version checking variables */
508                 struct {
509                         U32     maxLd;
510                         U32     maxSpanDepth;
511                         U32     maxRowSize;
512                         U32     maxPdCount;
513                         U32     maxArrays;
514                 } validationInfo;
515                 U32     version[5];
516                 U32     reserved1[5];
517         } u1;

519         U32                     ldCount;                /* count of lds */
520         U32                     Reserved1;

522         /*
523          * 0x20 This doesn't correspond to
```

```
524          * FW Ld Tgt Id to LD, but will purge. For example: if tgt Id is 4
525          * and FW LD is 2, and there is only one LD, FW will populate the
526          * array like this. [0xFF, 0xFF, 0xFF, 0xFF, 0x0.....]. This is to
527          * help reduce the entire structure size if there are few LDs or
528          * driver is looking info for 1 LD only.
529          */
530         U8                      ldTgtIdToLd[MAX_RAIDMAP_LOGICAL_DRIVES+ \
531                                 MAX_RAIDMAP_VIEWS]; /* 0x20 */
532         /* timeout value used by driver in FP IOs */
533         U8                      fpPdIoTimeoutSec;
534         U8                      reserved2[7];
535         MR_ARRAY_INFO           arMapInfo[MAX_RAIDMAP_ARRAYS];  /* 0x00a8 */
536         MR_DEV_HANDLE_INFO      devHndlInfo[MAX_RAIDMAP_PHYSICAL_DEVICES];

538         /* 0x28a8-[0 -MAX_RAIDMAP_LOGICAL_DRIVES+MAX_RAIDMAP_VIEWS+1]; */
539         MR_LD_SPAN_MAP          ldSpanMap[1];
540 }MR_FW_RAID_MAP;                                        /* 0x3288, Total Size */

542 typedef struct _LD_TARGET_SYNC {
543         U8      ldTargetId;
544         U8      reserved;
545         U16     seqNum;
546 } LD_TARGET_SYNC;

548 #pragma pack()

550 struct IO_REQUEST_INFO {
551         U64     ldStartBlock;
552         U32     numBlocks;
553         U16     ldTgtId;
554         U8      isRead;
555         U16     devHandle;
556         U64     pdBlock;
557         U8      fpOkForIo;
558         U8      ldPI;
559 };

561 #endif /* _FUSION_H_ */
```

```
**********************************************************
   13182 Tue Nov  6 14:28:52 2012
new/usr/src/uts/common/io/mr_sas/ld_pd_map.c
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
/*
 * ************************************************************************
 *
 * ld_pd_map.c
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 *
 *
 * This module contains functions for device drivers
 * to get pd-ld mapping information.
 *
 * ************************************************************************
 */

#include <sys/scsi/scsi.h>
#include "mr_sas.h"
#include "ld_pd_map.h"

/*
 * This function will check if FAST IO is possible on this logical drive
 * by checking the EVENT information available in the driver
 */
#define MR_LD_STATE_OPTIMAL 3
#define ABS_DIFF(a, b)   (((a) > (b)) ? ((a) - (b)) : ((b) - (a)))

static void mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *,
    PLD_LOAD_BALANCE_INFO);

#define FALSE 0
#define TRUE 1

typedef U64     REGION_KEY;
typedef U32     REGION_LEN;
extern int      debug_level_g;


MR_LD_RAID
*MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
{
        return (&map->raidMap.ldSpanMap[ld].ldRaid);
}

U16
MR_GetLDTgtId(U32 ld, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].ldRaid.targetId);
}


static MR_SPAN_BLOCK_INFO *
MR_LdSpanInfoGet(U32 ld, MR_FW_RAID_MAP_ALL *map)
```



```
{
        return (&map->raidMap.ldSpanMap[ld].spanBlock[0]);
}

static U8
MR_LdDataArmGet(U32 ld, U32 armIdx, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].dataArmMap[armIdx]);
}

static U16
MR_ArPdGet(U32 ar, U32 arm, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.arMapInfo[ar].pd[arm]);
}

static U16
MR_LdSpanArrayGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldSpanMap[ld].spanBlock[span].span.arrayRef);
}

static U16
MR_PdDevHandleGet(U32 pd, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.devHndlInfo[pd].curDevHdl);
}

U16
MR_TargetIdToLdGet(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
{
        return (map->raidMap.ldTgtIdToLd[ldTgtId]);
}

U16
MR_CheckDIF(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map)
{
        MR_LD_RAID      *raid;
        U32             ld;

        ld = MR_TargetIdToLdGet(ldTgtId, map);

        if (ld >= MAX_LOGICAL_DRIVES) {
                return (FALSE);
        }

        raid = MR_LdRaidGet(ld, map);

        return (raid->capability.ldPiMode == 0x8);
}

static MR_LD_SPAN *
MR_LdSpanPtrGet(U32 ld, U32 span, MR_FW_RAID_MAP_ALL *map)
{
        return (&map->raidMap.ldSpanMap[ld].spanBlock[span].span);
}

/*
 * This function will validate Map info data provided by FW
 */
U8
MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo)
{
        MR_FW_RAID_MAP *pFwRaidMap = &map->raidMap;
        U32 fwsize = sizeof (MR_FW_RAID_MAP) - sizeof (MR_LD_SPAN_MAP) +
            (sizeof (MR_LD_SPAN_MAP) * pFwRaidMap->ldCount);
```

```
129         if (pFwRaidMap->totalSize != fwsize) {

131                 con_log(CL_ANN1, (CE_NOTE,
132                     "map info structure size 0x%x is "
133                     "not matching with ld count\n", fwsize));
134                 /* sizeof (foo) returns size_t, which is *LONG*. */
135                 con_log(CL_ANN1, (CE_NOTE, "span map 0x%x total size 0x%x\n",\
136                     (int)sizeof (MR_LD_SPAN_MAP), pFwRaidMap->totalSize));

138                 return (0);
139         }

141         mr_update_load_balance_params(map, lbInfo);

143         return (1);
144 }

146 U32
147 MR_GetSpanBlock(U32 ld, U64 row, U64 *span_blk, MR_FW_RAID_MAP_ALL *map,
148     int *div_error)
149 {
150         MR_SPAN_BLOCK_INFO *pSpanBlock = MR_LdSpanInfoGet(ld, map);
151         MR_QUAD_ELEMENT *qe;
152         MR_LD_RAID      *raid = MR_LdRaidGet(ld, map);
153         U32             span, j;

155         for (span = 0; span < raid->spanDepth; span++, pSpanBlock++) {
156                 for (j = 0; j < pSpanBlock->block_span_info.noElements; j++) {
157                         qe = &pSpanBlock->block_span_info.quads[j];
158                         if (qe->diff == 0) {
159                                 *div_error = 1;
160                                 return (span);
161                         }
162                         if (qe->logStart <= row && row <= qe->logEnd &&
163                             (((row - qe->logStart) % qe->diff)) == 0) {
164                                 if (span_blk != NULL) {
165                                         U64     blk;
166                                         blk = ((row - qe->logStart) /
167                                             (qe->diff));

169                                         blk = (blk + qe->offsetInSpan) <<
170                                             raid->stripeShift;
171                                         *span_blk = blk;
172                                 }
173                                 return (span);
174                         }
175                 }
176         }
177         return (span);
178 }


181 /*
182  * ****************************************************************
183  *
184  * This routine calculates the arm, span and block for
185  * the specified stripe and reference in stripe.
186  *
187  * Inputs :
188  *
189  *    ld   - Logical drive number
190  *    stripRow        - Stripe number
191  *    stripRef    - Reference in stripe
192  *
193  * Outputs :
```



```
194  *
195  *    span          - Span number
196  *    block         - Absolute Block number in the physical disk
197  */
198 U8
199 MR_GetPhyParams(struct mrsas_instance *instance, U32 ld, U64 stripRow,
200     U16 stripRef, U64 *pdBlock, U16 *pDevHandle,
201     MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context, MR_FW_RAID_MAP_ALL *map)
202 {
203         MR_LD_RAID      *raid = MR_LdRaidGet(ld, map);
204         U32             pd, arRef;
205         U8              physArm, span;
206         U64             row;
207         int             error_code = 0;
208         U8              retval = TRUE;
209         U32             rowMod;
210         U32             armQ;
211         U32             arm;

213         ASSERT(raid->rowDataSize != 0);

215         row = (stripRow / raid->rowDataSize);

217         if (raid->level == 6) {
218                 U32 logArm =  (stripRow % (raid->rowDataSize));

220                 if (raid->rowSize == 0) {
221                         return (FALSE);
222                 }
223                 rowMod = (row % (raid->rowSize));
224                 armQ = raid->rowSize-1-rowMod;
225                 arm = armQ + 1 + logArm;
226                 if (arm >= raid->rowSize)
227                         arm -= raid->rowSize;
228                 physArm = (U8)arm;
229         } else {
230                 if (raid->modFactor == 0)
231                         return (FALSE);
232                 physArm = MR_LdDataArmGet(ld,
233                     (stripRow % (raid->modFactor)), map);
234         }
235         if (raid->spanDepth == 1) {
236                 span = 0;
237                 *pdBlock = row << raid->stripeShift;
238         } else
239                 span = (U8)MR_GetSpanBlock(ld, row, pdBlock, map, &error_code);

241         if (error_code == 1)
242                 return (FALSE);

244         /* Get the array on which this span is present. */
245         arRef           = MR_LdSpanArrayGet(ld, span, map);
246         /* Get the Pd. */
247         pd              = MR_ArPdGet(arRef, physArm, map);
248         /* Get dev handle from Pd. */
249         if (pd != MR_PD_INVALID) {
250                 *pDevHandle     = MR_PdDevHandleGet(pd, map);
251         } else {
252                 *pDevHandle = MR_PD_INVALID; /* set dev handle as invalid. */
253                 if ((raid->level >= 5) &&
254                     ((instance->device_id != PCI_DEVICE_ID_LSI_INVADER) ||
255                     (instance->device_id == PCI_DEVICE_ID_LSI_INVADER &&
256                     raid->regTypeReqOnRead != REGION_TYPE_UNUSED))) {
257                         pRAID_Context->regLockFlags = REGION_TYPE_EXCLUSIVE;
258                 } else if (raid->level == 1) {
259                         /* Get Alternate Pd. */
```

```
260                         pd = MR_ArPdGet(arRef, physArm + 1, map);
261                         /* Get dev handle from Pd. */
262                         if (pd != MR_PD_INVALID)
263                                 *pDevHandle = MR_PdDevHandleGet(pd, map);
264                 }
265         }

267         *pdBlock += stripRef + MR_LdSpanPtrGet(ld, span, map)->startBlk;

269         pRAID_Context->spanArm = (span << RAID_CTX_SPANARM_SPAN_SHIFT) |
270             physArm;

272         return (retval);
273 }



277 /*
278  * ****************************************************************************
279  *
280  * MR_BuildRaidContext function
281  *
282  * This function will initiate command processing.  The start/end row and strip
283  * information is calculated then the lock is acquired.
284  * This function will return 0 if region lock
285  * was acquired OR return num strips ???
286  */

288 U8
289 MR_BuildRaidContext(struct mrsas_instance *instance,
290     struct IO_REQUEST_INFO *io_info, MPI2_SCSI_IO_VENDOR_UNIQUE *pRAID_Context,
291     MR_FW_RAID_MAP_ALL *map)
292 {
293         MR_LD_RAID      *raid;
294         U32             ld, stripSize, stripe_mask;
295         U64             endLba, endStrip, endRow;
296         U64             start_row, start_strip;
297         REGION_KEY      regStart;
298         REGION_LEN      regSize;
299         U8              num_strips, numRows;
300         U16             ref_in_start_stripe;
301         U16             ref_in_end_stripe;

303         U64             ldStartBlock;
304         U32             numBlocks, ldTgtId;
305         U8              isRead;
306         U8              retval = 0;

308         ldStartBlock = io_info->ldStartBlock;
309         numBlocks = io_info->numBlocks;
310         ldTgtId = io_info->ldTgtId;
311         isRead = io_info->isRead;

313         if (map == NULL) {
314                 io_info->fpOkForIo = FALSE;
315                 return (FALSE);
316         }

318         ld = MR_TargetIdToLdGet(ldTgtId, map);

320         if (ld >= MAX_LOGICAL_DRIVES) {
321                 io_info->fpOkForIo = FALSE;
322                 return (FALSE);
323         }

325         raid = MR_LdRaidGet(ld, map);
```



```
327         stripSize = 1 << raid->stripeShift;
328         stripe_mask = stripSize-1;
329         /*
330          * calculate starting row and stripe, and number of strips and rows
331          */
332         start_strip             = ldStartBlock >> raid->stripeShift;
333         ref_in_start_stripe     = (U16)(ldStartBlock & stripe_mask);
334         endLba                  = ldStartBlock + numBlocks - 1;
335         ref_in_end_stripe       = (U16)(endLba & stripe_mask);
336         endStrip                = endLba >> raid->stripeShift;
337         num_strips              = (U8)(endStrip - start_strip + 1);
338         /* Check to make sure is not dividing by zero */
339         if (raid->rowDataSize == 0)
340                 return (FALSE);
341         start_row               =  (start_strip / raid->rowDataSize);
342         endRow                  =  (endStrip  / raid->rowDataSize);
343         /* get the row count */
344         numRows                 = (U8)(endRow - start_row + 1);

346         /*
347          * calculate region info.
348          */
349         regStart        = start_row << raid->stripeShift;
350         regSize         = stripSize;

352         /* Check if we can send this I/O via FastPath */
353         if (raid->capability.fpCapable) {
354                 if (isRead) {
355                         io_info->fpOkForIo = (raid->capability.fpReadCapable &&
356                             ((num_strips == 1) ||
357                             raid->capability.fpReadAcrossStripe));
358                 } else {
359                         io_info->fpOkForIo =
360                             (raid->capability.fpWriteCapable &&
361                             ((num_strips == 1) ||
362                             raid->capability.fpWriteAcrossStripe));
363                 }
364         } else
365                 io_info->fpOkForIo = FALSE;


368         /*
369          * Check for DIF support
370          */
371         if (!raid->capability.ldPiMode) {
372                 io_info->ldPI = FALSE;
373         } else {
374                 io_info->ldPI = TRUE;
375         }

377         if (numRows == 1) {
378                 if (num_strips == 1) {
379                         regStart += ref_in_start_stripe;
380                         regSize = numBlocks;
381                 }
382         } else {
383                 if (start_strip == (start_row + 1) * raid->rowDataSize - 1) {
384                         regStart += ref_in_start_stripe;
385                 regSize = stripSize - ref_in_start_stripe;
386                 }

388                 if (numRows > 2) {
389                         regSize += (numRows - 2) << raid->stripeShift;
390                 }
```

```
392                 if (endStrip == endRow * raid->rowDataSize) {
393                         regSize += ref_in_end_stripe + 1;
394                 } else {
395                         regSize += stripSize;
396                 }
397         }

399         pRAID_Context->timeoutValue = map->raidMap.fpPdIoTimeoutSec;

401         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
402                 pRAID_Context->regLockFlags = (isRead) ?
403                     raid->regTypeReqOnRead : raid->regTypeReqOnWrite;
404         } else {
405                 pRAID_Context->regLockFlags = (isRead) ?
406                     REGION_TYPE_SHARED_READ : raid->regTypeReqOnWrite;
407         }

409         pRAID_Context->ldTargetId = raid->targetId;
410         pRAID_Context->regLockRowLBA = regStart;
411         pRAID_Context->regLockLength = regSize;
412         pRAID_Context->configSeqNum = raid->seqNum;

414         /*
415          * Get Phy Params only if FP capable,
416          * or else leave it to MR firmware to do the calculation.
417          */
418         if (io_info->fpOkForIo) {
419                 /* if fast path possible then get the physical parameters */
420                 retval = MR_GetPhyParams(instance, ld, start_strip,
421                     ref_in_start_stripe, &io_info->pdBlock,
422                     &io_info->devHandle, pRAID_Context, map);

424                 /* If IO on an invalid Pd, then FP is not possible. */
425                 if (io_info->devHandle == MR_PD_INVALID)
426                         io_info->fpOkForIo = FALSE;

428                 return (retval);

430         } else if (isRead) {
431                 uint_t stripIdx;

433                 for (stripIdx = 0; stripIdx < num_strips; stripIdx++) {
434                         if (!MR_GetPhyParams(instance, ld,
435                             start_strip + stripIdx, ref_in_start_stripe,
436                             &io_info->pdBlock, &io_info->devHandle,
437                             pRAID_Context, map)) {
438                                 return (TRUE);
439                         }
440                 }
441         }
442         return (TRUE);
443 }


446 void
447 mr_update_load_balance_params(MR_FW_RAID_MAP_ALL *map,
448     PLD_LOAD_BALANCE_INFO lbInfo)
449 {
450         int ldCount;
451         U16 ld;
452         MR_LD_RAID *raid;

454         for (ldCount = 0; ldCount < MAX_LOGICAL_DRIVES; ldCount++) {
455                 ld = MR_TargetIdToLdGet(ldCount, map);

457                 if (ld >= MAX_LOGICAL_DRIVES) {
```



```
458                         con_log(CL_ANN1,
459                             (CE_NOTE, "mrsas: ld=%d Invalid ld \n", ld));
460                         continue;
461                 }

463                 raid = MR_LdRaidGet(ld, map);

465                 /* Two drive Optimal RAID 1 */
466                 if ((raid->level == 1) && (raid->rowSize == 2) &&
467                     (raid->spanDepth == 1) &&
468                     raid->ldState == MR_LD_STATE_OPTIMAL) {
469                         U32 pd, arRef;

471                         lbInfo[ldCount].loadBalanceFlag = 1;

473                         /* Get the array on which this span is present. */
474                         arRef = MR_LdSpanArrayGet(ld, 0, map);

476                         pd = MR_ArPdGet(arRef, 0, map);     /* Get the Pd. */
477                         /* Get dev handle from Pd. */
478                         lbInfo[ldCount].raid1DevHandle[0] =
479                             MR_PdDevHandleGet(pd, map);

481                         pd = MR_ArPdGet(arRef, 1, map);     /* Get the Pd. */
482                         /* Get dev handle from Pd. */
483                         lbInfo[ldCount].raid1DevHandle[1] =
484                             MR_PdDevHandleGet(pd, map);
485                         con_log(CL_ANN1, (CE_NOTE,
486                             "mrsas: ld=%d load balancing enabled \n", ldCount));
487                 } else {
488                         lbInfo[ldCount].loadBalanceFlag = 0;
489                 }
490         }
491 }


494 U8
495 megasas_get_best_arm(PLD_LOAD_BALANCE_INFO lbInfo, U8 arm, U64 block,
496     U32 count)
497 {
498         U16 pend0, pend1;
499         U64 diff0, diff1;
500         U8 bestArm;

502         /* get the pending cmds for the data and mirror arms */
503         pend0 = lbInfo->scsi_pending_cmds[0];
504         pend1 = lbInfo->scsi_pending_cmds[1];

506         /* Determine the disk whose head is nearer to the req. block */
507         diff0 = ABS_DIFF(block, lbInfo->last_accessed_block[0]);
508         diff1 = ABS_DIFF(block, lbInfo->last_accessed_block[1]);
509         bestArm = (diff0 <= diff1 ? 0 : 1);

511         if ((bestArm == arm && pend0 > pend1 + 16) ||
512             (bestArm != arm && pend1 > pend0 + 16)) {
513                 bestArm ^= 1;
514         }

516         /* Update the last accessed block on the correct pd */
517         lbInfo->last_accessed_block[bestArm] = block + count - 1;
518         return (bestArm);
519 }

521 U16
522 get_updated_dev_handle(PLD_LOAD_BALANCE_INFO lbInfo,
523     struct IO_REQUEST_INFO *io_info)
```

```
524 {
525         U8 arm, old_arm;
526         U16 devHandle;
527
528         old_arm = lbInfo->raid1DevHandle[0] == io_info->devHandle ? 0 : 1;
529
530         /* get best new arm */
531         arm  = megasas_get_best_arm(lbInfo, old_arm, io_info->ldStartBlock,
532             io_info->numBlocks);
533
534         devHandle = lbInfo->raid1DevHandle[arm];
535
536         lbInfo->scsi_pending_cmds[arm]++;
537
538         return (devHandle);
539 }
```

```
**********************************************************
    6952 Tue Nov  6 14:28:53 2012
new/usr/src/uts/common/io/mr_sas/ld_pd_map.h
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * ld_pd_map.h
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 */

#ifndef _LD_PD_MAP
#define _LD_PD_MAP
#include <sys/scsi/scsi.h>
#include "fusion.h"

struct mrsas_instance;  /* This will be defined in mr_sas.h */

/* raid->write_mode; raid->read_ahead; dcmd->state */
/* Write through */
#define WRITE_THROUGH                           0
/* Delayed Write */
#define WRITE_BACK                              1

/* SCSI CDB definitions */
#define READ_6          0x08
#define READ_16         0x88
#define READ_10         0x28
#define READ_12         0xA8
#define WRITE_16        0x8A
#define WRITE_10        0x2A

/* maximum disks per array */
#define MAX_ROW_SIZE                            32
/* maximum spans per logical drive */
#define MAX_SPAN_DEPTH                          8
#define MEGASAS_LOAD_BALANCE_FLAG               0x1
#define MR_DEFAULT_IO_TIMEOUT   20


union desc_value {
        U64 word;
        struct {
                U32 low;
                U32 high;
        } u1;
};

typedef struct _LD_LOAD_BALANCE_INFO
{
    U8      loadBalanceFlag;
    U8      reserved1;
    U16     raid1DevHandle[2];
    U16     scsi_pending_cmds[2];
    U64     last_accessed_block[2];
} LD_LOAD_BALANCE_INFO, *PLD_LOAD_BALANCE_INFO;
```



```
#pragma pack(1)
typedef struct _MR_FW_RAID_MAP_ALL {
        MR_FW_RAID_MAP raidMap;
        MR_LD_SPAN_MAP ldSpanMap[MAX_LOGICAL_DRIVES - 1];
} MR_FW_RAID_MAP_ALL;

/*
 * Raid Context structure which describes MegaRAID specific IO Parameters
 * This resides at offset 0x60 where the SGL normally starts in MPT IO Frames
 */
typedef struct _MPI2_SCSI_IO_VENDOR_UNIQUE {
        U8 nsegType;            /* 0x00 nseg[7:4], Type[3:0] */
        U8 resvd0;              /* 0x01 */
        U16 timeoutValue;       /* 0x02 -0x03 */
        U8 regLockFlags;        /* 0x04 */
        U8 reservedForHw1;      /* 0x05 */
        U16 ldTargetId;         /* 0x06 - 0x07 */
        U64 regLockRowLBA;      /* 0x08 - 0x0F */
        U32 regLockLength;      /* 0x10 - 0x13 */
        U16 nextLMId;           /* 0x14 - 0x15 */
        U8 extStatus;           /* 0x16 */
        U8 status;              /* 0x17 status */
        U8 RAIDFlags;           /* 0x18 resvd[7:6], ioSubType[5:4], */
                                /* resvd[3:1], preferredCpu[0] */
        U8 numSGE;              /* 0x19 numSge; not including chain entries */
        U16 configSeqNum;       /* 0x1A -0x1B */
        U8 spanArm;             /* 0x1C span[7:5], arm[4:0] */
        U8 resvd2[3];           /* 0x1D-0x1f */
} MPI2_SCSI_IO_VENDOR_UNIQUE, MPI25_SCSI_IO_VENDOR_UNIQUE;

#define RAID_CTX_SPANARM_ARM_SHIFT      (0)
#define RAID_CTX_SPANARM_ARM_MASK       (0x1f)

#define RAID_CTX_SPANARM_SPAN_SHIFT     (5)
#define RAID_CTX_SPANARM_SPAN_MASK      (0xE0)


/*
 * RAID SCSI IO Request Message
 * Total SGE count will be one less
 * than  _MPI2_SCSI_IO_REQUEST
 */
typedef struct _MPI2_RAID_SCSI_IO_REQUEST
{
        uint16_t                DevHandle;                      /* 0x00 */
        uint8_t                 ChainOffset;                    /* 0x02 */
        uint8_t                 Function;                       /* 0x03 */
        uint16_t                Reserved1;                      /* 0x04 */
        uint8_t                 Reserved2;                      /* 0x06 */
        uint8_t                 MsgFlags;                       /* 0x07 */
        uint8_t                 VP_ID;                          /* 0x08 */
        uint8_t                 VF_ID;                          /* 0x09 */
        uint16_t                Reserved3;                      /* 0x0A */
        uint32_t                SenseBufferLowAddress;          /* 0x0C */
        uint16_t                SGLFlags;                       /* 0x10 */
        uint8_t                 SenseBufferLength;              /* 0x12 */
        uint8_t                 Reserved4;                      /* 0x13 */
        uint8_t                 SGLOffset0;                     /* 0x14 */
        uint8_t                 SGLOffset1;                     /* 0x15 */
        uint8_t                 SGLOffset2;                     /* 0x16 */
        uint8_t                 SGLOffset3;                     /* 0x17 */
        uint32_t                SkipCount;                      /* 0x18 */
        uint32_t                DataLength;                     /* 0x1C */
        uint32_t                BidirectionalDataLength;        /* 0x20 */
        uint16_t                IoFlags;                        /* 0x24 */
```

```
128         uint16_t                EEDPFlags;                      /* 0x26 */
129         uint32_t                EEDPBlockSize;                  /* 0x28 */
130         uint32_t                SecondaryReferenceTag;          /* 0x2C */
131         uint16_t                SecondaryApplicationTag;        /* 0x30 */
132         uint16_t                ApplicationTagTranslationMask;  /* 0x32 */
133         uint8_t                 LUN[8];                         /* 0x34 */
134         uint32_t                Control;                        /* 0x3C */
135         Mpi2ScsiIoCdb_t         CDB;                            /* 0x40 */
136         MPI2_SCSI_IO_VENDOR_UNIQUE RaidContext;                 /* 0x60 */
137         Mpi2SGEIOUnion_t        SGL; /* 0x80 */
138 } MPI2_RAID_SCSI_IO_REQUEST, MPI2_POINTER PTR_MPI2_RAID_SCSI_IO_REQUEST,
139 Mpi2RaidSCSIIORequest_t, MPI2_POINTER pMpi2RaidSCSIIORequest_t;

141 /*
142  * define region lock types
143  */
144 typedef enum    _REGION_TYPE {
145         REGION_TYPE_UNUSED      = 0,    /* lock is currently not active */
146         REGION_TYPE_SHARED_READ = 1,    /* shared lock (for reads) */
147         REGION_TYPE_SHARED_WRITE = 2,
148         REGION_TYPE_EXCLUSIVE   = 3     /* exclusive lock (for writes) */
149 } REGION_TYPE;


152 #define DM_PATH_MAXPATH         2
153 #define DM_PATH_FIRSTPATH       0
154 #define DM_PATH_SECONDPATH      1

156 /* declare valid Region locking values */
157 typedef enum _REGION_LOCK {
158         REGION_LOCK_BYPASS              = 0,
159         /* for RAID 6 single-drive failure */
160         REGION_LOCK_UNCOND_SHARED_READ  = 1,
161         REGION_LOCK_UNCOND_SHARED_WRITE = 2,
162         REGION_LOCK_UNCOND_SHARED_OTHER = 3,
163         REGION_LOCK_UNCOND_SHARED_EXCLUSIVE = 0xFF
164 } REGION_LOCK;


167 struct mrsas_init_frame2 {
168         uint8_t cmd;                            /* 00h */
169         uint8_t reserved_0;                     /* 01h */
170         uint8_t cmd_status;                     /* 02h */

172         uint8_t reserved_1;                     /* 03h */
173         uint32_t reserved_2;                    /* 04h */

175         uint32_t context;                       /* 08h */
176         uint32_t pad_0;                         /* 0Ch */

178         uint16_t flags;                         /* 10h */
179         uint16_t reserved_3;                    /* 12h */
180         uint32_t data_xfer_len;                 /* 14h */

182         uint32_t queue_info_new_phys_addr_lo;   /* 18h */
183         uint32_t queue_info_new_phys_addr_hi;   /* 1Ch */
184         uint32_t queue_info_old_phys_addr_lo;   /* 20h */
185         uint32_t queue_info_old_phys_addr_hi;   /* 24h */
186         uint64_t        driverversion;          /* 28h */
187         uint32_t reserved_4[4];                 /* 30h */
188 };


191 /*
192  * Request descriptor types
193  */
```



```
194 #define MPI2_REQ_DESCRIPT_FLAGS_LD_IO           0x7
195 #define MPI2_REQ_DESCRIPT_FLAGS_MFA             0x1
196 #define MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK         0x2

198 #define MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT      1


201 /*
202  * MPT RAID MFA IO Descriptor.
203  */
204 typedef struct _MR_RAID_MFA_IO_DESCRIPTOR {
205         uint32_t        RequestFlags : 8;
206         uint32_t        MessageAddress1 : 24;   /* bits 31:8 */
207         uint32_t        MessageAddress2;        /* bits 61:32 */
208 } MR_RAID_MFA_IO_REQUEST_DESCRIPTOR,
209 *PMR_RAID_MFA_IO_REQUEST_DESCRIPTOR;

211 /* union of Request Descriptors */
212 typedef union _MRSAS_REQUEST_DESCRIPTOR_UNION
213 {
214         MPI2_DEFAULT_REQUEST_DESCRIPTOR         Default;
215         MPI2_HIGH_PRIORITY_REQUEST_DESCRIPTOR   HighPriority;
216         MPI2_SCSI_IO_REQUEST_DESCRIPTOR         SCSIIO;
217         MPI2_SCSI_TARGET_REQUEST_DESCRIPTOR     SCSITarget;
218         MPI2_RAID_ACCEL_REQUEST_DESCRIPTOR      RAIDAccelerator;
219         MR_RAID_MFA_IO_REQUEST_DESCRIPTOR       MFAIo;
220         U64 Words;
221 } MRSAS_REQUEST_DESCRIPTOR_UNION;

223 #pragma pack()

225 enum {
226         MRSAS_SCSI_VARIABLE_LENGTH_CMD          = 0x7F,
227         MRSAS_SCSI_SERVICE_ACTION_READ32        = 0x9,
228         MRSAS_SCSI_SERVICE_ACTION_WRITE32       = 0xB,
229         MRSAS_SCSI_ADDL_CDB_LEN                 = 0x18,
230         MRSAS_RD_WR_PROTECT                     = 0x20,
231         MRSAS_EEDPBLOCKSIZE                     = 512
232 };


235 #define IEEE_SGE_FLAGS_ADDR_MASK        (0x03)
236 #define IEEE_SGE_FLAGS_SYSTEM_ADDR      (0x00)
237 #define IEEE_SGE_FLAGS_IOCDDR_ADDR      (0x01)
238 #define IEEE_SGE_FLAGS_IOCPLB_ADDR      (0x02)
239 #define IEEE_SGE_FLAGS_IOCPLBNTA_ADDR   (0x03)
240 #define IEEE_SGE_FLAGS_CHAIN_ELEMENT    (0x80)
241 #define IEEE_SGE_FLAGS_END_OF_LIST      (0x40)


244 U8 MR_ValidateMapInfo(MR_FW_RAID_MAP_ALL *map, PLD_LOAD_BALANCE_INFO lbInfo);
245 U16 MR_CheckDIF(U32, MR_FW_RAID_MAP_ALL *);
246 U8 MR_BuildRaidContext(struct mrsas_instance *, struct IO_REQUEST_INFO *,
247     MPI2_SCSI_IO_VENDOR_UNIQUE *, MR_FW_RAID_MAP_ALL *);

249 #endif /* _LD_PD_MAP */
```

```
**********************************************************
  220825 Tue Nov  6 14:28:53 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.c
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
/*
 * mr_sas.c: source for mr_sas driver
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2010, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 *              Rajesh Prabhakaran
 *              Seokmann Ju
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. 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.
 */

/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2011 Bayard G. Bell. All rights reserved.
 * Copyright 2012 Nexenta System, Inc. All rights reserved.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/modctl.h>
#include <sys/conf.h>
```



```
#include <sys/devops.h>
#include <sys/cmn_err.h>
#include <sys/kmem.h>
#include <sys/stat.h>
#include <sys/mkdev.h>
#include <sys/pci.h>
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/atomic.h>
#include <sys/signal.h>
#include <sys/byteorder.h>
#include <sys/sdt.h>
#include <sys/fs/dv_node.h>     /* devfs_clean */

#include "mr_sas.h"

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

/*
 * Local static data
 */
static void     *mrsas_state = NULL;
static volatile boolean_t       mrsas_relaxed_ordering = B_TRUE;
volatile int    debug_level_g = CL_NONE;
static volatile int     debug_level_g = CL_NONE;
static volatile int     msi_enable = 1;
static volatile int     ctio_enable = 1;

/* Default Timeout value to issue online controller reset */
volatile int  debug_timeout_g  = 0xF0;          /* 0xB4; */
static volatile int  debug_timeout_g  = 0xB4;
/* Simulate consecutive firmware fault */
static volatile int  debug_fw_faults_after_ocr_g  = 0;

#ifdef OCRDEBUG
/* Simulate three consecutive timeout for an IO */
static volatile int  debug_consecutive_timeout_after_ocr_g  = 0;
#endif

#pragma weak scsi_hba_open
#pragma weak scsi_hba_close
#pragma weak scsi_hba_ioctl

/* Local static prototypes. */
static int      mrsas_getinfo(dev_info_t *, ddi_info_cmd_t,  void *, void **);
static int      mrsas_attach(dev_info_t *, ddi_attach_cmd_t);
#ifdef __sparc
static int      mrsas_reset(dev_info_t *, ddi_reset_cmd_t);
#else
static int      mrsas_quiesce(dev_info_t *);
#endif
static int      mrsas_detach(dev_info_t *, ddi_detach_cmd_t);
static int      mrsas_open(dev_t *, int, int, cred_t *);
static int      mrsas_close(dev_t, int, int, cred_t *);
static int      mrsas_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

static int      mrsas_tran_tgt_init(dev_info_t *, dev_info_t *,
                    scsi_hba_tran_t *, struct scsi_device *);
static struct scsi_pkt *mrsas_tran_init_pkt(struct scsi_address *, register
```

```
121                     struct scsi_pkt *, struct buf *, int, int, int, int,
122                     int (*)(), caddr_t);
123 static int      mrsas_tran_start(struct scsi_address *,
124                     register struct scsi_pkt *);
125 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
126 static int      mrsas_tran_reset(struct scsi_address *, int);
127 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
128 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
129 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
130                     struct scsi_pkt *);
131 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
132 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
133 static int      mrsas_tran_quiesce(dev_info_t *dip);
134 static int      mrsas_tran_unquiesce(dev_info_t *dip);
135 static uint_t   mrsas_isr();
136 static uint_t   mrsas_softintr();
137 static void     mrsas_undo_resources(dev_info_t *, struct mrsas_instance *);
138 static struct mrsas_cmd *get_mfi_pkt(struct mrsas_instance *);
139 static void     return_mfi_pkt(struct mrsas_instance *,
140                     struct mrsas_cmd *);

142 static void     free_space_for_mfi(struct mrsas_instance *);
143 static uint32_t read_fw_status_reg_ppc(struct mrsas_instance *);
144 static void     issue_cmd_ppc(struct mrsas_cmd *, struct mrsas_instance *);
145 static int      issue_cmd_in_poll_mode_ppc(struct mrsas_instance *,
146                     struct mrsas_cmd *);
147 static int      issue_cmd_in_sync_mode_ppc(struct mrsas_instance *,
148                     struct mrsas_cmd *);
149 static void     enable_intr_ppc(struct mrsas_instance *);
150 static void     disable_intr_ppc(struct mrsas_instance *);
151 static int      intr_ack_ppc(struct mrsas_instance *);
152 static void     flush_cache(struct mrsas_instance *instance);
153 void    display_scsi_inquiry(caddr_t);
154 static int      start_mfi_aen(struct mrsas_instance *instance);
155 static int      handle_drv_ioctl(struct mrsas_instance *instance,
156                     struct mrsas_ioctl *ioctl, int mode);
157 static int      handle_mfi_ioctl(struct mrsas_instance *instance,
158                     struct mrsas_ioctl *ioctl, int mode);
159 static int      handle_mfi_aen(struct mrsas_instance *instance,
160                     struct mrsas_aen *aen);
161 static struct mrsas_cmd *build_cmd(struct mrsas_instance *,
162     struct scsi_address *, struct scsi_pkt *, uchar_t *);
163 static int      alloc_additional_dma_buffer(struct mrsas_instance *);
164 static void     complete_cmd_in_sync_mode(struct mrsas_instance *,
165                     struct mrsas_cmd *);
166 static int      mrsas_kill_adapter(struct mrsas_instance *);
167 static int      mrsas_issue_init_mfi(struct mrsas_instance *);
168 static int      mrsas_reset_ppc(struct mrsas_instance *);
169 static uint32_t mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *);
170 static int      wait_for_outstanding(struct mrsas_instance *instance);
171 static int      register_mfi_aen(struct mrsas_instance *instance,
172                     uint32_t seq_num, uint32_t class_locale_word);
173 static int      issue_mfi_pthru(struct mrsas_instance *instance, struct
174                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
175 static int      issue_mfi_dcmd(struct mrsas_instance *instance, struct
176                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
177 static int      issue_mfi_smp(struct mrsas_instance *instance, struct
178                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
179 static int      issue_mfi_stp(struct mrsas_instance *instance, struct
180                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
181 static int      abort_aen_cmd(struct mrsas_instance *instance,
182                     struct mrsas_cmd *cmd_to_abort);

184 static void     mrsas_rem_intrs(struct mrsas_instance *instance);
185 static int      mrsas_add_intrs(struct mrsas_instance *instance, int intr_type);
```



```
187 static void     mrsas_tran_tgt_free(dev_info_t *, dev_info_t *,
188                     scsi_hba_tran_t *, struct scsi_device *);
189 static int      mrsas_tran_bus_config(dev_info_t *, uint_t,
190                     ddi_bus_config_op_t, void *, dev_info_t **);
191 static int      mrsas_parse_devname(char *, int *, int *);
192 static int      mrsas_config_all_devices(struct mrsas_instance *);
193 static int      mrsas_config_ld(struct mrsas_instance *, uint16_t,
194                         uint8_t, dev_info_t **);
195 static int      mrsas_name_node(dev_info_t *, char *, int);
196 static void     mrsas_issue_evt_taskq(struct mrsas_eventinfo *);
197 static void     free_additional_dma_buffer(struct mrsas_instance *);
198 static void io_timeout_checker(void *);
199 static void mrsas_fm_init(struct mrsas_instance *);
200 static void mrsas_fm_fini(struct mrsas_instance *);

202 static struct mrsas_function_template mrsas_function_template_ppc = {
203         .read_fw_status_reg = read_fw_status_reg_ppc,
204         .issue_cmd = issue_cmd_ppc,
205         .issue_cmd_in_sync_mode = issue_cmd_in_sync_mode_ppc,
206         .issue_cmd_in_poll_mode = issue_cmd_in_poll_mode_ppc,
207         .enable_intr = enable_intr_ppc,
208         .disable_intr = disable_intr_ppc,
209         .intr_ack = intr_ack_ppc,
210         .init_adapter = mrsas_init_adapter_ppc
211 };


214 static struct mrsas_function_template mrsas_function_template_fusion = {
215         .read_fw_status_reg = tbolt_read_fw_status_reg,
216         .issue_cmd = tbolt_issue_cmd,
217         .issue_cmd_in_sync_mode = tbolt_issue_cmd_in_sync_mode,
218         .issue_cmd_in_poll_mode = tbolt_issue_cmd_in_poll_mode,
219         .enable_intr = tbolt_enable_intr,
220         .disable_intr = tbolt_disable_intr,
221         .intr_ack = tbolt_intr_ack,
222         .init_adapter = mrsas_init_adapter_tbolt
223 };


226 ddi_dma_attr_t mrsas_generic_dma_attr = {
104 static ddi_dma_attr_t mrsas_generic_dma_attr = {
227         DMA_ATTR_V0,            /* dma_attr_version */
228         0,                      /* low DMA address range */
229         0xFFFFFFFFU,            /* high DMA address range */
230         0xFFFFFFFFU,            /* DMA counter register  */
231         8,                      /* DMA address alignment */
232         0x07,                   /* DMA burstsizes  */
233         1,                      /* min DMA size */
234         0xFFFFFFFFU,            /* max DMA size */
235         0xFFFFFFFFU,            /* segment boundary */
236         MRSAS_MAX_SGE_CNT,      /* dma_attr_sglen */
237         512,                    /* granularity of device */
238         0                       /* bus specific DMA flags */
239 };

241 int32_t mrsas_max_cap_maxxfer = 0x1000000;

243 /*
244  * Fix for: Thunderbolt controller IO timeout when IO write size is 1MEG,
245  * Limit size to 256K
246  */
247 uint32_t mrsas_tbolt_max_cap_maxxfer = (512 * 512);

249 /*
250  * cb_ops contains base level routines
251  */
```

```
252 static struct cb_ops mrsas_cb_ops = {
253         mrsas_open,             /* open */
254         mrsas_close,            /* close */
255         nodev,                  /* strategy */
256         nodev,                  /* print */
257         nodev,                  /* dump */
258         nodev,                  /* read */
259         nodev,                  /* write */
260         mrsas_ioctl,            /* ioctl */
261         nodev,                  /* devmap */
262         nodev,                  /* mmap */
263         nodev,                  /* segmap */
264         nochpoll,               /* poll */
265         nodev,                  /* cb_prop_op */
266         0,                      /* streamtab  */
267         D_NEW | D_HOTPLUG,      /* cb_flag */
268         CB_REV,                 /* cb_rev */
269         nodev,                  /* cb_aread */
270         nodev                   /* cb_awrite */
271 };
```

*unchanged_portion_omitted_*

```
318 /* Use the LSI Fast Path for the 2208 (tbolt) commands. */
319 unsigned int enable_fp = 1;


322 /*
323  * ************************************************************************** *
324  *                                                                            *
325  *         common entry points - for loadable kernel modules                  *
326  *                                                                            *
327  * ************************************************************************** *
328  */

330 /*
331  * _init - initialize a loadable module
332  * @void
333  *
334  * The driver should perform any one-time resource allocation or data
335  * initialization during driver loading in _init(). For example, the driver
336  * should initialize any mutexes global to the driver in this routine.
337  * The driver should not, however, use _init() to allocate or initialize
338  * anything that has to do with a particular instance of the device.
339  * Per-instance initialization must be done in attach().
340  */
341 int
342 _init(void)
343 {
344         int ret;

346         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

348         ret = ddi_soft_state_init(&mrsas_state,
349             sizeof (struct mrsas_instance), 0);

351         if (ret != DDI_SUCCESS) {
352                 cmn_err(CE_WARN, "mr_sas: could not init state");
210                 con_log(CL_ANN, (CE_WARN, "mr_sas: could not init state"));
353                 return (ret);
354         }

356         if ((ret = scsi_hba_init(&modlinkage)) != DDI_SUCCESS) {
357                 cmn_err(CE_WARN, "mr_sas: could not init scsi hba");
215                 con_log(CL_ANN, (CE_WARN, "mr_sas: could not init scsi hba"));
358                 ddi_soft_state_fini(&mrsas_state);
359                 return (ret);
```



```
360         }

362         ret = mod_install(&modlinkage);

364         if (ret != DDI_SUCCESS) {
365                 cmn_err(CE_WARN, "mr_sas: mod_install failed");
223                 con_log(CL_ANN, (CE_WARN, "mr_sas: mod_install failed"));
366                 scsi_hba_fini(&modlinkage);
367                 ddi_soft_state_fini(&mrsas_state);
368         }

370         return (ret);
371 }

373 /*
374  * _info - returns information about a loadable module.
375  * @void
376  *
377  * _info() is called to return module information. This is a typical entry
378  * point that does predefined role. It simply calls mod_info().
379  */
380 int
381 _info(struct modinfo *modinfop)
382 {
383         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

385         return (mod_info(&modlinkage, modinfop));
386 }

388 /*
389  * _fini - prepare a loadable module for unloading
390  * @void
391  *
392  * In _fini(), the driver should release any resources that were allocated in
393  * _init(). The driver must remove itself from the system module list.
394  */
395 int
396 _fini(void)
397 {
398         int ret;

400         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

402         if ((ret = mod_remove(&modlinkage)) != DDI_SUCCESS) {
403                 con_log(CL_ANN1,
404                     (CE_WARN, "_fini: mod_remove() failed, error 0x%X", ret));
246         if ((ret = mod_remove(&modlinkage)) != DDI_SUCCESS)
405                 return (ret);
406         }

408         scsi_hba_fini(&modlinkage);
409         con_log(CL_DLEVEL1, (CE_NOTE, "_fini: scsi_hba_fini() done."));

411         ddi_soft_state_fini(&mrsas_state);
412         con_log(CL_DLEVEL1, (CE_NOTE, "_fini: ddi_soft_state_fini() done."));

414         return (ret);
415 }


418 /*
419  * ************************************************************************** *
420  *                                                                            *
421  *               common entry points - for autoconfiguration                  *
422  *                                                                            *
423  * ************************************************************************** *
```

```
 */
/*
 * attach - adds a device to the system as part of initialization
 * @dip:
 * @cmd:
 *
 * The kernel calls a driver's attach() entry point to attach an instance of
 * a device (for MegaRAID, it is instance of a controller) or to resume
 * operation for an instance of a device that has been suspended or has been
 * shut down by the power management framework
 * The attach() entry point typically includes the following types of
 * processing:
 * - allocate a soft-state structure for the device instance (for MegaRAID,
 *   controller instance)
 * - initialize per-instance mutexes
 * - initialize condition variables
 * - register the device's interrupts (for MegaRAID, controller's interrupts)
 * - map the registers and memory of the device instance (for MegaRAID,
 *   controller instance)
 * - create minor device nodes for the device instance (for MegaRAID,
 *   controller instance)
 * - report that the device instance (for MegaRAID, controller instance) has
 *   attached
 */

static int
mrsas_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int             instance_no;
        int             nregs;
        int             i = 0;
        uint8_t         added_isr_f = 0;
        uint8_t         added_soft_isr_f = 0;
        uint8_t         create_devctl_node_f = 0;
        uint8_t         create_scsi_node_f = 0;
        uint8_t         create_ioc_node_f = 0;
        uint8_t         tran_alloc_f = 0;
        uint8_t         irq;
        uint16_t        vendor_id;
        uint16_t        device_id;
        uint16_t        subsysvid;
        uint16_t        subsysid;
        uint16_t        command;
        off_t           reglength = 0;
        int             intr_types = 0;
        char            *data;

        scsi_hba_tran_t         *tran;
        ddi_dma_attr_t  tran_dma_attr;
        struct mrsas_instance   *instance;

        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

        /* CONSTCOND */
        ASSERT(NO_COMPETING_THREADS);

        instance_no = ddi_get_instance(dip);

        /*
         * check to see whether this device is in a DMA-capable slot.
         */
        if (ddi_slaveonly(dip) == DDI_SUCCESS) {
                cmn_err(CE_WARN,
                con_log(CL_ANN, (CE_WARN,
                    "mr_sas%d: Device in slave-only slot, unused",
                    instance_no);
```



```
                    instance_no));
                return (DDI_FAILURE);
        }

        switch (cmd) {
        case DDI_ATTACH:
                con_log(CL_DLEVEL1, (CE_NOTE, "mr_sas: DDI_ATTACH"));
                /* allocate the soft state for the instance */
                if (ddi_soft_state_zalloc(mrsas_state, instance_no)
                    != DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                                con_log(CL_ANN, (CE_WARN,
                            "mr_sas%d: Failed to allocate soft state",
                            instance_no);
                                    instance_no));

                        return (DDI_FAILURE);
                }

                instance = (struct mrsas_instance *)ddi_get_soft_state
                    (mrsas_state, instance_no);

                if (instance == NULL) {
                        cmn_err(CE_WARN,
                            "mr_sas%d: Bad soft state", instance_no);
                                con_log(CL_ANN, (CE_WARN,
                                    "mr_sas%d: Bad soft state", instance_no));

                        ddi_soft_state_free(mrsas_state, instance_no);

                        return (DDI_FAILURE);
                }

                instance->unroll.softs  = 1;
                        bzero((caddr_t)instance,
                            sizeof (struct mrsas_instance));

                        instance->func_ptr = kmem_zalloc(
                            sizeof (struct mrsas_func_ptr), KM_SLEEP);
                        ASSERT(instance->func_ptr);

                /* Setup the PCI configuration space handles */
                if (pci_config_setup(dip, &instance->pci_handle) !=
                    DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                                con_log(CL_ANN, (CE_WARN,
                            "mr_sas%d: pci config setup failed ",
                            instance_no);
                                    instance_no));

                                kmem_free(instance->func_ptr,
                                    sizeof (struct mrsas_func_ptr));
                        ddi_soft_state_free(mrsas_state, instance_no);

                        return (DDI_FAILURE);
                }

                if (ddi_dev_nregs(dip, &nregs) != DDI_SUCCESS) {
                        cmn_err(CE_WARN,
                            "mr_sas: failed to get registers.");
                                con_log(CL_ANN, (CE_WARN,
                                    "mr_sas: failed to get registers."));

                        pci_config_teardown(&instance->pci_handle);
                                kmem_free(instance->func_ptr,
                                    sizeof (struct mrsas_func_ptr));
```

```
524                         ddi_soft_state_free(mrsas_state, instance_no);
525                         return (DDI_FAILURE);
526                 }
528                 vendor_id = pci_config_get16(instance->pci_handle,
529                     PCI_CONF_VENID);
530                 device_id = pci_config_get16(instance->pci_handle,
531                     PCI_CONF_DEVID);
533                 subsysvid = pci_config_get16(instance->pci_handle,
534                     PCI_CONF_SUBVENID);
535                 subsysid = pci_config_get16(instance->pci_handle,
536                     PCI_CONF_SUBSYSID);
538                 pci_config_put16(instance->pci_handle, PCI_CONF_COMM,
539                     (pci_config_get16(instance->pci_handle,
540                     PCI_CONF_COMM) | PCI_COMM_ME));
541                 irq = pci_config_get8(instance->pci_handle,
542                     PCI_CONF_ILINE);
544                 con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
545                     "0x%x:0x%x 0x%x:0x%x, irq:%d drv-ver:%s",
546                     instance_no, vendor_id, device_id, subsysvid,
547                     subsysid, irq, MRSAS_VERSION));
549                 /* enable bus-mastering */
550                 command = pci_config_get16(instance->pci_handle,
551                     PCI_CONF_COMM);
553                 if (!(command & PCI_COMM_ME)) {
554                         command |= PCI_COMM_ME;
556                         pci_config_put16(instance->pci_handle,
557                             PCI_CONF_COMM, command);
559                         con_log(CL_ANN, (CE_CONT, "mr_sas%d: "
560                             "enable bus-mastering", instance_no));
561                 } else {
562                         con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
563                             "bus-mastering already set", instance_no));
564                 }
566                 /* initialize function pointers */
567                 switch (device_id) {
568                 case PCI_DEVICE_ID_LSI_TBOLT:
569                 case PCI_DEVICE_ID_LSI_INVADER:
570                         con_log(CL_ANN, (CE_NOTE,
571                             "mr_sas: 2208 T.B. device detected"));
404                         if ((device_id == PCI_DEVICE_ID_LSI_2108VDE) ||
405                             (device_id == PCI_DEVICE_ID_LSI_2108V)) {
406                                 con_log(CL_DLEVEL1, (CE_CONT, "mr_sas%d: "
407                                     "2108V/DE detected", instance_no));
408                                 instance->func_ptr->read_fw_status_reg =
409                                     read_fw_status_reg_ppc;
410                                 instance->func_ptr->issue_cmd = issue_cmd_ppc;
411                                 instance->func_ptr->issue_cmd_in_sync_mode =
412                                     issue_cmd_in_sync_mode_ppc;
413                                 instance->func_ptr->issue_cmd_in_poll_mode =
414                                     issue_cmd_in_poll_mode_ppc;
415                                 instance->func_ptr->enable_intr =
416                                     enable_intr_ppc;
417                                 instance->func_ptr->disable_intr =
418                                     disable_intr_ppc;
419                                 instance->func_ptr->intr_ack = intr_ack_ppc;
420                         } else {
```



```
421                                 con_log(CL_ANN, (CE_WARN,
422                                     "mr_sas: Invalid device detected"));
573                         instance->func_ptr =
574                             &mrsas_function_template_fusion;
575                         instance->tbolt = 1;
576                         break;
578                 case PCI_DEVICE_ID_LSI_2108VDE:
579                 case PCI_DEVICE_ID_LSI_2108V:
580                         con_log(CL_ANN, (CE_NOTE,
581                             "mr_sas: 2108 Liberator device detected"));
583                         instance->func_ptr =
584                             &mrsas_function_template_ppc;
585                         break;
587                 default:
588                         cmn_err(CE_WARN,
589                             "mr_sas: Invalid device detected");
591                         pci_config_teardown(&instance->pci_handle);
425                                 kmem_free(instance->func_ptr,
426                                     sizeof (struct mrsas_func_ptr));
592                         ddi_soft_state_free(mrsas_state, instance_no);
593                         return (DDI_FAILURE);
594                 }
596                 instance->baseaddress = pci_config_get32(
597                     instance->pci_handle, PCI_CONF_BASE0);
598                 instance->baseaddress &= 0x0fffc;
600                 instance->dip           = dip;
601                 instance->vendor_id     = vendor_id;
602                 instance->device_id     = device_id;
603                 instance->subsysvid     = subsysvid;
604                 instance->subsysid      = subsysid;
605                 instance->instance      = instance_no;
607                 /* Initialize FMA */
608                 instance->fm_capabilities = ddi_prop_get_int(
609                     DDI_DEV_T_ANY, instance->dip, DDI_PROP_DONTPASS,
610                     "fm-capable", DDI_FM_EREPORT_CAPABLE |
611                     DDI_FM_ACCCHK_CAPABLE | DDI_FM_DMACHK_CAPABLE
612                     | DDI_FM_ERRCB_CAPABLE);
614                 mrsas_fm_init(instance);
616                 /* Setup register map */
452                         /* Initialize Interrupts */
617                 if ((ddi_dev_regsize(instance->dip,
618                     REGISTER_SET_IO_2108, &reglength) != DDI_SUCCESS) ||
619                     reglength < MINIMUM_MFI_MEM_SZ) {
620                         goto fail_attach;
456                                 return (DDI_FAILURE);
621                 }
622                 if (reglength > DEFAULT_MFI_MEM_SZ) {
623                         reglength = DEFAULT_MFI_MEM_SZ;
624                         con_log(CL_DLEVEL1, (CE_NOTE,
625                             "mr_sas: register length to map is 0x%lx bytes",
626                             reglength));
461                                 "mr_sas: register length to map is "
462                                 "0x%lx bytes", reglength));
627                 }
628                 if (ddi_regs_map_setup(instance->dip,
```

```
629                     REGISTER_SET_IO_2108, &instance->regmap, 0,
630                     reglength, &endian_attr, &instance->regmap_handle)
631                     != DDI_SUCCESS) {
632                         cmn_err(CE_WARN,
633                             "mr_sas: couldn't map control registers");
468                             con_log(CL_ANN, (CE_NOTE,
469                                 "mr_sas: couldn't map control registers"));
634                         goto fail_attach;
635                 }

637                 instance->unroll.regs = 1;

639                 /*
640                  * Disable Interrupt Now.
641                  * Setup Software interrupt
642                  */
643                 instance->func_ptr->disable_intr(instance);

645                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
646                     "mrsas-enable-msi", &data) == DDI_SUCCESS) {
647                         if (strncmp(data, "no", 3) == 0) {
648                                 msi_enable = 0;
649                                 con_log(CL_ANN1, (CE_WARN,
650                                     "msi_enable = %d disabled", msi_enable));
484                                         "msi_enable = %d disabled",
485                                         msi_enable));
651                         }
652                         ddi_prop_free(data);
653                 }

655                 con_log(CL_DLEVEL1, (CE_NOTE, "msi_enable = %d", msi_enable));
490                         con_log(CL_DLEVEL1, (CE_WARN, "msi_enable = %d",
491                             msi_enable));

657                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
658                     "mrsas-enable-fp", &data) == DDI_SUCCESS) {
659                         if (strncmp(data, "no", 3) == 0) {
660                                 enable_fp = 0;
661                                 cmn_err(CE_NOTE,
662                                     "enable_fp = %d, Fast-Path disabled.\n",
663                                     enable_fp);
664                         }

666                         ddi_prop_free(data);
667                 }

669                 con_log(CL_DLEVEL1, (CE_NOTE, "enable_fp = %d\n", enable_fp));

671                 /* Check for all supported interrupt types */
672                 if (ddi_intr_get_supported_types(
673                     dip, &intr_types) != DDI_SUCCESS) {
674                         cmn_err(CE_WARN,
675                             "ddi_intr_get_supported_types() failed");
496                             con_log(CL_ANN, (CE_WARN,
497                                 "ddi_intr_get_supported_types() failed"));
676                         goto fail_attach;
677                 }

679                 con_log(CL_DLEVEL1, (CE_NOTE,
680                     "ddi_intr_get_supported_types() ret: 0x%x", intr_types));
502                         "ddi_intr_get_supported_types() ret: 0x%x",
503                         intr_types));

682                 /* Initialize and Setup Interrupt handler */
683                 if (msi_enable && (intr_types & DDI_INTR_TYPE_MSIX)) {
684                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSIX) !=
```



```
685                             DDI_SUCCESS) {
686                                 cmn_err(CE_WARN,
687                                     "MSIX interrupt query failed");
507                                 if (mrsas_add_intrs(instance,
508                                     DDI_INTR_TYPE_MSIX) != DDI_SUCCESS) {
509                                         con_log(CL_ANN, (CE_WARN,
510                                             "MSIX interrupt query failed"));
688                                 goto fail_attach;
689                         }
690                         instance->intr_type = DDI_INTR_TYPE_MSIX;
691                 } else if (msi_enable && (intr_types & DDI_INTR_TYPE_MSI)) {
692                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_MSI) !=
693                             DDI_SUCCESS) {
694                                 cmn_err(CE_WARN,
695                                     "MSI interrupt query failed");
514                         } else if (msi_enable && (intr_types &
515                             DDI_INTR_TYPE_MSI)) {
516                                 if (mrsas_add_intrs(instance,
517                                     DDI_INTR_TYPE_MSI) != DDI_SUCCESS) {
518                                         con_log(CL_ANN, (CE_WARN,
519                                             "MSI interrupt query failed"));
696                                 goto fail_attach;
697                         }
698                         instance->intr_type = DDI_INTR_TYPE_MSI;
699                 } else if (intr_types & DDI_INTR_TYPE_FIXED) {
700                         msi_enable = 0;
701                         if (mrsas_add_intrs(instance, DDI_INTR_TYPE_FIXED) !=
702                             DDI_SUCCESS) {
703                                 cmn_err(CE_WARN,
704                                     "FIXED interrupt query failed");
525                                 if (mrsas_add_intrs(instance,
526                                     DDI_INTR_TYPE_FIXED) != DDI_SUCCESS) {
527                                         con_log(CL_ANN, (CE_WARN,
528                                             "FIXED interrupt query failed"));
705                                 goto fail_attach;
706                         }
707                         instance->intr_type = DDI_INTR_TYPE_FIXED;
708                 } else {
709                         cmn_err(CE_WARN, "Device cannot "
533                                 con_log(CL_ANN, (CE_WARN, "Device cannot "
710                             "suppport either FIXED or MSI/X "
711                             "interrupts");
535                                     "interrupts"));
712                         goto fail_attach;
713                 }

715                 instance->unroll.intr = 1;
539                         added_isr_f = 1;

717                 if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, 0,
718                     "mrsas-enable-ctio", &data) == DDI_SUCCESS) {
719                         if (strncmp(data, "no", 3) == 0) {
720                                 ctio_enable = 0;
721                                 con_log(CL_ANN1, (CE_WARN,
722                                     "ctio_enable = %d disabled", ctio_enable));
546                                         "ctio_enable = %d disabled",
547                                         ctio_enable));
723                         }
724                         ddi_prop_free(data);
725                 }

727                 con_log(CL_DLEVEL1, (CE_WARN, "ctio_enable = %d", ctio_enable));
552                         con_log(CL_DLEVEL1, (CE_WARN, "ctio_enable = %d",
553                             ctio_enable));

729                 /* setup the mfi based low level driver */
```

```
730                 if (mrsas_init_adapter(instance) != DDI_SUCCESS) {
731                         cmn_err(CE_WARN, "mr_sas: "
732                             "could not initialize the low level driver");
556                         if (init_mfi(instance) != DDI_SUCCESS) {
557                                 con_log(CL_ANN, (CE_WARN, "mr_sas: "
558                                 "could not initialize the low level driver"));

734                         goto fail_attach;
735                 }

737                 /* Initialize all Mutex */
738                 INIT_LIST_HEAD(&instance->completed_pool_list);
739                 mutex_init(&instance->completed_pool_mtx, NULL,
740                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
565                     mutex_init(&instance->completed_pool_mtx,
566                         "completed_pool_mtx", MUTEX_DRIVER,
567                         DDI_INTR_PRI(instance->intr_pri));

742                 mutex_init(&instance->sync_map_mtx, NULL,
743                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
569                     mutex_init(&instance->app_cmd_pool_mtx,
570                         "app_cmd_pool_mtx", MUTEX_DRIVER,
571                         DDI_INTR_PRI(instance->intr_pri));

745                 mutex_init(&instance->app_cmd_pool_mtx, NULL,
573                         mutex_init(&instance->cmd_pend_mtx, "cmd_pend_mtx",
746                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

748                 mutex_init(&instance->config_dev_mtx, NULL,
576                         mutex_init(&instance->ocr_flags_mtx, "ocr_flags_mtx",
749                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

751                 mutex_init(&instance->cmd_pend_mtx, NULL,
579                         mutex_init(&instance->int_cmd_mtx, "int_cmd_mtx",
752                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

754                 mutex_init(&instance->ocr_flags_mtx, NULL,
755                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

757                 mutex_init(&instance->int_cmd_mtx, NULL,
758                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));
759                 cv_init(&instance->int_cmd_cv, NULL, CV_DRIVER, NULL);

761                 mutex_init(&instance->cmd_pool_mtx, NULL,
583                         mutex_init(&instance->cmd_pool_mtx, "cmd_pool_mtx",
762                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

764                 mutex_init(&instance->reg_write_mtx, NULL,
765                     MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

767                 if (instance->tbolt) {
768                         mutex_init(&instance->cmd_app_pool_mtx, NULL,
769                             MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

771                         mutex_init(&instance->chip_mtx, NULL,
772                             MUTEX_DRIVER, DDI_INTR_PRI(instance->intr_pri));

774                 }

776                 instance->unroll.mutexs = 1;

778                 instance->timeout_id = (timeout_id_t)-1;

780                 /* Register our soft-isr for highlevel interrupts. */
781                 instance->isr_level = instance->intr_pri;
782                 if (!(instance->tbolt)) {
```



```
783                         if (instance->isr_level == HIGH_LEVEL_INTR) {
784                                 if (ddi_add_softintr(dip,
785                                     DDI_SOFTINT_HIGH,
591                                 if (ddi_add_softintr(dip, DDI_SOFTINT_HIGH,
786                                     &instance->soft_intr_id, NULL, NULL,
787                                     mrsas_softintr, (caddr_t)instance) !=
788                                     DDI_SUCCESS) {
789                                         cmn_err(CE_WARN,
790                                             "Software ISR did not register");
595                                         con_log(CL_ANN, (CE_WARN,
596                                             " Software ISR did not register"));

792                                         goto fail_attach;
793                                 }

795                                 instance->unroll.soft_isr = 1;

601                                 added_soft_isr_f = 1;
797                         }
798                 }

800                 instance->softint_running = 0;

802                 /* Allocate a transport structure */
803                 tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

805                 if (tran == NULL) {
806                         cmn_err(CE_WARN,
807                             "scsi_hba_tran_alloc failed");
608                                 con_log(CL_ANN, (CE_WARN,
609                                     "scsi_hba_tran_alloc failed"));
808                         goto fail_attach;
809                 }

613                         tran_alloc_f = 1;

811                 instance->tran = tran;
812                 instance->unroll.tran = 1;

814                 tran->tran_hba_private  = instance;
815                 tran->tran_tgt_init     = mrsas_tran_tgt_init;
816                 tran->tran_tgt_probe    = scsi_hba_probe;
817                 tran->tran_tgt_free     = mrsas_tran_tgt_free;
818                 if (instance->tbolt) {
819                         tran->tran_init_pkt     =
820                             mrsas_tbolt_tran_init_pkt;
821                         tran->tran_start        =
822                             mrsas_tbolt_tran_start;
823                 } else {
824                         tran->tran_init_pkt     = mrsas_tran_init_pkt;
825                         tran->tran_start        = mrsas_tran_start;
826                 }
827                 tran->tran_abort        = mrsas_tran_abort;
828                 tran->tran_reset        = mrsas_tran_reset;
829                 tran->tran_getcap       = mrsas_tran_getcap;
830                 tran->tran_setcap       = mrsas_tran_setcap;
831                 tran->tran_destroy_pkt  = mrsas_tran_destroy_pkt;
832                 tran->tran_dmafree      = mrsas_tran_dmafree;
833                 tran->tran_sync_pkt     = mrsas_tran_sync_pkt;
834                 tran->tran_quiesce      = mrsas_tran_quiesce;
835                 tran->tran_unquiesce    = mrsas_tran_unquiesce;
836                 tran->tran_bus_config   = mrsas_tran_bus_config;

838                 if (mrsas_relaxed_ordering)
839                         mrsas_generic_dma_attr.dma_attr_flags |=
840                             DDI_DMA_RELAXED_ORDERING;
```

```
843                 tran_dma_attr = mrsas_generic_dma_attr;
844                 tran_dma_attr.dma_attr_sgllen = instance->max_num_sge;
845
846                 /* Attach this instance of the hba */
847                 if (scsi_hba_attach_setup(dip, &tran_dma_attr, tran, 0)
848                     != DDI_SUCCESS) {
849                         cmn_err(CE_WARN,
850                             "scsi_hba_attach failed");
643                             con_log(CL_ANN, (CE_WARN,
644                                 "scsi_hba_attach failed"));
645
852                         goto fail_attach;
853                 }
854                 instance->unroll.tranSetup = 1;
855                 con_log(CL_ANN1,
856                     (CE_CONT, "scsi_hba_attach_setup()  done."));
857
858                 /* create devctl node for cfgadm command */
859                 if (ddi_create_minor_node(dip, "devctl",
860                     S_IFCHR, INST2DEVCTL(instance_no),
861                     DDI_NT_SCSI_NEXUS, 0) == DDI_FAILURE) {
862                         cmn_err(CE_WARN,
863                             "mr_sas: failed to create devctl node.");
653                             con_log(CL_ANN, (CE_WARN,
654                                 "mr_sas: failed to create devctl node."));
655
865                         goto fail_attach;
866                 }
867
868                 instance->unroll.devctl = 1;
659                         create_devctl_node_f = 1;
660
870                 /* create scsi node for cfgadm command */
871                 if (ddi_create_minor_node(dip, "scsi", S_IFCHR,
872                     INST2SCSI(instance_no), DDI_NT_SCSI_ATTACHMENT_POINT, 0) ==
663                         INST2SCSI(instance_no),
664                         DDI_NT_SCSI_ATTACHMENT_POINT, 0) ==
873                     DDI_FAILURE) {
874                         cmn_err(CE_WARN,
875                             "mr_sas: failed to create scsi node.");
666                             con_log(CL_ANN, (CE_WARN,
667                                 "mr_sas: failed to create scsi node."));
668
877                         goto fail_attach;
878                 }
879
880                 instance->unroll.scsictl = 1;
672                         create_scsi_node_f = 1;
673
882                 (void) sprintf(instance->iocnode, "%d:lsirdctl",
883                     instance_no);
884
885                 /*
886                  * Create a node for applications
887                  * for issuing ioctl to the driver.
888                  */
889                 if (ddi_create_minor_node(dip, instance->iocnode,
890                     S_IFCHR, INST2LSIRDCTL(instance_no), DDI_PSEUDO, 0) ==
891                     DDI_FAILURE) {
892                         cmn_err(CE_WARN,
893                             "mr_sas: failed to create ioctl node.");
682                             S_IFCHR, INST2LSIRDCTL(instance_no),
683                             DDI_PSEUDO, 0) == DDI_FAILURE) {
684                                 con_log(CL_ANN, (CE_WARN,
```



```
685                                     "mr_sas: failed to create ioctl node."));
686
895                         goto fail_attach;
896                 }
897
898                 instance->unroll.ioctl = 1;
690                         create_ioc_node_f = 1;
691
900                 /* Create a taskq to handle dr events */
901                 if ((instance->taskq = ddi_taskq_create(dip,
902                     "mrsas_dr_taskq", 1, TASKQ_DEFAULTPRI, 0)) == NULL) {
903                         cmn_err(CE_WARN,
904                             "mr_sas: failed to create taskq ");
694                             "mrsas_dr_taskq", 1,
695                             TASKQ_DEFAULTPRI, 0)) == NULL) {
696                                 con_log(CL_ANN, (CE_WARN,
697                                     "mr_sas: failed to create taskq "));
905                         instance->taskq = NULL;
906                         goto fail_attach;
907                 }
908                 instance->unroll.taskq = 1;
909                 con_log(CL_ANN1, (CE_CONT, "ddi_taskq_create()  done."));
910
911                 /* enable interrupt */
912                 instance->func_ptr->enable_intr(instance);
913
914                 /* initiate AEN */
915                 if (start_mfi_aen(instance)) {
916                         cmn_err(CE_WARN,
917                             "mr_sas: failed to initiate AEN.");
918                         goto fail_attach;
707                                 con_log(CL_ANN, (CE_WARN,
708                                     "mr_sas: failed to initiate AEN."));
709                                 goto fail_initiate_aen;
919                 }
920                 instance->unroll.aenPend = 1;
921                 con_log(CL_ANN1,
922                     (CE_CONT, "AEN started for instance %d.", instance_no));
923
712                         con_log(CL_DLEVEL1, (CE_NOTE,
713                             "AEN started for instance %d.", instance_no));
714
924                 /* Finally! We are on the air.  */
925                 ddi_report_dev(dip);
926
927                 /* FMA handle checking. */
928                 if (mrsas_check_acc_handle(instance->regmap_handle) !=
929                     DDI_SUCCESS) {
930                         goto fail_attach;
931                 }
932                 if (mrsas_check_acc_handle(instance->pci_handle) !=
933                     DDI_SUCCESS) {
934                         goto fail_attach;
935                 }
936
937                 instance->mr_ld_list =
938                     kmem_zalloc(MRDRV_MAX_LD * sizeof (struct mrsas_ld),
939                     KM_SLEEP);
940                 instance->unroll.ldlist_buff = 1;
941
942 #ifdef PDSUPPORT
943                 if (instance->tbolt) {
944                         instance->mr_tbolt_pd_max = MRSAS_TBOLT_PD_TGT_MAX;
945                         instance->mr_tbolt_pd_list =
946                             kmem_zalloc(MRSAS_TBOLT_GET_PD_MAX(instance) *
947                             sizeof (struct mrsas_tbolt_pd), KM_SLEEP);
```

```
948                         ASSERT(instance->mr_tbolt_pd_list);
949                         for (i = 0; i < instance->mr_tbolt_pd_max; i++) {
950                                 instance->mr_tbolt_pd_list[i].lun_type =
951                                     MRSAS_TBOLT_PD_LUN;
952                                 instance->mr_tbolt_pd_list[i].dev_id =
953                                     (uint8_t)i;
954                         }

956                         instance->unroll.pdlist_buff = 1;
957                 }
958 #endif
959                 break;
960         case DDI_PM_RESUME:
961                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_PM_RESUME"));
731                         con_log(CL_ANN, (CE_NOTE,
732                             "mr_sas: DDI_PM_RESUME"));
962                 break;
963         case DDI_RESUME:
964                 con_log(CL_ANN, (CE_NOTE, "mr_sas: DDI_RESUME"));
735                         con_log(CL_ANN, (CE_NOTE,
736                             "mr_sas: DDI_RESUME"));
965                 break;
966         default:
967                 con_log(CL_ANN,
968                     (CE_WARN, "mr_sas: invalid attach cmd=%x", cmd));
739                         con_log(CL_ANN, (CE_WARN,
740                             "mr_sas: invalid attach cmd=%x", cmd));
969                 return (DDI_FAILURE);
970         }


973         con_log(CL_DLEVEL1,
974             (CE_NOTE, "mrsas_attach() return SUCCESS instance_num %d",
975             instance_no));
976         return (DDI_SUCCESS);

746 fail_initiate_aen:
978 fail_attach:
748         if (create_devctl_node_f) {
749                 ddi_remove_minor_node(dip, "devctl");
750         }

980         mrsas_undo_resources(dip, instance);
752         if (create_scsi_node_f) {
753                 ddi_remove_minor_node(dip, "scsi");
754         }

756         if (create_ioc_node_f) {
757                 ddi_remove_minor_node(dip, instance->iocnode);
758         }

760         if (tran_alloc_f) {
761                 scsi_hba_tran_free(tran);
762         }


765         if (added_soft_isr_f) {
766                 ddi_remove_softintr(instance->soft_intr_id);
767         }

769         if (added_isr_f) {
770                 mrsas_rem_intrs(instance);
771         }

773         if (instance && instance->taskq) {
774                 ddi_taskq_destroy(instance->taskq);
```



```
775         }

982         mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
983         ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

985         mrsas_fm_fini(instance);

987         pci_config_teardown(&instance->pci_handle);

988         ddi_soft_state_free(mrsas_state, instance_no);

990         con_log(CL_ANN, (CE_WARN, "mr_sas: return failure from mrsas_attach"));
786         con_log(CL_ANN, (CE_NOTE,
787             "mr_sas: return failure from mrsas_attach"));

992         cmn_err(CE_WARN, "mrsas_attach() return FAILURE instance_num %d",
993             instance_no);

995         return (DDI_FAILURE);
996 }

998 /*
999  * getinfo - gets device information
1000 * @dip:
1001 * @cmd:
1002 * @arg:
1003 * @resultp:
1004 *
1005 * The system calls getinfo() to obtain configuration information that only
1006 * the driver knows. The mapping of minor numbers to device instance is
1007 * entirely under the control of the driver. The system sometimes needs to ask
1008 * the driver which device a particular dev_t represents.
1009 * Given the device number return the devinfo pointer from the scsi_device
1010 * structure.
1011 */
1012 /*ARGSUSED*/
1013 static int
1014 mrsas_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd,  void *arg, void **resultp)
1015 {
1016         int     rval;
1017         int     mrsas_minor = getminor((dev_t)arg);

