```
**********************************************************
    2008 Thu Jan 16 19:58:19 2014
new/usr/src/man/man7d/nv_sata.7d
1023 nv_sata support for NVIDIA MCP61
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Dan McDonald <danmcd@omniti.com>
**********************************************************
```

```
'\" te
.\" Copyright (c) 2008, Sun Microsystems, Inc.  All Rights Reserved
.\" Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
.\" The contents of this file are subject to the terms of the Common Development
.\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
.\" When distributing Covered Code, include this CDDL HEADER in each file and in
.TH nv_sata 7D "25 Sep 2011"
.TH NV_SATA 7D "Jul 22, 2008"
.SH NAME
nv_sata \- NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver
nv_sata \- Nvidia ck804/mcp55 SATA controller driver
.SH SYNOPSIS
.LP
.nf
\fBsata@unit-address\fR
.fi

.SH DESCRIPTION
.sp
.LP
The \fBnv_sata\fR driver is a SATA HBA driver that supports NVIDIA CK804/MCP04
and MCP51/MCP55/MCP61 SATA HBA controllers. Note that while these controllers
The \fBnv_sata\fR driver is a SATA HBA driver that supports Nvidia ck804 and
mcp55 SATA HBA controllers.      Note that while these Nvidia controllers
support standard SATA features including SATA-II drives, NCQ, hotplug and ATAPI
drives, the driver currently does not support NCQ features.
.SH CONFIGURATION
.sp
.LP
The \fBnv_sata\fR module contains no user configurable parameters.
.SH FILES
.sp
.ne 2
.na
\fB\fB/kernel/drv/nv_sata\fR\fR
.ad
.sp .6
.RS 4n
32-bit \fBELF\fR kernel module (x86).
.RE

.sp
.ne 2
.na
\fB\fB/kernel/drv/amd64/nv_sata\fR\fR
.ad
.sp .6
.RS 4n
64-bit \fBELF\fR kernel module (x86).
.RE

.SH ATTRIBUTES
.sp
.LP
See \fBattributes\fR(5) for descriptions of the following attribute:
.sp

.sp
.TS
```



```
box;
c | c
l | l .
ATTRIBUTE TYPE  ATTRIBUTE VALUE
_
Architecture    x86
.TE

.SH SEE ALSO
.sp
.LP
\fBcfgadm\fR(1M), \fBcfgadm_sata\fR(1M), \fBprtconf\fR(1M), \fBsata\fR(7D),
\fBsd\fR(7D)
.sp
.LP
\fIWriting Device Drivers\fR
```

```
**********************************************************
    2374 Thu Jan 16 19:58:19 2014
new/usr/src/pkg/manifests/driver-storage-nv_sata.mf
1023 nv_sata support for NVIDIA MCP61
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Dan McDonald <danmcd@omniti.com>
**********************************************************
#
# 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 2011 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/nv_sata@$(PKGVERS)
set name=pkg.description \
    value="NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver"
set name=pkg.summary \
    value="NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver"
    value="Nvidia ck804 pro / mcp55 pro combo SATA driver"
set name=pkg.summary value="Nvidia ck804 pro / mcp55 pro combo SATA driver"
set name=info.classification value=org.opensolaris.category.2008:Drivers/Ports
set name=variant.arch value=i386
dir path=kernel group=sys
dir path=kernel/drv group=sys
dir path=kernel/drv/$(ARCH64) group=sys
dir path=usr/share/man
dir path=usr/share/man/man7d
driver name=nv_sata class=scsi-self-identifying perms="* 0644 root sys" \
    alias=pci10de,266 \
    alias=pci10de,267 \
    alias=pci10de,36 \
    alias=pci10de,37e \
    alias=pci10de,37f \
    alias=pci10de,3e \
    alias=pci10de,3f6 \
    alias=pci10de,3f7 \
    alias=pci10de,54 \
    alias=pci10de,55
file path=kernel/drv/$(ARCH64)/nv_sata group=sys
file path=kernel/drv/nv_sata group=sys
```



```
file path=kernel/drv/nv_sata.conf group=sys
file path=usr/share/man/man7d/nv_sata.7d
legacy pkg=SUNWnvsata desc="Nvidia ck804 pro / mcp55 pro combo SATA driver" \
    name="Nvidia ck804 pro / mcp55 pro combo SATA driver"
license cr_Sun license=cr_Sun
license lic_CDDL license=lic_CDDL
```

```
**********************************************************
  180646 Thu Jan 16 19:58:19 2014
new/usr/src/uts/common/io/sata/adapters/nv_sata/nv_sata.c
1023 nv_sata support for NVIDIA MCP61
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Dan McDonald <danmcd@omniti.com>
**********************************************************
```

```
/*
 * 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) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 *
 * nv_sata is a combo SATA HBA driver for CK804/MCP04 (ck804) and
 * MCP55/MCP51/MCP61 (mcp5x) based chipsets.
 * nv_sata is a combo SATA HBA driver for ck804/mcp5x (mcp5x = mcp55/mcp51)
 * based chipsets.
 *
 * NCQ
 * ---
 *
 * A portion of the NCQ is in place, but is incomplete.  NCQ is disabled
 * and is likely to be revisited in the future.
 *
 *
 * Power Management
 * ----------------
 *
 * Normally power management would be responsible for ensuring the device
 * is quiescent and then changing power states to the device, such as
 * powering down parts or all of the device.  mcp5x/ck804 is unique in
 * that it is only available as part of a larger southbridge chipset, so
 * removing power to the device isn't possible.  Switches to control
 * power management states D0/D3 in the PCI configuration space appear to
 * be supported but changes to these states are apparently are ignored.
