```
*******************************************************
    2056 Wed Sep 28 11:57:27 2011
new/usr/src/man/man7d/nv_sata.7d
1023 nv_sata support for NVIDIA MCP61
*******************************************************
   1 '\" te
   2 .\" Copyright (c) 2008, Sun Microsystems, Inc.  All Rights Reserved
   3 .\" Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
   4 #endif /* ! codereview */
   5 .\" The contents of this file are subject to the terms of the Common Development
   6 .\" You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE or http:
   7 .\" When distributing Covered Code, include this CDDL HEADER in each file and in
   8 .TH nv_sata 7D "25 Sep 2011"
   3 .TH nv_sata 7D "22 Jul 2008" "SunOS 5.11" "Devices"
   9 .SH NAME
  10 nv_sata \- NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver
   5 nv_sata \- Nvidia ck804/mcp55 SATA controller driver
  11 .SH SYNOPSIS
  12 .LP
  13 .nf
  14 \fBsata@unit-address\fR
  15 .fi

  17 .SH DESCRIPTION
  18 .sp
  19 .LP
  20 The \fBnv_sata\fR driver is a SATA HBA driver that supports NVIDIA CK804/MCP04
  21 and MCP51/MCP55/MCP61 SATA HBA controllers. Note that while these controllers
  15 The \fBnv_sata\fR driver is a SATA HBA driver that supports Nvidia ck804 and
  16 mcp55 SATA HBA controllers.       Note that while these Nvidia controllers
  22 support standard SATA features including SATA-II drives, NCQ, hotplug and ATAPI
  23 drives, the driver currently does not support NCQ features.
  24 .SH CONFIGURATION
  25 .sp
  26 .LP
  27 The \fBnv_sata\fR module contains no user configurable parameters.
  28 .SH FILES
  29 .sp
  30 .ne 2
  31 .mk
  32 .na
  33 \fB\fB/kernel/drv/nv_sata\fR\fR
  34 .ad
  35 .sp .6
  36 .RS 4n
  37 32-bit \fBELF\fR kernel module (x86).
  38 .RE

  40 .sp
  41 .ne 2
  42 .mk
  43 .na
  44 \fB\fB/kernel/drv/amd64/nv_sata\fR\fR
  45 .ad
  46 .sp .6
  47 .RS 4n
  48 64-bit \fBELF\fR kernel module (x86).
  49 .RE

  51 .SH ATTRIBUTES
  52 .sp
  53 .LP
  54 See \fBattributes\fR(5) for descriptions of the following attribute:
  55 .sp

  57 .sp
```

```
  58 .TS
  59 tab(^G) box;
  60 cw(2.75i) |cw(2.75i)
  61 lw(2.75i) |lw(2.75i)
  62 .
  63 ATTRIBUTE TYPE^GATTRIBUTE VALUE
  64 _
  65 Architecture^Gx86
  66 .TE

  68 .SH SEE ALSO
  69 .sp
  70 .LP
  71 \fBcfgadm\fR(1M), \fBcfgadm_sata\fR(1M), \fBprtconf\fR(1M), \fBsata\fR(7D),
  72 \fBsd\fR(7D)
  73 .sp
  74 .LP
  75 \fIWriting Device Drivers\fR
```

```
**********************************************************
    2374 Wed Sep 28 11:57:30 2011
new/usr/src/pkg/manifests/driver-storage-nv_sata.mf
1023 nv_sata support for NVIDIA MCP61
**********************************************************
   1 #
   2 # CDDL HEADER START
   3 #
   4 # The contents of this file are subject to the terms of the
   5 # Common Development and Distribution License (the "License").
   6 # You may not use this file except in compliance with the License.
   7 #
   8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9 # or http://www.opensolaris.org/os/licensing.
  10 # See the License for the specific language governing permissions
  11 # and limitations under the License.
  12 #
  13 # When distributing Covered Code, include this CDDL HEADER in each
  14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15 # If applicable, add the following below this CDDL HEADER, with the
  16 # fields enclosed by brackets "[]" replaced with your own identifying
  17 # information: Portions Copyright [yyyy] [name of copyright owner]
  18 #
  19 # CDDL HEADER END
  20 #

  22 #
  23 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
  24 # Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  25 #endif /* ! codereview */
  26 #

  28 #
  29 # The default for payload-bearing actions in this package is to appear in the
  30 # global zone only.  See the include file for greater detail, as well as
  31 # information about overriding the defaults.
