```
**********************************************************
    4548 Fri Jan 11 13:34:53 2013
new/usr/src/pkg/manifests/driver-network-bge.mf
Just the 5719/5720 changes
**********************************************************
```

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

#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
#

#
# Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
#

# The default for payload-bearing actions in this package is to appear in the
# global zone only.  See the include file for greater detail, as well as
# information about overriding the defaults.
#
<include global_zone_only_component>
set name=pkg.fmri value=pkg:/driver/network/bge@$(PKGVERS)
set name=pkg.description \
    value="Broadcom 57xx Gigabit Ethernet Network Adapter Driver"
set name=pkg.summary value="Broadcom 57xx GE NIC Driver"
set name=info.classification \
    value=org.opensolaris.category.2008:Drivers/Networking
set name=variant.arch value=$(ARCH)
dir path=kernel group=sys
dir path=kernel/drv group=sys
dir path=kernel/drv/$(ARCH64) group=sys
dir path=usr/share/man
dir path=usr/share/man/man7d
$(i386_ONLY)driver name=bge clone_perms="bge 0666 root sys" \
    perms="* 0666 root sys" \
    alias=SUNW,bge \
    alias=pci108e,1647 \
    alias=pci108e,1648 \
    alias=pci108e,16a7 \
    alias=pci108e,16a8 \
    alias=pci14e4,1600 \
    alias=pci14e4,1601 \
    alias=pci14e4,1644 \
    alias=pci14e4,1645 \
    alias=pci14e4,1647 \
    alias=pci14e4,1648 \
    alias=pci14e4,1649 \
    alias=pci14e4,1653 \
```



```
    alias=pci14e4,1654 \
    alias=pci14e4,1657 \
    alias=pci14e4,1659 \
    alias=pci14e4,165d \
    alias=pci14e4,165e \
    alias=pci14e4,165f \
    alias=pci14e4,1668 \
    alias=pci14e4,1669 \
    alias=pci14e4,166a \
    alias=pci14e4,166e \
    alias=pci14e4,1677 \
    alias=pci14e4,1678 \
    alias=pci14e4,1679 \
    alias=pci14e4,167d \
    alias=pci14e4,1693 \
    alias=pci14e4,1696 \
    alias=pci14e4,1699 \
    alias=pci14e4,169b \
    alias=pci14e4,169c \
    alias=pci14e4,16a6 \
    alias=pci14e4,16a7 \
    alias=pci14e4,16a8 \
    alias=pci14e4,16c7 \
    alias=pciex14e4,1655 \
    alias=pciex14e4,1656 \
    alias=pciex14e4,1657 \
    alias=pciex14e4,165a \
    alias=pciex14e4,165b \
    alias=pciex14e4,165c \
    alias=pciex14e4,165f \
    alias=pciex14e4,1673 \
    alias=pciex14e4,1674 \
    alias=pciex14e4,1677 \
    alias=pciex14e4,167a \
    alias=pciex14e4,167b \
    alias=pciex14e4,1680 \
    alias=pciex14e4,1681 \
    alias=pciex14e4,1684 \
    alias=pciex14e4,1688 \
    alias=pciex14e4,1689 \
    alias=pciex14e4,1690 \
    alias=pciex14e4,1691 \
    alias=pciex14e4,1692 \
    alias=pciex14e4,1694 \
    alias=pciex14e4,1698 \
    alias=pciex14e4,169d \
    alias=pciex14e4,16fd \
    alias=pciex14e4,1713
$(sparc_ONLY)driver name=bge clone_perms="bge 0666 root sys" \
    perms="* 0666 root sys" \
    alias=SUNW,bge \
    alias=pci108e,1647 \
    alias=pci108e,1648 \
    alias=pci108e,16a7 \
    alias=pci108e,16a8 \
    alias=pci14e4,1645 \
    alias=pci14e4,1647 \
    alias=pci14e4,1648 \
    alias=pci14e4,1649 \
    alias=pci14e4,1668 \
    alias=pci14e4,1669 \
    alias=pci14e4,1677 \
    alias=pci14e4,1678 \
    alias=pci14e4,167d \
    alias=pci14e4,16a7 \
    alias=pci14e4,16a8 \
```

```
128     alias=pci14e4,16c7 \
129     alias=pciex14e4,1655 \
130     alias=pciex14e4,1656 \
131     alias=pciex14e4,1659 \
132     alias=pciex14e4,165a \
133     alias=pciex14e4,165b \
134     alias=pciex14e4,165c \
135     alias=pciex14e4,1677 \
136     alias=pciex14e4,167a \
137     alias=pciex14e4,167b
138 file path=kernel/drv/$(ARCH64)/bge group=sys
139 $(i386_ONLY)file path=kernel/drv/bge group=sys
140 file path=kernel/drv/bge.conf group=sys
141 file path=usr/share/man/man7d/bge.7d
142 legacy pkg=SUNWbge \
143     desc="Broadcom 57xx Gigabit Ethernet Network Adapter Driver" \
144     name="Broadcom 57xx GE NIC Driver"
145 license cr_Sun license=cr_Sun
146 license lic_CDDL license=lic_CDDL
147 license usr/src/uts/common/io/bge/THIRDPARTYLICENSE \
148     license=usr/src/uts/common/io/bge/THIRDPARTYLICENSE
```

```
**********************************************************
  175078 Fri Jan 11 13:34:54 2013
new/usr/src/uts/common/io/bge/bge_chip2.c
Just the 5719/5720 changes
**********************************************************
```

```
/*
 * 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) 2002, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2011, 2012 Nexenta Systems, Inc.  All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 */

#include "bge_impl.h"

#define PIO_ADDR(bgep, offset)  ((void *)((caddr_t)(bgep)->io_regs+(offset)))

/*
 * Future features ... ?
 */
#define BGE_CFG_IO8     1       /* 8/16-bit cfg space BIS/BIC   */
#define BGE_IND_IO32    1       /* indirect access code         */
#define BGE_SEE_IO32    1       /* SEEPROM access code          */
#define BGE_FLASH_IO32  1       /* FLASH access code            */

/*
 * BGE MSI tunable:
 *
 * By default MSI is enabled on all supported platforms but it is disabled
 * for some Broadcom chips due to known MSI hardware issues. Currently MSI
 * is enabled only for 5714C A2 and 5715C A2 broadcom chips.
 */
boolean_t bge_enable_msi = B_TRUE;

/*
 * PCI-X/PCI-E relaxed ordering tunable for OS/Nexus driver
 */
boolean_t bge_relaxed_ordering = B_TRUE;

/*
 * Property names
 */
static char knownids_propname[] = "bge-known-subsystems";
```



```
/*
 * Patchable globals:
 *
 *      bge_autorecover
 *              Enables/disables automatic recovery after fault detection
 *
 *      bge_mlcr_default
 *              Value to program into the MLCR; controls the chip's GPIO pins
 *
 *      bge_dma_{rd,wr}prio
 *              Relative priorities of DMA reads & DMA writes respectively.
 *              These may each be patched to any value 0-3.  Equal values
 *              will give "fair" (round-robin) arbitration for PCI access.
 *              Unequal values will give one or the other function priority.
 *
 *      bge_dma_rwctrl
 *              Value to put in the Read/Write DMA control register.  See
 *              the Broadcom PRM for things you can fiddle with in this
 *              register ...
 *
 *      bge_{tx,rx}_{count,ticks}_{norm,intr}
 *              Send/receive interrupt coalescing parameters.  Counts are
 *              #s of descriptors, ticks are in microseconds.  *norm* values
 *              apply between status updates/interrupts; the *intr* values
 *              refer to the 'during-interrupt' versions - see the PRM.
 *
 *              NOTE: these values have been determined by measurement. They
 *              differ significantly from the values recommended in the PRM.
 */
static uint32_t bge_autorecover = 1;
static uint32_t bge_mlcr_default_5714 = MLCR_DEFAULT_5714;

static uint32_t bge_dma_rdprio = 1;
static uint32_t bge_dma_wrprio = 0;
static uint32_t bge_dma_rwctrl = PDRWCR_VAR_DEFAULT;
static uint32_t bge_dma_rwctrl_5721 = PDRWCR_VAR_5721;
static uint32_t bge_dma_rwctrl_5714 = PDRWCR_VAR_5714;
static uint32_t bge_dma_rwctrl_5715 = PDRWCR_VAR_5715;

uint32_t bge_rx_ticks_norm = 128;
uint32_t bge_tx_ticks_norm = 2048;              /* 8 for FJ2+ !?!?      */
uint32_t bge_rx_count_norm = 8;
uint32_t bge_tx_count_norm = 128;

static uint32_t bge_rx_ticks_intr = 128;
static uint32_t bge_tx_ticks_intr = 0;          /* 8 for FJ2+ !?!?      */
static uint32_t bge_rx_count_intr = 2;
static uint32_t bge_tx_count_intr = 0;

/*
 * Memory pool configuration parameters.
 *
 * These are generally specific to each member of the chip family, since
 * each one may have a different memory size/configuration.
 *
 * Setting the mbuf pool length for a specific type of chip to 0 inhibits
 * the driver from programming the various registers; instead they are left
 * at their hardware defaults.  This is the preferred option for later chips
 * (5705+), whereas the older chips *required* these registers to be set,
 * since the h/w default was 0 ;-(
 */
static uint32_t bge_mbuf_pool_base      = MBUF_POOL_BASE_DEFAULT;
static uint32_t bge_mbuf_pool_base_5704 = MBUF_POOL_BASE_5704;
static uint32_t bge_mbuf_pool_base_5705 = MBUF_POOL_BASE_5705;
static uint32_t bge_mbuf_pool_base_5721 = MBUF_POOL_BASE_5721;
static uint32_t bge_mbuf_pool_len       = MBUF_POOL_LENGTH_DEFAULT;
```

```
127 static uint32_t bge_mbuf_pool_len_5704  = MBUF_POOL_LENGTH_5704;
128 static uint32_t bge_mbuf_pool_len_5705  = 0;    /* use h/w default      */
129 static uint32_t bge_mbuf_pool_len_5721  = 0;

131 /*
132  * Various high and low water marks, thresholds, etc ...
133  *
134  * Note: these are taken from revision 7 of the PRM, and some are different
135  * from both the values in earlier PRMs *and* those determined experimentally
136  * and used in earlier versions of this driver ...
137  */
138 static uint32_t bge_mbuf_hi_water       = MBUF_HIWAT_DEFAULT;
139 static uint32_t bge_mbuf_lo_water_rmac  = MAC_RX_MBUF_LOWAT_DEFAULT;
140 static uint32_t bge_mbuf_lo_water_rdma  = RDMA_MBUF_LOWAT_DEFAULT;

142 static uint32_t bge_dmad_lo_water       = DMAD_POOL_LOWAT_DEFAULT;
143 static uint32_t bge_dmad_hi_water       = DMAD_POOL_HIWAT_DEFAULT;
144 static uint32_t bge_lowat_recv_frames   = LOWAT_MAX_RECV_FRAMES_DEFAULT;

146 static uint32_t bge_replenish_std       = STD_RCV_BD_REPLENISH_DEFAULT;
147 static uint32_t bge_replenish_mini      = MINI_RCV_BD_REPLENISH_DEFAULT;
148 static uint32_t bge_replenish_jumbo     = JUMBO_RCV_BD_REPLENISH_DEFAULT;

150 static uint32_t bge_watchdog_count      = 1 << 16;
151 static uint16_t bge_dma_miss_limit      = 20;

153 static uint32_t bge_stop_start_on_sync  = 0;

155 /*
156  * bge_intr_max_loop controls the maximum loop number within bge_intr.
157  * When loading NIC with heavy network traffic, it is useful.
158  * Increasing this value could have positive effect to throughput,
159  * but it might also increase ticks of a bge ISR stick on CPU, which might
160  * lead to bad UI interactive experience. So tune this with caution.
161  */
162 static int bge_intr_max_loop = 1;

164 /*
165  * ========== Low-level chip & ring buffer manipulation ==========
166  */

168 #define BGE_DBG         BGE_DBG_REGS    /* debug flag for this code     */


171 /*
172  * Config space read-modify-write routines
173  */

175 #if     BGE_CFG_IO8

177 static void bge_cfg_clr16(bge_t *bgep, bge_regno_t regno, uint16_t bits);
178 #pragma inline(bge_cfg_clr16)

180 static void
181 bge_cfg_clr16(bge_t *bgep, bge_regno_t regno, uint16_t bits)
182 {
183         uint16_t regval;

185         BGE_TRACE(("bge_cfg_clr16($%p, 0x%lx, 0x%x)",
186             (void *)bgep, regno, bits));

188         regval = pci_config_get16(bgep->cfg_handle, regno);

190         BGE_DEBUG(("bge_cfg_clr16($%p, 0x%lx, 0x%x): 0x%x => 0x%x",
191             (void *)bgep, regno, bits, regval, regval & ~bits));
```



```
193         regval &= ~bits;
194         pci_config_put16(bgep->cfg_handle, regno, regval);
195 }
```

unchanged_portion_omitted

```
296 #endif  /* BGE_DEBUGGING */

298 /*
299  * Perform first-stage chip (re-)initialisation, using only config-space
300  * accesses:
301  *
302  * + Read the vendor/device/revision/subsystem/cache-line-size registers,
303  *   returning the data in the structure pointed to by <idp>.
304  * + Configure the target-mode endianness (swap) options.
305  * + Disable interrupts and enable Memory Space accesses.
306  * + Enable or disable Bus Mastering according to the <enable_dma> flag.
307  *
308  * This sequence is adapted from Broadcom document 570X-PG102-R,
309  * page 102, steps 1-3, 6-8 and 11-13.  The omitted parts of the sequence
310  * are 4 and 5 (Reset Core and wait) which are handled elsewhere.
311  *
312  * This function MUST be called before any non-config-space accesses
313  * are made; on this first call <enable_dma> is B_FALSE, and it
314  * effectively performs steps 3-1(!) of the initialisation sequence
315  * (the rest are not required but should be harmless).
316  *
317  * It MUST also be called after a chip reset, as this disables
318  * Memory Space cycles!  In this case, <enable_dma> is B_TRUE, and
319  * it is effectively performing steps 6-8.
320  */
321 void bge_chip_cfg_init(bge_t *bgep, chip_id_t *cidp, boolean_t enable_dma);
322 #pragma no_inline(bge_chip_cfg_init)

324 void
325 bge_chip_cfg_init(bge_t *bgep, chip_id_t *cidp, boolean_t enable_dma)
326 {
327         ddi_acc_handle_t handle;
328         uint16_t command;
329         uint32_t mhcr;
330         uint16_t value16;
331         int i;

333         BGE_TRACE(("bge_chip_cfg_init($%p, $%p, %d)",
334             (void *)bgep, (void *)cidp, enable_dma));

336         /*
337          * Step 3: save PCI cache line size and subsystem vendor ID
338          *
339          * Read all the config-space registers that characterise the
340          * chip, specifically vendor/device/revision/subsystem vendor
341          * and subsystem device id.  We expect (but don't check) that
342          * (vendor == VENDOR_ID_BROADCOM) && (device == DEVICE_ID_5704)
343          *
344          * Also save all bus-transaction related registers (cache-line
345          * size, bus-grant/latency parameters, etc).  Some of these are
346          * cleared by reset, so we'll have to restore them later.  This
347          * comes from the Broadcom document 570X-PG102-R ...
348          *
349          * Note: Broadcom document 570X-PG102-R seems to be in error
350          * here w.r.t. the offsets of the Subsystem Vendor ID and
351          * Subsystem (Device) ID registers, which are the opposite way
352          * round according to the PCI standard.  For good measure, we
353          * save/restore both anyway.
354          */
355         handle = bgep->cfg_handle;
```

```
357         /*
358          * For some chipsets (e.g., BCM5718), if MHCR_ENABLE_ENDIAN_BYTE_SWAP
359          * has been set in PCI_CONF_COMM already, we need to write the
360          * byte-swapped value to it. So we just write zero first for simplicity.
361          */
362         cidp->device = pci_config_get16(handle, PCI_CONF_DEVID);
363         if (DEVICE_5717_SERIES_CHIPSETS(bgep))
364                 pci_config_put32(handle, PCI_CONF_BGE_MHCR, 0);
365         mhcr = pci_config_get32(handle, PCI_CONF_BGE_MHCR);
366         cidp->asic_rev = (mhcr & MHCR_CHIP_REV_MASK) >> MHCR_CHIP_REV_SHIFT;
367         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_PRODID) {
368                 uint32_t reg;
369                 switch (cidp->device) {
370                 case DEVICE_ID_5717:
371                 case DEVICE_ID_5718:
372                 case DEVICE_ID_5719:
373                 case DEVICE_ID_5720:
374                         reg = PCI_CONF_GEN2_PRODID_ASICREV;
375                         break;
376                 case DEVICE_ID_57781:
377                 case DEVICE_ID_57785:
378                 case DEVICE_ID_57761:
379                 case DEVICE_ID_57765:
380                 case DEVICE_ID_57791:
381                 case DEVICE_ID_57795:
382                 case DEVICE_ID_57762:
383                 case DEVICE_ID_57766:
384                 case DEVICE_ID_57782:
385                 case DEVICE_ID_57786:
386                         reg = PCI_CONF_GEN15_PRODID_ASICREV;
387                         break;
388                 default:
389                         reg = PCI_CONF_PRODID_ASICREV;
390                         break;
391                 }
392                 cidp->asic_rev = pci_config_get32(handle, reg);
393         }
366         cidp->asic_rev = mhcr & MHCR_CHIP_REV_MASK;
394         cidp->businfo = pci_config_get32(handle, PCI_CONF_BGE_PCISTATE);
395         cidp->command = pci_config_get16(handle, PCI_CONF_COMM);

397         cidp->vendor = pci_config_get16(handle, PCI_CONF_VENID);
398         cidp->subven = pci_config_get16(handle, PCI_CONF_SUBVENID);
399         cidp->subdev = pci_config_get16(handle, PCI_CONF_SUBSYSID);
400         cidp->revision = pci_config_get8(handle, PCI_CONF_REVID);
401         cidp->clsize = pci_config_get8(handle, PCI_CONF_CACHE_LINESZ);
402         cidp->latency = pci_config_get8(handle, PCI_CONF_LATENCY_TIMER);

404         BGE_DEBUG(("bge_chip_cfg_init: %s bus is %s and %s; #INTA is %s",
405             cidp->businfo & PCISTATE_BUS_IS_PCI ? "PCI" : "PCI-X",
406             cidp->businfo & PCISTATE_BUS_IS_FAST ? "fast" : "slow",
407             cidp->businfo & PCISTATE_BUS_IS_32_BIT ? "narrow" : "wide",
408             cidp->businfo & PCISTATE_INTA_STATE ? "high" : "low"));
409         BGE_DEBUG(("bge_chip_cfg_init: vendor 0x%x device 0x%x revision 0x%x",
410             cidp->vendor, cidp->device, cidp->revision));
411         BGE_DEBUG(("bge_chip_cfg_init: subven 0x%x subdev 0x%x asic_rev 0x%x",
412             cidp->subven, cidp->subdev, cidp->asic_rev));
413         BGE_DEBUG(("bge_chip_cfg_init: clsize %d latency %d command 0x%x",
414             cidp->clsize, cidp->latency, cidp->command));

416         cidp->chip_type = 0;
417         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5717 ||
418             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5719 ||
419             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5720)
420                 cidp->chip_type |= CHIP_TYPE_5717_PLUS;
```



```
422         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_57765 ||
423             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_57766)
424                 cidp->chip_type |= CHIP_TYPE_57765_CLASS;

426         if (cidp->chip_type & CHIP_TYPE_57765_CLASS ||
427             cidp->chip_type & CHIP_TYPE_5717_PLUS)
428                 cidp->chip_type |= CHIP_TYPE_57765_PLUS;

430         /* Intentionally exclude ASIC_REV_5906 */
431         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5755 ||
432             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5787 ||
433             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5784 ||
434             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5761 ||
435             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5785 ||
436             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_57780 ||
437             cidp->chip_type & CHIP_TYPE_57765_PLUS)
438                 cidp->chip_type |= CHIP_TYPE_5755_PLUS;

440         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5780 ||
441             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5714)
442                 cidp->chip_type |= CHIP_TYPE_5780_CLASS;

444         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5750 ||
445             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5752 ||
446             MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5906 ||
447             cidp->chip_type & CHIP_TYPE_5755_PLUS ||
448             cidp->chip_type & CHIP_TYPE_5780_CLASS)
449                 cidp->chip_type |= CHIP_TYPE_5750_PLUS;

451         if (MHCR_CHIP_ASIC_REV(cidp->asic_rev) == MHCR_CHIP_ASIC_REV_5705 ||
452             cidp->chip_type & CHIP_TYPE_5750_PLUS)
453                 cidp->chip_type |= CHIP_TYPE_5705_PLUS;

455         /*
456          * Step 2 (also step 6): disable and clear interrupts.
457          * Steps 11-13: configure PIO endianness options, and enable
458          * indirect register access.  We'll also select any other
459          * options controlled by the MHCR (e.g. tagged status, mask
460          * interrupt mode) at this stage ...
461          *
462          * Note: internally, the chip is 64-bit and BIG-endian, but
463          * since it talks to the host over a (LITTLE-endian) PCI bus,
464          * it normally swaps bytes around at the PCI interface.
465          * However, the PCI host bridge on SPARC systems normally
466          * swaps the byte lanes around too, since SPARCs are also
467          * BIG-endian.  So it turns out that on SPARC, the right
468          * option is to tell the chip to swap (and the host bridge
469          * will swap back again), whereas on x86 we ask the chip
470          * NOT to swap, so the natural little-endianness of the
471          * PCI bus is assumed.  Then the only thing that doesn't
472          * automatically work right is access to an 8-byte register
473          * by a little-endian host; but we don't want to set the
474          * MHCR_ENABLE_REGISTER_WORD_SWAP bit because then 4-byte
475          * accesses don't go where expected ;-(  So we live with
476          * that, and perform word-swaps in software in the few cases
477          * where a chip register is defined as an 8-byte value --
478          * see the code below for details ...
479          *
480          * Note: the meaning of the 'MASK_INTERRUPT_MODE' bit isn't
481          * very clear in the register description in the PRM, but
482          * Broadcom document 570X-PG104-R page 248 explains a little
483          * more (under "Broadcom Mask Mode").  The bit changes the way
484          * the MASK_PCI_INT_OUTPUT bit works: with MASK_INTERRUPT_MODE
485          * clear, the chip interprets MASK_PCI_INT_OUTPUT in the same
486          * way as the 5700 did, which isn't very convenient.  Setting
487          * the MASK_INTERRUPT_MODE bit makes the MASK_PCI_INT_OUTPUT
```

```
488          * bit do just what its name says -- MASK the PCI #INTA output
489          * (i.e. deassert the signal at the pin) leaving all internal
490          * state unchanged.  This is much more convenient for our
491          * interrupt handler, so we set MASK_INTERRUPT_MODE here.
492          *
493          * Note: the inconvenient semantics of the interrupt mailbox
494          * (nonzero disables and acknowledges/clears the interrupt,
495          * zero enables AND CLEARS it) would make race conditions
496          * likely in the interrupt handler:
497          *
498          * (1)  acknowledge & disable interrupts
499          * (2)  while (more to do)
500          *              process packets
501          * (3)  enable interrupts -- also clears pending
502          *
503          * If the chip received more packets and internally generated
504          * an interrupt between the check at (2) and the mbox write
505          * at (3), this interrupt would be lost :-(
506          *
507          * The best way to avoid this is to use TAGGED STATUS mode,
508          * where the chip includes a unique tag in each status block
509          * update, and the host, when re-enabling interrupts, passes
510          * the last tag it saw back to the chip; then the chip can
511          * see whether the host is truly up to date, and regenerate
512          * its interrupt if not.
513          */
514         mhcr = MHCR_ENABLE_INDIRECT_ACCESS |
515             MHCR_ENABLE_TAGGED_STATUS_MODE |
516             MHCR_ENABLE_PCI_STATE_WRITE |
517             MHCR_MASK_INTERRUPT_MODE |
518             MHCR_CLEAR_INTERRUPT_INTA;

520         if (bgep->intr_type == DDI_INTR_TYPE_FIXED)
521                 mhcr |= MHCR_MASK_PCI_INT_OUTPUT;

523 #ifdef  _BIG_ENDIAN
524         mhcr |= MHCR_ENABLE_ENDIAN_WORD_SWAP | MHCR_ENABLE_ENDIAN_BYTE_SWAP;
525 #endif  /* _BIG_ENDIAN */

527         if (DEVICE_5717_SERIES_CHIPSETS(bgep))
528                 pci_config_put32(handle, PCI_CONF_BGE_MHCR, 0);
529         pci_config_put32(handle, PCI_CONF_BGE_MHCR, mhcr);

531 #ifdef BGE_IPMI_ASF
532         bgep->asf_wordswapped = B_FALSE;
533 #endif
534         /*
535          * Step 1 (also step 7): Enable PCI Memory Space accesses
536          *                       Disable Memory Write/Invalidate
537          *                       Enable or disable Bus Mastering
538          *
539          * Note that all other bits are taken from the original value saved
540          * the first time through here, rather than from the current register
541          * value, 'cos that will have been cleared by a soft RESET since.
542          * In this way we preserve the OBP/nexus-parent's preferred settings
543          * of the parity-error and system-error enable bits across multiple
544          * chip RESETs.
545          */
546         command = bgep->chipid.command | PCI_COMM_MAE;
547         command &= ~(PCI_COMM_ME|PCI_COMM_MEMWR_INVAL);
548         if (enable_dma)
549                 command |= PCI_COMM_ME;
550         /*
551          * on BCM5714 revision A0, false parity error gets generated
552          * due to a logic bug. Provide a workaround by disabling parity
553          * error.
```



```
554          */
555         if (((cidp->device == DEVICE_ID_5714C) ||
556             (cidp->device == DEVICE_ID_5714S)) &&
557             (cidp->revision == REVISION_ID_5714_A0)) {
558                 command &= ~PCI_COMM_PARITY_DETECT;
559         }
560         pci_config_put16(handle, PCI_CONF_COMM, command);

562         /*
563          * On some PCI-E device, there were instances when
564          * the device was still link training.
565          */
566         if (bgep->chipid.pci_type == BGE_PCI_E) {
567                 i = 0;
568                 value16 = pci_config_get16(handle, PCI_CONF_COMM);
569                 while ((value16 != command) && (i < 100)) {
570                         drv_usecwait(200);
571                         value16 = pci_config_get16(handle, PCI_CONF_COMM);
572                         ++i;
573                 }
574         }

576         /*
577          * Clear any remaining error status bits
578          */
579         pci_config_put16(handle, PCI_CONF_STAT, ~0);

581         /*
582          * Do following if and only if the device is NOT BCM5714C OR
583          * BCM5715C
584          */
585         if (!((cidp->device == DEVICE_ID_5714C) ||
586             (cidp->device == DEVICE_ID_5715C))) {
587                 /*
588                  * Make sure these indirect-access registers are sane
589                  * rather than random after power-up or reset
590                  */
591                 pci_config_put32(handle, PCI_CONF_BGE_RIAAR, 0);
592                 pci_config_put32(handle, PCI_CONF_BGE_MWBAR, 0);
593         }
594         /*
595          * Step 8: Disable PCI-X/PCI-E Relaxed Ordering
596          */
597         bge_cfg_clr16(bgep, PCIX_CONF_COMM, PCIX_COMM_RELAXED);

599         if (cidp->pci_type == BGE_PCI_E) {
600                 if (DEVICE_5723_SERIES_CHIPSETS(bgep)) {
601                         bge_cfg_clr16(bgep, PCI_CONF_DEV_CTRL_5723,
602                             DEV_CTRL_NO_SNOOP | DEV_CTRL_RELAXED);
603                 } else
604                         bge_cfg_clr16(bgep, PCI_CONF_DEV_CTRL,
605                             DEV_CTRL_NO_SNOOP | DEV_CTRL_RELAXED);
606         }
607 }
_____unchanged_portion_omitted_

1917 #endif  /* BGE_SEE_IO32 || BGE_FLASH_IO32 */

1919 /*
1920  * Determine the type of NVmem that is (or may be) attached to this chip,
1921  */
1922 static enum bge_nvmem_type bge_nvmem_id(bge_t *bgep);
1923 #pragma no_inline(bge_nvmem_id)

1925 static enum bge_nvmem_type
1926 bge_nvmem_id(bge_t *bgep)
```

```
1927 {
1928         enum bge_nvmem_type nvtype;
1929         uint32_t config1;

1931         BGE_TRACE(("bge_nvmem_id($%p)",
1932             (void *)bgep));

1934         switch (bgep->chipid.device) {
1935         default:
1936                 /*
1937                  * We shouldn't get here; it means we don't recognise
1938                  * the chip, which means we don't know how to determine
1939                  * what sort of NVmem (if any) it has.  So we'll say
1940                  * NONE, to disable the NVmem access code ...
1941                  */
1942                 nvtype = BGE_NVTYPE_NONE;
1943                 break;

1945         case DEVICE_ID_5700:
1946         case DEVICE_ID_5700x:
1947         case DEVICE_ID_5701:
1948                 /*
1949                  * These devices support *only* SEEPROMs
1950                  */
1951                 nvtype = BGE_NVTYPE_SEEPROM;
1952                 break;

1954         case DEVICE_ID_5702:
1955         case DEVICE_ID_5702fe:
1956         case DEVICE_ID_5703C:
1957         case DEVICE_ID_5703S:
1958         case DEVICE_ID_5704C:
1959         case DEVICE_ID_5704S:
1960         case DEVICE_ID_5704:
1961         case DEVICE_ID_5705M:
1962         case DEVICE_ID_5705C:
1963         case DEVICE_ID_5705_2:
1964         case DEVICE_ID_5717:
1965         case DEVICE_ID_5718:
1966         case DEVICE_ID_5719:
1967         case DEVICE_ID_5720:
1968         case DEVICE_ID_5724:
1969         case DEVICE_ID_57760:
1970         case DEVICE_ID_57780:
1971         case DEVICE_ID_57788:
1972         case DEVICE_ID_57790:
1973         case DEVICE_ID_5780:
1974         case DEVICE_ID_5782:
1975         case DEVICE_ID_5784M:
1976         case DEVICE_ID_5785:
1977         case DEVICE_ID_5787:
1978         case DEVICE_ID_5787M:
1979         case DEVICE_ID_5788:
1980         case DEVICE_ID_5789:
1981         case DEVICE_ID_5751:
1982         case DEVICE_ID_5751M:
1983         case DEVICE_ID_5752:
1984         case DEVICE_ID_5752M:
1985         case DEVICE_ID_5754:
1986         case DEVICE_ID_5755:
1987         case DEVICE_ID_5755M:
1988         case DEVICE_ID_5756M:
1989         case DEVICE_ID_5721:
1990         case DEVICE_ID_5722:
1991         case DEVICE_ID_5723:
1992         case DEVICE_ID_5761:
```