 * The only further PM that the driver _could_ do is shut down the PHY,
 * but in order to deliver the first rev of the driver sooner than later,
 * that will be deferred until some future phase.
 *
 * Since the driver currently will not directly change any power state to
 * the device, no power() entry point will be required.  However, it is
 * possible that in ACPI power state S3, aka suspend to RAM, that power
 * can be removed to the device, and the driver cannot rely on BIOS to
 * have reset any state.  For the time being, there is no known
```



```
 * non-default configurations that need to be programmed.  This judgement
 * is based on the port of the legacy ata driver not having any such
 * functionality and based on conversations with the PM team.  If such a
 * restoration is later deemed necessary it can be incorporated into the
 * DDI_RESUME processing.
 *
 */

#include <sys/scsi/scsi.h>
#include <sys/pci.h>
#include <sys/byteorder.h>
#include <sys/sunddi.h>
#include <sys/sata/sata_hba.h>
#ifdef SGPIO_SUPPORT
#include <sys/sata/adapters/nv_sata/nv_sgpio.h>
#include <sys/devctl.h>
#include <sys/sdt.h>
#endif
#include <sys/sata/adapters/nv_sata/nv_sata.h>
#include <sys/disp.h>
#include <sys/note.h>
#include <sys/promif.h>


/*
 * Function prototypes for driver entry points
 */
static int nv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int nv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int nv_quiesce(dev_info_t *dip);
static int nv_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd,
    void *arg, void **result);

/*
 * Function prototypes for entry points from sata service module
 * These functions are distinguished from other local functions
 * by the prefix "nv_sata_"
 */
static int nv_sata_start(dev_info_t *dip, sata_pkt_t *spkt);
static int nv_sata_abort(dev_info_t *dip, sata_pkt_t *spkt, int);
static int nv_sata_reset(dev_info_t *dip, sata_device_t *sd);
static int nv_sata_activate(dev_info_t *dip, sata_device_t *sd);
static int nv_sata_deactivate(dev_info_t *dip, sata_device_t *sd);

/*
 * Local function prototypes
 */
static uint_t mcp5x_intr(caddr_t arg1, caddr_t arg2);
static uint_t ck804_intr(caddr_t arg1, caddr_t arg2);
static int nv_add_legacy_intrs(nv_ctl_t *nvc);
#ifdef NV_MSI_SUPPORTED
static int nv_add_msi_intrs(nv_ctl_t *nvc);
#endif
static void nv_rem_intrs(nv_ctl_t *nvc);
static int nv_start_common(nv_port_t *nvp, sata_pkt_t *spkt);
static int nv_start_nodata(nv_port_t *nvp, int slot);
static void nv_intr_nodata(nv_port_t *nvp, nv_slot_t *spkt);
static int nv_start_pio_in(nv_port_t *nvp, int slot);
static int nv_start_pio_out(nv_port_t *nvp, int slot);
static void nv_intr_pio_in(nv_port_t *nvp, nv_slot_t *spkt);
static void nv_intr_pio_out(nv_port_t *nvp, nv_slot_t *spkt);
static int nv_start_pkt_pio(nv_port_t *nvp, int slot);
static void nv_intr_pkt_pio(nv_port_t *nvp, nv_slot_t *nv_slotp);
static int nv_start_dma(nv_port_t *nvp, int slot);
static void nv_intr_dma(nv_port_t *nvp, struct nv_slot *spkt);
static void nv_uninit_ctl(nv_ctl_t *nvc);
```

```
124 static void mcp5x_reg_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle);
125 static void ck804_reg_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle);
126 static void nv_uninit_port(nv_port_t *nvp);
127 static void nv_init_port(nv_port_t *nvp);
128 static int nv_init_ctl(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle);
129 static int mcp5x_packet_complete_intr(nv_ctl_t *nvc, nv_port_t *nvp);
130 #ifdef NCQ
131 static int mcp5x_dma_setup_intr(nv_ctl_t *nvc, nv_port_t *nvp);
132 #endif
133 static void nv_start_dma_engine(nv_port_t *nvp, int slot);
134 static void nv_port_state_change(nv_port_t *nvp, int event, uint8_t addr_type,
135     int state);
136 static void nv_common_reg_init(nv_ctl_t *nvc);
137 static void ck804_intr_process(nv_ctl_t *nvc, uint8_t intr_status);
138 static void nv_reset(nv_port_t *nvp, char *reason);
139 static void nv_complete_io(nv_port_t *nvp,  sata_pkt_t *spkt, int slot);
140 static void nv_timeout(void *);
141 static int nv_poll_wait(nv_port_t *nvp, sata_pkt_t *spkt);
142 static void nv_cmn_err(int ce, nv_ctl_t *nvc, nv_port_t *nvp, char *fmt, ...);
143 static void nv_read_signature(nv_port_t *nvp);
144 static void mcp5x_set_intr(nv_port_t *nvp, int flag);
145 static void ck804_set_intr(nv_port_t *nvp, int flag);
146 static void nv_resume(nv_port_t *nvp);
147 static void nv_suspend(nv_port_t *nvp);
148 static int nv_start_sync(nv_port_t *nvp, sata_pkt_t *spkt);
149 static int nv_abort_active(nv_port_t *nvp, sata_pkt_t *spkt, int abort_reason,
150     boolean_t reset);
151 static void nv_copy_registers(nv_port_t *nvp, sata_device_t *sd,
152     sata_pkt_t *spkt);
153 static void nv_link_event(nv_port_t *nvp, int flags);
154 static int nv_start_async(nv_port_t *nvp, sata_pkt_t *spkt);
155 static int nv_wait3(nv_port_t *nvp, uchar_t onbits1, uchar_t offbits1,
156     uchar_t failure_onbits2, uchar_t failure_offbits2,
157     uchar_t failure_onbits3, uchar_t failure_offbits3,
158     uint_t timeout_usec, int type_wait);
159 static int nv_wait(nv_port_t *nvp, uchar_t onbits, uchar_t offbits,