1019         struct mrsas_instance   *instance;

1021         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1023         switch (cmd) {
1024                 case DDI_INFO_DEVT2DEVINFO:
1025                         instance = (struct mrsas_instance *)
1026                             ddi_get_soft_state(mrsas_state,
1027                             MINOR2INST(mrsas_minor));

1029                         if (instance == NULL) {
1030                                 *resultp = NULL;
1031                                 rval = DDI_FAILURE;
1032                         } else {
1033                                 *resultp = instance->dip;
1034                                 rval = DDI_SUCCESS;
1035                         }
1036                         break;
1037                 case DDI_INFO_DEVT2INSTANCE:
1038                         *resultp = (void *)(intptr_t)
1039                             (MINOR2INST(getminor((dev_t)arg)));
1040                         rval = DDI_SUCCESS;
1041                         break;
1042                 default:
```

```
1043                        *resultp = NULL;
1044                        rval = DDI_FAILURE;
1045        }

1047        return (rval);
1048 }

1050 /*
1051  * detach - detaches a device from the system
1052  * @dip: pointer to the device's dev_info structure
1053  * @cmd: type of detach
1054  *
1055  * A driver's detach() entry point is called to detach an instance of a device
1056  * that is bound to the driver. The entry point is called with the instance of
1057  * the device node to be detached and with DDI_DETACH, which is specified as
1058  * the cmd argument to the entry point.
1059  * This routine is called during driver unload. We free all the allocated
1060  * resources and call the corresponding LLD so that it can also release all
1061  * its resources.
1062  */
1063 static int
1064 mrsas_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
1065 {
1066        int     instance_no;

1068        struct mrsas_instance   *instance;

1070        con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
 837        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


1073        /* CONSTCOND */
1074        ASSERT(NO_COMPETING_THREADS);

1076        instance_no = ddi_get_instance(dip);

1078        instance = (struct mrsas_instance *)ddi_get_soft_state(mrsas_state,
1079            instance_no);

1081        if (!instance) {
1082                cmn_err(CE_WARN,
 848                con_log(CL_ANN, (CE_WARN,
1083                    "mr_sas:%d could not get instance in detach",
1084                    instance_no);
 850                    instance_no));

1086                return (DDI_FAILURE);
1087        }

1089        con_log(CL_ANN, (CE_NOTE,
1090            "mr_sas%d: detaching device 0x%4x:0x%4x:0x%4x:0x%4x",
1091            instance_no, instance->vendor_id, instance->device_id,
1092            instance->subsysvid, instance->subsysid));

1094        switch (cmd) {
1095                case DDI_DETACH:
1096                        con_log(CL_ANN, (CE_NOTE,
1097                            "mrsas_detach: DDI_DETACH"));

1099                        mutex_enter(&instance->config_dev_mtx);
1100                        if (instance->timeout_id != (timeout_id_t)-1) {
1101                                mutex_exit(&instance->config_dev_mtx);
1102                                (void) untimeout(instance->timeout_id);
1103                                instance->timeout_id = (timeout_id_t)-1;
1104                                mutex_enter(&instance->config_dev_mtx);
1105                                instance->unroll.timer = 0;
```



```
 865                if (scsi_hba_detach(dip) != DDI_SUCCESS) {
 866                        con_log(CL_ANN, (CE_WARN,
 867                            "mr_sas:%d failed to detach",
 868                            instance_no));

 870                        return (DDI_FAILURE);
1106                        }
1107                        mutex_exit(&instance->config_dev_mtx);

1109                        if (instance->unroll.tranSetup == 1) {
1110                                if (scsi_hba_detach(dip) != DDI_SUCCESS) {
1111                                        cmn_err(CE_WARN,
1112                                            "mr_sas2%d: failed to detach",
1113                                            instance_no);
 873                scsi_hba_tran_free(instance->tran);

 875                flush_cache(instance);

 877                if (abort_aen_cmd(instance, instance->aen_cmd)) {
 878                        con_log(CL_ANN, (CE_WARN, "mrsas_detach: "
 879                            "failed to abort prevous AEN command"));

1114                                        return (DDI_FAILURE);
1115                                }
1116                                instance->unroll.tranSetup = 0;
1117                                con_log(CL_ANN1,
1118                                    (CE_CONT, "scsi_hba_dettach()  done."));

 884                instance->func_ptr->disable_intr(instance);

 886                if (instance->isr_level == HIGH_LEVEL_INTR) {
 887                        ddi_remove_softintr(instance->soft_intr_id);
1119                        }

1121                        flush_cache(instance);
 890                mrsas_rem_intrs(instance);

1123                        mrsas_undo_resources(dip, instance);
 892                if (instance->taskq) {
 893                        ddi_taskq_destroy(instance->taskq);
 894                }
 895                kmem_free(instance->mr_ld_list, MRDRV_MAX_LD
 896                    * sizeof (struct mrsas_ld));
 897                free_space_for_mfi(instance);

1125                        mrsas_fm_fini(instance);

1127                        pci_config_teardown(&instance->pci_handle);

 903                kmem_free(instance->func_ptr,
 904                    sizeof (struct mrsas_func_ptr));

 906                if (instance->timeout_id != (timeout_id_t)-1) {
 907                        (void) untimeout(instance->timeout_id);
 908                        instance->timeout_id = (timeout_id_t)-1;
 909                }
1128                        ddi_soft_state_free(mrsas_state, instance_no);
1129                        break;

1131                case DDI_PM_SUSPEND:
1132                        con_log(CL_ANN, (CE_NOTE,
1133                            "mrsas_detach: DDI_PM_SUSPEND"));

1135                        break;
1136                case DDI_SUSPEND:
1137                        con_log(CL_ANN, (CE_NOTE,
```

```
1138                            "mrsas_detach: DDI_SUSPEND"));

1140                    break;
1141            default:
1142                    con_log(CL_ANN, (CE_WARN,
1143                        "invalid detach command:0x%x", cmd));
1144                    return (DDI_FAILURE);
1145        }

1147        return (DDI_SUCCESS);
1148 }


1151 static void
1152 mrsas_undo_resources(dev_info_t *dip, struct mrsas_instance *instance)
1153 {
1154        int     instance_no;

1156        con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));


1159        instance_no = ddi_get_instance(dip);


1162        if (instance->unroll.ioctl == 1) {
1163                ddi_remove_minor_node(dip, instance->iocnode);
1164                instance->unroll.ioctl = 0;
1165        }

1167        if (instance->unroll.scsictl == 1) {
1168                ddi_remove_minor_node(dip, "scsi");
1169                instance->unroll.scsictl = 0;
1170        }

1172        if (instance->unroll.devctl == 1) {
1173                ddi_remove_minor_node(dip, "devctl");
1174                instance->unroll.devctl = 0;
1175        }

1177        if (instance->unroll.tranSetup == 1) {
1178                if (scsi_hba_detach(dip) != DDI_SUCCESS) {
1179                        cmn_err(CE_WARN,
1180                            "mr_sas2%d: failed to detach", instance_no);
1181                        return;  /* DDI_FAILURE */
1182                }
1183                instance->unroll.tranSetup = 0;
1184                con_log(CL_ANN1, (CE_CONT, "scsi_hba_dettach()  done."));
1185        }

1187        if (instance->unroll.tran == 1)  {
1188                scsi_hba_tran_free(instance->tran);
1189                instance->unroll.tran = 0;
1190                con_log(CL_ANN1, (CE_CONT, "scsi_hba_tran_free()  done."));
1191        }

1193        if (instance->unroll.syncCmd == 1) {
1194                if (instance->tbolt) {
1195                        if (abort_syncmap_cmd(instance,
1196                            instance->map_update_cmd)) {
1197                                cmn_err(CE_WARN, "mrsas_detach: "
1198                                    "failed to abort previous syncmap command");
1199                        }

1201                        instance->unroll.syncCmd = 0;
1202                        con_log(CL_ANN1, (CE_CONT, "sync cmd aborted, done."));
1203                }
```



```
1204        }

1206        if (instance->unroll.aenPend == 1) {
1207                if (abort_aen_cmd(instance, instance->aen_cmd))
1208                        cmn_err(CE_WARN, "mrsas_detach: "
1209                            "failed to abort prevous AEN command");

1211                instance->unroll.aenPend = 0;
1212                con_log(CL_ANN1, (CE_CONT, "aen cmd aborted, done."));
1213                /* This means the controller is fully initialized and running */
1214                /* Shutdown should be a last command to controller. */
1215                /* shutdown_controller(); */
1216        }


1219        if (instance->unroll.timer == 1)         {
1220                if (instance->timeout_id != (timeout_id_t)-1) {
1221                        (void) untimeout(instance->timeout_id);
1222                        instance->timeout_id = (timeout_id_t)-1;

1224                        instance->unroll.timer = 0;
1225                }
1226        }

1228        instance->func_ptr->disable_intr(instance);


1231        if (instance->unroll.mutexs == 1) {
1232                mutex_destroy(&instance->cmd_pool_mtx);
1233                mutex_destroy(&instance->app_cmd_pool_mtx);
1234                mutex_destroy(&instance->cmd_pend_mtx);
1235                mutex_destroy(&instance->completed_pool_mtx);
1236                mutex_destroy(&instance->sync_map_mtx);
1237                mutex_destroy(&instance->int_cmd_mtx);
1238                cv_destroy(&instance->int_cmd_cv);
1239                mutex_destroy(&instance->config_dev_mtx);
1240                mutex_destroy(&instance->ocr_flags_mtx);
1241                mutex_destroy(&instance->reg_write_mtx);

1243                if (instance->tbolt) {
1244                        mutex_destroy(&instance->cmd_app_pool_mtx);
1245                        mutex_destroy(&instance->chip_mtx);
1246                }

1248                instance->unroll.mutexs = 0;
1249                con_log(CL_ANN1, (CE_CONT, "Destroy mutex & cv,  done."));
1250        }


1253        if (instance->unroll.soft_isr == 1) {
1254                ddi_remove_softintr(instance->soft_intr_id);
1255                instance->unroll.soft_isr = 0;
1256        }

1258        if (instance->unroll.intr == 1) {
1259                mrsas_rem_intrs(instance);
1260                instance->unroll.intr = 0;
1261        }


1264        if (instance->unroll.taskq == 1)         {
1265                if (instance->taskq) {
1266                        ddi_taskq_destroy(instance->taskq);
1267                        instance->unroll.taskq = 0;
1268                }
```

```
1270         }

1272         /*
1273          * free dma memory allocated for
1274          * cmds/frames/queues/driver version etc
1275          */
1276         if (instance->unroll.verBuff == 1) {
1277                 (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);
1278                 instance->unroll.verBuff = 0;
1279         }

1281         if (instance->unroll.pdlist_buff == 1)  {
1282                 if (instance->mr_tbolt_pd_list != NULL) {
1283                         kmem_free(instance->mr_tbolt_pd_list,
1284                             MRSAS_TBOLT_GET_PD_MAX(instance) *
1285                             sizeof (struct mrsas_tbolt_pd));
1286                 }

1288                 instance->mr_tbolt_pd_list = NULL;
1289                 instance->unroll.pdlist_buff = 0;
1290         }

1292         if (instance->unroll.ldlist_buff == 1)  {
1293                 if (instance->mr_ld_list != NULL) {
1294                         kmem_free(instance->mr_ld_list, MRDRV_MAX_LD
1295                             * sizeof (struct mrsas_ld));
1296                 }

1298                 instance->mr_ld_list = NULL;
1299                 instance->unroll.ldlist_buff = 0;
1300         }

1302         if (instance->tbolt) {
1303                 if (instance->unroll.alloc_space_mpi2 == 1) {
1304                         free_space_for_mpi2(instance);
1305                         instance->unroll.alloc_space_mpi2 = 0;
1306                 }
1307         } else {
1308                 if (instance->unroll.alloc_space_mfi == 1) {
1309                         free_space_for_mfi(instance);
1310                         instance->unroll.alloc_space_mfi = 0;
1311                 }
1312         }

1314         if (instance->unroll.regs == 1)  {
1315                 ddi_regs_map_free(&instance->regmap_handle);
1316                 instance->unroll.regs = 0;
1317                 con_log(CL_ANN1, (CE_CONT, "ddi_regs_map_free()  done."));
1318         }
1319 }



1323 /*
1324  * ************************************************************************** *
1325  *                                                                            *
1326  *             common entry points - for character driver types               *
1327  *                                                                            *
1328  * ************************************************************************** *
1329  */
1330 /*
1331  * open - gets access to a device
1332  * @dev:
1333  * @openflags:
1334  * @otyp:
1335  * @credp:
```



```
1336  *
1337  * Access to a device by one or more application programs is controlled
1338  * through the open() and close() entry points. The primary function of
1339  * open() is to verify that the open request is allowed.
1340  */
1341 static  int
1342 mrsas_open(dev_t *dev, int openflags, int otyp, cred_t *credp)
1343 {
1344         int     rval = 0;

1346         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1348         /* Check root permissions */
1349         if (drv_priv(credp) != 0) {
1350                 con_log(CL_ANN, (CE_WARN,
1351                     "mr_sas: Non-root ioctl access denied!"));
1352                 return (EPERM);
1353         }

1355         /* Verify we are being opened as a character device */
1356         if (otyp != OTYP_CHR) {
1357                 con_log(CL_ANN, (CE_WARN,
1358                     "mr_sas: ioctl node must be a char node"));
1359                 return (EINVAL);
1360         }

1362         if (ddi_get_soft_state(mrsas_state, MINOR2INST(getminor(*dev)))
1363             == NULL) {
1364                 return (ENXIO);
1365         }

1367         if (scsi_hba_open) {
1368                 rval = scsi_hba_open(dev, openflags, otyp, credp);
1369         }

1371         return (rval);
1372 }

1374 /*
1375  * close - gives up access to a device
1376  * @dev:
1377  * @openflags:
1378  * @otyp:
1379  * @credp:
1380  *
1381  * close() should perform any cleanup necessary to finish using the minor
1382  * device, and prepare the device (and driver) to be opened again.
1383  */
1384 static  int
1385 mrsas_close(dev_t dev, int openflags, int otyp, cred_t *credp)
1386 {
1387         int     rval = 0;

1389         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1391         /* no need for locks! */

1393         if (scsi_hba_close) {
1394                 rval = scsi_hba_close(dev, openflags, otyp, credp);
1395         }

1397         return (rval);
1398 }

1400 /*
1401  * ioctl - performs a range of I/O commands for character drivers
```

```
1402  * @dev:
1403  * @cmd:
1404  * @arg:
1405  * @mode:
1406  * @credp:
1407  * @rvalp:
1408  *
1409  * ioctl() routine must make sure that user data is copied into or out of the
1410  * kernel address space explicitly using copyin(), copyout(), ddi_copyin(),
1411  * and ddi_copyout(), as appropriate.
1412  * This is a wrapper routine to serialize access to the actual ioctl routine.
1413  * ioctl() should return 0 on success, or the appropriate error number. The
1414  * driver may also set the value returned to the calling process through rvalp.
1415  */

1417 static int
1418 mrsas_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp,
1419     int *rvalp)
1420 {
1421         int     rval = 0;

1423         struct mrsas_instance   *instance;
1424         struct mrsas_ioctl      *ioctl;
1425         struct mrsas_aen        aen;
1426         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1428         instance = ddi_get_soft_state(mrsas_state, MINOR2INST(getminor(dev)));

1430         if (instance == NULL) {
1431                 /* invalid minor number */
1432                 con_log(CL_ANN, (CE_WARN, "mr_sas: adapter not found."));
1433                 return (ENXIO);
1434         }

1436         ioctl = (struct mrsas_ioctl *)kmem_zalloc(sizeof (struct mrsas_ioctl),
1437             KM_SLEEP);
1438         ASSERT(ioctl);

1440         switch ((uint_t)cmd) {
1441                 case MRSAS_IOCTL_FIRMWARE:
1442                         if (ddi_copyin((void *)arg, ioctl,
1443                             sizeof (struct mrsas_ioctl), mode)) {
1444                                 con_log(CL_ANN, (CE_WARN, "mrsas_ioctl: "
1445                                     "ERROR IOCTL copyin"));
1446                                 kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1447                                 return (EFAULT);
1448                         }

1450                         if (ioctl->control_code == MRSAS_DRIVER_IOCTL_COMMON) {
1451                                 rval = handle_drv_ioctl(instance, ioctl, mode);
1452                         } else {
1453                                 rval = handle_mfi_ioctl(instance, ioctl, mode);
1454                         }

1456                         if (ddi_copyout((void *)ioctl, (void *)arg,
1457                             (sizeof (struct mrsas_ioctl) - 1), mode)) {
1458                                 con_log(CL_ANN, (CE_WARN,
1459                                     "mrsas_ioctl: copy_to_user failed"));
1460                                 rval = 1;
1461                         }

1463                         break;
1464                 case MRSAS_IOCTL_AEN:
1465                         if (ddi_copyin((void *) arg, &aen,
1466                             sizeof (struct mrsas_aen), mode)) {
1467                                 con_log(CL_ANN, (CE_WARN,
```



```
1468                                     "mrsas_ioctl: ERROR AEN copyin"));
1469                                 kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1470                                 return (EFAULT);
1471                         }

1473                         rval = handle_mfi_aen(instance, &aen);

1475                         if (ddi_copyout((void *) &aen, (void *)arg,
1476                             sizeof (struct mrsas_aen), mode)) {
1477                                 con_log(CL_ANN, (CE_WARN,
1478                                     "mrsas_ioctl: copy_to_user failed"));
1479                                 rval = 1;
1480                         }

1482                         break;
1483                 default:
1484                         rval = scsi_hba_ioctl(dev, cmd, arg,
1485                             mode, credp, rvalp);

1487                         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_ioctl: "
1488                             "scsi_hba_ioctl called, ret = %x.", rval));
1489         }

1491         kmem_free(ioctl, sizeof (struct mrsas_ioctl));
1492         return (rval);
1493 }

1495 /*
1496  * ************************************************************************** *
1497  *                                                                            *
1498  *               common entry points - for block driver types                 *
1499  *                                                                            *
1500  * ************************************************************************** *
1501  */
1502 #ifdef __sparc
1503 /*
1504  * reset - TBD
1505  * @dip:
1506  * @cmd:
1507  *
1508  * TBD
1509  */
1510 /*ARGSUSED*/
1511 static int
1512 mrsas_reset(dev_info_t *dip, ddi_reset_cmd_t cmd)
1513 {
1514         int     instance_no;

1516         struct mrsas_instance   *instance;

1518         instance_no = ddi_get_instance(dip);
1519         instance = (struct mrsas_instance *)ddi_get_soft_state
1520             (mrsas_state, instance_no);

1522         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1524         if (!instance) {
1525                 con_log(CL_ANN, (CE_WARN, "mr_sas:%d could not get adapter "
1526                     "in reset", instance_no));
1527                 return (DDI_FAILURE);
1528         }

1530         instance->func_ptr->disable_intr(instance);

1532         con_log(CL_ANN1, (CE_CONT, "flushing cache for instance %d",
1095         con_log(CL_ANN1, (CE_NOTE, "flushing cache for instance %d",
```

```
1533             instance_no));

1535         flush_cache(instance);

1537         return (DDI_SUCCESS);
1538 }
1539 #else /* __sparc */
1540 /*ARGSUSED*/
1541 static int
1542 mrsas_quiesce(dev_info_t *dip)
1543 {
1544         int     instance_no;

1546         struct mrsas_instance   *instance;

1548         instance_no = ddi_get_instance(dip);
1549         instance = (struct mrsas_instance *)ddi_get_soft_state
1550             (mrsas_state, instance_no);

1552         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

1554         if (!instance) {
1555                 con_log(CL_ANN1, (CE_WARN, "mr_sas:%d could not get adapter "
1556                     "in quiesce", instance_no));
1557                 return (DDI_FAILURE);
1558         }
1559         if (instance->deadadapter || instance->adapterresetinprogress) {
1560                 con_log(CL_ANN1, (CE_WARN, "mr_sas:%d adapter is not in "
1561                     "healthy state", instance_no));
1562                 return (DDI_FAILURE);
1563         }

1565         if (abort_aen_cmd(instance, instance->aen_cmd)) {
1566                 con_log(CL_ANN1, (CE_WARN, "mrsas_quiesce: "
1567                     "failed to abort prevous AEN command QUIESCE"));
1568         }

1570         if (instance->tbolt) {
1571                 if (abort_syncmap_cmd(instance,
1572                     instance->map_update_cmd)) {
1573                         cmn_err(CE_WARN,
1574                             "mrsas_detach: failed to abort "
1575                             "previous syncmap command");
1576                         return (DDI_FAILURE);
1577                 }
1578         }

1580         instance->func_ptr->disable_intr(instance);

1582         con_log(CL_ANN1, (CE_CONT, "flushing cache for instance %d",
1135         con_log(CL_ANN1, (CE_NOTE, "flushing cache for instance %d",
1583             instance_no));

1585         flush_cache(instance);

1587         if (wait_for_outstanding(instance)) {
1588                 con_log(CL_ANN1,
1589                     (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
1590                 return (DDI_FAILURE);
1591         }
1592         return (DDI_SUCCESS);
1593 }
1594 #endif  /* __sparc */

1596 /*
1597  * ************************************************************************** *
```



```
1598  *                                                                            *
1599  *                         entry points (SCSI HBA)                            *
1600  *                                                                            *
1601  * ************************************************************************** *
1602  */
1603 /*
1604  * tran_tgt_init - initialize a target device instance
1605  * @hba_dip:
1606  * @tgt_dip:
1607  * @tran:
1608  * @sd:
1609  *
1610  * The tran_tgt_init() entry point enables the HBA to allocate and initialize
1611  * any per-target resources. tran_tgt_init() also enables the HBA to qualify
1612  * the device's address as valid and supportable for that particular HBA.
1613  * By returning DDI_FAILURE, the instance of the target driver for that device
1614  * is not probed or attached.
1615  */
1616 /*ARGSUSED*/
1617 static int
1618 mrsas_tran_tgt_init(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1619                 scsi_hba_tran_t *tran, struct scsi_device *sd)
1620 {
1621         struct mrsas_instance *instance;
1622         uint16_t tgt = sd->sd_address.a_target;
1623         uint8_t lun = sd->sd_address.a_lun;
1624         dev_info_t *child = NULL;

1626         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init target %d lun %d",
1163         con_log(CL_ANN1, (CE_NOTE, "mrsas_tgt_init target %d lun %d",
1627             tgt, lun));

1629         instance = ADDR2MR(&sd->sd_address);

1631         if (ndi_dev_is_persistent_node(tgt_dip) == 0) {
1632                 /*
1633                  * If no persistent node exists, we don't allow .conf node
1634                  * to be created.
1635                  */
1636                 if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
1637                         con_log(CL_DLEVEL2,
1638                             (CE_NOTE, "mrsas_tgt_init find child ="
1639                             " %p t = %d l = %d", (void *)child, tgt, lun));
1640                         if (ndi_merge_node(tgt_dip, mrsas_name_node) !=
1641                             DDI_SUCCESS)
1642                                 /* Create this .conf node */
1643                                 return (DDI_SUCCESS);
1644                 }
1645                 con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init in ndi_per "
1646                     "DDI_FAILURE t = %d l = %d", tgt, lun));
1647                 return (DDI_FAILURE);
1169                 (void) ndi_merge_node(tgt_dip, mrsas_name_node);
1170                 ddi_set_name_addr(tgt_dip, NULL);

1172                 con_log(CL_ANN1, (CE_NOTE, "mrsas_tgt_init in "
1173                     "ndi_dev_is_persistent_node DDI_FAILURE t = %d l = %d",
1174                     tgt, lun));
1175                 return (DDI_FAILURE);
1649         }

1651         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_tgt_init dev_dip %p tgt_dip %p",
1178         con_log(CL_ANN1, (CE_NOTE, "mrsas_tgt_init dev_dip %p tgt_dip %p",
1652             (void *)instance->mr_ld_list[tgt].dip, (void *)tgt_dip));

1654         if (tgt < MRDRV_MAX_LD && lun == 0) {
1655                 if (instance->mr_ld_list[tgt].dip == NULL &&
```

```
1656                     strcmp(ddi_driver_name(sd->sd_dev), "sd") == 0) {
1657                         mutex_enter(&instance->config_dev_mtx);
1658                         instance->mr_ld_list[tgt].dip = tgt_dip;
1659                         instance->mr_ld_list[tgt].lun_type = MRSAS_LD_LUN;
1660                         instance->mr_ld_list[tgt].flag = MRDRV_TGT_VALID;
1661                         mutex_exit(&instance->config_dev_mtx);
1662                 }
1663         }

1665 #ifdef PDSUPPORT
1666         else if (instance->tbolt) {
1667                 if (instance->mr_tbolt_pd_list[tgt].dip == NULL) {
1668                         mutex_enter(&instance->config_dev_mtx);
1669                         instance->mr_tbolt_pd_list[tgt].dip = tgt_dip;
1670                         instance->mr_tbolt_pd_list[tgt].flag =
1671                             MRDRV_TGT_VALID;
1672                         mutex_exit(&instance->config_dev_mtx);
1673                         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_tgt_init:"
1674                             "t%xl%x", tgt, lun));
1675                 }
1676         }
1677 #endif

1679         return (DDI_SUCCESS);
1680 }

1682 /*ARGSUSED*/
1683 static void
1684 mrsas_tran_tgt_free(dev_info_t *hba_dip, dev_info_t *tgt_dip,
1685     scsi_hba_tran_t *hba_tran, struct scsi_device *sd)
1686 {
1687         struct mrsas_instance *instance;
1688         int tgt = sd->sd_address.a_target;
1689         int lun = sd->sd_address.a_lun;

1691         instance = ADDR2MR(&sd->sd_address);

1693         con_log(CL_DLEVEL2, (CE_NOTE, "tgt_free t = %d l = %d", tgt, lun));
1202         con_log(CL_ANN1, (CE_NOTE, "tgt_free t = %d l = %d", tgt, lun));

1695         if (tgt < MRDRV_MAX_LD && lun == 0) {
1696                 if (instance->mr_ld_list[tgt].dip == tgt_dip) {
1697                         mutex_enter(&instance->config_dev_mtx);
1698                         instance->mr_ld_list[tgt].dip = NULL;
1699                         mutex_exit(&instance->config_dev_mtx);
1700                 }
1701         }

1703 #ifdef PDSUPPORT
1704         else if (instance->tbolt) {
1705                 mutex_enter(&instance->config_dev_mtx);
1706                 instance->mr_tbolt_pd_list[tgt].dip = NULL;
1707                 mutex_exit(&instance->config_dev_mtx);
1708                 con_log(CL_ANN1, (CE_NOTE, "tgt_free: Setting dip = NULL"
1709                     "for tgt:%x", tgt));
1710         }
1711 #endif

1713 }

1715 dev_info_t *
1211 static dev_info_t *
1716 mrsas_find_child(struct mrsas_instance *instance, uint16_t tgt, uint8_t lun)
1717 {
1718         dev_info_t *child = NULL;
1719         char addr[SCSI_MAXNAMELEN];
```



```
1720         char tmp[MAXNAMELEN];

1722         (void) sprintf(addr, "%x,%x", tgt, lun);
1723         for (child = ddi_get_child(instance->dip); child;
1724             child = ddi_get_next_sibling(child)) {

1726                 if (ndi_dev_is_persistent_node(child) == 0) {
1727                         continue;
1728                 }

1730                 if (mrsas_name_node(child, tmp, MAXNAMELEN) !=
1731                     DDI_SUCCESS) {
1732                         continue;
1733                 }

1735                 if (strcmp(addr, tmp) == 0) {
1736                         break;
1737                 }
1738         }
1739         con_log(CL_DLEVEL2, (CE_NOTE, "mrsas_find_child: return child = %p",
1231         con_log(CL_ANN1, (CE_NOTE, "mrsas_find_child: return child = %p",
1740             (void *)child));
1741         return (child);
1742 }

1744 /*
1745  * mrsas_name_node -
1746  * @dip:
1747  * @name:
1748  * @len:
1749  */
1750 static int
1751 mrsas_name_node(dev_info_t *dip, char *name, int len)
1752 {
1753         int tgt, lun;

1755         tgt = ddi_prop_get_int(DDI_DEV_T_ANY, dip,
1756             DDI_PROP_DONTPASS, "target", -1);
1757         con_log(CL_DLEVEL2, (CE_NOTE,
1243         con_log(CL_ANN1, (CE_NOTE,
1758             "mrsas_name_node: dip %p tgt %d", (void *)dip, tgt));
1759         if (tgt == -1) {
1760                 return (DDI_FAILURE);
1761         }
1762         lun = ddi_prop_get_int(DDI_DEV_T_ANY, dip, DDI_PROP_DONTPASS,
1763             "lun", -1);
1764         con_log(CL_DLEVEL2,
1250         con_log(CL_ANN1,
1765             (CE_NOTE, "mrsas_name_node: tgt %d lun %d", tgt, lun));
1766         if (lun == -1) {
1767                 return (DDI_FAILURE);
1768         }
1769         (void) snprintf(name, len, "%x,%x", tgt, lun);
1770         return (DDI_SUCCESS);
1771 }

1773 /*
1774  * tran_init_pkt - allocate & initialize a scsi_pkt structure
1775  * @ap:
1776  * @pkt:
1777  * @bp:
1778  * @cmdlen:
1779  * @statuslen:
1780  * @tgtlen:
1781  * @flags:
1782  * @callback:
```

```
 *
 * The tran_init_pkt() entry point allocates and initializes a scsi_pkt
 * structure and DMA resources for a target driver request. The
 * tran_init_pkt() entry point is called when the target driver calls the
 * SCSA function scsi_init_pkt(). Each call of the tran_init_pkt() entry point
 * is a request to perform one or more of three possible services:
 *  - allocation and initialization of a scsi_pkt structure
 *  - allocation of DMA resources for data transfer
 *  - reallocation of DMA resources for the next portion of the data transfer
 */
static struct scsi_pkt *
mrsas_tran_init_pkt(struct scsi_address *ap, register struct scsi_pkt *pkt,
        struct buf *bp, int cmdlen, int statuslen, int tgtlen,
        int flags, int (*callback)(), caddr_t arg)
{
        struct scsa_cmd *acmd;
        struct mrsas_instance   *instance;
        struct scsi_pkt *new_pkt;

        con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

        instance = ADDR2MR(ap);

        /* step #1 : pkt allocation */
        if (pkt == NULL) {
                pkt = scsi_hba_pkt_alloc(instance->dip, ap, cmdlen, statuslen,
                    tgtlen, sizeof (struct scsa_cmd), callback, arg);
                if (pkt == NULL) {
                        return (NULL);
                }

                acmd = PKT2CMD(pkt);

                /*
                 * Initialize the new pkt - we redundantly initialize
                 * all the fields for illustrative purposes.
                 */
                acmd->cmd_pkt           = pkt;
                acmd->cmd_flags         = 0;
                acmd->cmd_scblen        = statuslen;
                acmd->cmd_cdblen        = cmdlen;
                acmd->cmd_dmahandle     = NULL;
                acmd->cmd_ncookies      = 0;
                acmd->cmd_cookie        = 0;
                acmd->cmd_cookiecnt     = 0;
                acmd->cmd_nwin          = 0;

                pkt->pkt_address        = *ap;
                pkt->pkt_comp           = (void (*)())NULL;
                pkt->pkt_flags          = 0;
                pkt->pkt_time           = 0;
                pkt->pkt_resid          = 0;
                pkt->pkt_state          = 0;
                pkt->pkt_statistics     = 0;
                pkt->pkt_reason         = 0;
                new_pkt                 = pkt;
        } else {
                acmd = PKT2CMD(pkt);
                new_pkt = NULL;
        }

        /* step #2 : dma allocation/move */
        if (bp && bp->b_bcount != 0) {
                if (acmd->cmd_dmahandle == NULL) {
                        if (mrsas_dma_alloc(instance, pkt, bp, flags,
```



```
                            callback) == DDI_FAILURE) {
                                if (new_pkt) {
                                        scsi_hba_pkt_free(ap, new_pkt);
                                }
                                return ((struct scsi_pkt *)NULL);
                        }
                } else {
                        if (mrsas_dma_move(instance, pkt, bp) == DDI_FAILURE) {
                                return ((struct scsi_pkt *)NULL);
                        }
                }
        }

        return (pkt);
}

/*
 * tran_start - transport a SCSI command to the addressed target
 * @ap:
 * @pkt:
 *
 * The tran_start() entry point for a SCSI HBA driver is called to transport a
 * SCSI command to the addressed target. The SCSI command is described
 * entirely within the scsi_pkt structure, which the target driver allocated
 * through the HBA driver's tran_init_pkt() entry point. If the command
 * involves a data transfer, DMA resources must also have been allocated for
 * the scsi_pkt structure.
 *
 * Return Values :
 *      TRAN_BUSY - request queue is full, no more free scbs
 *      TRAN_ACCEPT - pkt has been submitted to the instance
 */
static int
mrsas_tran_start(struct scsi_address *ap, register struct scsi_pkt *pkt)
{
        uchar_t         cmd_done = 0;

        struct mrsas_instance   *instance = ADDR2MR(ap);
        struct mrsas_cmd        *cmd;

        con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
        if (instance->deadadapter == 1) {
                con_log(CL_ANN1, (CE_WARN,
                    "mrsas_tran_start: return TRAN_FATAL_ERROR "
                    "for IO, as the HBA doesnt take any more IOs"));
                if (pkt) {
                        pkt->pkt_reason         = CMD_DEV_GONE;
                        pkt->pkt_statistics     = STAT_DISCON;
                }
                return (TRAN_FATAL_ERROR);
        }

        if (instance->adapterresetinprogress) {
                con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_start: Reset flag set, "
                con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
                    "returning mfi_pkt and setting TRAN_BUSY\n"));
                return (TRAN_BUSY);
        }

        con_log(CL_ANN1, (CE_CONT, "chkpnt:%s:%d:SCSI CDB[0]=0x%x time:%x",
        con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d:SCSI CDB[0]=0x%x time:%x",
            __func__, __LINE__, pkt->pkt_cdbp[0], pkt->pkt_time));

        pkt->pkt_reason = CMD_CMPLT;
        *pkt->pkt_scbp = STATUS_GOOD; /* clear arq scsi_status */
```

```
1912         cmd = build_cmd(instance, ap, pkt, &cmd_done);

1914         /*
1915          * Check if the command is already completed by the mrsas_build_cmd()
1916          * routine. In which case the busy_flag would be clear and scb will be
1917          * NULL and appropriate reason provided in pkt_reason field
1918          */
1919         if (cmd_done) {
1920                 pkt->pkt_reason = CMD_CMPLT;
1921                 pkt->pkt_scbp[0] = STATUS_GOOD;
1922                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1923                     | STATE_SENT_CMD;
1924                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1925                         (*pkt->pkt_comp)(pkt);
1926                 }

1928                 return (TRAN_ACCEPT);
1929         }

1931         if (cmd == NULL) {
1932                 return (TRAN_BUSY);
1933         }

1935         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1936                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1937                         con_log(CL_ANN, (CE_CONT, "mr_sas:Firmware busy"));
1938                         DTRACE_PROBE2(start_tran_err,
1939                             uint16_t, instance->fw_outstanding,
1940                             uint16_t, instance->max_fw_cmds);
1941                         return_mfi_pkt(instance, cmd);
1942                         return (TRAN_BUSY);
1943                 }

1945                 /* Synchronize the Cmd frame for the controller */
1946                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1947                     DDI_DMA_SYNC_FORDEV);
1948                 con_log(CL_ANN, (CE_CONT, "issue_cmd_ppc: SCSI CDB[0]=0x%x"
1397                 con_log(CL_ANN1, (CE_NOTE, "Push SCSI CDB[0]=0x%x"
1949                     "cmd->index:%x\n", pkt->pkt_cdbp[0], cmd->index));
1950                 instance->func_ptr->issue_cmd(cmd, instance);

1952         } else {
1953                 struct mrsas_header *hdr = &cmd->frame->hdr;

1955                 instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);
1404                 cmd->sync_cmd = MRSAS_TRUE;

1406                 instance->func_ptr-> issue_cmd_in_poll_mode(instance, cmd);

1957                 pkt->pkt_reason         = CMD_CMPLT;
1958                 pkt->pkt_statistics     = 0;
1959                 pkt->pkt_state |= STATE_XFERRED_DATA | STATE_GOT_STATUS;

1961                 switch (ddi_get8(cmd->frame_dma_obj.acc_handle,
1962                     &hdr->cmd_status)) {
1963                 case MFI_STAT_OK:
1964                         pkt->pkt_scbp[0] = STATUS_GOOD;
1965                         break;

1967                 case MFI_STAT_SCSI_DONE_WITH_ERROR:
1968                         con_log(CL_ANN, (CE_CONT,
1969                             "mrsas_tran_start: scsi done with error"));

1970                         pkt->pkt_reason = CMD_CMPLT;
1971                         pkt->pkt_statistics = 0;
```



```
1973                         ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;
1974                         break;

1976                 case MFI_STAT_DEVICE_NOT_FOUND:
1977                         con_log(CL_ANN, (CE_CONT,
1978                             "mrsas_tran_start: device not found error"));
1979                         pkt->pkt_reason         = CMD_DEV_GONE;
1980                         pkt->pkt_statistics     = STAT_DISCON;
1981                         break;

1983                 default:
1984                         ((struct scsi_status *)pkt->pkt_scbp)->sts_busy = 1;
1985                 }

1987                 (void) mrsas_common_check(instance, cmd);
1988                 DTRACE_PROBE2(start_nointr_done, uint8_t, hdr->cmd,
1989                     uint8_t, hdr->cmd_status);
1990                 return_mfi_pkt(instance, cmd);

1992                 if (pkt->pkt_comp) {
1993                         (*pkt->pkt_comp)(pkt);
1994                 }

1996         }

1998         return (TRAN_ACCEPT);
1999 }

2001 /*
2002  * tran_abort - Abort any commands that are currently in transport
2003  * @ap:
2004  * @pkt:
2005  *
2006  * The tran_abort() entry point for a SCSI HBA driver is called to abort any
2007  * commands that are currently in transport for a particular target. This entry
2008  * point is called when a target driver calls scsi_abort(). The tran_abort()
2009  * entry point should attempt to abort the command denoted by the pkt
2010  * parameter. If the pkt parameter is NULL, tran_abort() should attempt to
2011  * abort all outstanding commands in the transport layer for the particular
2012  * target or logical unit.
2013  */
2014 /*ARGSUSED*/
2015 static int
2016 mrsas_tran_abort(struct scsi_address *ap, struct scsi_pkt *pkt)
2017 {
2018         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2020         /* abort command not supported by H/W */

2022         return (DDI_FAILURE);
2023 }

2025 /*
2026  * tran_reset - reset either the SCSI bus or target
2027  * @ap:
2028  * @level:
2029  *
2030  * The tran_reset() entry point for a SCSI HBA driver is called to reset either
2031  * the SCSI bus or a particular SCSI target device. This entry point is called
2032  * when a target driver calls scsi_reset(). The tran_reset() entry point must
2033  * reset the SCSI bus if level is RESET_ALL. If level is RESET_TARGET, just the
2034  * particular target or logical unit must be reset.
2035  */
2036 /*ARGSUSED*/
2037 static int
2038 mrsas_tran_reset(struct scsi_address *ap, int level)
```

```
2039 {
2040         struct mrsas_instance *instance = ADDR2MR(ap);

2042         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2044         if (wait_for_outstanding(instance)) {
2045                 con_log(CL_ANN1,
2046                     (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
2047                 return (DDI_FAILURE);
2048         } else {
2049                 return (DDI_SUCCESS);
2050         }
2051 }
1466         /* reset command not supported by H/W */

2053 #if 0
2054 /*
2055  * tran_bus_reset - reset the SCSI bus
2056  * @dip:
2057  * @level:
2058  *
2059  * The tran_bus_reset() vector in the scsi_hba_tran structure should be
2060  * initialized during the HBA driver's attach(). The vector should point to
2061  * an HBA entry point that is to be called when a user initiates a bus reset.
2062  * Implementation is hardware specific. If the HBA driver cannot reset the
2063  * SCSI bus without affecting the targets, the driver should fail RESET_BUS
2064  * or not initialize this vector.
2065  */
2066 /*ARGSUSED*/
2067 static int
2068 mrsas_tran_bus_reset(dev_info_t *dip, int level)
2069 {
2070         int     instance_no = ddi_get_instance(dip);

2072         struct mrsas_instance   *instance = ddi_get_soft_state(mrsas_state,
2073             instance_no);

2075         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2077         if (wait_for_outstanding(instance)) {
2078                 con_log(CL_ANN1,
2079                     (CE_CONT, "wait_for_outstanding: return FAIL.\n"));
2080                 return (DDI_FAILURE);
2081         } else {
2082                 return (DDI_SUCCESS);
2083         }

2084 }
2085 #endif

2087 /*
2088  * tran_getcap - get one of a set of SCSA-defined capabilities
2089  * @ap:
2090  * @cap:
2091  * @whom:
2092  *
2093  * The target driver can request the current setting of the capability for a
2094  * particular target by setting the whom parameter to nonzero. A whom value of
2095  * zero indicates a request for the current setting of the general capability
2096  * for the SCSI bus or for adapter hardware. The tran_getcap() should return -1
2097  * for undefined capabilities or the current value of the requested capability.
2098  */
2099 /*ARGSUSED*/
2100 static int
2101 mrsas_tran_getcap(struct scsi_address *ap, char *cap, int whom)
2102 {
```



```
2103         int     rval = 0;

2105         struct mrsas_instance   *instance = ADDR2MR(ap);

2107         con_log(CL_DLEVEL2, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1480         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2109         /* we do allow inquiring about capabilities for other targets */
2110         if (cap == NULL) {
2111                 return (-1);
2112         }

2114         switch (scsi_hba_lookup_capstr(cap)) {
2115         case SCSI_CAP_DMA_MAX:
2116                 if (instance->tbolt) {
2117                         /* Limit to 256k max transfer */
2118                         rval = mrsas_tbolt_max_cap_maxxfer;
2119                 } else {
2120                         /* Limit to 16MB max transfer */
2121                         rval = mrsas_max_cap_maxxfer;
2122                 }
2123                 break;
2124         case SCSI_CAP_MSG_OUT:
2125                 rval = 1;
2126                 break;
2127         case SCSI_CAP_DISCONNECT:
2128                 rval = 0;
2129                 break;
2130         case SCSI_CAP_SYNCHRONOUS:
2131                 rval = 0;
2132                 break;
2133         case SCSI_CAP_WIDE_XFER:
2134                 rval = 1;
2135                 break;
2136         case SCSI_CAP_TAGGED_QING:
2137                 rval = 1;
2138                 break;
2139         case SCSI_CAP_UNTAGGED_QING:
2140                 rval = 1;
2141                 break;
2142         case SCSI_CAP_PARITY:
2143                 rval = 1;
2144                 break;
2145         case SCSI_CAP_INITIATOR_ID:
2146                 rval = instance->init_id;
2147                 break;
2148         case SCSI_CAP_ARQ:
2149                 rval = 1;
2150                 break;
2151         case SCSI_CAP_LINKED_CMDS:
2152                 rval = 0;
2153                 break;
2154         case SCSI_CAP_RESET_NOTIFICATION:
2155                 rval = 1;
2156                 break;
2157         case SCSI_CAP_GEOMETRY:
2158                 rval = -1;

2160                 break;
2161         default:
2162                 con_log(CL_DLEVEL2, (CE_NOTE, "Default cap coming 0x%x",
2163                     scsi_hba_lookup_capstr(cap)));
2164                 rval = -1;
2165                 break;
2166         }
```

```
2168         return (rval);
2169 }

2171 /*
2172  * tran_setcap - set one of a set of SCSA-defined capabilities
2173  * @ap:
2174  * @cap:
2175  * @value:
2176  * @whom:
2177  *
2178  * The target driver might request that the new value be set for a particular
2179  * target by setting the whom parameter to nonzero. A whom value of zero
2180  * means that request is to set the new value for the SCSI bus or for adapter
2181  * hardware in general.
2182  * The tran_setcap() should return the following values as appropriate:
2183  * - -1 for undefined capabilities
2184  * - 0 if the HBA driver cannot set the capability to the requested value
2185  * - 1 if the HBA driver is able to set the capability to the requested value
2186  */
2187 /*ARGSUSED*/
2188 static int
2189 mrsas_tran_setcap(struct scsi_address *ap, char *cap, int value, int whom)
2190 {
2191         int             rval = 1;

2193         con_log(CL_DLEVEL2, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1545         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2195         /* We don't allow setting capabilities for other targets */
2196         if (cap == NULL || whom == 0) {
2197                 return (-1);
2198         }

2200         switch (scsi_hba_lookup_capstr(cap)) {
2201                 case SCSI_CAP_DMA_MAX:
2202                 case SCSI_CAP_MSG_OUT:
2203                 case SCSI_CAP_PARITY:
2204                 case SCSI_CAP_LINKED_CMDS:
2205                 case SCSI_CAP_RESET_NOTIFICATION:
2206                 case SCSI_CAP_DISCONNECT:
2207                 case SCSI_CAP_SYNCHRONOUS:
2208                 case SCSI_CAP_UNTAGGED_QING:
2209                 case SCSI_CAP_WIDE_XFER:
2210                 case SCSI_CAP_INITIATOR_ID:
2211                 case SCSI_CAP_ARQ:
2212                         /*
2213                          * None of these are settable via
2214                          * the capability interface.
2215                          */
2216                         break;
2217                 case SCSI_CAP_TAGGED_QING:
2218                         rval = 1;
2219                         break;
2220                 case SCSI_CAP_SECTOR_SIZE:
2221                         rval = 1;
2222                         break;

2224                 case SCSI_CAP_TOTAL_SECTORS:
2225                         rval = 1;
2226                         break;
2227                 default:
2228                         rval = -1;
2229                         break;
2230         }

2232         return (rval);
```



```
2233 }

2235 /*
2236  * tran_destroy_pkt - deallocate scsi_pkt structure
2237  * @ap:
2238  * @pkt:
2239  *
2240  * The tran_destroy_pkt() entry point is the HBA driver function that
2241  * deallocates scsi_pkt structures. The tran_destroy_pkt() entry point is
2242  * called when the target driver calls scsi_destroy_pkt(). The
2243  * tran_destroy_pkt() entry point must free any DMA resources that have been
2244  * allocated for the packet. An implicit DMA synchronization occurs if the
2245  * DMA resources are freed and any cached data remains after the completion
2246  * of the transfer.
2247  */
2248 static void
2249 mrsas_tran_destroy_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2250 {
2251         struct scsa_cmd *acmd = PKT2CMD(pkt);

2253         con_log(CL_DLEVEL2, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1592         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2255         if (acmd->cmd_flags & CFLAG_DMAVALID) {
2256                 acmd->cmd_flags &= ~CFLAG_DMAVALID;

2258                 (void) ddi_dma_unbind_handle(acmd->cmd_dmahandle);

2260                 ddi_dma_free_handle(&acmd->cmd_dmahandle);

2262                 acmd->cmd_dmahandle = NULL;
2263         }

2265         /* free the pkt */
2266         scsi_hba_pkt_free(ap, pkt);
2267 }

2269 /*
2270  * tran_dmafree - deallocates DMA resources
2271  * @ap:
2272  * @pkt:
2273  *
2274  * The tran_dmafree() entry point deallocates DMAQ resources that have been
2275  * allocated for a scsi_pkt structure. The tran_dmafree() entry point is
2276  * called when the target driver calls scsi_dmafree(). The tran_dmafree() must
2277  * free only DMA resources allocated for a scsi_pkt structure, not the
2278  * scsi_pkt itself. When DMA resources are freed, a DMA synchronization is
2279  * implicitly performed.
2280  */
2281 /*ARGSUSED*/
2282 static void
2283 mrsas_tran_dmafree(struct scsi_address *ap, struct scsi_pkt *pkt)
2284 {
2285         register struct scsa_cmd *acmd = PKT2CMD(pkt);

2287         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2289         if (acmd->cmd_flags & CFLAG_DMAVALID) {
2290                 acmd->cmd_flags &= ~CFLAG_DMAVALID;

2292                 (void) ddi_dma_unbind_handle(acmd->cmd_dmahandle);

2294                 ddi_dma_free_handle(&acmd->cmd_dmahandle);

2296                 acmd->cmd_dmahandle = NULL;
2297         }
```

```
2298 }

2300 /*
2301  * tran_sync_pkt - synchronize the DMA object allocated
2302  * @ap:
2303  * @pkt:
2304  *
2305  * The tran_sync_pkt() entry point synchronizes the DMA object allocated for
2306  * the scsi_pkt structure before or after a DMA transfer. The tran_sync_pkt()
2307  * entry point is called when the target driver calls scsi_sync_pkt(). If the
2308  * data transfer direction is a DMA read from device to memory, tran_sync_pkt()
2309  * must synchronize the CPU's view of the data. If the data transfer direction
2310  * is a DMA write from memory to device, tran_sync_pkt() must synchronize the
2311  * device's view of the data.
2312  */
2313 /*ARGSUSED*/
2314 static void
2315 mrsas_tran_sync_pkt(struct scsi_address *ap, struct scsi_pkt *pkt)
2316 {
2317         register struct scsa_cmd        *acmd = PKT2CMD(pkt);

2319         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2321         if (acmd->cmd_flags & CFLAG_DMAVALID) {
2322                 (void) ddi_dma_sync(acmd->cmd_dmahandle, acmd->cmd_dma_offset,
2323                     acmd->cmd_dma_len, (acmd->cmd_flags & CFLAG_DMASEND) ?
2324                     DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
2325         }
2326 }

2328 /*ARGSUSED*/
2329 static int
2330 mrsas_tran_quiesce(dev_info_t *dip)
2331 {
2332         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2334         return (1);
2335 }

2337 /*ARGSUSED*/
2338 static int
2339 mrsas_tran_unquiesce(dev_info_t *dip)
2340 {
2341         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2343         return (1);
2344 }


2347 /*
2348  * mrsas_isr(caddr_t)
2349  *
2350  * The Interrupt Service Routine
2351  *
2352  * Collect status for all completed commands and do callback
2353  *
2354  */
2355 static uint_t
2356 mrsas_isr(struct mrsas_instance *instance)
2357 {
2358         int             need_softintr;
2359         uint32_t        producer;
2360         uint32_t        consumer;
2361         uint32_t        context;
2362         int             retval;
```



```
2364         struct mrsas_cmd        *cmd;
2365         struct mrsas_header     *hdr;
2366         struct scsi_pkt         *pkt;

2368         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
2369         ASSERT(instance);
2370         if (instance->tbolt) {
2371                 mutex_enter(&instance->chip_mtx);
2372                 if ((instance->intr_type == DDI_INTR_TYPE_FIXED) &&
2373                     !(instance->func_ptr->intr_ack(instance))) {
2374                         mutex_exit(&instance->chip_mtx);
2375                         return (DDI_INTR_UNCLAIMED);
2376                 }
2377                 retval = mr_sas_tbolt_process_outstanding_cmd(instance);
2378                 mutex_exit(&instance->chip_mtx);
2379                 return (retval);
2380         } else {
2381                 if ((instance->intr_type == DDI_INTR_TYPE_FIXED) &&
2382                     !instance->func_ptr->intr_ack(instance)) {
2383                         return (DDI_INTR_UNCLAIMED);
2384                 }
2385         }

2387         (void) ddi_dma_sync(instance->mfi_internal_dma_obj.dma_handle,
2388             0, 0, DDI_DMA_SYNC_FORCPU);

2390         if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
2391             != DDI_SUCCESS) {
2392                 mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2393                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2394                 con_log(CL_ANN1, (CE_WARN,
2395                     "mr_sas_isr(): FMA check, returning DDI_INTR_UNCLAIMED"));
2396                 return (DDI_INTR_CLAIMED);
2397         }
2398         con_log(CL_ANN1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

2400 #ifdef OCRDEBUG
2401         if (debug_consecutive_timeout_after_ocr_g == 1) {
2402                 con_log(CL_ANN1, (CE_NOTE,
2403                     "simulating consecutive timeout after ocr"));
2404                 return (DDI_INTR_CLAIMED);
2405         }
2406 #endif

2408         mutex_enter(&instance->completed_pool_mtx);
2409         mutex_enter(&instance->cmd_pend_mtx);

2411         producer = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2412             instance->producer);
2413         consumer = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2414             instance->consumer);

2416         con_log(CL_ANN, (CE_CONT, " producer %x consumer %x ",
1697         con_log(CL_ANN1, (CE_NOTE, " producer %x consumer %x ",
2417             producer, consumer));
2418         if (producer == consumer) {
2419                 con_log(CL_ANN, (CE_WARN, "producer ==  consumer case"));
1700                 con_log(CL_ANN1, (CE_WARN, "producer =  consumer case"));
2420                 DTRACE_PROBE2(isr_pc_err, uint32_t, producer,
2421                     uint32_t, consumer);
2422                 mutex_exit(&instance->cmd_pend_mtx);
2423                 mutex_exit(&instance->completed_pool_mtx);
2424                 return (DDI_INTR_CLAIMED);
2425         }

2427         while (consumer != producer) {
```

```
2428                context = ddi_get32(instance->mfi_internal_dma_obj.acc_handle,
2429                    &instance->reply_queue[consumer]);
2430                cmd = instance->cmd_list[context];

2432                if (cmd->sync_cmd == MRSAS_TRUE) {
2433                        hdr = (struct mrsas_header *)&cmd->frame->hdr;
2434                        if (hdr) {
2435                                mlist_del_init(&cmd->list);
2436                        }
2437                } else {
2438                        pkt = cmd->pkt;
2439                        if (pkt) {
2440                                mlist_del_init(&cmd->list);
2441                        }
2442                }

2444                mlist_add_tail(&cmd->list, &instance->completed_pool_list);

2446                consumer++;
2447                if (consumer == (instance->max_fw_cmds + 1)) {
2448                        consumer = 0;
2449                }
2450        }
2451        ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
2452            instance->consumer, consumer);
2453        mutex_exit(&instance->cmd_pend_mtx);
2454        mutex_exit(&instance->completed_pool_mtx);

2456        (void) ddi_dma_sync(instance->mfi_internal_dma_obj.dma_handle,
2457            0, 0, DDI_DMA_SYNC_FORDEV);

2459        if (instance->softint_running) {
2460                need_softintr = 0;
2461        } else {
2462                need_softintr = 1;
2463        }

2465        if (instance->isr_level == HIGH_LEVEL_INTR) {
2466                if (need_softintr) {
2467                        ddi_trigger_softintr(instance->soft_intr_id);
2468                }
2469        } else {
2470                /*
2471                 * Not a high-level interrupt, therefore call the soft level
2472                 * interrupt explicitly
2473                 */
2474                (void) mrsas_softintr(instance);
2475        }

2477        return (DDI_INTR_CLAIMED);
2478 }


2481 /*
2482  * ************************************************************************** *
2483  *                                                                            *
2484  *                                libraries                                   *
2485  *                                                                            *
2486  * ************************************************************************** *
2487  */
2488 /*
2489  * get_mfi_pkt : Get a command from the free pool
2490  * After successful allocation, the caller of this routine
2491  * must clear the frame buffer (memset to zero) before
2492  * using the packet further.
2493  *
```



```
2494  * ***** Note *****
2495  * After clearing the frame buffer the context id of the
2496  * frame buffer SHOULD be restored back.
2497  */
2498 static struct mrsas_cmd *
2499 get_mfi_pkt(struct mrsas_instance *instance)
2500 {
2501        mlist_t                 *head = &instance->cmd_pool_list;
2502        struct mrsas_cmd        *cmd = NULL;

2504        mutex_enter(&instance->cmd_pool_mtx);
1786        ASSERT(mutex_owned(&instance->cmd_pool_mtx));

2506        if (!mlist_empty(head)) {
2507                cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2508                mlist_del_init(head->next);
2509        }
2510        if (cmd != NULL) {
2511                cmd->pkt = NULL;
2512                cmd->retry_count_for_ocr = 0;
2513                cmd->drv_pkt_time = 0;

2515        }
2516        mutex_exit(&instance->cmd_pool_mtx);

2518        return (cmd);
2519 }

2521 static struct mrsas_cmd *
2522 get_mfi_app_pkt(struct mrsas_instance *instance)
2523 {
2524        mlist_t                         *head = &instance->app_cmd_pool_list;
2525        struct mrsas_cmd        *cmd = NULL;

2527        mutex_enter(&instance->app_cmd_pool_mtx);
1809        ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2529        if (!mlist_empty(head)) {
2530                cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2531                mlist_del_init(head->next);
2532        }
2533        if (cmd != NULL) {
1815        if (cmd != NULL)
2534                cmd->pkt = NULL;
2535                cmd->retry_count_for_ocr = 0;
2536                cmd->drv_pkt_time = 0;
2537        }

2539        mutex_exit(&instance->app_cmd_pool_mtx);

2541        return (cmd);
2542 }
2543 /*
2544  * return_mfi_pkt : Return a cmd to free command pool
2545  */
2546 static void
2547 return_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2548 {
2549        mutex_enter(&instance->cmd_pool_mtx);
1828        ASSERT(mutex_owned(&instance->cmd_pool_mtx));
2550        /* use mlist_add_tail for debug assistance */
2551        mlist_add_tail(&cmd->list, &instance->cmd_pool_list);

2553        mutex_exit(&instance->cmd_pool_mtx);
2554 }
```

```
2556 static void
2557 return_mfi_app_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2558 {
2559         mutex_enter(&instance->app_cmd_pool_mtx);
1839         ASSERT(mutex_owned(&instance->app_cmd_pool_mtx));

2561         mlist_add(&cmd->list, &instance->app_cmd_pool_list);

2563         mutex_exit(&instance->app_cmd_pool_mtx);
2564 }
2565 void
1845 static void
2566 push_pending_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2567 {
2568         struct scsi_pkt *pkt;
2569         struct mrsas_header     *hdr;
2570         con_log(CL_DLEVEL2, (CE_NOTE, "push_pending_pkt(): Called\n"));
1850         con_log(CL_ANN1, (CE_NOTE, "push_pending_pkt(): Called\n"));
2571         mutex_enter(&instance->cmd_pend_mtx);
1852         ASSERT(mutex_owned(&instance->cmd_pend_mtx));
2572         mlist_del_init(&cmd->list);
2573         mlist_add_tail(&cmd->list, &instance->cmd_pend_list);
2574         if (cmd->sync_cmd == MRSAS_TRUE) {
2575                 hdr = (struct mrsas_header *)&cmd->frame->hdr;
2576                 if (hdr) {
2577                         con_log(CL_ANN1, (CE_CONT,
2578                             "push_pending_mfi_pkt: "
2579                             "cmd %p index %x "
2580                             "time %llx",
2581                             (void *)cmd, cmd->index,
2582                             gethrtime()));
2583                         /* Wait for specified interval  */
2584                         cmd->drv_pkt_time = ddi_get16(
2585                             cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2586                         if (cmd->drv_pkt_time < debug_timeout_g)
2587                                 cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2588                                 con_log(CL_ANN1, (CE_CONT,
2589                                     "push_pending_pkt(): "
2590                                     "Called IO Timeout Value %x\n",
2591                                     cmd->drv_pkt_time));
2592                 }
2593                 if (hdr && instance->timeout_id == (timeout_id_t)-1) {
2594                         instance->timeout_id = timeout(io_timeout_checker,
2595                             (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2596                 }
2597         } else {
2598                 pkt = cmd->pkt;
2599                 if (pkt) {
2600                         con_log(CL_ANN1, (CE_CONT,
2601                             "push_pending_mfi_pkt: "
2602                             "cmd %p index %x pkt %p, "
2603                             "time %llx",
2604                             (void *)cmd, cmd->index, (void *)pkt,
2605                             gethrtime()));
2606                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2607                 }
2608                 if (pkt && instance->timeout_id == (timeout_id_t)-1) {
2609                         instance->timeout_id = timeout(io_timeout_checker,
2610                             (void *) instance, drv_usectohz(MRSAS_1_SECOND));
2611                 }
2612         }

2614         mutex_exit(&instance->cmd_pend_mtx);

2616 }
```



```
2618 int
1898 static int
2619 mrsas_print_pending_cmds(struct mrsas_instance *instance)
2620 {
2621         mlist_t *head = &instance->cmd_pend_list;
2622         mlist_t *tmp = head;
2623         struct mrsas_cmd *cmd = NULL;
2624         struct mrsas_header     *hdr;
2625         unsigned int            flag = 1;
2626         struct scsi_pkt *pkt;
2627         int saved_level;
2628         int cmd_count = 0;

2630         saved_level = debug_level_g;
2631         debug_level_g = CL_ANN1;

2633         cmn_err(CE_NOTE, "mrsas_print_pending_cmds(): Called\n");

1907         struct scsi_pkt *pkt;
1908         con_log(CL_ANN1, (CE_NOTE,
1909             "mrsas_print_pending_cmds(): Called"));
2635         while (flag) {
2636                 mutex_enter(&instance->cmd_pend_mtx);
2637                 tmp     =       tmp->next;
2638                 if (tmp == head) {
2639                         mutex_exit(&instance->cmd_pend_mtx);
2640                         flag = 0;
2641                         con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds():"
2642                             " NO MORE CMDS PENDING....\n"));
2643                         break;
2644                 } else {
2645                         cmd = mlist_entry(tmp, struct mrsas_cmd, list);
2646                         mutex_exit(&instance->cmd_pend_mtx);
2647                         if (cmd) {
2648                                 if (cmd->sync_cmd == MRSAS_TRUE) {
2649                                         hdr = (struct mrsas_header *)
2650                                             &cmd->frame->hdr;
1922                                 hdr = (struct mrsas_header *)&cmd->frame->hdr;
2651                                         if (hdr) {
2652                                                 con_log(CL_ANN1, (CE_CONT,
2653                                                     "print: cmd %p index 0x%x "
2654                                                     "drv_pkt_time 0x%x (NO-PKT)"
2655                                                     " hdr %p\n", (void *)cmd,
2656                                                     cmd->index,
2657                                                     cmd->drv_pkt_time,
1925                                                 "print: cmd %p index %x hdr %p",
1926                                                 (void *)cmd, cmd->index,
2658                                                     (void *)hdr));
2659                                         }
2660                                 } else {
2661                                         pkt = cmd->pkt;
2662                                         if (pkt) {
2663                                         con_log(CL_ANN1, (CE_CONT,
2664                                             "print: cmd %p index 0x%x "
2665                                             "drv_pkt_time 0x%x pkt %p \n",
2666                                             (void *)cmd, cmd->index,
2667                                             cmd->drv_pkt_time, (void *)pkt));
1933                                             "print: cmd %p index %x "
1934                                             "pkt %p", (void *)cmd, cmd->index,
1935                                             (void *)pkt));
2668                                         }
2669                                 }

2671                                 if (++cmd_count == 1) {
2672                                         mrsas_print_cmd_details(instance, cmd,
2673                                             0xDD);
```

```
2674                                } else {
2675                                        mrsas_print_cmd_details(instance, cmd,
2676                                            1);
2677                                }

2679                        }
2680                }
2681        }
2682        con_log(CL_ANN1, (CE_CONT, "mrsas_print_pending_cmds(): Done\n"));


2685        debug_level_g = saved_level;

1941        con_log(CL_ANN1, (CE_NOTE, "mrsas_print_pending_cmds(): Done\n"));
2687        return (DDI_SUCCESS);
2688 }


2691 int
1946 static int
2692 mrsas_complete_pending_cmds(struct mrsas_instance *instance)
2693 {

2695        struct mrsas_cmd *cmd = NULL;
2696        struct scsi_pkt *pkt;
2697        struct mrsas_header *hdr;

2699        struct mlist_head               *pos, *next;

2701        con_log(CL_ANN1, (CE_NOTE,
2702            "mrsas_complete_pending_cmds(): Called"));

2704        mutex_enter(&instance->cmd_pend_mtx);
2705        mlist_for_each_safe(pos, next, &instance->cmd_pend_list) {
2706                cmd = mlist_entry(pos, struct mrsas_cmd, list);
2707                if (cmd) {
2708                        pkt = cmd->pkt;
2709                        if (pkt) { /* for IO */
2710                                if (((pkt->pkt_flags & FLAG_NOINTR)
2711                                    == 0) && pkt->pkt_comp) {
2712                                        pkt->pkt_reason
2713                                            = CMD_DEV_GONE;
2714                                        pkt->pkt_statistics
2715                                            = STAT_DISCON;
2716                                        con_log(CL_ANN1, (CE_CONT,
1971                                        con_log(CL_ANN1, (CE_NOTE,
2717                                            "fail and posting to scsa "
2718                                            "cmd %p index %x"
2719                                            " pkt %p "
2720                                            "time : %llx",
2721                                            (void *)cmd, cmd->index,
2722                                            (void *)pkt, gethrtime()));
2723                                        (*pkt->pkt_comp)(pkt);
2724                                }
2725                        } else { /* for DCMDS */
2726                                if (cmd->sync_cmd == MRSAS_TRUE) {
2727                                hdr = (struct mrsas_header *)&cmd->frame->hdr;
2728                                con_log(CL_ANN1, (CE_CONT,
1983                                con_log(CL_ANN1, (CE_NOTE,
2729                                    "posting invalid status to application "
2730                                    "cmd %p index %x"
2731                                    " hdr %p "
2732                                    "time : %llx",
2733                                    (void *)cmd, cmd->index,
2734                                    (void *)hdr, gethrtime()));
2735                                hdr->cmd_status = MFI_STAT_INVALID_STATUS;
```



```
2736                                    complete_cmd_in_sync_mode(instance, cmd);
2737                                    }
2738                            }
2739                            mlist_del_init(&cmd->list);
2740                    } else {
2741                            con_log(CL_ANN1, (CE_CONT,
1996                            con_log(CL_ANN1, (CE_NOTE,
2742                                "mrsas_complete_pending_cmds:"
2743                                "NULL command\n"));
2744                    }
2745                    con_log(CL_ANN1, (CE_CONT,
2000                    con_log(CL_ANN1, (CE_NOTE,
2746                        "mrsas_complete_pending_cmds:"
2747                        "looping for more commands\n"));
2748            }
2749            mutex_exit(&instance->cmd_pend_mtx);

2751            con_log(CL_ANN1, (CE_CONT, "mrsas_complete_pending_cmds(): DONE\n"));
2006            con_log(CL_ANN1, (CE_NOTE, "mrsas_complete_pending_cmds(): DONE\n"));
2752            return (DDI_SUCCESS);
2753 }

2755 void
2756 mrsas_print_cmd_details(struct mrsas_instance *instance, struct mrsas_cmd *cmd,
2757     int detail)
2758 {
2759            struct scsi_pkt *pkt = cmd->pkt;
2760            Mpi2RaidSCSIIORequest_t *scsi_io = cmd->scsi_io_request;
2761            int i;
2762            int saved_level;
2763            ddi_acc_handle_t acc_handle =
2764                instance->mpi2_frame_pool_dma_obj.acc_handle;

2766            if (detail == 0xDD) {
2767                    saved_level = debug_level_g;
2768                    debug_level_g = CL_ANN1;
2769            }


2772            if (instance->tbolt) {
2773                    con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p "
2774                        "cmd->index 0x%x SMID 0x%x timer 0x%x sec\n",
2775                        (void *)cmd, cmd->index, cmd->SMID, cmd->drv_pkt_time));
2776            } else {
2777                    con_log(CL_ANN1, (CE_CONT, "print_cmd_details: cmd %p "
2778                        "cmd->index 0x%x timer 0x%x sec\n",
2779                        (void *)cmd, cmd->index, cmd->drv_pkt_time));
2780            }

2782            if (pkt) {
2783                    con_log(CL_ANN1, (CE_CONT, "scsi_pkt CDB[0]=0x%x",
2784                        pkt->pkt_cdbp[0]));
2785            } else {
2786                    con_log(CL_ANN1, (CE_CONT, "NO-PKT"));
2787            }

2789            if ((detail == 0xDD) && instance->tbolt) {
2790                    con_log(CL_ANN1, (CE_CONT, "RAID_SCSI_IO_REQUEST\n"));
2791                    con_log(CL_ANN1, (CE_CONT, "DevHandle=0x%X Function=0x%X "
2792                        "IoFlags=0x%X SGLFlags=0x%X DataLength=0x%X\n",
2793                        ddi_get16(acc_handle, &scsi_io->DevHandle),
2794                        ddi_get8(acc_handle, &scsi_io->Function),
2795                        ddi_get16(acc_handle, &scsi_io->IoFlags),
2796                        ddi_get16(acc_handle, &scsi_io->SGLFlags),
2797                        ddi_get32(acc_handle, &scsi_io->DataLength)));
```

```
2799                 for (i = 0; i < 32; i++) {
2800                         con_log(CL_ANN1, (CE_CONT, "CDB[%d]=0x%x ", i,
2801                             ddi_get8(acc_handle, &scsi_io->CDB.CDB32[i])));
2802                 }

2804                 con_log(CL_ANN1, (CE_CONT, "RAID-CONTEXT\n"));
2805                 con_log(CL_ANN1, (CE_CONT, "status=0x%X extStatus=0x%X "
2806                     "ldTargetId=0x%X timeoutValue=0x%X regLockFlags=0x%X "
2807                     "RAIDFlags=0x%X regLockRowLBA=0x%" PRIu64
2808                     " regLockLength=0x%X spanArm=0x%X\n",
2809                     ddi_get8(acc_handle, &scsi_io->RaidContext.status),
2810                     ddi_get8(acc_handle, &scsi_io->RaidContext.extStatus),
2811                     ddi_get16(acc_handle, &scsi_io->RaidContext.ldTargetId),
2812                     ddi_get16(acc_handle, &scsi_io->RaidContext.timeoutValue),
2813                     ddi_get8(acc_handle, &scsi_io->RaidContext.regLockFlags),
2814                     ddi_get8(acc_handle, &scsi_io->RaidContext.RAIDFlags),
2815                     ddi_get64(acc_handle, &scsi_io->RaidContext.regLockRowLBA),
2816                     ddi_get32(acc_handle, &scsi_io->RaidContext.regLockLength),
2817                     ddi_get8(acc_handle, &scsi_io->RaidContext.spanArm)));
2818         }

2820         if (detail == 0xDD) {
2821                 debug_level_g = saved_level;
2822         }
2823 }


2826 int
2011 static int
2827 mrsas_issue_pending_cmds(struct mrsas_instance *instance)
2828 {
2829         mlist_t *head   =       &instance->cmd_pend_list;
2830         mlist_t *tmp    =       head->next;
2831         struct mrsas_cmd *cmd = NULL;
2832         struct scsi_pkt *pkt;

2834         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_pending_cmds(): Called"));
2835         while (tmp != head) {
2836                 mutex_enter(&instance->cmd_pend_mtx);
2837                 cmd = mlist_entry(tmp, struct mrsas_cmd, list);
2838                 tmp = tmp->next;
2839                 mutex_exit(&instance->cmd_pend_mtx);
2840                 if (cmd) {
2841                         con_log(CL_ANN1, (CE_CONT,
2026                         con_log(CL_ANN1, (CE_NOTE,
2842                             "mrsas_issue_pending_cmds(): "
2843                             "Got a cmd: cmd %p index 0x%x drv_pkt_time 0x%x ",
2844                             (void *)cmd, cmd->index, cmd->drv_pkt_time));

2846                         /* Reset command timeout value */
2847                         if (cmd->drv_pkt_time < debug_timeout_g)
2848                                 cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

2028                             "Got a cmd: cmd:%p\n", (void *)cmd));
2850                         cmd->retry_count_for_ocr++;

2852                         cmn_err(CE_CONT, "cmd retry count = %d\n",
2853                             cmd->retry_count_for_ocr);

2030                         con_log(CL_ANN1, (CE_NOTE,
2031                             "mrsas_issue_pending_cmds(): "
2032                             "cmd retry count = %d\n",
2033                             cmd->retry_count_for_ocr));
2855                         if (cmd->retry_count_for_ocr > IO_RETRY_COUNT) {
2856                                 cmn_err(CE_WARN, "mrsas_issue_pending_cmds(): "
2857                                     "cmd->retry_count exceeded limit >%d\n",
```



```
2858                                     IO_RETRY_COUNT);
2859                                 mrsas_print_cmd_details(instance, cmd, 0xDD);

2861                                 cmn_err(CE_WARN,
2035                                 con_log(CL_ANN1, (CE_NOTE,
2862                                     "mrsas_issue_pending_cmds():"
2863                                     "Calling KILL Adapter\n");
2864                                 if (instance->tbolt)
2865                                         mrsas_tbolt_kill_adapter(instance);
2866                                 else
2037                                     "Calling Kill Adapter\n"));
2867                                         (void) mrsas_kill_adapter(instance);
2868                                 return (DDI_FAILURE);
2869                         }

2871                         pkt = cmd->pkt;
2872                         if (pkt) {
2873                                 con_log(CL_ANN1, (CE_CONT,
2874                                     "PENDING PKT-CMD ISSUE: cmd %p index %x "
2043                                 con_log(CL_ANN1, (CE_NOTE,
2044                                     "PENDING ISSUE: cmd %p index %x "
2875                                     "pkt %p time %llx",
2876                                     (void *)cmd, cmd->index,
2877                                     (void *)pkt,
2878                                     gethrtime()));

2880                         } else {
2881                                 cmn_err(CE_CONT,
2882                                     "mrsas_issue_pending_cmds(): NO-PKT, "
2883                                     "cmd %p index 0x%x drv_pkt_time 0x%x ",
2884                                     (void *)cmd, cmd->index, cmd->drv_pkt_time);
2885                         }


2888                         if (cmd->sync_cmd == MRSAS_TRUE) {
2889                                 cmn_err(CE_CONT, "mrsas_issue_pending_cmds(): "
2890                                     "SYNC_CMD == TRUE \n");
2891                                 instance->func_ptr->issue_cmd_in_sync_mode(
2892                                     instance, cmd);
2893                         } else {
2894                                 instance->func_ptr->issue_cmd(cmd, instance);
2895                         }
2896                 } else {
2897                         con_log(CL_ANN1, (CE_CONT,
2058                         con_log(CL_ANN1, (CE_NOTE,
2898                             "mrsas_issue_pending_cmds: NULL command\n"));
2899                 }
2900                 con_log(CL_ANN1, (CE_CONT,
2061                 con_log(CL_ANN1, (CE_NOTE,
2901                     "mrsas_issue_pending_cmds:"
2902                     "looping for more commands"));
2903         }
2904         con_log(CL_ANN1, (CE_CONT, "mrsas_issue_pending_cmds(): DONE\n"));
2065         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_pending_cmds(): DONE\n"));
2905         return (DDI_SUCCESS);
2906 }


