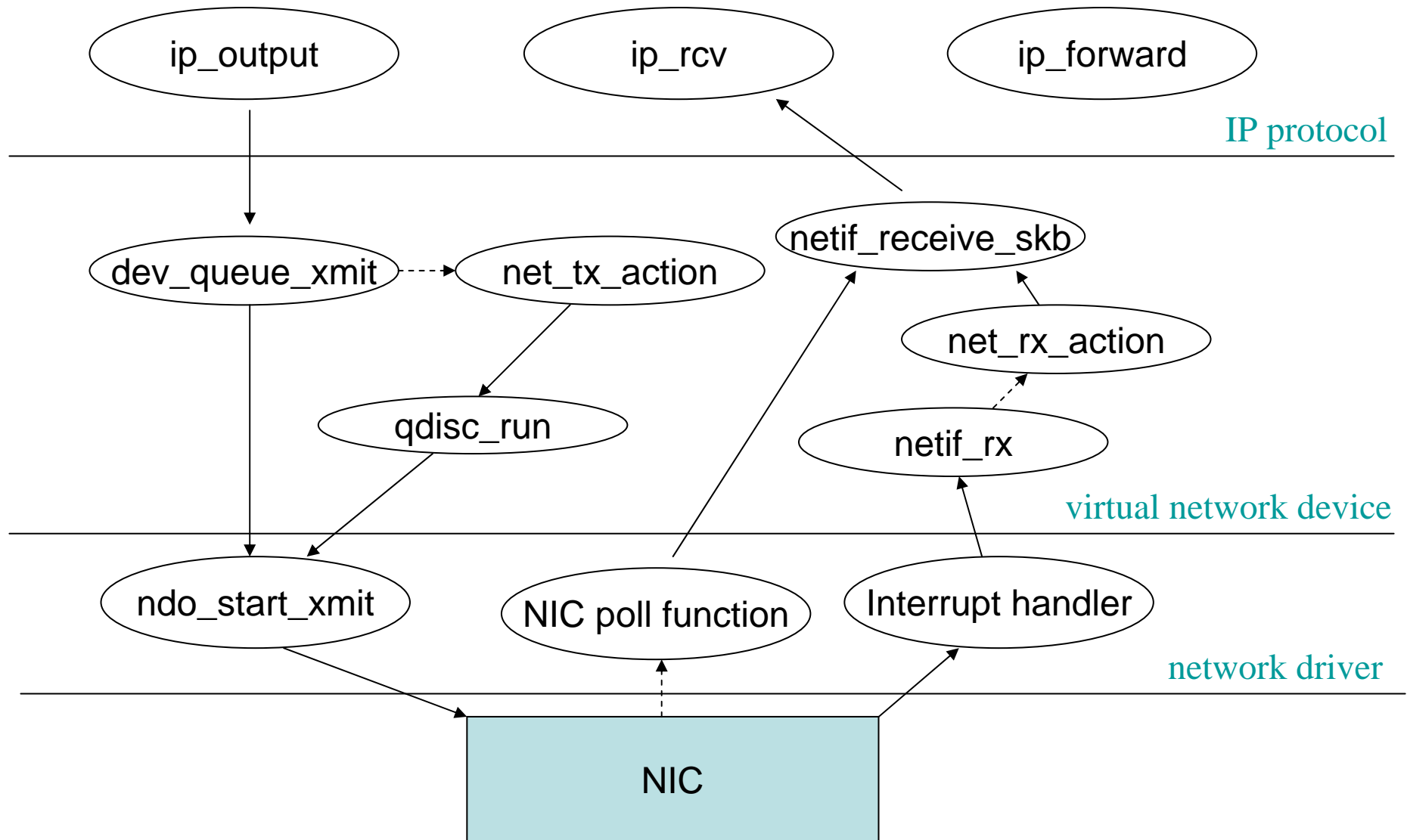


Architecture Overview



```

00001:  /* sis900.c: A SIS 900/7016 PCI Fast Ethernet driver for Linux.
00002:  Copyright 1999 Silicon Integrated System Corporation
00003:  Revision:   1.08.06 Sep. 24 2002
00004:
00005:  Modified from the driver which is originally written by Donald Becker.
00006:
00007:  This software may be used and distributed according to the terms
00008:  of the GNU General Public License (GPL), incorporated herein by reference.
00009:  Drivers based on this skeleton fall under the GPL and must retain
00010:  the authorship (implicit copyright) notice.
00011:
00012:  References:
00013:  SIS 7016 Fast Ethernet PCI Bus 10/100 Mbps LAN Controller with OnNow Support,
00014:  preliminary Rev. 1.0 Jan. 14, 1998
00015:  SIS 900 Fast Ethernet PCI Bus 10/100 Mbps LAN Single Chip with OnNow Support,
00016:  preliminary Rev. 1.0 Nov. 10, 1998
00017:  SIS 7014 Single Chip 100BASE-TX/10BASE-T Physical Layer Solution,
00018:  preliminary Rev. 1.0 Jan. 18, 1998
00019:  http://www.sis.com.tw/support/databook.htm
00020:
00021:  Rev 1.08.07 Nov. 2 2003 Daniele Venzano <webvenza@libero.it> add suspend/resume support
00022:  Rev 1.08.06 Sep. 24 2002 Mufasa Yang bug fix for Tx timeout & add SIS963 support
00023:  Rev 1.08.05 Jun. 6 2002 Mufasa Yang bug fix for read_eeprom & Tx descriptor over-boundary
00024:  Rev 1.08.04 Apr. 25 2002 Mufasa Yang <mufasa@sis.com.tw> added SIS962 support
00025:  Rev 1.08.03 Feb. 1 2002 Matt Domsch <Matt_Domsch@dell.com> update to use library crc32 function
00026:  Rev 1.08.02 Nov. 30 2001 Hui-Fen Hsu workaround for EDB & bug fix for dhcp problem
00027:  Rev 1.08.01 Aug. 25 2001 Hui-Fen Hsu update for 630ET & workaround for ICS1893 PHY
00028:  Rev 1.08.00 Jun. 11 2001 Hui-Fen Hsu workaround for RTL8201 PHY and some bug fix
00029:  Rev 1.07.11 Apr. 2 2001 Hui-Fen Hsu updates PCI drivers to use the new pci_set_dma_mask for kernel 2.4.3
00030:  Rev 1.07.10 Mar. 1 2001 Hui-Fen Hsu <hfhsu@sis.com.tw> some bug fix & 635M/B support
00031:  Rev 1.07.09 Feb. 9 2001 Dave Jones <davej@suse.de> PCI enable cleanup
00032:  Rev 1.07.08 Jan. 8 2001 Lei-Chun Chang added RTL8201 PHY support
00033:  Rev 1.07.07 Nov. 29 2000 Lei-Chun Chang added kernel-doc extractable documentation and 630 workaround fix
00034:  Rev 1.07.06 Nov. 7 2000 Jeff Garzik <jgarzik@pobox.com> some bug fix and cleaning
00035:  Rev 1.07.05 Nov. 6 2000 metapirat<metapirat@gmx.de> contribute media type select by ifconfig
00036:  Rev 1.07.04 Sep. 6 2000 Lei-Chun Chang added ICS1893 PHY support
00037:  Rev 1.07.03 Aug. 24 2000 Lei-Chun Chang (lcchang@sis.com.tw) modified 630E equalizer workaround rule
00038:  Rev 1.07.01 Aug. 08 2000 Ollie Lho minor update for SIS 630E and SIS 630E A1
00039:  Rev 1.07 Mar. 07 2000 Ollie Lho bug fix in Rx buffer ring
00040:  Rev 1.06.04 Feb. 11 2000 Jeff Garzik <jgarzik@pobox.com> softnet and init for kernel 2.4
00041:  Rev 1.06.03 Dec. 23 1999 Ollie Lho Third release
00042:  Rev 1.06.02 Nov. 23 1999 Ollie Lho bug in mac probing fixed
00043:  Rev 1.06.01 Nov. 16 1999 Ollie Lho CRC calculation provide by Joseph Zbiciak (im14u2c@primenet.com)
00044:  Rev 1.06 Nov. 4 1999 Ollie Lho (ollie@sis.com.tw) Second release
00045:  Rev 1.05.05 Oct. 29 1999 Ollie Lho (ollie@sis.com.tw) Single buffer Tx/Rx
00046:  Chin-Shan Li (lcs@sis.com.tw) Added AMD Am79c901 HomePNA PHY support
00047:  Rev 1.05 Aug. 7 1999 Jim Huang (cmhuang@sis.com.tw) Initial release
00048:  */
00049:
00050: #include <linux/module.h>
00051: #include <linux/moduleparam.h>
00052: #include <linux/kernel.h>
00053: #include <linux/string.h>
00054: #include <linux/timer.h>
00055: #include <linux/errno.h>
00056: #include <linux/loport.h>
00057: #include <linux/slab.h>
00058: #include <linux/interrupt.h>
00059: #include <linux/pci.h>
00060: #include <linux/netdevice.h>
00061: #include <linux/init.h>
00062: #include <linux/mii.h>
00063: #include <linux/etherdevice.h>
00064: #include <linux/skbuff.h>
00065: #include <linux/delay.h>
00066: #include <linux/ethtool.h>