160     uint_t timeout_usec, int type_wait);
161 static int nv_start_rqsense_pio(nv_port_t *nvp, nv_slot_t *nv_slotp);
162 static void nv_setup_timeout(nv_port_t *nvp, clock_t microseconds);
163 static clock_t nv_monitor_reset(nv_port_t *nvp);
164 static int nv_bm_status_clear(nv_port_t *nvp);
165 static void nv_log(nv_ctl_t *nvc, nv_port_t *nvp, const char *fmt, ...);

167 #ifdef SGPIO_SUPPORT
168 static int nv_open(dev_t *devp, int flag, int otyp, cred_t *credp);
169 static int nv_close(dev_t dev, int flag, int otyp, cred_t *credp);
170 static int nv_ioctl(dev_t dev, int cmd, intptr_t arg, int mode,
171     cred_t *credp, int *rvalp);

173 static void nv_sgp_led_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle);
174 static int nv_sgp_detect(ddi_acc_handle_t pci_conf_handle, uint16_t *csrpp,
175     uint32_t *cbpp);
176 static int nv_sgp_init(nv_ctl_t *nvc);
177 static int nv_sgp_check_set_cmn(nv_ctl_t *nvc);
178 static int nv_sgp_csr_read(nv_ctl_t *nvc);
179 static void nv_sgp_csr_write(nv_ctl_t *nvc, uint32_t val);
180 static int nv_sgp_write_data(nv_ctl_t *nvc);
181 static void nv_sgp_activity_led_ctl(void *arg);
182 static void nv_sgp_drive_connect(nv_ctl_t *nvc, int drive);
183 static void nv_sgp_drive_disconnect(nv_ctl_t *nvc, int drive);
184 static void nv_sgp_drive_active(nv_ctl_t *nvc, int drive);
185 static void nv_sgp_locate(nv_ctl_t *nvc, int drive, int value);
186 static void nv_sgp_error(nv_ctl_t *nvc, int drive, int value);
187 static void nv_sgp_cleanup(nv_ctl_t *nvc);
188 #endif
```



```
191 /*
192  * DMA attributes for the data buffer for x86.  dma_attr_burstsizes is unused.
193  * Verify if needed if ported to other ISA.
194  */
195 static ddi_dma_attr_t buffer_dma_attr = {
196         DMA_ATTR_V0,            /* dma_attr_version */
197         0,                      /* dma_attr_addr_lo: lowest bus address */
198         0xffffffffull,          /* dma_attr_addr_hi: */
199         NV_BM_64K_BOUNDARY - 1, /* dma_attr_count_max i.e for one cookie */
200         4,                      /* dma_attr_align */
201         1,                      /* dma_attr_burstsizes. */
202         1,                      /* dma_attr_minxfer */
203         0xffffffffull,          /* dma_attr_maxxfer including all cookies */
204         0xffffffffull,          /* dma_attr_seg */
205         NV_DMA_NSEGS,           /* dma_attr_sgllen */
206         512,                    /* dma_attr_granular */
207         0,                      /* dma_attr_flags */
208 };
______unchanged_portion_omitted_


310 static sata_tran_hotplug_ops_t nv_hotplug_ops;

312 extern struct mod_ops mod_driverops;

314 static  struct modldrv modldrv = {
315         &mod_driverops, /* driverops */
316         "NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 HBA",
315         "Nvidia ck804/mcp51/mcp55 HBA",
317         &nv_dev_ops,    /* driver ops */
318 };
______unchanged_portion_omitted_

553 #else

555 #define nv_put8 ddi_put8
556 #define nv_put32 ddi_put32
557 #define nv_get32 ddi_get32
558 #define nv_put16 ddi_put16
559 #define nv_get16 ddi_get16
560 #define nv_get8 ddi_get8

562 #endif


565 /*
566  * Driver attach
567  */
568 static int
569 nv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
570 {
571         int status, attach_state, intr_types, bar, i, j, command;
572         int inst = ddi_get_instance(dip);
573         ddi_acc_handle_t pci_conf_handle;
574         nv_ctl_t *nvc;
575         uint8_t subclass;
576         uint32_t reg32;
577 #ifdef SGPIO_SUPPORT
578         pci_regspec_t *regs;
579         int rlen;
580 #endif

582         switch (cmd) {

584         case DDI_ATTACH:
```

```
586                 attach_state = ATTACH_PROGRESS_NONE;

588                 status = ddi_soft_state_zalloc(nv_statep, inst);

590                 if (status != DDI_SUCCESS) {
591                         break;
592                 }

594                 nvc = ddi_get_soft_state(nv_statep, inst);

596                 nvc->nvc_dip = dip;

598                 NVLOG(NVDBG_INIT, nvc, NULL, "nv_attach(): DDI_ATTACH", NULL);

600                 attach_state |= ATTACH_PROGRESS_STATEP_ALLOC;

602                 if (pci_config_setup(dip, &pci_conf_handle) == DDI_SUCCESS) {
603                         nvc->nvc_devid = pci_config_get16(pci_conf_handle,
604                             PCI_CONF_DEVID);
605                         nvc->nvc_revid = pci_config_get8(pci_conf_handle,
606                             PCI_CONF_REVID);
607                         NVLOG(NVDBG_INIT, nvc, NULL,
608                             "inst %d: devid is %x silicon revid is %x"
609                             " nv_debug_flags=%x", inst, nvc->nvc_devid,
610                             nvc->nvc_revid, nv_debug_flags);
605                             "inst %d: silicon revid is %x nv_debug_flags=%x",
606                             inst, nvc->nvc_revid, nv_debug_flags);
611                 } else {
612                         break;
613                 }

615                 attach_state |= ATTACH_PROGRESS_CONF_HANDLE;

617                 /*
618                  * Set the PCI command register: enable IO/MEM/Master.