2910 /*
2911  * destroy_mfi_frame_pool
2912  */
2913 void
2072 static void
2914 destroy_mfi_frame_pool(struct mrsas_instance *instance)
2915 {
```

```
2916         int             i;
2917         uint32_t        max_cmd = instance->max_fw_cmds;

2919         struct mrsas_cmd        *cmd;

2921         /* return all frames to pool */
2081         for (i = 0; i < max_cmd+1; i++) {

2923         for (i = 0; i < max_cmd; i++) {

2925                 cmd = instance->cmd_list[i];

2927                 if (cmd->frame_dma_obj_status == DMA_OBJ_ALLOCATED)
2928                         (void) mrsas_free_dma_obj(instance, cmd->frame_dma_obj);

2930                 cmd->frame_dma_obj_status  = DMA_OBJ_FREED;
2931         }

2933 }

2935 /*
2936  * create_mfi_frame_pool
2937  */
2938 int
2096 static int
2939 create_mfi_frame_pool(struct mrsas_instance *instance)
2940 {
2941         int             i = 0;
2942         int             cookie_cnt;
2943         uint16_t        max_cmd;
2944         uint16_t        sge_sz;
2945         uint32_t        sgl_sz;
2946         uint32_t        tot_frame_size;
2947         struct mrsas_cmd        *cmd;
2948         int                     retval = DDI_SUCCESS;

2950         max_cmd = instance->max_fw_cmds;

2951         sge_sz  = sizeof (struct mrsas_sge_ieee);

2952         /* calculated the number of 64byte frames required for SGL */
2953         sgl_sz          = sge_sz * instance->max_num_sge;
2954         tot_frame_size  = sgl_sz + MRMFI_FRAME_SIZE + SENSE_LENGTH;

2956         con_log(CL_DLEVEL3, (CE_NOTE, "create_mfi_frame_pool: "
2957             "sgl_sz %x tot_frame_size %x", sgl_sz, tot_frame_size));

2959         while (i < max_cmd) {
2118         while (i < max_cmd+1) {
2960                 cmd = instance->cmd_list[i];

2962                 cmd->frame_dma_obj.size = tot_frame_size;
2963                 cmd->frame_dma_obj.dma_attr = mrsas_generic_dma_attr;
2964                 cmd->frame_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
2965                 cmd->frame_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
2966                 cmd->frame_dma_obj.dma_attr.dma_attr_sgllen = 1;
2967                 cmd->frame_dma_obj.dma_attr.dma_attr_align = 64;


2969                 cookie_cnt = mrsas_alloc_dma_obj(instance, &cmd->frame_dma_obj,
2970                     (uchar_t)DDI_STRUCTURE_LE_ACC);

2972                 if (cookie_cnt == -1 || cookie_cnt > 1) {
2973                         cmn_err(CE_WARN,
2974                             "create_mfi_frame_pool: could not alloc.");
2975                         retval = DDI_FAILURE;
```



```
2976                         goto mrsas_undo_frame_pool;
2133                         con_log(CL_ANN, (CE_WARN,
2134                             "create_mfi_frame_pool: could not alloc."));
2135                         return (DDI_FAILURE);
2977                 }

2979                 bzero(cmd->frame_dma_obj.buffer, tot_frame_size);

2981                 cmd->frame_dma_obj_status = DMA_OBJ_ALLOCATED;
2982                 cmd->frame = (union mrsas_frame *)cmd->frame_dma_obj.buffer;
2983                 cmd->frame_phys_addr =
2984                     cmd->frame_dma_obj.dma_cookie[0].dmac_address;

2986                 cmd->sense = (uint8_t *)(((unsigned long)
2987                     cmd->frame_dma_obj.buffer) +
2988                     tot_frame_size - SENSE_LENGTH);
2989                 cmd->sense_phys_addr =
2990                     cmd->frame_dma_obj.dma_cookie[0].dmac_address +
2991                     tot_frame_size - SENSE_LENGTH;

2993                 if (!cmd->frame || !cmd->sense) {
2994                         cmn_err(CE_WARN,
2995                             "mr_sas: pci_pool_alloc failed");
2996                         retval = ENOMEM;
2997                         goto mrsas_undo_frame_pool;
2153                         con_log(CL_ANN, (CE_NOTE,
2154                             "mr_sas: pci_pool_alloc failed"));

2156                         return (ENOMEM);
2998                 }

3000                 ddi_put32(cmd->frame_dma_obj.acc_handle,
3001                     &cmd->frame->io.context, cmd->index);
3002                 i++;

3004                 con_log(CL_DLEVEL3, (CE_NOTE, "[%x]-%x",
3005                     cmd->index, cmd->frame_phys_addr));
3006         }

3008         return (DDI_SUCCESS);

3010 mrsas_undo_frame_pool:
3011         if (i > 0)
3012                 destroy_mfi_frame_pool(instance);

3014         return (retval);
3015 }
_____unchanged_portion_omitted_

3036 /*
3037  * alloc_additional_dma_buffer
3038  */
3039 static int
3040 alloc_additional_dma_buffer(struct mrsas_instance *instance)
3041 {
3042         uint32_t        reply_q_sz;
3043         uint32_t        internal_buf_size = PAGESIZE*2;

3045         /* max cmds plus 1 + producer & consumer */
3046         reply_q_sz = sizeof (uint32_t) * (instance->max_fw_cmds + 1 + 2);

3048         instance->mfi_internal_dma_obj.size = internal_buf_size;
3049         instance->mfi_internal_dma_obj.dma_attr = mrsas_generic_dma_attr;
3050         instance->mfi_internal_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
3051         instance->mfi_internal_dma_obj.dma_attr.dma_attr_count_max =
3052             0xFFFFFFFFU;
```

```
3053         instance->mfi_internal_dma_obj.dma_attr.dma_attr_sgllen = 1;

3055         if (mrsas_alloc_dma_obj(instance, &instance->mfi_internal_dma_obj,
3056             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
3057                 cmn_err(CE_WARN,
3058                     "mr_sas: could not alloc reply queue");
2210                 con_log(CL_ANN, (CE_WARN,
2211                     "mr_sas: could not alloc reply queue"));
3059                 return (DDI_FAILURE);
3060         }

3062         bzero(instance->mfi_internal_dma_obj.buffer, internal_buf_size);

3064         instance->mfi_internal_dma_obj.status |= DMA_OBJ_ALLOCATED;

3066         instance->producer = (uint32_t *)((unsigned long)
3067             instance->mfi_internal_dma_obj.buffer);
3068         instance->consumer = (uint32_t *)((unsigned long)
3069             instance->mfi_internal_dma_obj.buffer + 4);
3070         instance->reply_queue = (uint32_t *)((unsigned long)
3071             instance->mfi_internal_dma_obj.buffer + 8);
3072         instance->internal_buf = (caddr_t)(((unsigned long)
3073             instance->mfi_internal_dma_obj.buffer) + reply_q_sz + 8);
3074         instance->internal_buf_dmac_add =
3075             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address +
3076             (reply_q_sz + 8);
3077         instance->internal_buf_size = internal_buf_size -
3078             (reply_q_sz + 8);

3080         /* allocate evt_detail */
3081         instance->mfi_evt_detail_obj.size = sizeof (struct mrsas_evt_detail);
3082         instance->mfi_evt_detail_obj.dma_attr = mrsas_generic_dma_attr;
3083         instance->mfi_evt_detail_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
3084         instance->mfi_evt_detail_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
3085         instance->mfi_evt_detail_obj.dma_attr.dma_attr_sgllen = 1;
3086         instance->mfi_evt_detail_obj.dma_attr.dma_attr_align = 1;

3088         if (mrsas_alloc_dma_obj(instance, &instance->mfi_evt_detail_obj,
3089             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
3090                 cmn_err(CE_WARN, "alloc_additional_dma_buffer: "
3091                     "could not allocate data transfer buffer.");
3092                 goto mrsas_undo_internal_buff;
2243                 con_log(CL_ANN, (CE_WARN, "alloc_additional_dma_buffer: "
2244                     "could not allocate data transfer buffer."));
2245                 return (DDI_FAILURE);
3093         }

3095         bzero(instance->mfi_evt_detail_obj.buffer,
3096             sizeof (struct mrsas_evt_detail));

3098         instance->mfi_evt_detail_obj.status |= DMA_OBJ_ALLOCATED;

3100         return (DDI_SUCCESS);

3102 mrsas_undo_internal_buff:
3103         if (instance->mfi_internal_dma_obj.status == DMA_OBJ_ALLOCATED) {
3104                 (void) mrsas_free_dma_obj(instance,
3105                     instance->mfi_internal_dma_obj);
3106                 instance->mfi_internal_dma_obj.status = DMA_OBJ_FREED;
3107         }

3109         return (DDI_FAILURE);
3110 }


3113 void
```



```
3114 mrsas_free_cmd_pool(struct mrsas_instance *instance)
2256 /*
2257  * free_space_for_mfi
2258  */
2259 static void
2260 free_space_for_mfi(struct mrsas_instance *instance)
2115 {
3116         int             i;
3117         uint32_t        max_cmd;
3118         size_t          sz;
2263         uint32_t        max_cmd = instance->max_fw_cmds;

3120         /* already freed */
3121         if (instance->cmd_list == NULL) {
3122                 return;
3123         }

3125         max_cmd = instance->max_fw_cmds;
2270         free_additional_dma_buffer(instance);

3127         /* size of cmd_list array */
3128         sz = sizeof (struct mrsas_cmd *) * max_cmd;
2272         /* first free the MFI frame pool */
2273         destroy_mfi_frame_pool(instance);

3130         /* First free each cmd */
3131         for (i = 0; i < max_cmd; i++) {
3132                 if (instance->cmd_list[i] != NULL) {
2275         /* free all the commands in the cmd_list */
2276         for (i = 0; i < instance->max_fw_cmds+1; i++) {
3133                         kmem_free(instance->cmd_list[i],
3134                             sizeof (struct mrsas_cmd));
3135                 }

3137                 instance->cmd_list[i] = NULL;
3138         }

3140         /* Now, free cmd_list array */
3141         if (instance->cmd_list != NULL)
3142                 kmem_free(instance->cmd_list, sz);
2283         /* free the cmd_list buffer itself */
2284         kmem_free(instance->cmd_list,
2285             sizeof (struct mrsas_cmd *) * (max_cmd+1));

3144         instance->cmd_list = NULL;

3146         INIT_LIST_HEAD(&instance->cmd_pool_list);
3147         INIT_LIST_HEAD(&instance->cmd_pend_list);
3148         if (instance->tbolt) {
3149                 INIT_LIST_HEAD(&instance->cmd_app_pool_list);
3150         } else {
3151                 INIT_LIST_HEAD(&instance->app_cmd_pool_list);
3152         }

2291         INIT_LIST_HEAD(&instance->cmd_pend_list);
3154 }


3157 /*
3158  * mrsas_alloc_cmd_pool
2295  * alloc_space_for_mfi
3159  */
3160 int
3161 mrsas_alloc_cmd_pool(struct mrsas_instance *instance)
2297 static int
2298 alloc_space_for_mfi(struct mrsas_instance *instance)
```

```
3162 {
3163         int             i;
3164         int             count;
3165         uint32_t        max_cmd;
3166         uint32_t        reserve_cmd;
3167         size_t          sz;

3169         struct mrsas_cmd        *cmd;

3171         max_cmd = instance->max_fw_cmds;
3172         con_log(CL_ANN1, (CE_NOTE, "mrsas_alloc_cmd_pool: "
3173             "max_cmd %x", max_cmd));

2309         /* reserve 1 more slot for flush_cache */
2310         sz = sizeof (struct mrsas_cmd *) * (max_cmd+1);

3176         sz = sizeof (struct mrsas_cmd *) * max_cmd;

3178         /*
3179          * instance->cmd_list is an array of struct mrsas_cmd pointers.
3180          * Allocate the dynamic array first and then allocate individual
3181          * commands.
3182          */
3183         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);
3184         ASSERT(instance->cmd_list);

3186         /* create a frame pool and assign one frame to each cmd */
3187         for (count = 0; count < max_cmd; count++) {
3188                 instance->cmd_list[count] =
3189                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
3190                 ASSERT(instance->cmd_list[count]);
2320         for (i = 0; i < max_cmd+1; i++) {
2321                 instance->cmd_list[i] = kmem_zalloc(sizeof (struct mrsas_cmd),
2322                     KM_SLEEP);
2323                 ASSERT(instance->cmd_list[i]);
3191         }

3193         /* add all the commands to command pool */

3195         INIT_LIST_HEAD(&instance->cmd_pool_list);
3196         INIT_LIST_HEAD(&instance->cmd_pend_list);
2328         /* add all the commands to command pool (instance->cmd_pool) */
2329         reserve_cmd     =       APP_RESERVE_CMDS;
3197         INIT_LIST_HEAD(&instance->app_cmd_pool_list);

3199         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

3201         for (i = 0; i < reserve_cmd; i++) {
2331         for (i = 0; i < reserve_cmd-1; i++) {
3202                 cmd = instance->cmd_list[i];
3203                 cmd->index = i;
3204                 mlist_add_tail(&cmd->list, &instance->app_cmd_pool_list);
3205         }


2336         /*
2337          * reserve slot instance->cmd_list[APP_RESERVE_CMDS-1]
2338          * for abort_aen_cmd
2339          */
3208         for (i = reserve_cmd; i < max_cmd; i++) {
3209                 cmd = instance->cmd_list[i];
3210                 cmd->index = i;
3211                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
3212         }

3214         return (DDI_SUCCESS);
```



```
2346         /* single slot for flush_cache won't be added in command pool */
2347         cmd             = instance->cmd_list[max_cmd];
2348         cmd->index      = i;

3216 mrsas_undo_cmds:
3217         if (count > 0) {
3218                 /* free each cmd */
3219                 for (i = 0; i < count; i++) {
3220                         if (instance->cmd_list[i] != NULL) {
3221                                 kmem_free(instance->cmd_list[i],
3222                                     sizeof (struct mrsas_cmd));
3223                         }
3224                         instance->cmd_list[i] = NULL;
3225                 }
3226         }

3228 mrsas_undo_cmd_list:
3229         if (instance->cmd_list != NULL)
3230                 kmem_free(instance->cmd_list, sz);
3231         instance->cmd_list = NULL;

2350         /* create a frame pool and assign one frame to each cmd */
2351         if (create_mfi_frame_pool(instance)) {
2352                 con_log(CL_ANN, (CE_NOTE, "error creating frame DMA pool"));
3233         return (DDI_FAILURE);
3234 }


3237 /*
3238  * free_space_for_mfi
3239  */
3240 static void
3241 free_space_for_mfi(struct mrsas_instance *instance)
3242 {

3244         /* already freed */
3245         if (instance->cmd_list == NULL) {
3246                 return;
3247         }

3249         /* Free additional dma buffer */
3250         free_additional_dma_buffer(instance);

3252         /* Free the MFI frame pool */
3253         destroy_mfi_frame_pool(instance);

3255         /* Free all the commands in the cmd_list */
3256         /* Free the cmd_list buffer itself */
3257         mrsas_free_cmd_pool(instance);
3258 }

3260 /*
3261  * alloc_space_for_mfi
3262  */
3263 static int
3264 alloc_space_for_mfi(struct mrsas_instance *instance)
3265 {
3266         /* Allocate command pool (memory for cmd_list & individual commands) */
3267         if (mrsas_alloc_cmd_pool(instance)) {
3268                 cmn_err(CE_WARN, "error creating cmd pool");
2356         /* create a frame pool and assign one frame to each cmd */
2357         if (alloc_additional_dma_buffer(instance)) {
2358                 con_log(CL_ANN, (CE_NOTE, "error creating frame DMA pool"));
3269                 return (DDI_FAILURE);
3270         }
```

```
3272         /* Allocate MFI Frame pool */
3273         if (create_mfi_frame_pool(instance)) {
3274                 cmn_err(CE_WARN, "error creating frame DMA pool");
3275                 goto mfi_undo_cmd_pool;
3276         }

3278         /* Allocate additional DMA buffer */
3279         if (alloc_additional_dma_buffer(instance)) {
3280                 cmn_err(CE_WARN, "error creating frame DMA pool");
3281                 goto mfi_undo_frame_pool;
3282         }

3284         return (DDI_SUCCESS);

3286 mfi_undo_frame_pool:
3287         destroy_mfi_frame_pool(instance);

3289 mfi_undo_cmd_pool:
3290         mrsas_free_cmd_pool(instance);

3292         return (DDI_FAILURE);
3293 }



3297 /*
3298  * get_ctrl_info
3299  */
3300 static int
3301 get_ctrl_info(struct mrsas_instance *instance,
3302     struct mrsas_ctrl_info *ctrl_info)
3303 {
3304         int     ret = 0;

3306         struct mrsas_cmd                *cmd;
3307         struct mrsas_dcmd_frame *dcmd;
3308         struct mrsas_ctrl_info  *ci;

3310         if (instance->tbolt) {
3311                 cmd = get_raid_msg_mfi_pkt(instance);
3312         } else {
3313                 cmd = get_mfi_pkt(instance);
3314         }

3316         if (!cmd) {
3317                 con_log(CL_ANN, (CE_WARN,
3318                     "Failed to get a cmd for ctrl info"));
3319                 DTRACE_PROBE2(info_mfi_err, uint16_t, instance->fw_outstanding,
3320                     uint16_t, instance->max_fw_cmds);
3321                 return (DDI_FAILURE);
3322         }

2388         cmd->retry_count_for_ocr = 0;
3324         /* Clear the frame buffer and assign back the context id */
3325         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3326         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3327             cmd->index);

3329         dcmd = &cmd->frame->dcmd;

3331         ci = (struct mrsas_ctrl_info *)instance->internal_buf;

3333         if (!ci) {
3334                 cmn_err(CE_WARN,
3335                     "Failed to alloc mem for ctrl info");
2399                 con_log(CL_ANN, (CE_WARN,
```



```
2400                     "Failed to alloc mem for ctrl info"));
3336                 return_mfi_pkt(instance, cmd);
3337                 return (DDI_FAILURE);
3338         }

3340         (void) memset(ci, 0, sizeof (struct mrsas_ctrl_info));

3342         /* for( i = 0; i < DCMD_MBOX_SZ; i++ ) dcmd->mbox.b[i] = 0; */
3343         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

3345         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3346         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status,
3347             MFI_CMD_STATUS_POLL_MODE);
3348         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3349         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3350             MFI_FRAME_DIR_READ);
3351         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3352         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3353             sizeof (struct mrsas_ctrl_info));
3354         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3355             MR_DCMD_CTRL_GET_INFO);
3356         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3357             instance->internal_buf_dmac_add);
3358         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3359             sizeof (struct mrsas_ctrl_info));

3361         cmd->frame_count = 1;

3363         if (instance->tbolt) {
3364                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3365         }

3367         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3368                 ret = 0;

3370                 ctrl_info->max_request_size = ddi_get32(
3371                     cmd->frame_dma_obj.acc_handle, &ci->max_request_size);

3373                 ctrl_info->ld_present_count = ddi_get16(
3374                     cmd->frame_dma_obj.acc_handle, &ci->ld_present_count);

3376                 ctrl_info->properties.on_off_properties = ddi_get32(
3377                     cmd->frame_dma_obj.acc_handle,
2437         ctrl_info->properties.on_off_properties =
2438             ddi_get32(cmd->frame_dma_obj.acc_handle,
3378                     &ci->properties.on_off_properties);

3379                 ddi_rep_get8(cmd->frame_dma_obj.acc_handle,
3380                     (uint8_t *)(ctrl_info->product_name),
3381                     (uint8_t *)(ci->product_name), 80 * sizeof (char),
3382                     DDI_DEV_AUTOINCR);
3383                 /* should get more members of ci with ddi_get when needed */
3384         } else {
3385                 cmn_err(CE_WARN, "get_ctrl_info: Ctrl info failed");
2447                 con_log(CL_ANN, (CE_WARN, "get_ctrl_info: Ctrl info failed"));
3386                 ret = -1;
3387         }

3389         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3390                 ret = -1;
3391         }
3392         if (instance->tbolt) {
3393                 return_raid_msg_mfi_pkt(instance, cmd);
3394         } else {
3395                 return_mfi_pkt(instance, cmd);
3396         }
```

```
3398         return (ret);
3399 }

3401 /*
3402  * abort_aen_cmd
3403  */
3404 static int
3405 abort_aen_cmd(struct mrsas_instance *instance,
3406     struct mrsas_cmd *cmd_to_abort)
3407 {
3408         int     ret = 0;

3410         struct mrsas_cmd                *cmd;
3411         struct mrsas_abort_frame        *abort_fr;

3413         con_log(CL_ANN1, (CE_NOTE, "chkpnt: abort_aen:%d", __LINE__));
2471         cmd = instance->cmd_list[APP_RESERVE_CMDS-1];

3415         if (instance->tbolt) {
3416                 cmd = get_raid_msg_mfi_pkt(instance);
3417         } else {
3418                 cmd = get_mfi_pkt(instance);
3419         }

3421         if (!cmd) {
3422                 con_log(CL_ANN1, (CE_WARN,
3423                     "abort_aen_cmd():Failed to get a cmd for abort_aen_cmd"));
3424                 DTRACE_PROBE2(abort_mfi_err, uint16_t, instance->fw_outstanding,
3425                     uint16_t, instance->max_fw_cmds);
3426                 return (DDI_FAILURE);
3427         }

2480         cmd->retry_count_for_ocr = 0;
3429         /* Clear the frame buffer and assign back the context id */
3430         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3431         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3432             cmd->index);

3434         abort_fr = &cmd->frame->abort;

3436         /* prepare and issue the abort frame */
3437         ddi_put8(cmd->frame_dma_obj.acc_handle,
3438             &abort_fr->cmd, MFI_CMD_OP_ABORT);
3439         ddi_put8(cmd->frame_dma_obj.acc_handle, &abort_fr->cmd_status,
3440             MFI_CMD_STATUS_SYNC_MODE);
3441         ddi_put16(cmd->frame_dma_obj.acc_handle, &abort_fr->flags, 0);
3442         ddi_put32(cmd->frame_dma_obj.acc_handle, &abort_fr->abort_context,
3443             cmd_to_abort->index);
3444         ddi_put32(cmd->frame_dma_obj.acc_handle,
3445             &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3446         ddi_put32(cmd->frame_dma_obj.acc_handle,
3447             &abort_fr->abort_mfi_phys_addr_hi, 0);

3449         instance->aen_cmd->abort_aen = 1;

2503         cmd->sync_cmd = MRSAS_TRUE;
3451         cmd->frame_count = 1;

3453         if (instance->tbolt) {
3454                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3455         }

3457         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3458                 con_log(CL_ANN1, (CE_WARN,
3459                     "abort_aen_cmd: issue_cmd_in_poll_mode failed"));
```



```
3460                 ret = -1;
3461         } else {
3462                 ret = 0;
3463         }

3465         instance->aen_cmd->abort_aen = 1;
3466         instance->aen_cmd = 0;

3468         if (instance->tbolt) {
3469                 return_raid_msg_mfi_pkt(instance, cmd);
3470         } else {
3471                 return_mfi_pkt(instance, cmd);
3472         }

3474         atomic_add_16(&instance->fw_outstanding, (-1));

3476         return (ret);
3477 }


2523 /*
2524  * init_mfi
2525  */
3480 static int
3481 mrsas_build_init_cmd(struct mrsas_instance *instance,
3482     struct mrsas_cmd **cmd_ptr)
2527 init_mfi(struct mrsas_instance *instance)
3483 {
3484         struct mrsas_cmd                *cmd;
2530         struct mrsas_ctrl_info          ctrl_info;
3485         struct mrsas_init_frame         *init_frame;
3486         struct mrsas_init_queue_info    *initq_info;
3487         struct mrsas_drv_ver            drv_ver_info;

2534         /* we expect the FW state to be READY */
2535         if (mfi_state_transition_to_ready(instance)) {
2536                 con_log(CL_ANN, (CE_WARN, "mr_sas: F/W is not ready"));
2537                 goto fail_ready_state;
2538         }

2540         /* get various operational parameters from status register */
2541         instance->max_num_sge =
2542             (instance->func_ptr->read_fw_status_reg(instance) &
2543             0xFF0000) >> 0x10;
3490         /*
2545          * Reduce the max supported cmds by 1. This is to ensure that the
2546          * reply_q_sz (1 more than the max cmd that driver may send)
2547          * does not exceed max cmds that the FW can support
2548          */
2549         instance->max_fw_cmds =
2550             instance->func_ptr->read_fw_status_reg(instance) & 0xFFFF;
2551         instance->max_fw_cmds = instance->max_fw_cmds - 1;

2553         instance->max_num_sge =
2554             (instance->max_num_sge > MRSAS_MAX_SGE_CNT) ?
2555             MRSAS_MAX_SGE_CNT : instance->max_num_sge;

2557         /* create a pool of commands */
2558         if (alloc_space_for_mfi(instance) != DDI_SUCCESS)
2559                 goto fail_alloc_fw_space;

2561         /*
3491          * Prepare a init frame. Note the init frame points to queue info
3492          * structure. Each frame has SGL allocated after first 64 bytes. For
3493          * this frame - since we don't need any SGL - we use SGL's space as
3494          * queue info structure
```

```
3495          */
3496         cmd = *cmd_ptr;
2567         cmd = get_mfi_pkt(instance);
2568         cmd->retry_count_for_ocr = 0;


3499         /* Clear the frame buffer and assign back the context id */
3500         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3501         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3502             cmd->index);

3504         init_frame = (struct mrsas_init_frame *)cmd->frame;
3505         initq_info = (struct mrsas_init_queue_info *)
3506             ((unsigned long)init_frame + 64);

3508         (void) memset(init_frame, 0, MRMFI_FRAME_SIZE);
3509         (void) memset(initq_info, 0, sizeof (struct mrsas_init_queue_info));

3511         ddi_put32(cmd->frame_dma_obj.acc_handle, &initq_info->init_flags, 0);

3513         ddi_put32(cmd->frame_dma_obj.acc_handle,
3514             &initq_info->reply_queue_entries, instance->max_fw_cmds + 1);

3516         ddi_put32(cmd->frame_dma_obj.acc_handle,
3517             &initq_info->producer_index_phys_addr_hi, 0);
3518         ddi_put32(cmd->frame_dma_obj.acc_handle,
3519             &initq_info->producer_index_phys_addr_lo,
3520             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address);

3522         ddi_put32(cmd->frame_dma_obj.acc_handle,
3523             &initq_info->consumer_index_phys_addr_hi, 0);
3524         ddi_put32(cmd->frame_dma_obj.acc_handle,
3525             &initq_info->consumer_index_phys_addr_lo,
3526             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 4);

3528         ddi_put32(cmd->frame_dma_obj.acc_handle,
3529             &initq_info->reply_queue_start_phys_addr_hi, 0);
3530         ddi_put32(cmd->frame_dma_obj.acc_handle,
3531             &initq_info->reply_queue_start_phys_addr_lo,
3532             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 8);

3534         ddi_put8(cmd->frame_dma_obj.acc_handle,
3535             &init_frame->cmd, MFI_CMD_OP_INIT);
3536         ddi_put8(cmd->frame_dma_obj.acc_handle, &init_frame->cmd_status,
3537             MFI_CMD_STATUS_POLL_MODE);
3538         ddi_put16(cmd->frame_dma_obj.acc_handle, &init_frame->flags, 0);
3539         ddi_put32(cmd->frame_dma_obj.acc_handle,
3540             &init_frame->queue_info_new_phys_addr_lo,
3541             cmd->frame_phys_addr + 64);
3542         ddi_put32(cmd->frame_dma_obj.acc_handle,
3543             &init_frame->queue_info_new_phys_addr_hi, 0);


3546         /* fill driver version information */
3547         fill_up_drv_ver(&drv_ver_info);

3549         /* allocate the driver version data transfer buffer */
3550         instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
3551         instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
3552         instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
3553         instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
3554         instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
3555         instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;

3557         if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
3558             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
```



```
3559                 con_log(CL_ANN, (CE_WARN,
3560                     "init_mfi : Could not allocate driver version buffer."));
3561                 return (DDI_FAILURE);
3562         }
3563         /* copy driver version to dma buffer */
3564         (void) memset(instance->drv_ver_dma_obj.buffer, 0,
3565             sizeof (drv_ver_info.drv_ver));
3566         ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
3567             (uint8_t *)drv_ver_info.drv_ver,
3568             (uint8_t *)instance->drv_ver_dma_obj.buffer,
3569             sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);


3572         /* copy driver version physical address to init frame */
3573         ddi_put64(cmd->frame_dma_obj.acc_handle, &init_frame->driverversion,
3574             instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);

3576         ddi_put32(cmd->frame_dma_obj.acc_handle, &init_frame->data_xfer_len,
3577             sizeof (struct mrsas_init_queue_info));

3579         cmd->frame_count = 1;

3581         *cmd_ptr = cmd;

3583         return (DDI_SUCCESS);
3584 }


3587 /*
3588  * mrsas_init_adapter_ppc - Initialize MFI interface adapter.
3589  */
3590 int
3591 mrsas_init_adapter_ppc(struct mrsas_instance *instance)
3592 {
3593         struct mrsas_cmd                *cmd;

3595         /*
3596          * allocate memory for mfi adapter(cmd pool, individual commands, mfi
3597          * frames etc
3598          */
3599         if (alloc_space_for_mfi(instance) != DDI_SUCCESS) {
3600                 con_log(CL_ANN, (CE_NOTE,
3601                     "Error, failed to allocate memory for MFI adapter"));
3602                 return (DDI_FAILURE);
3603         }

3605         /* Build INIT command */
3606         cmd = get_mfi_pkt(instance);

3608         if (mrsas_build_init_cmd(instance, &cmd) != DDI_SUCCESS) {
3609                 con_log(CL_ANN,
3610                     (CE_NOTE, "Error, failed to build INIT command"));

3612                 goto fail_undo_alloc_mfi_space;
3613         }

3615         /*
3616          * Disable interrupt before sending init frame ( see linux driver code)
3617          * send INIT MFI frame in polled mode
3618          */
2621         /* issue the init frame in polled mode */
3619         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3620                 con_log(CL_ANN, (CE_WARN, "failed to init firmware"));
2624                 return_mfi_pkt(instance, cmd);
3621                 goto fail_fw_init;
3622         }
```

```
3624         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
3628         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3629                 return_mfi_pkt(instance, cmd);
3625                 goto fail_fw_init;
2631         }
3626         return_mfi_pkt(instance, cmd);

3628         if (ctio_enable &&
3629             (instance->func_ptr->read_fw_status_reg(instance) & 0x04000000)) {
3630                 con_log(CL_ANN, (CE_NOTE, "mr_sas: IEEE SGL's supported"));
3631                 instance->flag_ieee = 1;
3632         } else {
3633                 instance->flag_ieee = 0;
3634         }

3636         instance->unroll.alloc_space_mfi = 1;
3637         instance->unroll.verBuff = 1;

3639         return (DDI_SUCCESS);


3642 fail_fw_init:
3643         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

3645 fail_undo_alloc_mfi_space:
3646         return_mfi_pkt(instance, cmd);
3647         free_space_for_mfi(instance);

3649         return (DDI_FAILURE);

3651 }

3653 /*
3654  * mrsas_init_adapter - Initialize adapter.
3655  */
3656 int
3657 mrsas_init_adapter(struct mrsas_instance *instance)
3658 {
3659         struct mrsas_ctrl_info          ctrl_info;


3662         /* we expect the FW state to be READY */
3663         if (mfi_state_transition_to_ready(instance)) {
3664                 con_log(CL_ANN, (CE_WARN, "mr_sas: F/W is not ready"));
3665                 return (DDI_FAILURE);
3666         }

3668         /* get various operational parameters from status register */
3669         instance->max_num_sge =
3670             (instance->func_ptr->read_fw_status_reg(instance) &
3671             0xFF0000) >> 0x10;
3672         instance->max_num_sge =
3673             (instance->max_num_sge > MRSAS_MAX_SGE_CNT) ?
3674             MRSAS_MAX_SGE_CNT : instance->max_num_sge;

3676         /*
3677          * Reduce the max supported cmds by 1. This is to ensure that the
3678          * reply_q_sz (1 more than the max cmd that driver may send)
3679          * does not exceed max cmds that the FW can support
3680          */
3681         instance->max_fw_cmds =
3682             instance->func_ptr->read_fw_status_reg(instance) & 0xFFFF;
3683         instance->max_fw_cmds = instance->max_fw_cmds - 1;
```



```
3687         /* Initialize adapter */
3688         if (instance->func_ptr->init_adapter(instance) != DDI_SUCCESS) {
3689                 con_log(CL_ANN,
3690                     (CE_WARN, "mr_sas: could not initialize adapter"));
3691                 return (DDI_FAILURE);
3692         }

3694         /* gather misc FW related information */
3695         instance->disable_online_ctrl_reset = 0;

2643         /* gather misc FW related information */
3697         if (!get_ctrl_info(instance, &ctrl_info)) {
3698                 instance->max_sectors_per_req = ctrl_info.max_request_size;
3699                 con_log(CL_ANN1, (CE_NOTE,
3700                     "product name %s ld present %d",
3701                     ctrl_info.product_name, ctrl_info.ld_present_count));
3702         } else {
3703                 instance->max_sectors_per_req = instance->max_num_sge *
3704                     PAGESIZE / 512;
3705         }

3707         if (ctrl_info.properties.on_off_properties & DISABLE_OCR_PROP_FLAG)
3708                 instance->disable_online_ctrl_reset = 1;

3710         return (DDI_SUCCESS);

2659 fail_fw_init:
2660 fail_alloc_fw_space:

2662         free_space_for_mfi(instance);

2664 fail_ready_state:
2665         ddi_regs_map_free(&instance->regmap_handle);

2667 fail_mfi_reg_setup:
2668         return (DDI_FAILURE);
3712 }



3716 static int
3717 mrsas_issue_init_mfi(struct mrsas_instance *instance)
3718 {
3719         struct mrsas_cmd                *cmd;
3720         struct mrsas_init_frame         *init_frame;
3721         struct mrsas_init_queue_info    *initq_info;

3723 /*
3724  * Prepare a init frame. Note the init frame points to queue info
3725  * structure. Each frame has SGL allocated after first 64 bytes. For
3726  * this frame - since we don't need any SGL - we use SGL's space as
3727  * queue info structure
3728  */
3729         con_log(CL_ANN1, (CE_NOTE,
3730             "mrsas_issue_init_mfi: entry\n"));
3731         cmd = get_mfi_app_pkt(instance);

3733         if (!cmd) {
3734                 con_log(CL_ANN1, (CE_WARN,
2694                 con_log(CL_ANN1, (CE_NOTE,
3735                     "mrsas_issue_init_mfi: get_pkt failed\n"));
3736                 return (DDI_FAILURE);
```

```
3737         }

3739         /* Clear the frame buffer and assign back the context id */
3740         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
3741         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3742             cmd->index);

3744         init_frame = (struct mrsas_init_frame *)cmd->frame;
3745         initq_info = (struct mrsas_init_queue_info *)
3746             ((unsigned long)init_frame + 64);

3748         (void) memset(init_frame, 0, MRMFI_FRAME_SIZE);
3749         (void) memset(initq_info, 0, sizeof (struct mrsas_init_queue_info));

3751         ddi_put32(cmd->frame_dma_obj.acc_handle, &initq_info->init_flags, 0);

3753         ddi_put32(cmd->frame_dma_obj.acc_handle,
3754             &initq_info->reply_queue_entries, instance->max_fw_cmds + 1);
3755         ddi_put32(cmd->frame_dma_obj.acc_handle,
3756             &initq_info->producer_index_phys_addr_hi, 0);
3757         ddi_put32(cmd->frame_dma_obj.acc_handle,
3758             &initq_info->producer_index_phys_addr_lo,
3759             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address);
3760         ddi_put32(cmd->frame_dma_obj.acc_handle,
3761             &initq_info->consumer_index_phys_addr_hi, 0);
3762         ddi_put32(cmd->frame_dma_obj.acc_handle,
3763             &initq_info->consumer_index_phys_addr_lo,
3764             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 4);

3766         ddi_put32(cmd->frame_dma_obj.acc_handle,
3767             &initq_info->reply_queue_start_phys_addr_hi, 0);
3768         ddi_put32(cmd->frame_dma_obj.acc_handle,
3769             &initq_info->reply_queue_start_phys_addr_lo,
3770             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address + 8);

3772         ddi_put8(cmd->frame_dma_obj.acc_handle,
3773             &init_frame->cmd, MFI_CMD_OP_INIT);
3774         ddi_put8(cmd->frame_dma_obj.acc_handle, &init_frame->cmd_status,
3775             MFI_CMD_STATUS_POLL_MODE);
3776         ddi_put16(cmd->frame_dma_obj.acc_handle, &init_frame->flags, 0);
3777         ddi_put32(cmd->frame_dma_obj.acc_handle,
3778             &init_frame->queue_info_new_phys_addr_lo,
3779             cmd->frame_phys_addr + 64);
3780         ddi_put32(cmd->frame_dma_obj.acc_handle,
3781             &init_frame->queue_info_new_phys_addr_hi, 0);

3783         ddi_put32(cmd->frame_dma_obj.acc_handle, &init_frame->data_xfer_len,
3784             sizeof (struct mrsas_init_queue_info));

3786         cmd->frame_count = 1;

3788         /* issue the init frame in polled mode */
3789         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3790                 con_log(CL_ANN1, (CE_WARN,
3791                     "mrsas_issue_init_mfi():failed to "
3792                     "init firmware"));
3793                 return_mfi_app_pkt(instance, cmd);
3794                 return (DDI_FAILURE);
3795         }

3797         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
3798                 return_mfi_pkt(instance, cmd);
3799                 return (DDI_FAILURE);
3800         }

3802         return_mfi_app_pkt(instance, cmd);
```



```
3803         con_log(CL_ANN1, (CE_CONT, "mrsas_issue_init_mfi: Done"));

2757         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_init_mfi: Done"));
3805         return (DDI_SUCCESS);
3806 }
3807 /*
3808  * mfi_state_transition_to_ready        : Move the FW to READY state
3809  *
3810  * @reg_set                    : MFI register set
3811  */
3812 int
2765 static int
3813 mfi_state_transition_to_ready(struct mrsas_instance *instance)
3814 {
3815         int             i;
3816         uint8_t         max_wait;
3817         uint32_t        fw_ctrl = 0;
2770         uint32_t        fw_ctrl;
3818         uint32_t        fw_state;
3819         uint32_t        cur_state;
3820         uint32_t        cur_abs_reg_val;
3821         uint32_t        prev_abs_reg_val;
3822         uint32_t        status;

3824         cur_abs_reg_val =
3825             instance->func_ptr->read_fw_status_reg(instance);
3826         fw_state =
3827             cur_abs_reg_val & MFI_STATE_MASK;
3828         con_log(CL_ANN1, (CE_CONT,
2780         con_log(CL_ANN1, (CE_NOTE,
3829             "mfi_state_transition_to_ready:FW state = 0x%x", fw_state));

3831         while (fw_state != MFI_STATE_READY) {
3832                 con_log(CL_ANN, (CE_CONT,
2784                 con_log(CL_ANN, (CE_NOTE,
3833                     "mfi_state_transition_to_ready:FW state%x", fw_state));

3835                 switch (fw_state) {
3836                 case MFI_STATE_FAULT:
3837                         con_log(CL_ANN, (CE_NOTE,
2789                         con_log(CL_ANN1, (CE_NOTE,
3838                             "mr_sas: FW in FAULT state!!"));

3840                         return (ENODEV);
3841                 case MFI_STATE_WAIT_HANDSHAKE:
3842                         /* set the CLR bit in IMR0 */
3843                         con_log(CL_ANN1, (CE_NOTE,
3844                             "mr_sas: FW waiting for HANDSHAKE"));
3845                         /*
3846                          * PCI_Hot Plug: MFI F/W requires
3847                          * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3848                          * to be set
3849                          */
3850                         /* WR_IB_MSG_0(MFI_INIT_CLEAR_HANDSHAKE, instance); */
3851                         if (!instance->tbolt) {
3852                                 WR_IB_DOORBELL(MFI_INIT_CLEAR_HANDSHAKE |
3853                                     MFI_INIT_HOTPLUG, instance);
3854                         } else {
3855                                 WR_RESERVED0_REGISTER(MFI_INIT_CLEAR_HANDSHAKE |
3856                                     MFI_INIT_HOTPLUG, instance);
3857                         }
3858                         max_wait        = (instance->tbolt == 1) ? 180 : 2;

2806                         max_wait        = 2;
3859                         cur_state       = MFI_STATE_WAIT_HANDSHAKE;
3860                         break;
```

```
3861                case MFI_STATE_BOOT_MESSAGE_PENDING:
3862                        /* set the CLR bit in IMR0 */
3863                        con_log(CL_ANN1, (CE_NOTE,
3864                            "mr_sas: FW state boot message pending"));
3865                        /*
3866                         * PCI_Hot Plug: MFI F/W requires
3867                         * (MFI_INIT_CLEAR_HANDSHAKE|MFI_INIT_HOTPLUG)
3868                         * to be set
3869                         */
3870                        if (!instance->tbolt) {
3871                                WR_IB_DOORBELL(MFI_INIT_HOTPLUG, instance);
3872                        } else {
3873                                WR_RESERVED0_REGISTER(MFI_INIT_HOTPLUG,
3874                                    instance);
3875                        }
3876                        max_wait        = (instance->tbolt == 1) ? 180 : 10;

2820                    max_wait        = 10;
3877                        cur_state       = MFI_STATE_BOOT_MESSAGE_PENDING;
3878                        break;
3879                case MFI_STATE_OPERATIONAL:
3880                        /* bring it to READY state; assuming max wait 2 secs */
3881                        instance->func_ptr->disable_intr(instance);
3882                        con_log(CL_ANN1, (CE_NOTE,
3883                            "mr_sas: FW in OPERATIONAL state"));
3884                        /*
3885                         * PCI_Hot Plug: MFI F/W requires
3886                         * (MFI_INIT_READY | MFI_INIT_MFIMODE | MFI_INIT_ABORT)
3887                         * to be set
3888                         */
3889                        /* WR_IB_DOORBELL(MFI_INIT_READY, instance); */
3890                        if (!instance->tbolt) {
3891                                WR_IB_DOORBELL(MFI_RESET_FLAGS, instance);
3892                        } else {
3893                                WR_RESERVED0_REGISTER(MFI_RESET_FLAGS,
3894                                    instance);

3896                                for (i = 0; i < (10 * 1000); i++) {
3897                                        status =
3898                                            RD_RESERVED0_REGISTER(instance);
3899                                        if (status & 1) {
3900                                                delay(1 *
3901                                                    drv_usectohz(MILLISEC));
3902                                        } else {
3903                                                break;
3904                                        }
3905                                }

3907                        }
3908                        max_wait        = (instance->tbolt == 1) ? 180 : 10;
2836                    max_wait        = 10;
3909                        cur_state       = MFI_STATE_OPERATIONAL;
3910                        break;
3911                case MFI_STATE_UNDEFINED:
3912                        /* this state should not last for more than 2 seconds */
3913                        con_log(CL_ANN1, (CE_NOTE, "FW state undefined"));

3915                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
2843                    max_wait        = 2;
3916                        cur_state       = MFI_STATE_UNDEFINED;
3917                        break;
3918                case MFI_STATE_BB_INIT:
3919                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
2847                    max_wait        = 2;
3920                        cur_state       = MFI_STATE_BB_INIT;
3921                        break;
```



```
3922                case MFI_STATE_FW_INIT:
3923                        max_wait        = (instance->tbolt == 1) ? 180 : 2;
2851                    max_wait        = 2;
3924                        cur_state       = MFI_STATE_FW_INIT;
3925                        break;
3926                case MFI_STATE_FW_INIT_2:
3927                        max_wait        = 180;
3928                        cur_state       = MFI_STATE_FW_INIT_2;
3929                        break;
3930                case MFI_STATE_DEVICE_SCAN:
3931                        max_wait        = 180;
3932                        cur_state       = MFI_STATE_DEVICE_SCAN;
3933                        prev_abs_reg_val = cur_abs_reg_val;
3934                        con_log(CL_NONE, (CE_NOTE,
3935                            "Device scan in progress ...\n"));
3936                        break;
3937                case MFI_STATE_FLUSH_CACHE:
3938                        max_wait        = 180;
3939                        cur_state       = MFI_STATE_FLUSH_CACHE;
3940                        break;
3941                default:
3942                        con_log(CL_ANN1, (CE_NOTE,
3943                            "mr_sas: Unknown state 0x%x", fw_state));
3944                        return (ENODEV);
3945                }

3947                /* the cur_state should not last for more than max_wait secs */
3948                for (i = 0; i < (max_wait * MILLISEC); i++) {
3949                        /* fw_state = RD_OB_MSG_0(instance) & MFI_STATE_MASK; */
3950                        cur_abs_reg_val =
3951                            instance->func_ptr->read_fw_status_reg(instance);
3952                        fw_state = cur_abs_reg_val & MFI_STATE_MASK;

3954                        if (fw_state == cur_state) {
3955                                delay(1 * drv_usectohz(MILLISEC));
3956                        } else {
3957                                break;
3958                        }
3959                }
3960                if (fw_state == MFI_STATE_DEVICE_SCAN) {
3961                        if (prev_abs_reg_val != cur_abs_reg_val) {
3962                                continue;
3963                        }
3964                }

3966                /* return error if fw_state hasn't changed after max_wait */
3967                if (fw_state == cur_state) {
3968                        con_log(CL_ANN1, (CE_WARN,
2888                        con_log(CL_ANN1, (CE_NOTE,
3969                            "FW state hasn't changed in %d secs", max_wait));
3970                        return (ENODEV);
3971                }
3972        };

3974        if (!instance->tbolt) {
3975                fw_ctrl = RD_IB_DOORBELL(instance);
3976                con_log(CL_ANN1, (CE_CONT,

2896        con_log(CL_ANN1, (CE_NOTE,
3977                    "mfi_state_transition_to_ready:FW ctrl = 0x%x", fw_ctrl));

3979                /*
3980                 * Write 0xF to the doorbell register to do the following.
3981                 * - Abort all outstanding commands (bit 0).
3982                 * - Transition from OPERATIONAL to READY state (bit 1).
3983                 * - Discard (possible) low MFA posted in 64-bit mode (bit-2).
```

```
3984                  * - Set to release FW to continue running (i.e. BIOS handshake
3985                  *   (bit 3).
3986                  */
3987                 WR_IB_DOORBELL(0xF, instance);
3988         }

3990         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
3991                 return (EIO);
2910                 return (ENODEV);
3992         }

3994         return (DDI_SUCCESS);
3995 }

3997 /*
3998  * get_seq_num
3999  */
4000 static int
4001 get_seq_num(struct mrsas_instance *instance,
4002     struct mrsas_evt_log_info *eli)
4003 {
4004         int     ret = DDI_SUCCESS;

4006         dma_obj_t                       dcmd_dma_obj;
4007         struct mrsas_cmd                *cmd;
4008         struct mrsas_dcmd_frame         *dcmd;
4009         struct mrsas_evt_log_info *eli_tmp;
4010         if (instance->tbolt) {
4011                 cmd = get_raid_msg_mfi_pkt(instance);
4012         } else {
4013                 cmd = get_mfi_pkt(instance);
4014         }

4016         if (!cmd) {
4017                 cmn_err(CE_WARN, "mr_sas: failed to get a cmd");
4018                 DTRACE_PROBE2(seq_num_mfi_err, uint16_t,
4019                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
4020                 return (ENOMEM);
4021         }

2936         cmd->retry_count_for_ocr = 0;
4023         /* Clear the frame buffer and assign back the context id */
4024         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4025         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4026             cmd->index);

4028         dcmd = &cmd->frame->dcmd;

4030         /* allocate the data transfer buffer */
4031         dcmd_dma_obj.size = sizeof (struct mrsas_evt_log_info);
4032         dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
4033         dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
4034         dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
4035         dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
4036         dcmd_dma_obj.dma_attr.dma_attr_align = 1;

4038         if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
4039             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
4040                 cmn_err(CE_WARN,
4041                     "get_seq_num: could not allocate data transfer buffer.");
2954                 con_log(CL_ANN, (CE_WARN,
2955                     "get_seq_num: could not allocate data transfer buffer."));
4042                 return (DDI_FAILURE);
4043         }

4045         (void) memset(dcmd_dma_obj.buffer, 0,
```



```
4046             sizeof (struct mrsas_evt_log_info));

4048         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4050         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
4051         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
4052         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
4053         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4054             MFI_FRAME_DIR_READ);
4055         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4056         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
4057             sizeof (struct mrsas_evt_log_info));
4058         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
4059             MR_DCMD_CTRL_EVENT_GET_INFO);
4060         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
4061             sizeof (struct mrsas_evt_log_info));
4062         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
4063             dcmd_dma_obj.dma_cookie[0].dmac_address);

4065         cmd->sync_cmd = MRSAS_TRUE;
4066         cmd->frame_count = 1;

4068         if (instance->tbolt) {
4069                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4070         }

4072         if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
4073                 cmn_err(CE_WARN, "get_seq_num: "
4074                     "failed to issue MRSAS_DCMD_CTRL_EVENT_GET_INFO");
4075                 ret = DDI_FAILURE;
4076         } else {
4077                 eli_tmp = (struct mrsas_evt_log_info *)dcmd_dma_obj.buffer;
4078                 eli->newest_seq_num = ddi_get32(cmd->frame_dma_obj.acc_handle,
4079                     &eli_tmp->newest_seq_num);
4080                 ret = DDI_SUCCESS;
4081         }

4083         if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
4084                 ret = DDI_FAILURE;

4086         if (instance->tbolt) {
4087                 return_raid_msg_mfi_pkt(instance, cmd);
4088         } else {
4089                 return_mfi_pkt(instance, cmd);
2996         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS) {
2997                 ret = DDI_FAILURE;
4090         }

3000         return_mfi_pkt(instance, cmd);

4092         return (ret);
4093 }

4095 /*
4096  * start_mfi_aen
4097  */
4098 static int
4099 start_mfi_aen(struct mrsas_instance *instance)
4100 {
4101         int     ret = 0;

4103         struct mrsas_evt_log_info       eli;
4104         union mrsas_evt_class_locale    class_locale;

4106         /* get the latest sequence number from FW */
4107         (void) memset(&eli, 0, sizeof (struct mrsas_evt_log_info));
```

```
4109         if (get_seq_num(instance, &eli)) {
4110                 cmn_err(CE_WARN, "start_mfi_aen: failed to get seq num");
4111                 return (-1);
4112         }

4114         /* register AEN with FW for latest sequence number plus 1 */
4115         class_locale.members.reserved   = 0;
4116         class_locale.members.locale     = LE_16(MR_EVT_LOCALE_ALL);
4117         class_locale.members.class      = MR_EVT_CLASS_INFO;
4118         class_locale.word       = LE_32(class_locale.word);
4119         ret = register_mfi_aen(instance, eli.newest_seq_num + 1,
4120             class_locale.word);

4122         if (ret) {
4123                 cmn_err(CE_WARN, "start_mfi_aen: aen registration failed");
4124                 return (-1);
4125         }


4128         return (ret);
4129 }

4131 /*
4132  * flush_cache
4133  */
4134 static void
4135 flush_cache(struct mrsas_instance *instance)
4136 {
4137         struct mrsas_cmd                *cmd = NULL;
4138         struct mrsas_dcmd_frame         *dcmd;
4139         if (instance->tbolt) {
4140                 cmd = get_raid_msg_mfi_pkt(instance);
4141         } else {
4142                 cmd = get_mfi_pkt(instance);
4143         }
3048         uint32_t        max_cmd = instance->max_fw_cmds;

3050         cmd = instance->cmd_list[max_cmd];

4145         if (!cmd) {
4146                 con_log(CL_ANN1, (CE_WARN,
4147                     "flush_cache():Failed to get a cmd for flush_cache"));
4148                 DTRACE_PROBE2(flush_cache_err, uint16_t,
4149                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
4150                 return;
4151         }

3059         cmd->retry_count_for_ocr = 0;
4153         /* Clear the frame buffer and assign back the context id */
4154         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
4155         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
4156             cmd->index);

4158         dcmd = &cmd->frame->dcmd;

4160         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

4162         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
4163         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
4164         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 0);
4165         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
4166             MFI_FRAME_DIR_NONE);
4167         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
4168         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len, 0);
4169         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
```



```
4170             MR_DCMD_CTRL_CACHE_FLUSH);
4171         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.b[0],
4172             MR_FLUSH_CTRL_CACHE | MR_FLUSH_DISK_CACHE);

4174         cmd->frame_count = 1;

4176         if (instance->tbolt) {
4177                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
4178         }

4180         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
4181                 con_log(CL_ANN1, (CE_WARN,
4182             "flush_cache: failed to issue MFI_DCMD_CTRL_CACHE_FLUSH"));
4183         }
4184         con_log(CL_ANN1, (CE_CONT, "flush_cache done"));
4185         if (instance->tbolt) {
4186                 return_raid_msg_mfi_pkt(instance, cmd);
4187         } else {
4188                 return_mfi_pkt(instance, cmd);
4189         }

3087         con_log(CL_ANN1, (CE_NOTE, "flush_cache done"));
4191 }

4193 /*
4194  * service_mfi_aen-     Completes an AEN command
4195  * @instance:                  Adapter soft state
4196  * @cmd:                       Command to be completed
4197  *
4198  */
4199 void
3096 static void
4200 service_mfi_aen(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
4201 {
4202         uint32_t        seq_num;
4203         struct mrsas_evt_detail *evt_detail =
4204             (struct mrsas_evt_detail *)instance->mfi_evt_detail_obj.buffer;
4205         int             rval = 0;
4206         int             tgt = 0;
4207         uint8_t         dtype;
4208 #ifdef PDSUPPORT
4209         mrsas_pd_address_t      *pd_addr;
4210 #endif
4211         ddi_acc_handle_t                acc_handle;

4213         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

4215         acc_handle = cmd->frame_dma_obj.acc_handle;

4216         cmd->cmd_status = ddi_get8(acc_handle, &cmd->frame->io.cmd_status);

4217         if (cmd->cmd_status == ENODATA) {
4218                 cmd->cmd_status = 0;
4219         }

4221         /*
4222          * log the MFI AEN event to the sysevent queue so that
4223          * application will get noticed
4224          */
4225         if (ddi_log_sysevent(instance->dip, DDI_VENDOR_LSI, "LSIMEGA", "SAS",
4226             NULL, NULL, DDI_NOSLEEP) != DDI_SUCCESS) {
4227                 int     instance_no = ddi_get_instance(instance->dip);
4228                 con_log(CL_ANN, (CE_WARN,
4229                     "mr_sas%d: Failed to log AEN event", instance_no));
4230         }
4231         /*
```

```
4232          * Check for any ld devices that has changed state. i.e. online
4233          * or offline.
4234          */
4235         con_log(CL_ANN1, (CE_CONT,
3128         con_log(CL_ANN1, (CE_NOTE,
4236             "AEN: code = %x class = %x locale = %x args = %x",
4237             ddi_get32(acc_handle, &evt_detail->code),
4238             evt_detail->cl.members.class,
4239             ddi_get16(acc_handle, &evt_detail->cl.members.locale),
4240             ddi_get8(acc_handle, &evt_detail->arg_type)));

4242         switch (ddi_get32(acc_handle, &evt_detail->code)) {
4243         case MR_EVT_CFG_CLEARED: {
4244                 for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
4245                         if (instance->mr_ld_list[tgt].dip != NULL) {
4246                                 mutex_enter(&instance->config_dev_mtx);
4247                                 instance->mr_ld_list[tgt].flag =
4248                                     (uint8_t)~MRDRV_TGT_VALID;
4249                                 mutex_exit(&instance->config_dev_mtx);
4250                                 rval = mrsas_service_evt(instance, tgt, 0,
4251                                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4252                                 con_log(CL_ANN1, (CE_WARN,
4253                                     "mr_sas: CFG CLEARED AEN rval = %d "
4254                                     "tgt id = %d", rval, tgt));
4255                         }
4256                 }
4257                 break;
4258         }

4260         case MR_EVT_LD_DELETED: {
4261                 tgt = ddi_get16(acc_handle, &evt_detail->args.ld.target_id);
4262                 mutex_enter(&instance->config_dev_mtx);
4263                 instance->mr_ld_list[tgt].flag = (uint8_t)~MRDRV_TGT_VALID;
4264                 mutex_exit(&instance->config_dev_mtx);
4265                 rval = mrsas_service_evt(instance,
4266                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4267                     MRSAS_EVT_UNCONFIG_TGT, NULL);
4268                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD DELETED AEN rval = %d "
4269                     "tgt id = %d index = %d", rval,
4270                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4271                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4272                 break;
4273         } /* End of MR_EVT_LD_DELETED */

4275         case MR_EVT_LD_CREATED: {
4276                 rval = mrsas_service_evt(instance,
4277                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id), 0,
4278                     MRSAS_EVT_CONFIG_TGT, NULL);
4279                 con_log(CL_ANN1, (CE_WARN, "mr_sas: LD CREATED AEN rval = %d "
4280                     "tgt id = %d index = %d", rval,
4281                     ddi_get16(acc_handle, &evt_detail->args.ld.target_id),
4282                     ddi_get8(acc_handle, &evt_detail->args.ld.ld_index)));
4283                 break;
4284         } /* End of MR_EVT_LD_CREATED */

4286 #ifdef PDSUPPORT
4287         case MR_EVT_PD_REMOVED_EXT: {
4288                 if (instance->tbolt) {
4289                         pd_addr = &evt_detail->args.pd_addr;
4290                         dtype = pd_addr->scsi_dev_type;
4291                         con_log(CL_DLEVEL1, (CE_NOTE,
4292                             " MR_EVT_PD_REMOVED_EXT: dtype = %x,"
4293                             " arg_type = %d ", dtype, evt_detail->arg_type));
4294                         tgt = ddi_get16(acc_handle,
4295                             &evt_detail->args.pd.device_id);
4296                         mutex_enter(&instance->config_dev_mtx);
```



```
4297                         instance->mr_tbolt_pd_list[tgt].flag =
4298                             (uint8_t)~MRDRV_TGT_VALID;
4299                         mutex_exit(&instance->config_dev_mtx);
4300                         rval = mrsas_service_evt(instance, ddi_get16(
4301                             acc_handle, &evt_detail->args.pd.device_id),
4302                             1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4303                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4304                             "rval = %d tgt id = %d ", rval,
4305                             ddi_get16(acc_handle,
4306                             &evt_detail->args.pd.device_id)));
4307                 }
4308                 break;
4309         } /* End of MR_EVT_PD_REMOVED_EXT */

4311         case MR_EVT_PD_INSERTED_EXT: {
4312                 if (instance->tbolt) {
4313                         rval = mrsas_service_evt(instance,
4314                             ddi_get16(acc_handle,
4315                             &evt_detail->args.pd.device_id),
4316                             1, MRSAS_EVT_CONFIG_TGT, NULL);
4317                         con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_INSERTEDi_EXT:"
4318                             "rval = %d tgt id = %d ", rval,
4319                             ddi_get16(acc_handle,
4320                             &evt_detail->args.pd.device_id)));
4321                 }
4322                 break;
4323         } /* End of MR_EVT_PD_INSERTED_EXT */

4325         case MR_EVT_PD_STATE_CHANGE: {
4326                 if (instance->tbolt) {
4327                         tgt = ddi_get16(acc_handle,
4328                             &evt_detail->args.pd.device_id);
4329                         if ((evt_detail->args.pd_state.prevState ==
4330                             PD_SYSTEM) &&
4331                             (evt_detail->args.pd_state.newState != PD_SYSTEM)) {
4332                                 mutex_enter(&instance->config_dev_mtx);
4333                                 instance->mr_tbolt_pd_list[tgt].flag =
4334                                     (uint8_t)~MRDRV_TGT_VALID;
4335                                 mutex_exit(&instance->config_dev_mtx);
4336                                 rval = mrsas_service_evt(instance,
4337                                     ddi_get16(acc_handle,
4338                                     &evt_detail->args.pd.device_id),
4339                                     1, MRSAS_EVT_UNCONFIG_TGT, NULL);
4340                                 con_log(CL_ANN1, (CE_WARN, "mr_sas: PD_REMOVED:"
4341                                     "rval = %d tgt id = %d ", rval,
4342                                     ddi_get16(acc_handle,
4343                                     &evt_detail->args.pd.device_id)));
4344                                 break;
4345                         }
4346                         if ((evt_detail->args.pd_state.prevState
4347                             == UNCONFIGURED_GOOD) &&
4348                             (evt_detail->args.pd_state.newState == PD_SYSTEM)) {
4349                                 rval = mrsas_service_evt(instance,
4350                                     ddi_get16(acc_handle,
4351                                     &evt_detail->args.pd.device_id),
4352                                     1, MRSAS_EVT_CONFIG_TGT, NULL);
4353                                 con_log(CL_ANN1, (CE_WARN,
4354                                     "mr_sas: PD_INSERTED: rval = %d "
4355                                     " tgt id = %d ", rval,
4356                                     ddi_get16(acc_handle,
4357                                     &evt_detail->args.pd.device_id)));
4358                                 break;
4359                         }
4360                 }
4361                 break;
4362         }
```

```
#endif
        } /* End of Main Switch */

        /* get copy of seq_num and class/locale for re-registration */
        seq_num = ddi_get32(acc_handle, &evt_detail->seq_num);
        seq_num++;
        (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
            sizeof (struct mrsas_evt_detail));

        ddi_put8(acc_handle, &cmd->frame->dcmd.cmd_status, 0x0);
        ddi_put32(acc_handle, &cmd->frame->dcmd.mbox.w[0], seq_num);

        instance->aen_seq_num = seq_num;

        cmd->frame_count = 1;

        cmd->retry_count_for_ocr = 0;
        cmd->drv_pkt_time = 0;

        /* Issue the aen registration frame */
        instance->func_ptr->issue_cmd(cmd, instance);
}

/*
 * complete_cmd_in_sync_mode -  Completes an internal command
 * @instance:                   Adapter soft state
 * @cmd:                        Command to be completed
 *
 * The issue_cmd_in_sync_mode() function waits for a command to complete
 * after it issues a command. This function wakes up that waiting routine by
 * calling wake_up() on the wait queue.
 */
static void
complete_cmd_in_sync_mode(struct mrsas_instance *instance,
    struct mrsas_cmd *cmd)
{
        cmd->cmd_status = ddi_get8(cmd->frame_dma_obj.acc_handle,
            &cmd->frame->io.cmd_status);

        cmd->sync_cmd = MRSAS_FALSE;

        con_log(CL_ANN1, (CE_NOTE, "complete_cmd_in_sync_mode called %p \n",
            (void *)cmd));

        mutex_enter(&instance->int_cmd_mtx);
        if (cmd->cmd_status == ENODATA) {
                cmd->cmd_status = 0;
        }
        cv_broadcast(&instance->int_cmd_cv);
        mutex_exit(&instance->int_cmd_mtx);

        con_log(CL_ANN1, (CE_NOTE, "complete_cmd_in_sync_mode called %p \n",
            (void *)cmd));

        cv_broadcast(&instance->int_cmd_cv);
}

/*
 * Call this function inside mrsas_softintr.
 * mrsas_initiate_ocr_if_fw_is_faulty  - Initiates OCR if FW status is faulty
 * @instance:                   Adapter soft state
 */

static uint32_t
mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *instance)
```



```
{
        uint32_t        cur_abs_reg_val;
        uint32_t        fw_state;

        cur_abs_reg_val =  instance->func_ptr->read_fw_status_reg(instance);
        fw_state = cur_abs_reg_val & MFI_STATE_MASK;
        if (fw_state == MFI_STATE_FAULT) {

                if (instance->disable_online_ctrl_reset == 1) {
                        cmn_err(CE_WARN,
                con_log(CL_ANN1, (CE_NOTE,
                                "mrsas_initiate_ocr_if_fw_is_faulty: "
                                "FW in Fault state, detected in ISR: "
                                "FW doesn't support ocr ");

                    "FW doesn't support ocr "));
                        return (ADAPTER_RESET_NOT_REQUIRED);
                } else {
                        con_log(CL_ANN, (CE_NOTE,
                            "mrsas_initiate_ocr_if_fw_is_faulty: FW in Fault "
                            "state, detected in ISR: FW supports ocr "));

                con_log(CL_ANN1, (CE_NOTE,
                    "mrsas_initiate_ocr_if_fw_is_faulty: "
                    "FW in Fault state, detected in ISR: FW supports ocr "));
                        return (ADAPTER_RESET_REQUIRED);
                }
        }

        return (ADAPTER_RESET_NOT_REQUIRED);
}

/*
 * mrsas_softintr - The Software ISR
 * @param arg   : HBA soft state
 *
 * called from high-level interrupt if hi-level interrupt are not there,
 * otherwise triggered as a soft interrupt
 */
static uint_t
mrsas_softintr(struct mrsas_instance *instance)
{
        struct scsi_pkt         *pkt;
        struct scsa_cmd         *acmd;
        struct mrsas_cmd        *cmd;
        struct mlist_head       *pos, *next;
        mlist_t                 process_list;
        struct mrsas_header     *hdr;
        struct scsi_arq_status  *arqstat;

        con_log(CL_ANN1, (CE_NOTE, "mrsas_softintr() called."));
        con_log(CL_ANN1, (CE_CONT, "mrsas_softintr called"));

        ASSERT(instance);

        mutex_enter(&instance->completed_pool_mtx);

        if (mlist_empty(&instance->completed_pool_list)) {
                mutex_exit(&instance->completed_pool_mtx);
                return (DDI_INTR_CLAIMED);
        }

        instance->softint_running = 1;

        INIT_LIST_HEAD(&process_list);
        mlist_splice(&instance->completed_pool_list, &process_list);
```

```
4484         INIT_LIST_HEAD(&instance->completed_pool_list);

4486         mutex_exit(&instance->completed_pool_mtx);

4488         /* perform all callbacks first, before releasing the SCBs */
4489         mlist_for_each_safe(pos, next, &process_list) {
4490                 cmd = mlist_entry(pos, struct mrsas_cmd, list);

4492                 /* syncronize the Cmd frame for the controller */
4493                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle,
4494                     0, 0, DDI_DMA_SYNC_FORCPU);

4496                 if (mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
4497                     DDI_SUCCESS) {
4498                         mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
4499                         ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4500                         con_log(CL_ANN1, (CE_WARN,
4501                             "mrsas_softintr: "
4502                             "FMA check reports DMA handle failure"));
4503                         return (DDI_INTR_CLAIMED);
4504                 }

4506                 hdr = &cmd->frame->hdr;

4508                 /* remove the internal command from the process list */
4509                 mlist_del_init(&cmd->list);

4511                 switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
4512                 case MFI_CMD_OP_PD_SCSI:
4513                 case MFI_CMD_OP_LD_SCSI:
4514                 case MFI_CMD_OP_LD_READ:
4515                 case MFI_CMD_OP_LD_WRITE:
4516                         /*
4517                          * MFI_CMD_OP_PD_SCSI and MFI_CMD_OP_LD_SCSI
4518                          * could have been issued either through an
4519                          * IO path or an IOCTL path. If it was via IOCTL,
4520                          * we will send it to internal completion.
4521                          */
4522                         if (cmd->sync_cmd == MRSAS_TRUE) {
4523                                 complete_cmd_in_sync_mode(instance, cmd);
4524                                 break;
4525                         }

4527                         /* regular commands */
4528                         acmd =  cmd->cmd;
4529                         pkt =   CMD2PKT(acmd);

4531                         if (acmd->cmd_flags & CFLAG_DMAVALID) {
4532                                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
4533                                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
4534                                             acmd->cmd_dma_offset,
4535                                             acmd->cmd_dma_len,
4536                                             DDI_DMA_SYNC_FORCPU);
4537                                 }
4538                         }

4540                         pkt->pkt_reason         = CMD_CMPLT;
4541                         pkt->pkt_statistics     = 0;
4542                         pkt->pkt_state = STATE_GOT_BUS
4543                             | STATE_GOT_TARGET | STATE_SENT_CMD
4544                             | STATE_XFERRED_DATA | STATE_GOT_STATUS;

4546                         con_log(CL_ANN, (CE_CONT,
3344                         con_log(CL_ANN1, (CE_CONT,
4547                             "CDB[0] = %x completed for %s: size %lx context %x",
4548                             pkt->pkt_cdbp[0], ((acmd->islogical) ? "LD" : "PD"),
```



```
4549                             acmd->cmd_dmacount, hdr->context));
4550                         DTRACE_PROBE3(softintr_cdb, uint8_t, pkt->pkt_cdbp[0],
4551                             uint_t, acmd->cmd_cdblen, ulong_t,
4552                             acmd->cmd_dmacount);

4554                         if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
4555                                 struct scsi_inquiry     *inq;

4557                                 if (acmd->cmd_dmacount != 0) {
4558                                         bp_mapin(acmd->cmd_buf);
4559                                         inq = (struct scsi_inquiry *)
4560                                             acmd->cmd_buf->b_un.b_addr;

4562                                         /* don't expose physical drives to OS */
4563                                         if (acmd->islogical &&
4564                                             (hdr->cmd_status == MFI_STAT_OK)) {
4565                                                 display_scsi_inquiry(
4566                                                     (caddr_t)inq);
4567                                         } else if ((hdr->cmd_status ==
4568                                             MFI_STAT_OK) && inq->inq_dtype ==
4569                                             DTYPE_DIRECT) {

4571                                                 display_scsi_inquiry(
4572                                                     (caddr_t)inq);

4574                                                 /* for physical disk */
4575                                                 hdr->cmd_status =
4576                                                     MFI_STAT_DEVICE_NOT_FOUND;
4577                                         }
4578                                 }
4579                         }

4581                         DTRACE_PROBE2(softintr_done, uint8_t, hdr->cmd,
4582                             uint8_t, hdr->cmd_status);

4584                         switch (hdr->cmd_status) {
4585                         case MFI_STAT_OK:
4586                                 pkt->pkt_scbp[0] = STATUS_GOOD;
4587                                 break;
4588                         case MFI_STAT_LD_CC_IN_PROGRESS:
4589                         case MFI_STAT_LD_RECON_IN_PROGRESS:
4590                                 pkt->pkt_scbp[0] = STATUS_GOOD;
4591                                 break;
4592                         case MFI_STAT_LD_INIT_IN_PROGRESS:
4593                                 con_log(CL_ANN,
4594                                     (CE_WARN, "Initialization in Progress"));
4595                                 pkt->pkt_reason = CMD_TRAN_ERR;

4597                                 break;
4598                         case MFI_STAT_SCSI_DONE_WITH_ERROR:
4599                                 con_log(CL_ANN, (CE_CONT, "scsi_done error"));
3397                                 con_log(CL_ANN1, (CE_CONT, "scsi_done error"));

4601                                 pkt->pkt_reason = CMD_CMPLT;
4602                                 ((struct scsi_status *)
4603                                     pkt->pkt_scbp)->sts_chk = 1;

4605                                 if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {

4606                                         con_log(CL_ANN,
4607                                             (CE_WARN, "TEST_UNIT_READY fail"));

4608                                 } else {
4609                                         pkt->pkt_state |= STATE_ARQ_DONE;
4610                                         arqstat = (void *)(pkt->pkt_scbp);
4611                                         arqstat->sts_rqpkt_reason = CMD_CMPLT;
```

```
4612                                    arqstat->sts_rqpkt_resid = 0;
4613                                    arqstat->sts_rqpkt_state |=
4614                                        STATE_GOT_BUS | STATE_GOT_TARGET
4615                                        | STATE_SENT_CMD
4616                                        | STATE_XFERRED_DATA;
4617                                    *(uint8_t *)&arqstat->sts_rqpkt_status =
4618                                        STATUS_GOOD;
4619                                    ddi_rep_get8(
4620                                        cmd->frame_dma_obj.acc_handle,
4621                                        (uint8_t *)
4622                                        &(arqstat->sts_sensedata),
4623                                        cmd->sense,
4624                                        sizeof (struct scsi_extended_sense),
4625                                        DDI_DEV_AUTOINCR);
3424                                        acmd->cmd_scblen -
3425                                        offsetof(struct scsi_arq_status,
3426                                        sts_sensedata), DDI_DEV_AUTOINCR);
4626                            }
4627                            break;
4628                    case MFI_STAT_LD_OFFLINE:
4629                    case MFI_STAT_DEVICE_NOT_FOUND:
4630                            con_log(CL_ANN, (CE_CONT,
3431                            con_log(CL_ANN1, (CE_CONT,
4631                            "mrsas_softintr:device not found error"));
4632                            pkt->pkt_reason = CMD_DEV_GONE;
4633                            pkt->pkt_statistics  = STAT_DISCON;
4634                            break;
4635                    case MFI_STAT_LD_LBA_OUT_OF_RANGE:
4636                            pkt->pkt_state |= STATE_ARQ_DONE;
4637                            pkt->pkt_reason = CMD_CMPLT;
4638                            ((struct scsi_status *)
4639                                pkt->pkt_scbp)->sts_chk = 1;

4641                            arqstat = (void *)(pkt->pkt_scbp);
4642                            arqstat->sts_rqpkt_reason = CMD_CMPLT;
4643                            arqstat->sts_rqpkt_resid = 0;
4644                            arqstat->sts_rqpkt_state |= STATE_GOT_BUS
4645                                | STATE_GOT_TARGET | STATE_SENT_CMD
4646                                | STATE_XFERRED_DATA;
4647                            *(uint8_t *)&arqstat->sts_rqpkt_status =
4648                                STATUS_GOOD;

4650                            arqstat->sts_sensedata.es_valid = 1;
4651                            arqstat->sts_sensedata.es_key =
4652                                KEY_ILLEGAL_REQUEST;
4653                            arqstat->sts_sensedata.es_class =
4654                                CLASS_EXTENDED_SENSE;