```
1993         case DEVICE_ID_5761E:
1994         case DEVICE_ID_5761S:
1995         case DEVICE_ID_5761SE:
1996         case DEVICE_ID_5764:
1997         case DEVICE_ID_5714C:
1998         case DEVICE_ID_5714S:
1999         case DEVICE_ID_5715C:
2000         case DEVICE_ID_5715S:
2001                 config1 = bge_reg_get32(bgep, NVM_CONFIG1_REG);
2002                 if (config1 & NVM_CFG1_FLASH_MODE)
2003                         if (config1 & NVM_CFG1_BUFFERED_MODE)
2004                                 nvtype = BGE_NVTYPE_BUFFERED_FLASH;
2005                         else
2006                                 nvtype = BGE_NVTYPE_UNBUFFERED_FLASH;
2007                 else
2008                         nvtype = BGE_NVTYPE_LEGACY_SEEPROM;
2009                 break;
2010         case DEVICE_ID_5906:
2011         case DEVICE_ID_5906M:
2012                 nvtype = BGE_NVTYPE_BUFFERED_FLASH;
2013                 break;
2014         }

2016         return (nvtype);
2017 }
_____unchanged_portion_omitted_

2038 /*
2039  * Using the values captured by bge_chip_cfg_init(), and additional probes
2040  * as required, characterise the chip fully: determine the label by which
2041  * to refer to this chip, the correct settings for various registers, and
2042  * of course whether the device and/or subsystem are supported!
2043  */
2044 int bge_chip_id_init(bge_t *bgep);
2045 #pragma no_inline(bge_chip_id_init)

2047 int
2048 bge_chip_id_init(bge_t *bgep)
2049 {
2050         char buf[MAXPATHLEN];           /* any risk of stack overflow?  */
2051         boolean_t sys_ok;
2052         boolean_t dev_ok;
2053         chip_id_t *cidp;
2054         uint32_t subid;
2055         char *devname;
2056         char *sysname;
2057         int *ids;
2058         int err;
2059         uint_t i;

2061         sys_ok = dev_ok = B_FALSE;
2062         cidp = &bgep->chipid;

2064         /*
2065          * Check the PCI device ID to determine the generic chip type and
2066          * select parameters that depend on this.
2067          *
2068          * Note: because the SPARC platforms in general don't fit the
2069          * SEEPROM 'behind' the chip, the PCI revision ID register reads
2070          * as zero - which is why we use <asic_rev> rather than <revision>
2071          * below ...
2072          *
2073          * Note: in general we can't distinguish between the Copper/SerDes
2074          * versions by ID alone, as some Copper devices (e.g. some but not
2075          * all 5703Cs) have the same ID as the SerDes equivalents.  So we
2076          * treat them the same here, and the MII code works out the media
```

```
2077          * type later on ...
2078          */
2079         cidp->mbuf_base = bge_mbuf_pool_base;
2080         cidp->mbuf_length = bge_mbuf_pool_len;
2081         cidp->recv_slots = BGE_RECV_SLOTS_USED;
2082         cidp->bge_dma_rwctrl = bge_dma_rwctrl;
2083         cidp->pci_type = BGE_PCI_X;
2084         cidp->statistic_type = BGE_STAT_BLK;
2085         cidp->mbuf_lo_water_rdma = bge_mbuf_lo_water_rdma;
2086         cidp->mbuf_lo_water_rmac = bge_mbuf_lo_water_rmac;
2087         cidp->mbuf_hi_water = bge_mbuf_hi_water;
2088         cidp->rx_ticks_norm = bge_rx_ticks_norm;
2089         cidp->rx_count_norm = bge_rx_count_norm;
2090         cidp->tx_ticks_norm = bge_tx_ticks_norm;
2091         cidp->tx_count_norm = bge_tx_count_norm;
2092         cidp->mask_pci_int = MHCR_MASK_PCI_INT_OUTPUT;

2094         if (cidp->rx_rings == 0 || cidp->rx_rings > BGE_RECV_RINGS_MAX)
2095                 cidp->rx_rings = BGE_RECV_RINGS_DEFAULT;
2096         if (cidp->tx_rings == 0 || cidp->tx_rings > BGE_SEND_RINGS_MAX)
2097                 cidp->tx_rings = BGE_SEND_RINGS_DEFAULT;

2099         cidp->msi_enabled = B_FALSE;

2101         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) >
2102             MHCR_CHIP_ASIC_REV_PRODID ||
2103             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
2104             MHCR_CHIP_ASIC_REV_5906 ||
2105             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
2106             MHCR_CHIP_ASIC_REV_5700 ||
2107             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
2108             MHCR_CHIP_ASIC_REV_5701 ||
2109             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
2110             MHCR_CHIP_ASIC_REV_5750)
2111                 /*
2112                  * Just a plain reset; the "check" code breaks these chips
2113                  */
2114                 cidp->flags |= CHIP_FLAG_NO_CHECK_RESET;

2116         switch (cidp->device) {
2117         case DEVICE_ID_5717:
2118         case DEVICE_ID_5718:
2119         case DEVICE_ID_5719:
2120         case DEVICE_ID_5720:
2121         case DEVICE_ID_5724:
2122                 if (cidp->device == DEVICE_ID_5717)
2123                         cidp->chip_label = 5717;
2124                 else if (cidp->device == DEVICE_ID_5718)
2125                         cidp->chip_label = 5718;
2126                 else if (cidp->device == DEVICE_ID_5719)
2127                         cidp->chip_label = 5719;
2128                 else if (cidp->device == DEVICE_ID_5720)
2129                         cidp->chip_label = 5720;
2130                 else
2131                         cidp->chip_label = 5724;
2132                 cidp->msi_enabled = bge_enable_msi;
2133 #ifdef __sparc
2134                 cidp->mask_pci_int = LE_32(MHCR_MASK_PCI_INT_OUTPUT);
2135 #endif
2136                 cidp->bge_dma_rwctrl = LE_32(PDRWCR_VAR_5717);
2137                 cidp->pci_type = BGE_PCI_E;
2138                 cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2139                 cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5717;
2140                 cidp->mbuf_hi_water = MBUF_HIWAT_5717;
2141                 cidp->mbuf_base = bge_mbuf_pool_base_5705;
2142                 cidp->mbuf_length = bge_mbuf_pool_len_5705;
```



```
2143                 cidp->recv_slots = BGE_RECV_SLOTS_5717;
2053                 cidp->recv_slots = BGE_RECV_SLOTS_5705;
2144                 cidp->bge_mlcr_default = MLCR_DEFAULT_5717;
2145                 cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2146                 cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2147                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2148                 cidp->statistic_type = BGE_STAT_REG;
2149                 dev_ok = B_TRUE;
2150                 break;

2152         case DEVICE_ID_5700:
2153         case DEVICE_ID_5700x:
2154                 cidp->chip_label = 5700;
2155                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2156                 break;

2158         case DEVICE_ID_5701:
2159                 cidp->chip_label = 5701;
2160                 dev_ok = B_TRUE;
2161                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2162                 break;

2164         case DEVICE_ID_5702:
2165         case DEVICE_ID_5702fe:
2166                 cidp->chip_label = 5702;
2167                 dev_ok = B_TRUE;
2168                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2169                 cidp->pci_type = BGE_PCI;
2170                 break;

2172         case DEVICE_ID_5703C:
2173         case DEVICE_ID_5703S:
2174         case DEVICE_ID_5703:
2175                 /*
2176                  * Revision A0 of the 5703/5793 had various errata
2177                  * that we can't or don't work around, so it's not
2178                  * supported, but all later versions are
2179                  */
2180                 cidp->chip_label = cidp->subven == VENDOR_ID_SUN ? 5793 : 5703;
2181                 if (bgep->chipid.asic_rev != MHCR_CHIP_REV_5703_A0)
2182                         dev_ok = B_TRUE;
2183                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2184                 break;

2186         case DEVICE_ID_5704C:
2187         case DEVICE_ID_5704S:
2188         case DEVICE_ID_5704:
2189                 cidp->chip_label = cidp->subven == VENDOR_ID_SUN ? 5794 : 5704;
2190                 cidp->mbuf_base = bge_mbuf_pool_base_5704;
2191                 cidp->mbuf_length = bge_mbuf_pool_len_5704;
2192                 dev_ok = B_TRUE;
2193                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2194                 break;

2196         case DEVICE_ID_5705C:
2197         case DEVICE_ID_5705M:
2198         case DEVICE_ID_5705MA3:
2199         case DEVICE_ID_5705F:
2200         case DEVICE_ID_5705_2:
2201         case DEVICE_ID_5754:
2202                 if (cidp->device == DEVICE_ID_5754) {
2203                         cidp->chip_label = 5754;
2204                         cidp->pci_type = BGE_PCI_E;
2205                 } else {
2206                         cidp->chip_label = 5705;
2207                         cidp->pci_type = BGE_PCI;
```

```
2208                        cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2209                }
2210                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2211                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2212                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2213                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2214                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2215                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2216                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2217                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2218                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2219                cidp->statistic_type = BGE_STAT_REG;
2220                dev_ok = B_TRUE;
2221                break;

2223        case DEVICE_ID_5906:
2224        case DEVICE_ID_5906M:
2225                cidp->chip_label = 5906;
2226                cidp->pci_type = BGE_PCI_E;
2227                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5906;
2228                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5906;
2229                cidp->mbuf_hi_water = MBUF_HIWAT_5906;
2230                cidp->mbuf_base = bge_mbuf_pool_base;
2231                cidp->mbuf_length = bge_mbuf_pool_len;
2232                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2233                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2234                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2235                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2236                cidp->statistic_type = BGE_STAT_REG;
2237                dev_ok = B_TRUE;
2238                break;

2240        case DEVICE_ID_5753:
2241                cidp->chip_label = 5753;
2242                cidp->pci_type = BGE_PCI_E;
2243                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2244                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2245                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2246                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2247                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2248                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2249                cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2250                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2251                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2252                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2253                cidp->statistic_type = BGE_STAT_REG;
2254                dev_ok = B_TRUE;
2255                break;

2257        case DEVICE_ID_5755:
2258        case DEVICE_ID_5755M:
2259                cidp->chip_label = 5755;
2260                cidp->pci_type = BGE_PCI_E;
2261                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2262                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2263                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2264                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2265                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2266                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2267                cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2268                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2269                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2270                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2271                if (cidp->device == DEVICE_ID_5755M)
2272                        cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2273                cidp->statistic_type = BGE_STAT_REG;
```



```
2274                dev_ok = B_TRUE;
2275                break;

2277        case DEVICE_ID_5756M:
2278                /*
2279                 * This is nearly identical to the 5755M.
2280                 * (Actually reports the 5755 chip ID.)
2281                 */
2282                cidp->chip_label = 5756;
2283                cidp->pci_type = BGE_PCI_E;
2284                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2285                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2286                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2287                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2288                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2289                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2290                cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2291                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2292                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2293                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2294                cidp->statistic_type = BGE_STAT_REG;
2295                dev_ok = B_TRUE;
2296                break;

2298        case DEVICE_ID_5787:
2299        case DEVICE_ID_5787M:
2300                cidp->chip_label = 5787;
2301                cidp->pci_type = BGE_PCI_E;
2302                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2303                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2304                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2305                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2306                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2307                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2308                cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2309                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2310                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2311                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2312                cidp->statistic_type = BGE_STAT_REG;
2313                dev_ok = B_TRUE;
2314                break;

2316        case DEVICE_ID_5723:
2317        case DEVICE_ID_5761:
2318        case DEVICE_ID_5761E:
2319        case DEVICE_ID_5761S:
2320        case DEVICE_ID_5761SE:
2321        case DEVICE_ID_5784M:
2322        case DEVICE_ID_57760:
2323        case DEVICE_ID_57780:
2324        case DEVICE_ID_57788:
2325        case DEVICE_ID_57790:
2326                cidp->msi_enabled = bge_enable_msi;
2327                /*
2328                 * We don't use MSI for BCM5764 and BCM5785, as the
2329                 * status block may fail to update when the network
2330                 * traffic is heavy.
2331                 */
2332                /* FALLTHRU */
2333        case DEVICE_ID_5785:
2334        case DEVICE_ID_5764:
2335                if (cidp->device == DEVICE_ID_5723)
2336                        cidp->chip_label = 5723;
2337                else if (cidp->device == DEVICE_ID_5764)
2338                        cidp->chip_label = 5764;
2339                else if (cidp->device == DEVICE_ID_5784M)
```

```
2340                        cidp->chip_label = 5784;
2341                else if (cidp->device == DEVICE_ID_5785)
2342                        cidp->chip_label = 5785;
2343                else if (cidp->device == DEVICE_ID_57760)
2344                        cidp->chip_label = 57760;
2345                else if (cidp->device == DEVICE_ID_57780)
2346                        cidp->chip_label = 57780;
2347                else if (cidp->device == DEVICE_ID_57788)
2348                        cidp->chip_label = 57788;
2349                else if (cidp->device == DEVICE_ID_57790)
2350                        cidp->chip_label = 57790;
2351                else
2352                        cidp->chip_label = 5761;
2353                cidp->bge_dma_rwctrl = bge_dma_rwctrl_5721;
2354                cidp->pci_type = BGE_PCI_E;
2355                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2356                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2357                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2358                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2359                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2360                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2361                cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2362                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2363                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2364                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2365                cidp->statistic_type = BGE_STAT_REG;
2366                dev_ok = B_TRUE;
2367                break;

2369        /* PCI-X device, identical to 5714 */
2370        case DEVICE_ID_5780:
2371                cidp->chip_label = 5780;
2372                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2373                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2374                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2375                cidp->mbuf_base = bge_mbuf_pool_base_5721;
2376                cidp->mbuf_length = bge_mbuf_pool_len_5721;
2377                cidp->recv_slots = BGE_RECV_SLOTS_5721;
2378                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2379                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2380                cidp->statistic_type = BGE_STAT_REG;
2381                dev_ok = B_TRUE;
2382                break;

2384        case DEVICE_ID_5782:
2385                /*
2386                 * Apart from the label, we treat this as a 5705(?)
2387                 */
2388                cidp->chip_label = 5782;
2389                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2390                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2391                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2392                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2393                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2394                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2395                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2396                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2397                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2398                cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2399                cidp->statistic_type = BGE_STAT_REG;
2400                dev_ok = B_TRUE;
2401                break;

2403        case DEVICE_ID_5788:
2404                /*
2405                 * Apart from the label, we treat this as a 5705(?)
```



```
2406                 */
2407                cidp->chip_label = 5788;
2408                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2409                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2410                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2411                cidp->mbuf_base = bge_mbuf_pool_base_5705;
2412                cidp->mbuf_length = bge_mbuf_pool_len_5705;
2413                cidp->recv_slots = BGE_RECV_SLOTS_5705;
2414                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2415                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2416                cidp->statistic_type = BGE_STAT_REG;
2417                cidp->flags |= CHIP_FLAG_NO_JUMBO;
2418                dev_ok = B_TRUE;
2419                break;

2421        case DEVICE_ID_5714C:
2422                if (cidp->revision >= REVISION_ID_5714_A2)
2423                        cidp->msi_enabled = bge_enable_msi;
2424                /* FALLTHRU */
2425        case DEVICE_ID_5714S:
2426                cidp->chip_label = 5714;
2427                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2428                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2429                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2430                cidp->mbuf_base = bge_mbuf_pool_base_5721;
2431                cidp->mbuf_length = bge_mbuf_pool_len_5721;
2432                cidp->recv_slots = BGE_RECV_SLOTS_5721;
2433                cidp->bge_dma_rwctrl = bge_dma_rwctrl_5714;
2434                cidp->bge_mlcr_default = bge_mlcr_default_5714;
2435                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2436                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2437                cidp->pci_type = BGE_PCI_E;
2438                cidp->statistic_type = BGE_STAT_REG;
2439                dev_ok = B_TRUE;
2440                break;

2442        case DEVICE_ID_5715C:
2443        case DEVICE_ID_5715S:
2444                cidp->chip_label = 5715;
2445                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2446                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2447                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2448                cidp->mbuf_base = bge_mbuf_pool_base_5721;
2449                cidp->mbuf_length = bge_mbuf_pool_len_5721;
2450                cidp->recv_slots = BGE_RECV_SLOTS_5721;
2451                cidp->bge_dma_rwctrl = bge_dma_rwctrl_5715;
2452                cidp->bge_mlcr_default = bge_mlcr_default_5714;
2453                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2454                cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2455                cidp->pci_type = BGE_PCI_E;
2456                cidp->statistic_type = BGE_STAT_REG;
2457                if (cidp->revision >= REVISION_ID_5715_A2)
2458                        cidp->msi_enabled = bge_enable_msi;
2459                dev_ok = B_TRUE;
2460                break;

2462        case DEVICE_ID_5721:
2463                cidp->chip_label = 5721;
2464                cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2465                cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2466                cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2467                cidp->mbuf_base = bge_mbuf_pool_base_5721;
2468                cidp->mbuf_length = bge_mbuf_pool_len_5721;
2469                cidp->recv_slots = BGE_RECV_SLOTS_5721;
2470                cidp->bge_dma_rwctrl = bge_dma_rwctrl_5721;
2471                cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
```

```
2472                 cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2473                 cidp->pci_type = BGE_PCI_E;
2474                 cidp->statistic_type = BGE_STAT_REG;
2475                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2476                 dev_ok = B_TRUE;
2477                 break;

2479         case DEVICE_ID_5722:
2480                 cidp->chip_label = 5722;
2481                 cidp->pci_type = BGE_PCI_E;
2482                 cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2483                 cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2484                 cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2485                 cidp->mbuf_base = bge_mbuf_pool_base_5705;
2486                 cidp->mbuf_length = bge_mbuf_pool_len_5705;
2487                 cidp->recv_slots = BGE_RECV_SLOTS_5705;
2488                 cidp->bge_mlcr_default |= MLCR_MISC_PINS_OUTPUT_ENABLE_1;
2489                 cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2490                 cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2491                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2492                 cidp->statistic_type = BGE_STAT_REG;
2493                 dev_ok = B_TRUE;
2494                 break;

2496         case DEVICE_ID_5751:
2497         case DEVICE_ID_5751M:
2498                 cidp->chip_label = 5751;
2499                 cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2500                 cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2501                 cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2502                 cidp->mbuf_base = bge_mbuf_pool_base_5721;
2503                 cidp->mbuf_length = bge_mbuf_pool_len_5721;
2504                 cidp->recv_slots = BGE_RECV_SLOTS_5721;
2505                 cidp->bge_dma_rwctrl = bge_dma_rwctrl_5721;
2506                 cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2507                 cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2508                 cidp->pci_type = BGE_PCI_E;
2509                 cidp->statistic_type = BGE_STAT_REG;
2510                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2511                 dev_ok = B_TRUE;
2512                 break;

2514         case DEVICE_ID_5752:
2515         case DEVICE_ID_5752M:
2516                 cidp->chip_label = 5752;
2517                 cidp->mbuf_lo_water_rdma = RDMA_MBUF_LOWAT_5705;
2518                 cidp->mbuf_lo_water_rmac = MAC_RX_MBUF_LOWAT_5705;
2519                 cidp->mbuf_hi_water = MBUF_HIWAT_5705;
2520                 cidp->mbuf_base = bge_mbuf_pool_base_5721;
2521                 cidp->mbuf_length = bge_mbuf_pool_len_5721;
2522                 cidp->recv_slots = BGE_RECV_SLOTS_5721;
2523                 cidp->bge_dma_rwctrl = bge_dma_rwctrl_5721;
2524                 cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2525                 cidp->tx_rings = BGE_SEND_RINGS_MAX_5705;
2526                 cidp->pci_type = BGE_PCI_E;
2527                 cidp->statistic_type = BGE_STAT_REG;
2528                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2529                 dev_ok = B_TRUE;
2530                 break;

2532         case DEVICE_ID_5789:
2533                 cidp->chip_label = 5789;
2534                 cidp->mbuf_base = bge_mbuf_pool_base_5721;
2535                 cidp->mbuf_length = bge_mbuf_pool_len_5721;
2536                 cidp->recv_slots = BGE_RECV_SLOTS_5721;
2537                 cidp->bge_dma_rwctrl = bge_dma_rwctrl_5721;
```



```
2538                 cidp->rx_rings = BGE_RECV_RINGS_MAX_5705;
2539                 cidp->tx_rings = BGE_RECV_RINGS_MAX_5705;
2540                 cidp->pci_type = BGE_PCI_E;
2541                 cidp->statistic_type = BGE_STAT_REG;
2542                 cidp->flags |= CHIP_FLAG_PARTIAL_CSUM;
2543                 cidp->flags |= CHIP_FLAG_NO_JUMBO;
2544                 cidp->msi_enabled = B_TRUE;
2545                 dev_ok = B_TRUE;
2546                 break;

2548         }

2550         /*
2551          * Setup the default jumbo parameter.
2552          */
2553         cidp->ethmax_size = ETHERMAX;
2554         cidp->snd_buff_size = BGE_SEND_BUFF_SIZE_DEFAULT;
2555         cidp->std_buf_size = BGE_STD_BUFF_SIZE;

2557         /*
2558          * If jumbo is enabled and this kind of chipset supports jumbo feature,
2559          * setup below jumbo specific parameters.
2560          *
2561          * For BCM5714/5715, there is only one standard receive ring. So the
2562          * std buffer size should be set to BGE_JUMBO_BUFF_SIZE when jumbo
2563          * feature is enabled.
2564          */
2565         if (!(cidp->flags & CHIP_FLAG_NO_JUMBO) &&
2566             (cidp->default_mtu > BGE_DEFAULT_MTU)) {
2567                 if (DEVICE_5714_SERIES_CHIPSETS(bgep)) {
2568                         cidp->mbuf_lo_water_rdma =
2569                             RDMA_MBUF_LOWAT_5714_JUMBO;
2570                         cidp->mbuf_lo_water_rmac =
2571                             MAC_RX_MBUF_LOWAT_5714_JUMBO;
2572                         cidp->mbuf_hi_water = MBUF_HIWAT_5714_JUMBO;
2573                         cidp->jumbo_slots = 0;
2574                         cidp->std_buf_size = BGE_JUMBO_BUFF_SIZE;
2575                 } else {
2576                         cidp->mbuf_lo_water_rdma =
2577                             RDMA_MBUF_LOWAT_JUMBO;
2578                         cidp->mbuf_lo_water_rmac =
2579                             MAC_RX_MBUF_LOWAT_JUMBO;
2580                         cidp->mbuf_hi_water = MBUF_HIWAT_JUMBO;
2581                         cidp->jumbo_slots = BGE_JUMBO_SLOTS_USED;
2582                 }
2583                 cidp->recv_jumbo_size = BGE_JUMBO_BUFF_SIZE;
2584                 cidp->snd_buff_size = BGE_SEND_BUFF_SIZE_JUMBO;
2585                 cidp->ethmax_size = cidp->default_mtu +
2586                     sizeof (struct ether_header);
2587         }

2589         /*
2590          * Identify the NV memory type: SEEPROM or Flash?
2591          */
2592         cidp->nvtype = bge_nvmem_id(bgep);

2594         /*
2595          * Now, we want to check whether this device is part of a
2596          * supported subsystem (e.g., on the motherboard of a Sun
2597          * branded platform).
2598          *
2599          * Rule 1: If the Subsystem Vendor ID is "Sun", then it’s OK ;-)
2600          */
2601         if (cidp->subven == VENDOR_ID_SUN)
2602                 sys_ok = B_TRUE;
```

```
2604         /*
2605          * Rule 2: If it's on the list on known subsystems, then it's OK.
2606          * Note: 0x14e41647 should *not* appear in the list, but the code
2607          * doesn't enforce that.
2608          */
2609         err = ddi_prop_lookup_int_array(DDI_DEV_T_ANY, bgep->devinfo,
2610             DDI_PROP_DONTPASS, knownids_propname, &ids, &i);
2611         if (err == DDI_PROP_SUCCESS) {
2612                 /*
2613                  * Got the list; scan for a matching subsystem vendor/device
2614                  */
2615                 subid = (cidp->subven << 16) | cidp->subdev;
2616                 while (i--)
2617                         if (ids[i] == subid)
2618                                 sys_ok = B_TRUE;
2619                 ddi_prop_free(ids);
2620         }

2622         /*
2623          * Rule 3: If it's a Taco/ENWS motherboard device, then it's OK
2624          *
2625          * Unfortunately, early SunBlade 1500s and 2500s didn't reprogram
2626          * the Subsystem Vendor ID, so it defaults to Broadcom.  Therefore,
2627          * we have to check specially for the exact device paths to the
2628          * motherboard devices on those platforms ;-(
2629          *
2630          * Note: we can't just use the "supported-subsystems" mechanism
2631          * above, because the entry would have to be 0x14e41647 -- which
2632          * would then accept *any* plugin card that *didn't* contain a
2633          * (valid) SEEPROM ;-(
2634          */
2635         sysname = ddi_node_name(ddi_root_node());
2636         devname = ddi_pathname(bgep->devinfo, buf);
2637         ASSERT(strlen(devname) > 0);
2638         if (strcmp(sysname, "SUNW,Sun-Blade-1500") == 0)        /* Taco */
2639                 if (strcmp(devname, "/pci@1f,700000/network@2") == 0)
2640                         sys_ok = B_TRUE;
2641         if (strcmp(sysname, "SUNW,Sun-Blade-2500") == 0)        /* ENWS */
2642                 if (strcmp(devname, "/pci@1c,600000/network@3") == 0)
2643                         sys_ok = B_TRUE;

2645         /*
2646          * Now check what we've discovered: is this truly a supported
2647          * chip on (the motherboard of) a supported platform?
2648          *
2649          * Possible problems here:
2650          * 1)   it's a completely unheard-of chip
2651          * 2)   it's a recognised but unsupported chip (e.g. 5701, 5703C-A0)
2652          * 3)   it's a chip we would support if it were on the motherboard
2653          *      of a Sun platform, but this one isn't ;-(
2654          */
2655         if (cidp->chip_label == 0)
2656                 bge_problem(bgep,
2657                     "Device 'pci%04x,%04x' not recognized (%d?)",
2658                     cidp->vendor, cidp->device, cidp->device);
2659         else if (!dev_ok)
2660                 bge_problem(bgep,
2661                     "Device 'pci%04x,%04x' (%d) revision %d not supported",
2662                     cidp->vendor, cidp->device, cidp->chip_label,
2663                     cidp->revision);
2664 #if     BGE_DEBUGGING
2665         else if (!sys_ok)
2666                 bge_problem(bgep,
2667                     "%d-based subsystem 'pci%04x,%04x' not validated",
2668                     cidp->chip_label, cidp->subven, cidp->subdev);
2669 #endif
```



```
2670         else
2671                 cidp->flags |= CHIP_FLAG_SUPPORTED;
2672         if (bge_check_acc_handle(bgep, bgep->io_handle) != DDI_FM_OK)
2673                 return (EIO);
2674         return (0);
2675 }
______unchanged_portion_omitted_

3448 /*
3449  * Maximum times of trying to get the NVRAM access lock
3450  * by calling bge_nvmem_acquire()
3451  */
3452 #define MAX_TRY_NVMEM_ACQUIRE   10000

3454 #ifdef BGE_IPMI_ASF
3455 int bge_chip_reset(bge_t *bgep, boolean_t enable_dma, uint_t asf_mode);
3456 #else
3457 int bge_chip_reset(bge_t *bgep, boolean_t enable_dma);
3458 #endif
3459 #pragma no_inline(bge_chip_reset)

3461 int
3462 #ifdef BGE_IPMI_ASF
3463 bge_chip_reset(bge_t *bgep, boolean_t enable_dma, uint_t asf_mode)
3464 #else
3465 bge_chip_reset(bge_t *bgep, boolean_t enable_dma)
3466 #endif
3467 {
3468         chip_id_t chipid;
3469         uint64_t mac;
3470         uint64_t magic;
3471         uint32_t modeflags;
3472         uint32_t mhcr;
3473         uint32_t sx0;
3474         uint32_t i, tries;
3475 #ifdef BGE_IPMI_ASF
3476         uint32_t mailbox;
3477 #endif
3478         int retval = DDI_SUCCESS;

3480         BGE_TRACE(("bge_chip_reset($%p, %d)",
3481                 (void *)bgep, enable_dma));

3483         ASSERT(mutex_owned(bgep->genlock));

3485         BGE_DEBUG(("bge_chip_reset($%p, %d): current state is %d",
3486                 (void *)bgep, enable_dma, bgep->bge_chip_state));

3488         /*
3489          * Do we need to stop the chip cleanly before resetting?
3490          */
3491         switch (bgep->bge_chip_state) {
3492         default:
3493                 _NOTE(NOTREACHED)
3494                 return (DDI_FAILURE);

3496         case BGE_CHIP_INITIAL:
3497         case BGE_CHIP_STOPPED:
3498         case BGE_CHIP_RESET:
3499                 break;

3501         case BGE_CHIP_RUNNING:
3502         case BGE_CHIP_ERROR:
3503         case BGE_CHIP_FAULT:
3504                 bge_chip_stop(bgep, B_FALSE);
3505                 break;
```

```
3506         }

3508 #ifdef BGE_IPMI_ASF
3509         if (bgep->asf_enabled) {
3510 #ifdef __sparc
3511                 mhcr = MHCR_ENABLE_INDIRECT_ACCESS |
3512                        MHCR_ENABLE_TAGGED_STATUS_MODE |
3513                        MHCR_MASK_INTERRUPT_MODE |
3514                        MHCR_CLEAR_INTERRUPT_INTA |
3515                        MHCR_ENABLE_ENDIAN_WORD_SWAP |
3516                        MHCR_ENABLE_ENDIAN_BYTE_SWAP;

3518                 if (bgep->intr_type == DDI_INTR_TYPE_FIXED)
3519                        mhcr |= MHCR_MASK_PCI_INT_OUTPUT;