  32 #
  33 <include global_zone_only_component>
  34 set name=pkg.fmri value=pkg:/driver/storage/nv_sata@$(PKGVERS)
  35 set name=pkg.description \
  36     value="NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver"
  37 set name=pkg.summary \
  38     value="NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 SATA controller driver"
  24     value="Nvidia ck804 pro / mcp55 pro combo SATA driver"
  25 set name=pkg.summary value="Nvidia ck804 pro / mcp55 pro combo SATA driver"
  39 set name=info.classification value=org.opensolaris.category.2008:Drivers/Ports
  40 set name=variant.arch value=i386
  41 dir path=kernel group=sys
  42 dir path=kernel/drv group=sys
  43 dir path=kernel/drv/$(ARCH64) group=sys
  44 dir path=usr/share/man
  45 dir path=usr/share/man/man7d
  46 driver name=nv_sata class=scsi-self-identifying perms="* 0644 root sys" \
  47     alias=pci10de,266 \
  48     alias=pci10de,267 \
  49     alias=pci10de,36 \
  50     alias=pci10de,37e \
  51 #endif /* ! codereview */
  52     alias=pci10de,37f \
  53     alias=pci10de,3e \
  54     alias=pci10de,3f6 \
  55     alias=pci10de,3f7 \
  56 #endif /* ! codereview */
  57     alias=pci10de,54 \
  58     alias=pci10de,55
  59 file path=kernel/drv/$(ARCH64)/nv_sata group=sys
```

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

```
**********************************************************
  180653 Wed Sep 28 11:57:32 2011
new/usr/src/uts/common/io/sata/adapters/nv_sata/nv_sata.c
1023 nv_sata support for NVIDIA MCP61
**********************************************************
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
  24  * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
  25 #endif /* ! codereview */
  26  */

  28 /*
  29  *
  30  * nv_sata is a combo SATA HBA driver for CK804/MCP04 (ck804) and
  31  * MCP55/MCP51/MCP61 (mcp5x) based chipsets.
  24  * nv_sata is a combo SATA HBA driver for ck804/mcp5x (mcp5x = mcp55/mcp51)
  25  * based chipsets.
  32  *
  33  * NCQ
  34  * ---
  35  *
  36  * A portion of the NCQ is in place, but is incomplete.  NCQ is disabled
  37  * and is likely to be revisited in the future.
  38  *
  39  *
  40  * Power Management
  41  * ----------------
  42  *
  43  * Normally power management would be responsible for ensuring the device
  44  * is quiescent and then changing power states to the device, such as
  45  * powering down parts or all of the device.  mcp5x/ck804 is unique in
  46  * that it is only available as part of a larger southbridge chipset, so
  47  * removing power to the device isn't possible.  Switches to control
  48  * power management states D0/D3 in the PCI configuration space appear to
  49  * be supported but changes to these states are apparently are ignored.
  50  * The only further PM that the driver _could_ do is shut down the PHY,
  51  * but in order to deliver the first rev of the driver sooner than later,
  52  * that will be deferred until some future phase.
  53  *
  54  * Since the driver currently will not directly change any power state to
  55  * the device, no power() entry point will be required.  However, it is
  56  * possible that in ACPI power state S3, aka suspend to RAM, that power
  57  * can be removed to the device, and the driver cannot rely on BIOS to
  58  * have reset any state.  For the time being, there is no known
  59  * non-default configurations that need to be programmed.  This judgement
```

```
  60  * is based on the port of the legacy ata driver not having any such
  61  * functionality and based on conversations with the PM team.  If such a
  62  * restoration is later deemed necessary it can be incorporated into the
  63  * DDI_RESUME processing.