4656                            /*
4657                             * LOGICAL BLOCK ADDRESS OUT OF RANGE:
4658                             * ASC: 0x21h; ASCQ: 0x00h;
4659                             */
4660                            arqstat->sts_sensedata.es_add_code = 0x21;
4661                            arqstat->sts_sensedata.es_qual_code = 0x00;

4663                            break;

4665                    default:
4666                            con_log(CL_ANN, (CE_CONT, "Unknown status!"));
4667                            pkt->pkt_reason = CMD_TRAN_ERR;

4669                            break;
4670                    }

4672                    atomic_add_16(&instance->fw_outstanding, (-1));
```



```
4674                    (void) mrsas_common_check(instance, cmd);

4676                    if (acmd->cmd_dmahandle) {
4677                            if (mrsas_check_dma_handle(
4678                                acmd->cmd_dmahandle) != DDI_SUCCESS) {
4679                                ddi_fm_service_impact(instance->dip,
4680                                    DDI_SERVICE_UNAFFECTED);
4681                                pkt->pkt_reason = CMD_TRAN_ERR;
4682                                pkt->pkt_statistics = 0;
4683                            }
4684                    }

4686                    /* Call the callback routine */
4687                    if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4688                        pkt->pkt_comp) {

4690                            con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_softintr: "
3491                            con_log(CL_ANN1, (CE_NOTE, "mrsas_softintr: "
4691                                "posting to scsa cmd %p index %x pkt %p "
4692                                "time %llx", (void *)cmd, cmd->index,
4693                                (void *)pkt, gethrtime()));
4694                            (*pkt->pkt_comp)(pkt);

4696                    }

4698                    return_mfi_pkt(instance, cmd);
4699                    break;

4701            case MFI_CMD_OP_SMP:
4702            case MFI_CMD_OP_STP:
4703                    complete_cmd_in_sync_mode(instance, cmd);
4704                    break;

4706            case MFI_CMD_OP_DCMD:
4707                    /* see if got an event notification */
4708                    if (ddi_get32(cmd->frame_dma_obj.acc_handle,
4709                        &cmd->frame->dcmd.opcode) ==
4710                        MR_DCMD_CTRL_EVENT_WAIT) {
4711                            if ((instance->aen_cmd == cmd) &&
4712                                (instance->aen_cmd->abort_aen)) {
4713                                con_log(CL_ANN, (CE_WARN,
4714                                    "mrsas_softintr: "
4715                                    "aborted_aen returned"));
4716                            } else {
4717                                    atomic_add_16(&instance->fw_outstanding,
4718                                        (-1));
4719                                    service_mfi_aen(instance, cmd);
4720                            }
4721                    } else {
4722                            complete_cmd_in_sync_mode(instance, cmd);
4723                    }

4725                    break;

4727            case MFI_CMD_OP_ABORT:
4728                    con_log(CL_ANN, (CE_NOTE, "MFI_CMD_OP_ABORT complete"));
3525                    con_log(CL_ANN, (CE_WARN, "MFI_CMD_OP_ABORT complete"));
4729                    /*
4730                     * MFI_CMD_OP_ABORT successfully completed
4731                     * in the synchronous mode
4732                     */
4733                    complete_cmd_in_sync_mode(instance, cmd);
4734                    break;

4736            default:
4737                    mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
```

```
4738                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

4740                        if (cmd->pkt != NULL) {
4741                                pkt = cmd->pkt;
4742                                if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
4743                                    pkt->pkt_comp) {

4745                                        con_log(CL_ANN1, (CE_CONT, "posting to "
4746                                            "scsa cmd %p index %x pkt %p"
4747                                            "time %llx, default ", (void *)cmd,
4748                                            cmd->index, (void *)pkt,
4749                                            gethrtime()));

4751                                        (*pkt->pkt_comp)(pkt);

4753                                }
4754                        }
4755                        con_log(CL_ANN, (CE_WARN, "Cmd type unknown !"));
4756                        break;
4757                }
4758        }

4760        instance->softint_running = 0;

4762        return (DDI_INTR_CLAIMED);
4763 }

4765 /*
4766  * mrsas_alloc_dma_obj
4767  *
4768  * Allocate the memory and other resources for an dma object.
4769  */
4770 int
3566 static int
4771 mrsas_alloc_dma_obj(struct mrsas_instance *instance, dma_obj_t *obj,
4772     uchar_t endian_flags)
4773 {
4774        int     i;
4775        size_t  alen = 0;
4776        uint_t  cookie_cnt;
4777        struct ddi_device_acc_attr tmp_endian_attr;

4779        tmp_endian_attr = endian_attr;
4780        tmp_endian_attr.devacc_attr_endian_flags = endian_flags;
4781        tmp_endian_attr.devacc_attr_access = DDI_DEFAULT_ACC;

4783        i = ddi_dma_alloc_handle(instance->dip, &obj->dma_attr,
4784            DDI_DMA_SLEEP, NULL, &obj->dma_handle);
4785        if (i != DDI_SUCCESS) {

4787                switch (i) {
4788                        case DDI_DMA_BADATTR :
4789                                con_log(CL_ANN, (CE_WARN,
4790                                "Failed ddi_dma_alloc_handle- Bad attribute"));
4791                                break;
4792                        case DDI_DMA_NORESOURCES :
4793                                con_log(CL_ANN, (CE_WARN,
4794                                "Failed ddi_dma_alloc_handle- No Resources"));
4795                                break;
4796                        default :
4797                                con_log(CL_ANN, (CE_WARN,
4798                                "Failed ddi_dma_alloc_handle: "
4799                                "unknown status %d", i));
4800                                break;
4801                }
```



```
4803                        return (-1);
4804                }

4806                if ((ddi_dma_mem_alloc(obj->dma_handle, obj->size, &tmp_endian_attr,
4807                    DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP, NULL,
4808                    &obj->buffer, &alen, &obj->acc_handle) != DDI_SUCCESS) ||
4809                    alen < obj->size) {

4811                        ddi_dma_free_handle(&obj->dma_handle);

4813                        con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_mem_alloc"));

4815                        return (-1);
4816                }

4818                if (ddi_dma_addr_bind_handle(obj->dma_handle, NULL, obj->buffer,
4819                    obj->size, DDI_DMA_RDWR | DDI_DMA_STREAMING, DDI_DMA_SLEEP,
4820                    NULL, &obj->dma_cookie[0], &cookie_cnt) != DDI_SUCCESS) {

4822                        ddi_dma_mem_free(&obj->acc_handle);
4823                        ddi_dma_free_handle(&obj->dma_handle);

4825                        con_log(CL_ANN, (CE_WARN, "Failed : ddi_dma_addr_bind_handle"));

4827                        return (-1);
4828                }

4830                if (mrsas_check_dma_handle(obj->dma_handle) != DDI_SUCCESS) {
4831                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4832                        return (-1);
4833                }

4835                if (mrsas_check_acc_handle(obj->acc_handle) != DDI_SUCCESS) {
4836                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
4837                        return (-1);
4838                }

4840                return (cookie_cnt);
4841 }

4843 /*
4844  * mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t)
4845  *
4846  * De-allocate the memory and other resources for an dma object, which must
4847  * have been alloated by a previous call to mrsas_alloc_dma_obj()
4848  */
4849 int
3645 static int
4850 mrsas_free_dma_obj(struct mrsas_instance *instance, dma_obj_t obj)
4851 {

4853                if ((obj.dma_handle == NULL) || (obj.acc_handle == NULL)) {
4854                        return (DDI_SUCCESS);
4855                }

4857                /*
4858                 * NOTE: These check-handle functions fail if *_handle == NULL, but
4859                 * this function succeeds because of the previous check.
4860                 */
4861                if (mrsas_check_dma_handle(obj.dma_handle) != DDI_SUCCESS) {
4862                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
4863                        return (DDI_FAILURE);
4864                }

4866                if (mrsas_check_acc_handle(obj.acc_handle) != DDI_SUCCESS) {
4867                        ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
```

```
4868                    return (DDI_FAILURE);
4869            }

4871            (void) ddi_dma_unbind_handle(obj.dma_handle);
4872            ddi_dma_mem_free(&obj.acc_handle);
4873            ddi_dma_free_handle(&obj.dma_handle);
4874            obj.acc_handle = NULL;

4875            return (DDI_SUCCESS);
4876    }

4878    /*
4879     * mrsas_dma_alloc(instance_t *, struct scsi_pkt *, struct buf *,
4880     * int, int (*)())
4881     *
4882     * Allocate dma resources for a new scsi command
4883     */
4884    int
3672    static int
4885    mrsas_dma_alloc(struct mrsas_instance *instance, struct scsi_pkt *pkt,
4886        struct buf *bp, int flags, int (*callback)())
4887    {
4888            int     dma_flags;
4889            int     (*cb)(caddr_t);
4890            int     i;

4892            ddi_dma_attr_t  tmp_dma_attr = mrsas_generic_dma_attr;
4893            struct scsa_cmd *acmd = PKT2CMD(pkt);

4895            acmd->cmd_buf = bp;

4897            if (bp->b_flags & B_READ) {
4898                    acmd->cmd_flags &= ~CFLAG_DMASEND;
4899                    dma_flags = DDI_DMA_READ;
4900            } else {
4901                    acmd->cmd_flags |= CFLAG_DMASEND;
4902                    dma_flags = DDI_DMA_WRITE;
4903            }

4905            if (flags & PKT_CONSISTENT) {
4906                    acmd->cmd_flags |= CFLAG_CONSISTENT;
4907                    dma_flags |= DDI_DMA_CONSISTENT;
4908            }

4910            if (flags & PKT_DMA_PARTIAL) {
4911                    dma_flags |= DDI_DMA_PARTIAL;
4912            }

4914            dma_flags |= DDI_DMA_REDZONE;

4916            cb = (callback == NULL_FUNC) ? DDI_DMA_DONTWAIT : DDI_DMA_SLEEP;

4918            tmp_dma_attr.dma_attr_sgllen = instance->max_num_sge;
4919            tmp_dma_attr.dma_attr_addr_hi = 0xffffffffffffffffull;
4920            if (instance->tbolt) {
4921                    /* OCR-RESET FIX */
4922                    tmp_dma_attr.dma_attr_count_max =
4923                        (U64)mrsas_tbolt_max_cap_maxxfer;  /* limit to 256K */
4924                    tmp_dma_attr.dma_attr_maxxfer =
4925                        (U64)mrsas_tbolt_max_cap_maxxfer;  /* limit to 256K */
4926            }

4928            if ((i = ddi_dma_alloc_handle(instance->dip, &tmp_dma_attr,
4929                cb, 0, &acmd->cmd_dmahandle)) != DDI_SUCCESS) {
4930                    switch (i) {
4931                    case DDI_DMA_BADATTR:
```



```
4932                            bioerror(bp, EFAULT);
4933                            return (DDI_FAILURE);

4935                    case DDI_DMA_NORESOURCES:
4936                            bioerror(bp, 0);
4937                            return (DDI_FAILURE);

4939                    default:
4940                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_alloc_handle: "
4941                                "impossible result (0x%x)", i));
4942                            bioerror(bp, EFAULT);
4943                            return (DDI_FAILURE);
4944                    }
4945            }

4947            i = ddi_dma_buf_bind_handle(acmd->cmd_dmahandle, bp, dma_flags,
4948                cb, 0, &acmd->cmd_dmacookies[0], &acmd->cmd_ncookies);

4950            switch (i) {
4951            case DDI_DMA_PARTIAL_MAP:
4952                    if ((dma_flags & DDI_DMA_PARTIAL) == 0) {
4953                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle: "
4954                                "DDI_DMA_PARTIAL_MAP impossible"));
4955                            goto no_dma_cookies;
4956                    }

4958                    if (ddi_dma_numwin(acmd->cmd_dmahandle, &acmd->cmd_nwin) ==
4959                        DDI_FAILURE) {
4960                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_numwin failed"));
4961                            goto no_dma_cookies;
4962                    }

4964                    if (ddi_dma_getwin(acmd->cmd_dmahandle, acmd->cmd_curwin,
4965                        &acmd->cmd_dma_offset, &acmd->cmd_dma_len,
4966                        &acmd->cmd_dmacookies[0], &acmd->cmd_ncookies) ==
4967                        DDI_FAILURE) {

4969                            con_log(CL_ANN, (CE_PANIC, "ddi_dma_getwin failed"));
4970                            goto no_dma_cookies;
4971                    }

4973                    goto get_dma_cookies;
4974            case DDI_DMA_MAPPED:
4975                    acmd->cmd_nwin = 1;
4976                    acmd->cmd_dma_len = 0;
4977                    acmd->cmd_dma_offset = 0;

4979    get_dma_cookies:
4980                    i = 0;
4981                    acmd->cmd_dmacount = 0;
4982                    for (;;) {
4983                            acmd->cmd_dmacount +=
4984                                acmd->cmd_dmacookies[i++].dmac_size;

4986                            if (i == instance->max_num_sge ||
4987                                i == acmd->cmd_ncookies)
4988                                    break;

4990                            ddi_dma_nextcookie(acmd->cmd_dmahandle,
4991                                &acmd->cmd_dmacookies[i]);
4992                    }

4994                    acmd->cmd_cookie = i;
4995                    acmd->cmd_cookiecnt = i;

4997                    acmd->cmd_flags |= CFLAG_DMAVALID;
```

```
4999                 if (bp->b_bcount >= acmd->cmd_dmacount) {
5000                         pkt->pkt_resid = bp->b_bcount - acmd->cmd_dmacount;
5001                 } else {
5002                         pkt->pkt_resid = 0;
5003                 }

5005                 return (DDI_SUCCESS);
5006         case DDI_DMA_NORESOURCES:
5007                 bioerror(bp, 0);
5008                 break;
5009         case DDI_DMA_NOMAPPING:
5010                 bioerror(bp, EFAULT);
5011                 break;
5012         case DDI_DMA_TOOBIG:
5013                 bioerror(bp, EINVAL);
5014                 break;
5015         case DDI_DMA_INUSE:
5016                 con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle:"
5017                     " DDI_DMA_INUSE impossible"));
5018                 break;
5019         default:
5020                 con_log(CL_ANN, (CE_PANIC, "ddi_dma_buf_bind_handle: "
5021                     "impossible result (0x%x)", i));
5022                 break;
5023         }

5025 no_dma_cookies:
5026         ddi_dma_free_handle(&acmd->cmd_dmahandle);
5027         acmd->cmd_dmahandle = NULL;
5028         acmd->cmd_flags &= ~CFLAG_DMAVALID;
5029         return (DDI_FAILURE);
5030 }

5032 /*
5033  * mrsas_dma_move(struct mrsas_instance *, struct scsi_pkt *, struct buf *)
5034  *
5035  * move dma resources to next dma window
5036  *
5037  */
5038 int
3819 static int
5039 mrsas_dma_move(struct mrsas_instance *instance, struct scsi_pkt *pkt,
5040     struct buf *bp)
5041 {
5042         int     i = 0;

5044         struct scsa_cmd *acmd = PKT2CMD(pkt);

5046         /*
5047          * If there are no more cookies remaining in this window,
5048          * must move to the next window first.
5049          */
5050         if (acmd->cmd_cookie == acmd->cmd_ncookies) {
5051                 if (acmd->cmd_curwin == acmd->cmd_nwin && acmd->cmd_nwin == 1) {
5052                         return (DDI_SUCCESS);
5053                 }

5055                 /* at last window, cannot move */
5056                 if (++acmd->cmd_curwin >= acmd->cmd_nwin) {
5057                         return (DDI_FAILURE);
5058                 }

5060                 if (ddi_dma_getwin(acmd->cmd_dmahandle, acmd->cmd_curwin,
5061                     &acmd->cmd_dma_offset, &acmd->cmd_dma_len,
5062                     &acmd->cmd_dmacookies[0], &acmd->cmd_ncookies) ==
```



```
5063                     DDI_FAILURE) {
5064                         return (DDI_FAILURE);
5065                 }

5067                 acmd->cmd_cookie = 0;
5068         } else {
5069                 /* still more cookies in this window - get the next one */
5070                 ddi_dma_nextcookie(acmd->cmd_dmahandle,
5071                     &acmd->cmd_dmacookies[0]);
5072         }

5074         /* get remaining cookies in this window, up to our maximum */
5075         for (;;) {
5076                 acmd->cmd_dmacount += acmd->cmd_dmacookies[i++].dmac_size;
5077                 acmd->cmd_cookie++;

5079                 if (i == instance->max_num_sge ||
5080                     acmd->cmd_cookie == acmd->cmd_ncookies) {
5081                         break;
5082                 }

5084                 ddi_dma_nextcookie(acmd->cmd_dmahandle,
5085                     &acmd->cmd_dmacookies[i]);
5086         }

5088         acmd->cmd_cookiecnt = i;

5090         if (bp->b_bcount >= acmd->cmd_dmacount) {
5091                 pkt->pkt_resid = bp->b_bcount - acmd->cmd_dmacount;
5092         } else {
5093                 pkt->pkt_resid = 0;
5094         }

5096         return (DDI_SUCCESS);
5097 }

5099 /*
5100  * build_cmd
5101  */
5102 static struct mrsas_cmd *
5103 build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
5104     struct scsi_pkt *pkt, uchar_t *cmd_done)
5105 {
5106         uint16_t        flags = 0;
5107         uint32_t        i;
5108         uint32_t        context;
5109         uint32_t        sge_bytes;
5110         uint32_t        tmp_data_xfer_len;
5111         ddi_acc_handle_t acc_handle;
5112         struct mrsas_cmd                *cmd;
5113         struct mrsas_sge64              *mfi_sgl;
5114         struct mrsas_sge_ieee           *mfi_sgl_ieee;
5115         struct scsa_cmd                 *acmd = PKT2CMD(pkt);
5116         struct mrsas_pthru_frame        *pthru;
5117         struct mrsas_io_frame           *ldio;

5119         /* find out if this is logical or physical drive command.  */
5120         acmd->islogical = MRDRV_IS_LOGICAL(ap);
5121         acmd->device_id = MAP_DEVICE_ID(instance, ap);
5122         *cmd_done = 0;

5124         /* get the command packet */
5125         if (!(cmd = get_mfi_pkt(instance))) {
5126                 DTRACE_PROBE2(build_cmd_mfi_err, uint16_t,
5127                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
5128                 return (NULL);
```

```
5129         }

3911         cmd->retry_count_for_ocr = 0;

5131         acc_handle = cmd->frame_dma_obj.acc_handle;

5133         /* Clear the frame buffer and assign back the context id */
5134         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
5135         ddi_put32(acc_handle, &cmd->frame->hdr.context, cmd->index);

5137         cmd->pkt = pkt;
5138         cmd->cmd = acmd;
5139         DTRACE_PROBE3(build_cmds, uint8_t, pkt->pkt_cdbp[0],
5140             ulong_t, acmd->cmd_dmacount, ulong_t, acmd->cmd_dma_len);

5142         /* lets get the command directions */
5143         if (acmd->cmd_flags & CFLAG_DMASEND) {
5144                 flags = MFI_FRAME_DIR_WRITE;

5146                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5147                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5148                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5149                             DDI_DMA_SYNC_FORDEV);
5150                 }
5151         } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
5152                 flags = MFI_FRAME_DIR_READ;

5154                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
5155                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
5156                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
5157                             DDI_DMA_SYNC_FORCPU);
5158                 }
5159         } else {
5160                 flags = MFI_FRAME_DIR_NONE;
5161         }

5163         if (instance->flag_ieee) {
5164                 flags |= MFI_FRAME_IEEE;
5165         }
5166         flags |= MFI_FRAME_SGL64;

5168         switch (pkt->pkt_cdbp[0]) {

5170         /*
5171          * case SCMD_SYNCHRONIZE_CACHE:
5172          *      flush_cache(instance);
5173          *      return_mfi_pkt(instance, cmd);
5174          *      *cmd_done = 1;
5175          *
5176          *      return (NULL);
5177          */

5179         case SCMD_READ:
5180         case SCMD_WRITE:
5181         case SCMD_READ_G1:
5182         case SCMD_WRITE_G1:
5183         case SCMD_READ_G4:
5184         case SCMD_WRITE_G4:
5185         case SCMD_READ_G5:
5186         case SCMD_WRITE_G5:
5187                 if (acmd->islogical) {
5188                         ldio = (struct mrsas_io_frame *)cmd->frame;

5190                         /*
5191                          * preare the Logical IO frame:
5192                          * 2nd bit is zero for all read cmds
```



```
5193                          */
5194                         ddi_put8(acc_handle, &ldio->cmd,
5195                             (pkt->pkt_cdbp[0] & 0x02) ? MFI_CMD_OP_LD_WRITE
5196                             : MFI_CMD_OP_LD_READ);
5197                         ddi_put8(acc_handle, &ldio->cmd_status, 0x0);
5198                         ddi_put8(acc_handle, &ldio->scsi_status, 0x0);
5199                         ddi_put8(acc_handle, &ldio->target_id, acmd->device_id);
5200                         ddi_put16(acc_handle, &ldio->timeout, 0);
5201                         ddi_put8(acc_handle, &ldio->reserved_0, 0);
5202                         ddi_put16(acc_handle, &ldio->pad_0, 0);
5203                         ddi_put16(acc_handle, &ldio->flags, flags);

5205                         /* Initialize sense Information */
5206                         bzero(cmd->sense, SENSE_LENGTH);
5207                         ddi_put8(acc_handle, &ldio->sense_len, SENSE_LENGTH);
5208                         ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_hi, 0);
5209                         ddi_put32(acc_handle, &ldio->sense_buf_phys_addr_lo,
5210                             cmd->sense_phys_addr);
5211                         ddi_put32(acc_handle, &ldio->start_lba_hi, 0);
5212                         ddi_put8(acc_handle, &ldio->access_byte,
5213                             (acmd->cmd_cdblen != 6) ? pkt->pkt_cdbp[1] : 0);
5214                         ddi_put8(acc_handle, &ldio->sge_count,
5215                             acmd->cmd_cookiecnt);
5216                         if (instance->flag_ieee) {
5217                                 mfi_sgl_ieee =
5218                                     (struct mrsas_sge_ieee *)&ldio->sgl;
5219                         } else {
5220                                 mfi_sgl = (struct mrsas_sge64   *)&ldio->sgl;
5221                         }

5223                         context = ddi_get32(acc_handle, &ldio->context);

5225                         if (acmd->cmd_cdblen == CDB_GROUP0) {
5226                                 /* 6-byte cdb */
5227                                 ddi_put32(acc_handle, &ldio->lba_count, (
5228                                     (uint16_t)(pkt->pkt_cdbp[4])));

5230                                 ddi_put32(acc_handle, &ldio->start_lba_lo, (
5231                                     ((uint32_t)(pkt->pkt_cdbp[3])) |
5232                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
5233                                     ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
5234                                     << 16)));
5235                         } else if (acmd->cmd_cdblen == CDB_GROUP1) {
5236                                 /* 10-byte cdb */
5237                                 ddi_put32(acc_handle, &ldio->lba_count, (
5238                                     ((uint16_t)(pkt->pkt_cdbp[8])) |
5239                                     ((uint16_t)(pkt->pkt_cdbp[7]) << 8)));

5241                                 ddi_put32(acc_handle, &ldio->start_lba_lo, (
5242                                     ((uint32_t)(pkt->pkt_cdbp[5])) |
5243                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5244                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5245                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5246                         } else if (acmd->cmd_cdblen == CDB_GROUP5) {
5247                                 /* 12-byte cdb */
4022                         } else if (acmd->cmd_cdblen == CDB_GROUP2) {
5248                                 ddi_put32(acc_handle, &ldio->lba_count, (
5249                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
5250                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5251                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5252                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));
4024                                     ((uint16_t)(pkt->pkt_cdbp[9])) |
4025                                     ((uint16_t)(pkt->pkt_cdbp[8]) << 8) |
4026                                     ((uint16_t)(pkt->pkt_cdbp[7]) << 16) |
4027                                     ((uint16_t)(pkt->pkt_cdbp[6]) << 24)));
```

```
5254                                ddi_put32(acc_handle, &ldio->start_lba_lo, (
5255                                    ((uint32_t)(pkt->pkt_cdbp[5])) |
5256                                    ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5257                                    ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5258                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5259                        } else if (acmd->cmd_cdblen == CDB_GROUP4) {
5260                                /* 16-byte cdb */
4034                        } else if (acmd->cmd_cdblen == CDB_GROUP3) {
5261                                ddi_put32(acc_handle, &ldio->lba_count, (
5262                                    ((uint32_t)(pkt->pkt_cdbp[13])) |
5263                                    ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
5264                                    ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
5265                                    ((uint32_t)(pkt->pkt_cdbp[10]) << 24)));
4036                                ((uint16_t)(pkt->pkt_cdbp[13])) |
4037                                ((uint16_t)(pkt->pkt_cdbp[12]) << 8) |
4038                                ((uint16_t)(pkt->pkt_cdbp[11]) << 16) |
4039                                ((uint16_t)(pkt->pkt_cdbp[10]) << 24)));

5267                                ddi_put32(acc_handle, &ldio->start_lba_lo, (
5268                                    ((uint32_t)(pkt->pkt_cdbp[9])) |
5269                                    ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
5270                                    ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
5271                                    ((uint32_t)(pkt->pkt_cdbp[6]) << 24)));

5273                                ddi_put32(acc_handle, &ldio->start_lba_hi, (
4047                                ddi_put32(acc_handle, &ldio->start_lba_lo, (
5274                                    ((uint32_t)(pkt->pkt_cdbp[5])) |
5275                                    ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
5276                                    ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
5277                                    ((uint32_t)(pkt->pkt_cdbp[2]) << 24)));
5278                        }

5280                        break;
5281                }
5282                /* fall through For all non-rd/wr cmds */
5283        default:

5285                switch (pkt->pkt_cdbp[0]) {
5286                case SCMD_MODE_SENSE:
5287                case SCMD_MODE_SENSE_G1: {
5288                        union scsi_cdb  *cdbp;
5289                        uint16_t        page_code;

5291                        cdbp = (void *)pkt->pkt_cdbp;
5292                        page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
5293                        switch (page_code) {
5294                        case 0x3:
5295                        case 0x4:
5296                                (void) mrsas_mode_sense_build(pkt);
5297                                return_mfi_pkt(instance, cmd);
5298                                *cmd_done = 1;
5299                                return (NULL);
5300                        }
5301                        break;
5302                }
5303                default:
5304                        break;
5305                }

5307                pthru   = (struct mrsas_pthru_frame *)cmd->frame;

5309                /* prepare the DCDB frame */
5310                ddi_put8(acc_handle, &pthru->cmd, (acmd->islogical) ?
5311                    MFI_CMD_OP_LD_SCSI : MFI_CMD_OP_PD_SCSI);
5312                ddi_put8(acc_handle, &pthru->cmd_status, 0x0);
5313                ddi_put8(acc_handle, &pthru->scsi_status, 0x0);
```



```
5314                ddi_put8(acc_handle, &pthru->target_id, acmd->device_id);
5315                ddi_put8(acc_handle, &pthru->lun, 0);
5316                ddi_put8(acc_handle, &pthru->cdb_len, acmd->cmd_cdblen);
5317                ddi_put16(acc_handle, &pthru->timeout, 0);
5318                ddi_put16(acc_handle, &pthru->flags, flags);
5319                tmp_data_xfer_len = 0;
5320                for (i = 0; i < acmd->cmd_cookiecnt; i++) {
5321                        tmp_data_xfer_len += acmd->cmd_dmacookies[i].dmac_size;
5322                }
5323                ddi_put32(acc_handle, &pthru->data_xfer_len,
5324                    tmp_data_xfer_len);
4094                    acmd->cmd_dmacount);
5325                ddi_put8(acc_handle, &pthru->sge_count, acmd->cmd_cookiecnt);
5326                if (instance->flag_ieee) {
5327                        mfi_sgl_ieee = (struct mrsas_sge_ieee *)&pthru->sgl;
5328                } else {
5329                        mfi_sgl = (struct mrsas_sge64 *)&pthru->sgl;
5330                }

5332                bzero(cmd->sense, SENSE_LENGTH);
5333                ddi_put8(acc_handle, &pthru->sense_len, SENSE_LENGTH);
5334                ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
5335                ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo,
5336                    cmd->sense_phys_addr);

5338                context = ddi_get32(acc_handle, &pthru->context);
5339                ddi_rep_put8(acc_handle, (uint8_t *)pkt->pkt_cdbp,
5340                    (uint8_t *)pthru->cdb, acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

5342                break;
5343        }
5344 #ifdef lint
5345        context = context;
5346 #endif
5347        /* prepare the scatter-gather list for the firmware */
5348        if (instance->flag_ieee) {
5349                for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl_ieee++) {
5350                        ddi_put64(acc_handle, &mfi_sgl_ieee->phys_addr,
5351                            acmd->cmd_dmacookies[i].dmac_laddress);
5352                        ddi_put32(acc_handle, &mfi_sgl_ieee->length,
5353                            acmd->cmd_dmacookies[i].dmac_size);
5354                }
5355                sge_bytes = sizeof (struct mrsas_sge_ieee)*acmd->cmd_cookiecnt;
5356        } else {
5357                for (i = 0; i < acmd->cmd_cookiecnt; i++, mfi_sgl++) {
5358                        ddi_put64(acc_handle, &mfi_sgl->phys_addr,
5359                            acmd->cmd_dmacookies[i].dmac_laddress);
5360                        ddi_put32(acc_handle, &mfi_sgl->length,
5361                            acmd->cmd_dmacookies[i].dmac_size);
5362                }
5363                sge_bytes = sizeof (struct mrsas_sge64)*acmd->cmd_cookiecnt;
5364        }

5366        cmd->frame_count = (sge_bytes / MRMFI_FRAME_SIZE) +
5367            ((sge_bytes % MRMFI_FRAME_SIZE) ? 1 : 0) + 1;

5369        if (cmd->frame_count >= 8) {
5370                cmd->frame_count = 8;
5371        }

5373        return (cmd);
5374 }

5376 #ifndef __sparc
5377 /*
5378  * wait_for_outstanding -       Wait for all outstanding cmds
```

```
 * @instance:                            Adapter soft state
 *
 * This function waits for upto MRDRV_RESET_WAIT_TIME seconds for FW to
 * complete all its outstanding commands. Returns error if one or more IOs
 * are pending after this time period.
 */
static int
wait_for_outstanding(struct mrsas_instance *instance)
{
        int             i;
        uint32_t        wait_time = 90;

        for (i = 0; i < wait_time; i++) {
                if (!instance->fw_outstanding) {
                        break;
                }

                drv_usecwait(MILLISEC); /* wait for 1000 usecs */
        }

        if (instance->fw_outstanding) {
                return (1);
        }

        return (0);
}
#endif  /* __sparc */

/*
 * issue_mfi_pthru
 */
static int
issue_mfi_pthru(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
    struct mrsas_cmd *cmd, int mode)
{
        void            *ubuf;
        uint32_t        kphys_addr = 0;
        uint32_t        xferlen = 0;
        uint32_t        new_xfer_length = 0;
        uint_t          model;
        ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
        dma_obj_t                       pthru_dma_obj;
        struct mrsas_pthru_frame        *kpthru;
        struct mrsas_pthru_frame        *pthru;
        int i;
        pthru = &cmd->frame->pthru;
        kpthru = (struct mrsas_pthru_frame *)&ioctl->frame[0];

        if (instance->adapterresetinprogress) {
                con_log(CL_ANN1, (CE_WARN, "issue_mfi_pthru: Reset flag set, "
                con_log(CL_ANN1, (CE_NOTE, "issue_mfi_pthru: Reset flag set, "
                "returning mfi_pkt and setting TRAN_BUSY\n"));
                return (DDI_FAILURE);
        }
        model = ddi_model_convert_from(mode & FMODELS);
        if (model == DDI_MODEL_ILP32) {
                con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP32"));
                con_log(CL_ANN1, (CE_NOTE, "issue_mfi_pthru: DDI_MODEL_LP32"));

                xferlen = kpthru->sgl.sge32[0].length;

                ubuf    = (void *)(ulong_t)kpthru->sgl.sge32[0].phys_addr;
        } else {
#ifdef _ILP32
                con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP32"));
                con_log(CL_ANN1, (CE_NOTE, "issue_mfi_pthru: DDI_MODEL_LP32"));
```



```
                xferlen = kpthru->sgl.sge32[0].length;
                ubuf    = (void *)(ulong_t)kpthru->sgl.sge32[0].phys_addr;
#else
                con_log(CL_ANN1, (CE_CONT, "issue_mfi_pthru: DDI_MODEL_LP64"));
                con_log(CL_ANN1, (CE_NOTE, "issue_mfi_pthru: DDI_MODEL_LP64"));
                xferlen = kpthru->sgl.sge64[0].length;
                ubuf    = (void *)(ulong_t)kpthru->sgl.sge64[0].phys_addr;
#endif
        }

        if (xferlen) {
                /* means IOCTL requires DMA */
                /* allocate the data transfer buffer */
                /* pthru_dma_obj.size = xferlen; */
                MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
                    PAGESIZE);
                pthru_dma_obj.size = new_xfer_length;
                pthru_dma_obj.size = xferlen;
                pthru_dma_obj.dma_attr = mrsas_generic_dma_attr;
                pthru_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
                pthru_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
                pthru_dma_obj.dma_attr.dma_attr_sgllen = 1;
                pthru_dma_obj.dma_attr.dma_attr_align = 1;

                /* allocate kernel buffer for DMA */
                if (mrsas_alloc_dma_obj(instance, &pthru_dma_obj,
                    (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
                        con_log(CL_ANN, (CE_WARN, "issue_mfi_pthru: "
                            "could not allocate data transfer buffer."));
                        return (DDI_FAILURE);
                }
                (void) memset(pthru_dma_obj.buffer, 0, xferlen);

                /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
                if (kpthru->flags & MFI_FRAME_DIR_WRITE) {
                        for (i = 0; i < xferlen; i++) {
                                if (ddi_copyin((uint8_t *)ubuf+i,
                                    (uint8_t *)pthru_dma_obj.buffer+i,
                                    1, mode)) {
                                        con_log(CL_ANN, (CE_WARN,
                                            "issue_mfi_pthru : "
                                            "copy from user space failed"));
                                        return (DDI_FAILURE);
                                }
                        }
                }

                kphys_addr = pthru_dma_obj.dma_cookie[0].dmac_address;
        }

        ddi_put8(acc_handle, &pthru->cmd, kpthru->cmd);
        ddi_put8(acc_handle, &pthru->sense_len, SENSE_LENGTH);
        ddi_put8(acc_handle, &pthru->sense_len, 0);
        ddi_put8(acc_handle, &pthru->cmd_status, 0);
        ddi_put8(acc_handle, &pthru->scsi_status, 0);
        ddi_put8(acc_handle, &pthru->target_id, kpthru->target_id);
        ddi_put8(acc_handle, &pthru->lun, kpthru->lun);
        ddi_put8(acc_handle, &pthru->cdb_len, kpthru->cdb_len);
        ddi_put8(acc_handle, &pthru->sge_count, kpthru->sge_count);
        ddi_put16(acc_handle, &pthru->timeout, kpthru->timeout);
        ddi_put32(acc_handle, &pthru->data_xfer_len, kpthru->data_xfer_len);

        ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_hi, 0);
        pthru->sense_buf_phys_addr_lo = cmd->sense_phys_addr;
        /* ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo, 0); */
        /* pthru->sense_buf_phys_addr_lo = cmd->sense_phys_addr; */
```

```
4258          ddi_put32(acc_handle, &pthru->sense_buf_phys_addr_lo, 0);

5505          ddi_rep_put8(acc_handle, (uint8_t *)kpthru->cdb, (uint8_t *)pthru->cdb,
5506              pthru->cdb_len, DDI_DEV_AUTOINCR);

5508          ddi_put16(acc_handle, &pthru->flags, kpthru->flags & ~MFI_FRAME_SGL64);
5509          ddi_put32(acc_handle, &pthru->sgl.sge32[0].length, xferlen);
5510          ddi_put32(acc_handle, &pthru->sgl.sge32[0].phys_addr, kphys_addr);

5512          cmd->sync_cmd = MRSAS_TRUE;
5513          cmd->frame_count = 1;

5515          if (instance->tbolt) {
5516                  mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5517          }

5519          if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5520                  con_log(CL_ANN, (CE_WARN,
5521                      "issue_mfi_pthru: fw_ioctl failed"));
5522          } else {
5523                  if (xferlen && kpthru->flags & MFI_FRAME_DIR_READ) {
5524                          for (i = 0; i < xferlen; i++) {
5525                                  if (ddi_copyout(
5526                                      (uint8_t *)pthru_dma_obj.buffer+i,
5527                                      (uint8_t *)ubuf+i, 1, mode)) {
5528                                          con_log(CL_ANN, (CE_WARN,
5529                                              "issue_mfi_pthru : "
5530                                              "copy to user space failed"));
5531                                          return (DDI_FAILURE);
5532                                  }
5533                          }
5534                  }
5535          }

5537          kpthru->cmd_status = ddi_get8(acc_handle, &pthru->cmd_status);
5538          kpthru->scsi_status = ddi_get8(acc_handle, &pthru->scsi_status);

5540          con_log(CL_ANN, (CE_CONT, "issue_mfi_pthru: cmd_status %x, "
4291          con_log(CL_ANN, (CE_NOTE, "issue_mfi_pthru: cmd_status %x, "
5541              "scsi_status %x", kpthru->cmd_status, kpthru->scsi_status));
5542          DTRACE_PROBE3(issue_pthru, uint8_t, kpthru->cmd, uint8_t,
5543              kpthru->cmd_status, uint8_t, kpthru->scsi_status);

5545          if (kpthru->sense_len) {
5546                  uint_t sense_len = SENSE_LENGTH;
5547                  void *sense_ubuf =
5548                      (void *)(ulong_t)kpthru->sense_buf_phys_addr_lo;
5549                  if (kpthru->sense_len <= SENSE_LENGTH) {
5550                          sense_len = kpthru->sense_len;
5551                  }

5553                  for (i = 0; i < sense_len; i++) {
5554                          if (ddi_copyout(
5555                              (uint8_t *)cmd->sense+i,
5556                              (uint8_t *)sense_ubuf+i, 1, mode)) {
5557                                  con_log(CL_ANN, (CE_WARN,
5558                                      "issue_mfi_pthru : "
5559                                      "copy to user space failed"));
5560                          }
5561                          con_log(CL_DLEVEL1, (CE_WARN,
5562                              "Copying Sense info sense_buff[%d] = 0x%X",
5563                              i, *((uint8_t *)cmd->sense + i)));
5564                  }
5565          }
5566          (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
5567              DDI_DMA_SYNC_FORDEV);
```



```
5569          if (xferlen) {
5570                  /* free kernel buffer */
5571                  if (mrsas_free_dma_obj(instance, pthru_dma_obj) != DDI_SUCCESS)
5572                          return (DDI_FAILURE);
5573          }

5575          return (DDI_SUCCESS);
5576  }

5578  /*
5579   * issue_mfi_dcmd
5580   */
5581  static int
5582  issue_mfi_dcmd(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5583      struct mrsas_cmd *cmd, int mode)
5584  {
5585          void            *ubuf;
5586          uint32_t        kphys_addr = 0;
5587          uint32_t        xferlen = 0;
5588          uint32_t        new_xfer_length = 0;
5589          uint32_t        model;
5590          dma_obj_t       dcmd_dma_obj;
5591          struct mrsas_dcmd_frame *kdcmd;
5592          struct mrsas_dcmd_frame *dcmd;
5593          ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5594          int i;
5595          dcmd = &cmd->frame->dcmd;
5596          kdcmd = (struct mrsas_dcmd_frame *)&ioctl->frame[0];

5598          if (instance->adapterresetinprogress) {
5599                  con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
5600                  "returning mfi_pkt and setting TRAN_BUSY"));
4325                  "returning mfi_pkt and setting TRAN_BUSY\n"));
5601                  return (DDI_FAILURE);
5602          }
5603          model = ddi_model_convert_from(mode & FMODELS);
5604          if (model == DDI_MODEL_ILP32) {
5605                  con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));
4330                  con_log(CL_ANN1, (CE_NOTE, "issue_mfi_dcmd: DDI_MODEL_ILP32"));

5607                  xferlen = kdcmd->sgl.sge32[0].length;

5609                  ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
5610          } else {
5611  #ifdef _ILP32
5612                  con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_ILP32"));
4337                  con_log(CL_ANN1, (CE_NOTE, "issue_mfi_dcmd: DDI_MODEL_ILP32"));
5613                  xferlen = kdcmd->sgl.sge32[0].length;
5614                  ubuf    = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
5615  #else
5616                  con_log(CL_ANN1, (CE_CONT, "issue_mfi_dcmd: DDI_MODEL_LP64"));
4341                  con_log(CL_ANN1, (CE_NOTE, "issue_mfi_dcmd: DDI_MODEL_LP64"));
5617                  xferlen = kdcmd->sgl.sge64[0].length;
5618                  ubuf    = (void *)(ulong_t)kdcmd->sgl.sge64[0].phys_addr;
5619  #endif
5620          }
5621          if (xferlen) {
5622                  /* means IOCTL requires DMA */
5623                  /* allocate the data transfer buffer */
5624                  /* dcmd_dma_obj.size = xferlen; */
5625                  MRSAS_GET_BOUNDARY_ALIGNED_LEN(xferlen, new_xfer_length,
5626                      PAGESIZE);
5627                  dcmd_dma_obj.size = new_xfer_length;
4349                  dcmd_dma_obj.size = xferlen;
5628                  dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
```

```
5629                dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5630                dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5631                dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
5632                dcmd_dma_obj.dma_attr.dma_attr_align = 1;

5634                /* allocate kernel buffer for DMA */
5635                        if (mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
5636                            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5637                                con_log(CL_ANN,
5638                                    (CE_WARN, "issue_mfi_dcmd: could not "
5639                                    "allocate data transfer buffer."));
4359                        con_log(CL_ANN, (CE_WARN, "issue_mfi_dcmd: "
4360                            "could not allocate data transfer buffer."));
5640                                return (DDI_FAILURE);
5641                        }
5642                (void) memset(dcmd_dma_obj.buffer, 0, xferlen);

5644                /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5645                if (kdcmd->flags & MFI_FRAME_DIR_WRITE) {
5646                        for (i = 0; i < xferlen; i++) {
5647                                if (ddi_copyin((uint8_t *)ubuf + i,
5648                                    (uint8_t *)dcmd_dma_obj.buffer + i,
5649                                    1, mode)) {
5650                                        con_log(CL_ANN, (CE_WARN,
5651                                            "issue_mfi_dcmd : "
5652                                            "copy from user space failed"));
5653                                        return (DDI_FAILURE);
5654                                }
5655                        }
5656                }

5658                kphys_addr = dcmd_dma_obj.dma_cookie[0].dmac_address;
5659        }

5661        ddi_put8(acc_handle, &dcmd->cmd, kdcmd->cmd);
5662        ddi_put8(acc_handle, &dcmd->cmd_status, 0);
5663        ddi_put8(acc_handle, &dcmd->sge_count, kdcmd->sge_count);
5664        ddi_put16(acc_handle, &dcmd->timeout, kdcmd->timeout);
5665        ddi_put32(acc_handle, &dcmd->data_xfer_len, kdcmd->data_xfer_len);
5666        ddi_put32(acc_handle, &dcmd->opcode, kdcmd->opcode);

5668        ddi_rep_put8(acc_handle, (uint8_t *)kdcmd->mbox.b,
5669            (uint8_t *)dcmd->mbox.b, DCMD_MBOX_SZ, DDI_DEV_AUTOINCR);

5671        ddi_put16(acc_handle, &dcmd->flags, kdcmd->flags & ~MFI_FRAME_SGL64);
5672        ddi_put32(acc_handle, &dcmd->sgl.sge32[0].length, xferlen);
5673        ddi_put32(acc_handle, &dcmd->sgl.sge32[0].phys_addr, kphys_addr);

5675        cmd->sync_cmd = MRSAS_TRUE;
5676        cmd->frame_count = 1;

5678        if (instance->tbolt) {
5679                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5680        }

5682        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5683                con_log(CL_ANN, (CE_WARN, "issue_mfi_dcmd: fw_ioctl failed"));
5684        } else {
5685                if (xferlen && (kdcmd->flags & MFI_FRAME_DIR_READ)) {
5686                        for (i = 0; i < xferlen; i++) {
5687                                if (ddi_copyout(
5688                                    (uint8_t *)dcmd_dma_obj.buffer + i,
5689                                    (uint8_t *)ubuf + i,
5690                                    1, mode)) {
5691                                        con_log(CL_ANN, (CE_WARN,
5692                                            "issue_mfi_dcmd : "
```



```
5693                                            "copy to user space failed"));
5694                                        return (DDI_FAILURE);
5695                                }
5696                        }
5697                }
5698        }

5700        kdcmd->cmd_status = ddi_get8(acc_handle, &dcmd->cmd_status);
5701        con_log(CL_ANN,
5702            (CE_CONT, "issue_mfi_dcmd: cmd_status %x", kdcmd->cmd_status));
5703        DTRACE_PROBE3(issue_dcmd, uint32_t, kdcmd->opcode, uint8_t,
5704            kdcmd->cmd, uint8_t, kdcmd->cmd_status);

5706        if (xferlen) {
5707                /* free kernel buffer */
5708                if (mrsas_free_dma_obj(instance, dcmd_dma_obj) != DDI_SUCCESS)
5709                        return (DDI_FAILURE);
5710        }

5712        return (DDI_SUCCESS);
5713 }

5715 /*
5716  * issue_mfi_smp
5717  */
5718 static int
5719 issue_mfi_smp(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5720     struct mrsas_cmd *cmd, int mode)
5721 {
5722        void            *request_ubuf;
5723        void            *response_ubuf;
5724        uint32_t        request_xferlen = 0;
5725        uint32_t        response_xferlen = 0;
5726        uint32_t        new_xfer_length1 = 0;
5727        uint32_t        new_xfer_length2 = 0;
5728        uint_t          model;
5729        dma_obj_t                       request_dma_obj;
5730        dma_obj_t                       response_dma_obj;
5731        ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5732        struct mrsas_smp_frame          *ksmp;
5733        struct mrsas_smp_frame          *smp;
5734        struct mrsas_sge32              *sge32;
5735 #ifndef _ILP32
5736        struct mrsas_sge64              *sge64;
5737 #endif
5738        int i;
5739        uint64_t                        tmp_sas_addr;

5741        smp = &cmd->frame->smp;
5742        ksmp = (struct mrsas_smp_frame *)&ioctl->frame[0];

5744        if (instance->adapterresetinprogress) {
5745                con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
4458                con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
5746                "returning mfi_pkt and setting TRAN_BUSY\n"));
5747                return (DDI_FAILURE);
5748        }
5749        model = ddi_model_convert_from(mode & FMODELS);
5750        if (model == DDI_MODEL_ILP32) {
5751                con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_ILP32"));
4464                con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: DDI_MODEL_ILP32"));

5753                sge32                   = &ksmp->sgl[0].sge32[0];
5754                response_xferlen        = sge32[0].length;
5755                request_xferlen         = sge32[1].length;
5756                con_log(CL_ANN, (CE_CONT, "issue_mfi_smp: "
```

```
4469                 con_log(CL_ANN, (CE_NOTE, "issue_mfi_smp: "
5757                     "response_xferlen = %x, request_xferlen = %x",
5758                     response_xferlen, request_xferlen));

5760                 response_ubuf   = (void *)(ulong_t)sge32[0].phys_addr;
5761                 request_ubuf    = (void *)(ulong_t)sge32[1].phys_addr;
5762                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: "
4475                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: "
5763                     "response_ubuf = %p, request_ubuf = %p",
5764                     response_ubuf, request_ubuf));
5765         } else {
5766 #ifdef _ILP32
5767                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_ILP32"));
4480                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: DDI_MODEL_ILP32"));

5769                 sge32                   = &ksmp->sgl[0].sge32[0];
5770                 response_xferlen        = sge32[0].length;
5771                 request_xferlen         = sge32[1].length;
5772                 con_log(CL_ANN, (CE_CONT, "issue_mfi_smp: "
4485                 con_log(CL_ANN, (CE_NOTE, "issue_mfi_smp: "
5773                     "response_xferlen = %x, request_xferlen = %x",
5774                     response_xferlen, request_xferlen));

5776                 response_ubuf   = (void *)(ulong_t)sge32[0].phys_addr;
5777                 request_ubuf    = (void *)(ulong_t)sge32[1].phys_addr;
5778                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: "
4491                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: "
5779                     "response_ubuf = %p, request_ubuf = %p",
5780                     response_ubuf, request_ubuf));
5781 #else
5782                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp: DDI_MODEL_LP64"));
4495                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: DDI_MODEL_LP64"));

5784                 sge64                   = &ksmp->sgl[0].sge64[0];
5785                 response_xferlen        = sge64[0].length;
5786                 request_xferlen         = sge64[1].length;

5788                 response_ubuf   = (void *)(ulong_t)sge64[0].phys_addr;
5789                 request_ubuf    = (void *)(ulong_t)sge64[1].phys_addr;
5790 #endif
5791         }
5792         if (request_xferlen) {
5793                 /* means IOCTL requires DMA */
5794                 /* allocate the data transfer buffer */
5795                 /* request_dma_obj.size = request_xferlen; */
5796                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(request_xferlen,
5797                     new_xfer_length1, PAGESIZE);
5798                 request_dma_obj.size = new_xfer_length1;
4508                 request_dma_obj.size = request_xferlen;
5799                 request_dma_obj.dma_attr = mrsas_generic_dma_attr;
5800                 request_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5801                 request_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5802                 request_dma_obj.dma_attr.dma_attr_sgllen = 1;
5803                 request_dma_obj.dma_attr.dma_attr_align = 1;

5805                 /* allocate kernel buffer for DMA */
5806                 if (mrsas_alloc_dma_obj(instance, &request_dma_obj,
5807                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5808                         con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5809                             "could not allocate data transfer buffer."));
5810                         return (DDI_FAILURE);
5811                 }
5812                 (void) memset(request_dma_obj.buffer, 0, request_xferlen);

5814                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5815                 for (i = 0; i < request_xferlen; i++) {
```



```
5816                         if (ddi_copyin((uint8_t *)request_ubuf + i,
5817                             (uint8_t *)request_dma_obj.buffer + i,
5818                             1, mode)) {
5819                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5820                                     "copy from user space failed"));
5821                                 return (DDI_FAILURE);
5822                         }
5823                 }
5824         }

5826         if (response_xferlen) {
5827                 /* means IOCTL requires DMA */
5828                 /* allocate the data transfer buffer */
5829                 /* response_dma_obj.size = response_xferlen; */
5830                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(response_xferlen,
5831                     new_xfer_length2, PAGESIZE);
5832                 response_dma_obj.size = new_xfer_length2;
4539                 response_dma_obj.size = response_xferlen;
5833                 response_dma_obj.dma_attr = mrsas_generic_dma_attr;
5834                 response_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
5835                 response_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
5836                 response_dma_obj.dma_attr.dma_attr_sgllen = 1;
5837                 response_dma_obj.dma_attr.dma_attr_align = 1;

5839                 /* allocate kernel buffer for DMA */
5840                 if (mrsas_alloc_dma_obj(instance, &response_dma_obj,
5841                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
5842                         con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5843                             "could not allocate data transfer buffer."));
5844                         return (DDI_FAILURE);
5845                 }
5846                 (void) memset(response_dma_obj.buffer, 0, response_xferlen);

5848                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
5849                 for (i = 0; i < response_xferlen; i++) {
5850                         if (ddi_copyin((uint8_t *)response_ubuf + i,
5851                             (uint8_t *)response_dma_obj.buffer + i,
5852                             1, mode)) {
5853                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_smp: "
5854                                     "copy from user space failed"));
5855                                 return (DDI_FAILURE);
5856                         }
5857                 }
5858         }

5860         ddi_put8(acc_handle, &smp->cmd, ksmp->cmd);
5861         ddi_put8(acc_handle, &smp->cmd_status, 0);
5862         ddi_put8(acc_handle, &smp->connection_status, 0);
5863         ddi_put8(acc_handle, &smp->sge_count, ksmp->sge_count);
5864         /* smp->context         = ksmp->context; */
5865         ddi_put16(acc_handle, &smp->timeout, ksmp->timeout);
5866         ddi_put32(acc_handle, &smp->data_xfer_len, ksmp->data_xfer_len);

5868         bcopy((void *)&ksmp->sas_addr, (void *)&tmp_sas_addr,
5869             sizeof (uint64_t));
5870         ddi_put64(acc_handle, &smp->sas_addr, tmp_sas_addr);

5872         ddi_put16(acc_handle, &smp->flags, ksmp->flags & ~MFI_FRAME_SGL64);

5874         model = ddi_model_convert_from(mode & FMODELS);
5875         if (model == DDI_MODEL_ILP32) {
5876                 con_log(CL_ANN1, (CE_CONT,
4583                 con_log(CL_ANN1, (CE_NOTE,
5877                     "issue_mfi_smp: DDI_MODEL_ILP32"));

5879                 sge32 = &smp->sgl[0].sge32[0];
```

```
5880                ddi_put32(acc_handle, &sge32[0].length, response_xferlen);
5881                ddi_put32(acc_handle, &sge32[0].phys_addr,
5882                    response_dma_obj.dma_cookie[0].dmac_address);
5883                ddi_put32(acc_handle, &sge32[1].length, request_xferlen);
5884                ddi_put32(acc_handle, &sge32[1].phys_addr,
5885                    request_dma_obj.dma_cookie[0].dmac_address);
5886        } else {
5887 #ifdef _ILP32
5888                con_log(CL_ANN1, (CE_CONT,
4595                con_log(CL_ANN1, (CE_NOTE,
5889                    "issue_mfi_smp: DDI_MODEL_ILP32"));
5890                sge32 = &smp->sgl[0].sge32[0];
5891                ddi_put32(acc_handle, &sge32[0].length, response_xferlen);
5892                ddi_put32(acc_handle, &sge32[0].phys_addr,
5893                    response_dma_obj.dma_cookie[0].dmac_address);
5894                ddi_put32(acc_handle, &sge32[1].length, request_xferlen);
5895                ddi_put32(acc_handle, &sge32[1].phys_addr,
5896                    request_dma_obj.dma_cookie[0].dmac_address);
5897 #else
5898                con_log(CL_ANN1, (CE_CONT,
4605                con_log(CL_ANN1, (CE_NOTE,
5899                    "issue_mfi_smp: DDI_MODEL_LP64"));
5900                sge64 = &smp->sgl[0].sge64[0];
5901                ddi_put32(acc_handle, &sge64[0].length, response_xferlen);
5902                ddi_put64(acc_handle, &sge64[0].phys_addr,
5903                    response_dma_obj.dma_cookie[0].dmac_address);
5904                ddi_put32(acc_handle, &sge64[1].length, request_xferlen);
5905                ddi_put64(acc_handle, &sge64[1].phys_addr,
5906                    request_dma_obj.dma_cookie[0].dmac_address);
5907 #endif
5908        }
5909        con_log(CL_ANN1, (CE_CONT, "issue_mfi_smp : "
4616        con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp : "
5910            "smp->response_xferlen = %d, smp->request_xferlen = %d "
5911            "smp->data_xfer_len = %d", ddi_get32(acc_handle, &sge32[0].length),
5912            ddi_get32(acc_handle, &sge32[1].length),
5913            ddi_get32(acc_handle, &smp->data_xfer_len)));

5915        cmd->sync_cmd = MRSAS_TRUE;
5916        cmd->frame_count = 1;

5918        if (instance->tbolt) {
5919                mr_sas_tbolt_build_mfi_cmd(instance, cmd);
5920        }

5922        if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
5923                con_log(CL_ANN, (CE_WARN,
5924                    "issue_mfi_smp: fw_ioctl failed"));
5925        } else {
5926                con_log(CL_ANN1, (CE_CONT,
4629                con_log(CL_ANN1, (CE_NOTE,
5927                    "issue_mfi_smp: copy to user space"));

5929                if (request_xferlen) {
5930                        for (i = 0; i < request_xferlen; i++) {
5931                                if (ddi_copyout(
5932                                    (uint8_t *)request_dma_obj.buffer +
5933                                    i, (uint8_t *)request_ubuf + i,
5934                                    1, mode)) {
5935                                        con_log(CL_ANN, (CE_WARN,
5936                                            "issue_mfi_smp : copy to user space"
5937                                            " failed"));
5938                                        return (DDI_FAILURE);
5939                                }
5940                        }
5941                }
```



```
5943                if (response_xferlen) {
5944                        for (i = 0; i < response_xferlen; i++) {
5945                                if (ddi_copyout(
5946                                    (uint8_t *)response_dma_obj.buffer
5947                                    + i, (uint8_t *)response_ubuf
5948                                    + i, 1, mode)) {
5949                                        con_log(CL_ANN, (CE_WARN,
5950                                            "issue_mfi_smp : copy to "
5951                                            "user space failed"));
5952                                        return (DDI_FAILURE);
5953                                }
5954                        }
5955                }
5956        }

5958        ksmp->cmd_status = ddi_get8(acc_handle, &smp->cmd_status);
5959        con_log(CL_ANN1, (CE_NOTE, "issue_mfi_smp: smp->cmd_status = %d",
5960            ksmp->cmd_status));
4663            ddi_get8(acc_handle, &smp->cmd_status)));
5961        DTRACE_PROBE2(issue_smp, uint8_t, ksmp->cmd, uint8_t, ksmp->cmd_status);

5963        if (request_xferlen) {
5964                /* free kernel buffer */
5965                if (mrsas_free_dma_obj(instance, request_dma_obj) !=
5966                    DDI_SUCCESS)
5967                        return (DDI_FAILURE);
5968        }

5970        if (response_xferlen) {
5971                /* free kernel buffer */
5972                if (mrsas_free_dma_obj(instance, response_dma_obj) !=
5973                    DDI_SUCCESS)
5974                        return (DDI_FAILURE);
5975        }

5977        return (DDI_SUCCESS);
5978 }

5980 /*
5981  * issue_mfi_stp
5982  */
5983 static int
5984 issue_mfi_stp(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
5985     struct mrsas_cmd *cmd, int mode)
5986 {
5987        void            *fis_ubuf;
5988        void            *data_ubuf;
5989        uint32_t        fis_xferlen = 0;
5990        uint32_t   new_xfer_length1 = 0;
5991        uint32_t   new_xfer_length2 = 0;
5992        uint32_t        data_xferlen = 0;
5993        uint_t          model;
5994        dma_obj_t       fis_dma_obj;
5995        dma_obj_t       data_dma_obj;
5996        struct mrsas_stp_frame  *kstp;
5997        struct mrsas_stp_frame  *stp;
5998        ddi_acc_handle_t        acc_handle = cmd->frame_dma_obj.acc_handle;
5999        int i;

6001        stp = &cmd->frame->stp;
6002        kstp = (struct mrsas_stp_frame *)&ioctl->frame[0];

6004        if (instance->adapterresetinprogress) {
6005                con_log(CL_ANN1, (CE_WARN, "Reset flag set, "
4706                con_log(CL_ANN1, (CE_NOTE, "Reset flag set, "
```

```
6006                 "returning mfi_pkt and setting TRAN_BUSY\n"));
6007                 return (DDI_FAILURE);
6008         }
6009         model = ddi_model_convert_from(mode & FMODELS);
6010         if (model == DDI_MODEL_ILP32) {
6011                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_ILP32"));
4712                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_stp: DDI_MODEL_ILP32"));

6013                 fis_xferlen     = kstp->sgl.sge32[0].length;
6014                 data_xferlen    = kstp->sgl.sge32[1].length;

6016                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge32[0].phys_addr;
6017                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge32[1].phys_addr;
6018         } else {
4719         }
4720         else
4721         {
6019 #ifdef _ILP32
6020                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_ILP32"));
4723                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_stp: DDI_MODEL_ILP32"));

6022                 fis_xferlen     = kstp->sgl.sge32[0].length;
6023                 data_xferlen    = kstp->sgl.sge32[1].length;

6025                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge32[0].phys_addr;
6026                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge32[1].phys_addr;
6027 #else
6028                 con_log(CL_ANN1, (CE_CONT, "issue_mfi_stp: DDI_MODEL_LP64"));
4731                 con_log(CL_ANN1, (CE_NOTE, "issue_mfi_stp: DDI_MODEL_LP64"));

6030                 fis_xferlen     = kstp->sgl.sge64[0].length;
6031                 data_xferlen    = kstp->sgl.sge64[1].length;

6033                 fis_ubuf        = (void *)(ulong_t)kstp->sgl.sge64[0].phys_addr;
6034                 data_ubuf       = (void *)(ulong_t)kstp->sgl.sge64[1].phys_addr;
6035 #endif
6036         }


6039         if (fis_xferlen) {
6040                 con_log(CL_ANN, (CE_CONT, "issue_mfi_stp: "
4743                 con_log(CL_ANN, (CE_NOTE, "issue_mfi_stp: "
6041                     "fis_ubuf = %p fis_xferlen = %x", fis_ubuf, fis_xferlen));

6043                 /* means IOCTL requires DMA */
6044                 /* allocate the data transfer buffer */
6045                 /* fis_dma_obj.size = fis_xferlen; */
6046                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(fis_xferlen,
6047                     new_xfer_length1, PAGESIZE);
6048                 fis_dma_obj.size = new_xfer_length1;
4748                 fis_dma_obj.size = fis_xferlen;
6049                 fis_dma_obj.dma_attr = mrsas_generic_dma_attr;
6050                 fis_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
6051                 fis_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
6052                 fis_dma_obj.dma_attr.dma_attr_sgllen = 1;
6053                 fis_dma_obj.dma_attr.dma_attr_align = 1;

6055                 /* allocate kernel buffer for DMA */
6056                 if (mrsas_alloc_dma_obj(instance, &fis_dma_obj,
6057                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
6058                         con_log(CL_ANN, (CE_WARN, "issue_mfi_stp : "
6059                             "could not allocate data transfer buffer."));
6060                         return (DDI_FAILURE);
6061                 }
6062                 (void) memset(fis_dma_obj.buffer, 0, fis_xferlen);
```



```
6064                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
6065                 for (i = 0; i < fis_xferlen; i++) {
6066                         if (ddi_copyin((uint8_t *)fis_ubuf + i,
6067                             (uint8_t *)fis_dma_obj.buffer + i, 1, mode)) {
6068                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6069                                     "copy from user space failed"));
6070                                 return (DDI_FAILURE);
6071                         }
6072                 }
6073         }

6075         if (data_xferlen) {
6076                 con_log(CL_ANN, (CE_CONT, "issue_mfi_stp: data_ubuf = %p "
4776                 con_log(CL_ANN, (CE_NOTE, "issue_mfi_stp: data_ubuf = %p "
6077                     "data_xferlen = %x", data_ubuf, data_xferlen));

6079                 /* means IOCTL requires DMA */
6080                 /* allocate the data transfer buffer */
6081                 /* data_dma_obj.size = data_xferlen; */
6082                 MRSAS_GET_BOUNDARY_ALIGNED_LEN(data_xferlen, new_xfer_length2,
6083                     PAGESIZE);
6084                 data_dma_obj.size = new_xfer_length2;
4781                 data_dma_obj.size = data_xferlen;
6085                 data_dma_obj.dma_attr = mrsas_generic_dma_attr;
6086                 data_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
6087                 data_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
6088                 data_dma_obj.dma_attr.dma_attr_sgllen = 1;
6089                 data_dma_obj.dma_attr.dma_attr_align = 1;

6091                 /* allocate kernel buffer for DMA */
4788 /* allocate kernel buffer for DMA */
6092                 if (mrsas_alloc_dma_obj(instance, &data_dma_obj,
6093                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
6094                         con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6095                             "could not allocate data transfer buffer."));
6096                         return (DDI_FAILURE);
6097                 }
6098                 (void) memset(data_dma_obj.buffer, 0, data_xferlen);

6100                 /* If IOCTL requires DMA WRITE, do ddi_copyin IOCTL data copy */
6101                 for (i = 0; i < data_xferlen; i++) {
6102                         if (ddi_copyin((uint8_t *)data_ubuf + i,
6103                             (uint8_t *)data_dma_obj.buffer + i, 1, mode)) {
6104                                 con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: "
6105                                     "copy from user space failed"));
6106                                 return (DDI_FAILURE);
6107                         }
6108                 }
6109         }

6111         ddi_put8(acc_handle, &stp->cmd, kstp->cmd);
6112         ddi_put8(acc_handle, &stp->cmd_status, 0);
6113         ddi_put8(acc_handle, &stp->connection_status, 0);
6114         ddi_put8(acc_handle, &stp->target_id, kstp->target_id);
6115         ddi_put8(acc_handle, &stp->sge_count, kstp->sge_count);

6117         ddi_put16(acc_handle, &stp->timeout, kstp->timeout);
6118         ddi_put32(acc_handle, &stp->data_xfer_len, kstp->data_xfer_len);

6120         ddi_rep_put8(acc_handle, (uint8_t *)kstp->fis, (uint8_t *)stp->fis, 10,
6121             DDI_DEV_AUTOINCR);

6123         ddi_put16(acc_handle, &stp->flags, kstp->flags & ~MFI_FRAME_SGL64);
6124         ddi_put32(acc_handle, &stp->stp_flags, kstp->stp_flags);
6125         ddi_put32(acc_handle, &stp->sgl.sge32[0].length, fis_xferlen);
6126         ddi_put32(acc_handle, &stp->sgl.sge32[0].phys_addr,
```

```
6127             fis_dma_obj.dma_cookie[0].dmac_address);
6128         ddi_put32(acc_handle, &stp->sgl.sge32[1].length, data_xferlen);
6129         ddi_put32(acc_handle, &stp->sgl.sge32[1].phys_addr,
6130             data_dma_obj.dma_cookie[0].dmac_address);

6132         cmd->sync_cmd = MRSAS_TRUE;
6133         cmd->frame_count = 1;

6135         if (instance->tbolt) {
6136                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
6137         }

6139         if (instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd)) {
6140                 con_log(CL_ANN, (CE_WARN, "issue_mfi_stp: fw_ioctl failed"));
6141         } else {

6143                 if (fis_xferlen) {
6144                         for (i = 0; i < fis_xferlen; i++) {
6145                                 if (ddi_copyout(
6146                                     (uint8_t *)fis_dma_obj.buffer + i,
6147                                     (uint8_t *)fis_ubuf + i, 1, mode)) {
6148                                         con_log(CL_ANN, (CE_WARN,
6149                                             "issue_mfi_stp : copy to "
6150                                             "user space failed"));
6151                                         return (DDI_FAILURE);
6152                                 }
6153                         }
6154                 }
6155         }
6156         if (data_xferlen) {
6157                 for (i = 0; i < data_xferlen; i++) {
6158                         if (ddi_copyout(
6159                             (uint8_t *)data_dma_obj.buffer + i,
6160                             (uint8_t *)data_ubuf + i, 1, mode)) {
6161                                 con_log(CL_ANN, (CE_WARN,
6162                                     "issue_mfi_stp : copy to"
6163                                     " user space failed"));
6164                                 return (DDI_FAILURE);
6165                         }
6166                 }
6167         }

6169         kstp->cmd_status = ddi_get8(acc_handle, &stp->cmd_status);
6170         con_log(CL_ANN1, (CE_NOTE, "issue_mfi_stp: stp->cmd_status = %d",
6171             kstp->cmd_status));
6172         DTRACE_PROBE2(issue_stp, uint8_t, kstp->cmd, uint8_t, kstp->cmd_status);

6174         if (fis_xferlen) {
6175                 /* free kernel buffer */
6176                 if (mrsas_free_dma_obj(instance, fis_dma_obj) != DDI_SUCCESS)
6177                         return (DDI_FAILURE);
6178         }

6180         if (data_xferlen) {
6181                 /* free kernel buffer */
6182                 if (mrsas_free_dma_obj(instance, data_dma_obj) != DDI_SUCCESS)
6183                         return (DDI_FAILURE);
6184         }

6186         return (DDI_SUCCESS);
6187 }

6189 /*
6190  * fill_up_drv_ver
6191  */
6192 void
```



```
4883 static void
6193 fill_up_drv_ver(struct mrsas_drv_ver *dv)
6194 {
6195         (void) memset(dv, 0, sizeof (struct mrsas_drv_ver));

6197         (void) memcpy(dv->signature, "$LSI LOGIC$", strlen("$LSI LOGIC$"));
6198         (void) memcpy(dv->os_name, "Solaris", strlen("Solaris"));
6199         (void) memcpy(dv->drv_name, "mr_sas", strlen("mr_sas"));
6200         (void) memcpy(dv->drv_ver, MRSAS_VERSION, strlen(MRSAS_VERSION));
6201         (void) memcpy(dv->drv_rel_date, MRSAS_RELDATE,
6202             strlen(MRSAS_RELDATE));

6204 }

6206 /*
6207  * handle_drv_ioctl
6208  */
6209 static int
6210 handle_drv_ioctl(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
6211     int mode)
6212 {
6213         int     i;
6214         int     rval = DDI_SUCCESS;
6215         int     *props = NULL;
6216         void    *ubuf;

6218         uint8_t         *pci_conf_buf;
6219         uint32_t        xferlen;
6220         uint32_t        num_props;
6221         uint_t          model;
6222         struct mrsas_dcmd_frame *kdcmd;
6223         struct mrsas_drv_ver    dv;
6224         struct mrsas_pci_information pi;

6226         kdcmd = (struct mrsas_dcmd_frame *)&ioctl->frame[0];

6228         model = ddi_model_convert_from(mode & FMODELS);
6229         if (model == DDI_MODEL_ILP32) {
6230                 con_log(CL_ANN1, (CE_CONT,
4920                 con_log(CL_ANN1, (CE_NOTE,
6231                     "handle_drv_ioctl: DDI_MODEL_ILP32"));

6233                 xferlen = kdcmd->sgl.sge32[0].length;

6235                 ubuf = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
6236         } else {
6237 #ifdef _ILP32
6238                 con_log(CL_ANN1, (CE_CONT,
4928                 con_log(CL_ANN1, (CE_NOTE,
6239                     "handle_drv_ioctl: DDI_MODEL_ILP32"));
6240                 xferlen = kdcmd->sgl.sge32[0].length;
6241                 ubuf = (void *)(ulong_t)kdcmd->sgl.sge32[0].phys_addr;
6242 #else
6243                 con_log(CL_ANN1, (CE_CONT,
4933                 con_log(CL_ANN1, (CE_NOTE,
6244                     "handle_drv_ioctl: DDI_MODEL_LP64"));
6245                 xferlen = kdcmd->sgl.sge64[0].length;
6246                 ubuf = (void *)(ulong_t)kdcmd->sgl.sge64[0].phys_addr;
6247 #endif
6248         }
6249         con_log(CL_ANN1, (CE_CONT, "handle_drv_ioctl: "
4939         con_log(CL_ANN1, (CE_NOTE, "handle_drv_ioctl: "
6250             "dataBuf=%p size=%d bytes", ubuf, xferlen));

6252         switch (kdcmd->opcode) {
6253         case MRSAS_DRIVER_IOCTL_DRIVER_VERSION:
```

```
6254                **con_log(CL_ANN1, (CE_CONT, "handle_drv_ioctl: "**
4944                *con_log(CL_ANN1, (CE_NOTE, "handle_drv_ioctl: "*
6255                    "MRSAS_DRIVER_IOCTL_DRIVER_VERSION"));

6257                fill_up_drv_ver(&dv);

6259                if (ddi_copyout(&dv, ubuf, xferlen, mode)) {
6260                        con_log(CL_ANN, (CE_WARN, "handle_drv_ioctl: "
6261                            "MRSAS_DRIVER_IOCTL_DRIVER_VERSION : "
6262                            "copy to user space failed"));
6263                        kdcmd->cmd_status = 1;
6264                        rval = 1;
6265                } else {
6266                        kdcmd->cmd_status = 0;
6267                }
6268                break;
6269        case MRSAS_DRIVER_IOCTL_PCI_INFORMATION:
6270                con_log(CL_ANN1, (CE_NOTE, "handle_drv_ioctl: "
6271                    "MRSAS_DRIVER_IOCTL_PCI_INFORMAITON"));

6273                if (ddi_prop_lookup_int_array(DDI_DEV_T_ANY, instance->dip,
6274                    0, "reg", &props, &num_props)) {
6275                        con_log(CL_ANN, (CE_WARN, "handle_drv_ioctl: "
6276                            "MRSAS_DRIVER_IOCTL_PCI_INFORMATION : "
6277                            "ddi_prop_look_int_array failed"));
6278                        rval = DDI_FAILURE;
6279                } else {

6281                        pi.busNumber = (props[0] >> 16) & 0xFF;
6282                        pi.deviceNumber = (props[0] >> 11) & 0x1f;
6283                        pi.functionNumber = (props[0] >> 8) & 0x7;
6284                        ddi_prop_free((void *)props);
6285                }

6287                pci_conf_buf = (uint8_t *)&pi.pciHeaderInfo;

6289                for (i = 0; i < (sizeof (struct mrsas_pci_information) -
6290                    offsetof(struct mrsas_pci_information, pciHeaderInfo));
6291                    i++) {
6292                        pci_conf_buf[i] =
6293                            pci_config_get8(instance->pci_handle, i);
6294                }

6296                if (ddi_copyout(&pi, ubuf, xferlen, mode)) {
6297                        con_log(CL_ANN, (CE_WARN, "handle_drv_ioctl: "
6298                            "MRSAS_DRIVER_IOCTL_PCI_INFORMATION : "
6299                            "copy to user space failed"));
6300                        kdcmd->cmd_status = 1;
6301                        rval = 1;
6302                } else {
6303                        kdcmd->cmd_status = 0;
6304                }
6305                break;
6306        default:
6307                con_log(CL_ANN, (CE_WARN, "handle_drv_ioctl: "
6308                    "invalid driver specific IOCTL opcode = 0x%x",
6309                    kdcmd->opcode));
6310                kdcmd->cmd_status = 1;
6311                rval = DDI_FAILURE;
6312                break;
6313        }

6315        return (rval);
6316 }

6318 /*
```



```
6319  * handle_mfi_ioctl
6320  */
6321 static int
6322 handle_mfi_ioctl(struct mrsas_instance *instance, struct mrsas_ioctl *ioctl,
6323     int mode)
6324 {
6325         int     rval = DDI_SUCCESS;

6327         struct mrsas_header     *hdr;
6328         struct mrsas_cmd        *cmd;

6330         **if (instance->tbolt) {**
6331                 **cmd = get_raid_msg_mfi_pkt(instance);**
6332         **} else {**
6333                 cmd = get_mfi_pkt(instance);
6334         **}**