3521                 if (DEVICE_5717_SERIES_CHIPSETS(bgep))
3522                        pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR,
3523                                         0);
3524 #else
3525                 mhcr = MHCR_ENABLE_INDIRECT_ACCESS |
3526                        MHCR_ENABLE_TAGGED_STATUS_MODE |
3527                        MHCR_MASK_INTERRUPT_MODE |
3528                        MHCR_MASK_PCI_INT_OUTPUT |
3529                        MHCR_CLEAR_INTERRUPT_INTA;
3530 #endif
3531                 pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR, mhcr);
3532                 bge_reg_put32(bgep, MEMORY_ARBITER_MODE_REG,
3533                        bge_reg_get32(bgep, MEMORY_ARBITER_MODE_REG) |
3534                        MEMORY_ARBITER_ENABLE);
3438 #endif
3535                 if (asf_mode == ASF_MODE_INIT) {
3536                        bge_asf_pre_reset_operations(bgep, BGE_INIT_RESET);
3537                 } else if (asf_mode == ASF_MODE_SHUTDOWN) {
3538                        bge_asf_pre_reset_operations(bgep, BGE_SHUTDOWN_RESET);
3539                 }
3540         }
3541 #endif
3542         /*
3543          * Adapted from Broadcom document 570X-PG102-R, pp 102-116.
3544          * Updated to reflect Broadcom document 570X-PG104-R, pp 146-159.
3545          *
3546          * Before reset Core clock,it is
3547          * also required to initialize the Memory Arbiter as specified in step9
3548          * and Misc Host Control Register as specified in step-13
3549          * Step 4-5: reset Core clock & wait for completion
3550          * Steps 6-8: are done by bge_chip_cfg_init()
3551          * put the T3_MAGIC_NUMBER into the GENCOMM port before reset
3552          */
3553         if (!bge_chip_enable_engine(bgep, MEMORY_ARBITER_MODE_REG, 0))
3554                 retval = DDI_FAILURE;

3556         mhcr = MHCR_ENABLE_INDIRECT_ACCESS |
3557             MHCR_ENABLE_TAGGED_STATUS_MODE |
3558             MHCR_ENABLE_PCI_STATE_WRITE |
3559             MHCR_MASK_INTERRUPT_MODE |
3560             MHCR_CLEAR_INTERRUPT_INTA;

3562         if (bgep->intr_type == DDI_INTR_TYPE_FIXED)
3563                 mhcr |= MHCR_MASK_PCI_INT_OUTPUT;

3565 #ifdef  _BIG_ENDIAN
3566         mhcr |= MHCR_ENABLE_ENDIAN_WORD_SWAP | MHCR_ENABLE_ENDIAN_BYTE_SWAP;
3567 #endif  /* _BIG_ENDIAN */
3568         if (DEVICE_5717_SERIES_CHIPSETS(bgep))
3569                 pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR, 0);
3570         pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR, mhcr);
```



```
3571 #ifdef BGE_IPMI_ASF
3572         if (bgep->asf_enabled)
3573                 bgep->asf_wordswapped = B_FALSE;
3574 #endif

3576         if (DEVICE_IS_5755_PLUS(bgep) ||
3577             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
3578             MHCR_CHIP_ASIC_REV_5752)
3579                 bge_reg_put32(bgep, GRC_FASTBOOT_PC, 0);

3581         /*
3582          * NVRAM Corruption Workaround
3583          */
3584         for (tries = 0; tries < MAX_TRY_NVMEM_ACQUIRE; tries++)
3585                 if (bge_nvmem_acquire(bgep) != EAGAIN)
3586                         break;
3587         if (tries >= MAX_TRY_NVMEM_ACQUIRE)
3588                 BGE_DEBUG(("%s: fail to acquire nvram lock",
3589                         bgep->ifname));

3591 #ifdef BGE_IPMI_ASF
3592         if (!bgep->asf_enabled) {
3593 #endif
3594                 magic = (uint64_t)T3_MAGIC_NUMBER << 32;
3595                 bge_nic_put64(bgep, NIC_MEM_GENCOMM, magic);
3596 #ifdef BGE_IPMI_ASF
3597         }
3598 #endif

3600         if (!bge_chip_reset_engine(bgep, MISC_CONFIG_REG))
3601                 retval = DDI_FAILURE;
3602         bge_chip_cfg_init(bgep, &chipid, enable_dma);

3604         /*
3605          * Step 8a: This may belong elsewhere, but BCM5721 needs
3606          * a bit set to avoid a fifo overflow/underflow bug.
3607          */
3608         if ((bgep->chipid.chip_label == 5721) ||
3609                 (bgep->chipid.chip_label == 5751) ||
3610                 (bgep->chipid.chip_label == 5752) ||
3611                 (bgep->chipid.chip_label == 5755) ||
3612                 (bgep->chipid.chip_label == 5756) ||
3613                 (bgep->chipid.chip_label == 5789) ||
3614                 (bgep->chipid.chip_label == 5906))
3615                 bge_reg_set32(bgep, TLP_CONTROL_REG, TLP_DATA_FIFO_PROTECT);


3618         /*
3619          * Step 9: enable MAC memory arbiter,bit30 and bit31 of 5714/5715 should
3620          * not be changed.
3621          */
3622         if (!bge_chip_enable_engine(bgep, MEMORY_ARBITER_MODE_REG, 0))
3623                 retval = DDI_FAILURE;

3625         /*
3626          * Steps 10-11: configure PIO endianness options and
3627          * enable indirect register access -- already done
3628          * Steps 12-13: enable writing to the PCI state & clock
3629          * control registers -- not required; we aren't going to
3630          * use those features.
3631          * Steps 14-15: Configure DMA endianness options.  See
3632          * the comments on the setting of the MHCR above.
3633          */
3634 #ifdef  _BIG_ENDIAN
3635         modeflags = MODE_WORD_SWAP_FRAME | MODE_BYTE_SWAP_FRAME |
3636                     MODE_WORD_SWAP_NONFRAME | MODE_BYTE_SWAP_NONFRAME;
```

```
3637 #else
3638         modeflags = MODE_WORD_SWAP_FRAME | MODE_BYTE_SWAP_FRAME;
3639 #endif  /* _BIG_ENDIAN */
3640         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
3641             MHCR_CHIP_ASIC_REV_5720)
3642                 modeflags |=
3643                     MODE_BYTE_SWAP_B2HRX_DATA | MODE_WORD_SWAP_B2HRX_DATA |
3644                     MODE_B2HRX_ENABLE | MODE_HTX2B_ENABLE;
3645 #ifdef BGE_IPMI_ASF
3646         if (bgep->asf_enabled)
3647                 modeflags |= MODE_HOST_STACK_UP;
3648 #endif
3649         bge_reg_put32(bgep, MODE_CONTROL_REG, modeflags);

3651 #ifdef BGE_IPMI_ASF
3652         if (bgep->asf_enabled) {
3653 #ifdef __sparc
3654                 bge_reg_put32(bgep, MEMORY_ARBITER_MODE_REG,
3655                         MEMORY_ARBITER_ENABLE |
3656                         bge_reg_get32(bgep, MEMORY_ARBITER_MODE_REG));
3657 #endif

3659 #ifdef  BGE_NETCONSOLE
3660                 if (!bgep->asf_newhandshake) {
3661                         if ((asf_mode == ASF_MODE_INIT) ||
3662                         (asf_mode == ASF_MODE_POST_INIT)) {
3663                                 bge_asf_post_reset_old_mode(bgep,
3664                                         BGE_INIT_RESET);
3665                         } else {
3666                                 bge_asf_post_reset_old_mode(bgep,
3667                                         BGE_SHUTDOWN_RESET);
3668                         }
3669                 }
3670 #endif

3672                 /* Wait for NVRAM init */
3673                 i = 0;
3674                 drv_usecwait(5000);
3675                 mailbox = bge_nic_get32(bgep, BGE_FIRMWARE_MAILBOX);

3677                 while ((mailbox != (uint32_t)
3678                         ~BGE_MAGIC_NUM_FIRMWARE_INIT_DONE) &&
3679                         (i < 10000)) {
3680                         drv_usecwait(100);
3681                         mailbox = bge_nic_get32(bgep,
3682                                 BGE_FIRMWARE_MAILBOX);
3683                         i++;
3684                 }

3686 #ifndef BGE_NETCONSOLE
3687                 if (!bgep->asf_newhandshake) {
3688                         if ((asf_mode == ASF_MODE_INIT) ||
3689                                 (asf_mode == ASF_MODE_POST_INIT)) {

3691                                 bge_asf_post_reset_old_mode(bgep,
3692                                         BGE_INIT_RESET);
3693                         } else {
3694                                 bge_asf_post_reset_old_mode(bgep,
3695                                         BGE_SHUTDOWN_RESET);
3696                         }
3697                 }
3698 #endif
3699         }
3700 #endif
3701         /*
3702          * Steps 16-17: poll for firmware completion
```



```
3703          */
3704         mac = bge_poll_firmware(bgep);

3706         /*
3707          * Step 18: enable external memory -- doesn't apply.
3708          *
3709          * However we take the opportunity to set the MLCR anyway, as
3710          * this register also controls the SEEPROM auto-access method
3711          * which we may want to use later ...
3712          *
3713          * The proper value here depends on the way the chip is wired
3714          * into the circuit board, as this register *also* controls which
3715          * of the "Miscellaneous I/O" pins are driven as outputs and the
3716          * values driven onto those pins!
3717          *
3718          * See also step 74 in the PRM ...
3719          */
3720         bge_reg_put32(bgep, MISC_LOCAL_CONTROL_REG,
3721             bgep->chipid.bge_mlcr_default);
3722         bge_reg_set32(bgep, SERIAL_EEPROM_ADDRESS_REG, SEEPROM_ACCESS_INIT);

3724         /*
3725          * Step 20: clear the Ethernet MAC mode register
3726          */
3727         bge_reg_put32(bgep, ETHERNET_MAC_MODE_REG, 0);

3729         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
3730             MHCR_CHIP_ASIC_REV_5720) {
3731                 uint32_t regval = bge_reg_get32(bgep, CPMU_CLCK_ORIDE_REG);
3732                 bge_reg_put32(bgep, CPMU_CLCK_ORIDE_REG,
3733                     regval & ~CPMU_CLCK_ORIDE_MAC_ORIDE_EN);
3734         }

3736         /*
3737          * Step 21: restore cache-line-size, latency timer, and
3738          * subsystem ID registers to their original values (not
3739          * those read into the local structure <chipid>, 'cos
3740          * that was after they were cleared by the RESET).
3741          *
3742          * Note: the Subsystem Vendor/Device ID registers are not
3743          * directly writable in config space, so we use the shadow
3744          * copy in "Page Zero" of register space to restore them
3745          * both in one go ...
3746          */
3747         pci_config_put8(bgep->cfg_handle, PCI_CONF_CACHE_LINESZ,
3748                 bgep->chipid.clsize);
3749         pci_config_put8(bgep->cfg_handle, PCI_CONF_LATENCY_TIMER,
3750                 bgep->chipid.latency);
3751         bge_reg_put32(bgep, PCI_CONF_SUBVENID,
3752                 (bgep->chipid.subdev << 16) | bgep->chipid.subven);

3754         /*
3755          * The SEND INDEX registers should be reset to zero by the
3756          * global chip reset; if they're not, there'll be trouble
3757          * later on.
3758          */
3759         sx0 = bge_reg_get32(bgep, NIC_DIAG_SEND_INDEX_REG(0));
3760         if (sx0 != 0) {
3761                 BGE_REPORT((bgep, "SEND INDEX - device didn't RESET"));
3762                 bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
3763                 retval = DDI_FAILURE;
3764         }

3766         /* Enable MSI code */
3767         if (bgep->intr_type == DDI_INTR_TYPE_MSI)
3768                 bge_reg_set32(bgep, MSI_MODE_REG,
```

```
3769                    MSI_PRI_HIGHEST|MSI_MSI_ENABLE|MSI_ERROR_ATTENTION);

3771        /*
3772         * On the first time through, save the factory-set MAC address
3773         * (if any).  If bge_poll_firmware() above didn't return one
3774         * (from a chip register) consider looking in the attached NV
3775         * memory device, if any.  Once we have it, we save it in both
3776         * register-image (64-bit) and byte-array forms.  All-zero and
3777         * all-one addresses are not valid, and we refuse to stash those.
3778         */
3779        if (bgep->bge_chip_state == BGE_CHIP_INITIAL) {
3780                if (mac == 0ULL)
3781                        mac = bge_get_nvmac(bgep);
3782                if (mac != 0ULL && mac != ~0ULL) {
3783                        bgep->chipid.hw_mac_addr = mac;
3784                        for (i = ETHERADDRL; i-- != 0; ) {
3785                                bgep->chipid.vendor_addr.addr[i] = (uchar_t)mac;
3786                                mac >>= 8;
3787                        }
3788                        bgep->chipid.vendor_addr.set = B_TRUE;
3789                }
3790        }

3792 #ifdef BGE_IPMI_ASF
3793        if (bgep->asf_enabled && bgep->asf_newhandshake) {
3794                if (asf_mode != ASF_MODE_NONE) {
3795                        if ((asf_mode == ASF_MODE_INIT) ||
3796                                (asf_mode == ASF_MODE_POST_INIT)) {

3798                                bge_asf_post_reset_new_mode(bgep,
3799                                        BGE_INIT_RESET);
3800                        } else {
3801                                bge_asf_post_reset_new_mode(bgep,
3802                                        BGE_SHUTDOWN_RESET);
3803                        }
3804                }
3805        }
3806 #endif

3808        /*
3809         * Record the new state
3810         */
3811        bgep->chip_resets += 1;
3812        bgep->bge_chip_state = BGE_CHIP_RESET;
3813        return (retval);
3814 }
```

*unchanged_portion_omitted_*

```
3836 int
3837 bge_chip_start(bge_t *bgep, boolean_t reset_phys)
3838 {
3839        uint32_t coalmode;
3840        uint32_t ledctl;
3841        uint32_t mtu;
3842        uint32_t maxring;
3843        uint32_t stats_mask;
3844        uint32_t dma_wrprio;
3845        uint64_t ring;
3846        uint32_t regval;
3847        int retval = DDI_SUCCESS;

3849        BGE_TRACE(("bge_chip_start($%p)",
3850            (void *)bgep));

3852        ASSERT(mutex_owned(bgep->genlock));
3853        ASSERT(bgep->bge_chip_state == BGE_CHIP_RESET);
```



```
3855        /*
3856         * Taken from Broadcom document 570X-PG102-R, pp 102-116.
3857         * The document specifies 95 separate steps to fully
3858         * initialise the chip!!!!
3859         *
3860         * The reset code above has already got us as far as step
3861         * 21, so we continue with ...
3862         *
3863         * Step 22: clear the MAC statistics block
3864         * (0x0300-0x0aff in NIC-local memory)
3865         */
3866        if (bgep->chipid.statistic_type == BGE_STAT_BLK)
3867                bge_nic_zero(bgep, NIC_MEM_STATISTICS,
3868                    NIC_MEM_STATISTICS_SIZE);

3870        /*
3871         * Step 23: clear the status block (in host memory)
3872         */
3873        DMA_ZERO(bgep->status_block);

3875        /*
3876         * Step 24: set DMA read/write control register
3877         */
3878        pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_PDRWCR,
3879            bgep->chipid.bge_dma_rwctrl);

3881        /*
3882         * Step 25: Configure DMA endianness -- already done (16/17)
3883         * Step 26: Configure Host-Based Send Rings
3884         * Step 27: Indicate Host Stack Up
3885         */
3886        bge_reg_set32(bgep, MODE_CONTROL_REG,
3887            MODE_HOST_SEND_BDS |
3888            MODE_HOST_STACK_UP);

3890        /*
3891         * Step 28: Configure checksum options:
3892         *      Solaris supports the hardware default checksum options.
3893         *
3894         *      Workaround for Incorrect pseudo-header checksum calculation.
3895         */
3896        if (bgep->chipid.flags & CHIP_FLAG_PARTIAL_CSUM)
3897                bge_reg_set32(bgep, MODE_CONTROL_REG,
3898                    MODE_SEND_NO_PSEUDO_HDR_CSUM);

3900        /*
3901         * Step 29: configure Timer Prescaler.  The value is always the
3902         * same: the Core Clock frequency in MHz (66), minus 1, shifted
3903         * into bits 7-1.  Don't set bit 0, 'cos that's the RESET bit
3904         * for the whole chip!
3905         */
3906        regval = bge_reg_get32(bgep, MISC_CONFIG_REG);
3907        regval = (regval & 0xffffff00) | MISC_CONFIG_DEFAULT;
3908        bge_reg_put32(bgep, MISC_CONFIG_REG, regval);

3910        if (DEVICE_5906_SERIES_CHIPSETS(bgep)) {
3911                drv_usecwait(40);
3912                /* put PHY into ready state */
3913                bge_reg_clr32(bgep, MISC_CONFIG_REG, MISC_CONFIG_EPHY_IDDQ);
3914                (void) bge_reg_get32(bgep, MISC_CONFIG_REG); /* flush */
3915                drv_usecwait(40);
3916        }

3918        /*
3919         * Steps 30-31: Configure MAC local memory pool & DMA pool registers
```

```
3920          *
3921          * If the mbuf_length is specified as 0, we just leave these at
3922          * their hardware defaults, rather than explicitly setting them.
3923          * As the Broadcom HRM,driver better not change the parameters
3924          * when the chipsets is 5705/5788/5721/5751/5714 and 5715.
3925          */
3926         if ((bgep->chipid.mbuf_length != 0) &&
3927             (DEVICE_5704_SERIES_CHIPSETS(bgep))) {
3928                         bge_reg_put32(bgep, MBUF_POOL_BASE_REG,
3929                             bgep->chipid.mbuf_base);
3930                         bge_reg_put32(bgep, MBUF_POOL_LENGTH_REG,
3931                             bgep->chipid.mbuf_length);
3932                         bge_reg_put32(bgep, DMAD_POOL_BASE_REG,
3933                             DMAD_POOL_BASE_DEFAULT);
3934                         bge_reg_put32(bgep, DMAD_POOL_LENGTH_REG,
3935                             DMAD_POOL_LENGTH_DEFAULT);
3936         }

3938         /*
3939          * Step 32: configure MAC memory pool watermarks
3940          */
3941         bge_reg_put32(bgep, RDMA_MBUF_LOWAT_REG,
3942             bgep->chipid.mbuf_lo_water_rdma);
3943         bge_reg_put32(bgep, MAC_RX_MBUF_LOWAT_REG,
3944             bgep->chipid.mbuf_lo_water_rmac);
3945         bge_reg_put32(bgep, MBUF_HIWAT_REG,
3946             bgep->chipid.mbuf_hi_water);

3948         /*
3949          * Step 33: configure DMA resource watermarks
3950          */
3951         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
3952                 bge_reg_put32(bgep, DMAD_POOL_LOWAT_REG,
3953                     bge_dmad_lo_water);
3954                 bge_reg_put32(bgep, DMAD_POOL_HIWAT_REG,
3955                     bge_dmad_hi_water);
3956         }
3957         bge_reg_put32(bgep, LOWAT_MAX_RECV_FRAMES_REG, bge_lowat_recv_frames);

3959         /*
3960          * Steps 34-36: enable buffer manager & internal h/w queues
3961          */

3963         regval = STATE_MACHINE_ATTN_ENABLE_BIT;
3964         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
3965             MHCR_CHIP_ASIC_REV_5719)
3966                 regval |= BUFF_MGR_NO_TX_UNDERRUN;
3967         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
3968             MHCR_CHIP_ASIC_REV_5717 ||
3969             bgep->chipid.asic_rev == MHCR_CHIP_REV_5719_A0 ||
3970             bgep->chipid.asic_rev == MHCR_CHIP_REV_5720_A0)
3971                 regval |= BUFF_MGR_MBUF_LOW_ATTN_ENABLE;
3972         if (!bge_chip_enable_engine(bgep, BUFFER_MANAGER_MODE_REG, regval))
3847         if (!bge_chip_enable_engine(bgep, BUFFER_MANAGER_MODE_REG,
3848             STATE_MACHINE_ATTN_ENABLE_BIT))
3973                 retval = DDI_FAILURE;
3974         if (!bge_chip_enable_engine(bgep, FTQ_RESET_REG, 0))
3975                 retval = DDI_FAILURE;

3977         /*
3978          * Steps 37-39: initialise Receive Buffer (Producer) RCBs
3979          */
3980         if (DEVICE_5717_SERIES_CHIPSETS(bgep)) {
3981                 buff_ring_t *brp = &bgep->buff[BGE_STD_BUFF_RING];
3982                 bge_reg_put64(bgep, STD_RCV_BD_RING_RCB_REG,
3983                     brp->desc.cookie.dmac_laddress);
```



```
3984                 bge_reg_put32(bgep, STD_RCV_BD_RING_RCB_REG + 8,
3985                     (brp->desc.nslots) << 16 | brp->buf[0].size << 2);
3986                 bge_reg_put32(bgep, STD_RCV_BD_RING_RCB_REG + 0xc,
3987                     NIC_MEM_SHADOW_BUFF_STD_5717);
3988         } else
3989                 bge_reg_putrcb(bgep, STD_RCV_BD_RING_RCB_REG,
3990                     &bgep->buff[BGE_STD_BUFF_RING].hw_rcb);

3992         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
3993                 bge_reg_putrcb(bgep, JUMBO_RCV_BD_RING_RCB_REG,
3994                     &bgep->buff[BGE_JUMBO_BUFF_RING].hw_rcb);
3995                 bge_reg_putrcb(bgep, MINI_RCV_BD_RING_RCB_REG,
3996                     &bgep->buff[BGE_MINI_BUFF_RING].hw_rcb);
3997         }

3999         /*
4000          * Step 40: set Receive Buffer Descriptor Ring replenish thresholds
4001          */
4002         bge_reg_put32(bgep, STD_RCV_BD_REPLENISH_REG, bge_replenish_std);
4003         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
4004                 bge_reg_put32(bgep, JUMBO_RCV_BD_REPLENISH_REG,
4005                     bge_replenish_jumbo);
4006                 bge_reg_put32(bgep, MINI_RCV_BD_REPLENISH_REG,
4007                     bge_replenish_mini);
4008         }

4010         /*
4011          * Steps 41-43: clear Send Ring Producer Indices and initialise
4012          * Send Producer Rings (0x0100-0x01ff in NIC-local memory)
4013          */
4014         if (DEVICE_5704_SERIES_CHIPSETS(bgep))
4015                 maxring = BGE_SEND_RINGS_MAX;
4016         else
4017                 maxring = BGE_SEND_RINGS_MAX_5705;
4018         for (ring = 0; ring < maxring; ++ring) {
4019                 bge_mbx_put(bgep, SEND_RING_HOST_INDEX_REG(ring), 0);
4020                 bge_mbx_put(bgep, SEND_RING_NIC_INDEX_REG(ring), 0);
4021                 bge_nic_putrcb(bgep, NIC_MEM_SEND_RING(ring),
4022                     &bgep->send[ring].hw_rcb);
4023         }

4025         /*
4026          * Steps 44-45: initialise Receive Return Rings
4027          * (0x0200-0x02ff in NIC-local memory)
4028          */
4029         if (DEVICE_5704_SERIES_CHIPSETS(bgep))
4030                 maxring = BGE_RECV_RINGS_MAX;
4031         else
4032                 maxring = BGE_RECV_RINGS_MAX_5705;
4033         for (ring = 0; ring < maxring; ++ring)
4034                 bge_nic_putrcb(bgep, NIC_MEM_RECV_RING(ring),
4035                     &bgep->recv[ring].hw_rcb);

4037         /*
4038          * Step 46: initialise Receive Buffer (Producer) Ring indexes
4039          */
4040         bge_mbx_put(bgep, RECV_STD_PROD_INDEX_REG, 0);
4041         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
4042                 bge_mbx_put(bgep, RECV_JUMBO_PROD_INDEX_REG, 0);
4043                 bge_mbx_put(bgep, RECV_MINI_PROD_INDEX_REG, 0);
4044         }
4045         /*
4046          * Step 47: configure the MAC unicast address
4047          * Step 48: configure the random backoff seed
4048          * Step 96: set up multicast filters
4049          */
```

```
4050 #ifdef BGE_IPMI_ASF
4051         if (bge_chip_sync(bgep, B_FALSE) == DDI_FAILURE)
4052 #else
4053         if (bge_chip_sync(bgep) == DDI_FAILURE)
4054 #endif
4055                 retval = DDI_FAILURE;

4057         /*
4058          * Step 49: configure the MTU
4059          */
4060         mtu = bgep->chipid.ethmax_size+ETHERFCSL+VLAN_TAGSZ;
4061         bge_reg_put32(bgep, MAC_RX_MTU_SIZE_REG, mtu);

4063         /*
4064          * Step 50: configure the IPG et al
4065          */
4066         regval = MAC_TX_LENGTHS_DEFAULT;
4067         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev)
4068             == MHCR_CHIP_ASIC_REV_5720)
4069                 regval |= bge_reg_get32(bgep, MAC_TX_LENGTHS_REG) &
4070                     (MAC_TX_LENGTHS_JMB_FRM_LEN_MSK |
4071                     MAC_TX_LENGTHS_CNT_DWN_VAL_MSK);
4072         bge_reg_put32(bgep, MAC_TX_LENGTHS_REG, regval);
3942         bge_reg_put32(bgep, MAC_TX_LENGTHS_REG, MAC_TX_LENGTHS_DEFAULT);

4074         /*
4075          * Step 51: configure the default Rx Return Ring
4076          */
4077         bge_reg_put32(bgep, RCV_RULES_CONFIG_REG, RCV_RULES_CONFIG_DEFAULT);

4079         /*
4080          * Steps 52-54: configure Receive List Placement,
4081          * and enable Receive List Placement Statistics
4082          */
4083         bge_reg_put32(bgep, RCV_LP_CONFIG_REG,
4084             RCV_LP_CONFIG(bgep->chipid.rx_rings));
4085         switch (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev)) {
4086         case MHCR_CHIP_ASIC_REV_5700:
4087         case MHCR_CHIP_ASIC_REV_5701:
4088         case MHCR_CHIP_ASIC_REV_5703:
4089         case MHCR_CHIP_ASIC_REV_5704:
4090                 bge_reg_put32(bgep, RCV_LP_STATS_ENABLE_MASK_REG, ~0);
4091                 break;
4092         case MHCR_CHIP_ASIC_REV_5705:
4093                 break;
4094         default:
4095                 stats_mask = bge_reg_get32(bgep, RCV_LP_STATS_ENABLE_MASK_REG);
4096                 stats_mask &= ~RCV_LP_STATS_DISABLE_MACTQ;
4097                 bge_reg_put32(bgep, RCV_LP_STATS_ENABLE_MASK_REG, stats_mask);
4098                 break;
4099         }
4100         bge_reg_set32(bgep, RCV_LP_STATS_CONTROL_REG, RCV_LP_STATS_ENABLE);

4102         if (bgep->chipid.rx_rings > 1)
4103                 bge_init_recv_rule(bgep);

4105         /*
4106          * Steps 55-56: enable Send Data Initiator Statistics
4107          */
4108         bge_reg_put32(bgep, SEND_INIT_STATS_ENABLE_MASK_REG, ~0);
4109         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
4110                 bge_reg_put32(bgep, SEND_INIT_STATS_CONTROL_REG,
4111                     SEND_INIT_STATS_ENABLE | SEND_INIT_STATS_FASTER);
4112         } else {
4113                 bge_reg_put32(bgep, SEND_INIT_STATS_CONTROL_REG,
4114                     SEND_INIT_STATS_ENABLE);
```



```
4115         }
4116         /*
4117          * Steps 57-58: stop (?) the Host Coalescing Engine
4118          */
4119         if (!bge_chip_disable_engine(bgep, HOST_COALESCE_MODE_REG, ~0))
4120                 retval = DDI_FAILURE;

4122         /*
4123          * Steps 59-62: initialise Host Coalescing parameters
4124          */
4125         bge_chip_coalesce_update(bgep);
4126         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
4127                 bge_reg_put32(bgep, SEND_COALESCE_INT_BD_REG,
4128                     bge_tx_count_intr);
4129                 bge_reg_put32(bgep, SEND_COALESCE_INT_TICKS_REG,
4130                     bge_tx_ticks_intr);
4131                 bge_reg_put32(bgep, RCV_COALESCE_INT_BD_REG,
4132                     bge_rx_count_intr);
4133                 bge_reg_put32(bgep, RCV_COALESCE_INT_TICKS_REG,
4134                     bge_rx_ticks_intr);
4135         }

4137         /*
4138          * Steps 63-64: initialise status block & statistics
4139          * host memory addresses
4140          * The statistic block does not exist in some chipsets
4141          * Step 65: initialise Statistics Coalescing Tick Counter
4142          */
4143         bge_reg_put64(bgep, STATUS_BLOCK_HOST_ADDR_REG,
4144             bgep->status_block.cookie.dmac_laddress);

4146         /*
4147          * Steps 66-67: initialise status block & statistics
4148          * NIC-local memory addresses
4149          */
4150         if (DEVICE_5704_SERIES_CHIPSETS(bgep)) {
4151                 bge_reg_put64(bgep, STATISTICS_HOST_ADDR_REG,
4152                     bgep->statistics.cookie.dmac_laddress);
4153                 bge_reg_put32(bgep, STATISTICS_TICKS_REG,
4154                     STATISTICS_TICKS_DEFAULT);
4155                 bge_reg_put32(bgep, STATUS_BLOCK_BASE_ADDR_REG,
4156                     NIC_MEM_STATUS_BLOCK);
4157                 bge_reg_put32(bgep, STATISTICS_BASE_ADDR_REG,
4158                     NIC_MEM_STATISTICS);
4159         }

4161         /*
4162          * Steps 68-71: start the Host Coalescing Engine, the Receive BD
4163          * Completion Engine, the Receive List Placement Engine, and the
4164          * Receive List selector.Pay attention:0x3400 is not exist in BCM5714
4165          * and BCM5715.
4166          */
4167         if (bgep->chipid.tx_rings <= COALESCE_64_BYTE_RINGS &&
4168             bgep->chipid.rx_rings <= COALESCE_64_BYTE_RINGS)
4169                 coalmode = COALESCE_64_BYTE_STATUS;
4170         else
4171                 coalmode = 0;
4172         if (DEVICE_5717_SERIES_CHIPSETS(bgep))
4173                 coalmode = COALESCE_CLR_TICKS_RX;
4174         if (!bge_chip_enable_engine(bgep, HOST_COALESCE_MODE_REG, coalmode))
4175                 retval = DDI_FAILURE;
4176         if (!bge_chip_enable_engine(bgep, RCV_BD_COMPLETION_MODE_REG,
4177             STATE_MACHINE_ATTN_ENABLE_BIT))
4178                 retval = DDI_FAILURE;
4179         if (!bge_chip_enable_engine(bgep, RCV_LIST_PLACEMENT_MODE_REG, 0))
4180                 retval = DDI_FAILURE;
```

```
4182         if (DEVICE_5704_SERIES_CHIPSETS(bgep))
4183                 if (!bge_chip_enable_engine(bgep, RCV_LIST_SELECTOR_MODE_REG,
4184                     STATE_MACHINE_ATTN_ENABLE_BIT))
4185                         retval = DDI_FAILURE;