619                  */
620                 command = pci_config_get16(pci_conf_handle, PCI_CONF_COMM);
621                 pci_config_put16(pci_conf_handle, PCI_CONF_COMM,
622                     command|PCI_COMM_IO|PCI_COMM_MAE|PCI_COMM_ME);

624                 subclass = pci_config_get8(pci_conf_handle, PCI_CONF_SUBCLASS);

626                 if (subclass & PCI_MASS_RAID) {
627                         cmn_err(CE_WARN,
628                             "attach failed: RAID mode not supported");

630                         break;
631                 }

633                 /*
634                  * the 6 bars of the controller are:
635                  * 0: port 0 task file
636                  * 1: port 0 status
637                  * 2: port 1 task file
638                  * 3: port 1 status
639                  * 4: bus master for both ports
640                  * 5: extended registers for SATA features
641                  */
642                 for (bar = 0; bar < 6; bar++) {
643                         status = ddi_regs_map_setup(dip, bar + 1,
644                             (caddr_t *)&nvc->nvc_bar_addr[bar], 0, 0, &accattr,
645                             &nvc->nvc_bar_hdl[bar]);

647                         if (status != DDI_SUCCESS) {
648                                 NVLOG(NVDBG_INIT, nvc, NULL,
```



```
649                                     "ddi_regs_map_setup failure for bar"
650                                     " %d status = %d", bar, status);
651                                 break;
652                         }
653                 }

655                 attach_state |= ATTACH_PROGRESS_BARS;

657                 /*
658                  * initialize controller structures
659                  */
660                 status = nv_init_ctl(nvc, pci_conf_handle);

662                 if (status == NV_FAILURE) {
663                         NVLOG(NVDBG_INIT, nvc, NULL, "nv_init_ctl failed",
664                             NULL);

666                         break;
667                 }

669                 attach_state |= ATTACH_PROGRESS_CTL_SETUP;

671                 /*
672                  * initialize mutexes
673                  */
674                 mutex_init(&nvc->nvc_mutex, NULL, MUTEX_DRIVER,
675                     DDI_INTR_PRI(nvc->nvc_intr_pri));

677                 attach_state |= ATTACH_PROGRESS_MUTEX_INIT;

679                 /*
680                  * get supported interrupt types
681                  */
682                 if (ddi_intr_get_supported_types(dip, &intr_types) !=
683                     DDI_SUCCESS) {
684                         nv_cmn_err(CE_WARN, nvc, NULL,
685                             "ddi_intr_get_supported_types failed");

687                         break;
688                 }

690                 NVLOG(NVDBG_INIT, nvc, NULL,
691                     "ddi_intr_get_supported_types() returned: 0x%x",
692                     intr_types);

694 #ifdef NV_MSI_SUPPORTED
695                 if (intr_types & DDI_INTR_TYPE_MSI) {
696                         NVLOG(NVDBG_INIT, nvc, NULL,
697                             "using MSI interrupt type", NULL);

699                         /*
700                          * Try MSI first, but fall back to legacy if MSI
701                          * attach fails
702                          */
703                         if (nv_add_msi_intrs(nvc) == DDI_SUCCESS) {
704                                 nvc->nvc_intr_type = DDI_INTR_TYPE_MSI;
705                                 attach_state |= ATTACH_PROGRESS_INTR_ADDED;
706                                 NVLOG(NVDBG_INIT, nvc, NULL,
707                                     "MSI interrupt setup done", NULL);
708                         } else {
709                                 nv_cmn_err(CE_CONT, nvc, NULL,
710                                     "MSI registration failed "
711                                     "will try Legacy interrupts");
712                         }
713                 }
714 #endif
```

```
716                 /*
717                  * Either the MSI interrupt setup has failed or only
718                  * the fixed interrupts are available on the system.