  64  *
  65  */

  67 #include <sys/scsi/scsi.h>
  68 #include <sys/pci.h>
  69 #include <sys/byteorder.h>
  70 #include <sys/sunddi.h>
  71 #include <sys/sata/sata_hba.h>
  72 #ifdef SGPIO_SUPPORT
  73 #include <sys/sata/adapters/nv_sata/nv_sgpio.h>
  74 #include <sys/devctl.h>
  75 #include <sys/sdt.h>
  76 #endif
  77 #include <sys/sata/adapters/nv_sata/nv_sata.h>
  78 #include <sys/disp.h>
  79 #include <sys/note.h>
  80 #include <sys/promif.h>


  83 /*
  84  * Function prototypes for driver entry points
  85  */
  86 static int nv_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
  87 static int nv_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
  88 static int nv_quiesce(dev_info_t *dip);
  89 static int nv_getinfo(dev_info_t *dip, ddi_info_cmd_t infocmd,
  90     void *arg, void **result);

  92 /*
  93  * Function prototypes for entry points from sata service module
  94  * These functions are distinguished from other local functions
  95  * by the prefix "nv_sata_"
  96  */
  97 static int nv_sata_start(dev_info_t *dip, sata_pkt_t *spkt);
  98 static int nv_sata_abort(dev_info_t *dip, sata_pkt_t *spkt, int);
  99 static int nv_sata_reset(dev_info_t *dip, sata_device_t *sd);
 100 static int nv_sata_activate(dev_info_t *dip, sata_device_t *sd);
 101 static int nv_sata_deactivate(dev_info_t *dip, sata_device_t *sd);

 103 /*
 104  * Local function prototypes
 105  */
 106 static uint_t mcp5x_intr(caddr_t arg1, caddr_t arg2);
 107 static uint_t ck804_intr(caddr_t arg1, caddr_t arg2);
 108 static int nv_add_legacy_intrs(nv_ctl_t *nvc);
 109 #ifdef NV_MSI_SUPPORTED
 110 static int nv_add_msi_intrs(nv_ctl_t *nvc);
 111 #endif
 112 static void nv_rem_intrs(nv_ctl_t *nvc);
 113 static int nv_start_common(nv_port_t *nvp, sata_pkt_t *spkt);
 114 static int nv_start_nodata(nv_port_t *nvp, int slot);
 115 static void nv_intr_nodata(nv_port_t *nvp, nv_slot_t *spkt);
 116 static int nv_start_pio_in(nv_port_t *nvp, int slot);
 117 static int nv_start_pio_out(nv_port_t *nvp, int slot);
 118 static void nv_intr_pio_in(nv_port_t *nvp, nv_slot_t *spkt);
 119 static void nv_intr_pio_out(nv_port_t *nvp, nv_slot_t *spkt);
 120 static int nv_start_pkt_pio(nv_port_t *nvp, int slot);
 121 static void nv_intr_pkt_pio(nv_port_t *nvp, nv_slot_t *nv_slotp);
 122 static int nv_start_dma(nv_port_t *nvp, int slot);
 123 static void nv_intr_dma(nv_port_t *nvp, struct nv_slot *spkt);
 124 static void nv_uninit_ctl(nv_ctl_t *nvc);
 125 static void mcp5x_reg_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle);
```

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

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

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

```
 192 /*
 193  * DMA attributes for the data buffer for x86.  dma_attr_burstsizes is unused.
 194  * Verify if needed if ported to other ISA.
 195  */
 196 static ddi_dma_attr_t buffer_dma_attr = {
 197         DMA_ATTR_V0,            /* dma_attr_version */
 198         0,                     /* dma_attr_addr_lo: lowest bus address */
 199         0xffffffffffull,       /* dma_attr_addr_hi: */
 200         NV_BM_64K_BOUNDARY - 1, /* dma_attr_count_max i.e for one cookie */
 201         4,                     /* dma_attr_align */
 202         1,                     /* dma_attr_burstsizes. */
 203         1,                     /* dma_attr_minxfer */
 204         0xffffffffffull,       /* dma_attr_maxxfer including all cookies */
 205         0xffffffffffull,       /* dma_attr_seg */
 206         NV_DMA_NSEGS,          /* dma_attr_sgllen */
 207         512,                   /* dma_attr_granular */
 208         0,                     /* dma_attr_flags */
 209 };
_____unchanged_portion_omitted_


 311 static sata_tran_hotplug_ops_t nv_hotplug_ops;

 313 extern struct mod_ops mod_driverops;

 315 static  struct modldrv modldrv = {
 316         &mod_driverops, /* driverops */
 317         "NVIDIA CK804/MCP04/MCP51/MCP55/MCP61 HBA",
 311         "Nvidia ck804/mcp51/mcp55 HBA",
 318         &nv_dev_ops,    /* driver ops */
 319 };
_____unchanged_portion_omitted_

 554 #else

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

 563 #endif


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

 583         switch (cmd) {

 585         case DDI_ATTACH:
```

```
 587                          attach_state = ATTACH_PROGRESS_NONE;

 589                          status = ddi_soft_state_zalloc(nv_statep, inst);

 591                          if (status != DDI_SUCCESS) {
 592                                  break;
 593                          }

 595                          nvc = ddi_get_soft_state(nv_statep, inst);

 597                          nvc->nvc_dip = dip;

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

 601                          attach_state |= ATTACH_PROGRESS_STATEP_ALLOC;

 603                          if (pci_config_setup(dip, &pci_conf_handle) == DDI_SUCCESS) {
 604                                  nvc->nvc_devid = pci_config_get16(pci_conf_handle,
 605                                      PCI_CONF_DEVID);
 606  #endif /* ! codereview */
 607                                  nvc->nvc_revid = pci_config_get8(pci_conf_handle,
 608                                      PCI_CONF_REVID);
 609                                  NVLOG(NVDBG_INIT, nvc, NULL,
 610                                      "inst %d: devid is %x silicon revid is %x"
 611                                      " nv_debug_flags=%x", inst, nvc->nvc_devid,
 612                                      nvc->nvc_revid, nv_debug_flags);
 598                                  "inst %d: silicon revid is %x nv_debug_flags=%x",
 599                                      inst, nvc->nvc_revid, nv_debug_flags);
 613                          } else {