4187         /*
4188          * Step 72: Enable MAC DMA engines
4189          * Step 73: Clear & enable MAC statistics
4190          */
4191         bge_reg_set32(bgep, ETHERNET_MAC_MODE_REG,
4192             ETHERNET_MODE_ENABLE_FHDE |
4193             ETHERNET_MODE_ENABLE_RDE |
4194             ETHERNET_MODE_ENABLE_TDE);
4195         bge_reg_set32(bgep, ETHERNET_MAC_MODE_REG,
4196             ETHERNET_MODE_ENABLE_TX_STATS |
4197             ETHERNET_MODE_ENABLE_RX_STATS |
4198             ETHERNET_MODE_CLEAR_TX_STATS |
4199             ETHERNET_MODE_CLEAR_RX_STATS);

4201         /*
4202          * Step 74: configure the MLCR (Miscellaneous Local Control
4203          * Register); not required, as we set up the MLCR in step 10
4204          * (part of the reset code) above.
4205          *
4206          * Step 75: clear Interrupt Mailbox 0
4207          */
4208         bge_mbx_put(bgep, INTERRUPT_MBOX_0_REG, 0);

4210         /*
4211          * Steps 76-87: Gentlemen, start your engines ...
4212          *
4213          * Enable the DMA Completion Engine, the Write DMA Engine,
4214          * the Read DMA Engine, Receive Data Completion Engine,
4215          * the MBuf Cluster Free Engine, the Send Data Completion Engine,
4216          * the Send BD Completion Engine, the Receive BD Initiator Engine,
4217          * the Receive Data Initiator Engine, the Send Data Initiator Engine,
4218          * the Send BD Initiator Engine, and the Send BD Selector Engine.
4219          *
4220          * Beware exhaust fumes?
4221          */
4222         if (DEVICE_5704_SERIES_CHIPSETS(bgep))
4223                 if (!bge_chip_enable_engine(bgep, DMA_COMPLETION_MODE_REG, 0))
4224                         retval = DDI_FAILURE;
4225         dma_wrprio = (bge_dma_wrprio << DMA_PRIORITY_SHIFT) |
4226             ALL_DMA_ATTN_BITS;
4227         if (DEVICE_IS_5755_PLUS(bgep))
4097         if ((MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4098             MHCR_CHIP_ASIC_REV_5755) ||
4099             DEVICE_5723_SERIES_CHIPSETS(bgep) ||
4100             DEVICE_5906_SERIES_CHIPSETS(bgep)) {
4228                 dma_wrprio |= DMA_STATUS_TAG_FIX_CQ12384;
4102         }
4229         if (!bge_chip_enable_engine(bgep, WRITE_DMA_MODE_REG,
4230             dma_wrprio))
4231                 retval = DDI_FAILURE;
4232         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4233             MHCR_CHIP_ASIC_REV_5761 ||
4234             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4235             MHCR_CHIP_ASIC_REV_5784 ||
4236             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4237             MHCR_CHIP_ASIC_REV_5785 ||
4238             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4239             MHCR_CHIP_ASIC_REV_57780 ||
4240             DEVICE_IS_57765_PLUS(bgep)) {
4241                 regval = bge_reg_get32(bgep, READ_DMA_RESERVED_CONTROL_REG);
```



```
4242                 if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4243                     MHCR_CHIP_ASIC_REV_5719 ||
4244                     MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4245                     MHCR_CHIP_ASIC_REV_5720) {
4246                         regval &= ~(RDMA_RSRVCTRL_TXMRGN_MASK |
4247                             RDMA_RSRVCTRL_FIFO_LWM_MASK |
4248                             RDMA_RSRVCTRL_FIFO_HWM_MASK);
4249                         regval |= RDMA_RSRVCTRL_TXMRGN_320B |
4250                             RDMA_RSRVCTRL_FIFO_LWM_1_5K |
4251                             RDMA_RSRVCTRL_FIFO_HWM_1_5K;
4252                 }
4253                 bge_reg_put32(bgep, READ_DMA_RESERVED_CONTROL_REG,
4254                     regval | RDMA_RSRVCTRL_FIFO_OFLW_FIX);
4255         }
4256         if (DEVICE_5723_SERIES_CHIPSETS(bgep) ||
4257             DEVICE_5717_SERIES_CHIPSETS(bgep))
4258                 bge_dma_rdprio = 0;
4259         regval = bge_dma_rdprio << DMA_PRIORITY_SHIFT;
4260         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4261             MHCR_CHIP_ASIC_REV_5720)
4262                 regval |= bge_reg_get32(bgep, READ_DMA_MODE_REG) &
4263                     DMA_H2BNC_VLAN_DET;
4264         if (!bge_chip_enable_engine(bgep, READ_DMA_MODE_REG,
4265             regval | ALL_DMA_ATTN_BITS))
4110             (bge_dma_rdprio << DMA_PRIORITY_SHIFT) | ALL_DMA_ATTN_BITS))
4266                 retval = DDI_FAILURE;
4267         if (!bge_chip_enable_engine(bgep, RCV_DATA_COMPLETION_MODE_REG,
4268             STATE_MACHINE_ATTN_ENABLE_BIT))
4269                 retval = DDI_FAILURE;
4270         if (DEVICE_5704_SERIES_CHIPSETS(bgep))
4271                 if (!bge_chip_enable_engine(bgep,
4272                     MBUF_CLUSTER_FREE_MODE_REG, 0))
4273                         retval = DDI_FAILURE;
4274         if (!bge_chip_enable_engine(bgep, SEND_DATA_COMPLETION_MODE_REG, 0))
4275                 retval = DDI_FAILURE;
4276         if (!bge_chip_enable_engine(bgep, SEND_BD_COMPLETION_MODE_REG,
4277             STATE_MACHINE_ATTN_ENABLE_BIT))
4278                 retval = DDI_FAILURE;
4279         if (!bge_chip_enable_engine(bgep, RCV_BD_INITIATOR_MODE_REG,
4280             RCV_BD_DISABLED_RING_ATTN))
4281                 retval = DDI_FAILURE;
4282         if (!bge_chip_enable_engine(bgep, RCV_DATA_BD_INITIATOR_MODE_REG,
4283             RCV_DATA_BD_ILL_RING_ATTN))
4284                 retval = DDI_FAILURE;
4285         if (!bge_chip_enable_engine(bgep, SEND_DATA_INITIATOR_MODE_REG, 0))
4286                 retval = DDI_FAILURE;
4287         if (!bge_chip_enable_engine(bgep, SEND_BD_INITIATOR_MODE_REG,
4288             STATE_MACHINE_ATTN_ENABLE_BIT))
4289                 retval = DDI_FAILURE;
4290         if (!bge_chip_enable_engine(bgep, SEND_BD_SELECTOR_MODE_REG,
4291             STATE_MACHINE_ATTN_ENABLE_BIT))
4292                 retval = DDI_FAILURE;

4294         /*
4295          * Step 88: download firmware -- doesn't apply
4296          * Steps 89-90: enable Transmit & Receive MAC Engines
4297          */
4298         if (DEVICE_IS_5755_PLUS(bgep) ||
4299             MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
4300             MHCR_CHIP_ASIC_REV_5906) {
4301                 regval = bge_reg_get32(bgep, TRANSMIT_MAC_MODE_REG);
4302                 regval |= TRANSMIT_MODE_MBUF_LOCKUP_FIX;
4303         } else {
4304                 regval = 0;
4305         }
4306         if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
```

```
4307             MHCR_CHIP_ASIC_REV_5720) {
4308                 regval &= ~(TRANSMIT_MODE_HTX2B_JMB_FRM_LEN |
4309                     TRANSMIT_MODE_HTX2B_CNT_DN_MODE);
4310                 regval |= bge_reg_get32(bgep, TRANSMIT_MAC_MODE_REG) &
4311                     (TRANSMIT_MODE_HTX2B_JMB_FRM_LEN |
4312                     TRANSMIT_MODE_HTX2B_CNT_DN_MODE);
4313         }
4314         if (!bge_chip_enable_engine(bgep, TRANSMIT_MAC_MODE_REG, regval))
4143         if (!bge_chip_enable_engine(bgep, TRANSMIT_MAC_MODE_REG, 0))
4315                 retval = DDI_FAILURE;
4316 #ifdef BGE_IPMI_ASF
4317         if (!bgep->asf_enabled) {
4318                 if (!bge_chip_enable_engine(bgep, RECEIVE_MAC_MODE_REG,
4319                     RECEIVE_MODE_KEEP_VLAN_TAG))
4320                         retval = DDI_FAILURE;
4321         } else {
4322                 if (!bge_chip_enable_engine(bgep, RECEIVE_MAC_MODE_REG, 0))
4323                         retval = DDI_FAILURE;
4324         }
4325 #else
4326         if (!bge_chip_enable_engine(bgep, RECEIVE_MAC_MODE_REG,
4327             RECEIVE_MODE_KEEP_VLAN_TAG))
4328                 retval = DDI_FAILURE;
4329 #endif

4331         /*
4332          * Step 91: disable auto-polling of PHY status
4333          */
4334         bge_reg_put32(bgep, MI_MODE_REG, MI_MODE_DEFAULT);

4336         /*
4337          * Step 92: configure D0 power state (not required)
4338          * Step 93: initialise LED control register ()
4339          */
4340         ledctl = LED_CONTROL_DEFAULT;
4341         switch (bgep->chipid.device) {
4342         case DEVICE_ID_5700:
4343         case DEVICE_ID_5700x:
4344         case DEVICE_ID_5701:
4345                 /*
4346                  * Switch to 5700 (MAC) mode on these older chips
4347                  */
4348                 ledctl &= ~LED_CONTROL_LED_MODE_MASK;
4349                 ledctl |= LED_CONTROL_LED_MODE_5700;
4350                 break;

4352         default:
4353                 break;
4354         }
4355         bge_reg_put32(bgep, ETHERNET_MAC_LED_CONTROL_REG, ledctl);

4357         /*
4358          * Step 94: activate link
4359          */
4360         bge_reg_put32(bgep, MI_STATUS_REG, MI_STATUS_LINK);

4362         /*
4363          * Step 95: set up physical layer (PHY/SerDes)
4364          * restart autoneg (if required)
4365          */
4366         if (reset_phys)
4367                 if (bge_phys_update(bgep) == DDI_FAILURE)
4368                         retval = DDI_FAILURE;

4370         /*
4371          * Extra step (DSG): hand over all the Receive Buffers to the chip
```



```
4372          */
4373         for (ring = 0; ring < BGE_BUFF_RINGS_USED; ++ring)
4374                 bge_mbx_put(bgep, bgep->buff[ring].chip_mbx_reg,
4375                     bgep->buff[ring].rf_next);

4377         /*
4378          * MSI bits:The least significant MSI 16-bit word.
4379          * ISR will be triggered different.
4380          */
4381         if (bgep->intr_type == DDI_INTR_TYPE_MSI)
4382                 bge_reg_set32(bgep, HOST_COALESCE_MODE_REG, 0x70);

4384         /*
4385          * Extra step (DSG): select which interrupts are enabled
4386          *
4387          * Program the Ethernet MAC engine to signal attention on
4388          * Link Change events, then enable interrupts on MAC, DMA,
4389          * and FLOW attention signals.
4390          */
4391         bge_reg_set32(bgep, ETHERNET_MAC_EVENT_ENABLE_REG,
4392             ETHERNET_EVENT_LINK_INT |
4393             ETHERNET_STATUS_PCS_ERROR_INT);
4394 #ifdef BGE_IPMI_ASF
4395         if (bgep->asf_enabled) {
4396                 bge_reg_set32(bgep, MODE_CONTROL_REG,
4397                     MODE_INT_ON_FLOW_ATTN |
4398                     MODE_INT_ON_DMA_ATTN |
4399                     MODE_HOST_STACK_UP|
4400                     MODE_INT_ON_MAC_ATTN);
4401         } else {
4402 #endif
4403                 bge_reg_set32(bgep, MODE_CONTROL_REG,
4404                     MODE_INT_ON_FLOW_ATTN |
4405                     MODE_INT_ON_DMA_ATTN |
4406                     MODE_INT_ON_MAC_ATTN);
4407 #ifdef BGE_IPMI_ASF
4408         }
4409 #endif

4411         /*
4412          * Step 97: enable PCI interrupts!!!
4413          */
4414         if (bgep->intr_type == DDI_INTR_TYPE_FIXED)
4415                 bge_cfg_clr32(bgep, PCI_CONF_BGE_MHCR,
4416                     bgep->chipid.mask_pci_int);

4417         /*
4418          * All done!
4419          */
4420         bgep->bge_chip_state = BGE_CHIP_RUNNING;
4421         return (retval);
4422 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   68677 Fri Jan 11 13:34:55 2013
new/usr/src/uts/common/io/bge/bge_hw.h
Just the 5719/5720 changes
**********************************************************
```

```
/*
 * 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) 2002, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
 */

#ifndef _BGE_HW_H
#define _BGE_HW_H

#ifdef __cplusplus
extern "C" {
#endif

#include <sys/types.h>


/*
 * First section:
 *      Identification of the various Broadcom chips
 *
 * Note: the various ID values are *not* all unique ;-(
 *
 * Note: the presence of an ID here does *not* imply that the chip is
 * supported.  At this time, only the 5703C, 5704C, and 5704S devices
 * used on the motherboards of certain Sun products are supported.
 *
 * Note: the revision-id values in the PCI revision ID register are
 * *NOT* guaranteed correct.  Use the chip ID from the MHCR instead.
 */

#define VENDOR_ID_BROADCOM              0x14e4
#define VENDOR_ID_SUN                   0x108e

#define DEVICE_ID_5700                  0x1644
#define DEVICE_ID_5700x                 0x0003
#define DEVICE_ID_5701                  0x1645
#define DEVICE_ID_5702                  0x16a6
#define DEVICE_ID_5702fe                0x164d
```



```
#define DEVICE_ID_5703C                 0x16a7
#define DEVICE_ID_5703S                 0x1647
#define DEVICE_ID_5703                  0x16c7
#define DEVICE_ID_5704C                 0x1648
#define DEVICE_ID_5704S                 0x16a8
#define DEVICE_ID_5704                  0x1649
#define DEVICE_ID_5705C                 0x1653
#define DEVICE_ID_5705_2                0x1654
#define DEVICE_ID_5717                  0x1655
#define DEVICE_ID_5718                  0x1656
#define DEVICE_ID_5724                  0x165c
#define DEVICE_ID_5705M                 0x165d
#define DEVICE_ID_5705MA3               0x165e
#define DEVICE_ID_5719                  0x1657
#define DEVICE_ID_5720                  0x165f
#define DEVICE_ID_5705F                 0x166e
#define DEVICE_ID_5780                  0x166a
#define DEVICE_ID_5782                  0x1696
#define DEVICE_ID_5784M                 0x1698
#define DEVICE_ID_5785                  0x1699
#define DEVICE_ID_5787                  0x169b
#define DEVICE_ID_5787M                 0x1693
#define DEVICE_ID_5788                  0x169c
#define DEVICE_ID_5789                  0x169d
#define DEVICE_ID_5751                  0x1677
#define DEVICE_ID_5751M                 0x167d
#define DEVICE_ID_5752                  0x1600
#define DEVICE_ID_5752M                 0x1601
#define DEVICE_ID_5753                  0x16fd
#define DEVICE_ID_5754                  0x167a
#define DEVICE_ID_5755                  0x167b
#define DEVICE_ID_5755M                 0x1673
#define DEVICE_ID_5756M                 0x1674
#define DEVICE_ID_5721                  0x1659
#define DEVICE_ID_5722                  0x165a
#define DEVICE_ID_5723                  0x165b
#define DEVICE_ID_5714C                 0x1668
#define DEVICE_ID_5714S                 0x1669
#define DEVICE_ID_5715C                 0x1678
#define DEVICE_ID_5715S                 0x1679
#define DEVICE_ID_5761                  0x1681
#define DEVICE_ID_5761E                 0x1680
#define DEVICE_ID_5761S                 0x1688
#define DEVICE_ID_5761SE                0x1689
#define DEVICE_ID_5764                  0x1684
#define DEVICE_ID_5906                  0x1712
#define DEVICE_ID_5906M                 0x1713
#define DEVICE_ID_57760                 0x1690
#define DEVICE_ID_57780                 0x1692
#define DEVICE_ID_57788                 0x1691
#define DEVICE_ID_57790                 0x1694
#define DEVICE_ID_57781                 0x16b1
#define DEVICE_ID_57785                 0x16b5
#define DEVICE_ID_57761                 0x16b0
#define DEVICE_ID_57765                 0x16b4
#define DEVICE_ID_57791                 0x16b2
#define DEVICE_ID_57795                 0x16b6
#define DEVICE_ID_57762                 0x1682
#define DEVICE_ID_57766                 0x1686
#define DEVICE_ID_57786                 0x16b3
#define DEVICE_ID_57782                 0x16b7

#define REVISION_ID_5700_B0             0x10
#define REVISION_ID_5700_B2             0x12
#define REVISION_ID_5700_B3             0x13
#define REVISION_ID_5700_C0             0x20
```

```
128 #define REVISION_ID_5700_C1             0x21
129 #define REVISION_ID_5700_C2             0x22

131 #define REVISION_ID_5701_A0             0x08
132 #define REVISION_ID_5701_A2             0x12
133 #define REVISION_ID_5701_A3             0x15

135 #define REVISION_ID_5702_A0             0x00

137 #define REVISION_ID_5703_A0             0x00
138 #define REVISION_ID_5703_A1             0x01
139 #define REVISION_ID_5703_A2             0x02

141 #define REVISION_ID_5704_A0             0x00
142 #define REVISION_ID_5704_A1             0x01
143 #define REVISION_ID_5704_A2             0x02
144 #define REVISION_ID_5704_A3             0x03
145 #define REVISION_ID_5704_B0             0x10

147 #define REVISION_ID_5705_A0             0x00
148 #define REVISION_ID_5705_A1             0x01
149 #define REVISION_ID_5705_A2             0x02
150 #define REVISION_ID_5705_A3             0x03

152 #define REVISION_ID_5721_A0             0x00
153 #define REVISION_ID_5721_A1             0x01

155 #define REVISION_ID_5751_A0             0x00
156 #define REVISION_ID_5751_A1             0x01

158 #define REVISION_ID_5714_A0             0x00
159 #define REVISION_ID_5714_A1             0x01
160 #define REVISION_ID_5714_A2             0xA2
161 #define REVISION_ID_5714_A3             0xA3

163 #define REVISION_ID_5715_A0             0x00
164 #define REVISION_ID_5715_A1             0x01
165 #define REVISION_ID_5715_A2             0xA2

167 #define REVISION_ID_5715S_A0            0x00
168 #define REVISION_ID_5715S_A1            0x01

170 #define REVISION_ID_5754_A0             0x00
171 #define REVISION_ID_5754_A1             0x01

173 #define DEVICE_5704_SERIES_CHIPSETS(bgep)\
174                 ((bgep->chipid.device == DEVICE_ID_5700) ||\
175                 (bgep->chipid.device == DEVICE_ID_5701) ||\
176                 (bgep->chipid.device == DEVICE_ID_5702) ||\
177                 (bgep->chipid.device == DEVICE_ID_5702fe)||\
178                 (bgep->chipid.device == DEVICE_ID_5703C) ||\
179                 (bgep->chipid.device == DEVICE_ID_5703S) ||\
180                 (bgep->chipid.device == DEVICE_ID_5703) ||\
181                 (bgep->chipid.device == DEVICE_ID_5704C) ||\
182                 (bgep->chipid.device == DEVICE_ID_5704S) ||\
183                 (bgep->chipid.device == DEVICE_ID_5704))

185 #define DEVICE_5702_SERIES_CHIPSETS(bgep) \
186                 ((bgep->chipid.device == DEVICE_ID_5702) ||\
187                 (bgep->chipid.device == DEVICE_ID_5702fe))

189 #define DEVICE_5705_SERIES_CHIPSETS(bgep) \
190                 ((bgep->chipid.device == DEVICE_ID_5705C) ||\
191                 (bgep->chipid.device == DEVICE_ID_5705M) ||\
192                 (bgep->chipid.device == DEVICE_ID_5705MA3) ||\
193                 (bgep->chipid.device == DEVICE_ID_5705F) ||\
```



```
194                 (bgep->chipid.device == DEVICE_ID_5780) ||\
195                 (bgep->chipid.device == DEVICE_ID_5782) ||\
196                 (bgep->chipid.device == DEVICE_ID_5788) ||\
197                 (bgep->chipid.device == DEVICE_ID_5705_2) ||\
198                 (bgep->chipid.device == DEVICE_ID_5754) ||\
199                 (bgep->chipid.device == DEVICE_ID_5755) ||\
200                 (bgep->chipid.device == DEVICE_ID_5756M) ||\
201                 (bgep->chipid.device == DEVICE_ID_5753))

203 #define DEVICE_5721_SERIES_CHIPSETS(bgep) \
204                 ((bgep->chipid.device == DEVICE_ID_5721) ||\
205                 (bgep->chipid.device == DEVICE_ID_5751) ||\
206                 (bgep->chipid.device == DEVICE_ID_5751M) ||\
207                 (bgep->chipid.device == DEVICE_ID_5752) ||\
208                 (bgep->chipid.device == DEVICE_ID_5752M) ||\
209                 (bgep->chipid.device == DEVICE_ID_5789))

211 #define DEVICE_5717_SERIES_CHIPSETS(bgep) \
212                 (bgep->chipid.device == DEVICE_ID_5717) ||\
213                 (bgep->chipid.device == DEVICE_ID_5718) ||\
214                 (bgep->chipid.device == DEVICE_ID_5719) ||\
215                 (bgep->chipid.device == DEVICE_ID_5720) ||\
216                 (bgep->chipid.device == DEVICE_ID_5724)

218 #define DEVICE_5723_SERIES_CHIPSETS(bgep) \
219                 ((bgep->chipid.device == DEVICE_ID_5723) ||\
220                 (bgep->chipid.device == DEVICE_ID_5761) ||\
221                 (bgep->chipid.device == DEVICE_ID_5761E) ||\
222                 (bgep->chipid.device == DEVICE_ID_5761S) ||\
223                 (bgep->chipid.device == DEVICE_ID_5761SE) ||\
224                 (bgep->chipid.device == DEVICE_ID_5764) ||\
225                 (bgep->chipid.device == DEVICE_ID_5784M) ||\
226                 (bgep->chipid.device == DEVICE_ID_5785) ||\
227                 (bgep->chipid.device == DEVICE_ID_57760) ||\
228                 (bgep->chipid.device == DEVICE_ID_57780) ||\
229                 (bgep->chipid.device == DEVICE_ID_57788) ||\
230                 (bgep->chipid.device == DEVICE_ID_57790))

232 #define DEVICE_5714_SERIES_CHIPSETS(bgep) \
233                 ((bgep->chipid.device == DEVICE_ID_5714C) ||\
234                 (bgep->chipid.device == DEVICE_ID_5714S) ||\
235                 (bgep->chipid.device == DEVICE_ID_5715C) ||\
236                 (bgep->chipid.device == DEVICE_ID_5715S))

238 #define DEVICE_5906_SERIES_CHIPSETS(bgep) \
239                 ((bgep->chipid.device == DEVICE_ID_5906) ||\
240                 (bgep->chipid.device == DEVICE_ID_5906M))

243 #define CHIP_TYPE_5705_PLUS   (1 << 0)
244 #define CHIP_TYPE_5750_PLUS   (1 << 1)
245 #define CHIP_TYPE_5780_CLASS  (1 << 2)
246 #define CHIP_TYPE_5755_PLUS   (1 << 3)
247 #define CHIP_TYPE_57765_CLASS (1 << 4)
248 #define CHIP_TYPE_57765_PLUS  (1 << 5)
249 #define CHIP_TYPE_5717_PLUS   (1 << 6)

251 #define DEVICE_IS_57765_PLUS(bgep) \
252         (bgep->chipid.chip_type & CHIP_TYPE_57765_PLUS)
253 #define DEVICE_IS_5755_PLUS(bgep) \
254         (bgep->chipid.chip_type & CHIP_TYPE_5755_PLUS)

256 /*
257  * Second section:
258  *      Offsets of important registers & definitions for bits therein
259  */
```

```
261 /*
262  * PCI-X registers & bits
263  */
264 #define PCIX_CONF_COMM                  0x42
265 #define PCIX_COMM_RELAXED               0x0002

267 /*
268  * Miscellaneous Host Control Register, in PCI config space
269  */
270 #define PCI_CONF_BGE_MHCR               0x68
271 #define MHCR_CHIP_REV_MASK              0xffff0000
272 #define MHCR_CHIP_REV_SHIFT             16
273 #define MHCR_ENABLE_TAGGED_STATUS_MODE  0x00000200
274 #define MHCR_MASK_INTERRUPT_MODE        0x00000100
275 #define MHCR_ENABLE_INDIRECT_ACCESS     0x00000080
276 #define MHCR_ENABLE_REGISTER_WORD_SWAP  0x00000040
277 #define MHCR_ENABLE_CLOCK_CONTROL_WRITE 0x00000020
278 #define MHCR_ENABLE_PCI_STATE_WRITE     0x00000010
279 #define MHCR_ENABLE_ENDIAN_WORD_SWAP    0x00000008
280 #define MHCR_ENABLE_ENDIAN_BYTE_SWAP    0x00000004
281 #define MHCR_MASK_PCI_INT_OUTPUT        0x00000002
282 #define MHCR_CLEAR_INTERRUPT_INTA       0x00000001

284 #define MHCR_CHIP_REV_5703_A0           0x1000
285 #define MHCR_CHIP_REV_5704_A0           0x2000
286 #define MHCR_CHIP_REV_5751_A0           0x4000
287 #define MHCR_CHIP_REV_5721_A0           0x4100
288 #define MHCR_CHIP_REV_5755_A0           0xa000
289 #define MHCR_CHIP_REV_5755_A1           0xa001
290 #define MHCR_CHIP_REV_5719_A0           0x05719000
291 #define MHCR_CHIP_REV_5720_A0           0x05720000
251 #define MHCR_CHIP_REV_5700_B0           0x71000000
252 #define MHCR_CHIP_REV_5700_B2           0x71020000
253 #define MHCR_CHIP_REV_5700_B3           0x71030000
254 #define MHCR_CHIP_REV_5700_C0           0x72000000
255 #define MHCR_CHIP_REV_5700_C1           0x72010000
256 #define MHCR_CHIP_REV_5700_C2           0x72020000

293 #define MHCR_CHIP_ASIC_REV(ChipRevId)   ((ChipRevId) >> 12)
294 #define MHCR_CHIP_ASIC_REV_5700         0x07
295 #define MHCR_CHIP_ASIC_REV_5701         0x00
296 #define MHCR_CHIP_ASIC_REV_5703         0x01
297 #define MHCR_CHIP_ASIC_REV_5704         0x02
298 #define MHCR_CHIP_ASIC_REV_5705         0x03
299 #define MHCR_CHIP_ASIC_REV_5750         0x04
300 #define MHCR_CHIP_ASIC_REV_5752         0x06
301 #define MHCR_CHIP_ASIC_REV_5780         0x08
302 #define MHCR_CHIP_ASIC_REV_5714         0x09
303 #define MHCR_CHIP_ASIC_REV_5755         0x0a
304 #define MHCR_CHIP_ASIC_REV_5787         0x0b
305 #define MHCR_CHIP_ASIC_REV_5906         0x0c
306 #define MHCR_CHIP_ASIC_REV_PRODID       0x0f
307 #define MHCR_CHIP_ASIC_REV_5784         0x5784
308 #define MHCR_CHIP_ASIC_REV_5761         0x5761
309 #define MHCR_CHIP_ASIC_REV_5785         0x5785
310 #define MHCR_CHIP_ASIC_REV_5717         0x5717
311 #define MHCR_CHIP_ASIC_REV_5719         0x5719
312 #define MHCR_CHIP_ASIC_REV_5720         0x5720
313 #define MHCR_CHIP_ASIC_REV_57780        0x57780
314 #define MHCR_CHIP_ASIC_REV_57765        0x57785
315 #define MHCR_CHIP_ASIC_REV_57766        0x57766
258 #define MHCR_CHIP_REV_5701_A0           0x00000000
259 #define MHCR_CHIP_REV_5701_A2           0x00020000
260 #define MHCR_CHIP_REV_5701_A3           0x00030000
261 #define MHCR_CHIP_REV_5701_A5           0x01050000
```



```
263 #define MHCR_CHIP_REV_5702_A0           0x10000000
264 #define MHCR_CHIP_REV_5702_A1           0x10010000
265 #define MHCR_CHIP_REV_5702_A2           0x10020000

267 #define MHCR_CHIP_REV_5703_A0           0x10000000
268 #define MHCR_CHIP_REV_5703_A1           0x10010000
269 #define MHCR_CHIP_REV_5703_A2           0x10020000
270 #define MHCR_CHIP_REV_5703_B0           0x11000000
271 #define MHCR_CHIP_REV_5703_B1           0x11010000

273 #define MHCR_CHIP_REV_5704_A0           0x20000000
274 #define MHCR_CHIP_REV_5704_A1           0x20010000
275 #define MHCR_CHIP_REV_5704_A2           0x20020000
276 #define MHCR_CHIP_REV_5704_A3           0x20030000
277 #define MHCR_CHIP_REV_5704_B0           0x21000000

279 #define MHCR_CHIP_REV_5705_A0           0x30000000
280 #define MHCR_CHIP_REV_5705_A1           0x30010000
281 #define MHCR_CHIP_REV_5705_A2           0x30020000
282 #define MHCR_CHIP_REV_5705_A3           0x30030000
283 #define MHCR_CHIP_REV_5705_A5           0x30050000

285 #define MHCR_CHIP_REV_5782_A0           0x30030000
286 #define MHCR_CHIP_REV_5782_A1           0x30030088

288 #define MHCR_CHIP_REV_5788_A1           0x30050000

290 #define MHCR_CHIP_REV_5751_A0           0x40000000
291 #define MHCR_CHIP_REV_5751_A1           0x40010000

293 #define MHCR_CHIP_REV_5721_A0           0x41000000
294 #define MHCR_CHIP_REV_5721_A1           0x41010000

296 #define MHCR_CHIP_REV_5714_A0           0x50000000
297 #define MHCR_CHIP_REV_5714_A1           0x90010000

299 #define MHCR_CHIP_REV_5715_A0           0x50000000
300 #define MHCR_CHIP_REV_5715_A1           0x90010000

302 #define MHCR_CHIP_REV_5715S_A0          0x50000000
303 #define MHCR_CHIP_REV_5715S_A1          0x90010000

305 #define MHCR_CHIP_REV_5754_A0           0xb0000000
306 #define MHCR_CHIP_REV_5754_A1           0xb0010000

308 #define MHCR_CHIP_REV_5787_A0           0xb0000000
309 #define MHCR_CHIP_REV_5787_A1           0xb0010000
310 #define MHCR_CHIP_REV_5787_A2           0xb0020000

312 #define MHCR_CHIP_REV_5755_A0           0xa0000000
313 #define MHCR_CHIP_REV_5755_A1           0xa0010000