6335         if (!cmd) {
6336                 con_log(CL_ANN, (CE_WARN, "mr_sas: "
6337                     "failed to get a cmd packet"));
6338                 DTRACE_PROBE2(mfi_ioctl_err, uint16_t,
6339                     instance->fw_outstanding, uint16_t, instance->max_fw_cmds);
6340                 return (DDI_FAILURE);
6341         }
5029         *cmd->retry_count_for_ocr = 0;*

6343         /* Clear the frame buffer and assign back the context id */
6344         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6345         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6346             cmd->index);

6348         hdr = (struct mrsas_header *)&ioctl->frame[0];

6350         switch (ddi_get8(cmd->frame_dma_obj.acc_handle, &hdr->cmd)) {
6351         case MFI_CMD_OP_DCMD:
6352                 rval = issue_mfi_dcmd(instance, ioctl, cmd, mode);
6353                 break;
6354         case MFI_CMD_OP_SMP:
6355                 rval = issue_mfi_smp(instance, ioctl, cmd, mode);
6356                 break;
6357         case MFI_CMD_OP_STP:
6358                 rval = issue_mfi_stp(instance, ioctl, cmd, mode);
6359                 break;
6360         case MFI_CMD_OP_LD_SCSI:
6361         case MFI_CMD_OP_PD_SCSI:
6362                 rval = issue_mfi_pthru(instance, ioctl, cmd, mode);
6363                 break;
6364         default:
6365                 con_log(CL_ANN, (CE_WARN, "handle_mfi_ioctl: "
6366                     "invalid mfi ioctl hdr->cmd = %d", hdr->cmd));
6367                 rval = DDI_FAILURE;
6368                 break;
6369         }

6371         if (mrsas_common_check(instance, cmd) != DDI_SUCCESS)
6372                 rval = DDI_FAILURE;

6374         **if (instance->tbolt) {**
6375                 **return_raid_msg_mfi_pkt(instance, cmd);**
6376         **} else {**
6377                 return_mfi_pkt(instance, cmd);
6378         **}**

6380         return (rval);
6381 }
_____***unchanged_portion_omitted_***
```

```
6399 static int
6400 register_mfi_aen(struct mrsas_instance *instance, uint32_t seq_num,
6401     uint32_t class_locale_word)
6402 {
6403         int     ret_val;

6405         struct mrsas_cmd        *cmd, *aen_cmd;
6406         struct mrsas_dcmd_frame *dcmd;
6407         union mrsas_evt_class_locale    curr_aen;
6408         union mrsas_evt_class_locale    prev_aen;

6410         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
6411         /*
6412          * If there an AEN pending already (aen_cmd), check if the
6413          * class_locale of that pending AEN is inclusive of the new
6414          * AEN request we currently have. If it is, then we don't have
6415          * to do anything. In other words, whichever events the current
6416          * AEN request is subscribing to, have already been subscribed
6417          * to.
6418          *
6419          * If the old_cmd is _not_ inclusive, then we have to abort
6420          * that command, form a class_locale that is superset of both
6421          * old and current and re-issue to the FW
6422          */

6424         curr_aen.word = LE_32(class_locale_word);
6425         curr_aen.members.locale = LE_16(curr_aen.members.locale);
6426         aen_cmd = instance->aen_cmd;
6427         if (aen_cmd) {
6428                 prev_aen.word = ddi_get32(aen_cmd->frame_dma_obj.acc_handle,
6429                     &aen_cmd->frame->dcmd.mbox.w[1]);
6430                 prev_aen.word = LE_32(prev_aen.word);
6431                 prev_aen.members.locale = LE_16(prev_aen.members.locale);
6432                 /*
6433                  * A class whose enum value is smaller is inclusive of all
6434                  * higher values. If a PROGRESS (= -1) was previously
6435                  * registered, then a new registration requests for higher
6436                  * classes need not be sent to FW. They are automatically
6437                  * included.
6438                  *
6439                  * Locale numbers don't have such hierarchy. They are bitmap
6440                  * values
6441                  */
6442                 if ((prev_aen.members.class <= curr_aen.members.class) &&
6443                     !((prev_aen.members.locale & curr_aen.members.locale) ^
6444                     curr_aen.members.locale)) {
6445                         /*
6446                          * Previously issued event registration includes
6447                          * current request. Nothing to do.
6448                          */

6450                         return (0);
6451                 } else {
6452                         curr_aen.members.locale |= prev_aen.members.locale;

6454                         if (prev_aen.members.class < curr_aen.members.class)
6455                                 curr_aen.members.class = prev_aen.members.class;

6457                         ret_val = abort_aen_cmd(instance, aen_cmd);

6459                         if (ret_val) {
6460                                 con_log(CL_ANN, (CE_WARN, "register_mfi_aen: "
6461                                     "failed to abort prevous AEN command"));

6463                                 return (ret_val);
```



```
6464                         }
6465                 }
6466         } else {
6467                 curr_aen.word = LE_32(class_locale_word);
6468                 curr_aen.members.locale = LE_16(curr_aen.members.locale);
6469         }

6471         if (instance->tbolt) {
6472                 cmd = get_raid_msg_mfi_pkt(instance);
6473         } else {
6474                 cmd = get_mfi_pkt(instance);
6475         }

6477         if (!cmd) {
6478                 DTRACE_PROBE2(mfi_aen_err, uint16_t, instance->fw_outstanding,
6479                     uint16_t, instance->max_fw_cmds);
6480                 return (ENOMEM);
6481         }

5161         cmd->retry_count_for_ocr = 0;
6483         /* Clear the frame buffer and assign back the context id */
6484         (void) memset((char *)&cmd->frame[0], 0, sizeof (union mrsas_frame));
6485         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
6486             cmd->index);

6488         dcmd = &cmd->frame->dcmd;

6490         /* for(i = 0; i < DCMD_MBOX_SZ; i++) dcmd->mbox.b[i] = 0; */
6491         (void) memset(dcmd->mbox.b, 0, DCMD_MBOX_SZ);

6493         (void) memset(instance->mfi_evt_detail_obj.buffer, 0,
6494             sizeof (struct mrsas_evt_detail));

6496         /* Prepare DCMD for aen registration */
6497         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
6498         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0x0);
6499         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
6500         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
6501             MFI_FRAME_DIR_READ);
6502         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
6503         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
6504             sizeof (struct mrsas_evt_detail));
6505         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
6506             MR_DCMD_CTRL_EVENT_WAIT);
6507         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], seq_num);
6508         curr_aen.members.locale = LE_16(curr_aen.members.locale);
6509         curr_aen.word = LE_32(curr_aen.word);
6510         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[1],
6511             curr_aen.word);
6512         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
6513             instance->mfi_evt_detail_obj.dma_cookie[0].dmac_address);
6514         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
6515             sizeof (struct mrsas_evt_detail));

6517         instance->aen_seq_num = seq_num;


6520         /*
6521          * Store reference to the cmd used to register for AEN. When an
6522          * application wants us to register for AEN, we have to abort this
6523          * cmd and re-register with a new EVENT LOCALE supplied by that app
6524          */
6525         instance->aen_cmd = cmd;

6527         cmd->frame_count = 1;
```

```
6529         /* Issue the aen registration frame */
6530         /* atomic_add_16 (&instance->fw_outstanding, 1); */
6531         if (instance->tbolt) {
6532                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
6533         }
6534         instance->func_ptr->issue_cmd(cmd, instance);

6536         return (0);
6537 }

6539 void
5215 static void
6540 display_scsi_inquiry(caddr_t scsi_inq)
6541 {
6542 #define MAX_SCSI_DEVICE_CODE    14
6543         int             i;
6544         char            inquiry_buf[256] = {0};
6545         int             len;
6546         const char      *const scsi_device_types[] = {
6547                 "Direct-Access    ",
6548                 "Sequential-Access",
6549                 "Printer          ",
6550                 "Processor        ",
6551                 "WORM             ",
6552                 "CD-ROM           ",
6553                 "Scanner          ",
6554                 "Optical Device   ",
6555                 "Medium Changer   ",
6556                 "Communications   ",
6557                 "Unknown          ",
6558                 "Unknown          ",
6559                 "Unknown          ",
6560                 "Enclosure        ",
6561         };

6563         len = 0;

6565         len += snprintf(inquiry_buf + len, 265 - len, "  Vendor: ");
6566         for (i = 8; i < 16; i++) {
6567                 len += snprintf(inquiry_buf + len, 265 - len, "%c",
6568                     scsi_inq[i]);
6569         }

6571         len += snprintf(inquiry_buf + len, 265 - len, "  Model: ");

6573         for (i = 16; i < 32; i++) {
6574                 len += snprintf(inquiry_buf + len, 265 - len, "%c",
6575                     scsi_inq[i]);
6576         }

6578         len += snprintf(inquiry_buf + len, 265 - len, "  Rev: ");

6580         for (i = 32; i < 36; i++) {
6581                 len += snprintf(inquiry_buf + len, 265 - len, "%c",
6582                     scsi_inq[i]);
6583         }

6585         len += snprintf(inquiry_buf + len, 265 - len, "\n");


6588         i = scsi_inq[0] & 0x1f;


6591         len += snprintf(inquiry_buf + len, 265 - len, "  Type:   %s ",
6592             i < MAX_SCSI_DEVICE_CODE ? scsi_device_types[i] :
6593             "Unknown          ");
```



```
6596         len += snprintf(inquiry_buf + len, 265 - len,
6597             "                 ANSI SCSI revision: %02x", scsi_inq[2] & 0x07);

6599         if ((scsi_inq[2] & 0x07) == 1 && (scsi_inq[3] & 0x0f) == 1) {
6600                 len += snprintf(inquiry_buf + len, 265 - len, " CCS\n");
6601         } else {
6602                 len += snprintf(inquiry_buf + len, 265 - len, "\n");
6603         }

6605         con_log(CL_DLEVEL2, (CE_CONT, inquiry_buf));
5281         con_log(CL_ANN1, (CE_CONT, inquiry_buf));
6606 }

6608 static void
6609 io_timeout_checker(void *arg)
6610 {
6611         struct scsi_pkt *pkt;
6612         struct mrsas_instance *instance = arg;
6613         struct mrsas_cmd        *cmd = NULL;
6614         struct mrsas_header     *hdr;
6615         int time = 0;
6616         int counter = 0;
6617         struct mlist_head       *pos, *next;
6618         mlist_t                 process_list;

6620         if (instance->adapterresetinprogress == 1) {
6621                 con_log(CL_ANN, (CE_NOTE, "io_timeout_checker:"
5297                 con_log(CL_ANN1, (CE_NOTE, "io_timeout_checker"
6622                     " reset in progress"));

6624                 instance->timeout_id = timeout(io_timeout_checker,
6625                     (void *) instance, drv_usectohz(MRSAS_1_SECOND));
6626                 return;
6627         }

6629         /* See if this check needs to be in the beginning or last in ISR */
6630         if (mrsas_initiate_ocr_if_fw_is_faulty(instance) ==  1) {
6631                 cmn_err(CE_WARN, "io_timeout_checker: "
6632                     "FW Fault, calling reset adapter");
6633                 cmn_err(CE_CONT, "io_timeout_checker: "
6634                     "fw_outstanding 0x%X max_fw_cmds 0x%X",
6635                     instance->fw_outstanding, instance->max_fw_cmds);
5306                 con_log(CL_ANN1, (CE_NOTE,
5307                     "Fw Fault state Handling in io_timeout_checker"));
6636                 if (instance->adapterresetinprogress == 0) {
6637                         instance->adapterresetinprogress = 1;
6638                         if (instance->tbolt)
6639                                 (void) mrsas_tbolt_reset_ppc(instance);
6640                         else
6641                                 (void) mrsas_reset_ppc(instance);
6642                         instance->adapterresetinprogress = 0;
6643                 }
6644                 instance->timeout_id = timeout(io_timeout_checker,
6645                     (void *) instance, drv_usectohz(MRSAS_1_SECOND));
6646                 return;
6647         }

6649         INIT_LIST_HEAD(&process_list);

6651         mutex_enter(&instance->cmd_pend_mtx);
6652         mlist_for_each_safe(pos, next, &instance->cmd_pend_list) {
6653                 cmd = mlist_entry(pos, struct mrsas_cmd, list);

6655                 if (cmd == NULL) {
```

```
6656                    continue;
6657            }

6659            if (cmd->sync_cmd == MRSAS_TRUE) {
6660                    hdr = (struct mrsas_header *)&cmd->frame->hdr;
6661                    if (hdr == NULL) {
6662                            continue;
6663                    }
6664                    time = --cmd->drv_pkt_time;
6665            } else {
6666                    pkt = cmd->pkt;
6667                    if (pkt == NULL) {
6668                            continue;
6669                    }
6670                    time = --cmd->drv_pkt_time;
6671            }
6672            if (time <= 0) {
6673                    cmn_err(CE_WARN, "%llx: "
6674                        "io_timeout_checker: TIMING OUT: pkt: %p, "
6675                        "cmd %p fw_outstanding 0x%X max_fw_cmds 0x%X\n",
6676                        gethrtime(), (void *)pkt, (void *)cmd,
6677                        instance->fw_outstanding, instance->max_fw_cmds);

5340                    con_log(CL_ANN1, (CE_NOTE, "%llx: "
5341                        "io_timeout_checker: TIMING OUT: pkt "
5342                        ": %p, cmd %p", gethrtime(), (void *)pkt,
5343                        (void *)cmd));
6679                    counter++;
6680                    break;
6681            }
6682    }
6683    mutex_exit(&instance->cmd_pend_mtx);

6685    if (counter) {
5351            con_log(CL_ANN1, (CE_NOTE,
5352                "io_timeout_checker "
5353                "cmd->retrycount_for_ocr %d, "
5354                "cmd index %d , cmd address %p ",
5355                cmd->retry_count_for_ocr+1, cmd->index, (void *)cmd));

6686            if (instance->disable_online_ctrl_reset == 1) {
6687                    cmn_err(CE_WARN, "mr_sas %d: %s(): OCR is NOT "
6688                        "supported by Firmware, KILL adapter!!!",
6689                        instance->instance, __func__);
5358                    con_log(CL_ANN1, (CE_NOTE, "mrsas: "
5359                        "OCR is not supported by the Firmware "
5360                        "Failing all the queued packets \n"));

6691                    if (instance->tbolt)
6692                            mrsas_tbolt_kill_adapter(instance);
6693                    else
6694                            (void) mrsas_kill_adapter(instance);

6696                    return;
6697            } else {
6698                    if (cmd->retry_count_for_ocr <=  IO_RETRY_COUNT) {
6699                            if (instance->adapterresetinprogress == 0) {
6700                                    if (instance->tbolt) {
6701                                            (void) mrsas_tbolt_reset_ppc(
6702                                                instance);
6703                                    } else {
6704                                            (void) mrsas_reset_ppc(
6705                                                instance);
5367                                    con_log(CL_ANN1, (CE_NOTE, "mrsas: "
5368                                        "OCR is supported by FW "
5369                                        "triggering  mrsas_reset_ppc"));
```



```
5370                                    (void) mrsas_reset_ppc(instance);
6706                                    }
6707                            }
6708                    } else {
6709                            cmn_err(CE_WARN,
6710                                "io_timeout_checker: "
6711                                "cmd %p cmd->index %d "
5373                            con_log(CL_ANN1, (CE_NOTE,
5374                                "io_timeout_checker:"
5375                                " cmdindex: %d,cmd address: %p "
6712                                "timed out even after 3 resets: "
6713                                "so KILL adapter", (void *)cmd, cmd->index);

6715                            mrsas_print_cmd_details(instance, cmd, 0xDD);

6717                            if (instance->tbolt)
6718                                    mrsas_tbolt_kill_adapter(instance);
6719                            else
5377                                "so kill adapter", cmd->index,
5378                                (void *)cmd));
6720                                    (void) mrsas_kill_adapter(instance);
6721                            return;
6722                    }
6723            }
6724    }
6725    con_log(CL_ANN, (CE_NOTE, "mrsas: "


5386    con_log(CL_ANN1, (CE_NOTE, "mrsas: "
6726        "schedule next timeout check: "
6727        "do timeout \n"));
6728    instance->timeout_id =
6729        timeout(io_timeout_checker, (void *)instance,
6730        drv_usectohz(MRSAS_1_SECOND));
6731 }

6733 static uint32_t
5393 static int
6734 read_fw_status_reg_ppc(struct mrsas_instance *instance)
6735 {
6736    return ((uint32_t)RD_OB_SCRATCH_PAD_0(instance));
5396    return ((int)RD_OB_SCRATCH_PAD_0(instance));
6737 }

6739 static void
6740 issue_cmd_ppc(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
6741 {
6742    struct scsi_pkt *pkt;
6743    atomic_add_16(&instance->fw_outstanding, 1);

6745    pkt = cmd->pkt;
6746    if (pkt) {
6747            con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
5407            con_log(CL_ANN1, (CE_CONT, "%llx : issue_cmd_ppc:"
6748                "ISSUED CMD TO FW : called : cmd:"
6749                ": %p instance : %p pkt : %p pkt_time : %x\n",
6750                gethrtime(), (void *)cmd, (void *)instance,
6751                (void *)pkt, cmd->drv_pkt_time));
6752            if (instance->adapterresetinprogress) {
6753                    cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
6754                    con_log(CL_ANN1, (CE_NOTE, "Reset the scsi_pkt timer"));
6755            } else {
6756                    push_pending_mfi_pkt(instance, cmd);
6757            }

6759    } else {
```

```
6760                con_log(CL_DLEVEL1, (CE_NOTE, "%llx : issue_cmd_ppc:"
5420                con_log(CL_ANN1, (CE_CONT, "%llx : issue_cmd_ppc:"
6761                    "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
6762                    "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
6763        }

6765        mutex_enter(&instance->reg_write_mtx);
6766        /* Issue the command to the FW */
6767        WR_IB_QPORT((cmd->frame_phys_addr) |
6768            (((cmd->frame_count - 1) << 1) | 1), instance);
6769        mutex_exit(&instance->reg_write_mtx);

6771 }

6773 /*
6774  * issue_cmd_in_sync_mode
6775  */
6776 static int
6777 issue_cmd_in_sync_mode_ppc(struct mrsas_instance *instance,
6778 struct mrsas_cmd *cmd)
6779 {
6780        int     i;
6781        uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * (10 * MILLISEC);
6782        struct mrsas_header *hdr = &cmd->frame->hdr;

6784        con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: called"));

6786        if (instance->adapterresetinprogress) {
6787                cmd->drv_pkt_time = ddi_get16(
6788                    cmd->frame_dma_obj.acc_handle, &hdr->timeout);
6789                if (cmd->drv_pkt_time < debug_timeout_g)
6790                        cmd->drv_pkt_time = (uint16_t)debug_timeout_g;

6792                con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: "
6793                    "issue and return in reset case\n"));
6794                WR_IB_QPORT((cmd->frame_phys_addr) |
6795                    (((cmd->frame_count - 1) << 1) | 1), instance);

6797                return (DDI_SUCCESS);
6798        } else {
6799                con_log(CL_ANN1, (CE_NOTE, "sync_mode_ppc: pushing the pkt\n"));
6800                push_pending_mfi_pkt(instance, cmd);
6801        }

6803        cmd->cmd_status = ENODATA;

6805        mutex_enter(&instance->reg_write_mtx);
6806        /* Issue the command to the FW */
6807        WR_IB_QPORT((cmd->frame_phys_addr) |
6808            (((cmd->frame_count - 1) << 1) | 1), instance);
6809        mutex_exit(&instance->reg_write_mtx);

6811        mutex_enter(&instance->int_cmd_mtx);

6812        for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
6813                cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
6814        }

6815        mutex_exit(&instance->int_cmd_mtx);

6817        con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_sync_mode_ppc: done"));

6819        if (i < (msecs -1)) {
6820                return (DDI_SUCCESS);
6821        } else {
6822                return (DDI_FAILURE);
```



```
6823        }
6824 }

6826 /*
6827  * issue_cmd_in_poll_mode
6828  */
6829 static int
6830 issue_cmd_in_poll_mode_ppc(struct mrsas_instance *instance,
6831     struct mrsas_cmd *cmd)
6832 {
6833        int             i;
6834        uint16_t        flags;
6835        uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
6836        struct mrsas_header *frame_hdr;

6838        con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_poll_mode_ppc: called"));

6840        frame_hdr = (struct mrsas_header *)cmd->frame;
6841        ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
6842            MFI_CMD_STATUS_POLL_MODE);
6843        flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
6844        flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;

6846        ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);

6848        /* issue the frame using inbound queue port */
6849        WR_IB_QPORT((cmd->frame_phys_addr) |
6850            (((cmd->frame_count - 1) << 1) | 1), instance);

6852        /* wait for cmd_status to change from 0xFF */
6853        for (i = 0; i < msecs && (
6854            ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
6855            == MFI_CMD_STATUS_POLL_MODE); i++) {
6856                drv_usecwait(MILLISEC); /* wait for 1000 usecs */
6857        }

6859        if (ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
6860            == MFI_CMD_STATUS_POLL_MODE) {
6861                con_log(CL_ANN, (CE_NOTE, "issue_cmd_in_poll_mode: "
5514                con_log(CL_ANN1, (CE_NOTE, "issue_cmd_in_poll_mode: "
6862                    "cmd polling timed out"));
6863                return (DDI_FAILURE);
6864        }

6866        return (DDI_SUCCESS);
6867 }
______unchanged_portion_omitted_


6972 static int
6973 mrsas_reset_ppc(struct mrsas_instance *instance)
6974 {
6975        uint32_t status;
6976        uint32_t retry = 0;
6977        uint32_t cur_abs_reg_val;
6978        uint32_t fw_state;

6980        con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));

6982        if (instance->deadadapter == 1) {
6983                cmn_err(CE_WARN, "mrsas_reset_ppc: "
6984                    "no more resets as HBA has been marked dead ");
5634                con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
5635                    "no more resets as HBA has been marked dead "));
6985                return (DDI_FAILURE);
6986        }
```

```
6987         mutex_enter(&instance->ocr_flags_mtx);
6988         instance->adapterresetinprogress = 1;
6989         mutex_exit(&instance->ocr_flags_mtx);
6990         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: adpterresetinprogress "
6991             "flag set, time %llx", gethrtime()));

6993         instance->func_ptr->disable_intr(instance);
6994 retry_reset:
6995         WR_IB_WRITE_SEQ(0, instance);
6996         WR_IB_WRITE_SEQ(4, instance);
6997         WR_IB_WRITE_SEQ(0xb, instance);
6998         WR_IB_WRITE_SEQ(2, instance);
6999         WR_IB_WRITE_SEQ(7, instance);
7000         WR_IB_WRITE_SEQ(0xd, instance);
7001         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: magic number written "
7002             "to write sequence register\n"));
7003         delay(100 * drv_usectohz(MILLISEC));
7004         status = RD_OB_DRWE(instance);

7006         while (!(status & DIAG_WRITE_ENABLE)) {
7007                 delay(100 * drv_usectohz(MILLISEC));
7008                 status = RD_OB_DRWE(instance);
7009                 if (retry++ == 100) {
7010                         cmn_err(CE_WARN, "mrsas_reset_ppc: DRWE bit "
7011                             "check retry count %d", retry);
5660                         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: DRWE bit "
5661                             "check retry count %d\n", retry));
7012                         return (DDI_FAILURE);
7013                 }
7014         }
7015         WR_IB_DRWE(status | DIAG_RESET_ADAPTER, instance);
7016         delay(100 * drv_usectohz(MILLISEC));
7017         status = RD_OB_DRWE(instance);
7018         while (status & DIAG_RESET_ADAPTER) {
7019                 delay(100 * drv_usectohz(MILLISEC));
7020                 status = RD_OB_DRWE(instance);
7021                 if (retry++ == 100) {
7022                         cmn_err(CE_WARN, "mrsas_reset_ppc: "
7023                             "RESET FAILED. KILL adapter called.");

7025                         (void) mrsas_kill_adapter(instance);
7026                         return (DDI_FAILURE);
7027                 }
7028         }
7029         con_log(CL_ANN, (CE_NOTE, "mrsas_reset_ppc: Adapter reset complete"));
5676         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: Adapter reset complete"));
7030         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7031             "Calling mfi_state_transition_to_ready"));

7033         /* Mark HBA as bad, if FW is fault after 3 continuous resets */
7034         if (mfi_state_transition_to_ready(instance) ||
7035             debug_fw_faults_after_ocr_g == 1) {
7036                 cur_abs_reg_val =
7037                     instance->func_ptr->read_fw_status_reg(instance);
7038                 fw_state        = cur_abs_reg_val & MFI_STATE_MASK;

7040 #ifdef OCRDEBUG
7041                 con_log(CL_ANN1, (CE_NOTE,
7042                     "mrsas_reset_ppc :before fake: FW is not ready "
7043                     "FW state = 0x%x", fw_state));
7044                 if (debug_fw_faults_after_ocr_g == 1)
7045                         fw_state = MFI_STATE_FAULT;
7046 #endif

7048                 con_log(CL_ANN1, (CE_NOTE,  "mrsas_reset_ppc : FW is not ready "
7049                     "FW state = 0x%x", fw_state));
```



```
7051                 if (fw_state == MFI_STATE_FAULT) {
7052                         /* increment the count */
7053                         instance->fw_fault_count_after_ocr++;
7054                         if (instance->fw_fault_count_after_ocr
7055                             < MAX_FW_RESET_COUNT) {
7056                                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
7057                                     "FW is in fault after OCR count %d "
7058                                     "Retry Reset",
7059                                     instance->fw_fault_count_after_ocr);
5703                                 con_log(CL_ANN1, (CE_WARN, "mrsas_reset_ppc: "
5704                                     "FW is in fault after OCR count %d ",
5705                                     instance->fw_fault_count_after_ocr));
7060                                 goto retry_reset;

7062                         } else {
7063                                 cmn_err(CE_WARN, "mrsas_reset_ppc: "
7064                                     "Max Reset Count exceeded >%d"
7065                                     "Mark HBA as bad, KILL adapter",
7066                                     MAX_FW_RESET_COUNT);

5709                                 con_log(CL_ANN1, (CE_WARN, "mrsas_reset_ppc: "
5710                                     "Max Reset Count exceeded "
5711                                     "Mark HBA as bad"));
7068                                 (void) mrsas_kill_adapter(instance);
7069                                 return (DDI_FAILURE);
7070                         }
7071                 }
7072         }
7073         /* reset the counter as FW is up after OCR */
7074         instance->fw_fault_count_after_ocr = 0;


7077         ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
7078             instance->producer, 0);

7080         ddi_put32(instance->mfi_internal_dma_obj.acc_handle,
7081             instance->consumer, 0);

7083         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7084             " after resetting produconsumer chck indexs:"
7085             "producer %x consumer %x", *instance->producer,
7086             *instance->consumer));

7088         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7089             "Calling mrsas_issue_init_mfi"));
7090         (void) mrsas_issue_init_mfi(instance);
7091         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7092             "mrsas_issue_init_mfi Done"));

7094         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7095             "Calling mrsas_print_pending_cmd\n"));
7096         (void) mrsas_print_pending_cmds(instance);
7097         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7098             "mrsas_print_pending_cmd done\n"));

7100         instance->func_ptr->enable_intr(instance);
7101         instance->fw_outstanding = 0;

7103         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7104             "Calling mrsas_issue_pending_cmds"));
7105         (void) mrsas_issue_pending_cmds(instance);
7106         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7107             "issue_pending_cmds done.\n"));

5748         "Complete"));
```

```
7109         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7110             "Calling aen registration"));


7113         instance->aen_cmd->retry_count_for_ocr = 0;
7114         instance->aen_cmd->drv_pkt_time = 0;

7116         instance->func_ptr->issue_cmd(instance->aen_cmd, instance);
7117         con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag.\n"));

7119         mutex_enter(&instance->ocr_flags_mtx);
7120         instance->adapterresetinprogress = 0;
7121         mutex_exit(&instance->ocr_flags_mtx);
7122         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc: "
7123             "adpterresetinprogress flag unset"));

7125         con_log(CL_ANN1, (CE_NOTE, "mrsas_reset_ppc done\n"));
7126         return (DDI_SUCCESS);
7127 }

7129 /*
7130  * FMA functions.
7131  */
7132 int
7133 mrsas_common_check(struct mrsas_instance *instance, struct  mrsas_cmd *cmd)
5761 static int
5762 mrsas_common_check(struct mrsas_instance *instance,
5763     struct  mrsas_cmd *cmd)
7134 {
7135         int ret = DDI_SUCCESS;

7137         if (cmd != NULL &&
7138             mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
5767         if (mrsas_check_dma_handle(cmd->frame_dma_obj.dma_handle) !=
7139             DDI_SUCCESS) {
7140                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7141                 if (cmd->pkt != NULL) {
7142                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7143                         cmd->pkt->pkt_statistics = 0;
7144                 }
7145                 ret = DDI_FAILURE;
7146         }
7147         if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
7148             != DDI_SUCCESS) {
7149                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7150                 if (cmd != NULL && cmd->pkt != NULL) {
5779                 if (cmd->pkt != NULL) {
7151                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7152                         cmd->pkt->pkt_statistics = 0;
7153                 }
7154                 ret = DDI_FAILURE;
7155         }
7156         if (mrsas_check_dma_handle(instance->mfi_evt_detail_obj.dma_handle) !=
7157             DDI_SUCCESS) {
7158                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);
7159                 if (cmd != NULL && cmd->pkt != NULL) {
5788                 if (cmd->pkt != NULL) {
7160                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7161                         cmd->pkt->pkt_statistics = 0;
7162                 }
7163                 ret = DDI_FAILURE;
7164         }
7165         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
7166                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_UNAFFECTED);

7168                 ddi_fm_acc_err_clear(instance->regmap_handle, DDI_FME_VER0);
```



```
7170                 if (cmd != NULL && cmd->pkt != NULL) {
5799                 if (cmd->pkt != NULL) {
7171                         cmd->pkt->pkt_reason = CMD_TRAN_ERR;
7172                         cmd->pkt->pkt_statistics = 0;
7173                 }
7174                 ret = DDI_FAILURE;
7175         }

7177         return (ret);
7178 }
_____unchanged_portion_omitted_

7306 static int
7307 mrsas_add_intrs(struct mrsas_instance *instance, int intr_type)
7308 {

7310         dev_info_t *dip = instance->dip;
7311         int     avail, actual, count;
7312         int     i, flag, ret;

7314         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: intr_type = %x",
5943         con_log(CL_DLEVEL1, (CE_WARN, "mrsas_add_intrs: intr_type = %x",
7315             intr_type));

7317         /* Get number of interrupts */
7318         ret = ddi_intr_get_nintrs(dip, intr_type, &count);
7319         if ((ret != DDI_SUCCESS) || (count == 0)) {
7320                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_nintrs() failed:"
7321                     "ret %d count %d", ret, count));

7323                 return (DDI_FAILURE);
7324         }

7326         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: count = %d ", count));
5955         con_log(CL_DLEVEL1, (CE_WARN, "mrsas_add_intrs: count = %d ", count));

7328         /* Get number of available interrupts */
7329         ret = ddi_intr_get_navail(dip, intr_type, &avail);
7330         if ((ret != DDI_SUCCESS) || (avail == 0)) {
7331                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_navail() failed:"
7332                     "ret %d avail %d", ret, avail));

7334                 return (DDI_FAILURE);
7335         }
7336         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: avail = %d ", avail));
5965         con_log(CL_DLEVEL1, (CE_WARN, "mrsas_add_intrs: avail = %d ", avail));

7338         /* Only one interrupt routine. So limit the count to 1 */
7339         if (count > 1) {
7340                 count = 1;
7341         }

7343         /*
7344          * Allocate an array of interrupt handlers. Currently we support
7345          * only one interrupt. The framework can be extended later.
7346          */
7347         instance->intr_htable_size = count * sizeof (ddi_intr_handle_t);
7348         instance->intr_htable = kmem_zalloc(instance->intr_htable_size,
7349             KM_SLEEP);
5976         instance->intr_size = count * sizeof (ddi_intr_handle_t);
5977         instance->intr_htable = kmem_zalloc(instance->intr_size, KM_SLEEP);
7350         ASSERT(instance->intr_htable);

7352         flag = ((intr_type == DDI_INTR_TYPE_MSI) ||
7353             (intr_type == DDI_INTR_TYPE_MSIX)) ?
```

```
7354             DDI_INTR_ALLOC_STRICT : DDI_INTR_ALLOC_NORMAL;
5980         flag = ((intr_type == DDI_INTR_TYPE_MSI) || (intr_type ==
5981             DDI_INTR_TYPE_MSIX)) ? DDI_INTR_ALLOC_STRICT:DDI_INTR_ALLOC_NORMAL;

7356         /* Allocate interrupt */
7357         ret = ddi_intr_alloc(dip, instance->intr_htable, intr_type, 0,
7358             count, &actual, flag);

7360         if ((ret != DDI_SUCCESS) || (actual == 0)) {
7361                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7362                     "avail = %d", avail));
7363                 goto mrsas_free_htable;
5990                 kmem_free(instance->intr_htable, instance->intr_size);
5991                 return (DDI_FAILURE);
7364         }

7366         if (actual < count) {
7367                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7368                     "Requested = %d  Received = %d", count, actual));
7369         }
7370         instance->intr_cnt = actual;

7372         /*
7373          * Get the priority of the interrupt allocated.
7374          */
7375         if ((ret = ddi_intr_get_pri(instance->intr_htable[0],
7376             &instance->intr_pri)) != DDI_SUCCESS) {
7377                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7378                     "get priority call failed"));
7379                 goto mrsas_free_handles;

6007                 for (i = 0; i < actual; i++) {
6008                         (void) ddi_intr_free(instance->intr_htable[i]);
7380         }
6010                 kmem_free(instance->intr_htable, instance->intr_size);
6011                 return (DDI_FAILURE);
6012         }

7382         /*
7383          * Test for high level mutex. we don't support them.
7384          */
7385         if (instance->intr_pri >= ddi_intr_get_hilevel_pri()) {
7386                 con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs: "
7387                     "High level interrupts not supported."));
7388                 goto mrsas_free_handles;

6021                 for (i = 0; i < actual; i++) {
6022                         (void) ddi_intr_free(instance->intr_htable[i]);
7389         }
6024                 kmem_free(instance->intr_htable, instance->intr_size);
6025                 return (DDI_FAILURE);
6026         }

7391         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_add_intrs: intr_pri = 0x%x ",
7392             instance->intr_pri));

7394         /* Call ddi_intr_add_handler() */
7395         for (i = 0; i < actual; i++) {
7396                 ret = ddi_intr_add_handler(instance->intr_htable[i],
7397                     (ddi_intr_handler_t *)mrsas_isr, (caddr_t)instance,
7398                     (caddr_t)(uintptr_t)i);

7400                 if (ret != DDI_SUCCESS) {
7401                         con_log(CL_ANN, (CE_WARN, "mrsas_add_intrs:"
7402                             "failed %d", ret));
7403                         goto mrsas_free_handles;
```



```
6041                         for (i = 0; i < actual; i++) {
6042                                 (void) ddi_intr_free(instance->intr_htable[i]);
7404                 }
6044                         kmem_free(instance->intr_htable, instance->intr_size);
6045                         return (DDI_FAILURE);
6046                 }

7406         }

7408         con_log(CL_DLEVEL1, (CE_NOTE, " ddi_intr_add_handler done"));
6050         con_log(CL_DLEVEL1, (CE_WARN, " ddi_intr_add_handler done"));

7410         if ((ret = ddi_intr_get_cap(instance->intr_htable[0],
7411             &instance->intr_cap)) != DDI_SUCCESS) {
7412                 con_log(CL_ANN, (CE_WARN, "ddi_intr_get_cap() failed %d",
7413                     ret));
7414                 goto mrsas_free_handlers;

6057                 /* Free already allocated intr */
6058                 for (i = 0; i < actual; i++) {
6059                         (void) ddi_intr_remove_handler(
6060                             instance->intr_htable[i]);
6061                         (void) ddi_intr_free(instance->intr_htable[i]);
7415         }
6063                 kmem_free(instance->intr_htable, instance->intr_size);
6064                 return (DDI_FAILURE);
6065         }

7417         if (instance->intr_cap &  DDI_INTR_FLAG_BLOCK) {
7418                 con_log(CL_ANN, (CE_WARN, "Calling ddi_intr_block _enable"));

7420                 (void) ddi_intr_block_enable(instance->intr_htable,
7421                     instance->intr_cnt);
7422         } else {
7423                 con_log(CL_ANN, (CE_NOTE, " calling ddi_intr_enable"));

7425                 for (i = 0; i < instance->intr_cnt; i++) {
7426                         (void) ddi_intr_enable(instance->intr_htable[i]);
7427                         con_log(CL_ANN, (CE_NOTE, "ddi intr enable returns "
7428                             "%d", i));
7429                 }
7430         }

7432         return (DDI_SUCCESS);

7434 mrsas_free_handlers:
7435         for (i = 0; i < actual; i++)
7436                 (void) ddi_intr_remove_handler(instance->intr_htable[i]);

7438 mrsas_free_handles:
7439         for (i = 0; i < actual; i++)
7440                 (void) ddi_intr_free(instance->intr_htable[i]);

7442 mrsas_free_htable:
7443         if (instance->intr_htable != NULL)
7444                 kmem_free(instance->intr_htable, instance->intr_htable_size);

7446         instance->intr_htable = NULL;
7447         instance->intr_htable_size = 0;

7449         return (DDI_FAILURE);

7451 }
```

```
7454 static void
7455 mrsas_rem_intrs(struct mrsas_instance *instance)
7456 {
7457         int i;

7459         con_log(CL_ANN, (CE_NOTE, "mrsas_rem_intrs called"));

7461         /* Disable all interrupts first */
7462         if (instance->intr_cap & DDI_INTR_FLAG_BLOCK) {
7463                 (void) ddi_intr_block_disable(instance->intr_htable,
7464                     instance->intr_cnt);
7465         } else {
7466                 for (i = 0; i < instance->intr_cnt; i++) {
7467                         (void) ddi_intr_disable(instance->intr_htable[i]);
7468                 }
7469         }

7471         /* Remove all the handlers */

7473         for (i = 0; i < instance->intr_cnt; i++) {
7474                 (void) ddi_intr_remove_handler(instance->intr_htable[i]);
7475                 (void) ddi_intr_free(instance->intr_htable[i]);
7476         }

7478         if (instance->intr_htable != NULL)
7479                 kmem_free(instance->intr_htable, instance->intr_htable_size);

7481         instance->intr_htable = NULL;
7482         instance->intr_htable_size = 0;

6111         kmem_free(instance->intr_htable, instance->intr_size);
7484 }

7486 static int
7487 mrsas_tran_bus_config(dev_info_t *parent, uint_t flags,
7488     ddi_bus_config_op_t op, void *arg, dev_info_t **childp)
7489 {
7490         struct mrsas_instance *instance;
7491         int config;
7492         int rval  = NDI_SUCCESS;
6120         int rval;

7494         char *ptr = NULL;
7495         int tgt, lun;

7497         con_log(CL_ANN1, (CE_NOTE, "Bus config called for op = %x", op));

7499         if ((instance = ddi_get_soft_state(mrsas_state,
7500             ddi_get_instance(parent))) == NULL) {
7501                 return (NDI_FAILURE);
7502         }

7504         /* Hold nexus during bus_config */
7505         ndi_devi_enter(parent, &config);
7506         switch (op) {
7507         case BUS_CONFIG_ONE: {

7509                 /* parse wwid/target name out of name given */
7510                 if ((ptr = strchr((char *)arg, '@')) == NULL) {
7511                         rval = NDI_FAILURE;
7512                         break;
7513                 }
7514                 ptr++;

7516                 if (mrsas_parse_devname(arg, &tgt, &lun) != 0) {
7517                         rval = NDI_FAILURE;
```



```
7518                         break;
7519                 }

7521                 if (lun == 0) {
7522                         rval = mrsas_config_ld(instance, tgt, lun, childp);
7523 #ifdef PDSUPPORT
7524                 } else if (instance->tbolt == 1 && lun != 0) {
7525                         rval = mrsas_tbolt_config_pd(instance,
7526                             tgt, lun, childp);
7527 #endif
7528                 } else {
7529                         rval = NDI_FAILURE;
7530                 }

7532                 break;
7533         }
7534         case BUS_CONFIG_DRIVER:
7535         case BUS_CONFIG_ALL: {

7537                 rval = mrsas_config_all_devices(instance);

7539                 rval = NDI_SUCCESS;
7540                 break;
7541         }
7542         }

7544         if (rval == NDI_SUCCESS) {
7545                 rval = ndi_busop_bus_config(parent, flags, op, arg, childp, 0);

7547         }
7548         ndi_devi_exit(parent, config);

7550         con_log(CL_ANN1, (CE_NOTE, "mrsas_tran_bus_config: rval = %x",
7551             rval));
7552         return (rval);
7553 }

7555 static int
7556 mrsas_config_all_devices(struct mrsas_instance *instance)
7557 {
7558         int rval, tgt;

7560         for (tgt = 0; tgt < MRDRV_MAX_LD; tgt++) {
7561                 (void) mrsas_config_ld(instance, tgt, 0, NULL);

7563         }

7565 #ifdef PDSUPPORT
7566         /* Config PD devices connected to the card */
7567         if (instance->tbolt) {
7568                 for (tgt = 0; tgt < instance->mr_tbolt_pd_max; tgt++) {
7569                         (void) mrsas_tbolt_config_pd(instance, tgt, 1, NULL);
7570                 }
7571         }
7572 #endif

7574         rval = NDI_SUCCESS;
7575         return (rval);
7576 }
______unchanged_portion_omitted_

7622 static int
7623 mrsas_config_ld(struct mrsas_instance *instance, uint16_t tgt,
7624     uint8_t lun, dev_info_t **ldip)
7625 {
7626         struct scsi_device *sd;
```

```
7627         dev_info_t *child;
7628         int rval;

7630         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_config_ld: t = %d l = %d",
6244         con_log(CL_ANN1, (CE_NOTE, "mrsas_config_ld: t = %d l = %d",
7631             tgt, lun));

7633         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
7634                 if (ldip) {
7635                         *ldip = child;
7636                 }
7637                 if (instance->mr_ld_list[tgt].flag != MRDRV_TGT_VALID) {
7638                         rval = mrsas_service_evt(instance, tgt, 0,
7639                             MRSAS_EVT_UNCONFIG_TGT, NULL);
7640                         con_log(CL_ANN1, (CE_WARN,
7641                             "mr_sas: DELETING STALE ENTRY rval = %d "
7642                             "tgt id = %d ", rval, tgt));
7643                         return (NDI_FAILURE);
7644                 }
7651                 con_log(CL_ANN1, (CE_NOTE,
6252                     "mrsas_config_ld: Child = %p found t = %d l = %d",
6253                     (void *)child, tgt, lun));
7645                 return (NDI_SUCCESS);
7646         }

7648         sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
7649         sd->sd_address.a_hba_tran = instance->tran;
7650         sd->sd_address.a_target = (uint16_t)tgt;
7651         sd->sd_address.a_lun = (uint8_t)lun;

7653         if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS)
7654                 rval = mrsas_config_scsi_device(instance, sd, ldip);
7655         else
7656                 rval = NDI_FAILURE;

7658         /* sd_unprobe is blank now. Free buffer manually */
7659         if (sd->sd_inq) {
7660                 kmem_free(sd->sd_inq, SUN_INQSIZE);
7661                 sd->sd_inq = (struct scsi_inquiry *)NULL;
7662         }

7664         kmem_free(sd, sizeof (struct scsi_device));
7665         con_log(CL_DLEVEL1, (CE_NOTE, "mrsas_config_ld: return rval = %d",
6274         con_log(CL_ANN1, (CE_NOTE, "mrsas_config_ld: return rval = %d",
7666             rval));
7667         return (rval);
7668 }

7670 int
6279 static int
7671 mrsas_config_scsi_device(struct mrsas_instance *instance,
7672     struct scsi_device *sd, dev_info_t **dipp)
7673 {
7674         char *nodename = NULL;
7675         char **compatible = NULL;
7676         int ncompatible = 0;
7677         char *childname;
7678         dev_info_t *ldip = NULL;
7679         int tgt = sd->sd_address.a_target;
7680         int lun = sd->sd_address.a_lun;
7681         int dtype = sd->sd_inq->inq_dtype & DTYPE_MASK;
7682         int rval;

7684         con_log(CL_DLEVEL1, (CE_NOTE, "mr_sas: scsi_device t%dL%d", tgt, lun));
6293         con_log(CL_ANN1, (CE_WARN, "mr_sas: scsi_device t%dL%d", tgt, lun));
7685         scsi_hba_nodename_compatible_get(sd->sd_inq, NULL, dtype,
```



```
7686             NULL, &nodename, &compatible, &ncompatible);

7688         if (nodename == NULL) {
7689                 con_log(CL_ANN1, (CE_WARN, "mr_sas: Found no compatible driver "
7690                     "for t%dL%d", tgt, lun));
7691                 rval = NDI_FAILURE;
7692                 goto finish;
7693         }

7695         childname = (dtype == DTYPE_DIRECT) ? "sd" : nodename;
7696         con_log(CL_DLEVEL1, (CE_NOTE,
6305         con_log(CL_ANN1, (CE_WARN,
7697             "mr_sas: Childname = %2s nodename = %s", childname, nodename));

7699         /* Create a dev node */
7700         rval = ndi_devi_alloc(instance->dip, childname, DEVI_SID_NODEID, &ldip);
7701         con_log(CL_DLEVEL1, (CE_NOTE,
6310         con_log(CL_ANN1, (CE_WARN,
7702             "mr_sas_config_scsi_device: ndi_devi_alloc rval = %x", rval));
7703         if (rval == NDI_SUCCESS) {
7704                 if (ndi_prop_update_int(DDI_DEV_T_NONE, ldip, "target", tgt) !=
7705                     DDI_PROP_SUCCESS) {
7706                         con_log(CL_ANN1, (CE_WARN, "mr_sas: unable to create "
7707                             "property for t%dl%d target", tgt, lun));
7708                         rval = NDI_FAILURE;
7709                         goto finish;
7710                 }
7711                 if (ndi_prop_update_int(DDI_DEV_T_NONE, ldip, "lun", lun) !=
7712                     DDI_PROP_SUCCESS) {
7713                         con_log(CL_ANN1, (CE_WARN, "mr_sas: unable to create "
7714                             "property for t%dl%d lun", tgt, lun));
7715                         rval = NDI_FAILURE;
7716                         goto finish;
7717                 }

7719                 if (ndi_prop_update_string_array(DDI_DEV_T_NONE, ldip,
7720                     "compatible", compatible, ncompatible) !=
7721                     DDI_PROP_SUCCESS) {
7722                         con_log(CL_ANN1, (CE_WARN, "mr_sas: unable to create "
7723                             "property for t%dl%d compatible", tgt, lun));
7724                         rval = NDI_FAILURE;
7725                         goto finish;
7726                 }

7728                 rval = ndi_devi_online(ldip, NDI_ONLINE_ATTACH);
7729                 if (rval != NDI_SUCCESS) {
7730                         con_log(CL_ANN1, (CE_WARN, "mr_sas: unable to online "
7731                             "t%dl%d", tgt, lun));
7732                         ndi_prop_remove_all(ldip);
7733                         (void) ndi_devi_free(ldip);
7734                 } else {
7735                         con_log(CL_ANN1, (CE_CONT, "mr_sas: online Done :"
6344                         con_log(CL_ANN1, (CE_WARN, "mr_sas: online Done :"
7736                             "0 t%dl%d", tgt, lun));
7737                 }

7739         }
7740 finish:
7741         if (dipp) {
7742                 *dipp = ldip;
7743         }

7745         con_log(CL_DLEVEL1, (CE_NOTE,
6354         con_log(CL_DLEVEL1, (CE_WARN,
7746             "mr_sas: config_scsi_device rval = %d t%dL%d",
7747             rval, tgt, lun));
```

```
7748         scsi_hba_nodename_compatible_free(nodename, compatible);
7749         return (rval);
7750 }

7752 /*ARGSUSED*/
7753 int
6362 static int
7754 mrsas_service_evt(struct mrsas_instance *instance, int tgt, int lun, int event,
7755     uint64_t wwn)
7756 {
7757         struct mrsas_eventinfo *mrevt = NULL;

7759         con_log(CL_ANN1, (CE_NOTE,
7760             "mrsas_service_evt called for t%dl%d event = %d",
7761             tgt, lun, event));

7763         if ((instance->taskq == NULL) || (mrevt =
7764             kmem_zalloc(sizeof (struct mrsas_eventinfo), KM_NOSLEEP)) == NULL) {
7765                 return (ENOMEM);
7766         }

7768         mrevt->instance = instance;
7769         mrevt->tgt = tgt;
7770         mrevt->lun = lun;
7771         mrevt->event = event;
7772         mrevt->wwn = wwn;

7774         if ((ddi_taskq_dispatch(instance->taskq,
7775             (void (*)(void *))mrsas_issue_evt_taskq, mrevt, DDI_NOSLEEP)) !=
7776             DDI_SUCCESS) {
7777                 con_log(CL_ANN1, (CE_NOTE,
7778                     "mr_sas: Event task failed for t%dl%d event = %d",
7779                     tgt, lun, event));
7780                 kmem_free(mrevt, sizeof (struct mrsas_eventinfo));
7781                 return (DDI_FAILURE);
7782         }
7783         DTRACE_PROBE3(service_evt, int, tgt, int, lun, int, event);
7784         return (DDI_SUCCESS);
7785 }

7787 static void
7788 mrsas_issue_evt_taskq(struct mrsas_eventinfo *mrevt)
7789 {
7790         struct mrsas_instance *instance = mrevt->instance;
7791         dev_info_t *dip, *pdip;
7792         int circ1 = 0;
7793         char *devname;

7795         con_log(CL_ANN1, (CE_NOTE, "mrsas_issue_evt_taskq: called for"
7796             " tgt %d lun %d event %d",
7797             mrevt->tgt, mrevt->lun, mrevt->event));

7799         if (mrevt->tgt < MRDRV_MAX_LD && mrevt->lun == 0) {
7800                 mutex_enter(&instance->config_dev_mtx);
7801                 dip = instance->mr_ld_list[mrevt->tgt].dip;
7802                 mutex_exit(&instance->config_dev_mtx);
7803 #ifdef PDSUPPORT
7804         } else {
7805                 mutex_enter(&instance->config_dev_mtx);
7806                 dip = instance->mr_tbolt_pd_list[mrevt->tgt].dip;
7807                 mutex_exit(&instance->config_dev_mtx);
7808 #endif
6410                 return;
7809         }
```



```
7812         ndi_devi_enter(instance->dip, &circ1);
7813         switch (mrevt->event) {
7814         case MRSAS_EVT_CONFIG_TGT:
7815                 if (dip == NULL) {

7817                         if (mrevt->lun == 0) {
7818                                 (void) mrsas_config_ld(instance, mrevt->tgt,
7819                                     0, NULL);
7820 #ifdef PDSUPPORT
7821                         } else if (instance->tbolt) {
7822                                 (void) mrsas_tbolt_config_pd(instance,
7823                                     mrevt->tgt,
7824                                     1, NULL);
7825 #endif
7826                         }
7827                         con_log(CL_ANN1, (CE_NOTE,
7828                             "mr_sas: EVT_CONFIG_TGT called:"
7829                             " for tgt %d lun %d event %d",
7830                             mrevt->tgt, mrevt->lun, mrevt->event));

7832                 } else {
7833                         con_log(CL_ANN1, (CE_NOTE,
7834                             "mr_sas: EVT_CONFIG_TGT dip != NULL:"
7835                             " for tgt %d lun %d event %d",
7836                             mrevt->tgt, mrevt->lun, mrevt->event));
7837                 }
7838                 break;
7839         case MRSAS_EVT_UNCONFIG_TGT:
7840                 if (dip) {
7841                         if (i_ddi_devi_attached(dip)) {

7843                                 pdip = ddi_get_parent(dip);

7845                                 devname = kmem_zalloc(MAXNAMELEN + 1, KM_SLEEP);
7846                                 (void) ddi_deviname(dip, devname);

7848                                 (void) devfs_clean(pdip, devname + 1,
7849                                     DV_CLEAN_FORCE);
7850                                 kmem_free(devname, MAXNAMELEN + 1);
7851                         }
7852                         (void) ndi_devi_offline(dip, NDI_DEVI_REMOVE);
7853                         con_log(CL_ANN1, (CE_NOTE,
7854                             "mr_sas: EVT_UNCONFIG_TGT called:"
7855                             " for tgt %d lun %d event %d",
7856                             mrevt->tgt, mrevt->lun, mrevt->event));
7857                 } else {
7858                         con_log(CL_ANN1, (CE_NOTE,
7859                             "mr_sas: EVT_UNCONFIG_TGT dip == NULL:"
7860                             " for tgt %d lun %d event %d",
7861                             mrevt->tgt, mrevt->lun, mrevt->event));
7862                 }
7863                 break;
7864         }
7865         kmem_free(mrevt, sizeof (struct mrsas_eventinfo));
7866         ndi_devi_exit(instance->dip, circ1);
7867 }


7870 int
6464 static int
7871 mrsas_mode_sense_build(struct scsi_pkt *pkt)
7872 {
7873         union scsi_cdb          *cdbp;
7874         uint16_t                page_code;
7875         struct scsa_cmd         *acmd;
7876         struct buf              *bp;
```

```
7877         struct mode_header      *modehdrp;

7879         cdbp = (void *)pkt->pkt_cdbp;
7880         page_code = cdbp->cdb_un.sg.scsi[0];
7881         acmd = PKT2CMD(pkt);
7882         bp = acmd->cmd_buf;
7883         if ((!bp) && bp->b_un.b_addr && bp->b_bcount && acmd->cmd_dmacount) {
7884                 con_log(CL_ANN1, (CE_WARN, "Failing MODESENSE Command"));
7885                 /* ADD pkt statistics as Command failed. */
7886                 return (NULL);
7887         }

7889         bp_mapin(bp);
7890         bzero(bp->b_un.b_addr, bp->b_bcount);

7892         switch (page_code) {
7893                 case 0x3: {
7894                         struct mode_format *page3p = NULL;
7895                         modehdrp = (struct mode_header *)(bp->b_un.b_addr);
7896                         modehdrp->bdesc_length = MODE_BLK_DESC_LENGTH;

7898                         page3p = (void *)((caddr_t)modehdrp +
7899                             MODE_HEADER_LENGTH + MODE_BLK_DESC_LENGTH);
7900                         page3p->mode_page.code = 0x3;
7901                         page3p->mode_page.length =
7902                             (uchar_t)(sizeof (struct mode_format));
7903                         page3p->data_bytes_sect = 512;
7904                         page3p->sect_track = 63;
7905                         break;
7906                 }
7907                 case 0x4: {
7908                         struct mode_geometry *page4p = NULL;
7909                         modehdrp = (struct mode_header *)(bp->b_un.b_addr);
7910                         modehdrp->bdesc_length = MODE_BLK_DESC_LENGTH;

7912                         page4p = (void *)((caddr_t)modehdrp +
7913                             MODE_HEADER_LENGTH + MODE_BLK_DESC_LENGTH);
7914                         page4p->mode_page.code = 0x4;
7915                         page4p->mode_page.length =
7916                             (uchar_t)(sizeof (struct mode_geometry));
7917                         page4p->heads = 255;
7918                         page4p->rpm = 10000;
7919                         break;
7920                 }
7921                 default:
7922                         break;
7923         }
7924         return (NULL);
7925 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
     289 Tue Nov  6 14:28:55 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.conf
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
#
# Copyright (c) 2008-2012, LSI Logic Corporation.
# Copyright (c) 2008-2009, LSI Logic Corporation.
# All rights reserved.
#
# Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
# Use is subject to license terms.
#

#
# mr_sas.conf for sol 10 (and later) for all supported architectures
#

# MSI specific flag. Default is "yes".
# mrsas-enable-msi="yes";

# Fast-Path specific flag. Default is "yes".
# mrsas-enable-fp="yes";
# global definitions
```

```
**********************************************************
   55863 Tue Nov  6 14:28:56 2012
new/usr/src/uts/common/io/mr_sas/mr_sas.h
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * mr_sas.h: header for mr_sas
 *
 * Solaris MegaRAID driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * Copyright (c) 2008-2009, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. 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.
 */

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

#ifndef _MR_SAS_H_
#define _MR_SAS_H_

#ifdef  __cplusplus
extern "C" {
#endif

#include <sys/scsi/scsi.h>
#include "mr_sas_list.h"
#include "ld_pd_map.h"

/*
 * MegaRAID SAS2.0 Driver meta data
```



```
 */
#define MRSAS_VERSION                   "6.503.00.00ILLUMOS"
#define MRSAS_RELDATE                   "July 30, 2012"
#define MRSAS_VERSION                   "LSIv2.7"
#define MRSAS_RELDATE                   "Apr 21, 2010"

#define MRSAS_TRUE                      1
#define MRSAS_FALSE                     0

#define ADAPTER_RESET_NOT_REQUIRED      0
#define ADAPTER_RESET_REQUIRED          1

#define PDSUPPORT       1

/*
 * MegaRAID SAS2.0 device id conversion definitions.
 */
#define INST2LSIRDCTL(x)                ((x) << INST_MINOR_SHIFT)
#define MRSAS_GET_BOUNDARY_ALIGNED_LEN(len, new_len, boundary_len)  { \
        int rem; \
        rem = (len / boundary_len); \
        if ((rem * boundary_len) != len) { \
                new_len = len + ((rem + 1) * boundary_len - len); \
        } else { \
                new_len = len; \
        } \
}


/*
 * MegaRAID SAS2.0 supported controllers
 */
#define PCI_DEVICE_ID_LSI_2108VDE       0x0078
#define PCI_DEVICE_ID_LSI_2108V         0x0079
#define PCI_DEVICE_ID_LSI_TBOLT         0x005b
#define PCI_DEVICE_ID_LSI_INVADER       0x005d

/*
 * Register Index for 2108 Controllers.
 */
#define REGISTER_SET_IO_2108            (2)

#define MRSAS_MAX_SGE_CNT               0x50
#define MRSAS_APP_RESERVED_CMDS         32

#define MRSAS_IOCTL_DRIVER              0x12341234
#define MRSAS_IOCTL_FIRMWARE            0x12345678
#define MRSAS_IOCTL_AEN                 0x87654321

#define MRSAS_1_SECOND                  1000000

#ifdef PDSUPPORT

#define UNCONFIGURED_GOOD               0x0
#define PD_SYSTEM                       0x40
#define MR_EVT_PD_STATE_CHANGE          0x0072
#define MR_EVT_PD_REMOVED_EXT           0x00f8
#define MR_EVT_PD_INSERTED_EXT          0x00f7
#define MR_DCMD_PD_GET_INFO             0x02020000
#define MRSAS_TBOLT_PD_LUN              1
#define MRSAS_TBOLT_PD_TGT_MAX  255
#define MRSAS_TBOLT_GET_PD_MAX(s)       ((s)->mr_tbolt_pd_max)

#endif

/* Raid Context Flags */
```

```
#define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT 0x4
#define MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_MASK 0x30
typedef enum MR_RAID_FLAGS_IO_SUB_TYPE {
        MR_RAID_FLAGS_IO_SUB_TYPE_NONE = 0,
        MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD = 1
} MR_RAID_FLAGS_IO_SUB_TYPE;

/* Dynamic Enumeration Flags */
#define MRSAS_PD_LUN            1
#define MRSAS_LD_LUN            0
#define MRSAS_PD_TGT_MAX        255
#define MRSAS_GET_PD_MAX(s)     ((s)->mr_pd_max)
#define WWN_STRLEN              17
#define LD_SYNC_BIT     1
#define LD_SYNC_SHIFT   14
/* ThunderBolt (TB) specific */
#define MRSAS_THUNDERBOLT_MSG_SIZE              256
#define MRSAS_THUNDERBOLT_MAX_COMMANDS          1024
#define MRSAS_THUNDERBOLT_MAX_REPLY_COUNT       1024
#define MRSAS_THUNDERBOLT_REPLY_SIZE            8
#define MRSAS_THUNDERBOLT_MAX_CHAIN_COUNT       1

#define MPI2_FUNCTION_PASSTHRU_IO_REQUEST       0xF0
#define MPI2_FUNCTION_LD_IO_REQUEST             0xF1

#define MR_EVT_LD_FAST_PATH_IO_STATUS_CHANGED   (0xFFFF)

#define MR_INTERNAL_MFI_FRAMES_SMID             1
#define MR_CTRL_EVENT_WAIT_SMID                 2
#define MR_INTERNAL_DRIVER_RESET_SMID           3


#define         APP_RESERVE_CMDS                32
/*
 * ======================================
 * MegaRAID SAS2.0 MFI firmware definitions
 * ======================================
 */
/*
 * MFI stands for  MegaRAID SAS2.0 FW Interface. This is just a moniker for
 * protocol between the software and firmware. Commands are issued using
 * "message frames"
 */

/*
 * FW posts its state in upper 4 bits of outbound_msg_0 register
 */
#define MFI_STATE_MASK                          0xF0000000
#define MFI_STATE_UNDEFINED                     0x00000000
#define MFI_STATE_BB_INIT                       0x10000000
#define MFI_STATE_FW_INIT                       0x40000000
#define MFI_STATE_WAIT_HANDSHAKE                0x60000000
#define MFI_STATE_FW_INIT_2                     0x70000000
#define MFI_STATE_DEVICE_SCAN                   0x80000000
#define MFI_STATE_BOOT_MESSAGE_PENDING          0x90000000
#define MFI_STATE_FLUSH_CACHE                   0xA0000000
#define MFI_STATE_READY                         0xB0000000
#define MFI_STATE_OPERATIONAL                   0xC0000000
#define MFI_STATE_FAULT                         0xF0000000
#define MFI_STATE_SHIFT                         28
#define MFI_STATE_MASK                          ((uint32_t)0xF<<MFI_STATE_SHIFT)
#define MFI_STATE_UNDEFINED                     ((uint32_t)0x0<<MFI_STATE_SHIFT)
#define MFI_STATE_BB_INIT                       ((uint32_t)0x1<<MFI_STATE_SHIFT)
#define MFI_STATE_FW_INIT                       ((uint32_t)0x4<<MFI_STATE_SHIFT)
#define MFI_STATE_WAIT_HANDSHAKE                ((uint32_t)0x6<<MFI_STATE_SHIFT)
#define MFI_STATE_FW_INIT_2                     ((uint32_t)0x7<<MFI_STATE_SHIFT)
```



```
#define MFI_STATE_DEVICE_SCAN                   ((uint32_t)0x8<<MFI_STATE_SHIFT)
#define MFI_STATE_BOOT_MESSAGE_PENDING          ((uint32_t)0x9<<MFI_STATE_SHIFT)
#define MFI_STATE_FLUSH_CACHE                   ((uint32_t)0xA<<MFI_STATE_SHIFT)
#define MFI_STATE_READY                         ((uint32_t)0xB<<MFI_STATE_SHIFT)
#define MFI_STATE_OPERATIONAL                   ((uint32_t)0xC<<MFI_STATE_SHIFT)
#define MFI_STATE_FAULT                         ((uint32_t)0xF<<MFI_STATE_SHIFT)

#define MRMFI_FRAME_SIZE                        64

/*
 * During FW init, clear pending cmds & reset state using inbound_msg_0
 *
 * ABORT        : Abort all pending cmds
 * READY        : Move from OPERATIONAL to READY state; discard queue info
 * MFIMODE      : Discard (possible) low MFA posted in 64-bit mode (??)
 * CLR_HANDSHAKE: FW is waiting for HANDSHAKE from BIOS or Driver
 */
#define MFI_INIT_ABORT                          0x00000001
#define MFI_INIT_READY                          0x00000002
#define MFI_INIT_MFIMODE                        0x00000004
#define MFI_INIT_CLEAR_HANDSHAKE                0x00000008
#define MFI_INIT_HOTPLUG                        0x00000010
#define MFI_STOP_ADP                            0x00000020
#define MFI_RESET_FLAGS         MFI_INIT_READY|MFI_INIT_MFIMODE|MFI_INIT_ABORT

/*
 * MFI frame flags
 */
#define MFI_FRAME_POST_IN_REPLY_QUEUE           0x0000
#define MFI_FRAME_DONT_POST_IN_REPLY_QUEUE      0x0001
#define MFI_FRAME_SGL32                         0x0000
#define MFI_FRAME_SGL64                         0x0002
#define MFI_FRAME_SENSE32                       0x0000
#define MFI_FRAME_SENSE64                       0x0004
#define MFI_FRAME_DIR_NONE                      0x0000
#define MFI_FRAME_DIR_WRITE                     0x0008
#define MFI_FRAME_DIR_READ                      0x0010
#define MFI_FRAME_DIR_BOTH                      0x0018
#define MFI_FRAME_IEEE                          0x0020

/*
 * Definition for cmd_status
 */
#define MFI_CMD_STATUS_POLL_MODE                0xFF
#define MFI_CMD_STATUS_SYNC_MODE                0xFF

/*
 * MFI command opcodes
 */
#define MFI_CMD_OP_INIT                         0x00
#define MFI_CMD_OP_LD_READ                      0x01
#define MFI_CMD_OP_LD_WRITE                     0x02
#define MFI_CMD_OP_LD_SCSI                      0x03
#define MFI_CMD_OP_PD_SCSI                      0x04
#define MFI_CMD_OP_DCMD                         0x05
#define MFI_CMD_OP_ABORT                        0x06
#define MFI_CMD_OP_SMP                          0x07
#define MFI_CMD_OP_STP                          0x08

#define MR_DCMD_CTRL_GET_INFO                   0x01010000

#define MR_DCMD_CTRL_CACHE_FLUSH                0x01101000
#define MR_FLUSH_CTRL_CACHE                     0x01
#define MR_FLUSH_DISK_CACHE                     0x02

#define MR_DCMD_CTRL_SHUTDOWN                   0x01050000
```

```
240 #define MRSAS_ENABLE_DRIVE_SPINDOWN             0x01

242 #define MR_DCMD_CTRL_EVENT_GET_INFO             0x01040100
243 #define MR_DCMD_CTRL_EVENT_GET                  0x01040300
244 #define MR_DCMD_CTRL_EVENT_WAIT                 0x01040500
245 #define MR_DCMD_LD_GET_PROPERTIES               0x03030000
185 #define MR_DCMD_PD_GET_INFO                     0x02020000

247 /*
248  * Solaris Specific MAX values
249  */
250 #define MAX_SGL                                 24

252 /*
253  * MFI command completion codes
254  */
255 enum MFI_STAT {
256         MFI_STAT_OK                             = 0x00,
257         MFI_STAT_INVALID_CMD                    = 0x01,
258         MFI_STAT_INVALID_DCMD                   = 0x02,
259         MFI_STAT_INVALID_PARAMETER              = 0x03,
260         MFI_STAT_INVALID_SEQUENCE_NUMBER        = 0x04,
261         MFI_STAT_ABORT_NOT_POSSIBLE             = 0x05,
262         MFI_STAT_APP_HOST_CODE_NOT_FOUND        = 0x06,
263         MFI_STAT_APP_IN_USE                     = 0x07,
264         MFI_STAT_APP_NOT_INITIALIZED            = 0x08,
265         MFI_STAT_ARRAY_INDEX_INVALID            = 0x09,
266         MFI_STAT_ARRAY_ROW_NOT_EMPTY            = 0x0a,
267         MFI_STAT_CONFIG_RESOURCE_CONFLICT       = 0x0b,
268         MFI_STAT_DEVICE_NOT_FOUND               = 0x0c,
269         MFI_STAT_DRIVE_TOO_SMALL                = 0x0d,
270         MFI_STAT_FLASH_ALLOC_FAIL               = 0x0e,
271         MFI_STAT_FLASH_BUSY                     = 0x0f,
272         MFI_STAT_FLASH_ERROR                    = 0x10,
273         MFI_STAT_FLASH_IMAGE_BAD                = 0x11,
274         MFI_STAT_FLASH_IMAGE_INCOMPLETE         = 0x12,
275         MFI_STAT_FLASH_NOT_OPEN                 = 0x13,
276         MFI_STAT_FLASH_NOT_STARTED              = 0x14,
277         MFI_STAT_FLUSH_FAILED                   = 0x15,
278         MFI_STAT_HOST_CODE_NOT_FOUNT            = 0x16,
279         MFI_STAT_LD_CC_IN_PROGRESS              = 0x17,
280         MFI_STAT_LD_INIT_IN_PROGRESS            = 0x18,
281         MFI_STAT_LD_LBA_OUT_OF_RANGE            = 0x19,
282         MFI_STAT_LD_MAX_CONFIGURED              = 0x1a,
283         MFI_STAT_LD_NOT_OPTIMAL                 = 0x1b,
284         MFI_STAT_LD_RBLD_IN_PROGRESS            = 0x1c,
285         MFI_STAT_LD_RECON_IN_PROGRESS           = 0x1d,
286         MFI_STAT_LD_WRONG_RAID_LEVEL            = 0x1e,
287         MFI_STAT_MAX_SPARES_EXCEEDED            = 0x1f,
288         MFI_STAT_MEMORY_NOT_AVAILABLE           = 0x20,
289         MFI_STAT_MFC_HW_ERROR                   = 0x21,
290         MFI_STAT_NO_HW_PRESENT                  = 0x22,
291         MFI_STAT_NOT_FOUND                      = 0x23,
292         MFI_STAT_NOT_IN_ENCL                    = 0x24,
293         MFI_STAT_PD_CLEAR_IN_PROGRESS           = 0x25,
294         MFI_STAT_PD_TYPE_WRONG                  = 0x26,
295         MFI_STAT_PR_DISABLED                    = 0x27,
296         MFI_STAT_ROW_INDEX_INVALID              = 0x28,
297         MFI_STAT_SAS_CONFIG_INVALID_ACTION      = 0x29,
298         MFI_STAT_SAS_CONFIG_INVALID_DATA        = 0x2a,
299         MFI_STAT_SAS_CONFIG_INVALID_PAGE        = 0x2b,
300         MFI_STAT_SAS_CONFIG_INVALID_TYPE        = 0x2c,
301         MFI_STAT_SCSI_DONE_WITH_ERROR           = 0x2d,
302         MFI_STAT_SCSI_IO_FAILED                 = 0x2e,
303         MFI_STAT_SCSI_RESERVATION_CONFLICT      = 0x2f,
304         MFI_STAT_SHUTDOWN_FAILED                = 0x30,
```



```
305         MFI_STAT_TIME_NOT_SET                   = 0x31,
306         MFI_STAT_WRONG_STATE                    = 0x32,
307         MFI_STAT_LD_OFFLINE                     = 0x33,
247         /* UNUSED: 0x34 to 0xfe */
308         MFI_STAT_INVALID_STATUS                 = 0xFF
309 };
_____unchanged_portion_omitted_

333 enum MR_EVT_ARGS {
334         MR_EVT_ARGS_NONE,
335         MR_EVT_ARGS_CDB_SENSE,
336         MR_EVT_ARGS_LD,
337         MR_EVT_ARGS_LD_COUNT,
338         MR_EVT_ARGS_LD_LBA,
339         MR_EVT_ARGS_LD_OWNER,
340         MR_EVT_ARGS_LD_LBA_PD_LBA,
341         MR_EVT_ARGS_LD_PROG,
342         MR_EVT_ARGS_LD_STATE,
343         MR_EVT_ARGS_LD_STRIP,
344         MR_EVT_ARGS_PD,
345         MR_EVT_ARGS_PD_ERR,
346         MR_EVT_ARGS_PD_LBA,
347         MR_EVT_ARGS_PD_LBA_LD,
348         MR_EVT_ARGS_PD_PROG,
349         MR_EVT_ARGS_PD_STATE,
350         MR_EVT_ARGS_PCI,
351         MR_EVT_ARGS_RATE,
352         MR_EVT_ARGS_STR,
353         MR_EVT_ARGS_TIME,
354         MR_EVT_ARGS_ECC
355 };

357 #define MR_EVT_CFG_CLEARED              0x0004
358 #define MR_EVT_LD_CREATED               0x008a
359 #define MR_EVT_LD_DELETED               0x008b
360 #define MR_EVT_CFG_FP_CHANGE            0x017B
276 #define MR_EVT_PD_REMOVED_EXT           0x00f8
277 #define MR_EVT_PD_INSERTED_EXT          0x00f7

362 enum LD_STATE {
363         LD_OFFLINE              = 0,
364         LD_PARTIALLY_DEGRADED   = 1,
365         LD_DEGRADED             = 2,
366         LD_OPTIMAL              = 3,
367         LD_INVALID              = 0xFF
368 };
_____unchanged_portion_omitted_

376 #define DMA_OBJ_ALLOCATED       1
377 #define DMA_OBJ_REALLOCATED     2
378 #define DMA_OBJ_FREED           3

380 /*
381  * dma_obj_t    - Our DMA object
382  * @param buffer        : kernel virtual address
383  * @param size          : size of the data to be allocated
384  * @param acc_handle    : access handle
385  * @param dma_handle    : dma handle
386  * @param dma_cookie    : scatter-gather list
387  * @param dma_attr      : dma attributes for this buffer
388  *
389  * Our DMA object. The caller must initialize the size and dma attributes
390  * (dma_attr) fields before allocating the resources.
391  */
392 typedef struct {
393         caddr_t                 buffer;
```

```
394         uint32_t                size;
395         ddi_acc_handle_t        acc_handle;
396         ddi_dma_handle_t        dma_handle;
397         ddi_dma_cookie_t        dma_cookie[MRSAS_MAX_SGE_CNT];
398         ddi_dma_attr_t          dma_attr;
399         uint8_t                 status;
400         uint8_t                 reserved[3];
401 } dma_obj_t;

403 struct mrsas_eventinfo {
404         struct mrsas_instance   *instance;
405         int                     tgt;
406         int                     lun;
407         int                     event;
408         uint64_t                wwn;
409 };

411 struct mrsas_ld {
412         dev_info_t              *dip;
413         uint8_t                 lun_type;
414         uint8_t                 flag;
415         uint8_t                 reserved[2];
329         uint8_t                 reserved[3];
416 };


419 #ifdef PDSUPPORT
420 struct mrsas_tbolt_pd {
332 struct mrsas_pd {
421         dev_info_t              *dip;
422         uint8_t                 lun_type;
423         uint8_t                 dev_id;
424         uint8_t                 flag;
336         uint8_t                 flags;
425         uint8_t                 reserved;
426 };
427 struct mrsas_tbolt_pd_info {