00067: #include <linux/crc32.h>
00068: #include <linux/bitops.h>
00069:
00070: #include <asm/processor.h> /* Processor type for cache alignment. */
00071: #include <asm/io.h>
00072: #include <asm/uaccess.h> /* User space memory access functions */
00073:
00074: #include "sis900.h"
00075:
00076: #define SIS900_MODULE_NAME "sis900"
00077: #define SIS900_DRV_VERSION "v1.08.07 11/02/2003"
00078:
00079: static char version[] __devinitdata =
00080: KERN_INFO "sis900.c: " SIS900_DRV_VERSION "\n";
00081:

```

```

00082: static int max_interrupt_work = 40;
00083: static int multicast_filter_limit = 128;
00084:
00085: #define sis900_debug debug
00086: static int sis900_debug;
00087:
00088: /* Time in jiffies before concluding the transmitter is hung. */
00089: #define TX_TIMEOUT (4*HZ)
00090: /* SIS 900 is capable of 32 bits BM DMA */
00091: #define SIS900_DMA_MASK 0xffffffff
00092:
00093: enum {
00094:     SIS_900 = 0,
00095:     SIS_7016
00096: };
00097: static char *card_names[] = {
00098:     "SIS 900 PCI Fast Ethernet",
00099:     "SIS 7016 PCI Fast Ethernet"
00100: };
00101: static struct pci_device_id sis900_pci_tbl[] = {
00102:     {PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_900,
00103:      PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_900},
00104:     {PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_7016,
00105:      PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_7016},
00106:     {0,}
00107: };
00108: MODULE_DEVICE_TABLE(pci, sis900_pci_tbl);
00109:
00110: static void sis900_read_mode(struct net_device *net_dev, int *speed, int *duplex);
00111:
00112: static struct mii_chip_info {
00113:     const char *name;
00114:     u16 phy_id0;
00115:     u16 phy_id1;
00116:     u8 phy_types;
00117: #define HOME 0x0001
00118: #define LAN 0x0002
00119: #define MIX 0x0003
00120: #define UNKNOWN 0x0
00121: } mii_chip_table[] = {
00122:     { "SIS 900 Internal MII PHY", 0x001d, 0x8000, LAN },
00123:     { "SIS 7014 Physical Layer Solution", 0x0016, 0xF830, LAN },
00124:     { "Altimata AC101LF PHY", 0x0022, 0x5520, LAN },
00125:     { "AMD 79C901 10BASE-T PHY", 0x0000, 0x6B70, LAN },
00126:     { "AMD 79C901 HomePNA PHY", 0x0000, 0x6B90, HOME },
00127:     { "ICS LAN PHY", 0x0015, 0xF440, LAN },
00128:     { "NS 83851 PHY", 0x2000, 0x5C20, MIX },
00129:     { "NS 83847 PHY", 0x2000, 0x5C30, MIX },
00130:     { "Realtek RTL8201 PHY", 0x0000, 0x8200, LAN },
00131:     { "VIA 6103 PHY", 0x0101, 0x8F20, LAN },
00132:     { NULL, },
00133: };
00134:
00135: struct mii_phy {
00136:     struct mii_phy *next;
00137:     int phy_addr;
00138:     u16 phy_id0;
00139:     u16 phy_id1;
00140:     u16 status;
00141:     u8 phy_types;
00142: };
00143:
00144: typedef struct _BufferDesc {
00145:     u32 link;
00146:     u32 cmdsts;
00147:     u32 bufptr;
00148: } BufferDesc;
00149:
00150: struct sis900_private {
00151:     struct net_device_stats stats;
00152:     struct pci_dev *pci_dev;
00153:
00154:     spinlock_t lock;
00155:
00156:     struct mii_phy *mii;
00157:     struct mii_phy *first_mii; /* record the first mii structure */
00158:     unsigned int cur_phy;
00159:
00160:     struct timer_list timer; /* Link status detection timer. */
00161:     u8 autong_complete; /* 1: auto-negotiate complete */
00162:

```

```

00163: unsigned int cur_rx, dirty_rx; /* producer/consumer pointers for Tx/Rx ring */
00164: unsigned int cur_tx, dirty_tx;
00165:
00166: /* The saved address of a sent/receive-in-place packet buffer */
00167: struct sk_buff *tx_skbuff[NUM_TX_DESC];
00168: struct sk_buff *rx_skbuff[NUM_RX_DESC];
00169: BufferDesc *tx_ring;
00170: BufferDesc *rx_ring;
00171:
00172: dma_addr_t tx_ring_dma;
00173: dma_addr_t rx_ring_dma;
00174:
00175: unsigned int tx_full; /* The Tx queue is full. */
00176: u8 host_bridge_rev;
00177: } ? end sis900_private ?;
00178:
00179: MODULE_AUTHOR("Jim Huang <cmhuang@sis.com.tw>, Ollie Lho <ollie@sis.com.tw>");
00180: MODULE_DESCRIPTION("SIS 900 PCI Fast Ethernet driver");
00181: MODULE_LICENSE("GPL");
00182:
00183: module_param(multicast_filter_limit, int, 0444);
00184: module_param(max_interrupt_work, int, 0444);
00185: module_param(debug, int, 0444);
00186: MODULE_PARM_DESC(multicast_filter_limit,
00186: "SIS 900/7016 maximum number of filtered multicast addresses");
00187: MODULE_PARM_DESC(max_interrupt_work, "SIS 900/7016 maximum events handled per interrupt");
00188: MODULE_PARM_DESC(debug, "SIS 900/7016 debug level (2-4)");
00189:
00190: static int sis900_open(struct net_device *net_dev);
00191: static int sis900_mii_probe(struct net_device *net_dev);
00192: static void sis900_init_rxfilter(struct net_device *net_dev);
00193: static u16 read_eeprom(long ioaddr, int location);
00194: static u16 mdio_read(struct net_device *net_dev, int phy_id, int location);
00195: static void mdio_write(struct net_device *net_dev, int phy_id, int location, int val);
00196: static void sis900_timer(unsigned long data);
00197: static void sis900_check_mode(struct net_device *net_dev, struct mii_phy *mii_phy);
00198: static void sis900_tx_timeout(struct net_device *net_dev);
00199: static void sis900_init_tx_ring(struct net_device *net_dev);
00200: static void sis900_init_rx_ring(struct net_device *net_dev);
00201: static int sis900_start_xmit(struct sk_buff *skb, struct net_device *net_dev);
00202: static int sis900_rx(struct net_device *net_dev);
00203: static void sis900_finish_xmit(struct net_device *net_dev);
00204: static irqreturn_t sis900_interrupt(int irq, void *dev_instance, struct pt_regs *regs);
00205: static int sis900_close(struct net_device *net_dev);
00206: static int mii_ioctl(struct net_device *net_dev, struct ifreq *rq, int cmd);
00207: static struct net_device_stats *sis900_get_stats(struct net_device *net_dev);
00208: static u16 sis900_mcast_bitnr(u8 *addr, u8 revision);
00209: static void set_rx_mode(struct net_device *net_dev);
00210: static void sis900_reset(struct net_device *net_dev);
00211: static void sis630_set_eq(struct net_device *net_dev, u8 revision);
00212: static int sis900_set_config(struct net_device *dev, struct ifmap *map);
00213: static u16 sis900_default_phy(struct net_device *net_dev);
00214: static void sis900_set_capability(struct net_device *net_dev, struct mii_phy *phy);
00215: static u16 sis900_reset_phy(struct net_device *net_dev, int phy_addr);
00216: static void sis900_auto_negotiate(struct net_device *net_dev, int phy_addr);
00217: static void sis900_set_mode(long ioaddr, int speed, int duplex);
00218: static struct ethtool_ops sis900_ethtool_ops;
00219:
00220: /**
00221:  * sis900_get_mac_addr - Get MAC address for stand alone SIS900 model
00222:  * @pci_dev: the sis900 pci device
00223:  * @net_dev: the net device to get address for
00224:  *
00225:  * Older SIS900 and friends, use EEPROM to store MAC address.
00226:  * MAC address is read from read_eeprom() into @net_dev->dev_addr.
00227:  */
00228:

```

```

00229: static int __devinit sis900_get_mac_addr(struct pci_dev *pci_dev, struct net_device *net_dev)
00230: {
00231:     long ioaddr = pci_resource_start(pci_dev, 0);
00232:     u16 signature;
00233:     int i;
00234:
00235:     /* check to see if we have sane EEPROM */
00236:     signature = (u16) read_eeprom(ioaddr, EEPROMSignature);
00237:     if (signature == 0xffff || signature == 0x0000) {
00238:         printk(KERN_INFO "%s: Error EEPROM read %x\n",
00239:             net_dev->name, signature);
00240:         return 0;
00241:     }
00242:
00243:     /* get MAC address from EEPROM */
00244:     for (i = 0; i < 3; i++)
00245:         ((u16 *) (net_dev->dev_addr))[i] = read_eeprom(ioaddr, i+EEPROMMACAddr);
00246:
00247:     return 1;
00248: } ? end sis900_get_mac_addr ?
00249:
00250: /**
00251:  * sis630e_get_mac_addr - Get MAC address for SIS630E model
00252:  * @pci_dev: the sis900 pci device
00253:  * @net_dev: the net device to get address for
00254:  *
00255:  * SIS630E model, use APC CMOS RAM to store MAC address.
00256:  * APC CMOS RAM is accessed through ISA bridge.
00257:  * MAC address is read into @net_dev->dev_addr.