315 #define MHCR_CHIP_REV_5906_A0           0xc0000000
316 #define MHCR_CHIP_REV_5906_A1           0xc0010000
317 #define MHCR_CHIP_REV_5906_A2           0xc0020000

319 #define MHCR_CHIP_REV_5723_A0           0xf0000000
320 #define MHCR_CHIP_REV_5723_A1           0xf0010000
321 #define MHCR_CHIP_REV_5723_A2           0xf0020000
322 #define MHCR_CHIP_REV_5723_B0           0xf1000000

324 #define MHCR_CHIP_ASIC_REV(ChipRevId)   ((ChipRevId) & 0xf0000000)
325 #define MHCR_CHIP_ASIC_REV_5700         (0x7 << 28)
326 #define MHCR_CHIP_ASIC_REV_5701         (0x0 << 28)
327 #define MHCR_CHIP_ASIC_REV_5703         (0x1 << 28)
```

```
328 #define MHCR_CHIP_ASIC_REV_5704         (0x2 << 28)
329 #define MHCR_CHIP_ASIC_REV_5705         (0x3 << 28)
330 #define MHCR_CHIP_ASIC_REV_5721_5751    (0x4 << 28)
331 #define MHCR_CHIP_ASIC_REV_5714         (0x5 << 28)
332 #define MHCR_CHIP_ASIC_REV_5752         (0x6 << 28)
333 #define MHCR_CHIP_ASIC_REV_5754         (0xb << 28)
334 #define MHCR_CHIP_ASIC_REV_5787         ((uint32_t)0xb << 28)
335 #define MHCR_CHIP_ASIC_REV_5755         ((uint32_t)0xa << 28)
336 #define MHCR_CHIP_ASIC_REV_5715         ((uint32_t)0x9 << 28)
337 #define MHCR_CHIP_ASIC_REV_5906         ((uint32_t)0xc << 28)
338 #define MHCR_CHIP_ASIC_REV_5723         ((uint32_t)0xf << 28)


317 /*
318  * PCI DMA read/write Control Register, in PCI config space
319  *
320  * Note that several fields previously defined here have been deleted
321  * as they are not implemented in the 5703/4.
322  *
323  * Note: the value of this register is critical.  It is possible to
324  * cause various unpleasant effects (DTOs, transaction deadlock, etc)
325  * by programming the wrong value.  The value #defined below has been
326  * tested and shown to avoid all known problems.  If it is to be changed,
327  * correct operation must be reverified on all supported platforms.
328  *
329  * In particular, we set both watermark fields to 2xCacheLineSize (128)
330  * bytes and DMA_MIN_BEATS to 0 in order to avoid unfortunate interactions
331  * with Tomatillo's internal pipelines, that otherwise result in stalls,
332  * repeated retries, and DTOs.
333  */
334 #define PCI_CONF_BGE_PDRWCR             0x6c
335 #define PDRWCR_RWCMD_MASK               0xFF000000
336 #define PDRWCR_PCIX32_BUGFIX_MASK       0x00800000
337 #define PDRWCR_WRITE_WATERMARK_MASK     0x00380000
338 #define PDRWCR_READ_WATERMARK_MASK      0x00070000
339 #define PDRWCR_CONCURRENCY_MASK         0x0000c000
340 #define PDRWCR_5704_FLOP_ON_RETRY       0x00008000
341 #define PDRWCR_ONE_DMA_AT_ONCE          0x00004000
342 #define PDRWCR_MIN_BEAT_MASK            0x000000ff

344 /*
345  * These are the actual values to be put into the fields shown above
346  */
347 #define PDRWCR_RWCMDS                   0x76000000      /* MW and MR    */
348 #define PDRWCR_DMA_WRITE_WATERMARK      0x00180000      /* 011 => 128   */
349 #define PDRWCR_DMA_READ_WATERMARK       0x00030000      /* 011 => 128   */
350 #define PDRWCR_MIN_BEATS                0x00000000

352 #define PDRWCR_VAR_DEFAULT              0x761b0000
353 #define PDRWCR_VAR_5721                 0x76180000
354 #define PDRWCR_VAR_5714                 0x76148000      /* OR of above  */
355 #define PDRWCR_VAR_5715                 0x76144000      /* OR of above  */
356 #define PDRWCR_VAR_5717                 0x00380000

358 /*
359  * PCI State Register, in PCI config space
360  *
361  * Note: this register is read-only unless the ENABLE_PCI_STATE_WRITE bit
362  * is set in the MHCR, EXCEPT for the RETRY_SAME_DMA bit which is always RW
363  */
364 #define PCI_CONF_BGE_PCISTATE           0x70
365 #define PCISTATE_RETRY_SAME_DMA         0x00002000
366 #define PCISTATE_FLAT_VIEW              0x00000100
367 #define PCISTATE_EXT_ROM_RETRY          0x00000040
368 #define PCISTATE_EXT_ROM_ENABLE         0x00000020
369 #define PCISTATE_BUS_IS_32_BIT          0x00000010
```



```
370 #define PCISTATE_BUS_IS_FAST            0x00000008
371 #define PCISTATE_BUS_IS_PCI             0x00000004
372 #define PCISTATE_INTA_STATE             0x00000002
373 #define PCISTATE_FORCE_RESET            0x00000001

375 /*
376  * PCI Clock Control Register, in PCI config space
377  */
378 #define PCI_CONF_BGE_CLKCTL             0x74
379 #define CLKCTL_PCIE_PLP_DISABLE         0x80000000
380 #define CLKCTL_PCIE_DLP_DISABLE         0x40000000
381 #define CLKCTL_PCIE_TLP_DISABLE         0x20000000
382 #define CLKCTL_PCI_READ_TOO_LONG_FIX    0x04000000
383 #define CLKCTL_PCI_WRITE_TOO_LONG_FIX   0x02000000
384 #define CLKCTL_PCIE_A0_FIX              0x00101000

386 /*
387  * Dual MAC Control Register, in PCI config space
388  */
389 #define PCI_CONF_BGE_DUAL_MAC_CONTROL   0xB8
390 #define DUALMAC_CHANNEL_CONTROL_MASK    0x00000003      /* RW   */
391 #define DUALMAC_CHANNEL_ID_MASK         0x00000004      /* RO   */

393 /*
394  * Register Indirect Access Address Register, 0x78 in PCI config
395  * space.  Once this is set, accesses to the Register Indirect
396  * Access Data Register (0x80) refer to the register whose address
397  * is given by *this* register.  This allows access to all the
398  * operating registers, while using only config space accesses.
399  *
400  * Note that the address written to the RIIAR should lie in one
401  * of the following ranges:
402  *      0x00000000 <= address < 0x00008000 (regular registers)
403  *      0x00030000 <= address < 0x00034000 (RxRISC scratchpad)
404  *      0x00034000 <= address < 0x00038000 (TxRISC scratchpad)
405  *      0x00038000 <= address < 0x00038800 (RxRISC ROM)
406  */
407 #define PCI_CONF_BGE_RIAAR              0x78
408 #define PCI_CONF_BGE_RIADR              0x80

410 #define RIAAR_REGISTER_MIN              0x00000000
411 #define RIAAR_REGISTER_MAX              0x00008000
412 #define RIAAR_RX_SCRATCH_MIN            0x00030000
413 #define RIAAR_RX_SCRATCH_MAX            0x00034000
414 #define RIAAR_TX_SCRATCH_MIN            0x00034000
415 #define RIAAR_TX_SCRATCH_MAX            0x00038000
416 #define RIAAR_RXROM_MIN                 0x00038000
417 #define RIAAR_RXROM_MAX                 0x00038800

419 /*
420  * Memory Window Base Address Register, 0x7c in PCI config space
421  * Once this is set, accesses to the Memory Window Data Access Register
422  * (0x84) refer to the word of NIC-local memory whose address is given
423  * by this register.  When used in this way, the whole of the address
424  * written to this register is significant.
425  *
426  * This register also provides the 32K-aligned base address for a 32K
427  * region of NIC-local memory that the host can directly address in
428  * the upper 32K of the 64K of PCI memory space allocated to the chip.
429  * In this case, the bottom 15 bits of the register are ignored.
430  *
431  * Note that the address written to the MWBAR should lie in the range
432  * 0x00000000 <= address < 0x00020000.  The rest of the range up to 1M
433  * (i.e. 0x00200000 <= address < 0x01000000) would be valid if external
434  * memory were present, but it's only supported on the 5700, not the
435  * 5701/5703/5704.
```

```
 436  */
 437 #define PCI_CONF_BGE_MWBAR              0x7c
 438 #define PCI_CONF_BGE_MWDAR              0x84
 439 #define MWBAR_GRANULARITY               0x00008000      /* 32k  */
 440 #define MWBAR_GRANULE_MASK              (MWBAR_GRANULARITY-1)
 441 #define MWBAR_ONCHIP_MAX                0x00020000      /* 128k */

 443 /*
 444  * The PCI express device control register and device status register
 445  * which are only applicable on BCM5751 and BCM5721.
 446  */
 447 #define PCI_CONF_DEV_CTRL               0xd8
 448 #define PCI_CONF_DEV_CTRL_5723          0xd4
 449 #define READ_REQ_SIZE_MAX               0x5000
 450 #define DEV_CTRL_NO_SNOOP               0x0800
 451 #define DEV_CTRL_RELAXED                0x0010

 453 #define PCI_CONF_DEV_STUS               0xda
 454 #define PCI_CONF_DEV_STUS_5723          0xd6
 455 #define DEVICE_ERROR_STUS               0xf

 457 #define PCI_CONF_PRODID_ASICREV         0x000000bc
 458 #define PCI_CONF_GEN2_PRODID_ASICREV    0x000000f4
 459 #define PCI_CONF_GEN15_PRODID_ASICREV   0x000000fc

 461 #define NIC_MEM_WINDOW_OFFSET           0x00008000      /* 32k  */

 463 /*
 464  * Where to find things in NIC-local (on-chip) memory
 465  */
 466 #define NIC_MEM_SEND_RINGS              0x0100
 467 #define NIC_MEM_SEND_RING(ring)         (0x0100+16*(ring))
 468 #define NIC_MEM_RECV_RINGS              0x0200
 469 #define NIC_MEM_RECV_RING(ring)         (0x0200+16*(ring))
 470 #define NIC_MEM_STATISTICS              0x0300
 471 #define NIC_MEM_STATISTICS_SIZE         0x0800
 472 #define NIC_MEM_STATUS_BLOCK            0x0b00
 473 #define NIC_MEM_STATUS_SIZE             0x0050
 474 #define NIC_MEM_GENCOMM                 0x0b50


 477 /*
 478  * Note: the (non-bogus) values below are appropriate for systems
 479  * without external memory.  They would be different on a 5700 with
 480  * external memory.
 481  *
 482  * Note: The higher send ring addresses and the mini ring shadow
 483  * buffer address are dummies - systems without external memory
 484  * are limited to 4 send rings and no mini receive ring.
 485  */
 486 #define NIC_MEM_SHADOW_DMA              0x2000
 487 #define NIC_MEM_SHADOW_SEND_1_4         0x4000
 488 #define NIC_MEM_SHADOW_SEND_5_6         0x6000          /* bogus        */
 489 #define NIC_MEM_SHADOW_SEND_7_8         0x7000          /* bogus        */
 490 #define NIC_MEM_SHADOW_SEND_9_16        0x8000          /* bogus        */
 491 #define NIC_MEM_SHADOW_BUFF_STD         0x6000
 492 #define NIC_MEM_SHADOW_BUFF_STD_5717            0x40000
 493 #define NIC_MEM_SHADOW_BUFF_JUMBO       0x7000
 494 #define NIC_MEM_SHADOW_BUFF_MINI        0x8000          /* bogus        */
 495 #define NIC_MEM_SHADOW_SEND_RING(ring, nslots)  (0x4000 + 4*(ring)*(nslots))

 497 /*
 498  * Put this in the GENCOMM port to tell the firmware not to run PXE
 499  */
 500 #define T3_MAGIC_NUMBER                 0x4b657654u
```



```
 502 /*
 503  * The remaining registers appear in the low 32K of regular
 504  * PCI Memory Address Space
 505  */

 507 /*
 508  * All the state machine control registers below have at least a
 509  * <RESET> bit and an <ENABLE> bit as defined below.  Some also
 510  * have an <ATTN_ENABLE> bit.
 511  */
 512 #define STATE_MACHINE_ATTN_ENABLE_BIT   0x00000004
 513 #define STATE_MACHINE_ENABLE_BIT        0x00000002
 514 #define STATE_MACHINE_RESET_BIT         0x00000001

 516 #define TRANSMIT_MAC_MODE_REG           0x045c
 517 #define SEND_DATA_INITIATOR_MODE_REG    0x0c00
 518 #define SEND_DATA_COMPLETION_MODE_REG   0x1000
 519 #define SEND_BD_SELECTOR_MODE_REG       0x1400
 520 #define SEND_BD_INITIATOR_MODE_REG      0x1800
 521 #define SEND_BD_COMPLETION_MODE_REG     0x1c00

 523 #define RECEIVE_MAC_MODE_REG            0x0468
 524 #define RCV_LIST_PLACEMENT_MODE_REG     0x2000
 525 #define RCV_DATA_BD_INITIATOR_MODE_REG  0x2400
 526 #define RCV_DATA_COMPLETION_MODE_REG    0x2800
 527 #define RCV_BD_INITIATOR_MODE_REG       0x2c00
 528 #define RCV_BD_COMPLETION_MODE_REG      0x3000
 529 #define RCV_LIST_SELECTOR_MODE_REG      0x3400

 531 #define MBUF_CLUSTER_FREE_MODE_REG      0x3800
 532 #define HOST_COALESCE_MODE_REG          0x3c00
 533 #define MEMORY_ARBITER_MODE_REG         0x4000
 534 #define BUFFER_MANAGER_MODE_REG         0x4400
 535 #define READ_DMA_MODE_REG               0x4800
 536 #define READ_DMA_RESERVED_CONTROL_REG   0x4900
 537 #define WRITE_DMA_MODE_REG              0x4c00
 538 #define DMA_COMPLETION_MODE_REG         0x6400

 540 /*
 541  * Other bits in some of the above state machine control registers
 542  */

 544 /*
 545  * Transmit MAC Mode Register
 546  * (TRANSMIT_MAC_MODE_REG, 0x045c)
 547  */
 548 #define TRANSMIT_MODE_HTX2B_CNT_DN_MODE 0x00800000
 549 #define TRANSMIT_MODE_HTX2B_JMB_FRM_LEN 0x00400000
 550 #define TRANSMIT_MODE_MBUF_LOCKUP_FIX   0x00000100
 551 #define TRANSMIT_MODE_LONG_PAUSE        0x00000040
 552 #define TRANSMIT_MODE_BIG_BACKOFF       0x00000020
 553 #define TRANSMIT_MODE_FLOW_CONTROL      0x00000010

 555 /*
 556  * Receive MAC Mode Register
 557  * (RECEIVE_MAC_MODE_REG, 0x0468)
 558  */
 559 #define RECEIVE_MODE_KEEP_VLAN_TAG      0x00000400
 560 #define RECEIVE_MODE_NO_CRC_CHECK       0x00000200
 561 #define RECEIVE_MODE_PROMISCUOUS        0x00000100
 562 #define RECEIVE_MODE_LENGTH_CHECK       0x00000080
 563 #define RECEIVE_MODE_ACCEPT_RUNTS       0x00000040
 564 #define RECEIVE_MODE_ACCEPT_OVERSIZE    0x00000020
 565 #define RECEIVE_MODE_KEEP_PAUSE         0x00000010
 566 #define RECEIVE_MODE_FLOW_CONTROL       0x00000004
```

```
568 /*
569  * Receive BD Initiator Mode Register
570  * (RCV_BD_INITIATOR_MODE_REG, 0x2c00)
571  *
572  * Each of these bits controls whether ATTN is asserted
573  * on a particular condition
574  */
575 #define RCV_BD_DISABLED_RING_ATTN       0x00000004

577 /*
578  * Receive Data & Receive BD Initiator Mode Register
579  * (RCV_DATA_BD_INITIATOR_MODE_REG, 0x2400)
580  *
581  * Each of these bits controls whether ATTN is asserted
582  * on a particular condition
583  */
584 #define RCV_DATA_BD_ILL_RING_ATTN       0x00000010
585 #define RCV_DATA_BD_FRAME_SIZE_ATTN     0x00000008
586 #define RCV_DATA_BD_NEED_JUMBO_ATTN     0x00000004

588 #define RCV_DATA_BD_ALL_ATTN_BITS       0x0000001c

590 /*
591  * Host Coalescing Mode Control Register
592  * (HOST_COALESCE_MODE_REG, 0x3c00)
593  */
594 #define COALESCE_64_BYTE_RINGS          12
595 #define COALESCE_NO_INT_ON_COAL_FORCE   0x00001000
596 #define COALESCE_NO_INT_ON_DMAD_FORCE   0x00000800
597 #define COALESCE_CLR_TICKS_TX           0x00000400
598 #define COALESCE_CLR_TICKS_RX           0x00000200
599 #define COALESCE_32_BYTE_STATUS         0x00000100
600 #define COALESCE_64_BYTE_STATUS         0x00000080
601 #define COALESCE_NOW                    0x00000008

603 /*
604  * Memory Arbiter Mode Register
605  * (MEMORY_ARBITER_MODE_REG, 0x4000)
606  */
607 #define MEMORY_ARBITER_ENABLE           0x00000002

609 /*
610  * Buffer Manager Mode Register
611  * (BUFFER_MANAGER_MODE_REG, 0x4400)
612  *
613  * In addition to the usual error-attn common to most state machines
614  * this register has a separate bit for attn on running-low-on-mbufs
615  */
616 #define BUFF_MGR_TEST_MODE              0x00000008
617 #define BUFF_MGR_MBUF_LOW_ATTN_ENABLE   0x00000010
618 #define BUFF_MGR_NO_TX_UNDERRUN         0x80000000

620 #define BUFF_MGR_ALL_ATTN_BITS          0x00000014

622 /*
623  * Read and Write DMA Mode Registers (READ_DMA_MODE_REG,
624  * 0x4800, READ_DMA_RESERVED_CONTROL_REG, 0x4900,
625  * WRITE_DMA_MODE_REG, 0x4c00)
639  * 0x4800 and WRITE_DMA_MODE_REG, 0x4c00)
626  *
627  * These registers each contain a 2-bit priority field, which controls
628  * the relative priority of that type of DMA (read vs. write vs. MSI),
629  * and a set of bits that control whether ATTN is asserted on each
630  * particular condition
631  */
632 #define DMA_PRIORITY_MASK               0xc0000000
```



```
633 #define DMA_PRIORITY_SHIFT              30
634 #define ALL_DMA_ATTN_BITS               0x000003fc

636 #define RDMA_RSRVCTRL_FIFO_OFLW_FIX     0x00000004
637 #define RDMA_RSRVCTRL_FIFO_LWM_1_5K     0x00000c00
638 #define RDMA_RSRVCTRL_FIFO_LWM_MASK     0x00000ff0
639 #define RDMA_RSRVCTRL_FIFO_HWM_1_5K     0x000c0000
640 #define RDMA_RSRVCTRL_FIFO_HWM_MASK     0x000ff000
641 #define RDMA_RSRVCTRL_TXMRGN_320B       0x28000000
642 #define RDMA_RSRVCTRL_TXMRGN_MASK       0xffe00000

645 /*
646  * BCM5755, 5755M, 5906, 5906M only
647  * 1 - Enable Fix. Device will send out the status block before
648  *     the interrupt message
649  * 0 - Disable fix. Device will send out the interrupt message
650  *     before the status block
651  */
652 #define DMA_STATUS_TAG_FIX_CQ12384      0x20000000

654 /* 5720 only */
655 #define DMA_H2BNC_VLAN_DET              0x20000000

658 /*
659  * End of state machine control register definitions
660  */

663 /*
664  * High priority mailbox registers.
665  * Mailbox Registers (8 bytes each, but high half unused)
666  */
667 #define INTERRUPT_MBOX_0_REG            0x0200
668 #define INTERRUPT_MBOX_1_REG            0x0208
669 #define INTERRUPT_MBOX_2_REG            0x0210
670 #define INTERRUPT_MBOX_3_REG            0x0218
671 #define INTERRUPT_MBOX_REG(n)           (0x0200+8*(n))

673 /*
674  * Low priority mailbox registers, for BCM5906, BCM5906M.
675  */
676 #define INTERRUPT_LP_MBOX_0_REG         0x5800

678 /*
679  * Ring Producer/Consumer Index (Mailbox) Registers
680  */
681 #define RECV_STD_PROD_INDEX_REG         0x0268
682 #define RECV_JUMBO_PROD_INDEX_REG       0x0270
683 #define RECV_MINI_PROD_INDEX_REG        0x0278
684 #define RECV_RING_CONS_INDEX_REGS       0x0280
685 #define SEND_RING_HOST_PROD_INDEX_REGS  0x0300
686 #define SEND_RING_NIC_PROD_INDEX_REGS   0x0380

688 #define RECV_RING_CONS_INDEX_REG(ring)  (0x0280+8*(ring))
689 #define SEND_RING_HOST_INDEX_REG(ring)  (0x0300+8*(ring))
690 #define SEND_RING_NIC_INDEX_REG(ring)   (0x0380+8*(ring))

692 /*
693  * Ethernet MAC Mode Register
694  */
695 #define ETHERNET_MAC_MODE_REG           0x0400
696 #define ETHERNET_MODE_ENABLE_FHDE       0x00800000
697 #define ETHERNET_MODE_ENABLE_RDE        0x00400000
698 #define ETHERNET_MODE_ENABLE_TDE        0x00200000
```

```
699 #define ETHERNET_MODE_ENABLE_MIP        0x00100000
700 #define ETHERNET_MODE_ENABLE_ACPI       0x00080000
701 #define ETHERNET_MODE_ENABLE_MAGIC_PKT  0x00040000
702 #define ETHERNET_MODE_SEND_CFGS         0x00020000
703 #define ETHERNET_MODE_FLUSH_TX_STATS    0x00010000
704 #define ETHERNET_MODE_CLEAR_TX_STATS    0x00008000
705 #define ETHERNET_MODE_ENABLE_TX_STATS   0x00004000
706 #define ETHERNET_MODE_FLUSH_RX_STATS    0x00002000
707 #define ETHERNET_MODE_CLEAR_RX_STATS    0x00001000
708 #define ETHERNET_MODE_ENABLE_RX_STATS   0x00000800
709 #define ETHERNET_MODE_LINK_POLARITY     0x00000400
710 #define ETHERNET_MODE_MAX_DEFER         0x00000200
711 #define ETHERNET_MODE_ENABLE_TX_BURST   0x00000100
712 #define ETHERNET_MODE_TAGGED_MODE       0x00000080
713 #define ETHERNET_MODE_MAC_LOOPBACK      0x00000010
714 #define ETHERNET_MODE_PORTMODE_MASK     0x0000000c
715 #define ETHERNET_MODE_PORTMODE_TBI      0x0000000c
716 #define ETHERNET_MODE_PORTMODE_GMII     0x00000008
717 #define ETHERNET_MODE_PORTMODE_MII      0x00000004
718 #define ETHERNET_MODE_PORTMODE_NONE     0x00000000
719 #define ETHERNET_MODE_HALF_DUPLEX       0x00000002
720 #define ETHERNET_MODE_GLOBAL_RESET      0x00000001

722 /*
723  * Ethernet MAC Status & Event Registers
724  */
725 #define ETHERNET_MAC_STATUS_REG         0x0404
726 #define ETHERNET_STATUS_MI_INT          0x00800000
727 #define ETHERNET_STATUS_MI_COMPLETE     0x00400000
728 #define ETHERNET_STATUS_LINK_CHANGED    0x00001000
729 #define ETHERNET_STATUS_PCS_ERROR       0x00000400
730 #define ETHERNET_STATUS_SYNC_CHANGED    0x00000010
731 #define ETHERNET_STATUS_CFG_CHANGED     0x00000008
732 #define ETHERNET_STATUS_RECEIVING_CFG   0x00000004
733 #define ETHERNET_STATUS_SIGNAL_DETECT   0x00000002
734 #define ETHERNET_STATUS_PCS_SYNCHED     0x00000001

736 #define ETHERNET_MAC_EVENT_ENABLE_REG   0x0408
737 #define ETHERNET_EVENT_MI_INT           0x00800000
738 #define ETHERNET_EVENT_LINK_INT         0x00001000
739 #define ETHERNET_STATUS_PCS_ERROR_INT   0x00000400

741 /*
742  * Ethernet MAC LED Control Register
743  *
744  * NOTE: PHY mode 1 *MUST* be selected; this is the hardware default and
745  * the external LED driver circuitry is wired up to assume that this mode
746  * will always be selected.  Software must not change it!
747  */
748 #define ETHERNET_MAC_LED_CONTROL_REG    0x040c
749 #define LED_CONTROL_OVERRIDE_BLINK      0x80000000
750 #define LED_CONTROL_BLINK_PERIOD_MASK   0x7ff80000
751 #define LED_CONTROL_LED_MODE_MASK       0x00001800
752 #define LED_CONTROL_LED_MODE_5700       0x00000000
753 #define LED_CONTROL_LED_MODE_PHY_1      0x00000800      /* mandatory    */
754 #define LED_CONTROL_LED_MODE_PHY_2      0x00001000
755 #define LED_CONTROL_LED_MODE_RESERVED   0x00001800
756 #define LED_CONTROL_TRAFFIC_LED_STATUS  0x00000400
757 #define LED_CONTROL_10MBPS_LED_STATUS   0x00000200
758 #define LED_CONTROL_100MBPS_LED_STATUS  0x00000100
759 #define LED_CONTROL_1000MBPS_LED_STATUS 0x00000080
760 #define LED_CONTROL_BLINK_TRAFFIC       0x00000040
761 #define LED_CONTROL_TRAFFIC_LED         0x00000020
762 #define LED_CONTROL_OVERRIDE_TRAFFIC    0x00000010
763 #define LED_CONTROL_10MBPS_LED          0x00000008
764 #define LED_CONTROL_100MBPS_LED         0x00000004
```



```
765 #define LED_CONTROL_1000MBPS_LED        0x00000002
766 #define LED_CONTROL_OVERRIDE_LINK       0x00000001
767 #define LED_CONTROL_DEFAULT             0x02000800

769 /*
770  * MAC Address registers
771  *
772  * These four eight-byte registers each hold one unicast address
773  * (six bytes), right justified & zero-filled on the left.
774  * They will normally all be set to the same value, as a station
775  * usually only has one h/w address.  The value in register 0 is
776  * used for pause packets; any of the four can be specified for
777  * substitution into other transmitted packets if required.
778  */
779 #define MAC_ADDRESS_0_REG               0x0410
780 #define MAC_ADDRESS_1_REG               0x0418
781 #define MAC_ADDRESS_2_REG               0x0420
782 #define MAC_ADDRESS_3_REG               0x0428

784 #define MAC_ADDRESS_REG(n)              (0x0410+8*(n))
785 #define MAC_ADDRESS_REGS_MAX            4

787 /*
788  * More MAC Registers ...
789  */
790 #define MAC_TX_RANDOM_BACKOFF_REG       0x0438
791 #define MAC_RX_MTU_SIZE_REG             0x043c
792 #define MAC_RX_MTU_DEFAULT              0x000005f2      /* 1522 */
793 #define MAC_TX_LENGTHS_REG              0x0464
794 #define MAC_TX_LENGTHS_DEFAULT          0x00002620
795 #define MAC_TX_LENGTHS_JMB_FRM_LEN_MSK   0x00ff0000
796 #define MAC_TX_LENGTHS_CNT_DWN_VAL_MSK   0xff000000

798 /*
799  * MII access registers
800  */
801 #define MI_COMMS_REG                    0x044c
802 #define MI_COMMS_START                  0x20000000
803 #define MI_COMMS_READ_FAILED            0x10000000
804 #define MI_COMMS_COMMAND_MASK           0x0c000000
805 #define MI_COMMS_COMMAND_READ           0x08000000
806 #define MI_COMMS_COMMAND_WRITE          0x04000000
807 #define MI_COMMS_ADDRESS_MASK           0x03e00000
808 #define MI_COMMS_ADDRESS_SHIFT          21
809 #define MI_COMMS_REGISTER_MASK          0x001f0000
810 #define MI_COMMS_REGISTER_SHIFT         16
811 #define MI_COMMS_DATA_MASK              0x0000ffff
812 #define MI_COMMS_DATA_SHIFT             0

814 #define MI_STATUS_REG                   0x0450
815 #define MI_STATUS_10MBPS                0x00000002
816 #define MI_STATUS_LINK                  0x00000001

818 #define MI_MODE_REG                     0x0454
819 #define MI_MODE_CLOCK_MASK              0x001f0000
820 #define MI_MODE_AUTOPOLL                0x00000010
821 #define MI_MODE_POLL_SHORT_PREAMBLE     0x00000002
822 #define MI_MODE_DEFAULT                 0x000c0000

824 #define MI_AUTOPOLL_STATUS_REG          0x0458
825 #define MI_AUTOPOLL_ERROR               0x00000001

827 #define TRANSMIT_MAC_STATUS_REG         0x0460
828 #define TRANSMIT_STATUS_ODI_OVERRUN     0x00000020
829 #define TRANSMIT_STATUS_ODI_UNDERRUN    0x00000010
830 #define TRANSMIT_STATUS_LINK_UP         0x00000008
```

```
831 #define TRANSMIT_STATUS_SENT_XON        0x00000004
832 #define TRANSMIT_STATUS_SENT_XOFF       0x00000002
833 #define TRANSMIT_STATUS_RCVD_XOFF       0x00000001

835 #define RECEIVE_MAC_STATUS_REG          0x046c
836 #define RECEIVE_STATUS_RCVD_XON         0x00000004
837 #define RECEIVE_STATUS_RCVD_XOFF        0x00000002
838 #define RECEIVE_STATUS_SENT_XOFF        0x00000001

840 /*
841  * These four-byte registers constitute a hash table for deciding
842  * whether to accept incoming multicast packets.  The bits are
843  * numbered in big-endian fashion, from hash 0 => the MSB of
844  * register 0 to hash 127 => the LSB of the highest-numbered
845  * register.
846  *
847  * NOTE: the 5704 can use a 256-bit table (registers 0-7) if
848  * enabled by setting the appropriate bit in the Rx MAC mode
849  * register.  Otherwise, and on all earlier chips, the table
850  * is only 128 bits (registers 0-3).
851  */
852 #define MAC_HASH_0_REG                  0x0470
853 #define MAC_HASH_1_REG                  0x0474
854 #define MAC_HASH_2_REG                  0x0478
855 #define MAC_HASH_3_REG                  0x047c
856 #define MAC_HASH_4_REG                  0x????
857 #define MAC_HASH_5_REG                  0x????
858 #define MAC_HASH_6_REG                  0x????
859 #define MAC_HASH_7_REG                  0x????
860 #define MAC_HASH_REG(n)                 (0x470+4*(n))