 614                                  break;
 615                          }

 617                          attach_state |= ATTACH_PROGRESS_CONF_HANDLE;

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

 626                          subclass = pci_config_get8(pci_conf_handle, PCI_CONF_SUBCLASS);

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

 632                                  break;
 633                          }

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

 649                                  if (status != DDI_SUCCESS) {
 650                                          NVLOG(NVDBG_INIT, nvc, NULL,
```

```
 651                                              "ddi_regs_map_setup failure for bar"
 652                                              " %d status = %d", bar, status);
 653                                          break;
 654                                  }
 655                          }

 657                          attach_state |= ATTACH_PROGRESS_BARS;

 659                          /*
 660                           * initialize controller structures
 661                           */
 662                          status = nv_init_ctl(nvc, pci_conf_handle);

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

 668                                  break;
 669                          }

 671                          attach_state |= ATTACH_PROGRESS_CTL_SETUP;

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

 679                          attach_state |= ATTACH_PROGRESS_MUTEX_INIT;

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

 689                                  break;
 690                          }

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

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

 701                                  /*
 702                                   * Try MSI first, but fall back to legacy if MSI
 703                                   * attach fails
 704                                   */
 705                                  if (nv_add_msi_intrs(nvc) == DDI_SUCCESS) {
 706                                          nvc->nvc_intr_type = DDI_INTR_TYPE_MSI;
 707                                          attach_state |= ATTACH_PROGRESS_INTR_ADDED;
 708                                          NVLOG(NVDBG_INIT, nvc, NULL,
 709                                              "MSI interrupt setup done", NULL);
 710                                  } else {
 711                                          nv_cmn_err(CE_CONT, nvc, NULL,
 712                                              "MSI registration failed "
 713                                              "will try Legacy interrupts");
 714                                  }
 715                          }
 716  #endif
```

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

725                            NVLOG(NVDBG_INIT, nvc, NULL,
726                                "using Legacy interrupt type", NULL);

728                            if (nv_add_legacy_intrs(nvc) == DDI_SUCCESS) {
729                                    nvc->nvc_intr_type = DDI_INTR_TYPE_FIXED;
730                                    attach_state |= ATTACH_PROGRESS_INTR_ADDED;
731                                    NVLOG(NVDBG_INIT, nvc, NULL,
732                                        "Legacy interrupt setup done", NULL);
733                            } else {
734                                    nv_cmn_err(CE_WARN, nvc, NULL,
735                                        "legacy interrupt setup failed");
736                                    NVLOG(NVDBG_INIT, nvc, NULL,
737                                        "legacy interrupt setup failed", NULL);
738                                    break;
739                            }
740                    }

742                    if (!(attach_state & ATTACH_PROGRESS_INTR_ADDED)) {
743                            NVLOG(NVDBG_INIT, nvc, NULL,
744                                "no interrupts registered", NULL);
745                            break;
746                    }

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

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

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

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

775                            if (SSTATUS_GET_DET(sstatus) ==
776                                SSTATUS_DET_DEVPRE_PHYCOM) {

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

```
783                                    while (nvp->nvp_state & NV_RESET) {
784                                            cv_wait(&nvp->nvp_reset_cv,
785                                                &nvp->nvp_mutex);
786                                    }

788                                    mutex_exit(&nvp->nvp_mutex);
789                            }
790                    }

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

800                            break;
801                    }

803                    pci_config_teardown(&pci_conf_handle);

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

807                    return (DDI_SUCCESS);

809            case DDI_RESUME:

811                    nvc = ddi_get_soft_state(nv_statep, inst);

813                    NVLOG(NVDBG_INIT, nvc, NULL,
814                        "nv_attach(): DDI_RESUME inst %d", inst);

816                    if (pci_config_setup(dip, &pci_conf_handle) != DDI_SUCCESS) {
817                            return (DDI_FAILURE);
818                    }

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

827                    /*
828                     * Need to set bit 2 to 1 at config offset 0x50
829                     * to enable access to the bar5 registers.