340 struct mrsas_pd_info {
428         uint16_t        deviceId;
429         uint16_t        seqNum;
430         uint8_t         inquiryData[96];
431         uint8_t         vpdPage83[64];
432         uint8_t         notSupported;
433         uint8_t         scsiDevType;
434         uint8_t         a;
435         uint8_t         device_speed;
436         uint32_t        mediaerrcnt;
437         uint32_t        other;
438         uint32_t        pred;
439         uint32_t        lastpred;
440         uint16_t        fwState;
441         uint8_t         disabled;
442         uint8_t         linkspwwd;
443         uint32_t        ddfType;
444         struct {
445                 uint8_t count;
446                 uint8_t isPathBroken;
447                 uint8_t connectorIndex[2];
448                 uint8_t reserved[4];
449                 uint64_t sasAddr[2];
450                 uint8_t reserved2[16];
451         } pathInfo;
452 };
453 #endif
```



```
455 typedef struct mrsas_instance {
456         uint32_t        *producer;
457         uint32_t        *consumer;

459         uint32_t        *reply_queue;
460         dma_obj_t       mfi_internal_dma_obj;
461         uint16_t        adapterresetinprogress;
462         uint16_t        deadadapter;
463         /* ThunderBolt (TB) specific */
464         dma_obj_t       mpi2_frame_pool_dma_obj;
465         dma_obj_t       request_desc_dma_obj;
466         dma_obj_t       reply_desc_dma_obj;
467         dma_obj_t       ld_map_obj[2];

469         uint8_t         init_id;
470         uint8_t         flag_ieee;
471         uint8_t         disable_online_ctrl_reset;
472         uint8_t         fw_fault_count_after_ocr;

474         uint16_t        max_num_sge;
475         uint16_t        max_fw_cmds;
476         uint32_t        max_sectors_per_req;

478         struct mrsas_cmd **cmd_list;

480         mlist_t         cmd_pool_list;
481         kmutex_t        cmd_pool_mtx;
482         kmutex_t        sync_map_mtx;

484         mlist_t         app_cmd_pool_list;
485         kmutex_t        app_cmd_pool_mtx;
486         mlist_t         cmd_app_pool_list;
487         kmutex_t        cmd_app_pool_mtx;


490         mlist_t         cmd_pend_list;
491         kmutex_t        cmd_pend_mtx;

493         dma_obj_t       mfi_evt_detail_obj;
494         struct mrsas_cmd *aen_cmd;

496         uint32_t        aen_seq_num;
497         uint32_t        aen_class_locale_word;

499         scsi_hba_tran_t         *tran;

501         kcondvar_t      int_cmd_cv;
502         kmutex_t        int_cmd_mtx;

504         kcondvar_t      aen_cmd_cv;
505         kmutex_t        aen_cmd_mtx;

507         kcondvar_t      abort_cmd_cv;
508         kmutex_t        abort_cmd_mtx;

510         kmutex_t        reg_write_mtx;
511         kmutex_t        chip_mtx;

513         dev_info_t              *dip;
514         ddi_acc_handle_t        pci_handle;

516         timeout_id_t    timeout_id;
517         uint32_t        unique_id;
518         uint16_t        fw_outstanding;
519         caddr_t         regmap;
520         ddi_acc_handle_t        regmap_handle;
```

```
521         uint8_t         isr_level;
522         ddi_iblock_cookie_t     iblock_cookie;
523         ddi_iblock_cookie_t     soft_iblock_cookie;
524         ddi_softintr_t          soft_intr_id;
525         uint8_t         softint_running;
526         uint8_t         tbolt_softint_running;
527         kmutex_t        completed_pool_mtx;
528         mlist_t         completed_pool_list;

530         caddr_t         internal_buf;
531         uint32_t        internal_buf_dmac_add;
532         uint32_t        internal_buf_size;

534         uint16_t        vendor_id;
535         uint16_t        device_id;
536         uint16_t        subsysvid;
537         uint16_t        subsysid;
538         int             instance;
539         int             baseaddress;
540         char            iocnode[16];

542         int             fm_capabilities;
543         /*
544          * Driver resources unroll flags.  The flag is set for resources that
545          * are needed to be free'd at detach() time.
546          */
547         struct _unroll {
548                 uint8_t softs;          /* The software state was allocated. */
549                 uint8_t regs;           /* Controller registers mapped. */
550                 uint8_t intr;           /* Interrupt handler added. */
551                 uint8_t reqs;           /* Request structs allocated. */
552                 uint8_t mutexs;         /* Mutex's allocated. */
553                 uint8_t taskq;          /* Task q's created. */
554                 uint8_t tran;           /* Tran struct allocated */
555                 uint8_t tranSetup;      /* Tran attached to the ddi. */
556                 uint8_t devctl;         /* Device nodes for cfgadm created. */
557                 uint8_t scsictl;        /* Device nodes for cfgadm created. */
558                 uint8_t ioctl;          /* Device nodes for ioctl's created. */
559                 uint8_t timer;          /* Timer started. */
560                 uint8_t aenPend;        /* AEN cmd pending f/w. */
561                 uint8_t mapUpdate_pend; /* LD MAP update cmd pending f/w. */
562                 uint8_t soft_isr;       /* Soft interrupt handler allocated. */
563                 uint8_t ldlist_buff;    /* Logical disk list allocated. */
564                 uint8_t pdlist_buff;    /* Physical disk list allocated. */
565                 uint8_t syncCmd;        /* Sync map command allocated. */
566                 uint8_t verBuff;        /* 2108 MFI buffer allocated. */
567                 uint8_t alloc_space_mfi;  /* Allocated space for 2108 MFI. */
568                 uint8_t alloc_space_mpi2; /* Allocated space for 2208 MPI2. */
569         } unroll;


572         /* function template pointer */
573         struct mrsas_function_template *func_ptr;


440         struct mrsas_func_ptr *func_ptr;
576         /* MSI interrupts specific */
577         ddi_intr_handle_t *intr_htable;         /* Interrupt handle array */
578         size_t          intr_htable_size;       /* Int. handle array size */
442         ddi_intr_handle_t *intr_htable;
579         int             intr_type;
580         int             intr_cnt;
445         size_t          intr_size;
581         uint_t          intr_pri;
582         int             intr_cap;
```



```
584         ddi_taskq_t     *taskq;
585         struct mrsas_ld *mr_ld_list;
586         kmutex_t        config_dev_mtx;
587         /* ThunderBolt (TB) specific */
588         ddi_softintr_t  tbolt_soft_intr_id;

590 #ifdef PDSUPPORT
591         uint32_t        mr_tbolt_pd_max;
592         struct mrsas_tbolt_pd *mr_tbolt_pd_list;
593 #endif

595         uint8_t         fast_path_io;

597         uint16_t        tbolt;
598         uint16_t        reply_read_index;
599         uint16_t        reply_size;             /* Single Reply struct size */
600         uint16_t        raid_io_msg_size;       /* Single message size */
601         uint32_t        io_request_frames_phy;
602         uint8_t         *io_request_frames;
603         /* Virtual address of request desc frame pool */
604         MRSAS_REQUEST_DESCRIPTOR_UNION  *request_message_pool;
605         /* Physical address of request desc frame pool */
606         uint32_t        request_message_pool_phy;
607         /* Virtual address of reply Frame */
608         MPI2_REPLY_DESCRIPTORS_UNION    *reply_frame_pool;
609         /* Physical address of reply Frame */
610         uint32_t        reply_frame_pool_phy;
611         uint8_t         *reply_pool_limit;      /* Last reply frame address */
612         /* Physical address of Last reply frame */
613         uint32_t        reply_pool_limit_phy;
614         uint32_t        reply_q_depth;          /* Reply Queue Depth */
615         uint8_t         max_sge_in_main_msg;
616         uint8_t         max_sge_in_chain;
617         uint8_t         chain_offset_io_req;
618         uint8_t         chain_offset_mpt_msg;
619         MR_FW_RAID_MAP_ALL *ld_map[2];
620         uint32_t        ld_map_phy[2];
621         uint32_t        size_map_info;
622         uint64_t        map_id;
623         LD_LOAD_BALANCE_INFO load_balance_info[MAX_LOGICAL_DRIVES];
624         struct mrsas_cmd *map_update_cmd;
625         uint32_t        SyncRequired;
626         kmutex_t        ocr_flags_mtx;
627         dma_obj_t       drv_ver_dma_obj;
628 } mrsas_t;


631 /*
632  * Function templates for various controller specific functions
633  */
634 struct mrsas_function_template {
635         uint32_t (*read_fw_status_reg)(struct mrsas_instance *);
454 struct mrsas_func_ptr {
455         int (*read_fw_status_reg)(struct mrsas_instance *);
636         void (*issue_cmd)(struct mrsas_cmd *, struct mrsas_instance *);
637         int (*issue_cmd_in_sync_mode)(struct mrsas_instance *,
638             struct mrsas_cmd *);
639         int (*issue_cmd_in_poll_mode)(struct mrsas_instance *,
640             struct mrsas_cmd *);
641         void (*enable_intr)(struct mrsas_instance *);
642         void (*disable_intr)(struct mrsas_instance *);
643         int (*intr_ack)(struct mrsas_instance *);
644         int (*init_adapter)(struct mrsas_instance *);
645 /*      int (*reset_adapter)(struct mrsas_instance *); */
646 };
```

```
 648 /*
 649  * ### Helper routines ###
 650  */

 652 /*
 653  * con_log() - console log routine
 654  * @param level         : indicates the severity of the message.
 655  * @fparam mt           : format string
 656  *
 657  * con_log displays the error messages on the console based on the current
 658  * debug level. Also it attaches the appropriate kernel severity level with
 659  * the message.
 660  *
 661  *
 662  * console messages debug levels
 663  */
 664 #define CL_NONE         0       /* No debug information */
 665 #define CL_ANN          1       /* print unconditionally, announcements */
 666 #define CL_ANN1         2       /* No-op  */
 667 #define CL_DLEVEL1      3       /* debug level 1, informative */
 668 #define CL_DLEVEL2      4       /* debug level 2, verbose */
 669 #define CL_DLEVEL3      5       /* debug level 3, very verbose */
 483 #define CL_TEST_OCR     1
 484 #define CL_ANN          2       /* print unconditionally, announcements */
 485 #define CL_ANN1         3       /* No o/p  */
 486 #define CL_DLEVEL1      4       /* debug level 1, informative */
 487 #define CL_DLEVEL2      5       /* debug level 2, verbose */
 488 #define CL_DLEVEL3      6       /* debug level 3, very verbose */


 671 #ifdef __SUNPRO_C
 672 #define __func__ ""
 673 #endif

 675 #define con_log(level, fmt) { if (debug_level_g >= level) cmn_err fmt; }

 677 /*
 678  * ### SCSA definitions ###
 679  */
 680 #define PKT2TGT(pkt)    ((pkt)->pkt_address.a_target)
 681 #define PKT2LUN(pkt)    ((pkt)->pkt_address.a_lun)
 682 #define PKT2TRAN(pkt)   ((pkt)->pkt_adress.a_hba_tran)
 683 #define ADDR2TRAN(ap)   ((ap)->a_hba_tran)

 685 #define TRAN2MR(tran)   (struct mrsas_instance *)(tran)->tran_hba_private)
 686 #define ADDR2MR(ap)     (TRAN2MR(ADDR2TRAN(ap))

 688 #define PKT2CMD(pkt)    ((struct scsa_cmd *)(pkt)->pkt_ha_private)
 689 #define CMD2PKT(sp)     ((sp)->cmd_pkt)
 690 #define PKT2REQ(pkt)    (&(PKT2CMD(pkt)->request))

 692 #define CMD2ADDR(cmd)   (&CMD2PKT(cmd)->pkt_address)
 693 #define CMD2TRAN(cmd)   (CMD2PKT(cmd)->pkt_address.a_hba_tran)
 694 #define CMD2MR(cmd)     (TRAN2MR(CMD2TRAN(cmd)))

 696 #define CFLAG_DMAVALID          0x0001  /* requires a dma operation */
 697 #define CFLAG_DMASEND           0x0002  /* Transfer from the device */
 698 #define CFLAG_CONSISTENT        0x0040  /* consistent data transfer */

 700 /*
 701  * ### Data structures for ioctl inteface and internal commands ###
 702  */

 704 /*
 705  * Data direction flags
 706  */
```



```
 707 #define UIOC_RD         0x00001
 708 #define UIOC_WR         0x00002

 710 #define SCP2HOST(scp)           (scp)->device->host     /* to host */
 711 #define SCP2HOSTDATA(scp)       SCP2HOST(scp)->hostdata /* to soft state */
 712 #define SCP2CHANNEL(scp)        (scp)->device->channel  /* to channel */
 713 #define SCP2TARGET(scp)         (scp)->device->id       /* to target */
 714 #define SCP2LUN(scp)            (scp)->device->lun      /* to LUN */

 716 #define SCSIHOST2ADAP(host)     (((caddr_t *)(host->hostdata))[0])
 717 #define SCP2ADAPTER(scp)                                \
 718         (struct mrsas_instance *)SCSIHOST2ADAP(SCP2HOST(scp))

 720 #define MRDRV_IS_LOGICAL_SCSA(instance, acmd)           \
 721         (acmd->device_id < MRDRV_MAX_LD) ? 1 : 0
 722 #define MRDRV_IS_LOGICAL(ap)                            \
 723         ((ap->a_target < MRDRV_MAX_LD) && (ap->a_lun == 0)) ? 1 : 0
 724 #define MAP_DEVICE_ID(instance, ap)                     \
 725         (ap->a_target)

 727 #define HIGH_LEVEL_INTR                 1
 728 #define NORMAL_LEVEL_INTR               0

 730 #define         IO_TIMEOUT_VAL          0
 731 #define         IO_RETRY_COUNT          3
 732 #define         MAX_FW_RESET_COUNT      3

 733 /*
 734  * scsa_cmd  - Per-command mr private data
 735  * @param cmd_dmahandle         :  dma handle
 736  * @param cmd_dmacookies        :  current dma cookies
 737  * @param cmd_pkt               :  scsi_pkt reference
 738  * @param cmd_dmacount          :  dma count
 739  * @param cmd_cookie            :  next cookie
 740  * @param cmd_ncookies          :  cookies per window
 741  * @param cmd_cookiecnt         :  cookies per sub-win
 742  * @param cmd_nwin              :  number of dma windows
 743  * @param cmd_curwin            :  current dma window
 744  * @param cmd_dma_offset        :  current window offset
 745  * @param cmd_dma_len           :  current window length
 746  * @param cmd_flags             :  private flags
 747  * @param cmd_cdblen            :  length of cdb
 748  * @param cmd_scblen            :  length of scb
 749  * @param cmd_buf               :  command buffer
 750  * @param channel               :  channel for scsi sub-system
 751  * @param target                :  target for scsi sub-system
 752  * @param lun                   :  LUN for scsi sub-system
 753  *
 754  * - Allocated at same time as scsi_pkt by scsi_hba_pkt_alloc(9E)
 755  * - Pointed to by pkt_ha_private field in scsi_pkt
 756  */
 757 struct scsa_cmd {
 758         ddi_dma_handle_t        cmd_dmahandle;
 759         ddi_dma_cookie_t        cmd_dmacookies[MRSAS_MAX_SGE_CNT];
 760         struct scsi_pkt         *cmd_pkt;
 761         ulong_t                 cmd_dmacount;
 762         uint_t                  cmd_cookie;
 763         uint_t                  cmd_ncookies;
 764         uint_t                  cmd_cookiecnt;
 765         uint_t                  cmd_nwin;
 766         uint_t                  cmd_curwin;
 767         off_t                   cmd_dma_offset;
 768         ulong_t                 cmd_dma_len;
 769         ulong_t                 cmd_flags;
 770         uint_t                  cmd_cdblen;
 771         uint_t                  cmd_scblen;
```

```
 772         struct buf              *cmd_buf;
 773         ushort_t                device_id;
 774         uchar_t                 islogical;
 775         uchar_t                 lun;
 776         struct mrsas_device     *mrsas_dev;
 777 };


 780 struct mrsas_cmd {
 781         /*
 782          * ThunderBolt(TB) We would be needing to have a placeholder
 783          * for RAID_MSG_IO_REQUEST inside this structure. We are
 784          * supposed to embed the mr_frame inside the RAID_MSG and post
 785          * it down to the firmware.
 786          */
 787         union mrsas_frame       *frame;
 788         uint32_t                frame_phys_addr;
 789         uint8_t                 *sense;
 790         uint8_t                 *sense1;
 791         uint32_t                sense_phys_addr;
 792         uint32_t                sense_phys_addr1;
 793         dma_obj_t               frame_dma_obj;
 794         uint8_t                 frame_dma_obj_status;

 795         uint32_t                index;
 796         uint8_t                 sync_cmd;
 797         uint8_t                 cmd_status;
 798         uint16_t                abort_aen;
 799         mlist_t                 list;
 800         uint32_t                frame_count;
 801         struct scsa_cmd         *cmd;
 802         struct scsi_pkt         *pkt;
 803         Mpi2RaidSCSIIORequest_t *scsi_io_request;
 804         Mpi2SGEIOUnion_t        *sgl;
 805         uint32_t                sgl_phys_addr;
 806         uint32_t                scsi_io_request_phys_addr;
 807         MRSAS_REQUEST_DESCRIPTOR_UNION  *request_desc;
 808         uint16_t                SMID;
 809         uint16_t                retry_count_for_ocr;
 810         uint16_t                drv_pkt_time;
 811         uint16_t                load_balance_flag;

 813 };
______unchanged_portion_omitted_


 836 #pragma pack(1)

 837 /*
 838  * SAS controller properties
 839  */
 840 struct mrsas_ctrl_prop {
 841         uint16_t        seq_num;
 842         uint16_t        pred_fail_poll_interval;
 843         uint16_t        intr_throttle_count;
 844         uint16_t        intr_throttle_timeouts;

 846         uint8_t         rebuild_rate;
 847         uint8_t         patrol_read_rate;
 848         uint8_t         bgi_rate;
 849         uint8_t         cc_rate;
 850         uint8_t         recon_rate;

 852         uint8_t         cache_flush_interval;

 854         uint8_t         spinup_drv_count;
```



```
 855         uint8_t         spinup_delay;

 857         uint8_t         cluster_enable;
 858         uint8_t         coercion_mode;
 859         uint8_t         alarm_enable;

 861         uint8_t         reserved_1[13];
 862         uint32_t        on_off_properties;
 863         uint8_t         reserved_4[28];
 864 };
______unchanged_portion_omitted_

1044 /*
1045  * ================================
1046  * MegaRAID SAS2.0 driver definitions
1047  * ================================
1048  */
1049 #define MRDRV_MAX_NUM_CMD                       1024

1051 #define MRDRV_MAX_PD_CHANNELS                   2
1052 #define MRDRV_MAX_LD_CHANNELS                   2
1053 #define MRDRV_MAX_CHANNELS                      (MRDRV_MAX_PD_CHANNELS + \
1054                                                 MRDRV_MAX_LD_CHANNELS)
1055 #define MRDRV_MAX_DEV_PER_CHANNEL               128
1056 #define MRDRV_DEFAULT_INIT_ID                   -1
1057 #define MRDRV_MAX_CMD_PER_LUN                   1000
1058 #define MRDRV_MAX_LUN                           1
1059 #define MRDRV_MAX_LD                            64

1061 #define MRDRV_RESET_WAIT_TIME                   300
1062 #define MRDRV_RESET_NOTICE_INTERVAL             5

1064 #define MRSAS_IOCTL_CMD                         0

1066 #define MRDRV_TGT_VALID                         1

1068 /*
1069  * FW can accept both 32 and 64 bit SGLs. We want to allocate 32/64 bit
1070  * SGLs based on the size of dma_addr_t
1071  */
1072 #define IS_DMA64                (sizeof (dma_addr_t) == 8)

1074 #define RESERVED0_REGISTER              0x00    /* XScale */
1075 #define IB_MSG_0_OFF                    0x10    /* XScale */
1076 #define OB_MSG_0_OFF                    0x18    /* XScale */
1077 #define IB_DOORBELL_OFF                 0x20    /* XScale & ROC */
1078 #define OB_INTR_STATUS_OFF              0x30    /* XScale & ROC */
1079 #define OB_INTR_MASK_OFF                0x34    /* XScale & ROC */
1080 #define IB_QPORT_OFF                    0x40    /* XScale & ROC */
1081 #define OB_DOORBELL_CLEAR_OFF           0xA0    /* ROC */
1082 #define OB_SCRATCH_PAD_0_OFF            0xB0    /* ROC */
1083 #define OB_INTR_MASK                    0xFFFFFFFF
1084 #define OB_DOORBELL_CLEAR_MASK          0xFFFFFFFF
1085 #define SYSTOIOP_INTERRUPT_MASK         0x80000000
1086 #define OB_SCRATCH_PAD_2_OFF            0xB4
1087 #define WRITE_TBOLT_SEQ_OFF             0x00000004
1088 #define DIAG_TBOLT_RESET_ADAPTER        0x00000004
1089 #define HOST_TBOLT_DIAG_OFF             0x00000008
1090 #define RESET_TBOLT_STATUS_OFF          0x000003C3
1091 #define WRITE_SEQ_OFF                   0x000000FC
1092 #define HOST_DIAG_OFF                   0x000000F8
1093 #define DIAG_RESET_ADAPTER              0x00000004
1094 #define DIAG_WRITE_ENABLE               0x00000080
1095 #define SYSTOIOP_INTERRUPT_MASK         0x80000000

 890 /*
```

```
 891  * All MFI register set macros accept mrsas_register_set*
 892  */
1097 #define WR_IB_WRITE_SEQ(v, instance)    ddi_put32((instance)->regmap_handle, \
1098         (uint32_t *)((uintptr_t)(instance)->regmap + WRITE_SEQ_OFF), (v))

1100 #define RD_OB_DRWE(instance)            ddi_get32((instance)->regmap_handle, \
1101         (uint32_t *)((uintptr_t)(instance)->regmap + HOST_DIAG_OFF))

1103 #define WR_IB_DRWE(v, instance)         ddi_put32((instance)->regmap_handle, \
1104         (uint32_t *)((uintptr_t)(instance)->regmap + HOST_DIAG_OFF), (v))

1106 #define IB_LOW_QPORT                    0xC0
1107 #define IB_HIGH_QPORT                   0xC4
1108 #define OB_DOORBELL_REGISTER            0x9C    /* 1078 implementation */

1110 /*
1111  * All MFI register set macros accept mrsas_register_set*
1112  */
1113 #define WR_IB_MSG_0(v, instance)        ddi_put32((instance)->regmap_handle, \
1114         (uint32_t *)((uintptr_t)(instance)->regmap + IB_MSG_0_OFF), (v))

1116 #define RD_OB_MSG_0(instance)           ddi_get32((instance)->regmap_handle, \
1117         (uint32_t *)((uintptr_t)(instance)->regmap + OB_MSG_0_OFF))

1119 #define WR_IB_DOORBELL(v, instance)     ddi_put32((instance)->regmap_handle, \
1120         (uint32_t *)((uintptr_t)(instance)->regmap + IB_DOORBELL_OFF), (v))

1122 #define RD_IB_DOORBELL(instance)        ddi_get32((instance)->regmap_handle, \
1123         (uint32_t *)((uintptr_t)(instance)->regmap + IB_DOORBELL_OFF))

1125 #define WR_OB_INTR_STATUS(v, instance)  ddi_put32((instance)->regmap_handle, \
1126         (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_STATUS_OFF), (v))

1128 #define RD_OB_INTR_STATUS(instance)     ddi_get32((instance)->regmap_handle, \
1129         (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_STATUS_OFF))

1131 #define WR_OB_INTR_MASK(v, instance)    ddi_put32((instance)->regmap_handle, \
1132         (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_MASK_OFF), (v))

1134 #define RD_OB_INTR_MASK(instance)       ddi_get32((instance)->regmap_handle, \
1135         (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_MASK_OFF))

1137 #define WR_IB_QPORT(v, instance)        ddi_put32((instance)->regmap_handle, \
1138         (uint32_t *)((uintptr_t)(instance)->regmap + IB_QPORT_OFF), (v))

1140 #define WR_OB_DOORBELL_CLEAR(v, instance) ddi_put32((instance)->regmap_handle, \
1141         (uint32_t *)((uintptr_t)(instance)->regmap + OB_DOORBELL_CLEAR_OFF), \
1142         (v))

1144 #define RD_OB_SCRATCH_PAD_0(instance)   ddi_get32((instance)->regmap_handle, \
1145         (uint32_t *)((uintptr_t)(instance)->regmap + OB_SCRATCH_PAD_0_OFF))

1147 /* Thunderbolt specific registers */
1148 #define RD_OB_SCRATCH_PAD_2(instance)   ddi_get32((instance)->regmap_handle, \
1149         (uint32_t *)((uintptr_t)(instance)->regmap + OB_SCRATCH_PAD_2_OFF))

1151 #define WR_TBOLT_IB_WRITE_SEQ(v, instance) \
1152         ddi_put32((instance)->regmap_handle, \
1153         (uint32_t *)((uintptr_t)(instance)->regmap + WRITE_TBOLT_SEQ_OFF), (v))

1155 #define RD_TBOLT_HOST_DIAG(instance)    ddi_get32((instance)->regmap_handle, \
1156         (uint32_t *)((uintptr_t)(instance)->regmap + HOST_TBOLT_DIAG_OFF))

1158 #define WR_TBOLT_HOST_DIAG(v, instance) ddi_put32((instance)->regmap_handle, \
1159         (uint32_t *)((uintptr_t)(instance)->regmap + HOST_TBOLT_DIAG_OFF), (v))
```



```
1161 #define RD_TBOLT_RESET_STAT(instance)   ddi_get32((instance)->regmap_handle, \
1162         (uint32_t *)((uintptr_t)(instance)->regmap + RESET_TBOLT_STATUS_OFF))


1165 #define WR_MPI2_REPLY_POST_INDEX(v, instance)\
1166         ddi_put32((instance)->regmap_handle,\
1167         (uint32_t *)\
1168         ((uintptr_t)(instance)->regmap + MPI2_REPLY_POST_HOST_INDEX_OFFSET),\
1169         (v))


1172 #define RD_MPI2_REPLY_POST_INDEX(instance)\
1173         ddi_get32((instance)->regmap_handle,\
1174         (uint32_t *)\
1175         ((uintptr_t)(instance)->regmap + MPI2_REPLY_POST_HOST_INDEX_OFFSET))

1177 #define WR_IB_LOW_QPORT(v, instance)    ddi_put32((instance)->regmap_handle, \
1178         (uint32_t *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT), (v))

1180 #define WR_IB_HIGH_QPORT(v, instance)   ddi_put32((instance)->regmap_handle, \
1181         (uint32_t *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT), (v))

1183 #define WR_OB_DOORBELL_REGISTER_CLEAR(v, instance)\
1184         ddi_put32((instance)->regmap_handle,\
1185         (uint32_t *)((uintptr_t)(instance)->regmap + OB_DOORBELL_REGISTER), \
1186         (v))

1188 #define WR_RESERVED0_REGISTER(v, instance) ddi_put32((instance)->regmap_handle,\
1189         (uint32_t *)((uintptr_t)(instance)->regmap + RESERVED0_REGISTER), \
1190         (v))

1192 #define RD_RESERVED0_REGISTER(instance) ddi_get32((instance)->regmap_handle, \
1193         (uint32_t *)((uintptr_t)(instance)->regmap + RESERVED0_REGISTER))



1197 /*
1198  * When FW is in MFI_STATE_READY or MFI_STATE_OPERATIONAL, the state data
1199  * of Outbound Msg Reg 0 indicates max concurrent cmds supported, max SGEs
1200  * supported per cmd and if 64-bit MFAs (M64) is enabled or disabled.
1201  */
1202 #define MFI_OB_INTR_STATUS_MASK         0x00000002

1204 /*
1205  * This MFI_REPLY_2108_MESSAGE_INTR flag is used also
1206  * in enable_intr_ppc also. Hence bit 2, i.e. 0x4 has
1207  * been set in this flag along with bit 1.
1208  */
1209 #define MFI_REPLY_2108_MESSAGE_INTR             0x00000001
1210 #define MFI_REPLY_2108_MESSAGE_INTR_MASK        0x00000005

1212 /* Fusion interrupt mask */
1213 #define MFI_FUSION_ENABLE_INTERRUPT_MASK        (0x00000008)

1215 #define MFI_POLL_TIMEOUT_SECS           60

1217 #define MFI_ENABLE_INTR(instance)  ddi_put32((instance)->regmap_handle, \
1218         (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_MASK_OFF), 1)
1219 #define MFI_DISABLE_INTR(instance)                                      \
1220 {                                                                       \
1221         uint32_t disable = 1;                                          \
1222         uint32_t mask =  ddi_get32((instance)->regmap_handle,          \
1223             (uint32_t *)((uintptr_t)(instance)->regmap + OB_INTR_MASK_OFF));\
1224         mask &= ~disable;                                              \
1225         ddi_put32((instance)->regmap_handle, (uint32_t *)              \
1226             (uintptr_t)((instance)->regmap + OB_INTR_MASK_OFF), mask);  \
```

```
1227 }

1229 /* By default, the firmware programs for 8 Kbytes of memory */
1230 #define DEFAULT_MFI_MEM_SZ      8192
1231 #define MINIMUM_MFI_MEM_SZ      4096

1233 /* DCMD Message Frame MAILBOX0-11 */
1234 #define DCMD_MBOX_SZ            12

1236 /*
1237  * on_off_property of mrsas_ctrl_prop
1238  * bit0-9, 11-31 are reserved
1239  */
1240 #define DISABLE_OCR_PROP_FLAG   0x00000400 /* bit 10 */

1242 struct mrsas_register_set {
1243         uint32_t        reserved_0[4];                  /* 0000h */
 979         uint32_t        reserved_0[4];

1245         uint32_t        inbound_msg_0;                  /* 0010h */
1246         uint32_t        inbound_msg_1;                  /* 0014h */
1247         uint32_t        outbound_msg_0;                 /* 0018h */
1248         uint32_t        outbound_msg_1;                 /* 001Ch */
 981         uint32_t        inbound_msg_0;
 982         uint32_t        inbound_msg_1;
 983         uint32_t        outbound_msg_0;
 984         uint32_t        outbound_msg_1;

1250         uint32_t        inbound_doorbell;               /* 0020h */
1251         uint32_t        inbound_intr_status;            /* 0024h */
1252         uint32_t        inbound_intr_mask;              /* 0028h */
 986         uint32_t        inbound_doorbell;
 987         uint32_t        inbound_intr_status;
 988         uint32_t        inbound_intr_mask;

1254         uint32_t        outbound_doorbell;              /* 002Ch */
1255         uint32_t        outbound_intr_status;           /* 0030h */
1256         uint32_t        outbound_intr_mask;             /* 0034h */
 990         uint32_t        outbound_doorbell;
 991         uint32_t        outbound_intr_status;
 992         uint32_t        outbound_intr_mask;

1258         uint32_t        reserved_1[2];                  /* 0038h */
 994         uint32_t        reserved_1[2];

1260         uint32_t        inbound_queue_port;             /* 0040h */
1261         uint32_t        outbound_queue_port;            /* 0044h */
 996         uint32_t        inbound_queue_port;
 997         uint32_t        outbound_queue_port;

1263         uint32_t        reserved_2[22];                 /* 0048h */
 999         uint32_t        reserved_2[22];

1265         uint32_t        outbound_doorbell_clear;        /* 00A0h */
1001         uint32_t        outbound_doorbell_clear;

1267         uint32_t        reserved_3[3];                  /* 00A4h */
1003         uint32_t        reserved_3[3];

1269         uint32_t        outbound_scratch_pad;           /* 00B0h */
1005         uint32_t        outbound_scratch_pad;

1271         uint32_t        reserved_4[3];                  /* 00B4h */
1007         uint32_t        reserved_4[3];

1273         uint32_t        inbound_low_queue_port;         /* 00C0h */
```



```
1009         uint32_t        inbound_low_queue_port;

1275         uint32_t        inbound_high_queue_port;        /* 00C4h */
1011         uint32_t        inbound_high_queue_port;

1277         uint32_t        reserved_5;                     /* 00C8h */
1278         uint32_t        index_registers[820];           /* 00CCh */
1013         uint32_t        reserved_5;
1014         uint32_t        index_registers[820];
1279 };
______unchanged_portion_omitted_

1303 struct mrsas_header {
1304         uint8_t         cmd;                            /* 00h */
1305         uint8_t         sense_len;                      /* 01h */
1306         uint8_t         cmd_status;                     /* 02h */
1307         uint8_t         scsi_status;                    /* 03h */
1040         uint8_t         cmd;
1041         uint8_t         sense_len;
1042         uint8_t         cmd_status;
1043         uint8_t         scsi_status;

1309         uint8_t         target_id;                      /* 04h */
1310         uint8_t         lun;                            /* 05h */
1311         uint8_t         cdb_len;                        /* 06h */
1312         uint8_t         sge_count;                      /* 07h */
1045         uint8_t         target_id;
1046         uint8_t         lun;
1047         uint8_t         cdb_len;
1048         uint8_t         sge_count;

1314         uint32_t        context;                        /* 08h */
1315         uint8_t         req_id;                         /* 0Ch */
1316         uint8_t         msgvector;                      /* 0Dh */
1317         uint16_t        pad_0;                          /* 0Eh */
1050         uint32_t        context;
1051         uint8_t         req_id;
1052         uint8_t         msgvector;
1053         uint16_t        pad_0;

1319         uint16_t        flags;                          /* 10h */
1320         uint16_t        timeout;                        /* 12h */
1321         uint32_t        data_xferlen;                   /* 14h */
1055         uint16_t        flags;
1056         uint16_t        timeout;
1057         uint32_t        data_xferlen;
1322 };
______unchanged_portion_omitted_

1329 struct mrsas_init_frame {
1330         uint8_t         cmd;                            /* 00h */
1331         uint8_t         reserved_0;                     /* 01h */
1332         uint8_t         cmd_status;                     /* 02h */
1066         uint8_t         cmd;
1067         uint8_t         reserved_0;
1068         uint8_t         cmd_status;

1334         uint8_t         reserved_1;                     /* 03h */
1335         uint32_t        reserved_2;                     /* 04h */
1070         uint8_t         reserved_1;
1071         uint32_t        reserved_2;

1337         uint32_t        context;                        /* 08h */
1338         uint8_t         req_id;                         /* 0Ch */
1339         uint8_t         msgvector;                      /* 0Dh */
1340         uint16_t        pad_0;                          /* 0Eh */
```

```
1073         uint32_t        context;
1074         uint8_t         req_id;
1075         uint8_t         msgvector;
1076         uint16_t        pad_0;

1342         uint16_t        flags;                          /* 10h */
1343         uint16_t        reserved_3;                     /* 12h */
1344         uint32_t        data_xfer_len;                  /* 14h */
1078         uint16_t        flags;
1079         uint16_t        reserved_3;
1080         uint32_t        data_xfer_len;

1346         uint32_t        queue_info_new_phys_addr_lo;    /* 18h */
1347         uint32_t        queue_info_new_phys_addr_hi;    /* 1Ch */
1348         uint32_t        queue_info_old_phys_addr_lo;    /* 20h */
1349         uint32_t        queue_info_old_phys_addr_hi;    /* 24h */
1350         uint64_t        driverversion;                  /* 28h */
1351         uint32_t        reserved_4[4];                  /* 30h */
1082         uint32_t        queue_info_new_phys_addr_lo;
1083         uint32_t        queue_info_new_phys_addr_hi;
1084         uint32_t        queue_info_old_phys_addr_lo;
1085         uint32_t        queue_info_old_phys_addr_hi;

1087         uint32_t        reserved_4[6];
1352 };

1354 struct mrsas_init_queue_info {
1355         uint32_t                init_flags;                     /* 00h */
1356         uint32_t                reply_queue_entries;            /* 04h */
1091         uint32_t                init_flags;
1092         uint32_t                reply_queue_entries;

1358         uint32_t                reply_queue_start_phys_addr_lo; /* 08h */
1359         uint32_t                reply_queue_start_phys_addr_hi; /* 0Ch */
1360         uint32_t                producer_index_phys_addr_lo;    /* 10h */
1361         uint32_t                producer_index_phys_addr_hi;    /* 14h */
1362         uint32_t                consumer_index_phys_addr_lo;    /* 18h */
1363         uint32_t                consumer_index_phys_addr_hi;    /* 1Ch */
1094         uint32_t                reply_queue_start_phys_addr_lo;
1095         uint32_t                reply_queue_start_phys_addr_hi;
1096         uint32_t                producer_index_phys_addr_lo;
1097         uint32_t                producer_index_phys_addr_hi;
1098         uint32_t                consumer_index_phys_addr_lo;
1099         uint32_t                consumer_index_phys_addr_hi;
1364 };

1366 struct mrsas_io_frame {
1367         uint8_t                 cmd;                    /* 00h */
1368         uint8_t                 sense_len;              /* 01h */
1369         uint8_t                 cmd_status;             /* 02h */
1370         uint8_t                 scsi_status;            /* 03h */
1103         uint8_t                 cmd;
1104         uint8_t                 sense_len;
1105         uint8_t                 cmd_status;
1106         uint8_t                 scsi_status;

1372         uint8_t                 target_id;              /* 04h */
1373         uint8_t                 access_byte;            /* 05h */
1374         uint8_t                 reserved_0;             /* 06h */
1375         uint8_t                 sge_count;              /* 07h */
1108         uint8_t                 target_id;
1109         uint8_t                 access_byte;
1110         uint8_t                 reserved_0;
1111         uint8_t                 sge_count;

1377         uint32_t                context;                /* 08h */
```



```
1378         uint8_t                 req_id;                 /* 0Ch */
1379         uint8_t                 msgvector;              /* 0Dh */
1380         uint16_t                pad_0;                  /* 0Eh */
1113         uint32_t                context;
1114         uint8_t                 req_id;
1115         uint8_t                 msgvector;
1116         uint16_t                pad_0;

1382         uint16_t                flags;                  /* 10h */
1383         uint16_t                timeout;                /* 12h */
1384         uint32_t                lba_count;              /* 14h */
1118         uint16_t                flags;
1119         uint16_t                timeout;
1120         uint32_t                lba_count;

1386         uint32_t                sense_buf_phys_addr_lo; /* 18h */
1387         uint32_t                sense_buf_phys_addr_hi; /* 1Ch */
1122         uint32_t                sense_buf_phys_addr_lo;
1123         uint32_t                sense_buf_phys_addr_hi;

1389         uint32_t                start_lba_lo;           /* 20h */
1390         uint32_t                start_lba_hi;           /* 24h */
1125         uint32_t                start_lba_lo;
1126         uint32_t                start_lba_hi;

1392         union mrsas_sgl         sgl;                    /* 28h */
1128         union mrsas_sgl         sgl;
1393 };

1395 struct mrsas_pthru_frame {
1396         uint8_t                 cmd;                    /* 00h */
1397         uint8_t                 sense_len;              /* 01h */
1398         uint8_t                 cmd_status;             /* 02h */
1399         uint8_t                 scsi_status;            /* 03h */
1132         uint8_t                 cmd;
1133         uint8_t                 sense_len;
1134         uint8_t                 cmd_status;
1135         uint8_t                 scsi_status;

1401         uint8_t                 target_id;              /* 04h */
1402         uint8_t                 lun;                    /* 05h */
1403         uint8_t                 cdb_len;                /* 06h */
1404         uint8_t                 sge_count;              /* 07h */
1137         uint8_t                 target_id;
1138         uint8_t                 lun;
1139         uint8_t                 cdb_len;
1140         uint8_t                 sge_count;

1406         uint32_t                context;                /* 08h */
1407         uint8_t                 req_id;                 /* 0Ch */
1408         uint8_t                 msgvector;              /* 0Dh */
1409         uint16_t                pad_0;                  /* 0Eh */
1142         uint32_t                context;
1143         uint8_t                 req_id;
1144         uint8_t                 msgvector;
1145         uint16_t                pad_0;

1411         uint16_t                flags;                  /* 10h */
1412         uint16_t                timeout;                /* 12h */
1413         uint32_t                data_xfer_len;          /* 14h */
1147         uint16_t                flags;
1148         uint16_t                timeout;
1149         uint32_t                data_xfer_len;

1415         uint32_t                sense_buf_phys_addr_lo; /* 18h */
1416         uint32_t                sense_buf_phys_addr_hi; /* 1Ch */
```

```
1151        uint32_t                sense_buf_phys_addr_lo;
1152        uint32_t                sense_buf_phys_addr_hi;

1418        uint8_t                 cdb[16];                /* 20h */
1419        union mrsas_sgl         sgl;                    /* 30h */
1154        uint8_t                 cdb[16];
1155        union mrsas_sgl         sgl;
1420 };

1422 struct mrsas_dcmd_frame {
1423        uint8_t                 cmd;                    /* 00h */
1424        uint8_t                 reserved_0;             /* 01h */
1425        uint8_t                 cmd_status;             /* 02h */
1426        uint8_t                 reserved_1[4];          /* 03h */
1427        uint8_t                 sge_count;              /* 07h */
1159        uint8_t                 cmd;
1160        uint8_t                 reserved_0;
1161        uint8_t                 cmd_status;
1162        uint8_t                 reserved_1[4];
1163        uint8_t                 sge_count;

1429        uint32_t                context;                /* 08h */
1430        uint8_t                 req_id;                 /* 0Ch */
1431        uint8_t                 msgvector;              /* 0Dh */
1432        uint16_t                pad_0;                  /* 0Eh */
1165        uint32_t                context;
1166        uint8_t                 req_id;
1167        uint8_t                 msgvector;
1168        uint16_t                pad_0;

1434        uint16_t                flags;                  /* 10h */
1435        uint16_t                timeout;                /* 12h */
1170        uint16_t                flags;
1171        uint16_t                timeout;

1437        uint32_t                data_xfer_len;          /* 14h */
1438        uint32_t                opcode;                 /* 18h */
1173        uint32_t                data_xfer_len;
1174        uint32_t                opcode;

1440        /* uint8_t              mbox[DCMD_MBOX_SZ]; */  /* 1Ch */
1441        union {                                         /* 1Ch */
1176        union {
1442                uint8_t b[DCMD_MBOX_SZ];
1443                uint16_t s[6];
1444                uint32_t w[3];
1445        } mbox;

1447        union mrsas_sgl         sgl;                    /* 28h */
1182        union mrsas_sgl         sgl;
1448 };

1450 struct mrsas_abort_frame {
1451        uint8_t         cmd;                            /* 00h */
1452        uint8_t         reserved_0;                     /* 01h */
1453        uint8_t         cmd_status;                     /* 02h */
1186        uint8_t         cmd;
1187        uint8_t         reserved_0;
1188        uint8_t         cmd_status;

1455        uint8_t         reserved_1;                     /* 03h */
1456        uint32_t        reserved_2;                     /* 04h */
1190        uint8_t         reserved_1;
1191        uint32_t        reserved_2;

1458        uint32_t        context;                        /* 08h */
```



```
1459        uint8_t         req_id;                         /* 0Ch */
1460        uint8_t         msgvector;                      /* 0Dh */
1461        uint16_t        pad_0;                          /* 0Eh */
1193        uint32_t        context;
1194        uint8_t         req_id;
1195        uint8_t         msgvector;
1196        uint16_t        pad_0;

1463        uint16_t        flags;                          /* 10h */
1464        uint16_t        reserved_3;                     /* 12h */
1465        uint32_t        reserved_4;                     /* 14h */
1198        uint16_t        flags;
1199        uint16_t        reserved_3;
1200        uint32_t        reserved_4;

1467        uint32_t        abort_context;                  /* 18h */
1468        uint32_t        pad_1;                          /* 1Ch */
1202        uint32_t        abort_context;
1203        uint32_t        pad_1;

1470        uint32_t        abort_mfi_phys_addr_lo;         /* 20h */
1471        uint32_t        abort_mfi_phys_addr_hi;         /* 24h */
1205        uint32_t        abort_mfi_phys_addr_lo;
1206        uint32_t        abort_mfi_phys_addr_hi;

1473        uint32_t        reserved_5[6];                  /* 28h */
1208        uint32_t        reserved_5[6];
1474 };

1476 struct mrsas_smp_frame {
1477        uint8_t         cmd;                            /* 00h */
1478        uint8_t         reserved_1;                     /* 01h */
1479        uint8_t         cmd_status;                     /* 02h */
1480        uint8_t         connection_status;              /* 03h */
1212        uint8_t         cmd;
1213        uint8_t         reserved_1;
1214        uint8_t         cmd_status;
1215        uint8_t         connection_status;

1482        uint8_t         reserved_2[3];                  /* 04h */
1483        uint8_t         sge_count;                      /* 07h */
1217        uint8_t         reserved_2[3];
1218        uint8_t         sge_count;

1485        uint32_t        context;                        /* 08h */
1486        uint8_t         req_id;                         /* 0Ch */
1487        uint8_t         msgvector;                      /* 0Dh */
1488        uint16_t        pad_0;                          /* 0Eh */
1220        uint32_t        context;
1221        uint8_t         req_id;
1222        uint8_t         msgvector;
1223        uint16_t        pad_0;

1490        uint16_t        flags;                          /* 10h */
1491        uint16_t        timeout;                        /* 12h */
1225        uint16_t        flags;
1226        uint16_t        timeout;

1493        uint32_t        data_xfer_len;                  /* 14h */
1228        uint32_t        data_xfer_len;

1495        uint64_t        sas_addr;                       /* 20h */
1230        uint64_t        sas_addr;

1497        union mrsas_sgl sgl[2];                         /* 28h */
1232        union mrsas_sgl sgl[2];
```

```
1498 };

1500 struct mrsas_stp_frame {
1501         uint8_t         cmd;                    /* 00h */
1502         uint8_t         reserved_1;             /* 01h */
1503         uint8_t         cmd_status;             /* 02h */
1504         uint8_t         connection_status;      /* 03h */
1236         uint8_t         cmd;
1237         uint8_t         reserved_1;
1238         uint8_t         cmd_status;
1239         uint8_t         connection_status;

1506         uint8_t         target_id;              /* 04h */
1507         uint8_t         reserved_2[2];          /* 04h */
1508         uint8_t         sge_count;              /* 07h */
1241         uint8_t         target_id;
1242         uint8_t         reserved_2[2];
1243         uint8_t         sge_count;

1510         uint32_t        context;                /* 08h */
1511         uint8_t         req_id;                 /* 0Ch */
1512         uint8_t         msgvector;              /* 0Dh */
1513         uint16_t        pad_0;                  /* 0Eh */
1245         uint32_t        context;
1246         uint8_t         req_id;
1247         uint8_t         msgvector;
1248         uint16_t        pad_0;

1515         uint16_t        flags;                  /* 10h */
1516         uint16_t        timeout;                /* 12h */
1250         uint16_t        flags;
1251         uint16_t        timeout;

1518         uint32_t        data_xfer_len;          /* 14h */
1253         uint32_t        data_xfer_len;

1520         uint16_t        fis[10];                /* 28h */
1521         uint32_t        stp_flags;              /* 3C */
1522         union mrsas_sgl sgl;                    /* 40 */
1255         uint16_t        fis[10];
1256         uint32_t        stp_flags;
1257         union mrsas_sgl sgl;
1523 };
______unchanged_portion_omitted_

1950 #pragma pack()

1952 #ifndef DDI_VENDOR_LSI
1953 #define DDI_VENDOR_LSI          "LSI"
1954 #endif /* DDI_VENDOR_LSI */

1956 int mrsas_config_scsi_device(struct mrsas_instance *,
1957     struct scsi_device *, dev_info_t **);
1690 #ifndef KMDB_MODULE
1691 static int      mrsas_getinfo(dev_info_t *, ddi_info_cmd_t,  void *, void **);
1692 static int      mrsas_attach(dev_info_t *, ddi_attach_cmd_t);
1693 #ifdef __sparc
1694 static int      mrsas_reset(dev_info_t *, ddi_reset_cmd_t);
1695 #else /* __sparc */
1696 static int      mrsas_quiesce(dev_info_t *);
1697 #endif  /* __sparc */
1698 static int      mrsas_detach(dev_info_t *, ddi_detach_cmd_t);
1699 static int      mrsas_open(dev_t *, int, int, cred_t *);
1700 static int      mrsas_close(dev_t, int, int, cred_t *);
1701 static int      mrsas_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);
```



```
1959 #ifdef PDSUPPORT
1960 int mrsas_tbolt_config_pd(struct mrsas_instance *, uint16_t,
1961     uint8_t, dev_info_t **);
1962 #endif

1964 dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t, uint8_t);
1965 int mrsas_service_evt(struct mrsas_instance *, int, int, int, uint64_t);
1966 void return_raid_msg_pkt(struct mrsas_instance *, struct mrsas_cmd *);
1967 struct mrsas_cmd *get_raid_msg_mfi_pkt(struct mrsas_instance *);
1968 void return_raid_msg_mfi_pkt(struct mrsas_instance *, struct mrsas_cmd *);

1970 int     alloc_space_for_mpi2(struct mrsas_instance *);
1971 void    fill_up_drv_ver(struct mrsas_drv_ver *dv);

1973 int     mrsas_issue_init_mpi2(struct mrsas_instance *);
1974 struct scsi_pkt *mrsas_tbolt_tran_init_pkt(struct scsi_address *, register
1703 static int      mrsas_tran_tgt_init(dev_info_t *, dev_info_t *,
1704                     scsi_hba_tran_t *, struct scsi_device *);
1705 static struct scsi_pkt *mrsas_tran_init_pkt(struct scsi_address *, register
1975                     struct scsi_pkt *, struct buf *, int, int, int, int,
1976                     int (*)(), caddr_t);
1977 int     mrsas_tbolt_tran_start(struct scsi_address *,
1708 static int      mrsas_tran_start(struct scsi_address *,
1978                     register struct scsi_pkt *);
1979 uint32_t tbolt_read_fw_status_reg(struct mrsas_instance *);
1980 void    tbolt_issue_cmd(struct mrsas_cmd *, struct mrsas_instance *);
1981 int     tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *,
1710 static int      mrsas_tran_abort(struct scsi_address *, struct scsi_pkt *);
1711 static int      mrsas_tran_reset(struct scsi_address *, int);
1712 static int      mrsas_tran_getcap(struct scsi_address *, char *, int);
1713 static int      mrsas_tran_setcap(struct scsi_address *, char *, int, int);
1714 static void     mrsas_tran_destroy_pkt(struct scsi_address *,
1715                     struct scsi_pkt *);
1716 static void     mrsas_tran_dmafree(struct scsi_address *, struct scsi_pkt *);
1717 static void     mrsas_tran_sync_pkt(struct scsi_address *, struct scsi_pkt *);
1718 static uint_t   mrsas_isr();
1719 static uint_t   mrsas_softintr();

1721 static int      init_mfi(struct mrsas_instance *);
1722 static int      mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t);
1723 static int      mrsas_alloc_dma_obj(struct mrsas_instance *, dma_obj_t *,
1724                     uchar_t);
1725 static struct mrsas_cmd *get_mfi_pkt(struct mrsas_instance *);
1726 static void     return_mfi_pkt(struct mrsas_instance *,
1982                     struct mrsas_cmd *);
1983 int     tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *,

1729 static void     free_space_for_mfi(struct mrsas_instance *);
1730 static void     free_additional_dma_buffer(struct mrsas_instance *);
1731 static int      alloc_additional_dma_buffer(struct mrsas_instance *);
1732 static int      read_fw_status_reg_ppc(struct mrsas_instance *);
1733 static void     issue_cmd_ppc(struct mrsas_cmd *, struct mrsas_instance *);
1734 static int      issue_cmd_in_poll_mode_ppc(struct mrsas_instance *,
1984                     struct mrsas_cmd *);
1985 void    tbolt_enable_intr(struct mrsas_instance *);
1986 void    tbolt_disable_intr(struct mrsas_instance *);
1987 int     tbolt_intr_ack(struct mrsas_instance *);
1988 uint_t  mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *);
1989     uint_t tbolt_softintr();
1990 int     mrsas_tbolt_dma(struct mrsas_instance *, uint32_t, int, int (*)());
1991 int     mrsas_check_dma_handle(ddi_dma_handle_t handle);
1992 int     mrsas_check_acc_handle(ddi_acc_handle_t handle);
1993 int     mrsas_dma_alloc(struct mrsas_instance *, struct scsi_pkt *,
1736 static int      issue_cmd_in_sync_mode_ppc(struct mrsas_instance *,
1737                     struct mrsas_cmd *);
1738 static void     enable_intr_ppc(struct mrsas_instance *);
```

```
1739 static void     disable_intr_ppc(struct mrsas_instance *);
1740 static int      intr_ack_ppc(struct mrsas_instance *);
1741 static int      mfi_state_transition_to_ready(struct mrsas_instance *);
1742 static void     destroy_mfi_frame_pool(struct mrsas_instance *);
1743 static int      create_mfi_frame_pool(struct mrsas_instance *);
1744 static int      mrsas_dma_alloc(struct mrsas_instance *, struct scsi_pkt *,
1994                     struct buf *, int, int (*)());
1995 int     mrsas_dma_move(struct mrsas_instance *,
1746 static int      mrsas_dma_move(struct mrsas_instance *,
1996                     struct scsi_pkt *, struct buf *);
1997 int     mrsas_alloc_dma_obj(struct mrsas_instance *, dma_obj_t *,
1998                     uchar_t);
1999 void    mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2000 int     mrsas_dma_alloc_dmd(struct mrsas_instance *, dma_obj_t *);
2001 void    tbolt_complete_cmd_in_sync_mode(struct mrsas_instance *,
2002         struct mrsas_cmd *);
2003 int     alloc_req_rep_desc(struct mrsas_instance *);
2004 int             mrsas_mode_sense_build(struct scsi_pkt *);
2005 void            push_pending_mfi_pkt(struct mrsas_instance *,
2006                     struct mrsas_cmd *);
2007 int     mrsas_issue_pending_cmds(struct mrsas_instance *);
2008 int     mrsas_print_pending_cmds(struct mrsas_instance *);
2009 int     mrsas_complete_pending_cmds(struct mrsas_instance *);
1748 static void     flush_cache(struct mrsas_instance *instance);
1749 static void     display_scsi_inquiry(caddr_t);
1750 static int      start_mfi_aen(struct mrsas_instance *instance);
1751 static int      handle_drv_ioctl(struct mrsas_instance *instance,
1752                     struct mrsas_ioctl *ioctl, int mode);
1753 static int      handle_mfi_ioctl(struct mrsas_instance *instance,
1754                     struct mrsas_ioctl *ioctl, int mode);
1755 static int      handle_mfi_aen(struct mrsas_instance *instance,
1756                     struct mrsas_aen *aen);
1757 static void     fill_up_drv_ver(struct mrsas_drv_ver *dv);
1758 static struct mrsas_cmd *build_cmd(struct mrsas_instance *instance,
1759                     struct scsi_address *ap, struct scsi_pkt *pkt,
1760                     uchar_t *cmd_done);
1761 #ifndef __sparc
1762 static int      wait_for_outstanding(struct mrsas_instance *instance);
1763 #endif  /* __sparc */
1764 static int      register_mfi_aen(struct mrsas_instance *instance,
1765                     uint32_t seq_num, uint32_t class_locale_word);
1766 static int      issue_mfi_pthru(struct mrsas_instance *instance, struct
1767                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1768 static int      issue_mfi_dcmd(struct mrsas_instance *instance, struct
1769                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1770 static int      issue_mfi_smp(struct mrsas_instance *instance, struct
1771                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1772 static int      issue_mfi_stp(struct mrsas_instance *instance, struct
1773                     mrsas_ioctl *ioctl, struct mrsas_cmd *cmd, int mode);
1774 static int      abort_aen_cmd(struct mrsas_instance *instance,
1775                     struct mrsas_cmd *cmd_to_abort);

2011 int     create_mfi_frame_pool(struct mrsas_instance *);
2012 void    destroy_mfi_frame_pool(struct mrsas_instance *);
2013 int     create_mfi_mpi_frame_pool(struct mrsas_instance *);
2014 void    destroy_mfi_mpi_frame_pool(struct mrsas_instance *);
2015 int     create_mpi2_frame_pool(struct mrsas_instance *);
2016 void    destroy_mpi2_frame_pool(struct mrsas_instance *);
2017 int     mrsas_free_dma_obj(struct mrsas_instance *, dma_obj_t);
2018 void    mrsas_tbolt_free_additional_dma_buffer(struct mrsas_instance *);
2019 void    free_req_desc_pool(struct mrsas_instance *);
2020 void    free_space_for_mpi2(struct mrsas_instance *);
2021 void    mrsas_dump_reply_desc(struct mrsas_instance *);
2022 void    tbolt_complete_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2023 void    display_scsi_inquiry(caddr_t);
2024 void    service_mfi_aen(struct mrsas_instance *, struct mrsas_cmd *);
```



```
2025 int     mrsas_mode_sense_build(struct scsi_pkt *);
2026 int     mrsas_tbolt_get_ld_map_info(struct mrsas_instance *);
2027 struct mrsas_cmd *mrsas_tbolt_build_poll_cmd(struct mrsas_instance *,
2028         struct scsi_address *, struct scsi_pkt *, uchar_t *);
2029 int     mrsas_tbolt_reset_ppc(struct mrsas_instance *instance);
2030 void    mrsas_tbolt_kill_adapter(struct mrsas_instance *instance);
2031 int     abort_syncmap_cmd(struct mrsas_instance *, struct mrsas_cmd *);
2032 void    mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2033     struct IO_REQUEST_INFO *, Mpi2RaidSCSIIORequest_t *, U32);
1777 static int      mrsas_common_check(struct mrsas_instance *instance,
1778                     struct  mrsas_cmd *cmd);
1779 static void     mrsas_fm_init(struct mrsas_instance *instance);
1780 static void     mrsas_fm_fini(struct mrsas_instance *instance);
1781 static int      mrsas_fm_error_cb(dev_info_t *, ddi_fm_error_t *,
1782                     const void *);
1783 static void     mrsas_fm_ereport(struct mrsas_instance *instance,
1784                     char *detail);
1785 static int      mrsas_check_dma_handle(ddi_dma_handle_t handle);
1786 static int      mrsas_check_acc_handle(ddi_acc_handle_t handle);

1788 static void     mrsas_rem_intrs(struct mrsas_instance *instance);
1789 static int      mrsas_add_intrs(struct mrsas_instance *instance, int intr_type);

2036 int mrsas_init_adapter_ppc(struct mrsas_instance *instance);
2037 int mrsas_init_adapter_tbolt(struct mrsas_instance *instance);
2038 int mrsas_init_adapter(struct mrsas_instance *instance);
1791 static void     mrsas_tran_tgt_free(dev_info_t *, dev_info_t *,
1792                     scsi_hba_tran_t *, struct scsi_device *);
1793 static int      mrsas_tran_bus_config(dev_info_t *, uint_t,
1794                     ddi_bus_config_op_t, void *, dev_info_t **);
1795 static int      mrsas_parse_devname(char *, int *, int *);
1796 static int      mrsas_config_all_devices(struct mrsas_instance *);
1797 static int      mrsas_config_scsi_device(struct mrsas_instance *,
1798                     struct scsi_device *, dev_info_t **);
1799 static int      mrsas_config_ld(struct mrsas_instance *, uint16_t,
1800                                 uint8_t, dev_info_t **);
1801 static dev_info_t *mrsas_find_child(struct mrsas_instance *, uint16_t,
1802                         uint8_t);
1803 static int      mrsas_name_node(dev_info_t *, char *, int);
1804 static void     mrsas_issue_evt_taskq(struct mrsas_eventinfo *);
1805 static int      mrsas_service_evt(struct mrsas_instance *, int, int, int,
1806                         uint64_t);
1807 static int      mrsas_mode_sense_build(struct scsi_pkt *);
1808 static void     push_pending_mfi_pkt(struct mrsas_instance *,
1809                         struct mrsas_cmd *);
1810 static int      mrsas_issue_init_mfi(struct mrsas_instance *);
1811 static int      mrsas_issue_pending_cmds(struct mrsas_instance *);
1812 static int      mrsas_print_pending_cmds(struct mrsas_instance *);
1813 static int  mrsas_complete_pending_cmds(struct mrsas_instance *);
1814 static int      mrsas_reset_ppc(struct mrsas_instance *);
1815 static uint32_t mrsas_initiate_ocr_if_fw_is_faulty(struct mrsas_instance *);
1816 static int  mrsas_kill_adapter(struct mrsas_instance *);
1817 static void io_timeout_checker(void *instance);
1818 static void complete_cmd_in_sync_mode(struct mrsas_instance *,
1819                 struct mrsas_cmd *);

2040 int mrsas_alloc_cmd_pool(struct mrsas_instance *instance);
2041 void mrsas_free_cmd_pool(struct mrsas_instance *instance);
1821 #endif  /* KMDB_MODULE */

2043 void mrsas_print_cmd_details(struct mrsas_instance *, struct mrsas_cmd *, int);
2044 struct mrsas_cmd *get_raid_msg_pkt(struct mrsas_instance *);

2046 int mfi_state_transition_to_ready(struct mrsas_instance *);
```

```
2049 /* FMA functions. */
2050 int mrsas_common_check(struct mrsas_instance *, struct  mrsas_cmd *);
2051 void mrsas_fm_ereport(struct mrsas_instance *, char *);


2054 #ifdef  __cplusplus
2055 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2874 Tue Nov  6 14:28:57 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_list.c
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * mr_sas_list.h: header for mr_sas
 *
 * Solaris MegaRAID driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 */

/* Copyright 2012 Nexenta Systems, Inc. All rights reserved. */

/*
 * Extract C functions from LSI-provided mr_sas_list.h such that we can both
 * be lint-clean and provide a slightly better source organizational model
 * beyond preprocessor abuse.
 */

#include "mr_sas_list.h"

/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void
__list_add(struct mlist_head *new, struct mlist_head *prev,
    struct mlist_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

/*
 * mlist_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
void
mlist_add(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head, head->next);
}

/*
 * mlist_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
void
mlist_add_tail(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head->prev, head);
}
```



```
/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void
__list_del(struct mlist_head *prev, struct mlist_head *next)
{
        next->prev = prev;
        prev->next = next;
}

#if 0
/*
 * mlist_del - deletes entry from list.
 * @entry:      the element to delete from the list.
 * Note:        list_empty on entry does not return true after this, the entry
 * is in an undefined state.
 */

void
mlist_del(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        entry->next = entry->prev = 0;
}
#endif

/*
 * mlist_del_init - deletes entry from list and reinitialize it.
 * @entry: the element to delete from the list.
 */
void
mlist_del_init(struct mlist_head *entry)
{
        __list_del(entry->prev, entry->next);
        INIT_LIST_HEAD(entry);
}

/*
 * mlist_empty - tests whether a list is empty
 * @head: the list to test.
 */
int
mlist_empty(struct mlist_head *head)
{
        return (head->next == head);
}

/*
 * mlist_splice - join two lists
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
void
mlist_splice(struct mlist_head *list, struct mlist_head *head)
{
        struct mlist_head *first = list->next;

        if (first != list) {
                struct mlist_head *last = list->prev;
                struct mlist_head *at = head->next;
```

```
128                 first->prev = head;
129                 head->next = first;

131                 last->next = at;
132                 at->prev = last;
133         }
134 }
```

```
**********************************************************
    3875 Tue Nov  6 14:28:57 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_list.h
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * mr_sas_list.h: header for mr_sas
 *
 * Solaris MegaRAID driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * Copyright (c) 2008-2009, LSI Logic Corporation.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice,
 *    this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright notice,
 *    this list of conditions and the following disclaimer in the documentation
 *    and/or other materials provided with the distribution.
 *
 * 3. 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.
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#ifndef _MR_SAS_LIST_H_
#define _MR_SAS_LIST_H_

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Simple doubly linked list implementation.
 *
 * Some of the internal functions ("__xxx") are useful when
 * manipulating whole lists rather than single entries, as
 * sometimes we already know the next/prev entries and we can
 * generate better code by using them directly rather than
 * using the generic single-entry routines.
 */

struct mlist_head {
        struct mlist_head *next, *prev;
};
```

_____*unchanged_portion_omitted_*


```
#ifndef KMDB_MODULE
/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static void __list_add(struct mlist_head *new,
        struct mlist_head *prev,
        struct mlist_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

void mlist_add(struct mlist_head *, struct mlist_head *);
void mlist_add_tail(struct mlist_head *, struct mlist_head *);
#if 0
void mlist_del(struct mlist_head *);
#endif
void mlist_del_init(struct mlist_head *);
int mlist_empty(struct mlist_head *);
void mlist_splice(struct mlist_head *, struct mlist_head *);

/* TODO: set this */
#if 0
#pragma inline(list_add, list_add_tail, __list_del, list_del,
                list_del_init, list_empty, list_splice)
#endif
/*
 * mlist_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static void mlist_add(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head, head->next);
}


/*
 * mlist_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static void mlist_add_tail(struct mlist_head *new, struct mlist_head *head)
{
        __list_add(new, head->prev, head);
}



/*
 * Delete a list entry by making the prev/next entries
 * point to each other.
 *
 * This is only for internal list manipulation where we know
```

```
125  * the prev/next entries already!
126  */
127 static void __list_del(struct mlist_head *prev,
128                        struct mlist_head *next)
129 {
130         next->prev = prev;
131         prev->next = next;
132 }


135 /*
136  * mlist_del_init - deletes entry from list and reinitialize it.
137  * @entry: the element to delete from the list.
138  */
139 static void mlist_del_init(struct mlist_head *entry)
140 {
141         __list_del(entry->prev, entry->next);
142         INIT_LIST_HEAD(entry);
143 }


146 /*
147  * mlist_empty - tests whether a list is empty
148  * @head: the list to test.
149  */
150 static int mlist_empty(struct mlist_head *head)
151 {
152         return (head->next == head);
153 }


156 /*
157  * mlist_splice - join two lists
158  * @list: the new list to add.
159  * @head: the place to add it in the first list.
160  */
161 static void mlist_splice(struct mlist_head *list, struct mlist_head *head)
162 {
163         struct mlist_head *first = list->next;

165         if (first != list) {
166                 struct mlist_head *last = list->prev;
167                 struct mlist_head *at = head->next;

169                 first->prev = head;
170                 head->next = first;

172                 last->next = at;
173                 at->prev = last;
174         }
175 }
176 #endif /* KMDB_MODULE */

178 /*
 86  * mlist_entry - get the struct for this entry
 87  * @ptr:        the &struct mlist_head pointer.
 88  * @type:       the type of the struct this is embedded in.
 89  * @member:     the name of the list_struct within the struct.
 90  */
 91 #define mlist_entry(ptr, type, member) \
 92         ((type *)((size_t)(ptr) - offsetof(type, member)))


 95 /*
 96  * mlist_for_each       -       iterate over a list
 97  * @pos:        the &struct mlist_head to use as a loop counter.
```



```
 98  * @head:       the head for your list.
 99  */
100 #define mlist_for_each(pos, head) \
101         for (pos = (head)->next, prefetch(pos->next); pos != (head); \
102                 pos = pos->next, prefetch(pos->next))


105 /*
106  * mlist_for_each_safe - iterate over a list safe against removal of list entry
107  * @pos:        the &struct mlist_head to use as a loop counter.
108  * @n:          another &struct mlist_head to use as temporary storage
109  * @head:       the head for your list.
110  */
111 #define mlist_for_each_safe(pos, n, head) \
112         for (pos = (head)->next, n = pos->next; pos != (head); \
113                 pos = n, n = pos->next)

115 #ifdef __cplusplus
116 }
```

_____unchanged_portion_omitted_

```
**********************************************************
  106990 Tue Nov  6 14:28:58 2012
new/usr/src/uts/common/io/mr_sas/mr_sas_tbolt.c
3178 Support for LSI 2208 chipset in mr_sas
**********************************************************
```

```
/*
 * mr_sas_tbolt.c: source for mr_sas driver for New Generation.
 * i.e. Thunderbolt and Invader
 *
 * Solaris MegaRAID device driver for SAS2.0 controllers
 * Copyright (c) 2008-2012, LSI Logic Corporation.
 * All rights reserved.
 *
 * Version:
 * Author:
 *              Swaminathan K S
 *              Arun Chandrashekhar
 *              Manju R
 *              Rasheed
 *              Shakeel Bukhari
 */


#include <sys/types.h>
#include <sys/file.h>
#include <sys/atomic.h>
#include <sys/scsi/scsi.h>
#include <sys/byteorder.h>
#include "ld_pd_map.h"
#include "mr_sas.h"
#include "fusion.h"

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


/* Pre-TB command size and TB command size. */
#define MR_COMMAND_SIZE (64*20) /* 1280 bytes */
MR_LD_RAID *MR_LdRaidGet(U32 ld, MR_FW_RAID_MAP_ALL *map);
U16 MR_TargetIdToLdGet(U32 ldTgtId, MR_FW_RAID_MAP_ALL *map);
U16 MR_GetLDTgtId(U32 ld, MR_FW_RAID_MAP_ALL *map);
U16 get_updated_dev_handle(PLD_LOAD_BALANCE_INFO, struct IO_REQUEST_INFO *);
extern ddi_dma_attr_t mrsas_generic_dma_attr;
extern uint32_t mrsas_tbolt_max_cap_maxxfer;
extern struct ddi_device_acc_attr endian_attr;
extern int      debug_level_g;
extern unsigned int     enable_fp;
volatile int dump_io_wait_time = 90;
extern void
io_timeout_checker(void *arg);
extern volatile int  debug_timeout_g;
extern int      mrsas_issue_pending_cmds(struct mrsas_instance *);
extern int mrsas_complete_pending_cmds(struct mrsas_instance *instance);
extern void     push_pending_mfi_pkt(struct mrsas_instance *,
                        struct mrsas_cmd *);
extern U8 MR_BuildRaidContext(struct mrsas_instance *, struct IO_REQUEST_INFO *,
            MPI2_SCSI_IO_VENDOR_UNIQUE *, MR_FW_RAID_MAP_ALL *);

/* Local static prototypes. */
static struct mrsas_cmd *mrsas_tbolt_build_cmd(struct mrsas_instance *,
    struct scsi_address *, struct scsi_pkt *, uchar_t *);
```



```
static void mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len_ptr,
    U64 start_blk, U32 num_blocks);
static int mrsas_tbolt_check_map_info(struct mrsas_instance *);
static int mrsas_tbolt_sync_map_info(struct mrsas_instance *);
static int mrsas_tbolt_prepare_pkt(struct scsa_cmd *);
static int mrsas_tbolt_ioc_init(struct mrsas_instance *, dma_obj_t *);
#ifdef PDSUPPORT
static void mrsas_tbolt_get_pd_info(struct mrsas_instance *,
    struct mrsas_tbolt_pd_info *, int);
#endif /* PDSUPPORT */

static int debug_tbolt_fw_faults_after_ocr_g = 0;

/*
 * destroy_mfi_mpi_frame_pool
 */
void
destroy_mfi_mpi_frame_pool(struct mrsas_instance *instance)
{
        int     i;

        struct mrsas_cmd        *cmd;

        /* return all mfi frames to pool */
        for (i = 0; i < MRSAS_APP_RESERVED_CMDS; i++) {
                cmd = instance->cmd_list[i];
                if (cmd->frame_dma_obj_status == DMA_OBJ_ALLOCATED) {
                        (void) mrsas_free_dma_obj(instance,
                            cmd->frame_dma_obj);
                }
                cmd->frame_dma_obj_status = DMA_OBJ_FREED;
        }
}

/*
 * destroy_mpi2_frame_pool
 */
void
destroy_mpi2_frame_pool(struct mrsas_instance *instance)
{

        if (instance->mpi2_frame_pool_dma_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->mpi2_frame_pool_dma_obj);
                instance->mpi2_frame_pool_dma_obj.status |= DMA_OBJ_FREED;
        }
}


/*
 * mrsas_tbolt_free_additional_dma_buffer
 */
void
mrsas_tbolt_free_additional_dma_buffer(struct mrsas_instance *instance)
{
        int i;

        if (instance->mfi_internal_dma_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->mfi_internal_dma_obj);
                instance->mfi_internal_dma_obj.status = DMA_OBJ_FREED;
        }
        if (instance->mfi_evt_detail_obj.status == DMA_OBJ_ALLOCATED) {
                (void) mrsas_free_dma_obj(instance,
                    instance->mfi_evt_detail_obj);
                instance->mfi_evt_detail_obj.status = DMA_OBJ_FREED;
```

```
128         }

130         for (i = 0; i < 2; i++) {
131                 if (instance->ld_map_obj[i].status == DMA_OBJ_ALLOCATED) {
132                         (void) mrsas_free_dma_obj(instance,
133                                 instance->ld_map_obj[i]);
134                         instance->ld_map_obj[i].status = DMA_OBJ_FREED;
135                 }
136         }
137 }


140 /*
141  * free_req_desc_pool
142  */
143 void
144 free_req_rep_desc_pool(struct mrsas_instance *instance)
145 {
146         if (instance->request_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
147                 (void) mrsas_free_dma_obj(instance,
148                     instance->request_desc_dma_obj);
149                 instance->request_desc_dma_obj.status = DMA_OBJ_FREED;
150         }

152         if (instance->reply_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
153                 (void) mrsas_free_dma_obj(instance,
154                     instance->reply_desc_dma_obj);
155                 instance->reply_desc_dma_obj.status = DMA_OBJ_FREED;
156         }


159 }


162 /*
163  * ThunderBolt(TB) Request Message Frame Pool
164  */
165 int
166 create_mpi2_frame_pool(struct mrsas_instance *instance)
167 {
168         int             i = 0;
169         uint16_t        max_cmd;
170         uint32_t        sgl_sz;
171         uint32_t        raid_msg_size;
172         uint32_t        total_size;
173         uint32_t        offset;
174         uint32_t        io_req_base_phys;
175         uint8_t         *io_req_base;
176         struct mrsas_cmd        *cmd;

178         max_cmd = instance->max_fw_cmds;

180         sgl_sz          = 1024;
181         raid_msg_size   = MRSAS_THUNDERBOLT_MSG_SIZE;

183         /* Allocating additional 256 bytes to accomodate SMID 0. */
184         total_size = MRSAS_THUNDERBOLT_MSG_SIZE + (max_cmd * raid_msg_size) +
185             (max_cmd * sgl_sz) + (max_cmd * SENSE_LENGTH);