862 /*
863  * Receive Rules Registers: 16 pairs of control+mask/value pairs
864  */
865 #define RCV_RULES_CONTROL_0_REG         0x0480
866 #define RCV_RULES_MASK_0_REG            0x0484
867 #define RCV_RULES_CONTROL_15_REG        0x04f8
868 #define RCV_RULES_MASK_15_REG           0x04fc
869 #define RCV_RULES_CONFIG_REG            0x0500
870 #define RCV_RULES_CONFIG_DEFAULT        0x00000008

872 #define RECV_RULES_NUM_MAX              16
873 #define RECV_RULE_CONTROL_REG(rule)     (RCV_RULES_CONTROL_0_REG+8*(rule))
874 #define RECV_RULE_MASK_REG(rule)        (RCV_RULES_MASK_0_REG+8*(rule))

876 #define RECV_RULE_CTL_ENABLE            0x80000000
877 #define RECV_RULE_CTL_AND               0x40000000
878 #define RECV_RULE_CTL_P1                0x20000000
879 #define RECV_RULE_CTL_P2                0x10000000
880 #define RECV_RULE_CTL_P3                0x08000000
881 #define RECV_RULE_CTL_MASK              0x04000000
882 #define RECV_RULE_CTL_DISCARD           0x02000000
883 #define RECV_RULE_CTL_MAP               0x01000000
884 #define RECV_RULE_CTL_RESV_BITS         0x00fc0000
885 #define RECV_RULE_CTL_OP                0x00030000
886 #define RECV_RULE_CTL_OP_EQ             0x00000000
887 #define RECV_RULE_CTL_OP_NEQ            0x00010000
888 #define RECV_RULE_CTL_OP_GREAT          0x00020000
889 #define RECV_RULE_CTL_OP_LESS           0x00030000
890 #define RECV_RULE_CTL_HEADER            0x0000e000
891 #define RECV_RULE_CTL_HEADER_FRAME      0x00000000
892 #define RECV_RULE_CTL_HEADER_IP         0x00002000
893 #define RECV_RULE_CTL_HEADER_TCP        0x00004000
894 #define RECV_RULE_CTL_HEADER_UDP        0x00006000
895 #define RECV_RULE_CTL_HEADER_DATA       0x00008000
896 #define RECV_RULE_CTL_CLASS_BITS        0x00001f00
```



```
897 #define RECV_RULE_CTL_CLASS(ring)       (((ring) << 8) & \
898                                             RECV_RULE_CTL_CLASS_BITS)
899 #define RECV_RULE_CTL_OFFSET            0x000000ff

901 /*
902  * Receive Rules definition
903  */
904 #define ETHERHEADER_DEST_OFFSET         0x00
905 #define IPHEADER_PROTO_OFFSET           0x08
906 #define IPHEADER_SIP_OFFSET             0x0c
907 #define IPHEADER_DIP_OFFSET             0x10
908 #define TCPHEADER_SPORT_OFFSET          0x00
909 #define TCPHEADER_DPORT_OFFSET          0x02
910 #define UDPHEADER_SPORT_OFFSET          0x00
911 #define UDPHEADER_DPORT_OFFSET          0x02

913 #define RULE_MATCH(ring)        (RECV_RULE_CTL_ENABLE | RECV_RULE_CTL_OP_EQ | \
914                                     RECV_RULE_CTL_CLASS((ring)))

916 #define RULE_MATCH_MASK(ring)   (RULE_MATCH(ring) | RECV_RULE_CTL_MASK)

918 #define RULE_DEST_MAC_1(ring)   (RULE_MATCH(ring) | \
919                                     RECV_RULE_CTL_HEADER_FRAME | \
920                                     ETHERHEADER_DEST_OFFSET)

922 #define RULE_DEST_MAC_2(ring)   (RULE_MATCH_MASK(ring) | \
923                                     RECV_RULE_CTL_HEADER_FRAME | \
924                                     ETHERHEADER_DEST_OFFSET + 4)

926 #define RULE_LOCAL_IP(ring)     (RULE_MATCH(ring) | RECV_RULE_CTL_HEADER_IP | \
927                                     IPHEADER_DIP_OFFSET)

929 #define RULE_REMOTE_IP(ring)    (RULE_MATCH(ring) | RECV_RULE_CTL_HEADER_IP | \
930                                     IPHEADER_SIP_OFFSET)

932 #define RULE_IP_PROTO(ring)     (RULE_MATCH_MASK(ring) | \
933                                     RECV_RULE_CTL_HEADER_IP | \
934                                     IPHEADER_PROTO_OFFSET)

936 #define RULE_TCP_SPORT(ring)    (RULE_MATCH_MASK(ring) | \
937                                     RECV_RULE_CTL_HEADER_TCP | \
938                                     TCPHEADER_SPORT_OFFSET)

940 #define RULE_TCP_DPORT(ring)    (RULE_MATCH_MASK(ring) | \
941                                     RECV_RULE_CTL_HEADER_TCP | \
942                                     TCPHEADER_DPORT_OFFSET)

944 #define RULE_UDP_SPORT(ring)    (RULE_MATCH_MASK(ring) | \
945                                     RECV_RULE_CTL_HEADER_UDP | \
946                                     UDPHEADER_SPORT_OFFSET)

948 #define RULE_UDP_DPORT(ring)    (RULE_MATCH_MASK(ring) | \
949                                     RECV_RULE_CTL_HEADER_UDP | \
950                                     UDPHEADER_DPORT_OFFSET)

952 /*
953  * 1000BaseX low-level access registers
954  */
955 #define MAC_GIGABIT_PCS_TEST_REG        0x0440
956 #define MAC_GIGABIT_PCS_TEST_ENABLE     0x00100000
957 #define MAC_GIGABIT_PCS_TEST_PATTERN    0x000fffff
958 #define TX_1000BASEX_AUTONEG_REG        0x0444
959 #define RX_1000BASEX_AUTONEG_REG        0x0448

961 /*
962  * Autoneg code bits for the 1000BASE-X AUTONEG registers
```

```
 963  */
 964 #define AUTONEG_CODE_PAUSE              0x00008000
 965 #define AUTONEG_CODE_HALF_DUPLEX        0x00004000
 966 #define AUTONEG_CODE_FULL_DUPLEX        0x00002000
 967 #define AUTONEG_CODE_NEXT_PAGE          0x00000080
 968 #define AUTONEG_CODE_ACKNOWLEDGE        0x00000040
 969 #define AUTONEG_CODE_FAULT_MASK         0x00000030
 970 #define AUTONEG_CODE_FAULT_ANEG_ERR     0x00000030
 971 #define AUTONEG_CODE_FAULT_LINK_FAIL    0x00000020
 972 #define AUTONEG_CODE_FAULT_OFFLINE      0x00000010
 973 #define AUTONEG_CODE_ASYM_PAUSE         0x00000001

 975 /*
 976  * SerDes Registers (5703S/5704S only)
 977  */
 978 #define SERDES_CONTROL_REG              0x0590
 979 #define SERDES_CONTROL_TBI_LOOPBACK     0x00020000
 980 #define SERDES_CONTROL_COMMA_DETECT     0x00010000
 981 #define SERDES_CONTROL_TX_DISABLE       0x00004000
 982 #define SERDES_STATUS_REG               0x0594
 983 #define SERDES_STATUS_COMMA_DETECTED    0x00000100
 984 #define SERDES_STATUS_RXSTAT            0x000000ff

 986 /*
 987  * SGMII Status Register (5717/5718 only)
 988  */
 989 #define SGMII_STATUS_REG        0x5B4
 990 #define MEDIA_SELECTION_MODE    0x00000100

 992 /*
 993  * Statistic Registers (5705/5788/5721/5751/5752/5714/5715 only)
 994  */
 995 #define STAT_IFHCOUT_OCTETS_REG         0x0800
 996 #define STAT_ETHER_COLLIS_REG           0x0808
 997 #define STAT_OUTXON_SENT_REG            0x080c
 998 #define STAT_OUTXOFF_SENT_REG           0x0810
 999 #define STAT_DOT3_INTMACTX_ERR_REG              0x0818
1000 #define STAT_DOT3_SCOLLI_FRAME_REG              0x081c
1001 #define STAT_DOT3_MCOLLI_FRAME_REG              0x0820
1002 #define STAT_DOT3_DEFERED_TX_REG                0x0824
1003 #define STAT_DOT3_EXCE_COLLI_REG                0x082c
1004 #define STAT_DOT3_LATE_COLLI_REG                0x0830
1005 #define STAT_IFHCOUT_UPKGS_REG          0x086c
1006 #define STAT_IFHCOUT_MPKGS_REG          0x0870
1007 #define STAT_IFHCOUT_BPKGS_REG          0x0874

1009 #define STAT_IFHCIN_OCTETS_REG          0x0880
1010 #define STAT_ETHER_FRAGMENT_REG         0x0888
1011 #define STAT_IFHCIN_UPKGS_REG           0x088c
1012 #define STAT_IFHCIN_MPKGS_REG           0x0890
1013 #define STAT_IFHCIN_BPKGS_REG           0x0894

1015 #define STAT_DOT3_FCS_ERR_REG           0x0898
1016 #define STAT_DOT3_ALIGN_ERR_REG         0x089c
1017 #define STAT_XON_PAUSE_RX_REG           0x08a0
1018 #define STAT_XOFF_PAUSE_RX_REG          0x08a4
1019 #define STAT_MAC_CTRL_RX_REG            0x08a8
1020 #define STAT_XOFF_STATE_ENTER_REG               0x08ac
1021 #define STAT_DOT3_FRAME_TOOLONG_REG             0x08b0
1022 #define STAT_ETHER_JABBERS_REG          0x08b4
1023 #define STAT_ETHER_UNDERSIZE_REG                0x08b8
1024 #define SIZE_OF_STATISTIC_REG           0x1B
1025 /*
1026  * Send Data Initiator Registers
1027  */
1028 #define SEND_INIT_STATS_CONTROL_REG     0x0c08
```



```
1029 #define SEND_INIT_STATS_ZERO            0x00000010
1030 #define SEND_INIT_STATS_FLUSH           0x00000008
1031 #define SEND_INIT_STATS_CLEAR           0x00000004
1032 #define SEND_INIT_STATS_FASTER          0x00000002
1033 #define SEND_INIT_STATS_ENABLE          0x00000001

1035 #define SEND_INIT_STATS_ENABLE_MASK_REG 0x0c0c

1037 /*
1038  * Send Buffer Descriptor Selector Control Registers
1039  */
1040 #define SEND_BD_SELECTOR_STATUS_REG     0x1404
1041 #define SEND_BD_SELECTOR_HWDIAG_REG     0x1408
1042 #define SEND_BD_SELECTOR_INDEX_REG(n)   (0x1440+4*(n))

1044 /*
1045  * Receive List Placement Registers
1046  */
1047 #define RCV_LP_CONFIG_REG               0x2010
1048 #define RCV_LP_CONFIG_DEFAULT           0x00000009
1049 #define RCV_LP_CONFIG(rings)            (((rings) << 3) | 0x1)

1051 #define RCV_LP_STATS_CONTROL_REG        0x2014
1052 #define RCV_LP_STATS_ZERO               0x00000010
1053 #define RCV_LP_STATS_FLUSH              0x00000008
1054 #define RCV_LP_STATS_CLEAR              0x00000004
1055 #define RCV_LP_STATS_FASTER             0x00000002
1056 #define RCV_LP_STATS_ENABLE             0x00000001

1058 #define RCV_LP_STATS_ENABLE_MASK_REG    0x2018
1059 #define RCV_LP_STATS_DISABLE_MACTQ      0x040000

1061 /*
1062  * Receive Data & BD Initiator Registers
1063  */
1064 #define RCV_INITIATOR_STATUS_REG        0x2404

1066 /*
1067  * Receive Buffer Descriptor Ring Control Block Registers
1068  * NB: sixteen bytes (128 bits) each
1069  */
1070 #define JUMBO_RCV_BD_RING_RCB_REG       0x2440
1071 #define STD_RCV_BD_RING_RCB_REG         0x2450
1072 #define MINI_RCV_BD_RING_RCB_REG        0x2460

1074 /*
1075  * Receive Buffer Descriptor Ring Replenish Threshold Registers
1076  */
1077 #define MINI_RCV_BD_REPLENISH_REG       0x2c14
1078 #define MINI_RCV_BD_REPLENISH_DEFAULT   0x00000080      /* 128  */
1079 #define STD_RCV_BD_REPLENISH_REG        0x2c18
1080 #define STD_RCV_BD_REPLENISH_DEFAULT    0x00000002      /* 2    */
1081 #define JUMBO_RCV_BD_REPLENISH_REG      0x2c1c
1082 #define JUMBO_RCV_BD_REPLENISH_DEFAULT  0x00000020      /* 32   */

1084 /*
1085  * CPMU registers (5717/5718/5719/5720 only)
1084  * CPMU registers (5717/5718 only)
1086  */
1087 #define CPMU_CLCK_ORIDE_REG             0x3624
1088 #define CPMU_CLCK_ORIDE_MAC_ORIDE_EN    0x80000000

1090 #define CPMU_STATUS_REG                 0x362c
1091 #define CPMU_STATUS_FUN_NUM_5717        0x20000000
1092 #define CPMU_STATUS_FUN_NUM_5719        0xc0000000
1093 #define CPMU_STATUS_FUN_NUM_5719_SHIFT  30
```

```
1087 #define CPMU_STATUS_FUN_NUM     0x20000000


1096 /*
1097  * Host Coalescing Engine Control Registers
1098  */
1099 #define RCV_COALESCE_TICKS_REG          0x3c08
1100 #define RCV_COALESCE_TICKS_DEFAULT      0x00000096      /* 150  */
1101 #define SEND_COALESCE_TICKS_REG         0x3c0c
1102 #define SEND_COALESCE_TICKS_DEFAULT     0x00000096      /* 150  */
1103 #define RCV_COALESCE_MAX_BD_REG         0x3c10
1104 #define RCV_COALESCE_MAX_BD_DEFAULT     0x0000000a      /* 10   */
1105 #define SEND_COALESCE_MAX_BD_REG        0x3c14
1106 #define SEND_COALESCE_MAX_BD_DEFAULT    0x0000000a      /* 10   */
1107 #define RCV_COALESCE_INT_TICKS_REG      0x3c18
1108 #define RCV_COALESCE_INT_TICKS_DEFAULT  0x00000000      /* 0    */
1109 #define SEND_COALESCE_INT_TICKS_REG     0x3c1c
1110 #define SEND_COALESCE_INT_TICKS_DEFAULT 0x00000000      /* 0    */
1111 #define RCV_COALESCE_INT_BD_REG         0x3c20
1112 #define RCV_COALESCE_INT_BD_DEFAULT     0x00000000      /* 0    */
1113 #define SEND_COALESCE_INT_BD_REG        0x3c24
1114 #define SEND_COALESCE_INT_BD_DEFAULT    0x00000000      /* 0    */
1115 #define STATISTICS_TICKS_REG            0x3c28
1116 #define STATISTICS_TICKS_DEFAULT        0x000f4240      /* 1000000 */
1117 #define STATISTICS_HOST_ADDR_REG        0x3c30
1118 #define STATUS_BLOCK_HOST_ADDR_REG      0x3c38
1119 #define STATISTICS_BASE_ADDR_REG        0x3c40
1120 #define STATUS_BLOCK_BASE_ADDR_REG      0x3c44
1121 #define FLOW_ATTN_REG                   0x3c48

1123 #define NIC_JUMBO_RECV_INDEX_REG        0x3c50
1124 #define NIC_STD_RECV_INDEX_REG          0x3c54
1125 #define NIC_MINI_RECV_INDEX_REG         0x3c58
1126 #define NIC_DIAG_RETURN_INDEX_REG(n)    (0x3c80+4*(n))
1127 #define NIC_DIAG_SEND_INDEX_REG(n)      (0x3cc0+4*(n))

1129 /*
1130  * Mbuf Pool Initialisation & Watermark Registers
1131  *
1132  * There are some conflicts in the PRM; compare the recommendations
1133  * on pp. 115, 236, and 339.  The values here were recommended by
1134  * dkim@broadcom.com (and the PRM should be corrected soon ;-)
1135  */
1136 #define BUFFER_MANAGER_STATUS_REG       0x4404
1137 #define MBUF_POOL_BASE_REG              0x4408
1138 #define MBUF_POOL_BASE_DEFAULT          0x00008000
1139 #define MBUF_POOL_BASE_5721             0x00010000
1140 #define MBUF_POOL_BASE_5704             0x00010000
1141 #define MBUF_POOL_BASE_5705             0x00010000
1142 #define MBUF_POOL_LENGTH_REG            0x440c
1143 #define MBUF_POOL_LENGTH_DEFAULT        0x00018000
1144 #define MBUF_POOL_LENGTH_5704           0x00010000
1145 #define MBUF_POOL_LENGTH_5705           0x00008000
1146 #define MBUF_POOL_LENGTH_5721           0x00008000
1147 #define RDMA_MBUF_LOWAT_REG             0x4410
1148 #define RDMA_MBUF_LOWAT_DEFAULT         0x00000050
1149 #define RDMA_MBUF_LOWAT_5705            0x00000000
1150 #define RDMA_MBUF_LOWAT_5906            0x00000000
1151 #define RDMA_MBUF_LOWAT_JUMBO           0x00000130
1152 #define RDMA_MBUF_LOWAT_5714_JUMBO      0x00000000
1153 #define MAC_RX_MBUF_LOWAT_REG           0x4414
1154 #define MAC_RX_MBUF_LOWAT_DEFAULT       0x00000020
1155 #define MAC_RX_MBUF_LOWAT_5705          0x00000010
1156 #define MAC_RX_MBUF_LOWAT_5906          0x00000004
1157 #define MAC_RX_MBUF_LOWAT_5717          0x0000002a
1158 #define MAC_RX_MBUF_LOWAT_JUMBO         0x00000098
```



```
1159 #define MAC_RX_MBUF_LOWAT_5714_JUMBO    0x0000004b
1160 #define MBUF_HIWAT_REG                  0x4418
1161 #define MBUF_HIWAT_DEFAULT              0x00000060
1162 #define MBUF_HIWAT_5705                 0x00000060
1163 #define MBUF_HIWAT_5906                 0x00000010
1164 #define MBUF_HIWAT_5717                 0x000000a0
1165 #define MBUF_HIWAT_JUMBO                0x0000017c
1166 #define MBUF_HIWAT_5714_JUMBO           0x00000096

1168 /*
1169  * DMA Descriptor Pool Initialisation & Watermark Registers
1170  */
1171 #define DMAD_POOL_BASE_REG              0x442c
1172 #define DMAD_POOL_BASE_DEFAULT          0x00002000
1173 #define DMAD_POOL_LENGTH_REG            0x4430
1174 #define DMAD_POOL_LENGTH_DEFAULT        0x00002000
1175 #define DMAD_POOL_LOWAT_REG             0x4434
1176 #define DMAD_POOL_LOWAT_DEFAULT         0x00000005      /* 5    */
1177 #define DMAD_POOL_HIWAT_REG             0x4438
1178 #define DMAD_POOL_HIWAT_DEFAULT         0x0000000a      /* 10   */

1180 /*
1181  * More threshold/watermark registers ...
1182  */
1183 #define RECV_FLOW_THRESHOLD_REG         0x4458
1184 #define LOWAT_MAX_RECV_FRAMES_REG       0x0504
1185 #define LOWAT_MAX_RECV_FRAMES_DEFAULT   0x00000002

1187 /*
1188  * Read/Write DMA Status Registers
1189  */
1190 #define READ_DMA_STATUS_REG             0x4804
1191 #define WRITE_DMA_STATUS_REG            0x4c04

1193 /*
1194  * RX/TX RISC Registers
1195  */
1196 #define RX_RISC_MODE_REG                0x5000
1197 #define RX_RISC_STATE_REG               0x5004
1198 #define RX_RISC_PC_REG                  0x501c
1199 #define TX_RISC_MODE_REG                0x5400
1200 #define TX_RISC_STATE_REG               0x5404
1201 #define TX_RISC_PC_REG                  0x541c

1203 /*
1204  * V? RISC Registerss
1205  */
1206 #define VCPU_STATUS_REG                 0x5100
1207 #define VCPU_INIT_DONE                  0x04000000
1208 #define VCPU_DRV_RESET                  0x08000000

1210 #define VCPU_EXT_CTL                    0x6890
1211 #define VCPU_EXT_CTL_HALF               0x00400000

1213 #define GRC_FASTBOOT_PC                 0x6894

1215 #define FTQ_RESET_REG                   0x5c00

1217 #define MSI_MODE_REG                    0x6000
1218 #define MSI_PRI_HIGHEST                 0xc0000000
1219 #define MSI_MSI_ENABLE                  0x00000002
1220 #define MSI_ERROR_ATTENTION             0x0000001c

1222 #define MSI_STATUS_REG                  0x6004

1224 #define MODE_CONTROL_REG                0x6800
```

```
1225 #define MODE_ROUTE_MCAST_TO_RX_RISC     0x40000000
1226 #define MODE_4X_NIC_SEND_RINGS          0x20000000
1227 #define MODE_INT_ON_FLOW_ATTN           0x10000000
1228 #define MODE_INT_ON_DMA_ATTN            0x08000000
1229 #define MODE_INT_ON_MAC_ATTN            0x04000000
1230 #define MODE_INT_ON_RXRISC_ATTN         0x02000000
1231 #define MODE_INT_ON_TXRISC_ATTN         0x01000000
1232 #define MODE_RECV_NO_PSEUDO_HDR_CSUM    0x00800000
1233 #define MODE_SEND_NO_PSEUDO_HDR_CSUM    0x00100000
1234 #define MODE_HTX2B_ENABLE               0x00040000
1235 #define MODE_HOST_SEND_BDS              0x00020000
1236 #define MODE_HOST_STACK_UP              0x00010000
1237 #define MODE_FORCE_32_BIT_PCI           0x00008000
1238 #define MODE_B2HRX_ENABLE               0x00008000
1239 #define MODE_NO_INT_ON_RECV             0x00004000
1240 #define MODE_NO_INT_ON_SEND             0x00002000
1241 #define MODE_ALLOW_BAD_FRAMES           0x00000800
1242 #define MODE_NO_CRC                     0x00000400
1243 #define MODE_NO_FRAME_CRACKING          0x00000200
1244 #define MODE_WORD_SWAP_B2HRX_DATA       0x00000080
1245 #define MODE_BYTE_SWAP_B2HRX_DATA       0x00000040
1246 #define MODE_WORD_SWAP_FRAME            0x00000020
1247 #define MODE_BYTE_SWAP_FRAME            0x00000010
1248 #define MODE_WORD_SWAP_NONFRAME         0x00000004
1249 #define MODE_BYTE_SWAP_NONFRAME         0x00000002
1250 #define MODE_UPDATE_ON_COAL_ONLY        0x00000001

1252 /*
1253  * Miscellaneous Configuration Register
1254  *
1255  * This contains various bits relating to power control (which differ
1256  * among different members of the chip family), but the important bits
1257  * for our purposes are the RESET bit and the Timer Prescaler field.
1258  *
1259  * The RESET bit in this register serves to reset the whole chip, even
1260  * including the PCI interface(!)  Once it's set, the chip will not
1261  * respond to ANY accesses -- not even CONFIG space -- until the reset
1262  * completes internally.  According to the PRM, this should take less
1263  * than 100us.  Any access during this period will get a bus error.
1264  *
1265  * The Timer Prescaler field must be programmed so that the timer period
1266  * is as near as possible to 1us.  The value in this field should be
1267  * the Core Clock frequency in MHz minus 1.  From my reading of the PRM,
1268  * the Core Clock should always be 66MHz (independently of the bus speed,
1269  * at least for PCI rather than PCI-X), so this register must be set to
1270  * the value 0x82 ((66-1) << 1).
1271  */
1272 #define CORE_CLOCK_MHZ                  66
1273 #define MISC_CONFIG_REG                 0x6804
1274 #define MISC_CONFIG_GRC_RESET_DISABLE   0x20000000
1275 #define MISC_CONFIG_GPHY_POWERDOWN_OVERRIDE 0x04000000
1276 #define MISC_CONFIG_POWERDOWN           0x00100000
1277 #define MISC_CONFIG_POWER_STATE         0x00060000
1278 #define MISC_CONFIG_PRESCALE_MASK       0x000000fe
1279 #define MISC_CONFIG_RESET_BIT           0x00000001
1280 #define MISC_CONFIG_DEFAULT             (((CORE_CLOCK_MHZ)-1) << 1)
1281 #define MISC_CONFIG_EPHY_IDDQ           0x00200000

1283 /*
1284  * Miscellaneous Local Control Register (MLCR)
1285  */
1286 #define MISC_LOCAL_CONTROL_REG          0x6808
1287 #define MLCR_PCI_CTRL_SELECT            0x10000000
1288 #define MLCR_LEGACY_PCI_MODE            0x08000000
1289 #define MLCR_AUTO_SEEPROM_ACCESS        0x01000000
1290 #define MLCR_SSRAM_CYCLE_DESELECT       0x00800000
```



```
1291 #define MLCR_SSRAM_TYPE                 0x00400000
1292 #define MLCR_BANK_SELECT                0x00200000
1293 #define MLCR_SRAM_SIZE_MASK             0x001c0000
1294 #define MLCR_ENABLE_EXTERNAL_MEMORY     0x00020000

1296 #define MLCR_MISC_PINS_OUTPUT_2         0x00010000
1297 #define MLCR_MISC_PINS_OUTPUT_1         0x00008000
1298 #define MLCR_MISC_PINS_OUTPUT_0         0x00004000
1299 #define MLCR_MISC_PINS_OUTPUT_ENABLE_2  0x00002000
1300 #define MLCR_MISC_PINS_OUTPUT_ENABLE_1  0x00001000
1301 #define MLCR_MISC_PINS_OUTPUT_ENABLE_0  0x00000800
1302 #define MLCR_MISC_PINS_INPUT_2          0x00000400      /* R/O  */
1303 #define MLCR_MISC_PINS_INPUT_1          0x00000200      /* R/O  */
1304 #define MLCR_MISC_PINS_INPUT_0          0x00000100      /* R/O  */

1306 #define MLCR_INT_ON_ATTN                0x00000008      /* R/W  */
1307 #define MLCR_SET_INT                    0x00000004      /* W/O  */
1308 #define MLCR_CLR_INT                    0x00000002      /* W/O  */
1309 #define MLCR_INTA_STATE                 0x00000001      /* R/O  */

1311 /*
1312  * This value defines all GPIO bits as INPUTS, but sets their default
1313  * values as outputs to HIGH, on the assumption that external circuits
1314  * (if any) will probably be active-LOW with passive pullups.
1315  *
1316  * The Claymore blade uses GPIO1 to control writing to the SEEPROM in
1317  * just this fashion.  It has to be set as an OUTPUT and driven LOW to
1318  * enable writing.  Otherwise, the SEEPROM is protected.
1319  */
1320 #define MLCR_DEFAULT                    0x0101c000
1321 #define MLCR_DEFAULT_5714               0x1901c000
1322 #define MLCR_DEFAULT_5717               0x01000000

1324 /*
1325  * Serial EEPROM Data/Address Registers (auto-access mode)
1326  */
1327 #define SERIAL_EEPROM_DATA_REG          0x683c
1328 #define SERIAL_EEPROM_ADDRESS_REG       0x6838
1329 #define SEEPROM_ACCESS_READ             0x80000000
1330 #define SEEPROM_ACCESS_WRITE            0x00000000
1331 #define SEEPROM_ACCESS_COMPLETE         0x40000000
1332 #define SEEPROM_ACCESS_RESET            0x20000000
1333 #define SEEPROM_ACCESS_DEVID_MASK       0x1c000000
1334 #define SEEPROM_ACCESS_START            0x02000000
1335 #define SEEPROM_ACCESS_HALFCLOCK_MASK   0x01ff0000
1336 #define SEEPROM_ACCESS_ADDRESS_MASK     0x0000fffc

1338 #define SEEPROM_ACCESS_DEVID_SHIFT      26              /* bits */
1339 #define SEEPROM_ACCESS_HALFCLOCK_SHIFT  16              /* bits */
1340 #define SEEPROM_ACCESS_ADDRESS_SIZE     16              /* bits */

1342 #define SEEPROM_ACCESS_HALFCLOCK_340KHZ 0x0060          /* 340kHz */
1343 #define SEEPROM_ACCESS_INIT             0x20600000      /* reset+clock  */

1345 /*
1346  * "Linearised" address mask, treating multiple devices as consecutive
1347  */
1348 #define SEEPROM_DEV_AND_ADDR_MASK       0x0007fffc      /* 8x64k devices */

1350 /*
1351  * Non-Volatile Memory Interface Registers
1352  * Note: on chips that support the flash interface (5702+), flash is the
1353  * default and the legacy seeprom interface must be explicitly enabled
1354  * if required. On older chips (5700/01), SEEPROM is the default (and
1355  * only) non-volatile memory available, and these registers don't exist!
1356  */
```

```
#define NVM_FLASH_CMD_REG               0x7000
#define NVM_FLASH_CMD_LAST              0x00000100
#define NVM_FLASH_CMD_FIRST             0x00000080
#define NVM_FLASH_CMD_RD                0x00000000
#define NVM_FLASH_CMD_WR                0x00000020
#define NVM_FLASH_CMD_DOIT              0x00000010
#define NVM_FLASH_CMD_DONE              0x00000008

#define NVM_FLASH_WRITE_REG             0x7008
#define NVM_FLASH_READ_REG              0x7010

#define NVM_FLASH_ADDR_REG              0x700c
#define NVM_FLASH_ADDR_MASK             0x00fffffc

#define NVM_CONFIG1_REG                 0x7014
#define NVM_CFG1_LEGACY_SEEPROM_MODE    0x80000000
#define NVM_CFG1_SEE_CLK_DIV_MASK       0x003ff800
#define NVM_CFG1_SPI_CLK_DIV_MASK       0x00000780
#define NVM_CFG1_BUFFERED_MODE          0x00000002
#define NVM_CFG1_FLASH_MODE             0x00000001

#define NVM_SW_ARBITRATION_REG          0x7020
#define NVM_READ_REQ3                   0X00008000
#define NVM_READ_REQ2                   0X00004000
#define NVM_READ_REQ1                   0X00002000
#define NVM_READ_REQ0                   0X00001000
#define NVM_WON_REQ3                    0X00000800
#define NVM_WON_REQ2                    0X00000400
#define NVM_WON_REQ1                    0X00000200
#define NVM_WON_REQ0                    0X00000100
#define NVM_RESET_REQ3                  0X00000080
#define NVM_RESET_REQ2                  0X00000040
#define NVM_RESET_REQ1                  0X00000020
#define NVM_RESET_REQ0                  0X00000010
#define NVM_SET_REQ3                    0X00000008
#define NVM_SET_REQ2                    0X00000004
#define NVM_SET_REQ1                    0X00000002
#define NVM_SET_REQ0                    0X00000001