719                  */
720                 if (!(attach_state & ATTACH_PROGRESS_INTR_ADDED) &&
721                     (intr_types & DDI_INTR_TYPE_FIXED)) {

723                         NVLOG(NVDBG_INIT, nvc, NULL,
724                             "using Legacy interrupt type", NULL);

726                         if (nv_add_legacy_intrs(nvc) == DDI_SUCCESS) {
727                                 nvc->nvc_intr_type = DDI_INTR_TYPE_FIXED;
728                                 attach_state |= ATTACH_PROGRESS_INTR_ADDED;
729                                 NVLOG(NVDBG_INIT, nvc, NULL,
730                                     "Legacy interrupt setup done", NULL);
731                         } else {
732                                 nv_cmn_err(CE_WARN, nvc, NULL,
733                                     "legacy interrupt setup failed");
734                                 NVLOG(NVDBG_INIT, nvc, NULL,
735                                     "legacy interrupt setup failed", NULL);
736                                 break;
737                         }
738                 }

740                 if (!(attach_state & ATTACH_PROGRESS_INTR_ADDED)) {
741                         NVLOG(NVDBG_INIT, nvc, NULL,
742                             "no interrupts registered", NULL);
743                         break;
744                 }

746 #ifdef SGPIO_SUPPORT
747                 /*
748                  * save off the controller number
749                  */
750                 (void) ddi_getlongprop(DDI_DEV_T_NONE, dip, DDI_PROP_DONTPASS,
751                     "reg", (caddr_t)&regs, &rlen);
752                 nvc->nvc_ctlr_num = PCI_REG_FUNC_G(regs->pci_phys_hi);
753                 kmem_free(regs, rlen);

755                 /*
756                  * initialize SGPIO
757                  */
758                 nv_sgp_led_init(nvc, pci_conf_handle);
759 #endif  /* SGPIO_SUPPORT */

761                 /*
762                  * Do initial reset so that signature can be gathered
763                  */
764                 for (j = 0; j < NV_NUM_PORTS; j++) {
765                         ddi_acc_handle_t bar5_hdl;
766                         uint32_t sstatus;
767                         nv_port_t *nvp;

769                         nvp = &(nvc->nvc_port[j]);
770                         bar5_hdl = nvp->nvp_ctlp->nvc_bar_hdl[5];
771                         sstatus = ddi_get32(bar5_hdl, nvp->nvp_sstatus);

773                         if (SSTATUS_GET_DET(sstatus) ==
774                             SSTATUS_DET_DEVPRE_PHYCOM) {

776                                 nvp->nvp_state |= NV_ATTACH;
777                                 nvp->nvp_type = SATA_DTYPE_UNKNOWN;
778                                 mutex_enter(&nvp->nvp_mutex);
779                                 nv_reset(nvp, "attach");
```



```
781                                 while (nvp->nvp_state & NV_RESET) {
782                                         cv_wait(&nvp->nvp_reset_cv,
783                                             &nvp->nvp_mutex);
784                                 }

786                                 mutex_exit(&nvp->nvp_mutex);
787                         }
788                 }

790                 /*
791                  * attach to sata module
792                  */
793                 if (sata_hba_attach(nvc->nvc_dip,
794                     &nvc->nvc_sata_hba_tran,
795                     DDI_ATTACH) != DDI_SUCCESS) {
796                         attach_state |= ATTACH_PROGRESS_SATA_MODULE;

798                         break;
799                 }

801                 pci_config_teardown(&pci_conf_handle);

803                 NVLOG(NVDBG_INIT, nvc, NULL, "nv_attach DDI_SUCCESS", NULL);

805                 return (DDI_SUCCESS);

807         case DDI_RESUME:

809                 nvc = ddi_get_soft_state(nv_statep, inst);

811                 NVLOG(NVDBG_INIT, nvc, NULL,
812                     "nv_attach(): DDI_RESUME inst %d", inst);

814                 if (pci_config_setup(dip, &pci_conf_handle) != DDI_SUCCESS) {
815                         return (DDI_FAILURE);
816                 }

818                 /*
819                  * Set the PCI command register: enable IO/MEM/Master.
820                  */
821                 command = pci_config_get16(pci_conf_handle, PCI_CONF_COMM);
822                 pci_config_put16(pci_conf_handle, PCI_CONF_COMM,
823                     command|PCI_COMM_IO|PCI_COMM_MAE|PCI_COMM_ME);

825                 /*
826                  * Need to set bit 2 to 1 at config offset 0x50
827                  * to enable access to the bar5 registers.
828                  */
829                 reg32 = pci_config_get32(pci_conf_handle, NV_SATA_CFG_20);

831                 if ((reg32 & NV_BAR5_SPACE_EN) != NV_BAR5_SPACE_EN) {
832                         pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
833                             reg32 | NV_BAR5_SPACE_EN);
834                 }

836                 nvc->nvc_state &= ~NV_CTRL_SUSPEND;

838                 for (i = 0; i < NV_MAX_PORTS(nvc); i++) {
839                         nv_resume(&(nvc->nvc_port[i]));
840                 }

842                 pci_config_teardown(&pci_conf_handle);

844                 return (DDI_SUCCESS);

846         default:
```

```
 847                 return (DDI_FAILURE);
 848         }


 851         /*
 852          * DDI_ATTACH failure path starts here
 853          */

 855         if (attach_state & ATTACH_PROGRESS_INTR_ADDED) {
 856                 nv_rem_intrs(nvc);
 857         }

 859         if (attach_state & ATTACH_PROGRESS_SATA_MODULE) {
 860                 /*
 861                  * Remove timers
 862                  */
 863                 int port = 0;
 864                 nv_port_t *nvp;

 866                 for (; port < NV_MAX_PORTS(nvc); port++) {
 867                         nvp = &(nvc->nvc_port[port]);
 868                         if (nvp->nvp_timeout_id != 0) {