830                     */
831                    reg32 = pci_config_get32(pci_conf_handle, NV_SATA_CFG_20);

833                    if ((reg32 & NV_BAR5_SPACE_EN) != NV_BAR5_SPACE_EN) {
834                            pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
835                                reg32 | NV_BAR5_SPACE_EN);
836                    }

838                    nvc->nvc_state &= ~NV_CTRL_SUSPEND;

840                    for (i = 0; i < NV_MAX_PORTS(nvc); i++) {
841                            nv_resume(&(nvc->nvc_port[i]));
842                    }

844                    pci_config_teardown(&pci_conf_handle);

846                    return (DDI_SUCCESS);

848            default:
```

```
 849                        return (DDI_FAILURE);
 850                }


 853                /*
 854                 * DDI_ATTACH failure path starts here
 855                 */

 857                if (attach_state & ATTACH_PROGRESS_INTR_ADDED) {
 858                        nv_rem_intrs(nvc);
 859                }

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

 868                        for (; port < NV_MAX_PORTS(nvc); port++) {
 869                                nvp = &(nvc->nvc_port[port]);
 870                                if (nvp->nvp_timeout_id != 0) {
 871                                        (void) untimeout(nvp->nvp_timeout_id);
 872                                }
 873                        }
 874                }

 876                if (attach_state & ATTACH_PROGRESS_MUTEX_INIT) {
 877                        mutex_destroy(&nvc->nvc_mutex);
 878                }

 880                if (attach_state & ATTACH_PROGRESS_CTL_SETUP) {
 881                        nv_uninit_ctl(nvc);
 882                }

 884                if (attach_state & ATTACH_PROGRESS_BARS) {
 885                        while (--bar >= 0) {
 886                                ddi_regs_map_free(&nvc->nvc_bar_hdl[bar]);
 887                        }
 888                }

 890                if (attach_state & ATTACH_PROGRESS_STATEP_ALLOC) {
 891                        ddi_soft_state_free(nv_statep, inst);
 892                }

 894                if (attach_state & ATTACH_PROGRESS_CONF_HANDLE) {
 895                        pci_config_teardown(&pci_conf_handle);
 896                }

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

 900                return (DDI_FAILURE);
 901 }
_____unchanged_portion_omitted_


2502 /*
2503  * Initialize register handling specific to mcp51/mcp55/mcp61
2490  * Initialize register handling specific to mcp51/mcp55
2504  */
2505 /* ARGSUSED */
2506 static void
2507 mcp5x_reg_init(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle)
2508 {
2509        nv_port_t *nvp;
2510        uchar_t *bar5  = nvc->nvc_bar_addr[5];
```

```
2511        uint8_t off, port;

2513        nvc->nvc_mcp5x_ctl = (uint32_t *)(bar5 + MCP5X_CTL);
2514        nvc->nvc_mcp5x_ncq = (uint32_t *)(bar5 + MCP5X_NCQ);

2516        for (port = 0, off = 0; port < NV_MAX_PORTS(nvc); port++, off += 2) {
2517                nvp = &(nvc->nvc_port[port]);
2518                nvp->nvp_mcp5x_int_status =
2519                    (uint16_t *)(bar5 + MCP5X_INT_STATUS + off);
2520                nvp->nvp_mcp5x_int_ctl =
2521                    (uint16_t *)(bar5 + MCP5X_INT_CTL + off);

2523                /*
2524                 * clear any previous interrupts asserted
2525                 */
2526                nv_put16(nvc->nvc_bar_hdl[5], nvp->nvp_mcp5x_int_status,
2527                    MCP5X_INT_CLEAR);

2529                /*
2530                 * These are the interrupts to accept for now.  The spec
2531                 * says these are enable bits, but nvidia has indicated
2532                 * these are masking bits.  Even though they may be masked
2533                 * out to prevent asserting the main interrupt, they can
2534                 * still be asserted while reading the interrupt status
2535                 * register, so that needs to be considered in the interrupt
2536                 * handler.
2537                 */
2538                nv_put16(nvc->nvc_bar_hdl[5], nvp->nvp_mcp5x_int_ctl,
2539                    ~(MCP5X_INT_IGNORE));
2540        }

2542        /*
2543         * Allow the driver to program the BM on the first command instead
2544         * of waiting for an interrupt.
2545         */
2546 #ifdef NCQ
2547        flags = MCP_SATA_AE_NCQ_PDEV_FIRST_CMD | MCP_SATA_AE_NCQ_SDEV_FIRST_CMD;
2548        nv_put32(nvc->nvc_bar_hdl[5], nvc->nvc_mcp5x_ncq, flags);
2549        flags = MCP_SATA_AE_CTL_PRI_SWNCQ | MCP_SATA_AE_CTL_SEC_SWNCQ;
2550        nv_put32(nvc->nvc_bar_hdl[5], nvc->nvc_mcp5x_ctl, flags);
2551 #endif

2553        /*
2554         * mcp55 rev A03 and above supports 40-bit physical addressing.