/*
 * NVM access register
 * Applicable to BCM5721,BCM5751,BCM5752,BCM5714
 * and BCM5715 only.
 */
#define NVM_ACCESS_REG                  0X7024
#define NVM_WRITE_ENABLE                0X00000002
#define NVM_ACCESS_ENABLE               0X00000001

/*
 * TLP Control Register
 * Applicable to BCM5721 and BCM5751 only
 */
#define TLP_CONTROL_REG                 0x7c00
#define TLP_DATA_FIFO_PROTECT           0x02000000

/*
 * PHY Test Control Register
 * Applicable to BCM5721 and BCM5751 only
 */
#define PHY_TEST_CTRL_REG               0x7e2c
#define PHY_PCIE_SCRAM_MODE             0x20
#define PHY_PCIE_LTASS_MODE             0x40

/*
 * The internal firmware expects a certain layout of the non-volatile
 * memory (if fitted), and will check for it during startup, and use the
```



```
 * contents to initialise various internal parameters if it looks good.
 *
 * The offsets and field definitions below refer to where to find some
 * important values, and how to interpret them ...
 */
#define NVMEM_DATA_MAC_ADDRESS          0x007c          /* 8 bytes      */
#define NVMEM_DATA_MAC_ADDRESS_5906     0x0010          /* 8 bytes      */

/*
 * Vendor-specific MII registers
 */
#define MII_EXT_CONTROL                 MII_VENDOR(0)
#define MII_EXT_STATUS                  MII_VENDOR(1)
#define MII_RCV_ERR_COUNT               MII_VENDOR(2)
#define MII_FALSE_CARR_COUNT            MII_VENDOR(3)
#define MII_RCV_NOT_OK_COUNT            MII_VENDOR(4)
#define MII_AUX_CONTROL                 MII_VENDOR(8)
#define MII_AUX_STATUS                  MII_VENDOR(9)
#define MII_INTR_STATUS                 MII_VENDOR(10)
#define MII_INTR_MASK                   MII_VENDOR(11)
#define MII_HCD_STATUS                  MII_VENDOR(13)

#define MII_MAXREG                      MII_VENDOR(15)  /* 31, 0x1f     */

/*
 * Bits in the MII_EXT_CONTROL register
 */
#define MII_EXT_CTRL_INTERFACE_TBI      0x8000
#define MII_EXT_CTRL_DISABLE_AUTO_MDIX  0x4000
#define MII_EXT_CTRL_DISABLE_TRANSMIT   0x2000
#define MII_EXT_CTRL_DISABLE_INTERRUPT  0x1000
#define MII_EXT_CTRL_FORCE_INTERRUPT    0x0800
#define MII_EXT_CTRL_BYPASS_4B5B        0x0400
#define MII_EXT_CTRL_BYPASS_SCRAMBLER   0x0200
#define MII_EXT_CTRL_BYPASS_MLT3        0x0100
#define MII_EXT_CTRL_BYPASS_RX_ALIGN    0x0080
#define MII_EXT_CTRL_RESET_SCRAMBLER    0x0040
#define MII_EXT_CTRL_LED_TRAFFIC_MODE   0x0020
#define MII_EXT_CTRL_FORCE_LEDS_ON      0x0010
#define MII_EXT_CTRL_FORCE_LEDS_OFF     0x0008
#define MII_EXT_CTRL_EXTEND_TX_IPG      0x0004
#define MII_EXT_CTRL_3LINK_LED_MODE     0x0002
#define MII_EXT_CTRL_FIFO_ELASTICITY    0x0001

/*
 * Bits in the MII_EXT_STATUS register
 */
#define MII_EXT_STAT_S3MII_FIFO_ERROR   0x8000
#define MII_EXT_STAT_WIRESPEED_DOWNGRADE 0x4000
#define MII_EXT_STAT_MDIX_STATE         0x2000
#define MII_EXT_STAT_INTERRUPT_STATUS   0x1000
#define MII_EXT_STAT_REMOTE_RCVR_STATUS 0x0800
#define MII_EXT_STAT_LOCAL_RDVR_STATUS  0x0400
#define MII_EXT_STAT_DESCRAMBLER_LOCKED 0x0200
#define MII_EXT_STAT_LINK_STATUS        0x0100
#define MII_EXT_STAT_CRC_ERROR          0x0080
#define MII_EXT_STAT_CARR_EXT_ERROR     0x0040
#define MII_EXT_STAT_BAD_SSD_ERROR      0x0020
#define MII_EXT_STAT_BAD_ESD_ERROR      0x0010
#define MII_EXT_STAT_RECEIVE_ERROR      0x0008
#define MII_EXT_STAT_TRANSMIT_ERROR     0x0004
#define MII_EXT_STAT_LOCK_ERROR         0x0002
#define MII_EXT_STAT_MLT3_CODE_ERROR    0x0001

/*
 * The AUX CONTROL register is seriously weird!
```

```
1489  *
1490  * It hides (up to) eight 'shadow' registers.  When writing, which one
1491  * of them is written is determined by the low-order bits of the data
1492  * written(!), but when reading, which one is read is determined by the
1493  * value previously written to (part of) one of the shadow registers!!!
1494  */

1496 /*
1497  * Shadow register numbers
1498  */
1499 #define MII_AUX_CTRL_NORMAL             0
1500 #define MII_AUX_CTRL_10BASE_T           1
1501 #define MII_AUX_CTRL_POWER              2
1502 #define MII_AUX_CTRL_TEST_1             4
1503 #define MII_AUX_CTRL_MISC               7

1505 /*
1506  * Selected bits in some of the shadow registers ...
1507  */
1508 #define MII_AUX_CTRL_NORM_EXT_LOOPBACK  0x8000
1509 #define MII_AUX_CTRL_NORM_LONG_PKTS     0x4000
1510 #define MII_AUX_CTRL_NORM_EDGE_CTRL     0x3000
1511 #define MII_AUX_CTRL_NORM_TX_MODE       0x0400
1512 #define MII_AUX_CTRL_NORM_CABLE_TEST    0x0008

1514 #define MII_AUX_CTRL_TEST_TX_HALF       0x0008

1516 #define MII_AUX_CTRL_MISC_WRITE_ENABLE  0x8000
1517 #define MII_AUX_CTRL_MISC_WIRE_SPEED    0x0010

1519 /*
1520  * Write this value to the AUX control register
1521  * to select which shadow register will be read
1522  */
1523 #define MII_AUX_CTRL_SHADOW_READ(x)     (((x) << 12) | MII_AUX_CTRL_MISC)

1525 /*
1526  * Bits in the MII_AUX_STATUS register
1527  */
1528 #define MII_AUX_STATUS_MODE_MASK        0x0700
1529 #define MII_AUX_STATUS_MODE_1000_F      0x0700
1530 #define MII_AUX_STATUS_MODE_1000_H      0x0600
1531 #define MII_AUX_STATUS_MODE_100_F       0x0500
1532 #define MII_AUX_STATUS_MODE_100_4       0x0400
1533 #define MII_AUX_STATUS_MODE_100_H       0x0300
1534 #define MII_AUX_STATUS_MODE_10_F        0x0200
1535 #define MII_AUX_STATUS_MODE_10_H        0x0100
1536 #define MII_AUX_STATUS_MODE_NONE        0x0000
1537 #define MII_AUX_STATUS_MODE_SHIFT       8

1539 #define MII_AUX_STATUS_PAR_FAULT        0x0080
1540 #define MII_AUX_STATUS_REM_FAULT        0x0040
1541 #define MII_AUX_STATUS_LP_ANEG_ABLE     0x0010
1542 #define MII_AUX_STATUS_LP_NP_ABLE       0x0008

1544 #define MII_AUX_STATUS_LINKUP           0x0004
1545 #define MII_AUX_STATUS_RX_PAUSE         0x0002
1546 #define MII_AUX_STATUS_TX_PAUSE         0x0001

1548 #define MII_AUX_STATUS_SPEED_IND_5906   0x0008
1549 #define MII_AUX_STATUS_NEG_ENABLED_5906         0x0002
1550 #define MII_AUX_STATUS_DUPLEX_IND_5906          0x0001

1552 /*
1553  * Bits in the MII_INTR_STATUS and MII_INTR_MASK registers
1554  */
```



```
1555 #define MII_INTR_RMT_RX_STATUS_CHANGE   0x0020
1556 #define MII_INTR_LCL_RX_STATUS_CHANGE   0x0010
1557 #define MII_INTR_LINK_DUPLEX_CHANGE     0x0008
1558 #define MII_INTR_LINK_SPEED_CHANGE      0x0004
1559 #define MII_INTR_LINK_STATUS_CHANGE     0x0002


1562 /*
1563  * Third section:
1564  *      Hardware-defined data structures
1565  *
1566  * Note that the chip is naturally BIG-endian, so, for a big-endian
1567  * host, the structures defined below match those described in the PRM.
1568  * For little-endian hosts, some structures have to be swapped around.
1569  */

1571 #if     !defined(_BIG_ENDIAN) && !defined(_LITTLE_ENDIAN)
1572 #error  Host endianness not defined
1573 #endif

1575 /*
1576  * Architectural constants: absolute maximum numbers of each type of ring
1577  */
1578 #ifdef BGE_EXT_MEM
1579 #define BGE_SEND_RINGS_MAX              16      /* only with ext mem    */
1580 #else
1581 #define BGE_SEND_RINGS_MAX              4
1582 #endif
1583 #define BGE_SEND_RINGS_MAX_5705         1
1584 #define BGE_RECV_RINGS_MAX              16
1585 #define BGE_RECV_RINGS_MAX_5705         1
1586 #define BGE_BUFF_RINGS_MAX              3       /* jumbo/std/mini (mini */
1587                                                 /* only with ext mem)   */

1589 #define BGE_SEND_SLOTS_MAX              512
1590 #define BGE_STD_SLOTS_MAX               512
1591 #define BGE_JUMBO_SLOTS_MAX             256
1592 #define BGE_MINI_SLOTS_MAX              1024
1593 #define BGE_RECV_SLOTS_MAX              2048
1594 #define BGE_RECV_SLOTS_5705             512
1595 #define BGE_RECV_SLOTS_5717             1024
1596 #define BGE_RECV_SLOTS_5782             512
1597 #define BGE_RECV_SLOTS_5721             512

1599 /*
1600  * Hardware-defined Ring Control Block
1601  */
1602 typedef struct {
1603         uint64_t        host_ring_addr;
1604 #ifdef  _BIG_ENDIAN
1605         uint16_t        max_len;
1606         uint16_t        flags;
1607         uint32_t        nic_ring_addr;
1608 #else
1609         uint32_t        nic_ring_addr;
1610         uint16_t        flags;
1611         uint16_t        max_len;
1612 #endif  /* _BIG_ENDIAN */
1613 } bge_rcb_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   40861 Fri Jan 11 13:34:55 2013
new/usr/src/uts/common/io/bge/bge_impl.h
Just the 5719/5720 changes
**********************************************************
```

```
   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) 2002, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 /*
  27  * Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
  28  */

  30 #ifndef _BGE_IMPL_H
  31 #define _BGE_IMPL_H


  34 #ifdef __cplusplus
  35 extern "C" {
  36 #endif

  38 #include <sys/types.h>
  39 #include <sys/stream.h>
  40 #include <sys/strsun.h>
  41 #include <sys/strsubr.h>
  42 #include <sys/stat.h>
  43 #include <sys/pci.h>
  44 #include <sys/note.h>
  45 #include <sys/modctl.h>
  46 #include <sys/crc32.h>
  47 #ifdef  __sparcv9
  48 #include <v9/sys/membar.h>
  49 #endif  /* __sparcv9 */
  50 #include <sys/kstat.h>
  51 #include <sys/ethernet.h>
  52 #include <sys/vlan.h>
  53 #include <sys/errno.h>
  54 #include <sys/dlpi.h>
  55 #include <sys/devops.h>
  56 #include <sys/debug.h>
  57 #include <sys/conf.h>

  59 #include <netinet/ip6.h>

  61 #include <inet/common.h>
```



```
  62 #include <inet/ip.h>
  63 #include <inet/mi.h>
  64 #include <inet/nd.h>
  65 #include <sys/pattr.h>

  67 #include <sys/disp.h>
  68 #include <sys/cmn_err.h>
  69 #include <sys/ddi.h>
  70 #include <sys/sunddi.h>

  72 #include <sys/ddifm.h>
  73 #include <sys/fm/protocol.h>
  74 #include <sys/fm/util.h>
  75 #include <sys/fm/io/ddi.h>

  77 #include <sys/mac_provider.h>
  78 #include <sys/mac_ether.h>

  80 #ifdef __amd64
  81 #include <sys/x86_archext.h>
  82 #endif

  84 /*
  85  * <sys/ethernet.h> *may* already have provided the typedef ether_addr_t;
  86  * but of course C doesn't provide a way to check this directly.  So here
  87  * we rely on the fact that the symbol ETHERTYPE_AT was added to the
  88  * header file (as a #define, which we *can* test for) at the same time
  89  * as the typedef for ether_addr_t ;-!
  90  */
  91 #ifndef ETHERTYPE_AT
  92 typedef uchar_t ether_addr_t[ETHERADDRL];
  93 #endif  /* ETHERTYPE_AT */

  95 /*
  96  * Reconfiguring the network devices requires the net_config privilege
  97  * in Solaris 10+.
  98  */
  99 extern int secpolicy_net_config(const cred_t *, boolean_t);

 101 #include <sys/netlb.h>                  /* originally from cassini      */
 102 #include <sys/miiregs.h>                /* by fjlite out of intel       */

 104 #include "bge.h"
 105 #include "bge_hw.h"

 107 /*
 108  * Compile-time feature switches ...
 109  */
 110 #define BGE_DO_PPIO             0       /* peek/poke ioctls             */
 111 #define BGE_RX_SOFTINT          0       /* softint per receive ring     */
 112 #define BGE_CHOOSE_SEND_METHOD  0       /* send by copying only         */

 114 /*
 115  * NOTES:
 116  *
 117  * #defines:
 118  *
 119  *      BGE_PCI_CONFIG_RNUMBER and BGE_PCI_OPREGS_RNUMBER are the
 120  *      register-set numbers to use for the config space registers
 121  *      and the operating registers respectively.  On an OBP-based
 122  *      machine, regset 0 refers to CONFIG space, and regset 1 will
 123  *      be the operating registers in MEMORY space.  If an expansion
 124  *      ROM is fitted, it may appear as a further register set.
 125  *
 126  *      BGE_DMA_MODE defines the mode (STREAMING/CONSISTENT) used
 127  *      for the data buffers.  The descriptors are always set up
```

```
 128  *      in CONSISTENT mode.
 129  *
 130  *      BGE_HEADROOM defines how much space we'll leave in allocated
 131  *      mblks before the first valid data byte.  This should be chosen
 132  *      to be 2 modulo 4, so that once the ethernet header (14 bytes)
 133  *      has been stripped off, the packet data will be 4-byte aligned.
 134  *      The remaining space can be used by upstream modules to prepend
 135  *      any headers required.
 136  */

 138 #define BGE_PCI_CONFIG_RNUMBER  0
 139 #define BGE_PCI_OPREGS_RNUMBER  1
 140 #define BGE_DMA_MODE            DDI_DMA_STREAMING
 141 #define BGE_HEADROOM            34

 143 /*
 144  *      BGE_HALFTICK is half the period of the cyclic callback (in
 145  *      nanoseconds), chosen so that 0.5s <= cyclic period <= 1s.
 146  *      Other time values are derived as odd multiples of this value
 147  *      so that there's little chance of ambiguity w.r.t. which tick
 148  *      a timeout expires on.
 149  *
 150  *      BGE_PHY_STABLE_TIME is the period for which the contents of the
 151  *      PHY's status register must remain unchanging before we accept
 152  *      that the link has come up.  [Sometimes the link comes up, only
 153  *      to go down again within a short time as the autonegotiation
 154  *      process cycles through various options before finding the best
 155  *      compatible mode.  We don't want to report repeated link up/down
 156  *      cycles, so we wait until we think it's stable.]
 157  *
 158  *      BGE_SERDES_STABLE_TIME is the analogous value for the SerDes
 159  *      interface.  It's much shorter, 'cos the SerDes doesn't show
 160  *      these effects as much as the copper PHY.
 161  *
 162  *      BGE_LINK_SETTLE_TIME is the period during which we regard link
 163  *      up/down cycles as an normal event after resetting/reprogramming
 164  *      the PHY.  During this time, link up/down messages are sent to
 165  *      the log only, not the console.  At any other time, link change
 166  *      events are regarded as unexpected and sent to both console & log.
 167  *
 168  *      These latter two values have no theoretical justification, but
 169  *      are derived from observations and heuristics - the values below
 170  *      just seem to work quite well.
 171  */

 173 #define BGE_HALFTICK            268435456LL             /* 2**28 ns!    */
 174 #define BGE_CYCLIC_PERIOD       (4*BGE_HALFTICK)        /*    ~1.0s     */
 175 #define BGE_SERDES_STABLE_TIME  (3*BGE_HALFTICK)        /*    ~0.8s     */
 176 #define BGE_PHY_STABLE_TIME     (11*BGE_HALFTICK)       /*    ~3.0s     */
 177 #define BGE_LINK_SETTLE_TIME    (111*BGE_HALFTICK)      /*   ~30.0s     */

 179 /*
 180  * Indices used to identify the different buffer rings internally
 181  */
 182 #define BGE_STD_BUFF_RING       0
 183 #define BGE_JUMBO_BUFF_RING     1
 184 #define BGE_MINI_BUFF_RING      2

 186 /*
 187  * Current implementation limits
 188  */
 189 #define BGE_BUFF_RINGS_USED     2               /* std & jumbo ring     */
 190                                                 /* for now              */
 191 #define BGE_RECV_RINGS_USED     16              /* up to 16 rtn rings   */
 192                                                 /* for now              */
 193 #define BGE_SEND_RINGS_USED     4               /* up to 4 tx rings     */
```



```
 194                                                 /* for now              */
 195 #define BGE_HASH_TABLE_SIZE     128             /* may be 256 later     */

 197 /*
 198  * Ring/buffer size parameters
 199  *
 200  * All of the (up to) 16 TX rings & and the corresponding buffers are the
 201  * same size.
 202  *
 203  * Each of the (up to) 3 receive producer (aka buffer) rings is a different
 204  * size and has different sized buffers associated with it too.
 205  *
 206  * The (up to) 16 receive return rings have no buffers associated with them.
 207  * The number of slots per receive return ring must be 2048 if the mini
 208  * ring is enabled, otherwise it may be 1024.  See Broadcom document
 209  * 570X-PG102-R page 56.
 210  *
 211  * Note: only the 5700 supported external memory (and therefore the mini
 212  * ring); the 5702/3/4 don't.  This driver doesn't support the original
 213  * 5700, so we won't ever use the mini ring capability.
 214  */

 216 #define BGE_SEND_RINGS_DEFAULT          1
 217 #define BGE_RECV_RINGS_DEFAULT          1

 219 #define BGE_SEND_BUFF_SIZE_DEFAULT      1536
 220 #define BGE_SEND_BUFF_SIZE_JUMBO        9022
 221 #define BGE_SEND_SLOTS_USED     512

 223 #define BGE_STD_BUFF_SIZE       1536            /* 0x600                */
 224 #define BGE_STD_SLOTS_USED      512

 226 #define BGE_JUMBO_BUFF_SIZE     9022            /* 9k                   */
 227 #define BGE_JUMBO_SLOTS_USED    256

 229 #define BGE_MINI_BUFF_SIZE      128             /* 64? 256?             */
 230 #define BGE_MINI_SLOTS_USED     0               /* must be 0; see above */

 232 #define BGE_RECV_BUFF_SIZE      0
 233 #if     BGE_MINI_SLOTS_USED > 0
 234 #define BGE_RECV_SLOTS_USED     2048            /* required             */
 235 #else
 236 #define BGE_RECV_SLOTS_USED     1024            /* could be 2048 anyway */
 237 #endif

 239 #define BGE_SEND_BUF_NUM        512
 240 #define BGE_SEND_BUF_ARRAY      16
 241 #define BGE_SEND_BUF_ARRAY_JUMBO        3
 242 #define BGE_SEND_BUF_MAX        (BGE_SEND_BUF_NUM*BGE_SEND_BUF_ARRAY)

 244 /*
 245  * PCI type. PCI-Express or PCI/PCIX
 246  */
 247 #define BGE_PCI         0
 248 #define BGE_PCI_E       1
 249 #define BGE_PCI_X       2

 251 /*
 252  * Statistic type. There are two type of statistic:
 253  * statistic block and statistic registers
 254  */
 255 #define BGE_STAT_BLK    1
 256 #define BGE_STAT_REG    2

 258 /*
 259  * MTU.for all chipsets ,the default is 1500 ,and some chipsets
```

```
 260  * support 9k jumbo frames size
 261  */
 262 #define BGE_DEFAULT_MTU         1500
 263 #define BGE_MAXIMUM_MTU         9000

 265 /*
 266  * Pad the h/w defined status block (which can be up to 80 bytes long)
 267  * to a power-of-two boundary
 268  */
 269 #define BGE_STATUS_PADDING      (128 - sizeof (bge_status_t))

 271 /*
 272  * On platforms which support DVMA, we can simply allocate one big piece
 273  * of memory for all the Tx buffers and another for the Rx buffers, and
 274  * then carve them up as required.  It doesn't matter if they aren't just
 275  * one physically contiguous piece each, because both the CPU *and* the
 276  * I/O device can see them *as though they were*.
 277  *
 278  * However, if only physically-addressed DMA is possible, this doesn't
 279  * work; we can't expect to get enough contiguously-addressed memory for
 280  * all the buffers of each type, so in this case we request a number of
 281  * smaller pieces, each still large enough for several buffers but small
 282  * enough to fit within "an I/O page" (e.g. 64K).
 283  *
 284  * The #define below specifies how many pieces of memory are to be used;
 285  * 16 has been shown to work on an i86pc architecture but this could be
 286  * different on other non-DVMA platforms ...
 287  */
 288 #ifdef  _DMA_USES_VIRTADDR
 289 #define BGE_SPLIT               1               /* no split required    */
 290 #else
 291 #if ((BGE_BUFF_RINGS_USED > 1) || (BGE_SEND_RINGS_USED > 1) || \
 292         (BGE_RECV_RINGS_USED > 1))
 293 #define BGE_SPLIT               128             /* split 128 ways       */
 294 #else
 295 #define BGE_SPLIT               16              /* split 16 ways        */
 296 #endif
 297 #endif  /* _DMA_USES_VIRTADDR */

 299 #define BGE_RECV_RINGS_SPLIT    (BGE_RECV_RINGS_MAX + 1)

 301 /*
 302  * STREAMS parameters
 303  */
 304 #define BGE_IDNUM               0               /* zero seems to work   */
 305 #define BGE_LOWAT               (256)
 306 #define BGE_HIWAT               (256*1024)


 309 /*
 310  * Basic data types, for clarity in distinguishing 'numbers'
 311  * used for different purposes ...
 312  *
 313  * A <bge_regno_t> is a register 'address' (offset) in any one of
 314  * various address spaces (PCI config space, PCI memory-mapped I/O
 315  * register space, MII registers, etc).  None of these exceeds 64K,
 316  * so we could use a 16-bit representation but pointer-sized objects
 317  * are more "natural" in most architectures; they seem to be handled
 318  * more efficiently on SPARC and no worse on x86.
 319  *
 320  * BGE_REGNO_NONE represents the non-existent value in this space.
 321  */
 322 typedef uintptr_t bge_regno_t;                  /* register # (offset)  */
 323 #define BGE_REGNO_NONE          (~(uintptr_t)0u)

 325 /*
```



```
 326  * Describes one chunk of allocated DMA-able memory
 327  *
 328  * In some cases, this is a single chunk as allocated from the system;
 329  * but we also use this structure to represent slices carved off such
 330  * a chunk.  Even when we don't really need all the information, we
 331  * use this structure as a convenient way of correlating the various
 332  * ways of looking at a piece of memory (kernel VA, IO space DVMA,
 333  * handle+offset, etc).
 334  */
 335 typedef struct {
 336         ddi_acc_handle_t        acc_hdl;        /* handle for memory    */
 337         void                    *mem_va;        /* CPU VA of memory     */
 338         uint32_t                nslots;         /* number of slots      */
 339         uint32_t                size;           /* size per slot        */
 340         size_t                  alength;        /* allocated size       */
 341                                                 /* >= product of above  */

 343         ddi_dma_handle_t        dma_hdl;        /* DMA handle           */
 344         offset_t                offset;         /* relative to handle   */
 345         ddi_dma_cookie_t        cookie;         /* associated cookie    */
 346         uint32_t                ncookies;       /* must be 1            */
 347         uint32_t                token;          /* arbitrary identifier */
 348 } dma_area_t;                                   /* 0x50 (80) bytes      */
_____unchanged_portion_omitted_

 590 /*
 591  * Describes the characteristics of a specific chip
 592  *
 593  * Note: elements from <businfo> to <latency> are filled in by during
 594  * the first phase of chip initialisation (see bge_chip_cfg_init()).
 595  * The remaining ones are determined just after the first RESET, in
 596  * bge_poll_firmware().  Thereafter, the entire structure is readonly.
 597  */
 598 typedef struct {
 599         uint32_t                asic_rev;       /* masked from MHCR     */
 600         uint32_t                businfo;        /* from private reg     */
 601         uint16_t                command;        /* saved during attach  */

 603         uint16_t                vendor;         /* vendor-id            */
 604         uint16_t                device;         /* device-id            */
 605         uint16_t                subven;         /* subsystem-vendor-id  */
 606         uint16_t                subdev;         /* subsystem-id         */
 607         uint8_t                 revision;       /* revision-id          */
 608         uint8_t                 clsize;         /* cache-line-size      */
 609         uint8_t                 latency;        /* latency-timer        */

 611         uint8_t                 flags;
 612         uint32_t                chip_type;      /* see CHIP_TYPE_ in bge_hw.h */
 613         uint16_t                chip_label;     /* numeric part only    */
 614                                                 /* (e.g. 5703/5794/etc) */
 615         uint32_t                mbuf_base;      /* Mbuf pool parameters */
 616         uint32_t                mbuf_length;    /* depend on chiptype   */
 617         uint32_t                pci_type;
 618         uint32_t                statistic_type;
 619         uint32_t                bge_dma_rwctrl;
 620         uint32_t                bge_mlcr_default;
 621         uint32_t                recv_slots;     /* receive ring size    */
 622         enum bge_nvmem_type     nvtype;         /* SEEPROM or Flash     */

 624         uint16_t                jumbo_slots;
 625         uint16_t                ethmax_size;
 626         uint16_t                snd_buff_size;
 627         uint16_t                recv_jumbo_size;
 628         uint16_t                std_buf_size;
 629         uint32_t                mbuf_hi_water;
 630         uint32_t                mbuf_lo_water_rmac;
```

```
631         uint32_t                mbuf_lo_water_rdma;

633         uint32_t                rx_rings;       /* from bge.conf        */
634         uint32_t                tx_rings;       /* from bge.conf        */
635         uint32_t                default_mtu;    /* from bge.conf        */

637         uint64_t                hw_mac_addr;    /* from chip register   */
638         bge_mac_addr_t          vendor_addr;    /* transform of same    */
639         boolean_t               msi_enabled;    /* default to true */

641         uint32_t                rx_ticks_norm;
642         uint32_t                rx_count_norm;
643         uint32_t                tx_ticks_norm;
644         uint32_t                tx_count_norm;
645         uint32_t                mask_pci_int;
646 } chip_id_t;

648 #define CHIP_FLAG_SUPPORTED      0x80
649 #define CHIP_FLAG_SERDES         0x40
650 #define CHIP_FLAG_PARTIAL_CSUM   0x20
651 #define CHIP_FLAG_NO_CHECK_RESET 0x2
652 #define CHIP_FLAG_NO_JUMBO       0x1

654 /*
655  * Collection of physical-layer functions to:
656  *      (re)initialise the physical layer
657  *      update it to match software settings
658  *      check for link status change
659  */
660 typedef struct {
661         int                     (*phys_restart)(struct bge *, boolean_t);
662         int                     (*phys_update)(struct bge *);
663         boolean_t               (*phys_check)(struct bge *, boolean_t);
664 } phys_ops_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
  102491 Fri Jan 11 13:34:56 2013
new/usr/src/uts/common/io/bge/bge_main2.c
Just the 5719/5720 changes
**********************************************************
```

```
/*
 * 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) 2002, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
 */

#include "bge_impl.h"
#include <sys/sdt.h>
#include <sys/mac_provider.h>
#include <sys/mac.h>
#include <sys/mac_flow.h>

/*
 * This is the string displayed by modinfo, etc.
 */
static char bge_ident[] = "Broadcom Gb Ethernet";

/*
 * Property names
 */
static char debug_propname[] = "bge-debug-flags";
static char clsize_propname[] = "cache-line-size";
static char latency_propname[] = "latency-timer";
static char localmac_boolname[] = "local-mac-address?";
static char localmac_propname[] = "local-mac-address";
static char macaddr_propname[] = "mac-address";
static char subdev_propname[] = "subsystem-id";
static char subven_propname[] = "subsystem-vendor-id";
static char rxrings_propname[] = "bge-rx-rings";
static char txrings_propname[] = "bge-tx-rings";
static char fm_cap[] = "fm-capable";
static char default_mtu[] = "default_mtu";

static int bge_add_intrs(bge_t *, int);
static void bge_rem_intrs(bge_t *);
static int bge_unicst_set(void *, const uint8_t *, int);

/*
```



```
 * Describes the chip's DMA engine
 */
static ddi_dma_attr_t dma_attr = {
        DMA_ATTR_V0,                    /* dma_attr version     */
        0x0000000000000000ull,          /* dma_attr_addr_lo     */
        0xFFFFFFFFFFFFFFFFull,          /* dma_attr_addr_hi     */
        0x00000000FFFFFFFFull,          /* dma_attr_count_max   */
        0x0000000000000001ull,          /* dma_attr_align       */
        0x00000FFF,                     /* dma_attr_burstsizes  */
        0x00000001,                     /* dma_attr_minxfer     */
        0x000000000000FFFFull,          /* dma_attr_maxxfer     */
        0xFFFFFFFFFFFFFFFFull,          /* dma_attr_seg         */
        1,                              /* dma_attr_sgllen      */
        0x00000001,                     /* dma_attr_granular    */
        DDI_DMA_FLAGERR                 /* dma_attr_flags */
};
_____unchanged_portion_omitted_