 869                                 (void) untimeout(nvp->nvp_timeout_id);
 870                         }
 871                 }
 872         }

 874         if (attach_state & ATTACH_PROGRESS_MUTEX_INIT) {
 875                 mutex_destroy(&nvc->nvc_mutex);
 876         }

 878         if (attach_state & ATTACH_PROGRESS_CTL_SETUP) {
 879                 nv_uninit_ctl(nvc);
 880         }

 882         if (attach_state & ATTACH_PROGRESS_BARS) {
 883                 while (--bar >= 0) {
 884                         ddi_regs_map_free(&nvc->nvc_bar_hdl[bar]);
 885                 }
 886         }

 888         if (attach_state & ATTACH_PROGRESS_STATEP_ALLOC) {
 889                 ddi_soft_state_free(nv_statep, inst);
 890         }

 892         if (attach_state & ATTACH_PROGRESS_CONF_HANDLE) {
 893                 pci_config_teardown(&pci_conf_handle);
 894         }

 896         cmn_err(CE_WARN, "nv_sata%d attach failed", inst);

 898         return (DDI_FAILURE);
 899 }
```

*_____unchanged_portion_omitted_*

```
2500 /*
2501  * Initialize register handling specific to mcp51/mcp55/mcp61
2497  * Initialize register handling specific to mcp51/mcp55
2502  */
2503 /* ARGSUSED */
2504 static void
2505 mcp5x_reg_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle)
2506 {
2507         nv_port_t *nvp;
2508         uchar_t *bar5  = nvc->nvc_bar_addr[5];
```



```
2509         uint8_t off, port;

2511         nvc->nvc_mcp5x_ctl = (uint32_t *)(bar5 + MCP5X_CTL);
2512         nvc->nvc_mcp5x_ncq = (uint32_t *)(bar5 + MCP5X_NCQ);

2514         for (port = 0, off = 0; port < NV_MAX_PORTS(nvc); port++, off += 2) {
2515                 nvp = &(nvc->nvc_port[port]);
2516                 nvp->nvp_mcp5x_int_status =
2517                     (uint16_t *)(bar5 + MCP5X_INT_STATUS + off);
2518                 nvp->nvp_mcp5x_int_ctl =
2519                     (uint16_t *)(bar5 + MCP5X_INT_CTL + off);

2521                 /*
2522                  * clear any previous interrupts asserted
2523                  */
2524                 nv_put16(nvc->nvc_bar_hdl[5], nvp->nvp_mcp5x_int_status,
2525                     MCP5X_INT_CLEAR);

2527                 /*
2528                  * These are the interrupts to accept for now.  The spec
2529                  * says these are enable bits, but nvidia has indicated
2530                  * these are masking bits.  Even though they may be masked
2531                  * out to prevent asserting the main interrupt, they can
2532                  * still be asserted while reading the interrupt status
2533                  * register, so that needs to be considered in the interrupt
2534                  * handler.
2535                  */
2536                 nv_put16(nvc->nvc_bar_hdl[5], nvp->nvp_mcp5x_int_ctl,
2537                     ~(MCP5X_INT_IGNORE));
2538         }

2540         /*
2541          * Allow the driver to program the BM on the first command instead
2542          * of waiting for an interrupt.
2543          */
2544 #ifdef NCQ
2545         flags = MCP_SATA_AE_NCQ_PDEV_FIRST_CMD | MCP_SATA_AE_NCQ_SDEV_FIRST_CMD;
2546         nv_put32(nvc->nvc_bar_hdl[5], nvc->nvc_mcp5x_ncq, flags);
2547         flags = MCP_SATA_AE_CTL_PRI_SWNCQ | MCP_SATA_AE_CTL_SEC_SWNCQ;
2548         nv_put32(nvc->nvc_bar_hdl[5], nvc->nvc_mcp5x_ctl, flags);
2549 #endif

2551         /*
2552          * mcp55 rev A03 and above supports 40-bit physical addressing.
2553          * Enable DMA to take advantage of that.
2554          *
2555          */
2556         if ((nvc->nvc_devid > 0x37f) ||
2557             ((nvc->nvc_devid == 0x37f) && (nvc->nvc_revid >= 0xa3))) {
2552         if (nvc->nvc_revid >= 0xa3) {
2558                 if (nv_sata_40bit_dma == B_TRUE) {
2559                         uint32_t reg32;
2560                         NVLOG(NVDBG_INIT, nvp->nvp_ctlp, nvp,
2561                             "devid is %X revid is %X. 40-bit DMA"
2562                             " addressing enabled", nvc->nvc_devid,
2563                             nvc->nvc_revid);
2556                             "rev id is %X.  40-bit DMA addressing"
2557                             " enabled", nvc->nvc_revid);
2564                         nvc->dma_40bit = B_TRUE;

2566                         reg32 = pci_config_get32(pci_conf_handle,
2567                             NV_SATA_CFG_20);
2568                         pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
2569                             reg32 | NV_40BIT_PRD);

2571                         /*
```

```
2572                         * CFG_23 bits 0-7 contain the top 8 bits (of 40
2573                         * bits) for the primary PRD table, and bits 8-15
2574                         * contain the top 8 bits for the secondary.  Set
2575                         * to zero because the DMA attribute table for PRD
2576                         * allocation forces it into 32 bit address space
2577                         * anyway.
2578                         */
2579                        reg32 = pci_config_get32(pci_conf_handle,
2580                            NV_SATA_CFG_23);
2581                        pci_config_put32(pci_conf_handle, NV_SATA_CFG_23,
2582                            reg32 & 0xffff0000);
2583                } else {
2584                        NVLOG(NVDBG_INIT, nvp->nvp_ctlp, nvp,
2585                            "40-bit DMA disabled by nv_sata_40bit_dma", NULL);
2586                }
2587        } else {
2588                nv_cmn_err(CE_NOTE, nvp->nvp_ctlp, nvp, "devid is %X revid is"
2589                    " %X. Not capable of 40-bit DMA addressing",
2590                    nvc->nvc_devid, nvc->nvc_revid);
2582                nv_cmn_err(CE_NOTE, nvp->nvp_ctlp, nvp, "rev id is %X and is "
2583                    "not capable of 40-bit DMA addressing", nvc->nvc_revid);
2591        }
2592 }
```

*unchanged_portion_omitted_*

```
2643 /*
2644  * Initialize the controller and set up driver data structures.
2645  * determine if ck804 or mcp5x class.