00258:  */
00259:
00260: static int __devinit sis630e_get_mac_addr(struct pci_dev *pci_dev,
00261:     struct net_device *net_dev)
00262: {
00263:     struct pci_dev *isa_bridge = NULL;
00264:     u8 reg;
00265:     int i;
00266:
00267:     isa_bridge = pci_get_device(PCI_VENDOR_ID_SI, 0x0008, isa_bridge);
00268:     if (!isa_bridge)
00269:         isa_bridge = pci_get_device(PCI_VENDOR_ID_SI, 0x0018, isa_bridge);
00270:     if (!isa_bridge) {
00271:         printk("%s: Can not find ISA bridge\n", net_dev->name);
00272:         return 0;
00273:     }
00274:     pci_read_config_byte(isa_bridge, 0x48, &reg);
00275:     pci_write_config_byte(isa_bridge, 0x48, reg | 0x40);
00276:
00277:     for (i = 0; i < 6; i++) {
00278:         outb(0x09 + i, 0x70);
00279:         ((u8 *) (net_dev->dev_addr))[i] = inb(0x71);
00280:     }
00281:     pci_write_config_byte(isa_bridge, 0x48, reg & ~0x40);
00282:     pci_dev_put(isa_bridge);
00283:
00284:     return 1;
00285: } ? end sis630e_get_mac_addr ?
00286:
00287: /**
00288:  * sis635_get_mac_addr - Get MAC address for SIS635 model
00289:  * @pci_dev: the sis900 pci device
00290:  * @net_dev: the net device to get address for
00291:  *
00292:  * SIS635 model, set MAC Reload Bit to load Mac address from APC
00293:  * to rfrd. rfrd is accessed through rfrd. MAC address is read into
00294:  * @net_dev->dev_addr.
00295:  */
00296:
00297:
00298: static int __devinit sis635_get_mac_addr(struct pci_dev *pci_dev,
00299:     struct net_device *net_dev)
00300: {
00301:     long ioaddr = net_dev->base_addr;
00302:     u32 rfrdSave;
00303:     u32 i;
00304:
00305:     rfrdSave = inl(rfrd + ioaddr);
00306:
00307:     outl(rfrdSave | RELOAD, ioaddr + cr);
00308:     outl(0, ioaddr + cr);
00309:

```

```

00310:  /* disable packet filtering before setting filter */
00311:  outl(rfcrSave & ~RFEN, rfcr + loaddr);
00312:
00313:  /* load MAC addr to filter data register */
00314:  for (i = 0; i < 3; i++) {
00315:      outl((i << RFADDR_shift), loaddr + rfcr);
00316:      * ((u16 *)net_dev->dev_addr) + i) = inw(loaddr + rfdr);
00317:  }
00318:
00319:  /* enable packet filtering */
00320:  outl(rfcrSave | RFEN, rfcr + loaddr);
00321:
00322:  return 1;
00323: } ? end sis635_get_mac_addr ?
00324:
00325: /**
00326:  * sis96x_get_mac_addr - Get MAC address for SIS962 or SIS963 model
00327:  * @pci_dev: the sis900 pci device
00328:  * @net_dev: the net device to get address for
00329:  *
00330:  * SIS962 or SIS963 model, use EEPROM to store MAC address. And EEPROM
00331:  * is shared by
00332:  * LAN and 1394. When access EEPROM, send EEREQ signal to hardware first
00333:  * and wait for EEGNT. If EEGNT is ON, EEPROM is permitted to be access
00334:  * by LAN, otherwise is not. After MAC address is read from EEPROM, send
00335:  * EEDONE signal to refuse EEPROM access by LAN.
00336:  * The EEPROM map of SIS962 or SIS963 is different to SIS900.
00337:  * The signature field in SIS962 or SIS963 spec is meaningless.
00338:  * MAC address is read into @net_dev->dev_addr.
00339:  */
00340:
00341: static int __devinit sis96x_get_mac_addr(struct pci_dev * pci_dev,
00342:                                         struct net_device *net_dev)
00343: {
00344:     long loaddr = net_dev->base_addr;
00345:     long ee_addr = loaddr + mear;
00346:     u32 waittime = 0;
00347:     int i;
00348:
00349:     outl(EEREQ, ee_addr);
00350:     while(waittime < 2000) {
00351:         if(inl(ee_addr) & EEGNT) {
00352:
00353:             /* get MAC address from EEPROM */
00354:             for (i = 0; i < 3; i++)
00355:                 ((u16 *)net_dev->dev_addr)[i] = read_eeprom(loaddr, i+EEDROMMACAddr);
00356:
00357:             outl(EEDONE, ee_addr);
00358:             return 1;
00359:         } else {
00360:             udelay(1);
00361:             waittime++;
00362:         }
00363:     }
00364:     outl(EEDONE, ee_addr);
00365:     return 0;
00366: } ? end sis96x_get_mac_addr ?
00367:
00368: /**
00369:  * sis900_probe - Probe for sis900 device
00370:  * @pci_dev: the sis900 pci device
00371:  * @pci_id: the pci device ID
00372:  *
00373:  * Check and probe sis900 net device for @pci_dev.
00374:  * Get mac address according to the chip revision,
00375:  * and assign SIS900-specific entries in the device structure.
00376:  * ie: sis900_open(), sis900_start_xmit(), sis900_close(), etc.