/*
 * attach(9E) -- Attach a device to the system
 *
 * Called once for each board successfully probed.
 */
static int
bge_attach(dev_info_t *devinfo, ddi_attach_cmd_t cmd)
{
        bge_t *bgep;                            /* Our private data     */
        mac_register_t *macp;
        chip_id_t *cidp;
        caddr_t regs;
        int instance;
        int err;
        int intr_types;
#ifdef BGE_IPMI_ASF
        uint32_t mhcrValue;
#ifdef __sparc
        uint16_t value16;
#endif
#ifdef BGE_NETCONSOLE
        int retval;
#endif
#endif

        instance = ddi_get_instance(devinfo);

        BGE_GTRACE(("bge_attach($%p, %d) instance %d",
            (void *)devinfo, cmd, instance));
        BGE_BRKPT(NULL, "bge_attach");

        switch (cmd) {
        default:
                return (DDI_FAILURE);

        case DDI_RESUME:
                return (bge_resume(devinfo));

        case DDI_ATTACH:
                break;
        }

        bgep = kmem_zalloc(sizeof (*bgep), KM_SLEEP);
        bgep->pstats = kmem_zalloc(sizeof (bge_statistics_reg_t), KM_SLEEP);
        ddi_set_driver_private(devinfo, bgep);
        bgep->bge_guard = BGE_GUARD;
        bgep->devinfo = devinfo;
        bgep->param_drain_max = 64;
```

```
3148         bgep->param_msi_cnt = 0;
3149         bgep->param_loop_mode = 0;
3150
3151         /*
3152          * Initialize more fields in BGE private data
3153          */
3154         bgep->debug = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3155             DDI_PROP_DONTPASS, debug_propname, bge_debug);
3156         (void) snprintf(bgep->ifname, sizeof (bgep->ifname), "%s%d",
3157             BGE_DRIVER_NAME, instance);
3158
3159         /*
3160          * Initialize for fma support
3161          */
3162         bgep->fm_capabilities = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3163             DDI_PROP_DONTPASS, fm_cap,
3164             DDI_FM_EREPORT_CAPABLE | DDI_FM_ACCCHK_CAPABLE |
3165             DDI_FM_DMACHK_CAPABLE | DDI_FM_ERRCB_CAPABLE);
3166         BGE_DEBUG(("bgep->fm_capabilities = %d", bgep->fm_capabilities));
3167         bge_fm_init(bgep);
3168
3169         /*
3170          * Look up the IOMMU's page size for DVMA mappings (must be
3171          * a power of 2) and convert to a mask.  This can be used to
3172          * determine whether a message buffer crosses a page boundary.
3173          * Note: in 2s complement binary notation, if X is a power of
3174          * 2, then -X has the representation "11...1100...00".
3175          */
3176         bgep->pagemask = dvma_pagesize(devinfo);
3177         ASSERT(ddi_ffs(bgep->pagemask) == ddi_fls(bgep->pagemask));
3178         bgep->pagemask = -bgep->pagemask;
3179
3180         /*
3181          * Map config space registers
3182          * Read chip ID & set up config space command register(s)
3183          *
3184          * Note: this leaves the chip accessible by Memory Space
3185          * accesses, but with interrupts and Bus Mastering off.
3186          * This should ensure that nothing untoward will happen
3187          * if it has been left active by the (net-)bootloader.
3188          * We'll re-enable Bus Mastering once we've reset the chip,
3189          * and allow interrupts only when everything else is set up.
3190          */
3191         err = pci_config_setup(devinfo, &bgep->cfg_handle);
3192 #ifdef BGE_IPMI_ASF
3193 #ifdef __sparc
3194         /*
3195          * We need to determine the type of chipset for accessing some configure
3196          * registers. (This information will be used by bge_ind_put32,
3197          * bge_ind_get32 and bge_nic_read32)
3198          */
3199         bgep->chipid.device = pci_config_get16(bgep->cfg_handle,
3200             PCI_CONF_DEVID);
3201         value16 = pci_config_get16(bgep->cfg_handle, PCI_CONF_COMM);
3202         value16 = value16 | (PCI_COMM_MAE | PCI_COMM_ME);
3203         pci_config_put16(bgep->cfg_handle, PCI_CONF_COMM, value16);
3204         mhcrValue = MHCR_ENABLE_INDIRECT_ACCESS |
3205             MHCR_ENABLE_TAGGED_STATUS_MODE |
3206             MHCR_MASK_INTERRUPT_MODE |
3207             MHCR_MASK_PCI_INT_OUTPUT |
3208             MHCR_CLEAR_INTERRUPT_INTA |
3209             MHCR_ENABLE_ENDIAN_WORD_SWAP |
3210             MHCR_ENABLE_ENDIAN_BYTE_SWAP;
3211         /*
3212          * For some chipsets (e.g., BCM5718), if MHCR_ENABLE_ENDIAN_BYTE_SWAP
3213          * has been set in PCI_CONF_COMM already, we need to write the
```



```
3214          * byte-swapped value to it. So we just write zero first for simplicity.
3215          */
3216         if (DEVICE_5717_SERIES_CHIPSETS(bgep))
3217                 pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR, 0);
3218 #else
3219         mhcrValue = MHCR_ENABLE_INDIRECT_ACCESS |
3220             MHCR_ENABLE_TAGGED_STATUS_MODE |
3221             MHCR_MASK_INTERRUPT_MODE |
3222             MHCR_MASK_PCI_INT_OUTPUT |
3223             MHCR_CLEAR_INTERRUPT_INTA;
3224 #endif
3225         pci_config_put32(bgep->cfg_handle, PCI_CONF_BGE_MHCR, mhcrValue);
3226         bge_ind_put32(bgep, MEMORY_ARBITER_MODE_REG,
3227             bge_ind_get32(bgep, MEMORY_ARBITER_MODE_REG) |
3228             MEMORY_ARBITER_ENABLE);
3218 #else
3219         mhcrValue = pci_config_get32(bgep->cfg_handle, PCI_CONF_BGE_MHCR);
3220 #endif
3229         if (mhcrValue & MHCR_ENABLE_ENDIAN_WORD_SWAP) {
3230                 bgep->asf_wordswapped = B_TRUE;
3231         } else {
3232                 bgep->asf_wordswapped = B_FALSE;
3233         }
3234         bge_asf_get_config(bgep);
3235 #endif
3236         if (err != DDI_SUCCESS) {
3237                 bge_problem(bgep, "pci_config_setup() failed");
3238                 goto attach_fail;
3239         }
3240         bgep->progress |= PROGRESS_CFG;
3241         cidp = &bgep->chipid;
3242         bzero(cidp, sizeof (*cidp));
3243         bge_chip_cfg_init(bgep, cidp, B_FALSE);
3244         if (bge_check_acc_handle(bgep, bgep->cfg_handle) != DDI_FM_OK) {
3245                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3246                 goto attach_fail;
3247         }
3248
3249 #ifdef BGE_IPMI_ASF
3250         if (DEVICE_5721_SERIES_CHIPSETS(bgep) ||
3251             DEVICE_5714_SERIES_CHIPSETS(bgep)) {
3252                 bgep->asf_newhandshake = B_TRUE;
3253         } else {
3254                 bgep->asf_newhandshake = B_FALSE;
3255         }
3256 #endif
3257
3258         /*
3259          * Update those parts of the chip ID derived from volatile
3260          * registers with the values seen by OBP (in case the chip
3261          * has been reset externally and therefore lost them).
3262          */
3263         cidp->subven = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3264             DDI_PROP_DONTPASS, subven_propname, cidp->subven);
3265         cidp->subdev = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3266             DDI_PROP_DONTPASS, subdev_propname, cidp->subdev);
3267         cidp->clsize = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3268             DDI_PROP_DONTPASS, clsize_propname, cidp->clsize);
3269         cidp->latency = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3270             DDI_PROP_DONTPASS, latency_propname, cidp->latency);
3271         cidp->rx_rings = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3272             DDI_PROP_DONTPASS, rxrings_propname, cidp->rx_rings);
3273         cidp->tx_rings = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
3274             DDI_PROP_DONTPASS, txrings_propname, cidp->tx_rings);
3275
3276         cidp->default_mtu = ddi_prop_get_int(DDI_DEV_T_ANY, devinfo,
```

```
3277             DDI_PROP_DONTPASS, default_mtu, BGE_DEFAULT_MTU);
3278         if ((cidp->default_mtu < BGE_DEFAULT_MTU) ||
3279             (cidp->default_mtu > BGE_MAXIMUM_MTU)) {
3280                 cidp->default_mtu = BGE_DEFAULT_MTU;
3281         }

3283         /*
3284          * Map operating registers
3285          */
3286         err = ddi_regs_map_setup(devinfo, BGE_PCI_OPREGS_RNUMBER,
3287             &regs, 0, 0, &bge_reg_accattr, &bgep->io_handle);
3288         if (err != DDI_SUCCESS) {
3289                 bge_problem(bgep, "ddi_regs_map_setup() failed");
3290                 goto attach_fail;
3291         }
3292         bgep->io_regs = regs;
3293         bgep->progress |= PROGRESS_REGS;

3295         /*
3296          * Characterise the device, so we know its requirements.
3297          * Then allocate the appropriate TX and RX descriptors & buffers.
3298          */
3299         if (bge_chip_id_init(bgep) == EIO) {
3300                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3301                 goto attach_fail;
3302         }

3304         err = bge_alloc_bufs(bgep);
3305         if (err != DDI_SUCCESS) {
3306                 bge_problem(bgep, "DMA buffer allocation failed");
3307                 goto attach_fail;
3308         }
3309         bgep->progress |= PROGRESS_BUFS;

3311         /*
3312          * Add the softint handlers:
3313          *
3314          * Both of these handlers are used to avoid restrictions on the
3315          * context and/or mutexes required for some operations.  In
3316          * particular, the hardware interrupt handler and its subfunctions
3317          * can detect a number of conditions that we don't want to handle
3318          * in that context or with that set of mutexes held.  So, these
3319          * softints are triggered instead:
3320          *
3321          * the <resched> softint is triggered if we have previously
3322          * had to refuse to send a packet because of resource shortage
3323          * (we've run out of transmit buffers), but the send completion
3324          * interrupt handler has now detected that more buffers have
3325          * become available.
3326          *
3327          * the <factotum> is triggered if the h/w interrupt handler
3328          * sees the <link state changed> or <error> bits in the status
3329          * block.  It's also triggered periodically to poll the link
3330          * state, just in case we aren't getting link status change
3331          * interrupts ...
3332          */
3333         err = ddi_add_softintr(devinfo, DDI_SOFTINT_LOW, &bgep->drain_id,
3334             NULL, NULL, bge_send_drain, (caddr_t)bgep);
3335         if (err != DDI_SUCCESS) {
3336                 bge_problem(bgep, "ddi_add_softintr() failed");
3337                 goto attach_fail;
3338         }
3339         bgep->progress |= PROGRESS_RESCHED;
3340         err = ddi_add_softintr(devinfo, DDI_SOFTINT_LOW, &bgep->factotum_id,
3341             NULL, NULL, bge_chip_factotum, (caddr_t)bgep);
3342         if (err != DDI_SUCCESS) {
```



```
3343                 bge_problem(bgep, "ddi_add_softintr() failed");
3344                 goto attach_fail;
3345         }
3346         bgep->progress |= PROGRESS_FACTOTUM;

3348         /* Get supported interrupt types */
3349         if (ddi_intr_get_supported_types(devinfo, &intr_types) != DDI_SUCCESS) {
3350                 bge_error(bgep, "ddi_intr_get_supported_types failed\n");

3352                 goto attach_fail;
3353         }

3355         BGE_DEBUG(("%s: ddi_intr_get_supported_types() returned: %x",
3356             bgep->ifname, intr_types));

3358         if ((intr_types & DDI_INTR_TYPE_MSI) && bgep->chipid.msi_enabled) {
3359                 if (bge_add_intrs(bgep, DDI_INTR_TYPE_MSI) != DDI_SUCCESS) {
3360                         bge_error(bgep, "MSI registration failed, "
3361                             "trying FIXED interrupt type\n");
3362                 } else {
3363                         BGE_DEBUG(("%s: Using MSI interrupt type",
3364                             bgep->ifname));
3365                         bgep->intr_type = DDI_INTR_TYPE_MSI;
3366                         bgep->progress |= PROGRESS_HWINT;
3367                 }
3368         }

3370         if (!(bgep->progress & PROGRESS_HWINT) &&
3371             (intr_types & DDI_INTR_TYPE_FIXED)) {
3372                 if (bge_add_intrs(bgep, DDI_INTR_TYPE_FIXED) != DDI_SUCCESS) {
3373                         bge_error(bgep, "FIXED interrupt "
3374                             "registration failed\n");
3375                         goto attach_fail;
3376                 }

3378                 BGE_DEBUG(("%s: Using FIXED interrupt type", bgep->ifname));

3380                 bgep->intr_type = DDI_INTR_TYPE_FIXED;
3381                 bgep->progress |= PROGRESS_HWINT;
3382         }

3384         if (!(bgep->progress & PROGRESS_HWINT)) {
3385                 bge_error(bgep, "No interrupts registered\n");
3386                 goto attach_fail;
3387         }

3389         /*
3390          * Note that interrupts are not enabled yet as
3391          * mutex locks are not initialized. Initialize mutex locks.
3392          */
3393         mutex_init(bgep->genlock, NULL, MUTEX_DRIVER,
3394             DDI_INTR_PRI(bgep->intr_pri));
3395         mutex_init(bgep->softintrlock, NULL, MUTEX_DRIVER,
3396             DDI_INTR_PRI(bgep->intr_pri));
3397         rw_init(bgep->errlock, NULL, RW_DRIVER,
3398             DDI_INTR_PRI(bgep->intr_pri));

3400         /*
3401          * Initialize rings.
3402          */
3403         bge_init_rings(bgep);

3405         /*
3406          * Now that mutex locks are initialized, enable interrupts.
3407          */
3408         bge_intr_enable(bgep);
```

```
3409         bgep->progress |= PROGRESS_INTR;

3411         /*
3412          * Initialise link state variables
3413          * Stop, reset & reinitialise the chip.
3414          * Initialise the (internal) PHY.
3415          */
3416         bgep->link_state = LINK_STATE_UNKNOWN;

3418         mutex_enter(bgep->genlock);

3420         /*
3421          * Reset chip & rings to initial state; also reset address
3422          * filtering, promiscuity, loopback mode.
3423          */
3424 #ifdef BGE_IPMI_ASF
3425 #ifdef BGE_NETCONSOLE
3426         if (bge_reset(bgep, ASF_MODE_INIT) != DDI_SUCCESS) {
3427 #else
3428         if (bge_reset(bgep, ASF_MODE_SHUTDOWN) != DDI_SUCCESS) {
3429 #endif
3430 #else
3431         if (bge_reset(bgep) != DDI_SUCCESS) {
3432 #endif
3433                 (void) bge_check_acc_handle(bgep, bgep->cfg_handle);
3434                 (void) bge_check_acc_handle(bgep, bgep->io_handle);
3435                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3436                 mutex_exit(bgep->genlock);
3437                 goto attach_fail;
3438         }

3440 #ifdef BGE_IPMI_ASF
3441         if (bgep->asf_enabled) {
3442                 bgep->asf_status = ASF_STAT_RUN_INIT;
3443         }
3444 #endif

3446         bzero(bgep->mcast_hash, sizeof (bgep->mcast_hash));
3447         bzero(bgep->mcast_refs, sizeof (bgep->mcast_refs));
3448         bgep->promisc = B_FALSE;
3449         bgep->param_loop_mode = BGE_LOOP_NONE;
3450         if (bge_check_acc_handle(bgep, bgep->cfg_handle) != DDI_FM_OK) {
3451                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3452                 mutex_exit(bgep->genlock);
3453                 goto attach_fail;
3454         }
3455         if (bge_check_acc_handle(bgep, bgep->io_handle) != DDI_FM_OK) {
3456                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3457                 mutex_exit(bgep->genlock);
3458                 goto attach_fail;
3459         }

3461         mutex_exit(bgep->genlock);

3463         if (bge_phys_init(bgep) == EIO) {
3464                 ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
3465                 goto attach_fail;
3466         }
3467         bgep->progress |= PROGRESS_PHY;

3469         /*
3470          * initialize NDD-tweakable parameters
3471          */
3472         if (bge_nd_init(bgep)) {
3473                 bge_problem(bgep, "bge_nd_init() failed");
3474                 goto attach_fail;
```



```
3475         }
3476         bgep->progress |= PROGRESS_NDD;

3478         /*
3479          * Create & initialise named kstats
3480          */
3481         bge_init_kstats(bgep, instance);
3482         bgep->progress |= PROGRESS_KSTATS;

3484         /*
3485          * Determine whether to override the chip's own MAC address
3486          */
3487         bge_find_mac_address(bgep, cidp);

3489         bgep->unicst_addr_total = MAC_ADDRESS_REGS_MAX;
3490         bgep->unicst_addr_avail = MAC_ADDRESS_REGS_MAX;

3492         if ((macp = mac_alloc(MAC_VERSION)) == NULL)
3493                 goto attach_fail;
3494         macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
3495         macp->m_driver = bgep;
3496         macp->m_dip = devinfo;
3497         macp->m_src_addr = cidp->vendor_addr.addr;
3498         macp->m_callbacks = &bge_m_callbacks;
3499         macp->m_min_sdu = 0;
3500         macp->m_max_sdu = cidp->ethmax_size - sizeof (struct ether_header);
3501         macp->m_margin = VLAN_TAGSZ;
3502         macp->m_priv_props = bge_priv_prop;
3503         macp->m_v12n = MAC_VIRT_LEVEL1;

3505         /*
3506          * Finally, we're ready to register ourselves with the MAC layer
3507          * interface; if this succeeds, we're all ready to start()
3508          */
3509         err = mac_register(macp, &bgep->mh);
3510         mac_free(macp);
3511         if (err != 0)
3512                 goto attach_fail;

3514         mac_link_update(bgep->mh, LINK_STATE_UNKNOWN);

3516         /*
3517          * Register a periodical handler.
3518          * bge_chip_cyclic() is invoked in kernel context.
3519          */
3520         bgep->periodic_id = ddi_periodic_add(bge_chip_cyclic, bgep,
3521             BGE_CYCLIC_PERIOD, DDI_IPL_0);

3523         bgep->progress |= PROGRESS_READY;
3524         ASSERT(bgep->bge_guard == BGE_GUARD);
3525 #ifdef BGE_IPMI_ASF
3526 #ifdef BGE_NETCONSOLE
3527         if (bgep->asf_enabled) {
3528                 mutex_enter(bgep->genlock);
3529                 retval = bge_chip_start(bgep, B_TRUE);
3530                 mutex_exit(bgep->genlock);
3531                 if (retval != DDI_SUCCESS)
3532                         goto attach_fail;
3533         }
3534 #endif
3535 #endif

3537         ddi_report_dev(devinfo);

3539         return (DDI_SUCCESS);
```

```
3541 attach_fail:
3542 #ifdef BGE_IPMI_ASF
3543         bge_unattach(bgep, ASF_MODE_SHUTDOWN);
3544 #else
3545         bge_unattach(bgep);
3546 #endif
3547         return (DDI_FAILURE);
3548 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   46461 Fri Jan 11 13:34:57 2013
new/usr/src/uts/common/io/bge/bge_mii.c
Just the 5719/5720 changes
**********************************************************
```

```
/*
 * 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) 2002, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright 2012 Nexenta Systems, Inc.  All rights reserved.
 */

#include "bge_impl.h"

/*
 * Bit test macros, returning boolean_t values
 */
#define BIS(w, b)       (((w) & (b)) ? B_TRUE : B_FALSE)
#define BIC(w, b)       (((w) & (b)) ? B_FALSE : B_TRUE)
#define UPORDOWN(x)     ((x) ? "up" : "down")

/*
 * ========== Copper (PHY) support ==========
 */

#define BGE_DBG         BGE_DBG_PHY     /* debug flag for this code     */

/*
 * #defines:
 *      BGE_COPPER_WIRESPEED controls whether the Broadcom WireSpeed(tm)
 *      feature is enabled.  We need to recheck whether this can be
 *      enabled; at one time it seemed to interact unpleasantly with the
 *      loopback modes.
 *
 *      BGE_COPPER_IDLEOFF controls whether the (copper) PHY power is
 *      turned off when the PHY is idled i.e. during driver suspend().
 *      For now this is disabled because the chip doesn't seem to
 *      resume cleanly if the PHY power is turned off.
 */
#define BGE_COPPER_WIRESPEED    B_TRUE
#define BGE_COPPER_IDLEOFF      B_FALSE

/*
 * The arrays below can be indexed by the MODE bits from the Auxiliary
```



```
 * Status register to determine the current speed/duplex settings.
 */
static const int16_t bge_copper_link_speed[] = {
        0,                              /* MII_AUX_STATUS_MODE_NONE     */
        10,                             /* MII_AUX_STATUS_MODE_10_H     */
        10,                             /* MII_AUX_STATUS_MODE_10_F     */
        100,                            /* MII_AUX_STATUS_MODE_100_H    */
        0,                              /* MII_AUX_STATUS_MODE_100_4    */
        100,                            /* MII_AUX_STATUS_MODE_100_F    */
        1000,                           /* MII_AUX_STATUS_MODE_1000_H   */
        1000                            /* MII_AUX_STATUS_MODE_1000_F   */
};
```

*unchanged_portion_omitted_*

```
/*
 * Basic low-level function to reset the PHY.
 * Doesn't incorporate any special-case workarounds.
 *
 * Returns TRUE on success, FALSE if the RESET bit doesn't clear
 */
static boolean_t
bge_phy_reset(bge_t *bgep)
{
        uint16_t control;
        uint_t count;
        boolean_t ret = B_FALSE;

        BGE_TRACE(("bge_phy_reset($%p)", (void *)bgep));

        ASSERT(mutex_owned(bgep->genlock));

        if (DEVICE_5906_SERIES_CHIPSETS(bgep)) {
                drv_usecwait(40);
                /* put PHY into ready state */
                bge_reg_clr32(bgep, MISC_CONFIG_REG, MISC_CONFIG_EPHY_IDDQ);
                (void) bge_reg_get32(bgep, MISC_CONFIG_REG); /* flush */
                drv_usecwait(40);
        }

        /*
         * Set the PHY RESET bit, then wait up to 50 ms for it to self-clear
         * Set the PHY RESET bit, then wait up to 5 ms for it to self-clear
         */
        bge_mii_put16(bgep, MII_CONTROL, MII_CONTROL_RESET);
        for (count = 0; ++count < 5000; ) {
        for (count = 0; ++count < 1000; ) {
                drv_usecwait(5);
                control = bge_mii_get16(bgep, MII_CONTROL);
                if (BIC(control, MII_CONTROL_RESET)) {
                        drv_usecwait(40);
                        ret = B_TRUE;
                        break;
                if (BIC(control, MII_CONTROL_RESET))
                        return (B_TRUE);
                }
                drv_usecwait(10);
        }

        if (ret == B_TRUE && DEVICE_5906_SERIES_CHIPSETS(bgep))
        if (DEVICE_5906_SERIES_CHIPSETS(bgep))
                (void) bge_adj_volt_5906(bgep);

        if (ret == B_FALSE)
                BGE_DEBUG(("bge_phy_reset: FAILED, control now 0x%x", control));

        return (ret);
```

```
239         return (B_FALSE);
249 }
_____unchanged_portion_omitted_

538 /*
539  * End of Broadcom-derived workaround code                              *
540  */

542 static int
543 bge_restart_copper(bge_t *bgep, boolean_t powerdown)
544 {
545         uint16_t phy_status;
546         boolean_t reset_ok;
547         uint16_t extctrl, auxctrl;

549         BGE_TRACE(("bge_restart_copper($%p, %d)", (void *)bgep, powerdown));

551         ASSERT(mutex_owned(bgep->genlock));

553         if (bgep->chipid.flags & CHIP_FLAG_NO_CHECK_RESET) {
544         switch (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev)) {
545         default:
546                 /*
547                  * Shouldn't happen; it means we don't recognise this chip.
548                  * It's probably a new one, so we'll try our best anyway ...
549                  */
550         case MHCR_CHIP_ASIC_REV_5703:
551         case MHCR_CHIP_ASIC_REV_5704:
552         case MHCR_CHIP_ASIC_REV_5705:
553         case MHCR_CHIP_ASIC_REV_5752:
554         case MHCR_CHIP_ASIC_REV_5714:
555         case MHCR_CHIP_ASIC_REV_5715:
556                 reset_ok = bge_phy_reset_and_check(bgep);
557                 break;

559         case MHCR_CHIP_ASIC_REV_5906:
560         case MHCR_CHIP_ASIC_REV_5700:
561         case MHCR_CHIP_ASIC_REV_5701:
562         case MHCR_CHIP_ASIC_REV_5723:
563         case MHCR_CHIP_ASIC_REV_5721_5751:
564                 /*
565                  * Just a plain reset; the "check" code breaks these chips
566                  */
554                 reset_ok = bge_phy_reset(bgep);
555                 if (!reset_ok)
556                         bge_fm_ereport(bgep, DDI_FM_DEVICE_NO_RESPONSE);
557         } else {
558                 reset_ok = bge_phy_reset_and_check(bgep);
570                 break;
559         }

561         if (!reset_ok) {
562                 BGE_REPORT((bgep, "PHY failed to reset correctly"));
563                 return (DDI_FAILURE);
564         }

566         /*
567          * Step 5: disable WOL (not required after RESET)
568          *
569          * Step 6: refer to errata
570          */
571         switch (bgep->chipid.asic_rev) {
572         default:
573                 break;

575         case MHCR_CHIP_REV_5704_A0:
```



```
576                 bge_phy_tweak_gmii(bgep);
577                 break;
578         }

580         switch (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev)) {
581         case MHCR_CHIP_ASIC_REV_5705:
582         case MHCR_CHIP_ASIC_REV_5750:
593         case MHCR_CHIP_ASIC_REV_5721_5751:
583                 bge_phy_bit_err_fix(bgep);
584                 break;
585         }

587         if (!(bgep->chipid.flags & CHIP_FLAG_NO_JUMBO) &&
588             (bgep->chipid.default_mtu > BGE_DEFAULT_MTU)) {
589                 /* Set the GMII Fifo Elasticity to high latency */
590                 extctrl = bge_mii_get16(bgep, 0x10);
591                 bge_mii_put16(bgep, 0x10, extctrl | 0x1);

593                 /* Allow reception of extended length packets */
594                 bge_mii_put16(bgep, MII_AUX_CONTROL, 0x0007);
595                 auxctrl = bge_mii_get16(bgep, MII_AUX_CONTROL);
596                 auxctrl |= 0x4000;
597                 bge_mii_put16(bgep, MII_AUX_CONTROL, auxctrl);
598         }

600         /*
601          * Step 7: read the MII_INTR_STATUS register twice,
602          * in order to clear any sticky bits (but they should
603          * have been cleared by the RESET, I think), and we're
604          * not using PHY interrupts anyway.
605          *
606          * Step 8: enable the PHY to interrupt on link status
607          * change (not required)
608          *
609          * Step 9: configure PHY LED Mode - not applicable?
610          *
611          * Step 10: read the MII_STATUS register twice, in
612          * order to clear any sticky bits (but they should
613          * have been cleared by the RESET, I think).
614          */
615         phy_status = bge_mii_get16(bgep, MII_STATUS);
616         phy_status = bge_mii_get16(bgep, MII_STATUS);
617         BGE_DEBUG(("bge_restart_copper: status 0x%x", phy_status));

619         /*
620          * Finally, shut down the PHY, if required
621          */
622         if (powerdown)
623                 bge_phy_powerdown(bgep);
624         return (DDI_SUCCESS);
625 }
_____unchanged_portion_omitted_

1473 /*
1474  * ========== Exported physical layer control routines ==========
1475  */

1477 #undef  BGE_DBG
1478 #define BGE_DBG         BGE_DBG_PHYS    /* debug flag for this code     */

1480 /*
1481  * Here we have to determine which media we're using (copper or serdes).
1482  * Once that's done, we can initialise the physical layer appropriately.
1483  */
1484 int
1485 bge_phys_init(bge_t *bgep)
```

```
1486 {
1487         BGE_TRACE(("bge_phys_init($%p)", (void *)bgep));

1489         mutex_enter(bgep->genlock);

1491         /*
1492          * Probe for the (internal) PHY.  If it's not there, we'll assume
1493          * that this is a 5703/4S, with a SerDes interface rather than
1494          * a PHY. BCM5714S/BCM5715S are not supported.It are based on
1495          * BCM800x PHY.
1496          */
1497         bgep->phy_mii_addr = 1;
1498         if (DEVICE_5717_SERIES_CHIPSETS(bgep)) {
1499                 uint32_t regval = bge_reg_get32(bgep, CPMU_STATUS_REG);
1500                 if (MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
1501                     MHCR_CHIP_ASIC_REV_5719 ||
1502                     MHCR_CHIP_ASIC_REV(bgep->chipid.asic_rev) ==
1503                     MHCR_CHIP_ASIC_REV_5720) {
1504                         bgep->phy_mii_addr +=
1505                             (regval & CPMU_STATUS_FUN_NUM_5719) >>
1506                             CPMU_STATUS_FUN_NUM_5719_SHIFT;
1507                 } else {
1508                         bgep->phy_mii_addr +=
1509                             (regval & CPMU_STATUS_FUN_NUM_5717) ? 1 : 0;
1510                 }
1510                 int regval = bge_reg_get32(bgep, CPMU_STATUS_REG);
1511                 if (regval & CPMU_STATUS_FUN_NUM)
1512                         bgep->phy_mii_addr += 1;
1511                 regval = bge_reg_get32(bgep, SGMII_STATUS_REG);
1512                 if (regval & MEDIA_SELECTION_MODE)
1513                         bgep->phy_mii_addr += 7;
1514         }

1515         if (bge_phy_probe(bgep)) {
1516                 bgep->chipid.flags &= ~CHIP_FLAG_SERDES;
1517                 bgep->physops = &copper_ops;
1518         } else {
1519                 bgep->chipid.flags |= CHIP_FLAG_SERDES;
1520                 bgep->physops = &serdes_ops;
1521         }

1523         if ((*bgep->physops->phys_restart)(bgep, B_FALSE) != DDI_SUCCESS) {
1524                 mutex_exit(bgep->genlock);
1525                 return (EIO);
1526         }
1527         if (bge_check_acc_handle(bgep, bgep->io_handle) != DDI_FM_OK) {
1528                 mutex_exit(bgep->genlock);
1529                 return (EIO);
1530         }
1531         mutex_exit(bgep->genlock);
1532         return (0);
1533 }
```

_____*unchanged_portion_omitted_*