187         con_log(CL_ANN1, (CE_NOTE, "create_mpi2_frame_pool: "
188             "max_cmd %x", max_cmd));

190         con_log(CL_DLEVEL3, (CE_NOTE, "create_mpi2_frame_pool: "
191             "request message frame pool size %x", total_size));

193         /*
```



```
194          * ThunderBolt(TB) We need to create a single chunk of DMA'ble memory
195          * and then split the memory to 1024 commands. Each command should be
196          * able to contain a RAID MESSAGE FRAME which will embed a MFI_FRAME
197          * within it. Further refer the "alloc_req_rep_desc" function where
198          * we allocate request/reply descriptors queues for a clue.
199          */

201         instance->mpi2_frame_pool_dma_obj.size = total_size;
202         instance->mpi2_frame_pool_dma_obj.dma_attr = mrsas_generic_dma_attr;
203         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_addr_hi =
204             0xFFFFFFFFU;
205         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_count_max =
206             0xFFFFFFFFU;
207         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_sgllen = 1;
208         instance->mpi2_frame_pool_dma_obj.dma_attr.dma_attr_align = 256;

210         if (mrsas_alloc_dma_obj(instance, &instance->mpi2_frame_pool_dma_obj,
211             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
212                 cmn_err(CE_WARN,
213                     "mr_sas: could not alloc mpi2 frame pool");
214                 return (DDI_FAILURE);
215         }

217         bzero(instance->mpi2_frame_pool_dma_obj.buffer, total_size);
218         instance->mpi2_frame_pool_dma_obj.status |= DMA_OBJ_ALLOCATED;

220         instance->io_request_frames =
221             (uint8_t *)instance->mpi2_frame_pool_dma_obj.buffer;
222         instance->io_request_frames_phy =
223             (uint32_t)
224             instance->mpi2_frame_pool_dma_obj.dma_cookie[0].dmac_address;

226         con_log(CL_DLEVEL3, (CE_NOTE, "io_request_frames 0x%p",
227             (void *)instance->io_request_frames));

229         con_log(CL_DLEVEL3, (CE_NOTE, "io_request_frames_phy 0x%x",
230             instance->io_request_frames_phy));

232         io_req_base = (uint8_t *)instance->io_request_frames +
233             MRSAS_THUNDERBOLT_MSG_SIZE;
234         io_req_base_phys = instance->io_request_frames_phy +
235             MRSAS_THUNDERBOLT_MSG_SIZE;

237         con_log(CL_DLEVEL3, (CE_NOTE,
238             "io req_base_phys 0x%x", io_req_base_phys));

240         for (i = 0; i < max_cmd; i++) {
241                 cmd = instance->cmd_list[i];

243                 offset = i * MRSAS_THUNDERBOLT_MSG_SIZE;

245                 cmd->scsi_io_request = (Mpi2RaidSCSIIORequest_t *)
246                     ((uint8_t *)io_req_base + offset);
247                 cmd->scsi_io_request_phys_addr = io_req_base_phys + offset;

249                 cmd->sgl = (Mpi2SGEIOUnion_t *)((uint8_t *)io_req_base +
250                     (max_cmd * raid_msg_size) + i * sgl_sz);

252                 cmd->sgl_phys_addr = (io_req_base_phys +
253                     (max_cmd * raid_msg_size) + i * sgl_sz);

255                 cmd->sense1 = (uint8_t *)((uint8_t *)io_req_base +
256                     (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
257                     (i * SENSE_LENGTH));

259                 cmd->sense_phys_addr1 = (io_req_base_phys +
```

```
260                     (max_cmd * raid_msg_size) + (max_cmd * sgl_sz) +
261                     (i * SENSE_LENGTH));


264                 cmd->SMID = i + 1;

266                 con_log(CL_DLEVEL3, (CE_NOTE, "Frame Pool Addr [%x]0x%p",
267                     cmd->index, (void *)cmd->scsi_io_request));

269                 con_log(CL_DLEVEL3, (CE_NOTE, "Frame Pool Phys Addr [%x]0x%x",
270                     cmd->index, cmd->scsi_io_request_phys_addr));

272                 con_log(CL_DLEVEL3, (CE_NOTE, "Sense Addr [%x]0x%p",
273                     cmd->index, (void *)cmd->sense1));

275                 con_log(CL_DLEVEL3, (CE_NOTE, "Sense Addr Phys [%x]0x%x",
276                     cmd->index, cmd->sense_phys_addr1));

278                 con_log(CL_DLEVEL3, (CE_NOTE, "Sgl bufffers [%x]0x%p",
279                     cmd->index, (void *)cmd->sgl));

281                 con_log(CL_DLEVEL3, (CE_NOTE, "Sgl bufffers phys [%x]0x%x",
282                     cmd->index, cmd->sgl_phys_addr));
283         }

285         return (DDI_SUCCESS);

287 }


290 /*
291  * alloc_additional_dma_buffer for AEN
292  */
293 int
294 mrsas_tbolt_alloc_additional_dma_buffer(struct mrsas_instance *instance)
295 {
296         uint32_t        internal_buf_size = PAGESIZE*2;
297         int i;

299         /* Initialize buffer status as free */
300         instance->mfi_internal_dma_obj.status = DMA_OBJ_FREED;
301         instance->mfi_evt_detail_obj.status = DMA_OBJ_FREED;
302         instance->ld_map_obj[0].status = DMA_OBJ_FREED;
303         instance->ld_map_obj[1].status = DMA_OBJ_FREED;


306         instance->mfi_internal_dma_obj.size = internal_buf_size;
307         instance->mfi_internal_dma_obj.dma_attr = mrsas_generic_dma_attr;
308         instance->mfi_internal_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
309         instance->mfi_internal_dma_obj.dma_attr.dma_attr_count_max =
310             0xFFFFFFFFU;
311         instance->mfi_internal_dma_obj.dma_attr.dma_attr_sgllen = 1;

313         if (mrsas_alloc_dma_obj(instance, &instance->mfi_internal_dma_obj,
314             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
315                 cmn_err(CE_WARN,
316                     "mr_sas: could not alloc reply queue");
317                 return (DDI_FAILURE);
318         }

320         bzero(instance->mfi_internal_dma_obj.buffer, internal_buf_size);

322         instance->mfi_internal_dma_obj.status |= DMA_OBJ_ALLOCATED;
323         instance->internal_buf =
324             (caddr_t)(((unsigned long)instance->mfi_internal_dma_obj.buffer));
325         instance->internal_buf_dmac_add =
```



```
326             instance->mfi_internal_dma_obj.dma_cookie[0].dmac_address;
327         instance->internal_buf_size = internal_buf_size;

329         /* allocate evt_detail */
330         instance->mfi_evt_detail_obj.size = sizeof (struct mrsas_evt_detail);
331         instance->mfi_evt_detail_obj.dma_attr = mrsas_generic_dma_attr;
332         instance->mfi_evt_detail_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
333         instance->mfi_evt_detail_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
334         instance->mfi_evt_detail_obj.dma_attr.dma_attr_sgllen = 1;
335         instance->mfi_evt_detail_obj.dma_attr.dma_attr_align = 8;

337         if (mrsas_alloc_dma_obj(instance, &instance->mfi_evt_detail_obj,
338             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
339                 cmn_err(CE_WARN, "mrsas_tbolt_alloc_additional_dma_buffer: "
340                     "could not allocate data transfer buffer.");
341                 goto fail_tbolt_additional_buff;
342         }

344         bzero(instance->mfi_evt_detail_obj.buffer,
345             sizeof (struct mrsas_evt_detail));

347         instance->mfi_evt_detail_obj.status |= DMA_OBJ_ALLOCATED;

349         instance->size_map_info = sizeof (MR_FW_RAID_MAP) +
350             (sizeof (MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));

352         for (i = 0; i < 2; i++) {
353                 /* allocate the data transfer buffer */
354                 instance->ld_map_obj[i].size = instance->size_map_info;
355                 instance->ld_map_obj[i].dma_attr = mrsas_generic_dma_attr;
356                 instance->ld_map_obj[i].dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
357                 instance->ld_map_obj[i].dma_attr.dma_attr_count_max =
358                     0xFFFFFFFFU;
359                 instance->ld_map_obj[i].dma_attr.dma_attr_sgllen = 1;
360                 instance->ld_map_obj[i].dma_attr.dma_attr_align = 1;

362                 if (mrsas_alloc_dma_obj(instance, &instance->ld_map_obj[i],
363                     (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
364                         cmn_err(CE_WARN,
365                             "could not allocate data transfer buffer.");
366                         goto fail_tbolt_additional_buff;
367                 }

369                 instance->ld_map_obj[i].status |= DMA_OBJ_ALLOCATED;

371                 bzero(instance->ld_map_obj[i].buffer, instance->size_map_info);

373                 instance->ld_map[i] =
374                     (MR_FW_RAID_MAP_ALL *)instance->ld_map_obj[i].buffer;
375                 instance->ld_map_phy[i] = (uint32_t)instance->
376                     ld_map_obj[i].dma_cookie[0].dmac_address;

378                 con_log(CL_DLEVEL3, (CE_NOTE,
379                     "ld_map Addr Phys 0x%x", instance->ld_map_phy[i]));

381                 con_log(CL_DLEVEL3, (CE_NOTE,
382                     "size_map_info 0x%x", instance->size_map_info));
383         }

385         return (DDI_SUCCESS);

387 fail_tbolt_additional_buff:
388         mrsas_tbolt_free_additional_dma_buffer(instance);

390         return (DDI_FAILURE);
391 }
```

```
393 MRSAS_REQUEST_DESCRIPTOR_UNION *
394 mr_sas_get_request_descriptor(struct mrsas_instance *instance, uint16_t index)
395 {
396         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc;
397
398         if (index > instance->max_fw_cmds) {
399                 con_log(CL_ANN1, (CE_NOTE,
400                     "Invalid SMID 0x%x request for descriptor", index));
401                 con_log(CL_ANN1, (CE_NOTE,
402                     "max_fw_cmds : 0x%x", instance->max_fw_cmds));
403                 return (NULL);
404         }
405
406         req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
407             ((char *)instance->request_message_pool +
408             (sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION) * index));
409
410         con_log(CL_ANN1, (CE_NOTE,
411             "request descriptor : 0x%08lx", (unsigned long)req_desc));
412
413         con_log(CL_ANN1, (CE_NOTE,
414             "request descriptor base phy : 0x%08lx",
415             (unsigned long)instance->request_message_pool_phy));
416
417         return ((MRSAS_REQUEST_DESCRIPTOR_UNION *)req_desc);
418 }
419
420
421 /*
422  * Allocate Request and Reply  Queue Descriptors.
423  */
424 int
425 alloc_req_rep_desc(struct mrsas_instance *instance)
426 {
427         uint32_t        request_q_sz, reply_q_sz;
428         int             i, max_reply_q_sz;
429         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
430
431         /*
432          * ThunderBolt(TB) There's no longer producer consumer mechanism.
433          * Once we have an interrupt we are supposed to scan through the list of
434          * reply descriptors and process them accordingly. We would be needing
435          * to allocate memory for 1024 reply descriptors
436          */
437
438         /* Allocate Reply Descriptors */
439         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x",
440             (uint_t)sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));
441
442         /* reply queue size should be multiple of 16 */
443         max_reply_q_sz = ((instance->max_fw_cmds + 1 + 15)/16)*16;
444
445         reply_q_sz = 8 * max_reply_q_sz;
446
447
448         con_log(CL_ANN1, (CE_NOTE, " reply q desc len = %x",
449             (uint_t)sizeof (MPI2_REPLY_DESCRIPTORS_UNION)));
450
451         instance->reply_desc_dma_obj.size = reply_q_sz;
452         instance->reply_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
453         instance->reply_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
454         instance->reply_desc_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
455         instance->reply_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
456         instance->reply_desc_dma_obj.dma_attr.dma_attr_align = 16;
```



```
458         if (mrsas_alloc_dma_obj(instance, &instance->reply_desc_dma_obj,
459             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
460                 cmn_err(CE_WARN,
461                     "mr_sas: could not alloc reply queue");
462                 return (DDI_FAILURE);
463         }
464
465         bzero(instance->reply_desc_dma_obj.buffer, reply_q_sz);
466         instance->reply_desc_dma_obj.status |= DMA_OBJ_ALLOCATED;
467
468         /* virtual address of  reply queue */
469         instance->reply_frame_pool = (MPI2_REPLY_DESCRIPTORS_UNION *)(
470             instance->reply_desc_dma_obj.buffer);
471
472         instance->reply_q_depth = max_reply_q_sz;
473
474         con_log(CL_ANN1, (CE_NOTE, "[reply queue depth]0x%x",
475             instance->reply_q_depth));
476
477         con_log(CL_ANN1, (CE_NOTE, "[reply queue virt addr]0x%p",
478             (void *)instance->reply_frame_pool));
479
480         /* initializing reply address to 0xFFFFFFFF */
481         reply_desc = instance->reply_frame_pool;
482
483         for (i = 0; i < instance->reply_q_depth; i++) {
484                 reply_desc->Words = (uint64_t)~0;
485                 reply_desc++;
486         }
487
488
489         instance->reply_frame_pool_phy =
490             (uint32_t)instance->reply_desc_dma_obj.dma_cookie[0].dmac_address;
491
492         con_log(CL_ANN1, (CE_NOTE,
493             "[reply queue phys addr]0x%x", instance->reply_frame_pool_phy));
494
495
496         instance->reply_pool_limit_phy = (instance->reply_frame_pool_phy +
497             reply_q_sz);
498
499         con_log(CL_ANN1, (CE_NOTE, "[reply pool limit phys addr]0x%x",
500             instance->reply_pool_limit_phy));
501
502
503         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x",
504             (int)sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));
505
506         /* Allocate Request Descriptors */
507         con_log(CL_ANN1, (CE_NOTE, " request q desc len = %x",
508             (int)sizeof (MRSAS_REQUEST_DESCRIPTOR_UNION)));
509
510         request_q_sz = 8 *
511             (instance->max_fw_cmds);
512
513         instance->request_desc_dma_obj.size = request_q_sz;
514         instance->request_desc_dma_obj.dma_attr = mrsas_generic_dma_attr;
515         instance->request_desc_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
516         instance->request_desc_dma_obj.dma_attr.dma_attr_count_max =
517             0xFFFFFFFFU;
518         instance->request_desc_dma_obj.dma_attr.dma_attr_sgllen = 1;
519         instance->request_desc_dma_obj.dma_attr.dma_attr_align = 16;
520
521         if (mrsas_alloc_dma_obj(instance, &instance->request_desc_dma_obj,
522             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
523                 cmn_err(CE_WARN,
```

```
524                     "mr_sas: could not alloc request queue desc");
525                 goto fail_undo_reply_queue;
526         }

528         bzero(instance->request_desc_dma_obj.buffer, request_q_sz);
529         instance->request_desc_dma_obj.status |= DMA_OBJ_ALLOCATED;

531         /* virtual address of  request queue desc */
532         instance->request_message_pool = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
533             (instance->request_desc_dma_obj.buffer);

535         instance->request_message_pool_phy =
536             (uint32_t)instance->request_desc_dma_obj.dma_cookie[0].dmac_address;

538         return (DDI_SUCCESS);

540 fail_undo_reply_queue:
541         if (instance->reply_desc_dma_obj.status == DMA_OBJ_ALLOCATED) {
542                 (void) mrsas_free_dma_obj(instance,
543                     instance->reply_desc_dma_obj);
544                 instance->reply_desc_dma_obj.status = DMA_OBJ_FREED;
545         }

547         return (DDI_FAILURE);
548 }

550 /*
551  * mrsas_alloc_cmd_pool_tbolt
552  *
553  * TODO: merge tbolt-specific code into mrsas_alloc_cmd_pool() to have single
554  * routine
555  */
556 int
557 mrsas_alloc_cmd_pool_tbolt(struct mrsas_instance *instance)
558 {
559         int             i;
560         int             count;
561         uint32_t        max_cmd;
562         uint32_t        reserve_cmd;
563         size_t          sz;

565         struct mrsas_cmd        *cmd;

567         max_cmd = instance->max_fw_cmds;
568         con_log(CL_ANN1, (CE_NOTE, "mrsas_alloc_cmd_pool: "
569             "max_cmd %x", max_cmd));


572         sz = sizeof (struct mrsas_cmd *) * max_cmd;

574         /*
575          * instance->cmd_list is an array of struct mrsas_cmd pointers.
576          * Allocate the dynamic array first and then allocate individual
577          * commands.
578          */
579         instance->cmd_list = kmem_zalloc(sz, KM_SLEEP);

581         /* create a frame pool and assign one frame to each cmd */
582         for (count = 0; count < max_cmd; count++) {
583                 instance->cmd_list[count] =
584                     kmem_zalloc(sizeof (struct mrsas_cmd), KM_SLEEP);
585         }

587         /* add all the commands to command pool */

589         INIT_LIST_HEAD(&instance->cmd_pool_list);
```



```
590         INIT_LIST_HEAD(&instance->cmd_pend_list);
591         INIT_LIST_HEAD(&instance->cmd_app_pool_list);

593         reserve_cmd = MRSAS_APP_RESERVED_CMDS;

595         /* cmd index 0 reservered for IOC INIT */
596         for (i = 1; i < reserve_cmd; i++) {
597                 cmd             = instance->cmd_list[i];
598                 cmd->index      = i;
599                 mlist_add_tail(&cmd->list, &instance->cmd_app_pool_list);
600         }


603         for (i = reserve_cmd; i < max_cmd; i++) {
604                 cmd             = instance->cmd_list[i];
605                 cmd->index      = i;
606                 mlist_add_tail(&cmd->list, &instance->cmd_pool_list);
607         }

609         return (DDI_SUCCESS);

611 mrsas_undo_cmds:
612         if (count > 0) {
613                 /* free each cmd */
614                 for (i = 0; i < count; i++) {
615                         if (instance->cmd_list[i] != NULL) {
616                                 kmem_free(instance->cmd_list[i],
617                                     sizeof (struct mrsas_cmd));
618                         }
619                         instance->cmd_list[i] = NULL;
620                 }
621         }

623 mrsas_undo_cmd_list:
624         if (instance->cmd_list != NULL)
625                 kmem_free(instance->cmd_list, sz);
626         instance->cmd_list = NULL;

628         return (DDI_FAILURE);
629 }


632 /*
633  * free_space_for_mpi2
634  */
635 void
636 free_space_for_mpi2(struct mrsas_instance *instance)
637 {
638         /* already freed */
639         if (instance->cmd_list == NULL) {
640                 return;
641         }

643         /* First free the additional DMA buffer */
644         mrsas_tbolt_free_additional_dma_buffer(instance);

646         /* Free the request/reply descriptor pool */
647         free_req_rep_desc_pool(instance);

649         /*  Free the MPI message pool */
650         destroy_mpi2_frame_pool(instance);

652         /* Free the MFI frame pool */
653         destroy_mfi_frame_pool(instance);

655         /* Free all the commands in the cmd_list */
```

```
656         /* Free the cmd_list buffer itself */
657         mrsas_free_cmd_pool(instance);
658 }


661 /*
662  * ThunderBolt(TB) memory allocations for commands/messages/frames.
663  */
664 int
665 alloc_space_for_mpi2(struct mrsas_instance *instance)
666 {
667         /* Allocate command pool (memory for cmd_list & individual commands) */
668         if (mrsas_alloc_cmd_pool_tbolt(instance)) {
669                 cmn_err(CE_WARN, "Error creating cmd pool");
670                 return (DDI_FAILURE);
671         }

673         /* Initialize single reply size and Message size */
674         instance->reply_size = MRSAS_THUNDERBOLT_REPLY_SIZE;
675         instance->raid_io_msg_size = MRSAS_THUNDERBOLT_MSG_SIZE;

677         instance->max_sge_in_main_msg = (MRSAS_THUNDERBOLT_MSG_SIZE -
678             (sizeof (MPI2_RAID_SCSI_IO_REQUEST) -
679             sizeof (MPI2_SGE_IO_UNION)))/ sizeof (MPI2_SGE_IO_UNION);
680         instance->max_sge_in_chain = (MR_COMMAND_SIZE -
681             MRSAS_THUNDERBOLT_MSG_SIZE) / sizeof (MPI2_SGE_IO_UNION);

683         /* Reduce SG count by 1 to take care of group cmds feature in FW */
684         instance->max_num_sge = (instance->max_sge_in_main_msg +
685             instance->max_sge_in_chain - 2);
686         instance->chain_offset_mpt_msg =
687             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16;
688         instance->chain_offset_io_req = (MRSAS_THUNDERBOLT_MSG_SIZE -
689             sizeof (MPI2_SGE_IO_UNION)) / 16;
690         instance->reply_read_index = 0;


693         /* Allocate Request and Reply descriptors Array */
694         /* Make sure the buffer is aligned to 8 for req/rep  descriptor Pool */
695         if (alloc_req_rep_desc(instance)) {
696                 cmn_err(CE_WARN,
697                     "Error, allocating memory for descripter-pool");
698                 goto mpi2_undo_cmd_pool;
699         }
700         con_log(CL_ANN1, (CE_NOTE, "[request message pool phys addr]0x%x",
701             instance->request_message_pool_phy));


704         /* Allocate MFI Frame pool - for MPI-MFI passthru commands */
705         if (create_mfi_frame_pool(instance)) {
706                 cmn_err(CE_WARN,
707                     "Error, allocating memory for MFI frame-pool");
708                 goto mpi2_undo_descripter_pool;
709         }


712         /* Allocate MPI2 Message pool */
713         /*
714          * Make sure the buffer is alligned to 256 for raid message packet
715          * create a io request pool and assign one frame to each cmd
716          */

718         if (create_mpi2_frame_pool(instance)) {
719                 cmn_err(CE_WARN,
720                     "Error, allocating memory for MPI2 Message-pool");
721                 goto mpi2_undo_mfi_frame_pool;
```



```
722         }

724 #ifdef DEBUG
725         con_log(CL_ANN1, (CE_CONT, "[max_sge_in_main_msg]0x%x",
726             instance->max_sge_in_main_msg));
727         con_log(CL_ANN1, (CE_CONT, "[max_sge_in_chain]0x%x",
728             instance->max_sge_in_chain));
729         con_log(CL_ANN1, (CE_CONT,
730             "[max_sge]0x%x", instance->max_num_sge));
731         con_log(CL_ANN1, (CE_CONT, "[chain_offset_mpt_msg]0x%x",
732             instance->chain_offset_mpt_msg));
733         con_log(CL_ANN1, (CE_CONT, "[chain_offset_io_req]0x%x",
734             instance->chain_offset_io_req));
735 #endif


738         /* Allocate additional dma buffer */
739         if (mrsas_tbolt_alloc_additional_dma_buffer(instance)) {
740                 cmn_err(CE_WARN,
741                     "Error, allocating tbolt additional DMA buffer");
742                 goto mpi2_undo_message_pool;
743         }

745         return (DDI_SUCCESS);

747 mpi2_undo_message_pool:
748         destroy_mpi2_frame_pool(instance);

750 mpi2_undo_mfi_frame_pool:
751         destroy_mfi_frame_pool(instance);

753 mpi2_undo_descripter_pool:
754         free_req_rep_desc_pool(instance);

756 mpi2_undo_cmd_pool:
757         mrsas_free_cmd_pool(instance);

759         return (DDI_FAILURE);
760 }


763 /*
764  * mrsas_init_adapter_tbolt - Initialize fusion interface adapter.
765  */
766 int
767 mrsas_init_adapter_tbolt(struct mrsas_instance *instance)
768 {

770         /*
771          * Reduce the max supported cmds by 1. This is to ensure that the
772          * reply_q_sz (1 more than the max cmd that driver may send)
773          * does not exceed max cmds that the FW can support
774          */

776         if (instance->max_fw_cmds > 1008) {
777                 instance->max_fw_cmds = 1008;
778                 instance->max_fw_cmds = instance->max_fw_cmds-1;
779         }

781         con_log(CL_ANN, (CE_NOTE, "mrsas_init_adapter_tbolt: "
782             " instance->max_fw_cmds 0x%X.", instance->max_fw_cmds));


785         /* create a pool of commands */
786         if (alloc_space_for_mpi2(instance) != DDI_SUCCESS) {
787                 cmn_err(CE_WARN,
```

```
788                     " alloc_space_for_mpi2() failed.");
789
790                 return (DDI_FAILURE);
791         }
792
793         /* Send ioc init message */
794         /* NOTE: the issue_init call does FMA checking already. */
795         if (mrsas_issue_init_mpi2(instance) != DDI_SUCCESS) {
796                 cmn_err(CE_WARN,
797                     " mrsas_issue_init_mpi2() failed.");
798
799                 goto fail_init_fusion;
800         }
801
802         instance->unroll.alloc_space_mpi2 = 1;
803
804         con_log(CL_ANN, (CE_NOTE,
805             "mrsas_init_adapter_tbolt: SUCCESSFUL"));
806
807         return (DDI_SUCCESS);
808
809 fail_init_fusion:
810         free_space_for_mpi2(instance);
811
812         return (DDI_FAILURE);
813 }
814
815
816
817 /*
818  * init_mpi2
819  */
820 int
821 mrsas_issue_init_mpi2(struct mrsas_instance *instance)
822 {
823         dma_obj_t init2_dma_obj;
824         int ret_val = DDI_SUCCESS;
825
826         /* allocate DMA buffer for IOC INIT message */
827         init2_dma_obj.size = sizeof (Mpi2IOCInitRequest_t);
828         init2_dma_obj.dma_attr = mrsas_generic_dma_attr;
829         init2_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
830         init2_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
831         init2_dma_obj.dma_attr.dma_attr_sgllen = 1;
832         init2_dma_obj.dma_attr.dma_attr_align = 256;
833
834         if (mrsas_alloc_dma_obj(instance, &init2_dma_obj,
835             (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
836                 cmn_err(CE_WARN, "mr_sas_issue_init_mpi2 "
837                     "could not allocate data transfer buffer.");
838                 return (DDI_FAILURE);
839         }
840         (void) memset(init2_dma_obj.buffer, 2, sizeof (Mpi2IOCInitRequest_t));
841
842         con_log(CL_ANN1, (CE_NOTE,
843             "mrsas_issue_init_mpi2 _phys adr: %x",
844             init2_dma_obj.dma_cookie[0].dmac_address));
845
846
847         /* Initialize and send ioc init message */
848         ret_val = mrsas_tbolt_ioc_init(instance, &init2_dma_obj);
849         if (ret_val == DDI_FAILURE) {
850                 con_log(CL_ANN1, (CE_WARN,
851                     "mrsas_issue_init_mpi2: Failed"));
852                 goto fail_init_mpi2;
853         }
```



```
854
855         /* free IOC init DMA buffer */
856         if (mrsas_free_dma_obj(instance, init2_dma_obj)
857             != DDI_SUCCESS) {
858                 con_log(CL_ANN1, (CE_WARN,
859                     "mrsas_issue_init_mpi2: Free Failed"));
860                 return (DDI_FAILURE);
861         }
862
863         /* Get/Check and sync ld_map info */
864         instance->map_id = 0;
865         if (mrsas_tbolt_check_map_info(instance) == DDI_SUCCESS)
866                 (void) mrsas_tbolt_sync_map_info(instance);
867
868
869         /* No mrsas_cmd to send, so send NULL. */
870         if (mrsas_common_check(instance, NULL) != DDI_SUCCESS)
871                 goto fail_init_mpi2;
872
873         con_log(CL_ANN, (CE_NOTE,
874             "mrsas_issue_init_mpi2: SUCCESSFUL"));
875
876         return (DDI_SUCCESS);
877
878 fail_init_mpi2:
879         (void) mrsas_free_dma_obj(instance, init2_dma_obj);
880
881         return (DDI_FAILURE);
882 }
883
884 static int
885 mrsas_tbolt_ioc_init(struct mrsas_instance *instance, dma_obj_t *mpi2_dma_obj)
886 {
887         int                             numbytes;
888         uint16_t                        flags;
889         struct mrsas_init_frame2        *mfiFrameInit2;
890         struct mrsas_header             *frame_hdr;
891         Mpi2IOCInitRequest_t            *init;
892         struct mrsas_cmd                *cmd = NULL;
893         struct mrsas_drv_ver            drv_ver_info;
894         MRSAS_REQUEST_DESCRIPTOR_UNION  *req_desc;
895
896         con_log(CL_ANN, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
897
898
899 #ifdef DEBUG
900         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
901             (int)sizeof (*mfiFrameInit2)));
902         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n", (int)sizeof (*init)));
903         con_log(CL_ANN1, (CE_CONT, " mfiFrameInit2 len = %x\n",
904             (int)sizeof (struct mrsas_init_frame2)));
905         con_log(CL_ANN1, (CE_CONT, " MPI len = %x\n",
906             (int)sizeof (Mpi2IOCInitRequest_t)));
907 #endif
908
909         init = (Mpi2IOCInitRequest_t *)mpi2_dma_obj->buffer;
910         numbytes = sizeof (*init);
911         bzero(init, numbytes);
912
913         ddi_put8(mpi2_dma_obj->acc_handle, &init->Function,
914             MPI2_FUNCTION_IOC_INIT);
915
916         ddi_put8(mpi2_dma_obj->acc_handle, &init->WhoInit,
917             MPI2_WHOINIT_HOST_DRIVER);
918
919         /* set MsgVersion and HeaderVersion host driver was built with */
```

```
920         ddi_put16(mpi2_dma_obj->acc_handle, &init->MsgVersion,
921             MPI2_VERSION);
922
923         ddi_put16(mpi2_dma_obj->acc_handle, &init->HeaderVersion,
924             MPI2_HEADER_VERSION);
925
926         ddi_put16(mpi2_dma_obj->acc_handle, &init->SystemRequestFrameSize,
927             instance->raid_io_msg_size / 4);
928
929         ddi_put16(mpi2_dma_obj->acc_handle, &init->ReplyFreeQueueDepth,
930             0);
931
932         ddi_put16(mpi2_dma_obj->acc_handle,
933             &init->ReplyDescriptorPostQueueDepth,
934             instance->reply_q_depth);
935         /*
936          * These addresses are set using the DMA cookie addresses from when the
937          * memory was allocated.  Sense buffer hi address should be 0.
938          * ddi_put32(accessp, &init->SenseBufferAddressHigh, 0);
939          */
940
941         ddi_put32(mpi2_dma_obj->acc_handle,
942             &init->SenseBufferAddressHigh, 0);
943
944         ddi_put64(mpi2_dma_obj->acc_handle,
945             (uint64_t *)&init->SystemRequestFrameBaseAddress,
946             instance->io_request_frames_phy);
947
948         ddi_put64(mpi2_dma_obj->acc_handle,
949             &init->ReplyDescriptorPostQueueAddress,
950             instance->reply_frame_pool_phy);
951
952         ddi_put64(mpi2_dma_obj->acc_handle,
953             &init->ReplyFreeQueueAddress, 0);
954
955         cmd = instance->cmd_list[0];
956         if (cmd == NULL) {
957                 return (DDI_FAILURE);
958         }
959         cmd->retry_count_for_ocr = 0;
960         cmd->pkt = NULL;
961         cmd->drv_pkt_time = 0;
962
963         mfiFrameInit2 = (struct mrsas_init_frame2 *)cmd->scsi_io_request;
964         con_log(CL_ANN1, (CE_CONT, "[mfi vaddr]%p", (void *)mfiFrameInit2));
965
966         frame_hdr = &cmd->frame->hdr;
967
968         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
969             MFI_CMD_STATUS_POLL_MODE);
970
971         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
972
973         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;
974
975         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);
976
977         con_log(CL_ANN, (CE_CONT,
978             "mrsas_tbolt_ioc_init: SMID:%x\n", cmd->SMID));
979
980         /* Init the MFI Header */
981         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
982             &mfiFrameInit2->cmd, MFI_CMD_OP_INIT);
983
984         con_log(CL_ANN1, (CE_CONT, "[CMD]%x", mfiFrameInit2->cmd));
```



```
986         ddi_put8(instance->mpi2_frame_pool_dma_obj.acc_handle,
987             &mfiFrameInit2->cmd_status,
988             MFI_STAT_INVALID_STATUS);
989
990         con_log(CL_ANN1, (CE_CONT, "[Status]%x", mfiFrameInit2->cmd_status));
991
992         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
993             &mfiFrameInit2->queue_info_new_phys_addr_lo,
994             mpi2_dma_obj->dma_cookie[0].dmac_address);
995
996         ddi_put32(instance->mpi2_frame_pool_dma_obj.acc_handle,
997             &mfiFrameInit2->data_xfer_len,
998             sizeof (Mpi2IOCInitRequest_t));
999
1000        con_log(CL_ANN1, (CE_CONT, "[reply q desc addr]%x",
1001            (int)init->ReplyDescriptorPostQueueAddress));
1002
1003        /* fill driver version information */
1004        fill_up_drv_ver(&drv_ver_info);
1005
1006        /* allocate the driver version data transfer buffer */
1007        instance->drv_ver_dma_obj.size = sizeof (drv_ver_info.drv_ver);
1008        instance->drv_ver_dma_obj.dma_attr = mrsas_generic_dma_attr;
1009        instance->drv_ver_dma_obj.dma_attr.dma_attr_addr_hi = 0xFFFFFFFFU;
1010        instance->drv_ver_dma_obj.dma_attr.dma_attr_count_max = 0xFFFFFFFFU;
1011        instance->drv_ver_dma_obj.dma_attr.dma_attr_sgllen = 1;
1012        instance->drv_ver_dma_obj.dma_attr.dma_attr_align = 1;
1013
1014        if (mrsas_alloc_dma_obj(instance, &instance->drv_ver_dma_obj,
1015            (uchar_t)DDI_STRUCTURE_LE_ACC) != 1) {
1016                cmn_err(CE_WARN,
1017                    "fusion init: Could not allocate driver version buffer.");
1018                return (DDI_FAILURE);
1019        }
1020        /* copy driver version to dma buffer */
1021        bzero(instance->drv_ver_dma_obj.buffer, sizeof (drv_ver_info.drv_ver));
1022        ddi_rep_put8(cmd->frame_dma_obj.acc_handle,
1023            (uint8_t *)drv_ver_info.drv_ver,
1024            (uint8_t *)instance->drv_ver_dma_obj.buffer,
1025            sizeof (drv_ver_info.drv_ver), DDI_DEV_AUTOINCR);
1026
1027        /* send driver version physical address to firmware */
1028        ddi_put64(cmd->frame_dma_obj.acc_handle, &mfiFrameInit2->driverversion,
1029            instance->drv_ver_dma_obj.dma_cookie[0].dmac_address);
1030
1031        con_log(CL_ANN1, (CE_CONT, "[MPIINIT2 frame Phys addr ]0x%x len = %x",
1032            mfiFrameInit2->queue_info_new_phys_addr_lo,
1033            (int)sizeof (Mpi2IOCInitRequest_t)));
1034
1035        con_log(CL_ANN1, (CE_CONT, "[Length]%x", mfiFrameInit2->data_xfer_len));
1036
1037        con_log(CL_ANN1, (CE_CONT, "[MFI frame Phys Address]%x len = %x",
1038            cmd->scsi_io_request_phys_addr,
1039            (int)sizeof (struct mrsas_init_frame2)));
1040
1041        /* disable interrupts before sending INIT2 frame */
1042        instance->func_ptr->disable_intr(instance);
1043
1044        req_desc = (MRSAS_REQUEST_DESCRIPTOR_UNION *)
1045            instance->request_message_pool;
1046        req_desc->Words = cmd->scsi_io_request_phys_addr;
1047        req_desc->MFAIo.RequestFlags =
1048            (MPI2_REQ_DESCRIPT_FLAGS_MFA << MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1049
1050        cmd->request_desc = req_desc;
```

```
1052         /* issue the init frame */
1053         instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd);

1055         con_log(CL_ANN1, (CE_CONT, "[cmd = %d] ", frame_hdr->cmd));
1056         con_log(CL_ANN1, (CE_CONT, "[cmd  Status= %x] ",
1057             frame_hdr->cmd_status));

1059         if (ddi_get8(instance->mpi2_frame_pool_dma_obj.acc_handle,
1060             &mfiFrameInit2->cmd_status) == 0) {
1061                 con_log(CL_ANN, (CE_NOTE, "INIT2 Success"));
1062         } else {
1063                 con_log(CL_ANN, (CE_WARN, "INIT2 Fail"));
1064                 mrsas_dump_reply_desc(instance);
1065                 goto fail_ioc_init;
1066         }

1068         mrsas_dump_reply_desc(instance);

1070         instance->unroll.verBuff = 1;

1072         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_ioc_init: SUCCESSFUL"));

1074         return (DDI_SUCCESS);


1077 fail_ioc_init:

1079         (void) mrsas_free_dma_obj(instance, instance->drv_ver_dma_obj);

1081         return (DDI_FAILURE);
1082 }

1084 int
1085 wait_for_outstanding_poll_io(struct mrsas_instance *instance)
1086 {
1087         int i;
1088         uint32_t wait_time = dump_io_wait_time;
1089         for (i = 0; i < wait_time; i++) {
1090                 /*
1091                  * Check For Outstanding poll Commands
1092                  * except ldsync command and aen command
1093                  */
1094                 if (instance->fw_outstanding <= 2) {
1095                         break;
1096                 }
1097                 drv_usecwait(10*MILLISEC);
1098                 /* complete commands from reply queue */
1099                 (void) mr_sas_tbolt_process_outstanding_cmd(instance);
1100         }
1101         if (instance->fw_outstanding > 2) {
1102                 return (1);
1103         }
1104         return (0);
1105 }
1106 /*
1107  * scsi_pkt handling
1108  *
1109  * Visible to the external world via the transport structure.
1110  */

1112 int
1113 mrsas_tbolt_tran_start(struct scsi_address *ap, struct scsi_pkt *pkt)
1114 {
1115         struct mrsas_instance   *instance = ADDR2MR(ap);
1116         struct scsa_cmd         *acmd = PKT2CMD(pkt);
1117         struct mrsas_cmd        *cmd = NULL;
```



```
1118         uchar_t                 cmd_done = 0;

1120         con_log(CL_DLEVEL1, (CE_NOTE, "chkpnt:%s:%d", __func__, __LINE__));
1121         if (instance->deadadapter == 1) {
1122                 cmn_err(CE_WARN,
1123                     "mrsas_tran_start:TBOLT return TRAN_FATAL_ERROR "
1124                     "for IO, as the HBA doesnt take any more IOs");
1125                 if (pkt) {
1126                         pkt->pkt_reason         = CMD_DEV_GONE;
1127                         pkt->pkt_statistics     = STAT_DISCON;
1128                 }
1129                 return (TRAN_FATAL_ERROR);
1130         }
1131         if (instance->adapterresetinprogress) {
1132                 con_log(CL_ANN, (CE_NOTE, "Reset flag set, "
1133                     "returning mfi_pkt and setting TRAN_BUSY\n"));
1134                 return (TRAN_BUSY);
1135         }
1136         (void) mrsas_tbolt_prepare_pkt(acmd);

1138         cmd = mrsas_tbolt_build_cmd(instance, ap, pkt, &cmd_done);

1140         /*
1141          * Check if the command is already completed by the mrsas_build_cmd()
1142          * routine. In which case the busy_flag would be clear and scb will be
1143          * NULL and appropriate reason provided in pkt_reason field
1144          */
1145         if (cmd_done) {
1146                 pkt->pkt_reason = CMD_CMPLT;
1147                 pkt->pkt_scbp[0] = STATUS_GOOD;
1148                 pkt->pkt_state |= STATE_GOT_BUS | STATE_GOT_TARGET
1149                     | STATE_SENT_CMD;
1150                 if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp) {
1151                         (*pkt->pkt_comp)(pkt);
1152                 }

1154                 return (TRAN_ACCEPT);
1155         }

1157         if (cmd == NULL) {
1158                 return (TRAN_BUSY);
1159         }


1162         if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
1163                 if (instance->fw_outstanding > instance->max_fw_cmds) {
1164                         cmn_err(CE_WARN,
1165                             "Command Queue Full... Returning BUSY");
1166                         return_raid_msg_pkt(instance, cmd);
1167                         return (TRAN_BUSY);
1168                 }

1170                 /* Synchronize the Cmd frame for the controller */
1171                 (void) ddi_dma_sync(cmd->frame_dma_obj.dma_handle, 0, 0,
1172                     DDI_DMA_SYNC_FORDEV);

1174                 con_log(CL_ANN, (CE_CONT, "tbolt_issue_cmd: SCSI CDB[0]=0x%x "
1175                     "cmd->index:0x%x SMID 0x%x\n", pkt->pkt_cdbp[0],
1176                     cmd->index, cmd->SMID));

1178                 instance->func_ptr->issue_cmd(cmd, instance);
1179         } else {
1180                 instance->func_ptr->issue_cmd(cmd, instance);
1181                 (void) wait_for_outstanding_poll_io(instance);
1182                 (void) mrsas_common_check(instance, cmd);
1183         }
```

```
1185         return (TRAN_ACCEPT);
1186 }

1188 /*
1189  * prepare the pkt:
1190  * the pkt may have been resubmitted or just reused so
1191  * initialize some fields and do some checks.
1192  */
1193 static int
1194 mrsas_tbolt_prepare_pkt(struct scsa_cmd *acmd)
1195 {
1196         struct scsi_pkt *pkt = CMD2PKT(acmd);


1199         /*
1200          * Reinitialize some fields that need it; the packet may
1201          * have been resubmitted
1202          */
1203         pkt->pkt_reason = CMD_CMPLT;
1204         pkt->pkt_state = 0;
1205         pkt->pkt_statistics = 0;
1206         pkt->pkt_resid = 0;

1208         /*
1209          * zero status byte.
1210          */
1211         *(pkt->pkt_scbp) = 0;

1213         return (0);
1214 }


1217 int
1218 mr_sas_tbolt_build_sgl(struct mrsas_instance *instance,
1219     struct scsa_cmd *acmd,
1220     struct mrsas_cmd *cmd,
1221     Mpi2RaidSCSIIORequest_t *scsi_raid_io,
1222     uint32_t *datalen)
1223 {
1224         uint32_t                MaxSGEs;
1225         int                     sg_to_process;
1226         uint32_t                i, j;
1227         uint32_t                numElements, endElement;
1228         Mpi25IeeeSgeChain64_t   *ieeeChainElement = NULL;
1229         Mpi25IeeeSgeChain64_t   *scsi_raid_io_sgl_ieee = NULL;
1230         ddi_acc_handle_t acc_handle =
1231             instance->mpi2_frame_pool_dma_obj.acc_handle;

1233         con_log(CL_ANN1, (CE_NOTE,
1234             "chkpnt: Building Chained SGL :%d", __LINE__));

1236         /* Calulate SGE size in number of Words(32bit) */
1237         /* Clear the datalen before updating it. */
1238         *datalen = 0;

1240         MaxSGEs = instance->max_sge_in_main_msg;

1242         ddi_put16(acc_handle, &scsi_raid_io->SGLFlags,
1243             MPI2_SGE_FLAGS_64_BIT_ADDRESSING);

1245         /* set data transfer flag. */
1246         if (acmd->cmd_flags & CFLAG_DMASEND) {
1247                 ddi_put32(acc_handle, &scsi_raid_io->Control,
1248                     MPI2_SCSIIO_CONTROL_WRITE);
1249         } else {
```



```
1250                 ddi_put32(acc_handle, &scsi_raid_io->Control,
1251                     MPI2_SCSIIO_CONTROL_READ);
1252         }


1255         numElements = acmd->cmd_cookiecnt;

1257         con_log(CL_DLEVEL1, (CE_NOTE, "[SGE Count]:%x", numElements));

1259         if (numElements > instance->max_num_sge) {
1260                 con_log(CL_ANN, (CE_NOTE,
1261                     "[Max SGE Count Exceeded]:%x", numElements));
1262                 return (numElements);
1263         }

1265         ddi_put8(acc_handle, &scsi_raid_io->RaidContext.numSGE,
1266             (uint8_t)numElements);

1268         /* set end element in main message frame */
1269         endElement = (numElements <= MaxSGEs) ? numElements : (MaxSGEs - 1);

1271         /* prepare the scatter-gather list for the firmware */
1272         scsi_raid_io_sgl_ieee =
1273             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

1275         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1276                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = scsi_raid_io_sgl_ieee;
1277                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;

1279                 ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
1280         }

1282         for (i = 0; i < endElement; i++, scsi_raid_io_sgl_ieee++) {
1283                 ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
1284                     acmd->cmd_dmacookies[i].dmac_laddress);

1286                 ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length,
1287                     acmd->cmd_dmacookies[i].dmac_size);

1289                 ddi_put8(acc_handle, &scsi_raid_io_sgl_ieee->Flags, 0);

1291                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1292                         if (i == (numElements - 1)) {
1293                                 ddi_put8(acc_handle,
1294                                     &scsi_raid_io_sgl_ieee->Flags,
1295                                     IEEE_SGE_FLAGS_END_OF_LIST);
1296                         }
1297                 }

1299                 *datalen += acmd->cmd_dmacookies[i].dmac_size;

1301 #ifdef DEBUG
1302                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Address]: %" PRIx64,
1303                     scsi_raid_io_sgl_ieee->Address));
1304                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Length]:%x",
1305                     scsi_raid_io_sgl_ieee->Length));
1306                 con_log(CL_DLEVEL1, (CE_NOTE, "[SGL Flags]:%x",
1307                     scsi_raid_io_sgl_ieee->Flags));
1308 #endif

1310         }

1312         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset, 0);

1314         /* check if chained SGL required */
1315         if (i < numElements) {
```

```
1317                 con_log(CL_ANN1, (CE_NOTE, "[Chain Element index]:%x", i));

1319                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1320                         uint16_t ioFlags =
1321                             ddi_get16(acc_handle, &scsi_raid_io->IoFlags);

1323                         if ((ioFlags &
1324                             MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) !=
1325                             MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH) {
1326                                 ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1327                                     (U8)instance->chain_offset_io_req);
1328                         } else {
1329                                 ddi_put8(acc_handle,
1330                                     &scsi_raid_io->ChainOffset, 0);
1331                         }
1332                 } else {
1333                         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
1334                             (U8)instance->chain_offset_io_req);
1335                 }

1337                 /* prepare physical chain element */
1338                 ieeeChainElement = scsi_raid_io_sgl_ieee;

1340                 ddi_put8(acc_handle, &ieeeChainElement->NextChainOffset, 0);

1342                 if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1343                         ddi_put8(acc_handle, &ieeeChainElement->Flags,
1344                             IEEE_SGE_FLAGS_CHAIN_ELEMENT);
1345                 } else {
1346                         ddi_put8(acc_handle, &ieeeChainElement->Flags,
1347                             (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
1348                             MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
1349                 }

1351                 ddi_put32(acc_handle, &ieeeChainElement->Length,
1352                     (sizeof (MPI2_SGE_IO_UNION) * (numElements - i)));

1354                 ddi_put64(acc_handle, &ieeeChainElement->Address,
1355                     (U64)cmd->sgl_phys_addr);

1357                 sg_to_process = numElements - i;

1359                 con_log(CL_ANN1, (CE_NOTE,
1360                     "[Additional SGE Count]:%x", endElement));

1362                 /* point to the chained SGL buffer */
1363                 scsi_raid_io_sgl_ieee = (Mpi25IeeeSgeChain64_t *)cmd->sgl;

1365                 /* build rest of the SGL in chained buffer */
1366                 for (j = 0; j < sg_to_process; j++, scsi_raid_io_sgl_ieee++) {
1367                         con_log(CL_DLEVEL3, (CE_NOTE, "[remaining SGL]:%x", i));

1369                         ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
1370                             acmd->cmd_dmacookies[i].dmac_laddress);

1372                         ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length,
1373                             acmd->cmd_dmacookies[i].dmac_size);

1375                         ddi_put8(acc_handle, &scsi_raid_io_sgl_ieee->Flags, 0);

1377                         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1378                                 if (i == (numElements - 1)) {
1379                                         ddi_put8(acc_handle,
1380                                             &scsi_raid_io_sgl_ieee->Flags,
1381                                             IEEE_SGE_FLAGS_END_OF_LIST);
```



```
1382                                 }
1383                         }

1385                         *datalen += acmd->cmd_dmacookies[i].dmac_size;

1387 #if DEBUG
1388                         con_log(CL_DLEVEL1, (CE_NOTE,
1389                             "[SGL Address]: %" PRIx64,
1390                             scsi_raid_io_sgl_ieee->Address));
1391                         con_log(CL_DLEVEL1, (CE_NOTE,
1392                             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
1393                         con_log(CL_DLEVEL1, (CE_NOTE,
1394                             "[SGL Flags]:%x", scsi_raid_io_sgl_ieee->Flags));
1395 #endif

1397                         i++;
1398                 }
1399         }

1401         return (0);
1402 } /*end of BuildScatterGather */


1405 /*
1406  * build_cmd
1407  */
1408 static struct mrsas_cmd *
1409 mrsas_tbolt_build_cmd(struct mrsas_instance *instance, struct scsi_address *ap,
1410     struct scsi_pkt *pkt, uchar_t *cmd_done)
1411 {
1412         uint8_t         fp_possible = 0;
1413         uint32_t        index;
1414         uint32_t        lba_count = 0;
1415         uint32_t        start_lba_hi = 0;
1416         uint32_t        start_lba_lo = 0;
1417         ddi_acc_handle_t acc_handle =
1418             instance->mpi2_frame_pool_dma_obj.acc_handle;
1419         struct mrsas_cmd                *cmd = NULL;
1420         struct scsa_cmd                 *acmd = PKT2CMD(pkt);
1421         MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
1422         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
1423         uint32_t                        datalen;
1424         struct IO_REQUEST_INFO io_info;
1425         MR_FW_RAID_MAP_ALL *local_map_ptr;
1426         uint16_t pd_cmd_cdblen;

1428         con_log(CL_DLEVEL1, (CE_NOTE,
1429             "chkpnt: Entered mrsas_tbolt_build_cmd:%d", __LINE__));

1431         /* find out if this is logical or physical drive command.  */
1432         acmd->islogical = MRDRV_IS_LOGICAL(ap);
1433         acmd->device_id = MAP_DEVICE_ID(instance, ap);

1435         *cmd_done = 0;

1437         /* get the command packet */
1438         if (!(cmd = get_raid_msg_pkt(instance))) {
1439                 return (NULL);
1440         }

1442         index = cmd->index;
1443         ReqDescUnion =  mr_sas_get_request_descriptor(instance, index);
1444         ReqDescUnion->Words = 0;
1445         ReqDescUnion->SCSIIO.SMID = cmd->SMID;
1446         ReqDescUnion->SCSIIO.RequestFlags =
1447             (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
```

```
1448             MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);


1451         cmd->request_desc = ReqDescUnion;
1452         cmd->pkt = pkt;
1453         cmd->cmd = acmd;

1455         /* lets get the command directions */
1456         if (acmd->cmd_flags & CFLAG_DMASEND) {
1457                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1458                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
1459                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
1460                             DDI_DMA_SYNC_FORDEV);
1461                 }
1462         } else if (acmd->cmd_flags & ~CFLAG_DMASEND) {
1463                 if (acmd->cmd_flags & CFLAG_CONSISTENT) {
1464                         (void) ddi_dma_sync(acmd->cmd_dmahandle,
1465                             acmd->cmd_dma_offset, acmd->cmd_dma_len,
1466                             DDI_DMA_SYNC_FORCPU);
1467                 }
1468         } else {
1469                 con_log(CL_ANN, (CE_NOTE, "NO DMA"));
1470         }


1473         /* get SCSI_IO raid message frame pointer */
1474         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

1476         /* zero out SCSI_IO raid message frame */
1477         bzero(scsi_raid_io, sizeof (Mpi2RaidSCSIIORequest_t));

1479         /* Set the ldTargetId set by BuildRaidContext() */
1480         ddi_put16(acc_handle, &scsi_raid_io->RaidContext.ldTargetId,
1481             acmd->device_id);

1483         /*  Copy CDB to scsi_io_request message frame */
1484         ddi_rep_put8(acc_handle,
1485             (uint8_t *)pkt->pkt_cdbp, (uint8_t *)scsi_raid_io->CDB.CDB32,
1486             acmd->cmd_cdblen, DDI_DEV_AUTOINCR);

1488         /*
1489          * Just the CDB length, rest of the Flags are zero
1490          * This will be modified later.
1491          */
1492         ddi_put16(acc_handle, &scsi_raid_io->IoFlags, acmd->cmd_cdblen);

1494         pd_cmd_cdblen = acmd->cmd_cdblen;

1496         switch (pkt->pkt_cdbp[0]) {
1497         case SCMD_READ:
1498         case SCMD_WRITE:
1499         case SCMD_READ_G1:
1500         case SCMD_WRITE_G1:
1501         case SCMD_READ_G4:
1502         case SCMD_WRITE_G4:
1503         case SCMD_READ_G5:
1504         case SCMD_WRITE_G5:

1506                 if (acmd->islogical) {
1507                         /* Initialize sense Information */
1508                         if (cmd->sense1 == NULL) {
1509                                 con_log(CL_ANN, (CE_NOTE, "tbolt_build_cmd: "
1510                                     "Sense buffer ptr NULL "));
1511                         }
1512                         bzero(cmd->sense1, SENSE_LENGTH);
1513                         con_log(CL_DLEVEL2, (CE_NOTE, "tbolt_build_cmd "
```



```
1514                             "CDB[0] = %x\n", pkt->pkt_cdbp[0]));

1516                         if (acmd->cmd_cdblen == CDB_GROUP0) {
1517                                 /* 6-byte cdb */
1518                                 lba_count = (uint16_t)(pkt->pkt_cdbp[4]);
1519                                 start_lba_lo = ((uint32_t)(pkt->pkt_cdbp[3]) |
1520                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 8) |
1521                                     ((uint32_t)((pkt->pkt_cdbp[1]) & 0x1F)
1522                                     << 16));
1523                         } else if (acmd->cmd_cdblen == CDB_GROUP1) {
1524                                 /* 10-byte cdb */
1525                                 lba_count =
1526                                     (((uint16_t)(pkt->pkt_cdbp[8])) |
1527                                     ((uint16_t)(pkt->pkt_cdbp[7]) << 8));

1529                                 start_lba_lo =
1530                                     (((uint32_t)(pkt->pkt_cdbp[5])) |
1531                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1532                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1533                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1535                         } else if (acmd->cmd_cdblen == CDB_GROUP5) {
1536                                 /* 12-byte cdb */
1537                                 lba_count = (
1538                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
1539                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1540                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1541                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24));

1543                                 start_lba_lo =
1544                                     (((uint32_t)(pkt->pkt_cdbp[5])) |
1545                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1546                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1547                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));

1549                         } else if (acmd->cmd_cdblen == CDB_GROUP4) {
1550                                 /* 16-byte cdb */
1551                                 lba_count = (
1552                                     ((uint32_t)(pkt->pkt_cdbp[13])) |
1553                                     ((uint32_t)(pkt->pkt_cdbp[12]) << 8) |
1554                                     ((uint32_t)(pkt->pkt_cdbp[11]) << 16) |
1555                                     ((uint32_t)(pkt->pkt_cdbp[10]) << 24));

1557                                 start_lba_lo = (
1558                                     ((uint32_t)(pkt->pkt_cdbp[9])) |
1559                                     ((uint32_t)(pkt->pkt_cdbp[8]) << 8) |
1560                                     ((uint32_t)(pkt->pkt_cdbp[7]) << 16) |
1561                                     ((uint32_t)(pkt->pkt_cdbp[6]) << 24));

1563                                 start_lba_hi = (
1564                                     ((uint32_t)(pkt->pkt_cdbp[5])) |
1565                                     ((uint32_t)(pkt->pkt_cdbp[4]) << 8) |
1566                                     ((uint32_t)(pkt->pkt_cdbp[3]) << 16) |
1567                                     ((uint32_t)(pkt->pkt_cdbp[2]) << 24));
1568                         }

1570                         if (instance->tbolt &&
1571                             ((lba_count * 512) > mrsas_tbolt_max_cap_maxxfer)) {
1572                                 cmn_err(CE_WARN, " IO SECTOR COUNT exceeds "
1573                                     "controller limit 0x%x sectors",
1574                                     lba_count);
1575                         }

1577                         bzero(&io_info, sizeof (struct IO_REQUEST_INFO));
1578                         io_info.ldStartBlock = ((uint64_t)start_lba_hi << 32) |
1579                             start_lba_lo;
```

```
1580                        io_info.numBlocks = lba_count;
1581                        io_info.ldTgtId = acmd->device_id;

1583                        if (acmd->cmd_flags & CFLAG_DMASEND)
1584                                io_info.isRead = 0;
1585                        else
1586                                io_info.isRead = 1;


1589                        /* Acquire SYNC MAP UPDATE lock */
1590                        mutex_enter(&instance->sync_map_mtx);

1592                        local_map_ptr =
1593                            instance->ld_map[(instance->map_id & 1)];

1595                        if ((MR_TargetIdToLdGet(
1596                            acmd->device_id, local_map_ptr) >=
1597                            MAX_LOGICAL_DRIVES) || !instance->fast_path_io) {
1598                                cmn_err(CE_NOTE, "Fast Path NOT Possible, "
1599                                    "targetId >= MAX_LOGICAL_DRIVES || "
1600                                    "!instance->fast_path_io");
1601                                fp_possible = 0;
1602                                /* Set Regionlock flags to BYPASS */
1603                                /* io_request->RaidContext.regLockFlags  = 0; */
1604                                ddi_put8(acc_handle,
1605                                    &scsi_raid_io->RaidContext.regLockFlags, 0);
1606                        } else {
1607                                if (MR_BuildRaidContext(instance, &io_info,
1608                                    &scsi_raid_io->RaidContext, local_map_ptr))
1609                                        fp_possible = io_info.fpOkForIo;
1610                        }

1612                        if (!enable_fp)
1613                                fp_possible = 0;

1615                        con_log(CL_ANN1, (CE_NOTE, "enable_fp %d  "
1616                            "instance->fast_path_io %d fp_possible %d",
1617                            enable_fp, instance->fast_path_io, fp_possible));

1619                if (fp_possible) {

1621                        /* Check for DIF enabled LD */
1622                        if (MR_CheckDIF(acmd->device_id, local_map_ptr)) {
1623                                /* Prepare 32 Byte CDB for DIF capable Disk */
1624                                mrsas_tbolt_prepare_cdb(instance,
1625                                    scsi_raid_io->CDB.CDB32,
1626                                    &io_info, scsi_raid_io, start_lba_lo);
1627                        } else {
1628                                mrsas_tbolt_set_pd_lba(scsi_raid_io->CDB.CDB32,
1629                                    (uint8_t *)&pd_cmd_cdblen,
1630                                    io_info.pdBlock, io_info.numBlocks);
1631                                ddi_put16(acc_handle,
1632                                    &scsi_raid_io->IoFlags, pd_cmd_cdblen);
1633                        }

1635                        ddi_put8(acc_handle, &scsi_raid_io->Function,
1636                            MPI2_FUNCTION_SCSI_IO_REQUEST);

1638                        ReqDescUnion->SCSIIO.RequestFlags =
1639                            (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
1640                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1642                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1643                                uint8_t regLockFlags = ddi_get8(acc_handle,
1644                                    &scsi_raid_io->RaidContext.regLockFlags);
1645                                uint16_t IoFlags = ddi_get16(acc_handle,
```



```
1646                                    &scsi_raid_io->IoFlags);

1648                                if (regLockFlags == REGION_TYPE_UNUSED)
1649                                        ReqDescUnion->SCSIIO.RequestFlags =
1650                                            (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
1651                                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1653                                IoFlags |=
1654                                    MPI25_SAS_DEVICE0_FLAGS_ENABLED_FAST_PATH;
1655                                regLockFlags |=
1656                                    (MR_RL_FLAGS_GRANT_DESTINATION_CUDA |
1657                                    MR_RL_FLAGS_SEQ_NUM_ENABLE);

1659                                ddi_put8(acc_handle,
1660                                    &scsi_raid_io->ChainOffset, 0);
1661                                ddi_put8(acc_handle,
1662                                    &scsi_raid_io->RaidContext.nsegType,
1663                                    ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1664                                    MPI2_TYPE_CUDA));
1665                                ddi_put8(acc_handle,
1666                                    &scsi_raid_io->RaidContext.regLockFlags,
1667                                    regLockFlags);
1668                                ddi_put16(acc_handle,
1669                                    &scsi_raid_io->IoFlags, IoFlags);
1670                        }

1672                        if ((instance->load_balance_info[
1673                            acmd->device_id].loadBalanceFlag) &&
1674                            (io_info.isRead)) {
1675                                io_info.devHandle =
1676                                    get_updated_dev_handle(&instance->
1677                                    load_balance_info[acmd->device_id],
1678                                    &io_info);
1679                                cmd->load_balance_flag |=
1680                                    MEGASAS_LOAD_BALANCE_FLAG;
1681                        } else {
1682                                cmd->load_balance_flag &=
1683                                    ~MEGASAS_LOAD_BALANCE_FLAG;
1684                        }

1686                        ReqDescUnion->SCSIIO.DevHandle = io_info.devHandle;
1687                        ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1688                            io_info.devHandle);

1690                } else {
1691                        ddi_put8(acc_handle, &scsi_raid_io->Function,
1692                            MPI2_FUNCTION_LD_IO_REQUEST);

1694                        ddi_put16(acc_handle,
1695                            &scsi_raid_io->DevHandle, acmd->device_id);

1697                        ReqDescUnion->SCSIIO.RequestFlags =
1698                            (MPI2_REQ_DESCRIPT_FLAGS_LD_IO <<
1699                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

1701                        ddi_put16(acc_handle,
1702                            &scsi_raid_io->RaidContext.timeoutValue,
1703                            local_map_ptr->raidMap.fpPdIoTimeoutSec);

1705                        if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
1706                                uint8_t regLockFlags = ddi_get8(acc_handle,
1707                                    &scsi_raid_io->RaidContext.regLockFlags);

1709                                if (regLockFlags == REGION_TYPE_UNUSED) {
1710                                        ReqDescUnion->SCSIIO.RequestFlags =
1711                                            (MPI2_REQ_DESCRIPT_FLAGS_NO_LOCK <<
```

```
1712                                            MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1713                                }
1714
1715                                regLockFlags |=
1716                                    (MR_RL_FLAGS_GRANT_DESTINATION_CPU0 |
1717                                    MR_RL_FLAGS_SEQ_NUM_ENABLE);
1718
1719                                ddi_put8(acc_handle,
1720                                    &scsi_raid_io->RaidContext.nsegType,
1721                                    ((0x01 << MPI2_NSEG_FLAGS_SHIFT) |
1722                                    MPI2_TYPE_CUDA));
1723                                ddi_put8(acc_handle,
1724                                    &scsi_raid_io->RaidContext.regLockFlags,
1725                                    regLockFlags);
1726                        }
1727                } /* Not FP */
1728
1729                /* Release SYNC MAP UPDATE lock */
1730                mutex_exit(&instance->sync_map_mtx);
1731
1732
1733                /*
1734                 * Set sense buffer physical address/length in scsi_io_request.
1735                 */
1736                ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
1737                    cmd->sense_phys_addr1);
1738                ddi_put8(acc_handle, &scsi_raid_io->SenseBufferLength,
1739                    SENSE_LENGTH);
1740
1741                /* Construct SGL */
1742                ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1743                    offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);
1744
1745                (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1746                    scsi_raid_io, &datalen);
1747
1748                ddi_put32(acc_handle, &scsi_raid_io->DataLength, datalen);
1749
1750                break;
1751 #ifndef PDSUPPORT       /* if PDSUPPORT, skip break and fall through */
1752        } else {
1753                break;
1754 #endif
1755        }
1756        /* fall through For all non-rd/wr cmds */
1757        default:
1758                switch (pkt->pkt_cdbp[0]) {
1759                case 0x35: { /* SCMD_SYNCHRONIZE_CACHE */
1760                        return_raid_msg_pkt(instance, cmd);
1761                        *cmd_done = 1;
1762                        return (NULL);
1763                }
1764
1765                case SCMD_MODE_SENSE:
1766                case SCMD_MODE_SENSE_G1: {
1767                        union scsi_cdb  *cdbp;
1768                        uint16_t        page_code;
1769
1770                        cdbp = (void *)pkt->pkt_cdbp;
1771                        page_code = (uint16_t)cdbp->cdb_un.sg.scsi[0];
1772                        switch (page_code) {
1773                        case 0x3:
1774                        case 0x4:
1775                                (void) mrsas_mode_sense_build(pkt);
1776                                return_raid_msg_pkt(instance, cmd);
1777                                *cmd_done = 1;
```



```
1778                                return (NULL);
1779                        }
1780                        break;
1781                }
1782
1783                default: {
1784                        /*
1785                         * Here we need to handle PASSTHRU for
1786                         * Logical Devices. Like Inquiry etc.
1787                         */
1788
1789                        if (!(acmd->islogical)) {
1790
1791                                /* Acquire SYNC MAP UPDATE lock */
1792                                mutex_enter(&instance->sync_map_mtx);
1793
1794                                local_map_ptr =
1795                                    instance->ld_map[(instance->map_id & 1)];
1796
1797                                ddi_put8(acc_handle, &scsi_raid_io->Function,
1798                                    MPI2_FUNCTION_SCSI_IO_REQUEST);
1799
1800                                ReqDescUnion->SCSIIO.RequestFlags =
1801                                    (MPI2_REQ_DESCRIPT_FLAGS_HIGH_PRIORITY <<
1802                                    MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
1803
1804                                ddi_put16(acc_handle, &scsi_raid_io->DevHandle,
1805                                    local_map_ptr->raidMap.
1806                                    devHndlInfo[acmd->device_id].curDevHdl);
1807
1808
1809                                /* Set regLockFlasgs to REGION_TYPE_BYPASS */
1810                                ddi_put8(acc_handle,
1811                                    &scsi_raid_io->RaidContext.regLockFlags, 0);
1812                                ddi_put64(acc_handle,
1813                                    &scsi_raid_io->RaidContext.regLockRowLBA,
1814                                    0);
1815                                ddi_put32(acc_handle,
1816                                    &scsi_raid_io->RaidContext.regLockLength,
1817                                    0);
1818                                ddi_put8(acc_handle,
1819                                    &scsi_raid_io->RaidContext.RAIDFlags,
1820                                    MR_RAID_FLAGS_IO_SUB_TYPE_SYSTEM_PD <<
1821                                    MR_RAID_CTX_RAID_FLAGS_IO_SUB_TYPE_SHIFT);
1822                                ddi_put16(acc_handle,
1823                                    &scsi_raid_io->RaidContext.timeoutValue,
1824                                    local_map_ptr->raidMap.fpPdIoTimeoutSec);
1825                                ddi_put16(acc_handle,
1826                                    &scsi_raid_io->RaidContext.ldTargetId,
1827                                    acmd->device_id);
1828                                ddi_put8(acc_handle,
1829                                    &scsi_raid_io->LUN[1], acmd->lun);
1830
1831                                /* Release SYNC MAP UPDATE lock */
1832                                mutex_exit(&instance->sync_map_mtx);
1833
1834                        } else {
1835                                ddi_put8(acc_handle, &scsi_raid_io->Function,
1836                                    MPI2_FUNCTION_LD_IO_REQUEST);
1837                                ddi_put8(acc_handle,
1838                                    &scsi_raid_io->LUN[1], acmd->lun);
1839                                ddi_put16(acc_handle,
1840                                    &scsi_raid_io->DevHandle, acmd->device_id);
1841                                ReqDescUnion->SCSIIO.RequestFlags =
1842                                    (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
1843                                    MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);
```

```
1844                        }

1846                        /*
1847                         * Set sense buffer physical address/length in
1848                         * scsi_io_request.
1849                         */
1850                        ddi_put32(acc_handle,
1851                            &scsi_raid_io->SenseBufferLowAddress,
1852                            cmd->sense_phys_addr1);
1853                        ddi_put8(acc_handle,
1854                            &scsi_raid_io->SenseBufferLength, SENSE_LENGTH);

1856                        /* Construct SGL */
1857                        ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
1858                            offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

1860                        (void) mr_sas_tbolt_build_sgl(instance, acmd, cmd,
1861                            scsi_raid_io, &datalen);

1863                        ddi_put32(acc_handle,
1864                            &scsi_raid_io->DataLength, datalen);


1867                        con_log(CL_ANN, (CE_CONT,
1868                            "tbolt_build_cmd CDB[0] =%x, TargetID =%x\n",
1869                            pkt->pkt_cdbp[0], acmd->device_id));
1870                        con_log(CL_DLEVEL1, (CE_CONT,
1871                            "data length = %x\n",
1872                            scsi_raid_io->DataLength));
1873                        con_log(CL_DLEVEL1, (CE_CONT,
1874                            "cdb length = %x\n",
1875                            acmd->cmd_cdblen));
1876                }
1877                        break;
1878                }

1880        }

1882        return (cmd);
1883 }

1885 /*
1886  * mrsas_tbolt_tran_init_pkt - allocate & initialize a scsi_pkt structure
1887  * @ap:
1888  * @pkt:
1889  * @bp:
1890  * @cmdlen:
1891  * @statuslen:
1892  * @tgtlen:
1893  * @flags:
1894  * @callback:
1895  *
1896  * The tran_init_pkt() entry point allocates and initializes a scsi_pkt
1897  * structure and DMA resources for a target driver request. The
1898  * tran_init_pkt() entry point is called when the target driver calls the
1899  * SCSA function scsi_init_pkt(). Each call of the tran_init_pkt() entry point
1900  * is a request to perform one or more of three possible services:
1901  *  - allocation and initialization of a scsi_pkt structure
1902  *  - allocation of DMA resources for data transfer
1903  *  - reallocation of DMA resources for the next portion of the data transfer
1904  */
1905 struct scsi_pkt *
1906 mrsas_tbolt_tran_init_pkt(struct scsi_address *ap,
1907         register struct scsi_pkt *pkt,
1908         struct buf *bp, int cmdlen, int statuslen, int tgtlen,
1909         int flags, int (*callback)(), caddr_t arg)
```



```
1910 {
1911         struct scsa_cmd *acmd;
1912         struct mrsas_instance   *instance;
1913         struct scsi_pkt *new_pkt;

1915         instance = ADDR2MR(ap);

1917         /* step #1 : pkt allocation */
1918         if (pkt == NULL) {
1919                 pkt = scsi_hba_pkt_alloc(instance->dip, ap, cmdlen, statuslen,
1920                     tgtlen, sizeof (struct scsa_cmd), callback, arg);
1921                 if (pkt == NULL) {
1922                         return (NULL);
1923                 }

1925                 acmd = PKT2CMD(pkt);

1927                 /*
1928                  * Initialize the new pkt - we redundantly initialize
1929                  * all the fields for illustrative purposes.
1930                  */
1931                 acmd->cmd_pkt           = pkt;
1932                 acmd->cmd_flags         = 0;
1933                 acmd->cmd_scblen        = statuslen;
1934                 acmd->cmd_cdblen        = cmdlen;
1935                 acmd->cmd_dmahandle     = NULL;
1936                 acmd->cmd_ncookies      = 0;
1937                 acmd->cmd_cookie        = 0;
1938                 acmd->cmd_cookiecnt     = 0;
1939                 acmd->cmd_nwin          = 0;

1941                 pkt->pkt_address        = *ap;
1942                 pkt->pkt_comp           = (void (*)())NULL;
1943                 pkt->pkt_flags          = 0;
1944                 pkt->pkt_time           = 0;
1945                 pkt->pkt_resid          = 0;
1946                 pkt->pkt_state          = 0;
1947                 pkt->pkt_statistics     = 0;
1948                 pkt->pkt_reason         = 0;
1949                 new_pkt                 = pkt;
1950         } else {
1951                 acmd = PKT2CMD(pkt);
1952                 new_pkt = NULL;
1953         }

1955         /* step #2 : dma allocation/move */
1956         if (bp && bp->b_bcount != 0) {
1957                 if (acmd->cmd_dmahandle == NULL) {
1958                         if (mrsas_dma_alloc(instance, pkt, bp, flags,
1959                             callback) == DDI_FAILURE) {
1960                                 if (new_pkt) {
1961                                         scsi_hba_pkt_free(ap, new_pkt);
1962                                 }
1963                                 return ((struct scsi_pkt *)NULL);
1964                         }
1965                 } else {
1966                         if (mrsas_dma_move(instance, pkt, bp) == DDI_FAILURE) {
1967                                 return ((struct scsi_pkt *)NULL);
1968                         }
1969                 }
1970         }
1971         return (pkt);
1972 }


1975 uint32_t
```

```
1976 tbolt_read_fw_status_reg(struct mrsas_instance *instance)
1977 {
1978         return ((uint32_t)RD_OB_SCRATCH_PAD_0(instance));
1979 }

1981 void
1982 tbolt_issue_cmd(struct mrsas_cmd *cmd, struct mrsas_instance *instance)
1983 {
1984         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;
1985         atomic_add_16(&instance->fw_outstanding, 1);

1987         struct scsi_pkt *pkt;

1989         con_log(CL_ANN1,
1990             (CE_NOTE, "tbolt_issue_cmd: cmd->[SMID]=0x%X", cmd->SMID));

1992         con_log(CL_DLEVEL1, (CE_CONT,
1993             " [req desc Words] %" PRIx64 " \n", req_desc->Words));
1994         con_log(CL_DLEVEL1, (CE_CONT,
1995             " [req desc low part] %x \n",
1996             (uint_t)(req_desc->Words & 0xffffffffff)));
1997         con_log(CL_DLEVEL1, (CE_CONT,
1998             " [req desc high part] %x \n", (uint_t)(req_desc->Words >> 32)));
1999         pkt = cmd->pkt;

2001         if (pkt) {
2002                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
2003                     "ISSUED CMD TO FW : called : cmd:"
2004                     ": %p instance : %p pkt : %p pkt_time : %x\n",
2005                     gethrtime(), (void *)cmd, (void *)instance,
2006                     (void *)pkt, cmd->drv_pkt_time));
2007                 if (instance->adapterresetinprogress) {
2008                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2009                         con_log(CL_ANN, (CE_NOTE,
2010                             "TBOLT Reset the scsi_pkt timer"));
2011                 } else {
2012                         push_pending_mfi_pkt(instance, cmd);
2013                 }

2015         } else {
2016                 con_log(CL_ANN1, (CE_CONT, "%llx :TBOLT issue_cmd_ppc:"
2017                     "ISSUED CMD TO FW : called : cmd : %p, instance: %p"
2018                     "(NO PKT)\n", gethrtime(), (void *)cmd, (void *)instance));
2019         }