2555         * Enable DMA to take advantage of that.
2556         *
2557         */
2558        if ((nvc->nvc_devid > 0x37f) ||
2559            ((nvc->nvc_devid == 0x37f) && (nvc->nvc_revid >= 0xa3))) {
2545        if (nvc->nvc_revid >= 0xa3) {
2560                if (nv_sata_40bit_dma == B_TRUE) {
2561                        uint32_t reg32;
2562                        NVLOG(NVDBG_INIT, nvp->nvp_ctlp, nvp,
2563                            "devid is %X revid is %X. 40-bit DMA"
2564                            " addressing enabled", nvc->nvc_devid,
2565                            nvc->nvc_revid);
2549                        "rev id is %X.  40-bit DMA addressing"
2550                        " enabled", nvc->nvc_revid);
2566                        nvc->dma_40bit = B_TRUE;

2568                        reg32 = pci_config_get32(pci_conf_handle,
2569                            NV_SATA_CFG_20);
2570                        pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
2571                            reg32 | NV_40BIT_PRD);

2573                        /*
```

```
2574                                  * CFG_23 bits 0-7 contain the top 8 bits (of 40
2575                                  * bits) for the primary PRD table, and bits 8-15
2576                                  * contain the top 8 bits for the secondary.  Set
2577                                  * to zero because the DMA attribute table for PRD
2578                                  * allocation forces it into 32 bit address space
2579                                  * anyway.
2580                                  */
2581                                 reg32 = pci_config_get32(pci_conf_handle,
2582                                     NV_SATA_CFG_23);
2583                                 pci_config_put32(pci_conf_handle, NV_SATA_CFG_23,
2584                                     reg32 & 0xffff0000);
2585                         } else {
2586                                 NVLOG(NVDBG_INIT, nvp->nvp_ctlp, nvp,
2587                                     "40-bit DMA disabled by nv_sata_40bit_dma", NULL);
2588                         }
2589                 } else {
2590                         nv_cmn_err(CE_NOTE, nvp->nvp_ctlp, nvp, "devid is %X revid is"
2591                             " %X. Not capable of 40-bit DMA addressing",
2592                             nvc->nvc_devid, nvc->nvc_revid);
2575                         nv_cmn_err(CE_NOTE, nvp->nvp_ctlp, nvp, "rev id is %X and is "
2576                             "not capable of 40-bit DMA addressing", nvc->nvc_revid);
2593                 }
2594 }
_____unchanged_portion_omitted_


2645 /*
2646  * Initialize the controller and set up driver data structures.
2647  * determine if ck804 or mcp5x class.
2648  */
2649 static int
2650 nv_init_ctl(nv_ctl_t *nvc, ddi_acc_handle_t pci_conf_handle)
2651 {
2652         struct sata_hba_tran stran;
2653         nv_port_t *nvp;
2654         int j;
2638         int j, ck804;
2655         uchar_t *cmd_addr, *ctl_addr, *bm_addr;
2656         ddi_acc_handle_t bar5_hdl = nvc->nvc_bar_hdl[5];
2657         uchar_t *bar5  = nvc->nvc_bar_addr[5];
2658         uint32_t reg32;
2659         uint8_t reg8, reg8_save;

2661         NVLOG(NVDBG_INIT, nvc, NULL, "nv_init_ctl entered", NULL);

2647         ck804 = B_TRUE;
2648 #ifdef SGPIO_SUPPORT
2663         nvc->nvc_mcp5x_flag = B_FALSE;
2650 #endif

2665         /*
2666          * Need to set bit 2 to 1 at config offset 0x50
2667          * to enable access to the bar5 registers.
2668          */
2669         reg32 = pci_config_get32(pci_conf_handle, NV_SATA_CFG_20);
2670         if (!(reg32 & NV_BAR5_SPACE_EN)) {
2671                 pci_config_put32(pci_conf_handle, NV_SATA_CFG_20,
2672                     reg32 | NV_BAR5_SPACE_EN);
2673         }

2675         /*
2676          * Determine if this is ck804 or mcp5x.  ck804 will map in the
2677          * task file registers into bar5 while mcp5x won't.  The offset of
2678          * the task file registers in mcp5x's space is unused, so it will
2679          * return zero.  So check one of the task file registers to see if it is
2680          * writable and reads back what was written.  If it's mcp5x it will
```

```
2681          * return back 0xff whereas ck804 will return the value written.