00377:  */
00378:

```

```

00379: static int __devinit sis900_probe(struct pci_dev *pci_dev,
00380:                                  const struct pci_device_id *pci_id)
00381: {
00382:     struct sis900_private *sis_priv;
00383:     struct net_device *net_dev;
00384:     struct pci_dev *dev;
00385:     dma_addr_t ring_dma;
00386:     void *ring_space;
00387:     long loaddr;
00388:     int i, ret;
00389:     u8 revision;
00390:     char *card_name = card_names[pci_id->driver_data];
00391:
00392:     /* when built into the kernel, we only print version if device is found */
00393:     #ifndef MODULE
00394:     static int printed_version;
00395:     if (!printed_version++)
00396:         printk(version);
00397:     #endif
00398:
00399:     /* setup various bits in PCI command register */
00400:     ret = pci_enable_device(pci_dev);
00401:     if(ret) return ret;
00402:
00403:     i = pci_set_dma_mask(pci_dev, SIS900_DMA_MASK);
00404:     if(i){
00405:         printk(KERN_ERR "sis900.c: architecture does not support"
00406:              "32bit PCI busmaster DMA\n");
00407:         return i;
00408:     }
00409:
00410:     pci_set_master(pci_dev);
00411:
00412:     net_dev = alloc_etherdev(sizeof(struct sis900_private));
00413:     if (!net_dev)
00414:         return -ENOMEM;
00415:     SET_MODULE_OWNED(net_dev);
00416:     SET_NETDEV_DEV(net_dev, &pci_dev->dev);
00417:
00418:     /* We do a request_region() to register /proc/loports info. */
00419:     loaddr = pci_resource_start(pci_dev, 0);
00420:     ret = pci_request_regions(pci_dev, "sis900");
00421:     if (ret)
00422:         goto ↓err_out;
00423:
00424:     sis_priv = net_dev->priv;
00425:     net_dev->base_addr = loaddr;
00426:     net_dev->irq = pci_dev->irq;
00427:     sis_priv->pci_dev = pci_dev;
00428:     spin_lock_init(&sis_priv->lock);
00429:
00430:     pci_set_drvdata(pci_dev, net_dev);
00431:
00432:     ring_space = pci_alloc_consistent(pci_dev, TX_TOTAL_SIZE, &ring_dma);
00433:     if (!ring_space) {
00434:         ret = -ENOMEM;
00435:         goto ↓err_out_cleardev;
00436:     }
00437:     sis_priv->tx_ring = (BufferDesc *)ring_space;
00438:     sis_priv->tx_ring_dma = ring_dma;
00439:
00440:     ring_space = pci_alloc_consistent(pci_dev, RX_TOTAL_SIZE, &ring_dma);
00441:     if (!ring_space) {
00442:         ret = -ENOMEM;
00443:         goto ↓err_unmap_tx;
00444:     }
00445:     sis_priv->rx_ring = (BufferDesc *)ring_space;
00446:     sis_priv->rx_ring_dma = ring_dma;
00447:
00448:     /* The SIS900-specific entries in the device structure. */
00449:     net_dev->open = &sis900_open;
00450:     net_dev->hard_start_xmit = &sis900_start_xmit;
00451:     net_dev->stop = &sis900_close;
00452:     net_dev->get_stats = &sis900_get_stats;
00453:     net_dev->set_config = &sis900_set_config;
00454:     net_dev->set_multicast_list = &set_rx_mode;
00455:     net_dev->do_ioctl = &mii_ioctl;
00456:     net_dev->tx_timeout = sis900_tx_timeout;
00457:     net_dev->watchdog_timeo = TX_TIMEOUT;
00458:     net_dev->ethtool_ops = &sis900_ethtool_ops;
00459:

```

```

00460: ret = register_netdev(net_dev);
00461: if (ret)
00462:     goto ↓err_unmap_rx;
00463:
00464: /* Get Mac address according to the chip revision */
00465: pci_read_config_byte(pci_dev, PCI_CLASS_REVISION, &revision);
00466: ret = 0;
00467:
00468: if (revision == SIS630E_900_REV)
00469:     ret = sis630e_get_mac_addr(pci_dev, net_dev);
00470: else if ((revision > 0x81) && (revision <= 0x90) )
00471:     ret = sis635_get_mac_addr(pci_dev, net_dev);
00472: else if (revision == SIS96x_900_REV)
00473:     ret = sis96x_get_mac_addr(pci_dev, net_dev);
00474: else
00475:     ret = sis900_get_mac_addr(pci_dev, net_dev);
00476:
00477: if (ret == 0) {
00478:     ret = -ENODEV;
00479:     goto ↓err_out_unregister;
00480: }
00481:
00482: /* 630ET : set the mii access mode as software-mode */
00483: if (revision == SIS630ET_900_REV)
00484:     outl(ACCESSMODE | inl(loaddr + cr), loaddr + cr);
00485:
00486: /* probe for mii transceiver */
00487: if (sis900_mii_probe(net_dev) == 0) {
00488:     ret = -ENODEV;
00489:     goto ↓err_out_unregister;
00490: }
00491:
00492: /* save our host bridge revision */
00493: dev = pci_get_device(PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_630, NULL);
00494: if (dev) {
00495:     pci_read_config_byte(dev, PCI_CLASS_REVISION, &sis_priv->host_bridge_rev);
00496:     pci_dev_put(dev);
00497: }
00498:
00499: /* print some information about our NIC */
00500: printk(KERN_INFO "%s: %s at %#lx, IRQ %d, ", net_dev->name,
00501:     card_name, loaddr, net_dev->irq);
00502: for (i = 0; i < 5; i++)
00503:     printk("%2.2x:", (u8)net_dev->dev_addr[i]);
00504: printk("%2.2x.\n", net_dev->dev_addr[i]);
00505:
00506: return 0;
00507:
00508: err_out_unregister:
00509: unregister_netdev(net_dev);
00510: err_unmap_rx:
00511: pci_free_consistent(pci_dev, RX_TOTAL_SIZE, sis_priv->rx_ring,
00512:     sis_priv->rx_ring_dma);
00513: err_unmap_tx:
00514: pci_free_consistent(pci_dev, TX_TOTAL_SIZE, sis_priv->tx_ring,
00515:     sis_priv->tx_ring_dma);
00516: err_out_cleardev:
00517: pci_set_drvdata(pci_dev, NULL);
00518: pci_release_regions(pci_dev);
00519: err_out:
00520: free_netdev(net_dev);
00521: return ret;
00522: } ? end sis900_probe ?
00523:
00524: /**
00525:  * sis900_mii_probe - Probe MII PHY for sis900
00526:  * @net_dev: the net device to probe for
00527:  *
00528:  * Search for total of 32 possible mii phy addresses.
00529:  * Identify and set current phy if found one,
00530:  * return error if it failed to found.