2021         /* Issue the command to the FW */
2022         mutex_enter(&instance->reg_write_mtx);
2023         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2024         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2025         mutex_exit(&instance->reg_write_mtx);
2026 }

2028 /*
2029  * issue_cmd_in_sync_mode
2030  */
2031 int
2032 tbolt_issue_cmd_in_sync_mode(struct mrsas_instance *instance,
2033     struct mrsas_cmd *cmd)
2034 {
2035         int             i;
2036         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2037         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;

2039         struct mrsas_header     *hdr;
2040         hdr = (struct mrsas_header *)&cmd->frame->hdr;
```



```
2042         con_log(CL_ANN,
2043             (CE_NOTE, "tbolt_issue_cmd_in_sync_mode: cmd->[SMID]=0x%X",
2044             cmd->SMID));


2047         if (instance->adapterresetinprogress) {
2048                 cmd->drv_pkt_time = ddi_get16
2049                     (cmd->frame_dma_obj.acc_handle, &hdr->timeout);
2050                 if (cmd->drv_pkt_time < debug_timeout_g)
2051                         cmd->drv_pkt_time = (uint16_t)debug_timeout_g;
2052                 con_log(CL_ANN, (CE_NOTE, "tbolt_issue_cmd_in_sync_mode:"
2053                     "RESET-IN-PROGRESS, issue cmd & return."));

2055                 mutex_enter(&instance->reg_write_mtx);
2056                 WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2057                 WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2058                 mutex_exit(&instance->reg_write_mtx);

2060                 return (DDI_SUCCESS);
2061         } else {
2062                 con_log(CL_ANN1, (CE_NOTE,
2063                     "tbolt_issue_cmd_in_sync_mode: pushing the pkt"));
2064                 push_pending_mfi_pkt(instance, cmd);
2065         }

2067         con_log(CL_DLEVEL2, (CE_NOTE,
2068             "HighQport offset :%p",
2069             (void *)((uintptr_t)(instance)->regmap + IB_HIGH_QPORT)));
2070         con_log(CL_DLEVEL2, (CE_NOTE,
2071             "LowQport offset :%p",
2072             (void *)((uintptr_t)(instance)->regmap + IB_LOW_QPORT)));

2074         cmd->sync_cmd = MRSAS_TRUE;
2075         cmd->cmd_status =  ENODATA;


2078         mutex_enter(&instance->reg_write_mtx);
2079         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2080         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2081         mutex_exit(&instance->reg_write_mtx);

2083         con_log(CL_ANN1, (CE_NOTE,
2084             " req desc high part %x", (uint_t)(req_desc->Words >> 32)));
2085         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x",
2086             (uint_t)(req_desc->Words & 0xffffffff)));

2088         mutex_enter(&instance->int_cmd_mtx);
2089         for (i = 0; i < msecs && (cmd->cmd_status == ENODATA); i++) {
2090                 cv_wait(&instance->int_cmd_cv, &instance->int_cmd_mtx);
2091         }
2092         mutex_exit(&instance->int_cmd_mtx);


2095         if (i < (msecs -1)) {
2096                 return (DDI_SUCCESS);
2097         } else {
2098                 return (DDI_FAILURE);
2099         }
2100 }

2102 /*
2103  * issue_cmd_in_poll_mode
2104  */
2105 int
2106 tbolt_issue_cmd_in_poll_mode(struct mrsas_instance *instance,
2107     struct mrsas_cmd *cmd)
```

```
2108 {
2109         int             i;
2110         uint16_t        flags;
2111         uint32_t        msecs = MFI_POLL_TIMEOUT_SECS * MILLISEC;
2112         struct mrsas_header *frame_hdr;
2113
2114         con_log(CL_ANN,
2115             (CE_NOTE, "tbolt_issue_cmd_in_poll_mode: cmd->[SMID]=0x%X",
2116             cmd->SMID));
2117
2118         MRSAS_REQUEST_DESCRIPTOR_UNION *req_desc = cmd->request_desc;
2119
2120         frame_hdr = (struct mrsas_header *)&cmd->frame->hdr;
2121         ddi_put8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status,
2122             MFI_CMD_STATUS_POLL_MODE);
2123         flags = ddi_get16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags);
2124         flags   |= MFI_FRAME_DONT_POST_IN_REPLY_QUEUE;
2125         ddi_put16(cmd->frame_dma_obj.acc_handle, &frame_hdr->flags, flags);
2126
2127         con_log(CL_ANN1, (CE_NOTE, " req desc low part %x",
2128             (uint_t)(req_desc->Words & 0xffffffff)));
2129         con_log(CL_ANN1, (CE_NOTE,
2130             " req desc high part %x", (uint_t)(req_desc->Words >> 32)));
2131
2132         /* issue the frame using inbound queue port */
2133         mutex_enter(&instance->reg_write_mtx);
2134         WR_IB_LOW_QPORT((uint32_t)(req_desc->Words), instance);
2135         WR_IB_HIGH_QPORT((uint32_t)(req_desc->Words >> 32), instance);
2136         mutex_exit(&instance->reg_write_mtx);
2137
2138         for (i = 0; i < msecs && (
2139             ddi_get8(cmd->frame_dma_obj.acc_handle, &frame_hdr->cmd_status)
2140             == MFI_CMD_STATUS_POLL_MODE); i++) {
2141                 /* wait for cmd_status to change from 0xFF */
2142                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
2143         }
2144
2145         if (ddi_get8(cmd->frame_dma_obj.acc_handle,
2146             &frame_hdr->cmd_status) == MFI_CMD_STATUS_POLL_MODE) {
2147                 con_log(CL_ANN1, (CE_NOTE,
2148                     " cmd failed %" PRIx64, (req_desc->Words)));
2149                 return (DDI_FAILURE);
2150         }
2151
2152         return (DDI_SUCCESS);
2153 }
2154
2155 void
2156 tbolt_enable_intr(struct mrsas_instance *instance)
2157 {
2158         /* TODO: For Thunderbolt/Invader also clear intr on enable */
2159         /* writel(~0, &regs->outbound_intr_status); */
2160         /* readl(&regs->outbound_intr_status); */
2161
2162         WR_OB_INTR_MASK(~(MFI_FUSION_ENABLE_INTERRUPT_MASK), instance);
2163
2164         /* dummy read to force PCI flush */
2165         (void) RD_OB_INTR_MASK(instance);
2166
2167 }
2168
2169 void
2170 tbolt_disable_intr(struct mrsas_instance *instance)
2171 {
2172         uint32_t mask = 0xFFFFFFFF;
```



```
2174         WR_OB_INTR_MASK(mask, instance);
2175
2176         /* Dummy readl to force pci flush */
2177
2178         (void) RD_OB_INTR_MASK(instance);
2179 }
2180
2181
2182 int
2183 tbolt_intr_ack(struct mrsas_instance *instance)
2184 {
2185         uint32_t        status;
2186
2187         /* check if it is our interrupt */
2188         status = RD_OB_INTR_STATUS(instance);
2189         con_log(CL_ANN1, (CE_NOTE,
2190             "chkpnt: Entered tbolt_intr_ack status = %d", status));
2191
2192         if (!(status & MFI_FUSION_ENABLE_INTERRUPT_MASK)) {
2193                 return (DDI_INTR_UNCLAIMED);
2194         }
2195
2196         if (mrsas_check_acc_handle(instance->regmap_handle) != DDI_SUCCESS) {
2197                 ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2198                 return (DDI_INTR_UNCLAIMED);
2199         }
2200
2201         if ((status & 1) || (status & MFI_FUSION_ENABLE_INTERRUPT_MASK)) {
2202                 /* clear the interrupt by writing back the same value */
2203                 WR_OB_INTR_STATUS(status, instance);
2204                 /* dummy READ */
2205                 (void) RD_OB_INTR_STATUS(instance);
2206         }
2207         return (DDI_INTR_CLAIMED);
2208 }
2209
2210 /*
2211  * get_raid_msg_pkt : Get a command from the free pool
2212  * After successful allocation, the caller of this routine
2213  * must clear the frame buffer (memset to zero) before
2214  * using the packet further.
2215  *
2216  * ***** Note *****
2217  * After clearing the frame buffer the context id of the
2218  * frame buffer SHOULD be restored back.
2219  */
2220
2221 struct mrsas_cmd *
2222 get_raid_msg_pkt(struct mrsas_instance *instance)
2223 {
2224         mlist_t                 *head = &instance->cmd_pool_list;
2225         struct mrsas_cmd        *cmd = NULL;
2226
2227         mutex_enter(&instance->cmd_pool_mtx);
2228         ASSERT(mutex_owned(&instance->cmd_pool_mtx));
2229
2230
2231         if (!mlist_empty(head)) {
2232                 cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2233                 mlist_del_init(head->next);
2234         }
2235         if (cmd != NULL) {
2236                 cmd->pkt = NULL;
2237                 cmd->retry_count_for_ocr = 0;
2238                 cmd->drv_pkt_time = 0;
2239         }
```

```
2240         mutex_exit(&instance->cmd_pool_mtx);

2242         if (cmd != NULL)
2243                 bzero(cmd->scsi_io_request,
2244                     sizeof (Mpi2RaidSCSIIORequest_t));
2245         return (cmd);
2246 }

2248 struct mrsas_cmd *
2249 get_raid_msg_mfi_pkt(struct mrsas_instance *instance)
2250 {
2251         mlist_t                 *head = &instance->cmd_app_pool_list;
2252         struct mrsas_cmd        *cmd = NULL;

2254         mutex_enter(&instance->cmd_app_pool_mtx);
2255         ASSERT(mutex_owned(&instance->cmd_app_pool_mtx));

2257         if (!mlist_empty(head)) {
2258                 cmd = mlist_entry(head->next, struct mrsas_cmd, list);
2259                 mlist_del_init(head->next);
2260         }
2261         if (cmd != NULL) {
2262                 cmd->retry_count_for_ocr = 0;
2263                 cmd->drv_pkt_time = 0;
2264                 cmd->pkt = NULL;
2265                 cmd->request_desc = NULL;

2267         }

2269         mutex_exit(&instance->cmd_app_pool_mtx);

2271         if (cmd != NULL) {
2272                 bzero(cmd->scsi_io_request,
2273                     sizeof (Mpi2RaidSCSIIORequest_t));
2274         }

2276         return (cmd);
2277 }

2279 /*
2280  * return_raid_msg_pkt : Return a cmd to free command pool
2281  */
2282 void
2283 return_raid_msg_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2284 {
2285         mutex_enter(&instance->cmd_pool_mtx);
2286         ASSERT(mutex_owned(&instance->cmd_pool_mtx));


2289         mlist_add_tail(&cmd->list, &instance->cmd_pool_list);

2291         mutex_exit(&instance->cmd_pool_mtx);
2292 }

2294 void
2295 return_raid_msg_mfi_pkt(struct mrsas_instance *instance, struct mrsas_cmd *cmd)
2296 {
2297         mutex_enter(&instance->cmd_app_pool_mtx);
2298         ASSERT(mutex_owned(&instance->cmd_app_pool_mtx));

2300         mlist_add_tail(&cmd->list, &instance->cmd_app_pool_list);

2302         mutex_exit(&instance->cmd_app_pool_mtx);
2303 }
```



```
2306 void
2307 mr_sas_tbolt_build_mfi_cmd(struct mrsas_instance *instance,
2308     struct mrsas_cmd *cmd)
2309 {
2310         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2311         Mpi25IeeeSgeChain64_t           *scsi_raid_io_sgl_ieee;
2312         MRSAS_REQUEST_DESCRIPTOR_UNION  *ReqDescUnion;
2313         uint32_t                        index;
2314         ddi_acc_handle_t acc_handle =
2315             instance->mpi2_frame_pool_dma_obj.acc_handle;

2317         if (!instance->tbolt) {
2318                 con_log(CL_ANN, (CE_NOTE, "Not MFA enabled."));
2319                 return;
2320         }

2322         index = cmd->index;

2324         ReqDescUnion = mr_sas_get_request_descriptor(instance, index);

2326         if (!ReqDescUnion) {
2327                 con_log(CL_ANN1, (CE_NOTE, "[NULL REQDESC]"));
2328                 return;
2329         }

2331         con_log(CL_ANN1, (CE_NOTE, "[SMID]%x", cmd->SMID));

2333         ReqDescUnion->Words = 0;

2335         ReqDescUnion->SCSIIO.RequestFlags =
2336             (MPI2_REQ_DESCRIPT_FLAGS_SCSI_IO <<
2337             MPI2_REQ_DESCRIPT_FLAGS_TYPE_SHIFT);

2339         ReqDescUnion->SCSIIO.SMID = cmd->SMID;

2341         cmd->request_desc = ReqDescUnion;

2343         /* get raid message frame pointer */
2344         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2346         if (instance->device_id == PCI_DEVICE_ID_LSI_INVADER) {
2347                 Mpi25IeeeSgeChain64_t *sgl_ptr_end = (Mpi25IeeeSgeChain64_t *)
2348                     &scsi_raid_io->SGL.IeeeChain;
2349                 sgl_ptr_end += instance->max_sge_in_main_msg - 1;
2350                 ddi_put8(acc_handle, &sgl_ptr_end->Flags, 0);
2351         }

2353         ddi_put8(acc_handle, &scsi_raid_io->Function,
2354             MPI2_FUNCTION_PASSTHRU_IO_REQUEST);

2356         ddi_put8(acc_handle, &scsi_raid_io->SGLOffset0,
2357             offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 4);

2359         ddi_put8(acc_handle, &scsi_raid_io->ChainOffset,
2360             (U8)offsetof(MPI2_RAID_SCSI_IO_REQUEST, SGL) / 16);

2362         ddi_put32(acc_handle, &scsi_raid_io->SenseBufferLowAddress,
2363             cmd->sense_phys_addr1);


2366         scsi_raid_io_sgl_ieee =
2367             (Mpi25IeeeSgeChain64_t *)&scsi_raid_io->SGL.IeeeChain;

2369         ddi_put64(acc_handle, &scsi_raid_io_sgl_ieee->Address,
2370             (U64)cmd->frame_phys_addr);
```

```
2372         ddi_put8(acc_handle,
2373             &scsi_raid_io_sgl_ieee->Flags, (IEEE_SGE_FLAGS_CHAIN_ELEMENT |
2374             MPI2_IEEE_SGE_FLAGS_IOCPLBNTA_ADDR));
2375         /* LSI put hardcoded 1024 instead of MEGASAS_MAX_SZ_CHAIN_FRAME. */
2376         ddi_put32(acc_handle, &scsi_raid_io_sgl_ieee->Length, 1024);

2378         con_log(CL_ANN1, (CE_NOTE,
2379             "[MFI CMD PHY ADDRESS]:%" PRIx64,
2380             scsi_raid_io_sgl_ieee->Address));
2381         con_log(CL_ANN1, (CE_NOTE,
2382             "[SGL Length]:%x", scsi_raid_io_sgl_ieee->Length));
2383         con_log(CL_ANN1, (CE_NOTE, "[SGL Flags]:%x",
2384             scsi_raid_io_sgl_ieee->Flags));
2385 }


2388 void
2389 tbolt_complete_cmd(struct mrsas_instance *instance,
2390     struct mrsas_cmd *cmd)
2391 {
2392         uint8_t                         status;
2393         uint8_t                         extStatus;
2394         uint8_t                         arm;
2395         struct scsa_cmd                 *acmd;
2396         struct scsi_pkt                 *pkt;
2397         struct scsi_arq_status          *arqstat;
2398         Mpi2RaidSCSIIORequest_t         *scsi_raid_io;
2399         LD_LOAD_BALANCE_INFO            *lbinfo;
2400         ddi_acc_handle_t acc_handle =
2401             instance->mpi2_frame_pool_dma_obj.acc_handle;

2403         scsi_raid_io = (Mpi2RaidSCSIIORequest_t *)cmd->scsi_io_request;

2405         status = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.status);
2406         extStatus = ddi_get8(acc_handle, &scsi_raid_io->RaidContext.extStatus);

2408         con_log(CL_DLEVEL3, (CE_NOTE, "status %x", status));
2409         con_log(CL_DLEVEL3, (CE_NOTE, "extStatus %x", extStatus));

2411         if (status != MFI_STAT_OK) {
2412                 con_log(CL_ANN, (CE_WARN,
2413                     "IO Cmd Failed SMID %x", cmd->SMID));
2414         } else {
2415                 con_log(CL_ANN, (CE_NOTE,
2416                     "IO Cmd Success  SMID %x", cmd->SMID));
2417         }

2419         /* regular commands */

2421         switch (ddi_get8(acc_handle, &scsi_raid_io->Function)) {

2423         case MPI2_FUNCTION_SCSI_IO_REQUEST :  /* Fast Path IO. */
2424                 acmd =  (struct scsa_cmd *)cmd->cmd;
2425                 lbinfo = &instance->load_balance_info[acmd->device_id];

2427                 if (cmd->load_balance_flag & MEGASAS_LOAD_BALANCE_FLAG) {
2428                         arm = lbinfo->raid1DevHandle[0] ==
2429                             scsi_raid_io->DevHandle ? 0 : 1;

2431                         lbinfo->scsi_pending_cmds[arm]--;
2432                         cmd->load_balance_flag &= ~MEGASAS_LOAD_BALANCE_FLAG;
2433                 }
2434                 con_log(CL_DLEVEL3, (CE_NOTE,
2435                     "FastPath IO Completion Success "));
2436                 /* FALLTHRU */
```



```
2438         case MPI2_FUNCTION_LD_IO_REQUEST :   { /* Regular Path IO. */
2439                 acmd =  (struct scsa_cmd *)cmd->cmd;
2440                 pkt =   (struct scsi_pkt *)CMD2PKT(acmd);

2442                 if (acmd->cmd_flags & CFLAG_DMAVALID) {
2443                         if (acmd->cmd_flags & CFLAG_CONSISTENT) {
2444                                 (void) ddi_dma_sync(acmd->cmd_dmahandle,
2445                                     acmd->cmd_dma_offset, acmd->cmd_dma_len,
2446                                     DDI_DMA_SYNC_FORCPU);
2447                         }
2448                 }

2450                 pkt->pkt_reason         = CMD_CMPLT;
2451                 pkt->pkt_statistics     = 0;
2452                 pkt->pkt_state = STATE_GOT_BUS | STATE_GOT_TARGET |
2453                     STATE_SENT_CMD | STATE_XFERRED_DATA | STATE_GOT_STATUS;

2455                 con_log(CL_ANN, (CE_CONT, " CDB[0] = %x completed for %s: "
2456                     "size %lx SMID %x cmd_status %x", pkt->pkt_cdbp[0],
2457                     ((acmd->islogical) ? "LD" : "PD"),
2458                     acmd->cmd_dmacount, cmd->SMID, status));

2460                 if (pkt->pkt_cdbp[0] == SCMD_INQUIRY) {
2461                         struct scsi_inquiry     *inq;

2463                         if (acmd->cmd_dmacount != 0) {
2464                                 bp_mapin(acmd->cmd_buf);
2465                                 inq = (struct scsi_inquiry *)
2466                                     acmd->cmd_buf->b_un.b_addr;

2468                                 /* don't expose physical drives to OS */
2469                                 if (acmd->islogical &&
2470                                     (status == MFI_STAT_OK)) {
2471                                         display_scsi_inquiry((caddr_t)inq);
2472 #ifdef PDSUPPORT
2473                                 } else if ((status == MFI_STAT_OK) &&
2474                                     inq->inq_dtype == DTYPE_DIRECT) {
2475                                         display_scsi_inquiry((caddr_t)inq);
2476 #endif
2477                                 } else {
2478                                         /* for physical disk */
2479                                         status = MFI_STAT_DEVICE_NOT_FOUND;
2480                                 }
2481                         }
2482                 }

2484                 switch (status) {
2485                 case MFI_STAT_OK:
2486                         pkt->pkt_scbp[0] = STATUS_GOOD;
2487                         break;
2488                 case MFI_STAT_LD_CC_IN_PROGRESS:
2489                 case MFI_STAT_LD_RECON_IN_PROGRESS:
2490                         pkt->pkt_scbp[0] = STATUS_GOOD;
2491                         break;
2492                 case MFI_STAT_LD_INIT_IN_PROGRESS:
2493                         pkt->pkt_reason = CMD_TRAN_ERR;
2494                         break;
2495                 case MFI_STAT_SCSI_IO_FAILED:
2496                         cmn_err(CE_WARN, "tbolt_complete_cmd: scsi_io failed");
2497                         pkt->pkt_reason = CMD_TRAN_ERR;
2498                         break;
2499                 case MFI_STAT_SCSI_DONE_WITH_ERROR:
2500                         con_log(CL_ANN, (CE_WARN,
2501                             "tbolt_complete_cmd: scsi_done with error"));

2503                         pkt->pkt_reason = CMD_CMPLT;
```

```
2504                        ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;

2506                        if (pkt->pkt_cdbp[0] == SCMD_TEST_UNIT_READY) {
2507                                con_log(CL_ANN,
2508                                    (CE_WARN, "TEST_UNIT_READY fail"));
2509                        } else {
2510                                pkt->pkt_state |= STATE_ARQ_DONE;
2511                                arqstat = (void *)(pkt->pkt_scbp);
2512                                arqstat->sts_rqpkt_reason = CMD_CMPLT;
2513                                arqstat->sts_rqpkt_resid = 0;
2514                                arqstat->sts_rqpkt_state |=
2515                                    STATE_GOT_BUS | STATE_GOT_TARGET
2516                                    | STATE_SENT_CMD
2517                                    | STATE_XFERRED_DATA;
2518                                *(uint8_t *)&arqstat->sts_rqpkt_status =
2519                                    STATUS_GOOD;
2520                                con_log(CL_ANN1,
2521                                    (CE_NOTE, "Copying Sense data %x",
2522                                    cmd->SMID));

2524                                ddi_rep_get8(acc_handle,
2525                                    (uint8_t *)&(arqstat->sts_sensedata),
2526                                    cmd->sense1,
2527                                    sizeof (struct scsi_extended_sense),
2528                                    DDI_DEV_AUTOINCR);

2530                        }
2531                        break;
2532                case MFI_STAT_LD_OFFLINE:
2533                        cmn_err(CE_WARN,
2534                            "tbolt_complete_cmd: ld offline "
2535                            "CDB[0]=0x%x targetId=0x%x devhandle=0x%x",
2536                            /* UNDO: */
2537                            ddi_get8(acc_handle, &scsi_raid_io->CDB.CDB32[0]),

2539                            ddi_get16(acc_handle,
2540                            &scsi_raid_io->RaidContext.ldTargetId),

2542                            ddi_get16(acc_handle, &scsi_raid_io->DevHandle));

2544                        pkt->pkt_reason = CMD_DEV_GONE;
2545                        pkt->pkt_statistics  = STAT_DISCON;
2546                        break;
2547                case MFI_STAT_DEVICE_NOT_FOUND:
2548                        con_log(CL_ANN, (CE_CONT,
2549                            "tbolt_complete_cmd: device not found error"));
2550                        pkt->pkt_reason = CMD_DEV_GONE;
2551                        pkt->pkt_statistics  = STAT_DISCON;
2552                        break;

2554                case MFI_STAT_LD_LBA_OUT_OF_RANGE:
2555                        pkt->pkt_state |= STATE_ARQ_DONE;
2556                        pkt->pkt_reason = CMD_CMPLT;
2557                        ((struct scsi_status *)pkt->pkt_scbp)->sts_chk = 1;

2559                        arqstat = (void *)(pkt->pkt_scbp);
2560                        arqstat->sts_rqpkt_reason = CMD_CMPLT;
2561                        arqstat->sts_rqpkt_resid = 0;
2562                        arqstat->sts_rqpkt_state |= STATE_GOT_BUS
2563                            | STATE_GOT_TARGET | STATE_SENT_CMD
2564                            | STATE_XFERRED_DATA;
2565                        *(uint8_t *)&arqstat->sts_rqpkt_status = STATUS_GOOD;

2567                        arqstat->sts_sensedata.es_valid = 1;
2568                        arqstat->sts_sensedata.es_key = KEY_ILLEGAL_REQUEST;
2569                        arqstat->sts_sensedata.es_class = CLASS_EXTENDED_SENSE;
```



```
2571                        /*
2572                         * LOGICAL BLOCK ADDRESS OUT OF RANGE:
2573                         * ASC: 0x21h; ASCQ: 0x00h;
2574                         */
2575                        arqstat->sts_sensedata.es_add_code = 0x21;
2576                        arqstat->sts_sensedata.es_qual_code = 0x00;
2577                        break;
2578                case MFI_STAT_INVALID_CMD:
2579                case MFI_STAT_INVALID_DCMD:
2580                case MFI_STAT_INVALID_PARAMETER:
2581                case MFI_STAT_INVALID_SEQUENCE_NUMBER:
2582                default:
2583                        cmn_err(CE_WARN, "tbolt_complete_cmd: Unknown status!");
2584                        pkt->pkt_reason = CMD_TRAN_ERR;

2586                        break;
2587                }

2589                atomic_add_16(&instance->fw_outstanding, (-1));

2591                (void) mrsas_common_check(instance, cmd);
2592                if (acmd->cmd_dmahandle) {
2593                        if (mrsas_check_dma_handle(acmd->cmd_dmahandle) !=
2594                            DDI_SUCCESS) {
2595                                ddi_fm_service_impact(instance->dip,
2596                                    DDI_SERVICE_UNAFFECTED);
2597                                pkt->pkt_reason = CMD_TRAN_ERR;
2598                                pkt->pkt_statistics = 0;
2599                        }
2600                }

2602                /* Call the callback routine */
2603                if (((pkt->pkt_flags & FLAG_NOINTR) == 0) && pkt->pkt_comp)
2604                        (*pkt->pkt_comp)(pkt);

2606                con_log(CL_ANN1, (CE_NOTE, "Free smid %x", cmd->SMID));

2608                ddi_put8(acc_handle, &scsi_raid_io->RaidContext.status, 0);

2610                ddi_put8(acc_handle, &scsi_raid_io->RaidContext.extStatus, 0);

2612                return_raid_msg_pkt(instance, cmd);
2613                break;
2614        }
2615        case MPI2_FUNCTION_PASSTHRU_IO_REQUEST:  /* MFA command. */

2617                if (cmd->frame->dcmd.opcode == MR_DCMD_LD_MAP_GET_INFO &&
2618                    cmd->frame->dcmd.mbox.b[1] == 1) {

2620                        mutex_enter(&instance->sync_map_mtx);

2622                        con_log(CL_ANN, (CE_NOTE,
2623                            "LDMAP sync command SMID RECEIVED 0x%X",
2624                            cmd->SMID));
2625                        if (cmd->frame->hdr.cmd_status != 0) {
2626                                cmn_err(CE_WARN,
2627                                    "map sync failed, status = 0x%x.",
2628                                    cmd->frame->hdr.cmd_status);
2629                        } else {
2630                                instance->map_id++;
2631                                cmn_err(CE_NOTE,
2632                                    "map sync received, switched map_id to %"
2633                                    PRIu64 " \n", instance->map_id);
2634                        }
```

```
2636                        if (MR_ValidateMapInfo(instance->ld_map[
2637                            (instance->map_id & 1)],
2638                            instance->load_balance_info)) {
2639                                instance->fast_path_io = 1;
2640                        } else {
2641                                instance->fast_path_io = 0;
2642                        }

2644                        con_log(CL_ANN, (CE_NOTE,
2645                            "instance->fast_path_io %d",
2646                            instance->fast_path_io));

2648                        instance->unroll.syncCmd = 0;

2650                        if (instance->map_update_cmd == cmd) {
2651                                return_raid_msg_pkt(instance, cmd);
2652                                atomic_add_16(&instance->fw_outstanding, (-1));
2653                                (void) mrsas_tbolt_sync_map_info(instance);
2654                        }

2656                        cmn_err(CE_NOTE, "LDMAP sync completed.");
2657                        mutex_exit(&instance->sync_map_mtx);
2658                        break;
2659                }

2661                if (cmd->frame->dcmd.opcode == MR_DCMD_CTRL_EVENT_WAIT) {
2662                        con_log(CL_ANN1, (CE_CONT,
2663                            "AEN command SMID RECEIVED 0x%X",
2664                            cmd->SMID));
2665                        if ((instance->aen_cmd == cmd) &&
2666                            (instance->aen_cmd->abort_aen)) {
2667                                con_log(CL_ANN, (CE_WARN, "mrsas_softintr: "
2668                                    "aborted_aen returned"));
2669                        } else {
2670                                atomic_add_16(&instance->fw_outstanding, (-1));
2671                                service_mfi_aen(instance, cmd);
2672                        }
2673                }

2675                if (cmd->sync_cmd == MRSAS_TRUE) {
2676                        con_log(CL_ANN1, (CE_CONT,
2677                            "Sync-mode Command Response SMID RECEIVED 0x%X",
2678                            cmd->SMID));

2680                        tbolt_complete_cmd_in_sync_mode(instance, cmd);
2681                } else {
2682                        con_log(CL_ANN, (CE_CONT,
2683                            "tbolt_complete_cmd: Wrong SMID RECEIVED 0x%X",
2684                            cmd->SMID));
2685                }
2686                break;
2687        default:
2688                mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2689                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);

2691                /* free message */
2692                con_log(CL_ANN,
2693                    (CE_NOTE, "tbolt_complete_cmd: Unknown Type!!!!!!!!"));
2694                break;
2695        }
2696 }

2698 uint_t
2699 mr_sas_tbolt_process_outstanding_cmd(struct mrsas_instance *instance)
2700 {
2701        uint8_t                         replyType;
```



```
2702        Mpi2SCSIIOSuccessReplyDescriptor_t *replyDesc;
2703        Mpi2ReplyDescriptorsUnion_t     *desc;
2704        uint16_t                        smid;
2705        union desc_value                d_val;
2706        struct mrsas_cmd                *cmd;

2708        struct mrsas_header     *hdr;
2709        struct scsi_pkt         *pkt;

2711        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2712            0, 0, DDI_DMA_SYNC_FORDEV);

2714        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2715            0, 0, DDI_DMA_SYNC_FORCPU);

2717        desc = instance->reply_frame_pool;
2718        desc += instance->reply_read_index;

2720        replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;
2721        replyType = replyDesc->ReplyFlags &
2722            MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2724        if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2725                return (DDI_INTR_UNCLAIMED);

2727        if (mrsas_check_dma_handle(instance->mfi_internal_dma_obj.dma_handle)
2728            != DDI_SUCCESS) {
2729                mrsas_fm_ereport(instance, DDI_FM_DEVICE_NO_RESPONSE);
2730                ddi_fm_service_impact(instance->dip, DDI_SERVICE_LOST);
2731                con_log(CL_ANN1,
2732                    (CE_WARN, "mr_sas_tbolt_process_outstanding_cmd(): "
2733                    "FMA check, returning DDI_INTR_UNCLAIMED"));
2734                return (DDI_INTR_CLAIMED);
2735        }

2737        con_log(CL_ANN1, (CE_NOTE, "Reply Desc  = %p  Words = %" PRIx64,
2738            (void *)desc, desc->Words));

2740        d_val.word = desc->Words;


2743        /* Read Reply descriptor */
2744        while ((d_val.u1.low != 0xffffffff) &&
2745            (d_val.u1.high != 0xffffffff)) {

2747                (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2748                    0, 0, DDI_DMA_SYNC_FORCPU);

2750                smid = replyDesc->SMID;

2752                if (!smid || smid > instance->max_fw_cmds + 1) {
2753                        con_log(CL_ANN1, (CE_NOTE,
2754                            "Reply Desc at Break  = %p  Words = %" PRIx64,
2755                            (void *)desc, desc->Words));
2756                        break;
2757                }

2759                cmd     = instance->cmd_list[smid - 1];
2760                if (!cmd) {
2761                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_tbolt_process_"
2762                            "outstanding_cmd: Invalid command "
2763                            " or Poll commad Received in completion path"));
2764                } else {
2765                        mutex_enter(&instance->cmd_pend_mtx);
2766                        if (cmd->sync_cmd == MRSAS_TRUE) {
2767                                hdr = (struct mrsas_header *)&cmd->frame->hdr;
```

```
2768                                if (hdr) {
2769                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2770                                            "tbolt_process_outstanding_cmd:"
2771                                            " mlist_del_init(&cmd->list)."));
2772                                        mlist_del_init(&cmd->list);
2773                                }
2774                        } else {
2775                                pkt = cmd->pkt;
2776                                if (pkt) {
2777                                        con_log(CL_ANN1, (CE_NOTE, "mr_sas_"
2778                                            "tbolt_process_outstanding_cmd:"
2779                                            "mlist_del_init(&cmd->list)."));
2780                                        mlist_del_init(&cmd->list);
2781                                }
2782                        }

2784                        mutex_exit(&instance->cmd_pend_mtx);

2786                        tbolt_complete_cmd(instance, cmd);
2787                }
2788                /* set it back to all 1s. */
2789                desc->Words = -1LL;

2791                instance->reply_read_index++;

2793                if (instance->reply_read_index >= (instance->reply_q_depth)) {
2794                        con_log(CL_ANN1, (CE_NOTE, "wrap around"));
2795                        instance->reply_read_index = 0;
2796                }

2798                /* Get the next reply descriptor */
2799                if (!instance->reply_read_index)
2800                        desc = instance->reply_frame_pool;
2801                else
2802                        desc++;

2804                replyDesc = (MPI2_SCSI_IO_SUCCESS_REPLY_DESCRIPTOR *)desc;

2806                d_val.word = desc->Words;

2808                con_log(CL_ANN1, (CE_NOTE,
2809                    "Next Reply Desc  = %p Words = %" PRIx64,
2810                    (void *)desc, desc->Words));

2812                replyType = replyDesc->ReplyFlags &
2813                    MPI2_RPY_DESCRIPT_FLAGS_TYPE_MASK;

2815                if (replyType == MPI2_RPY_DESCRIPT_FLAGS_UNUSED)
2816                        break;

2818        } /* End of while loop. */

2820        /* update replyIndex to FW */
2821        WR_MPI2_REPLY_POST_INDEX(instance->reply_read_index, instance);


2824        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2825            0, 0, DDI_DMA_SYNC_FORDEV);

2827        (void) ddi_dma_sync(instance->reply_desc_dma_obj.dma_handle,
2828            0, 0, DDI_DMA_SYNC_FORCPU);
2829        return (DDI_INTR_CLAIMED);
2830 }
```



```
2835 /*
2836  * complete_cmd_in_sync_mode -  Completes an internal command
2837  * @instance:                  Adapter soft state
2838  * @cmd:                       Command to be completed
2839  *
2840  * The issue_cmd_in_sync_mode() function waits for a command to complete
2841  * after it issues a command. This function wakes up that waiting routine by
2842  * calling wake_up() on the wait queue.
2843  */
2844 void
2845 tbolt_complete_cmd_in_sync_mode(struct mrsas_instance *instance,
2846     struct mrsas_cmd *cmd)
2847 {

2849         cmd->cmd_status = ddi_get8(cmd->frame_dma_obj.acc_handle,
2850             &cmd->frame->io.cmd_status);

2852         cmd->sync_cmd = MRSAS_FALSE;

2854         mutex_enter(&instance->int_cmd_mtx);
2855         if (cmd->cmd_status == ENODATA) {
2856                 cmd->cmd_status = 0;
2857         }
2858         cv_broadcast(&instance->int_cmd_cv);
2859         mutex_exit(&instance->int_cmd_mtx);

2861 }

2863 /*
2864  * mrsas_tbolt_get_ld_map_info -        Returns  ld_map structure
2865  * instance:                            Adapter soft state
2866  *
2867  * Issues an internal command (DCMD) to get the FW's controller PD
2868  * list structure.  This information is mainly used to find out SYSTEM
2869  * supported by the FW.
2870  */
2871 int
2872 mrsas_tbolt_get_ld_map_info(struct mrsas_instance *instance)
2873 {
2874         int ret = 0;
2875         struct mrsas_cmd        *cmd = NULL;
2876         struct mrsas_dcmd_frame *dcmd;
2877         MR_FW_RAID_MAP_ALL *ci;
2878         uint32_t ci_h = 0;
2879         U32 size_map_info;

2881         cmd = get_raid_msg_pkt(instance);

2883         if (cmd == NULL) {
2884                 cmn_err(CE_WARN,
2885                     "Failed to get a cmd from free-pool in get_ld_map_info()");
2886                 return (DDI_FAILURE);
2887         }

2889         dcmd = &cmd->frame->dcmd;

2891         size_map_info = sizeof (MR_FW_RAID_MAP) +
2892             (sizeof (MR_LD_SPAN_MAP) *
2893             (MAX_LOGICAL_DRIVES - 1));

2895         con_log(CL_ANN, (CE_NOTE,
2896             "size_map_info : 0x%x", size_map_info));

2898         ci = instance->ld_map[(instance->map_id & 1)];
2899         ci_h = instance->ld_map_phy[(instance->map_id & 1)];
```

```
2901         if (!ci) {
2902                 cmn_err(CE_WARN, "Failed to alloc mem for ld_map_info");
2903                 return_raid_msg_pkt(instance, cmd);
2904                 return (-1);
2905         }

2907         bzero(ci, sizeof (*ci));
2908         bzero(dcmd->mbox.b, DCMD_MBOX_SZ);

2910         dcmd->cmd = MFI_CMD_OP_DCMD;
2911         dcmd->cmd_status = 0xFF;
2912         dcmd->sge_count = 1;
2913         dcmd->flags = MFI_FRAME_DIR_READ;
2914         dcmd->timeout = 0;
2915         dcmd->pad_0 = 0;
2916         dcmd->data_xfer_len = size_map_info;
2917         dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
2918         dcmd->sgl.sge32[0].phys_addr = ci_h;
2919         dcmd->sgl.sge32[0].length = size_map_info;


2922         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

2924         if (!instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
2925                 ret = 0;
2926                 con_log(CL_ANN1, (CE_NOTE, "Get LD Map Info success"));
2927         } else {
2928                 cmn_err(CE_WARN, "Get LD Map Info failed");
2929                 ret = -1;
2930         }

2932         return_raid_msg_pkt(instance, cmd);

2934         return (ret);
2935 }

2937 void
2938 mrsas_dump_reply_desc(struct mrsas_instance *instance)
2939 {
2940         uint32_t i;
2941         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
2942         union desc_value d_val;

2944         reply_desc = instance->reply_frame_pool;

2946         for (i = 0; i < instance->reply_q_depth; i++, reply_desc++) {
2947                 d_val.word = reply_desc->Words;
2948                 con_log(CL_DLEVEL3, (CE_NOTE,
2949                     "i=%d, %x:%x",
2950                     i, d_val.u1.high, d_val.u1.low));
2951         }
2952 }

2954 /*
2955  * mrsas_tbolt_command_create - Create command for fast path.
2956  * @io_info:    MegaRAID IO request packet pointer.
2957  * @ref_tag:    Reference tag for RD/WRPROTECT
2958  *
2959  * Create the command for fast path.
2960  */
2961 void
2962 mrsas_tbolt_prepare_cdb(struct mrsas_instance *instance, U8 cdb[],
2963     struct IO_REQUEST_INFO *io_info, Mpi2RaidSCSIIORequest_t *scsi_io_request,
2964     U32 ref_tag)
2965 {
```



```
2966         uint16_t                EEDPFlags;
2967         uint32_t                Control;
2968         ddi_acc_handle_t acc_handle =
2969             instance->mpi2_frame_pool_dma_obj.acc_handle;

2971         /* Prepare 32-byte CDB if DIF is supported on this device */
2972         con_log(CL_ANN, (CE_NOTE, "Prepare DIF CDB"));

2974         bzero(cdb, 32);

2976         cdb[0] =  MRSAS_SCSI_VARIABLE_LENGTH_CMD;


2979         cdb[7] =  MRSAS_SCSI_ADDL_CDB_LEN;

2981         if (io_info->isRead)
2982                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_READ32;
2983         else
2984                 cdb[9] = MRSAS_SCSI_SERVICE_ACTION_WRITE32;

2986         /* Verify within linux driver, set to MEGASAS_RD_WR_PROTECT_CHECK_ALL */
2987         cdb[10] = MRSAS_RD_WR_PROTECT;

2989         /* LOGICAL BLOCK ADDRESS */
2990         cdb[12] = (U8)(((io_info->pdBlock) >> 56) & 0xff);
2991         cdb[13] = (U8)(((io_info->pdBlock) >> 48) & 0xff);
2992         cdb[14] = (U8)(((io_info->pdBlock) >> 40) & 0xff);
2993         cdb[15] = (U8)(((io_info->pdBlock) >> 32) & 0xff);
2994         cdb[16] = (U8)(((io_info->pdBlock) >> 24) & 0xff);
2995         cdb[17] = (U8)(((io_info->pdBlock) >> 16) & 0xff);
2996         cdb[18] = (U8)(((io_info->pdBlock) >> 8) & 0xff);
2997         cdb[19] = (U8)((io_info->pdBlock) & 0xff);

2999         /* Logical block reference tag */
3000         ddi_put32(acc_handle, &scsi_io_request->CDB.EEDP32.PrimaryReferenceTag,
3001             BE_32(ref_tag));

3003         ddi_put16(acc_handle,
3004             &scsi_io_request->CDB.EEDP32.PrimaryApplicationTagMask, 0xffff);

3006         ddi_put32(acc_handle, &scsi_io_request->DataLength,
3007             ((io_info->numBlocks)*512));
3008         /* Specify 32-byte cdb */
3009         ddi_put16(acc_handle, &scsi_io_request->IoFlags, 32);

3011         /* Transfer length */
3012         cdb[28] = (U8)(((io_info->numBlocks) >> 24) & 0xff);
3013         cdb[29] = (U8)(((io_info->numBlocks) >> 16) & 0xff);
3014         cdb[30] = (U8)(((io_info->numBlocks) >> 8) & 0xff);
3015         cdb[31] = (U8)((io_info->numBlocks) & 0xff);

3017         /* set SCSI IO EEDPFlags */
3018         EEDPFlags = ddi_get16(acc_handle, &scsi_io_request->EEDPFlags);
3019         Control = ddi_get32(acc_handle, &scsi_io_request->Control);

3021         /* set SCSI IO EEDPFlags bits */
3022         if (io_info->isRead) {
3023                 /*
3024                  * For READ commands, the EEDPFlags shall be set to specify to
3025                  * Increment the Primary Reference Tag, to Check the Reference
3026                  * Tag, and to Check and Remove the Protection Information
3027                  * fields.
3028                  */
3029                 EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3030                     MPI2_SCSIIO_EEDPFLAGS_CHECK_REFTAG  |
3031                     MPI2_SCSIIO_EEDPFLAGS_CHECK_REMOVE_OP       |
```

```
3032                    MPI2_SCSIIO_EEDPFLAGS_CHECK_APPTAG  |
3033                    MPI2_SCSIIO_EEDPFLAGS_CHECK_GUARD;
3034            } else {
3035                    /*
3036                     * For WRITE commands, the EEDPFlags shall be set to specify to
3037                     * Increment the Primary Reference Tag, and to Insert
3038                     * Protection Information fields.
3039                     */
3040                    EEDPFlags = MPI2_SCSIIO_EEDPFLAGS_INC_PRI_REFTAG        |
3041                        MPI2_SCSIIO_EEDPFLAGS_INSERT_OP;
3042            }
3043            Control |= (0x4 << 26);

3045            ddi_put16(acc_handle, &scsi_io_request->EEDPFlags, EEDPFlags);
3046            ddi_put32(acc_handle, &scsi_io_request->Control, Control);
3047            ddi_put32(acc_handle,
3048                &scsi_io_request->EEDPBlockSize, MRSAS_EEDPBLOCKSIZE);
3049    }


3052    /*
3053     * mrsas_tbolt_set_pd_lba -     Sets PD LBA
3054     * @cdb:               CDB
3055     * @cdb_len:           cdb length
3056     * @start_blk:         Start block of IO
3057     *
3058     * Used to set the PD LBA in CDB for FP IOs
3059     */
3060    static void
3061    mrsas_tbolt_set_pd_lba(U8 cdb[], uint8_t *cdb_len_ptr, U64 start_blk,
3062        U32 num_blocks)
3063    {
3064            U8 cdb_len = *cdb_len_ptr;
3065            U8 flagvals = 0, opcode = 0, groupnum = 0, control = 0;

3067            /* Some drives don't support 16/12 byte CDB's, convert to 10 */
3068            if (((cdb_len == 12) || (cdb_len == 16)) &&
3069                (start_blk <= 0xffffffff)) {
3070                    if (cdb_len == 16) {
3071                            con_log(CL_ANN,
3072                                (CE_NOTE, "Converting READ/WRITE(16) to READ10"));
3073                            opcode = cdb[0] == READ_16 ? READ_10 : WRITE_10;
3074                            flagvals = cdb[1];
3075                            groupnum = cdb[14];
3076                            control = cdb[15];
3077                    } else {
3078                            con_log(CL_ANN,
3079                                (CE_NOTE, "Converting READ/WRITE(12) to READ10"));
3080                            opcode = cdb[0] == READ_12 ? READ_10 : WRITE_10;
3081                            flagvals = cdb[1];
3082                            groupnum = cdb[10];
3083                            control = cdb[11];
3084                    }

3086                    bzero(cdb, sizeof (cdb));

3088                    cdb[0] = opcode;
3089                    cdb[1] = flagvals;
3090                    cdb[6] = groupnum;
3091                    cdb[9] = control;
3092                    /* Set transfer length */
3093                    cdb[8] = (U8)(num_blocks & 0xff);
3094                    cdb[7] = (U8)((num_blocks >> 8) & 0xff);
3095                    cdb_len = 10;
3096            } else if ((cdb_len < 16) && (start_blk > 0xffffffff)) {
3097                    /* Convert to 16 byte CDB for large LBA's */
```



```
3098                    con_log(CL_ANN,
3099                        (CE_NOTE, "Converting 6/10/12 CDB to 16 byte CDB"));
3100                    switch (cdb_len) {
3101                    case 6:
3102                            opcode = cdb[0] == READ_6 ? READ_16 : WRITE_16;
3103                            control = cdb[5];
3104                            break;
3105                    case 10:
3106                            opcode = cdb[0] == READ_10 ? READ_16 : WRITE_16;
3107                            flagvals = cdb[1];
3108                            groupnum = cdb[6];
3109                            control = cdb[9];
3110                            break;
3111                    case 12:
3112                            opcode = cdb[0] == READ_12 ? READ_16 : WRITE_16;
3113                            flagvals = cdb[1];
3114                            groupnum = cdb[10];
3115                            control = cdb[11];
3116                            break;
3117                    }

3119                    bzero(cdb, sizeof (cdb));

3121                    cdb[0] = opcode;
3122                    cdb[1] = flagvals;
3123                    cdb[14] = groupnum;
3124                    cdb[15] = control;

3126                    /* Transfer length */
3127                    cdb[13] = (U8)(num_blocks & 0xff);
3128                    cdb[12] = (U8)((num_blocks >> 8) & 0xff);
3129                    cdb[11] = (U8)((num_blocks >> 16) & 0xff);
3130                    cdb[10] = (U8)((num_blocks >> 24) & 0xff);

3132                    /* Specify 16-byte cdb */
3133                    cdb_len = 16;
3134            } else if ((cdb_len == 6) && (start_blk > 0x1fffff)) {
3135                    /* convert to 10 byte CDB */
3136                    opcode = cdb[0] == READ_6 ? READ_10 : WRITE_10;
3137                    control = cdb[5];

3139                    bzero(cdb, sizeof (cdb));
3140                    cdb[0] = opcode;
3141                    cdb[9] = control;

3143                    /* Set transfer length */
3144                    cdb[8] = (U8)(num_blocks & 0xff);
3145                    cdb[7] = (U8)((num_blocks >> 8) & 0xff);

3147                    /* Specify 10-byte cdb */
3148                    cdb_len = 10;
3149            }


3152            /* Fall through Normal case, just load LBA here */
3153            switch (cdb_len) {
3154            case 6:
3155            {
3156                    U8 val = cdb[1] & 0xE0;
3157                    cdb[3] = (U8)(start_blk & 0xff);
3158                    cdb[2] = (U8)((start_blk >> 8) & 0xff);
3159                    cdb[1] = val | ((U8)(start_blk >> 16) & 0x1f);
3160                    break;
3161            }
3162            case 10:
3163                    cdb[5] = (U8)(start_blk & 0xff);
```

```
3164                 cdb[4] = (U8)((start_blk >> 8) & 0xff);
3165                 cdb[3] = (U8)((start_blk >> 16) & 0xff);
3166                 cdb[2] = (U8)((start_blk >> 24) & 0xff);
3167                 break;
3168         case 12:
3169                 cdb[5]    = (U8)(start_blk & 0xff);
3170                 cdb[4]    = (U8)((start_blk >> 8) & 0xff);
3171                 cdb[3]    = (U8)((start_blk >> 16) & 0xff);
3172                 cdb[2]    = (U8)((start_blk >> 24) & 0xff);
3173                 break;

3175         case 16:
3176                 cdb[9]  = (U8)(start_blk & 0xff);
3177                 cdb[8]  = (U8)((start_blk >> 8) & 0xff);
3178                 cdb[7]  = (U8)((start_blk >> 16) & 0xff);
3179                 cdb[6]  = (U8)((start_blk >> 24) & 0xff);
3180                 cdb[5]  = (U8)((start_blk >> 32) & 0xff);
3181                 cdb[4]  = (U8)((start_blk >> 40) & 0xff);
3182                 cdb[3]  = (U8)((start_blk >> 48) & 0xff);
3183                 cdb[2]  = (U8)((start_blk >> 56) & 0xff);
3184                 break;
3185         }

3187         *cdb_len_ptr = cdb_len;
3188 }


3191 static int
3192 mrsas_tbolt_check_map_info(struct mrsas_instance *instance)
3193 {
3194         MR_FW_RAID_MAP_ALL *ld_map;

3196         if (!mrsas_tbolt_get_ld_map_info(instance)) {

3198                 ld_map = instance->ld_map[(instance->map_id & 1)];

3200                 con_log(CL_ANN1, (CE_NOTE, "ldCount=%d, map size=%d",
3201                     ld_map->raidMap.ldCount, ld_map->raidMap.totalSize));

3203                 if (MR_ValidateMapInfo(instance->ld_map[
3204                     (instance->map_id & 1)], instance->load_balance_info)) {
3205                         con_log(CL_ANN,
3206                             (CE_CONT, "MR_ValidateMapInfo success"));

3208                         instance->fast_path_io = 1;
3209                         con_log(CL_ANN,
3210                             (CE_NOTE, "instance->fast_path_io %d",
3211                             instance->fast_path_io));

3213                         return (DDI_SUCCESS);
3214                 }

3216         }

3218         instance->fast_path_io = 0;
3219         cmn_err(CE_WARN, "MR_ValidateMapInfo failed");
3220         con_log(CL_ANN, (CE_NOTE,
3221             "instance->fast_path_io %d", instance->fast_path_io));

3223         return (DDI_FAILURE);
3224 }

3226 /*
3227  * Marks HBA as bad. This will be called either when an
3228  * IO packet times out even after 3 FW resets
3229  * or FW is found to be fault even after 3 continuous resets.
```



```
3230  */

3232 void
3233 mrsas_tbolt_kill_adapter(struct mrsas_instance *instance)
3234 {
3235         cmn_err(CE_NOTE, "TBOLT Kill adapter called");

3237         if (instance->deadadapter == 1)
3238                 return;

3240         con_log(CL_ANN1, (CE_NOTE, "tbolt_kill_adapter: "
3241             "Writing to doorbell with MFI_STOP_ADP "));
3242         mutex_enter(&instance->ocr_flags_mtx);
3243         instance->deadadapter = 1;
3244         mutex_exit(&instance->ocr_flags_mtx);
3245         instance->func_ptr->disable_intr(instance);
3246         WR_RESERVED0_REGISTER(MFI_STOP_ADP, instance);
3247         /* Flush */
3248         (void) RD_RESERVED0_REGISTER(instance);

3250         (void) mrsas_print_pending_cmds(instance);
3251         (void) mrsas_complete_pending_cmds(instance);
3252 }

3254 void
3255 mrsas_reset_reply_desc(struct mrsas_instance *instance)
3256 {
3257         int i;
3258         MPI2_REPLY_DESCRIPTORS_UNION *reply_desc;
3259         instance->reply_read_index = 0;

3261         /* initializing reply address to 0xFFFFFFFF */
3262         reply_desc = instance->reply_frame_pool;

3264         for (i = 0; i < instance->reply_q_depth; i++) {
3265                 reply_desc->Words = (uint64_t)~0;
3266                 reply_desc++;
3267         }
3268 }

3270 int
3271 mrsas_tbolt_reset_ppc(struct mrsas_instance *instance)
3272 {
3273         uint32_t status = 0x00;
3274         uint32_t retry = 0;
3275         uint32_t cur_abs_reg_val;
3276         uint32_t fw_state;
3277         uint32_t abs_state;
3278         uint32_t i;

3280         con_log(CL_ANN, (CE_NOTE,
3281             "mrsas_tbolt_reset_ppc entered"));

3283         if (instance->deadadapter == 1) {
3284                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3285                     "no more resets as HBA has been marked dead ");
3286                 return (DDI_FAILURE);
3287         }

3289         mutex_enter(&instance->ocr_flags_mtx);
3290         instance->adapterresetinprogress = 1;
3291         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc:"
3292             "adpterresetinprogress flag set, time %llx", gethrtime()));
3293         mutex_exit(&instance->ocr_flags_mtx);

3295         instance->func_ptr->disable_intr(instance);
```

```
3297         /* Add delay inorder to complete the ioctl & io cmds in-flight */
3298         for (i = 0; i < 3000; i++) {
3299                 drv_usecwait(MILLISEC); /* wait for 1000 usecs */
3300         }

3302         instance->reply_read_index = 0;

3304 retry_reset:
3305         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3306             ":Resetting TBOLT "));

3308         WR_TBOLT_IB_WRITE_SEQ(0xF, instance);
3309         WR_TBOLT_IB_WRITE_SEQ(4, instance);
3310         WR_TBOLT_IB_WRITE_SEQ(0xb, instance);
3311         WR_TBOLT_IB_WRITE_SEQ(2, instance);
3312         WR_TBOLT_IB_WRITE_SEQ(7, instance);
3313         WR_TBOLT_IB_WRITE_SEQ(0xd, instance);
3314         con_log(CL_ANN1, (CE_NOTE,
3315             "mrsas_tbolt_reset_ppc: magic number written "
3316             "to write sequence register"));
3317         delay(100 * drv_usectohz(MILLISEC));
3318         status = RD_TBOLT_HOST_DIAG(instance);
3319         con_log(CL_ANN1, (CE_NOTE,
3320             "mrsas_tbolt_reset_ppc: READ HOSTDIAG SUCCESS "
3321             "to write sequence register"));

3323         while (status & DIAG_TBOLT_RESET_ADAPTER) {
3324                 delay(100 * drv_usectohz(MILLISEC));
3325                 status = RD_TBOLT_HOST_DIAG(instance);
3326                 if (retry++ == 100) {
3327                         cmn_err(CE_WARN,
3328                             "mrsas_tbolt_reset_ppc:"
3329                             "resetadapter bit is set already "
3330                             "check retry count %d", retry);
3331                         return (DDI_FAILURE);
3332                 }
3333         }

3335         WR_TBOLT_HOST_DIAG(status | DIAG_TBOLT_RESET_ADAPTER, instance);
3336         delay(100 * drv_usectohz(MILLISEC));

3338         ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3339             (uint8_t *)((uintptr_t)(instance)->regmap +
3340             RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);

3342         while ((status & DIAG_TBOLT_RESET_ADAPTER)) {
3343                 delay(100 * drv_usectohz(MILLISEC));
3344                 ddi_rep_get8((instance)->regmap_handle, (uint8_t *)&status,
3345                     (uint8_t *)((uintptr_t)(instance)->regmap +
3346                     RESET_TBOLT_STATUS_OFF), 4, DDI_DEV_AUTOINCR);
3347                 if (retry++ == 100) {
3348                         /* Dont call kill adapter here */
3349                         /* RESET BIT ADAPTER is cleared by firmare */
3350                         /* mrsas_tbolt_kill_adapter(instance); */
3351                         cmn_err(CE_WARN,
3352                             "mr_sas %d: %s(): RESET FAILED; return failure!!!",
3353                             instance->instance, __func__);
3354                         return (DDI_FAILURE);
3355                 }
3356         }

3358         con_log(CL_ANN,
3359             (CE_NOTE, "mrsas_tbolt_reset_ppc: Adapter reset complete"));
3360         con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3361             "Calling mfi_state_transition_to_ready"));
```



```
3363         abs_state = instance->func_ptr->read_fw_status_reg(instance);
3364         retry = 0;
3365         while ((abs_state <= MFI_STATE_FW_INIT) && (retry++ < 1000)) {
3366                 delay(100 * drv_usectohz(MILLISEC));
3367                 abs_state = instance->func_ptr->read_fw_status_reg(instance);
3368         }
3369         if (abs_state <= MFI_STATE_FW_INIT) {
3370                 cmn_err(CE_WARN,
3371                     "mrsas_tbolt_reset_ppc: firmware state < MFI_STATE_FW_INIT"
3372                     "state = 0x%x, RETRY RESET.", abs_state);
3373                 goto retry_reset;
3374         }

3376         /* Mark HBA as bad, if FW is fault after 3 continuous resets */
3377         if (mfi_state_transition_to_ready(instance) ||
3378             debug_tbolt_fw_faults_after_ocr_g == 1) {
3379                 cur_abs_reg_val =
3380                     instance->func_ptr->read_fw_status_reg(instance);
3381                 fw_state        = cur_abs_reg_val & MFI_STATE_MASK;

3383                 con_log(CL_ANN1, (CE_NOTE,
3384                     "mrsas_tbolt_reset_ppc :before fake: FW is not ready "
3385                     "FW state = 0x%x", fw_state));
3386                 if (debug_tbolt_fw_faults_after_ocr_g == 1)
3387                         fw_state = MFI_STATE_FAULT;

3389                 con_log(CL_ANN,
3390                     (CE_NOTE,  "mrsas_tbolt_reset_ppc : FW is not ready "
3391                     "FW state = 0x%x", fw_state));

3393                 if (fw_state == MFI_STATE_FAULT) {
3394                         /* increment the count */
3395                         instance->fw_fault_count_after_ocr++;
3396                         if (instance->fw_fault_count_after_ocr
3397                             < MAX_FW_RESET_COUNT) {
3398                                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
3399                                     "FW is in fault after OCR count %d "
3400                                     "Retry Reset",
3401                                     instance->fw_fault_count_after_ocr);
3402                                 goto retry_reset;

3404                         } else {
3405                                 cmn_err(CE_WARN, "mrsas %d: %s:"
3406                                     "Max Reset Count exceeded >%d"
3407                                     "Mark HBA as bad, KILL adapter",
3408                                     instance->instance, __func__,
3409                                     MAX_FW_RESET_COUNT);

3411                                 mrsas_tbolt_kill_adapter(instance);
3412                                 return (DDI_FAILURE);
3413                         }
3414                 }
3415         }

3417         /* reset the counter as FW is up after OCR */
3418         instance->fw_fault_count_after_ocr = 0;

3420         mrsas_reset_reply_desc(instance);


3423         con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3424             "Calling mrsas_issue_init_mpi2"));
3425         abs_state = mrsas_issue_init_mpi2(instance);
3426         if (abs_state == (uint32_t)DDI_FAILURE) {
3427                 cmn_err(CE_WARN, "mrsas_tbolt_reset_ppc: "
```

```
3428                    "INIT failed Retrying Reset");
3429                goto retry_reset;
3430        }
3431        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3432            "mrsas_issue_init_mpi2 Done"));

3434        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3435            "Calling mrsas_print_pending_cmd"));
3436        (void) mrsas_print_pending_cmds(instance);
3437        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3438            "mrsas_print_pending_cmd done"));

3440        instance->func_ptr->enable_intr(instance);
3441        instance->fw_outstanding = 0;

3443        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3444            "Calling mrsas_issue_pending_cmds"));
3445        (void) mrsas_issue_pending_cmds(instance);
3446        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3447        "issue_pending_cmds done."));

3449        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3450            "Calling aen registration"));

3452        instance->aen_cmd->retry_count_for_ocr = 0;
3453        instance->aen_cmd->drv_pkt_time = 0;

3455        instance->func_ptr->issue_cmd(instance->aen_cmd, instance);

3457        con_log(CL_ANN1, (CE_NOTE, "Unsetting adpresetinprogress flag."));
3458        mutex_enter(&instance->ocr_flags_mtx);
3459        instance->adapterresetinprogress = 0;
3460        mutex_exit(&instance->ocr_flags_mtx);
3461        con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_reset_ppc: "
3462            "adpterresetinprogress flag unset"));

3464        con_log(CL_ANN, (CE_NOTE, "mrsas_tbolt_reset_ppc done"));
3465        return (DDI_SUCCESS);

3467 }


3470 /*
3471  * mrsas_sync_map_info -        Returns FW's ld_map structure
3472  * @instance:                           Adapter soft state
3473  *
3474  * Issues an internal command (DCMD) to get the FW's controller PD
3475  * list structure.  This information is mainly used to find out SYSTEM
3476  * supported by the FW.
3477  */

3479 static int
3480 mrsas_tbolt_sync_map_info(struct mrsas_instance *instance)
3481 {
3482        int                     ret = 0, i;
3483        struct mrsas_cmd        *cmd = NULL;
3484        struct mrsas_dcmd_frame *dcmd;
3485        uint32_t size_sync_info, num_lds;
3486        LD_TARGET_SYNC *ci = NULL;
3487        MR_FW_RAID_MAP_ALL *map;
3488        MR_LD_RAID  *raid;
3489        LD_TARGET_SYNC *ld_sync;
3490        uint32_t ci_h = 0;
3491        uint32_t size_map_info;

3493        cmd = get_raid_msg_pkt(instance);
```



```
3495        if (cmd == NULL) {
3496                cmn_err(CE_WARN, "Failed to get a cmd from free-pool in "
3497                    "mrsas_tbolt_sync_map_info(). ");
3498                return (DDI_FAILURE);
3499        }

3501        /* Clear the frame buffer and assign back the context id */
3502        bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3503        ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3504            cmd->index);
3505        bzero(cmd->scsi_io_request, sizeof (Mpi2RaidSCSIIORequest_t));


3508        map = instance->ld_map[instance->map_id & 1];

3510        num_lds = map->raidMap.ldCount;

3512        dcmd = &cmd->frame->dcmd;

3514        size_sync_info = sizeof (LD_TARGET_SYNC) * num_lds;

3516        con_log(CL_ANN, (CE_NOTE, "size_sync_info =0x%x ; ld count = 0x%x",
3517            size_sync_info, num_lds));

3519        ci = (LD_TARGET_SYNC *)instance->ld_map[(instance->map_id - 1) & 1];

3521        bzero(ci, sizeof (MR_FW_RAID_MAP_ALL));
3522        ci_h = instance->ld_map_phy[(instance->map_id - 1) & 1];

3524        bzero(dcmd->mbox.b, DCMD_MBOX_SZ);

3526        ld_sync = (LD_TARGET_SYNC *)ci;

3528        for (i = 0; i < num_lds; i++, ld_sync++) {
3529                raid = MR_LdRaidGet(i, map);

3531                con_log(CL_ANN1,
3532                    (CE_NOTE, "i : 0x%x, Seq Num : 0x%x, Sync Reqd : 0x%x",
3533                    i, raid->seqNum, raid->flags.ldSyncRequired));

3535                ld_sync->ldTargetId = MR_GetLDTgtId(i, map);

3537                con_log(CL_ANN1, (CE_NOTE, "i : 0x%x, tgt : 0x%x",
3538                    i, ld_sync->ldTargetId));

3540                ld_sync->seqNum = raid->seqNum;
3541        }


3544        size_map_info = sizeof (MR_FW_RAID_MAP) +
3545            (sizeof (MR_LD_SPAN_MAP) * (MAX_LOGICAL_DRIVES - 1));

3547        dcmd->cmd = MFI_CMD_OP_DCMD;
3548        dcmd->cmd_status = 0xFF;
3549        dcmd->sge_count = 1;
3550        dcmd->flags = MFI_FRAME_DIR_WRITE;
3551        dcmd->timeout = 0;
3552        dcmd->pad_0 = 0;
3553        dcmd->data_xfer_len = size_map_info;
3554        ASSERT(num_lds <= 255);
3555        dcmd->mbox.b[0] = (U8)num_lds;
3556        dcmd->mbox.b[1] = 1; /* Pend */
3557        dcmd->opcode = MR_DCMD_LD_MAP_GET_INFO;
3558        dcmd->sgl.sge32[0].phys_addr = ci_h;
3559        dcmd->sgl.sge32[0].length = size_map_info;
```

```
3562         instance->map_update_cmd = cmd;
3563         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

3565         instance->func_ptr->issue_cmd(cmd, instance);

3567         instance->unroll.syncCmd = 1;
3568         con_log(CL_ANN1, (CE_NOTE, "sync cmd issued. [SMID]:%x", cmd->SMID));

3570         return (ret);
3571 }

3573 /*
3574  * abort_syncmap_cmd
3575  */
3576 int
3577 abort_syncmap_cmd(struct mrsas_instance *instance,
3578     struct mrsas_cmd *cmd_to_abort)
3579 {
3580         int     ret = 0;

3582         struct mrsas_cmd                *cmd;
3583         struct mrsas_abort_frame        *abort_fr;

3585         con_log(CL_ANN1, (CE_NOTE, "chkpnt: abort_ldsync:%d", __LINE__));

3587         cmd = get_raid_msg_mfi_pkt(instance);

3589         if (!cmd) {
3590                 cmn_err(CE_WARN,
3591                     "Failed to get a cmd from free-pool abort_syncmap_cmd().");
3592                 return (DDI_FAILURE);
3593         }
3594         /* Clear the frame buffer and assign back the context id */
3595         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3596         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3597             cmd->index);

3599         abort_fr = &cmd->frame->abort;

3601         /* prepare and issue the abort frame */
3602         ddi_put8(cmd->frame_dma_obj.acc_handle,
3603             &abort_fr->cmd, MFI_CMD_OP_ABORT);
3604         ddi_put8(cmd->frame_dma_obj.acc_handle, &abort_fr->cmd_status,
3605             MFI_CMD_STATUS_SYNC_MODE);
3606         ddi_put16(cmd->frame_dma_obj.acc_handle, &abort_fr->flags, 0);
3607         ddi_put32(cmd->frame_dma_obj.acc_handle, &abort_fr->abort_context,
3608             cmd_to_abort->index);
3609         ddi_put32(cmd->frame_dma_obj.acc_handle,
3610             &abort_fr->abort_mfi_phys_addr_lo, cmd_to_abort->frame_phys_addr);
3611         ddi_put32(cmd->frame_dma_obj.acc_handle,
3612             &abort_fr->abort_mfi_phys_addr_hi, 0);

3614         cmd->frame_count = 1;

3616         mr_sas_tbolt_build_mfi_cmd(instance, cmd);

3618         if (instance->func_ptr->issue_cmd_in_poll_mode(instance, cmd)) {
3619                 con_log(CL_ANN1, (CE_WARN,
3620                     "abort_ldsync_cmd: issue_cmd_in_poll_mode failed"));
3621                 ret = -1;
3622         } else {
3623                 ret = 0;
3624         }
```



```
3626         return_raid_msg_mfi_pkt(instance, cmd);

3628         atomic_add_16(&instance->fw_outstanding, (-1));

3630         return (ret);
3631 }


3634 #ifdef PDSUPPORT
3635 int
3636 mrsas_tbolt_config_pd(struct mrsas_instance *instance, uint16_t tgt,
3637     uint8_t lun, dev_info_t **ldip)
3638 {
3639         struct scsi_device *sd;
3640         dev_info_t *child;
3641         int rval, dtype;
3642         struct mrsas_tbolt_pd_info *pds = NULL;

3644         con_log(CL_ANN1, (CE_NOTE, "mrsas_tbolt_config_pd: t = %d l = %d",
3645             tgt, lun));

3647         if ((child = mrsas_find_child(instance, tgt, lun)) != NULL) {
3648                 if (ldip) {
3649                         *ldip = child;
3650                 }
3651                 if (instance->mr_tbolt_pd_list[tgt].flag != MRDRV_TGT_VALID) {
3652                         rval = mrsas_service_evt(instance, tgt, 1,
3653                             MRSAS_EVT_UNCONFIG_TGT, NULL);
3654                         con_log(CL_ANN1, (CE_WARN,
3655                             "mr_sas:DELETING STALE ENTRY  rval = %d "
3656                             "tgt id = %d", rval, tgt));
3657                         return (NDI_FAILURE);
3658                 }
3659                 return (NDI_SUCCESS);
3660         }

3662         pds = (struct mrsas_tbolt_pd_info *)
3663             kmem_zalloc(sizeof (struct mrsas_tbolt_pd_info), KM_SLEEP);
3664         mrsas_tbolt_get_pd_info(instance, pds, tgt);
3665         dtype = pds->scsiDevType;

3667         /* Check for Disk */
3668         if ((dtype == DTYPE_DIRECT)) {
3669                 if ((dtype == DTYPE_DIRECT) &&
3670                     (LE_16(pds->fwState) != PD_SYSTEM)) {
3671                         kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3672                         return (NDI_FAILURE);
3673                 }
3674                 sd = kmem_zalloc(sizeof (struct scsi_device), KM_SLEEP);
3675                 sd->sd_address.a_hba_tran = instance->tran;
3676                 sd->sd_address.a_target = (uint16_t)tgt;
3677                 sd->sd_address.a_lun = (uint8_t)lun;

3679                 if (scsi_hba_probe(sd, NULL) == SCSIPROBE_EXISTS) {
3680                         rval = mrsas_config_scsi_device(instance, sd, ldip);
3681                         con_log(CL_DLEVEL1, (CE_NOTE,
3682                             "Phys. device found: tgt %d dtype %d: %s",
3683                             tgt, dtype, sd->sd_inq->inq_vid));
3684                 } else {
3685                         rval = NDI_FAILURE;
3686                         con_log(CL_DLEVEL1, (CE_NOTE, "Phys. device Not found "
3687                             "scsi_hba_probe Failed: tgt %d dtype %d: %s",
3688                             tgt, dtype, sd->sd_inq->inq_vid));
3689                 }

3691                 /* sd_unprobe is blank now. Free buffer manually */
```

```
3692                 if (sd->sd_inq) {
3693                         kmem_free(sd->sd_inq, SUN_INQSIZE);
3694                         sd->sd_inq = (struct scsi_inquiry *)NULL;
3695                 }
3696                 kmem_free(sd, sizeof (struct scsi_device));
3697                 rval = NDI_SUCCESS;
3698         } else {
3699                 con_log(CL_ANN1, (CE_NOTE,
3700                     "Device not supported: tgt %d lun %d dtype %d",
3701                     tgt, lun, dtype));
3702                 rval = NDI_FAILURE;
3703         }

3705         kmem_free(pds, sizeof (struct mrsas_tbolt_pd_info));
3706         con_log(CL_ANN1, (CE_NOTE, "mrsas_config_pd: return rval = %d",
3707             rval));
3708         return (rval);
3709 }

3711 static void
3712 mrsas_tbolt_get_pd_info(struct mrsas_instance *instance,
3713     struct mrsas_tbolt_pd_info *pds, int tgt)
3714 {
3715         struct mrsas_cmd        *cmd;
3716         struct mrsas_dcmd_frame *dcmd;
3717         dma_obj_t               dcmd_dma_obj;

3719         cmd = get_raid_msg_pkt(instance);

3721         if (!cmd) {
3722                 con_log(CL_ANN1,
3723                     (CE_WARN, "Failed to get a cmd for get pd info"));
3724                 return;
3725         }

3727         /* Clear the frame buffer and assign back the context id */
3728         bzero((char *)&cmd->frame[0], sizeof (union mrsas_frame));
3729         ddi_put32(cmd->frame_dma_obj.acc_handle, &cmd->frame->hdr.context,
3730             cmd->index);


3733         dcmd = &cmd->frame->dcmd;
3734         dcmd_dma_obj.size = sizeof (struct mrsas_tbolt_pd_info);
3735         dcmd_dma_obj.dma_attr = mrsas_generic_dma_attr;
3736         dcmd_dma_obj.dma_attr.dma_attr_addr_hi = 0xffffffff;
3737         dcmd_dma_obj.dma_attr.dma_attr_count_max = 0xffffffff;
3738         dcmd_dma_obj.dma_attr.dma_attr_sgllen = 1;
3739         dcmd_dma_obj.dma_attr.dma_attr_align = 1;

3741         (void) mrsas_alloc_dma_obj(instance, &dcmd_dma_obj,
3742             DDI_STRUCTURE_LE_ACC);
3743         bzero(dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info));
3744         bzero(dcmd->mbox.b, 12);
3745         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd, MFI_CMD_OP_DCMD);
3746         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->cmd_status, 0);
3747         ddi_put8(cmd->frame_dma_obj.acc_handle, &dcmd->sge_count, 1);
3748         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->flags,
3749             MFI_FRAME_DIR_READ);
3750         ddi_put16(cmd->frame_dma_obj.acc_handle, &dcmd->timeout, 0);
3751         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->data_xfer_len,
3752             sizeof (struct mrsas_tbolt_pd_info));
3753         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->opcode,
3754             MR_DCMD_PD_GET_INFO);
3755         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->mbox.w[0], tgt);
3756         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].length,
3757             sizeof (struct mrsas_tbolt_pd_info));
```



```
3758         ddi_put32(cmd->frame_dma_obj.acc_handle, &dcmd->sgl.sge32[0].phys_addr,
3759             dcmd_dma_obj.dma_cookie[0].dmac_address);

3761         cmd->sync_cmd = MRSAS_TRUE;
3762         cmd->frame_count = 1;

3764         if (instance->tbolt) {
3765                 mr_sas_tbolt_build_mfi_cmd(instance, cmd);
3766         }

3768         instance->func_ptr->issue_cmd_in_sync_mode(instance, cmd);

3770         ddi_rep_get8(cmd->frame_dma_obj.acc_handle, (uint8_t *)pds,
3771             (uint8_t *)dcmd_dma_obj.buffer, sizeof (struct mrsas_tbolt_pd_info),
3772             DDI_DEV_AUTOINCR);
3773         (void) mrsas_free_dma_obj(instance, dcmd_dma_obj);
3774         return_raid_msg_pkt(instance, cmd);
3775 }
3776 #endif
```