2682          */
2683         reg8_save = nv_get8(bar5_hdl,
2684             (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X));


2687         for (j = 1; j < 3; j++) {

2689                 nv_put8(bar5_hdl, (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X), j);
2690                 reg8 = nv_get8(bar5_hdl,
2691                     (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X));

2693                 if (reg8 != j) {
2681                         ck804 = B_FALSE;
2694                         nvc->nvc_mcp5x_flag = B_TRUE;
2695                         break;
2696                 }
2697         }

2699         nv_put8(bar5_hdl, (uint8_t *)(bar5 + NV_BAR5_TRAN_LEN_CH_X), reg8_save);

2701         if (nvc->nvc_mcp5x_flag == B_FALSE) {
2702                 NVLOG(NVDBG_INIT, nvc, NULL, "controller is CK804/MCP04",
2703                     NULL);
2689         if (ck804 == B_TRUE) {
2690                 NVLOG(NVDBG_INIT, nvc, NULL, "controller is CK804", NULL);
2704                 nvc->nvc_interrupt = ck804_intr;
2705                 nvc->nvc_reg_init = ck804_reg_init;
2706                 nvc->nvc_set_intr = ck804_set_intr;
2707         } else {
2708                 NVLOG(NVDBG_INIT, nvc, NULL, "controller is MCP51/MCP55/MCP61",
2709                     NULL);
2695                 NVLOG(NVDBG_INIT, nvc, NULL, "controller is MCP51/MCP55", NULL);
2710                 nvc->nvc_interrupt = mcp5x_intr;
2711                 nvc->nvc_reg_init = mcp5x_reg_init;
2712                 nvc->nvc_set_intr = mcp5x_set_intr;
2713         }


2716         stran.sata_tran_hba_rev = SATA_TRAN_HBA_REV;
2717         stran.sata_tran_hba_dip = nvc->nvc_dip;
2718         stran.sata_tran_hba_num_cports = NV_NUM_PORTS;
2719         stran.sata_tran_hba_features_support =
2720             SATA_CTLF_HOTPLUG | SATA_CTLF_ASN | SATA_CTLF_ATAPI;
2721         stran.sata_tran_hba_qdepth = NV_QUEUE_SLOTS;
2722         stran.sata_tran_probe_port = nv_sata_probe;
2723         stran.sata_tran_start = nv_sata_start;
2724         stran.sata_tran_abort = nv_sata_abort;
2725         stran.sata_tran_reset_dport = nv_sata_reset;
2726         stran.sata_tran_selftest = NULL;
2727         stran.sata_tran_hotplug_ops = &nv_hotplug_ops;
2728         stran.sata_tran_pwrmgt_ops = NULL;
2729         stran.sata_tran_ioctl = NULL;
2730         nvc->nvc_sata_hba_tran = stran;

2732         nvc->nvc_port = kmem_zalloc(sizeof (nv_port_t) * NV_MAX_PORTS(nvc),
2733             KM_SLEEP);

2735         /*
2736          * initialize registers common to all chipsets
2737          */
2738         nv_common_reg_init(nvc);

2740         for (j = 0; j < NV_MAX_PORTS(nvc); j++) {
2741                 nvp = &(nvc->nvc_port[j]);
```

```
2743                    cmd_addr = nvp->nvp_cmd_addr;
2744                    ctl_addr = nvp->nvp_ctl_addr;
2745                    bm_addr = nvp->nvp_bm_addr;

2747                    mutex_init(&nvp->nvp_mutex, NULL, MUTEX_DRIVER,
2748                        DDI_INTR_PRI(nvc->nvc_intr_pri));

2750                    cv_init(&nvp->nvp_sync_cv, NULL, CV_DRIVER, NULL);
2751                    cv_init(&nvp->nvp_reset_cv, NULL, CV_DRIVER, NULL);

2753                    nvp->nvp_data   = cmd_addr + NV_DATA;
2754                    nvp->nvp_error  = cmd_addr + NV_ERROR;
2755                    nvp->nvp_feature = cmd_addr + NV_FEATURE;
2756                    nvp->nvp_count  = cmd_addr + NV_COUNT;
2757                    nvp->nvp_sect   = cmd_addr + NV_SECT;
2758                    nvp->nvp_lcyl   = cmd_addr + NV_LCYL;
2759                    nvp->nvp_hcyl   = cmd_addr + NV_HCYL;
2760                    nvp->nvp_drvhd  = cmd_addr + NV_DRVHD;
2761                    nvp->nvp_status = cmd_addr + NV_STATUS;
2762                    nvp->nvp_cmd    = cmd_addr + NV_CMD;
2763                    nvp->nvp_altstatus = ctl_addr + NV_ALTSTATUS;
2764                    nvp->nvp_devctl = ctl_addr + NV_DEVCTL;

2766                    nvp->nvp_bmicx  = bm_addr + BMICX_REG;
2767                    nvp->nvp_bmisx  = bm_addr + BMISX_REG;
2768                    nvp->nvp_bmidtpx = (uint32_t *)(bm_addr + BMIDTPX_REG);

2770                    nvp->nvp_state = 0;

2772                    /*
2773                     * Initialize dma handles, etc.