00531:  */
00532:
00533: static int __init sis900_mii_probe(struct net_device * net_dev)
00534: {
00535:     struct sis900_private * sis_priv = net_dev->priv;
00536:     u16 poll_bit = MII_STAT_LINK, status = 0;
00537:     unsigned long timeout = jiffies + 5 * HZ;
00538:     int phy_addr;
00539:     u8 revision;

```

```

00541: sis_priv->mii = NULL;
00542:
00543: /* search for total of 32 possible mii phy addresses */
00544: for (phy_addr = 0; phy_addr < 32; phy_addr++) {
00545:     struct mii_phy * mii_phy = NULL;
00546:     u16 mii_status;
00547:     int i;
00548:
00549:     mii_phy = NULL;
00550:     for(i = 0; i < 2; i++)
00551:         mii_status = mdio_read(net_dev, phy_addr, MII_STATUS);
00552:
00553:     if (mii_status == 0xffff || mii_status == 0x0000)
00554:         /* the mii is not accessible, try next one */
00555:         continue;
00556:
00557:     if ((mii_phy = kmalloc(sizeof(struct mii_phy), GFP_KERNEL)) == NULL) {
00558:         printk(KERN_INFO "Cannot allocate mem for struct mii_phy\n");
00559:         mii_phy = sis_priv->first_mii;
00560:         while (mii_phy) {
00561:             struct mii_phy * phy;
00562:             phy = mii_phy;
00563:             mii_phy = mii_phy->next;
00564:             kfree(phy);
00565:         }
00566:         return 0;
00567:     }
00568:
00569:     mii_phy->phy_id0 = mdio_read(net_dev, phy_addr, MII_PHY_ID0);
00570:     mii_phy->phy_id1 = mdio_read(net_dev, phy_addr, MII_PHY_ID1);
00571:     mii_phy->phy_addr = phy_addr;
00572:     mii_phy->status = mii_status;
00573:     mii_phy->next = sis_priv->mii;
00574:     sis_priv->mii = mii_phy;
00575:     sis_priv->first_mii = mii_phy;
00576:
00577:     for (i = 0; mii_chip_table[i].phy_id1; i++)
00578:         if ((mii_phy->phy_id0 == mii_chip_table[i].phy_id0) &&
00579:             ((mii_phy->phy_id1 & 0xFFFF) == mii_chip_table[i].phy_id1)){
00580:             mii_phy->phy_types = mii_chip_table[i].phy_types;
00581:             if (mii_chip_table[i].phy_types == MIX)
00582:                 mii_phy->phy_types =
00583:                     (mii_status & (MII_STAT_CAN_TX_FDX | MII_STAT_CAN_TX)) ? LAN : HOME;
00584:             printk(KERN_INFO "%s: %s transceiver found at address %d.\n",
00585:                 net_dev->name, mii_chip_table[i].name,
00586:                 phy_addr);
00587:             break;
00588:         }
00589:
00590:     if (! mii_chip_table[i].phy_id1 ) {
00591:         printk(KERN_INFO "%s: Unknown PHY transceiver found at address %d.\n",
00592:             net_dev->name, phy_addr);
00593:         mii_phy->phy_types = UNKNOWN;
00594:     }
00595: } ? end for phy_addr=0; phy_addr<3... ?
00596:
00597: if (sis_priv->mii == NULL) {
00598:     printk(KERN_INFO "%s: No MII transceivers found! \n",
00599:         net_dev->name);
00600:     return 0;
00601: }
00602:
00603: /* select default PHY for mac */
00604: sis_priv->mii = NULL;
00605: sis900_default_phy( net_dev );
00606:
00607: /* Reset phy if default phy is internal sis900 */
00608: if ((sis_priv->mii->phy_id0 == 0x001d) &&
00609:     ((sis_priv->mii->phy_id1 & 0xFFFF) == 0x8000))
00610:     status = sis900_reset_phy(net_dev, sis_priv->cur_phy);
00611:
00612: /* workaround for ICS1893 PHY */
00613: if ((sis_priv->mii->phy_id0 == 0x0015) &&
00614:     ((sis_priv->mii->phy_id1 & 0xFFFF) == 0xF440))
00615:     mdio_write(net_dev, sis_priv->cur_phy, 0x0018, 0xb200);
00616:
00617: if (status & MII_STAT_LINK){
00618:     while (poll_bit) {
00619:         yield();
00620:     }

```

```

00621:         poll_bit ^= (mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS) & poll_bit);
00622:         if (time_after_eq(jiffies, timeout)) {
00623:             printk(KERN_WARNING "%s: reset phy and link down now!\n",
00624:                 net_dev->name);
00625:             return -ETIME;
00626:         }
00627:     }
00628: }
00629:
00630: pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
00631: if (revision == SIS630E_900_REV) {
00632:     /* SIS 630E has some bugs on default value of PHY registers */
00633:     mdio_write(net_dev, sis_priv->cur_phy, MII_ANADV, 0x05e1);
00634:     mdio_write(net_dev, sis_priv->cur_phy, MII_CONFIG1, 0x22);
00635:     mdio_write(net_dev, sis_priv->cur_phy, MII_CONFIG2, 0xff00);
00636:     mdio_write(net_dev, sis_priv->cur_phy, MII_MASK, 0xff00);
00637:     //mdio_write(net_dev, sis_priv->cur_phy, MII_CONTROL, 0x1000);
00638: }
00639:
00640: if (sis_priv->mii->status & MII_STAT_LINK)
00641:     netif_carrier_on(net_dev);
00642: else
00643:     netif_carrier_off(net_dev);
00644:
00645: return 1;
00646: } ? end sis900_mii_probe ?
00647:
00648: /**
00649:  * sis900_default_phy - Select default PHY for sis900 mac.
00650:  * @net_dev: the net device to probe for
00651:  *
00652:  * Select first detected PHY with link as default.
00653:  * If no one is link on, select PHY whose types is HOME as default.
00654:  * If HOME doesn't exist, select LAN.
00655:  */
00656:
00657: static u16 sis900_default_phy(struct net_device * net_dev)
00658: {
00659:     struct sis900_private * sis_priv = net_dev->priv;
00660:     struct mii_phy *phy = NULL, *phy_home = NULL,
00661:         *default_phy = NULL, *phy_lan = NULL;
00662:     u16 status;
00663:
00664:     for (phy=sis_priv->first_mii; phy; phy=phy->next) {
00665:         status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
00666:         status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
00667:
00668:         /* Link ON & Not select default PHY & not ghost PHY */
00669:         if ((status & MII_STAT_LINK) && !default_phy &&
00670:             (phy->phy_types != UNKNOWN))
00671:             default_phy = phy;
00672:         else {
00673:             status = mdio_read(net_dev, phy->phy_addr, MII_CONTROL);
00674:             mdio_write(net_dev, phy->phy_addr, MII_CONTROL,
00675:                 status | MII_CNTL_AUTO | MII_CNTL_ISOLATE);
00676:             if (phy->phy_types == HOME)
00677:                 phy_home = phy;
00678:             else if (phy->phy_types == LAN)
00679:                 phy_lan = phy;
00680:         }
00681:     }
00682:
00683:     if (!default_phy && phy_home)
00684:         default_phy = phy_home;
00685:     else if (!default_phy && phy_lan)
00686:         default_phy = phy_lan;
00687:     else if (!default_phy)
00688:         default_phy = sis_priv->first_mii;
00689:
00690:     if (sis_priv->mii != default_phy) {
00691:         sis_priv->mii = default_phy;
00692:         sis_priv->cur_phy = default_phy->phy_addr;
00693:         printk(KERN_INFO "%s: Using transceiver found at address %d as default!\n",
00694:             net_dev->name, sis_priv->cur_phy);
00695:     }
00696:
00697:     status = mdio_read(net_dev, sis_priv->cur_phy, MII_CONTROL);
00698:     status &= (~MII_CNTL_ISOLATE);
00699:

```

```

00700:     mdio_write(net_dev, sis_priv->cur_phy, MII_CONTROL, status);
00701:     status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
00702:     status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
00703:
00704:     return status;
00705: } ? end sis900_default_phy ?