2646  */
2647 static int
2648 nv_init_ctl(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle)
2649 {
2650        struct sata_hba_tran stran;
2651        nv_port_t *nvp;
2652        int j;
2645        int j, ck804;
2653        uchar_t *cmd_addr, *ctl_addr, *bm_addr;
2654        ddi_acc_handle_t bar5_hdl = nvc->nvc_bar_hdl[5];
2655        uchar_t *bar5  = nvc->nvc_bar_addr[5];
2656        uint32_t reg32;
2657        uint8_t reg8, reg8_save;

2659        NVLOG(NVDBG_INIT, nvc, NULL, "nv_init_ctl entered", NULL);

2654        ck804 = B_TRUE;
2655 #ifdef SGPIO_SUPPORT
2661        nvc->nvc_mcp5x_flag = B_FALSE;
2657 #endif

2663        /*
2664         * Need to set bit 2 to 1 at config offset 0x50
2665         * to enable access to the bar5 registers.
2666         */
2667        reg32 = pci_config_get32(pci_conf_handle, NV_SATA_CFG_20);
2668        if (!(reg32 & NV_BAR5_SPACE_EN)) {
2669                pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
2670                    reg32 | NV_BAR5_SPACE_EN);
2671        }

2673        /*
2674         * Determine if this is ck804 or mcp5x.  ck804 will map in the
2675         * task file registers into bar5 while mcp5x won't.  The offset of
2676         * the task file registers in mcp5x's space is unused, so it will
2677         * return zero.  So check one of the task file registers to see if it is
2678         * writable and reads back what was written.  If it's mcp5x it will
```



```
2679         * return back 0xff whereas ck804 will return the value written.
2680         */
2681        reg8_save = nv_get8(bar5_hdl,
2682            (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X));


2685        for (j = 1; j < 3; j++) {

2687                nv_put8(bar5_hdl, (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X), j);
2688                reg8 = nv_get8(bar5_hdl,
2689                    (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X));

2691                if (reg8 != j) {
2688                        ck804 = B_FALSE;
2692                        nvc->nvc_mcp5x_flag = B_TRUE;
2693                        break;
2694                }
2695        }

2697        nv_put8(bar5_hdl, (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X), reg8_save);

2699        if (nvc->nvc_mcp5x_flag == B_FALSE) {
2700                NVLOG(NVDBG_INIT, nvc, NULL, "controller is CK804/MCP04",
2701                    NULL);
2696        if (ck804 == B_TRUE) {
2697                NVLOG(NVDBG_INIT, nvc, NULL, "controller is CK804", NULL);
2702                nvc->nvc_interrupt = ck804_intr;
2703                nvc->nvc_reg_init = ck804_reg_init;
2704                nvc->nvc_set_intr = ck804_set_intr;
2705        } else {
2706                NVLOG(NVDBG_INIT, nvc, NULL, "controller is MCP51/MCP55/MCP61",
2707                    NULL);
2702                NVLOG(NVDBG_INIT, nvc, NULL, "controller is MCP51/MCP55", NULL);
2708                nvc->nvc_interrupt = mcp5x_intr;
2709                nvc->nvc_reg_init = mcp5x_reg_init;
2710                nvc->nvc_set_intr = mcp5x_set_intr;
2711        }


2714        stran.sata_tran_hba_rev = SATA_TRAN_HBA_REV;
2715        stran.sata_tran_hba_dip = nvc->nvc_dip;
2716        stran.sata_tran_hba_num_cports = NV_NUM_PORTS;
2717        stran.sata_tran_hba_features_support =
2718            SATA_CTLF_HOTPLUG | SATA_CTLF_ASN | SATA_CTLF_ATAPI;
2719        stran.sata_tran_hba_qdepth = NV_QUEUE_SLOTS;
2720        stran.sata_tran_probe_port = nv_sata_probe;
2721        stran.sata_tran_start = nv_sata_start;
2722        stran.sata_tran_abort = nv_sata_abort;
2723        stran.sata_tran_reset_dport = nv_sata_reset;
2724        stran.sata_tran_selftest = NULL;
2725        stran.sata_tran_hotplug_ops = &nv_hotplug_ops;
2726        stran.sata_tran_pwrmgt_ops = NULL;
2727        stran.sata_tran_ioctl = NULL;
2728        nvc->nvc_sata_hba_tran = stran;

2730        nvc->nvc_port = kmem_zalloc(sizeof (nv_port_t) * NV_MAX_PORTS(nvc),
2731            KM_SLEEP);

2733        /*
2734         * initialize registers common to all chipsets
2735         */
2736        nv_common_reg_init(nvc);

2738        for (j = 0; j < NV_MAX_PORTS(nvc); j++) {
2739                nvp = &(nvc->nvc_port[j]);
```

```
2741                cmd_addr = nvp->nvp_cmd_addr;
2742                ctl_addr = nvp->nvp_ctl_addr;
2743                bm_addr = nvp->nvp_bm_addr;

2745                mutex_init(&nvp->nvp_mutex, NULL, MUTEX_DRIVER,
2746                    DDI_INTR_PRI(nvc->nvc_intr_pri));

2748                cv_init(&nvp->nvp_sync_cv, NULL, CV_DRIVER, NULL);
2749                cv_init(&nvp->nvp_reset_cv, NULL, CV_DRIVER, NULL);

2751                nvp->nvp_data   = cmd_addr + NV_DATA;
2752                nvp->nvp_error  = cmd_addr + NV_ERROR;
2753                nvp->nvp_feature = cmd_addr + NV_FEATURE;
2754                nvp->nvp_count  = cmd_addr + NV_COUNT;
2755                nvp->nvp_sect   = cmd_addr + NV_SECT;
2756                nvp->nvp_lcyl   = cmd_addr + NV_LCYL;
2757                nvp->nvp_hcyl   = cmd_addr + NV_HCYL;
2758                nvp->nvp_drvhd  = cmd_addr + NV_DRVHD;
2759                nvp->nvp_status = cmd_addr + NV_STATUS;
2760                nvp->nvp_cmd    = cmd_addr + NV_CMD;
2761                nvp->nvp_altstatus = ctl_addr + NV_ALTSTATUS;
2762                nvp->nvp_devctl = ctl_addr + NV_DEVCTL;

2764                nvp->nvp_bmicx  = bm_addr + BMICX_REG;
2765                nvp->nvp_bmisx  = bm_addr + BMISX_REG;
2766                nvp->nvp_bmidtpx = (uint32_t *)(bm_addr + BMIDTPX_REG);

2768                nvp->nvp_state = 0;

2770                /*
2771                 * Initialize dma handles, etc.