2774                     * If it fails, the port is in inactive state.
2775                     */
2776                    nv_init_port(nvp);
2777            }

2779            /*
2780             * initialize register by calling chip specific reg initialization
2781             */
2782            (*(nvc->nvc_reg_init))(nvc, pci_conf_handle);

2784            /* initialize the hba dma attribute */
2785            if (nvc->dma_40bit == B_TRUE)
2786                    nvc->nvc_sata_hba_tran.sata_tran_hba_dma_attr =
2787                        &buffer_dma_40bit_attr;
2788            else
2789                    nvc->nvc_sata_hba_tran.sata_tran_hba_dma_attr =
2790                        &buffer_dma_attr;

2792            return (NV_SUCCESS);
2793 }
```
_____unchanged_portion_omitted_

```
**********************************************************
   20356 Wed Sep 28 11:57:35 2011
new/usr/src/uts/common/sys/sata/adapters/nv_sata/nv_sata.h
1023 nv_sata support for NVIDIA MCP61
**********************************************************
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 #ifndef _NV_SATA_H
  27 #define _NV_SATA_H


  30 #ifdef  __cplusplus
  31 extern "C" {
  32 #endif


  35 /*
  36  * SGPIO Support
  37  * Enable SGPIO support only on x86/x64, because it is implemented using
  38  * functions that are only available on x86/x64.
  39  */

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

  43 typedef struct nv_port nv_port_t;

  45 #ifdef SGPIO_SUPPORT
  46 typedef struct nv_sgp_cmn nv_sgp_cmn_t;
  47 #endif

  49 /*
  50  * sizes of strings to allocate
  51  */
  52 #define NV_STR_LEN       10
  53 #define NV_LOGBUF_LEN    512
  54 #define NV_REASON_LEN    30


  57 typedef struct nv_ctl {
  58         /*
  59          * Each of these are specific to the chipset in use.
  60          */
  61         uint_t          (*nvc_interrupt)(caddr_t arg1, caddr_t arg2);
```

```
  62         void            (*nvc_reg_init)(struct nv_ctl *nvc,
  63                             ddi_acc_handle_t pci_conf_handle);

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

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

  69         /*
  70          * handle and base address to register space.
  71          *
  72          * 0: port 0 task file
  73          * 1: port 0 status
  74          * 2: port 1 task file
  75          * 3: port 1 status
  76          * 4: bus master for both ports
  77          * 5: extended registers for SATA features
  78          */
  79         ddi_acc_handle_t nvc_bar_hdl[6];
  80         uchar_t         *nvc_bar_addr[6];

  82         /*
  83          * sata registers in bar 5 which are shared on all devices
  84          * on the channel.
  85          */
  86         uint32_t        *nvc_mcp5x_ctl;
  87         uint32_t        *nvc_mcp5x_ncq; /* NCQ status control bits */

  89         kmutex_t        nvc_mutex; /* ctrl level lock */

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

  99         sata_hba_tran_t nvc_sata_hba_tran; /* sata_hba_tran for ctrl */

 101         /*
 102          * enable/disable interrupts, controller specific
 103          */
 104         void            (*nvc_set_intr)(nv_port_t *nvp, int flag);
 105         int             nvc_state;      /* state flags of ctrl see below */
 106         uint16_t        nvc_devid;      /* PCI devid of device */
 107 #endif /* ! codereview */
 108         uint8_t         nvc_revid;      /* PCI revid of device */
 109         boolean_t       dma_40bit;      /* 40bit DMA support */
 110         boolean_t       nvc_mcp5x_flag; /* is the controller MCP51/MCP55 */
 111 #endif /* ! codereview */

 113 #ifdef SGPIO_SUPPORT
 106         int             nvc_mcp5x_flag; /* is the controller MCP51/MCP55 */
 114         uint8_t         nvc_ctlr_num;   /* controller number within the part */
 115         uint32_t        nvc_sgp_csr;    /* SGPIO CSR i/o address */
 116         volatile nv_sgp_cb_t *nvc_sgp_cbp; /* SGPIO Control Block */
 117         nv_sgp_cmn_t    *nvc_sgp_cmn;   /* SGPIO shared data */
 118 #endif
 119 } nv_ctl_t;
_____unchanged_portion_omitted_
```