00706:
00707:
00708: /**
00709:  * sis900_set_capability - set the media capability of network adapter.
00710:  * @net_dev: the net device to probe for
00711:  * @phy: default PHY
00712:  *
00713:  * Set the media capability of network adapter according to
00714:  * mii status register. It's necessary before auto-negotiate.
00715:  */
00716:
00717: static void sis900_set_capability(struct net_device *net_dev, struct mii_phy *phy)
00718: {
00719:     u16 cap;
00720:     u16 status;
00721:
00722:     status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
00723:     status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
00724:
00725:     cap = MII_NWAY_CSMA_CD |
00726:         ((phy->status & MII_STAT_CAN_TX_FDX) ? MII_NWAY_TX_FDX:0) |
00727:         ((phy->status & MII_STAT_CAN_TX) ? MII_NWAY_TX:0) |
00728:         ((phy->status & MII_STAT_CAN_T_FDX) ? MII_NWAY_T_FDX:0) |
00729:         ((phy->status & MII_STAT_CAN_T) ? MII_NWAY_T:0);
00730:
00731:     mdio_write(net_dev, phy->phy_addr, MII_ANADV, cap);
00732: }
00733:
00734:
00735: /* Delay between EEPROM clock transitions. */
00736: #define eeprom_delay() inl(ee_addr)
00737:
00738: /**
00739:  * read_eeprom - Read Serial EEPROM
00740:  * @ioaddr: base i/o address
00741:  * @location: the EEPROM location to read
00742:  *
00743:  * Read Serial EEPROM through EEPROM Access Register.
00744:  * Note that location is in word (16 bits) unit
00745:  */
00746:
00747: static u16 __devinit read_eeprom(long ioaddr, int location)
00748: {
00749:     int i;
00750:     u16 retval = 0;
00751:     long ee_addr = ioaddr + mear;
00752:     u32 read_cmd = location | EERead;
00753:
00754:     outl(0, ee_addr);
00755:     eeprom_delay();
00756:     outl(E ECS, ee_addr);
00757:     eeprom_delay();
00758:
00759:     /* Shift the read command (9) bits out. */
00760:     for (i = 8; i >= 0; i--) {
00761:         u32 dataval = (read_cmd & (1 << i)) ? EEDI | EECS : EECS;
00762:         outl(dataval, ee_addr);
00763:         eeprom_delay();
00764:         outl(dataval | EECLK, ee_addr);
00765:         eeprom_delay();
00766:     }
00767:     outl(E ECS, ee_addr);
00768:     eeprom_delay();
00769:
00770:     /* read the 16-bits data in */
00771:     for (i = 16; i > 0; i--) {
00772:         outl(E ECS, ee_addr);
00773:         eeprom_delay();
00774:         outl(E ECS | EECLK, ee_addr);
00775:         eeprom_delay();
00776:         retval = (retval << 1) | ((inl(ee_addr) & EEDO) ? 1 : 0);
00777:         eeprom_delay();
00778:     }
00779:
00780:     /* Terminate the EEPROM access. */

```



```

00781:     outl(0, ee_addr);
00782:     eeprom_delay();
00783:
00784:     return (retval);
00785: } ? end read_eeprom ?
00786:
00787: /* Read and write the MII management registers using software-generated
00788:    serial MDIO protocol. Note that the command bits and data bits are
00789:    send out separately */
00790: #define mdio_delay()    inl(mdio_addr)
00791:
00792: static void mdio_idle(long mdio_addr)
00793: {
00794:     outl(MDIO | MDDIR, mdio_addr);
00795:     mdio_delay();
00796:     outl(MDIO | MDDIR | MDC, mdio_addr);
00797: }
00798:
00799: /* Synchronize the MII management interface by shifting 32 one bits out. */
00800: static void mdio_reset(long mdio_addr)
00801: {
00802:     int i;
00803:
00804:     for (i = 31; i >= 0; i--) {
00805:         outl(MDDIR | MDIO, mdio_addr);
00806:         mdio_delay();
00807:         outl(MDDIR | MDIO | MDC, mdio_addr);
00808:         mdio_delay();
00809:     }
00810:     return;
00811: }
00812:
00813: /**
00814:  * mdio_read - read MII PHY register
00815:  * @net_dev: the net device to read
00816:  * @phy_id: the phy address to read
00817:  * @location: the phy register id to read
00818:  *
00819:  * Read MII registers through MDIO and MDC
00820:  * using MDIO management frame structure and protocol(defined by ISO/ IEC).
00821:  * Please see Sis7014 or ICS spec
00822:  */
00823:
00824: static u16 mdio_read(struct net_device *net_dev, int phy_id, int location)
00825: {
00826:     long mdio_addr = net_dev->base_addr + mear;
00827:     int mii_cmd = MIRead | (phy_id << MIIPmdShift) | (location << MIRegShift);
00828:     u16 retval = 0;
00829:     int i;
00830:
00831:     mdio_reset(mdio_addr);
00832:     mdio_idle(mdio_addr);
00833:
00834:     for (i = 15; i >= 0; i--) {
00835:         int dataval = (mii_cmd & (1 << i)) ? MDDIR | MDIO : MDDIR;
00836:         outl(dataval, mdio_addr);
00837:         mdio_delay();
00838:         outl(dataval | MDC, mdio_addr);
00839:         mdio_delay();
00840:     }
00841:
00842:     /* Read the 16 data bits. */
00843:     for (i = 16; i > 0; i--) {
00844:         outl(0, mdio_addr);
00845:         mdio_delay();
00846:         retval = (retval << 1) | ((inl(mdio_addr) & MDIO) ? 1 : 0);
00847:         outl(MDC, mdio_addr);
00848:         mdio_delay();
00849:     }
00850:     outl(0x00, mdio_addr);
00851:
00852:     return retval;
00853: } ? end mdio_read ?