2772                 * If it fails, the port is in inactive state.
2773                 */
2774                nv_init_port(nvp);
2775        }

2777        /*
2778         * initialize register by calling chip specific reg initialization
2779         */
2780        (*(nvc->nvc_reg_init))(nvc, pci_conf_handle);

2782        /* initialize the hba dma attribute */
2783        if (nvc->dma_40bit == B_TRUE)
2784                nvc->nvc_sata_hba_tran.sata_tran_hba_dma_attr =
2785                    &buffer_dma_40bit_attr;
2786        else
2787                nvc->nvc_sata_hba_tran.sata_tran_hba_dma_attr =
2788                    &buffer_dma_attr;

2790        return (NV_SUCCESS);
2791 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   20356 Thu Jan 16 19:58:20 2014
new/usr/src/uts/common/sys/sata/adapters/nv_sata/nv_sata.h
1023 nv_sata support for NVIDIA MCP61
Reviewed by: Garrett D'Amore <garrett@damore.org>
Reviewed by: Dan McDonald <danmcd@omniti.com>
**********************************************************
/*
 * 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) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#ifndef _NV_SATA_H
#define _NV_SATA_H


#ifdef  __cplusplus
extern "C" {
#endif


/*
 * SGPIO Support
 * Enable SGPIO support only on x86/x64, because it is implemented using
 * functions that are only available on x86/x64.
 */

#define NV_MAX_PORTS(nvc) nvc->nvc_sata_hba_tran.sata_tran_hba_num_cports

typedef struct nv_port nv_port_t;

#ifdef SGPIO_SUPPORT
typedef struct nv_sgp_cmn nv_sgp_cmn_t;
#endif

/*
 * sizes of strings to allocate
 */
#define NV_STR_LEN      10
#define NV_LOGBUF_LEN   512
#define NV_REASON_LEN   30


typedef struct nv_ctl {
        /*
         * Each of these are specific to the chipset in use.
```



```
         */
        uint_t          (*nvc_interrupt)(caddr_t arg1, caddr_t arg2);
        void            (*nvc_reg_init)(struct nv_ctl *nvc,
                            ddi_acc_handle_t pci_conf_handle);

        dev_info_t      *nvc_dip; /* devinfo pointer of controller */

        struct nv_port  *nvc_port; /* array of pointers to port struct */

        /*
         * handle and base address to register space.
         *
         * 0: port 0 task file
         * 1: port 0 status
         * 2: port 1 task file
         * 3: port 1 status
         * 4: bus master for both ports
         * 5: extended registers for SATA features
         */
        ddi_acc_handle_t nvc_bar_hdl[6];
        uchar_t         *nvc_bar_addr[6];

        /*
         * sata registers in bar 5 which are shared on all devices
         * on the channel.
         */
        uint32_t        *nvc_mcp5x_ctl;
        uint32_t        *nvc_mcp5x_ncq; /* NCQ status control bits */

        kmutex_t        nvc_mutex; /* ctrl level lock */

        ddi_intr_handle_t *nvc_htable;  /* For array of interrupts */
        int              nvc_intr_type; /* What type of interrupt */
        int             nvc_intr_cnt;   /* # of intrs count returned */
        size_t          nvc_intr_size;  /* Size of intr array to */
        uint_t          nvc_intr_pri;   /* Interrupt priority */
        int             nvc_intr_cap;   /* Interrupt capabilities */
        uint8_t         *nvc_ck804_int_status; /* interrupt status ck804 */

        sata_hba_tran_t nvc_sata_hba_tran; /* sata_hba_tran for ctrl */

        /*
         * enable/disable interrupts, controller specific
         */
        void            (*nvc_set_intr)(nv_port_t *nvp, int flag);
        int             nvc_state;      /* state flags of ctrl see below */
        uint16_t        nvc_devid;      /* PCI devid of device */
        uint8_t         nvc_revid;      /* PCI revid of device */
        boolean_t       dma_40bit;      /* 40bit DMA support */
        boolean_t       nvc_mcp5x_flag; /* is the controller MCP51/MCP55 */

#ifdef SGPIO_SUPPORT
        int             nvc_mcp5x_flag; /* is the controller MCP51/MCP55 */
        uint8_t         nvc_ctlr_num;   /* controller number within the part */
        uint32_t        nvc_sgp_csr;    /* SGPIO CSR i/o address */
        volatile nv_sgp_cb_t *nvc_sgp_cbp; /* SGPIO Control Block */
        nv_sgp_cmn_t    *nvc_sgp_cmn;   /* SGPIO shared data */
#endif
} nv_ctl_t;
______unchanged_portion_omitted_
```