00854:
00855: /**
00856:  * mdio_write - write MII PHY register
00857:  * @net_dev: the net device to write
00858:  * @phy_id: the phy address to write
00859:  * @location: the phy register id to write
00860:  * @value: the register value to write with
00861:  */

```

```

00862:  * Write MII registers with @value through MDIO and MDC
00863:  * using MDIO management frame structure and protocol(defined by ISO/ IEC)
00864:  * please see Sis7014 or ICS spec
00865:  */
00866:
00867: static void mdio_write(struct net_device *net_dev, int phy_id, int location,
00868:     int value)
00869: {
00870:     long mdio_addr = net_dev->base_addr + mear;
00871:     int mii_cmd = MIWrite | (phy_id << MIIPmdShift) | (location << MIRegShift);
00872:     int i;
00873:
00874:     mdio_reset(mdio_addr);
00875:     mdio_idle(mdio_addr);
00876:
00877:     /* Shift the command bits out. */
00878:     for (i = 15; i >= 0; i--) {
00879:         int dataval = (mii_cmd & (1 << i)) ? MDDIR | MDIO : MDDIR;
00880:         outb(dataval, mdio_addr);
00881:         mdio_delay();
00882:         outb(dataval | MDC, mdio_addr);
00883:         mdio_delay();
00884:     }
00885:     mdio_delay();
00886:
00887:     /* Shift the value bits out. */
00888:     for (i = 15; i >= 0; i--) {
00889:         int dataval = (value & (1 << i)) ? MDDIR | MDIO : MDDIR;
00890:         outl(dataval, mdio_addr);
00891:         mdio_delay();
00892:         outl(dataval | MDC, mdio_addr);
00893:         mdio_delay();
00894:     }
00895:     mdio_delay();
00896:
00897:     /* Clear out extra bits. */
00898:     for (i = 2; i > 0; i--) {
00899:         outb(0, mdio_addr);
00900:         mdio_delay();
00901:         outb(MDC, mdio_addr);
00902:         mdio_delay();
00903:     }
00904:     outl(0x00, mdio_addr);
00905:
00906:     return;
00907: } ? end mdio_write ?
00908:
00909:
00910: /**
00911:  * sis900_reset_phy - reset sis900 mii phy.
00912:  * @net_dev: the net device to write
00913:  * @phy_addr: default phy address
00914:  *
00915:  * Some specific phy can't work properly without reset.
00916:  * This function will be called during initialization and
00917:  * link status change from ON to DOWN.
00918:  */
00919:
00920: static u16 sis900_reset_phy(struct net_device *net_dev, int phy_addr)
00921: {
00922:     int i = 0;
00923:     u16 status;
00924:
00925:     while (i++ < 2)
00926:         status = mdio_read(net_dev, phy_addr, MII_STATUS);
00927:
00928:     mdio_write(net_dev, phy_addr, MII_CONTROL, MII_CNTL_RESET);
00929:
00930:     return status;
00931: }
00932:
00933: /**
00934:  * sis900_open - open sis900 device
00935:  * @net_dev: the net device to open
00936:  *
00937:  * Do some initialization and start net interface.
00938:  * enable interrupts and set sis900 timer.
00939:  */
00940:
00941: static int

```

```

00942: sis900_open(struct net_device *net_dev)
00943: {
00944:     struct sis900_private *sis_priv = net_dev->priv;
00945:     long loaddr = net_dev->base_addr;
00946:     u8 revision;
00947:     int ret;
00948:
00949:     /* Soft reset the chip. */
00950:     sis900_reset(net_dev);
00951:
00952:     /* Equalizer workaround Rule */
00953:     pcl_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
00954:     sis630_set_eq(net_dev, revision);
00955:
00956:     ret = request_irq(net_dev->irq, &sis900_interrupt, SA_SHIRO,
00957:                      net_dev->name, net_dev);
00958:     if (ret)
00959:         return ret;
00960:
00961:     sis900_init_rxfilter(net_dev);
00962:
00963:     sis900_init_tx_ring(net_dev);
00964:     sis900_init_rx_ring(net_dev);
00965:
00966:     set_rx_mode(net_dev);
00967:
00968:     netif_start_queue(net_dev);
00969:
00970:     /* Workaround for EDB */
00971:     sis900_set_mode(loaddr, HW_SPEED_10_MBPS, FDX_CAPABLE_HALF_SELECTED);
00972:
00973:     /* Enable all known interrupts by setting the interrupt mask. */
00974:     outl((RxSOVR| RxORN| RxERR| RxOK| TxURN| TxERR| TxIDLE), loaddr + imr);
00975:     outl(RxENA | inl(loaddr + cr), loaddr + cr);
00976:     outl(IE, loaddr + ier);
00977:
00978:     sis900_check_mode(net_dev, sis_priv->mii);
00979:
00980:     /* Set the timer to switch to check for link beat and perhaps switch
00981:        to an alternate media type. */
00982:     init_timer(&sis_priv->timer);
00983:     sis_priv->timer.expires = jiffies + HZ;
00984:     sis_priv->timer.data = (unsigned long)net_dev;
00985:     sis_priv->timer.function = &sis900_timer;
00986:     add_timer(&sis_priv->timer);
00987:
00988:     return 0;
00989: } ? end sis900_open ?
00990:
00991: /**
00992:  * sis900_init_rxfilter - Initialize the Rx filter
00993:  * @net_dev: the net device to initialize for
00994:  *
00995:  * Set receive filter address to our MAC address
00996:  * and enable packet filtering.
00997:  */
00998:
00999: static void
01000: sis900_init_rxfilter (struct net_device * net_dev)
01001: {
01002:     long loaddr = net_dev->base_addr;
01003:     u32 rfcSave;
01004:     u32 i;
01005:
01006:     rfcSave = inl(rfcr + loaddr);
01007:
01008:     /* disable packet filtering before setting filter */
01009:     outl(rfcrSave & ~RFEN, rfcr + loaddr);
01010:
01011:     /* load MAC addr to filter data register */
01012:     for (i = 0 ; i < 3 ; i++) {
01013:         u32 w;
01014:
01015:         w = (u32) * ((u16 *) (net_dev->dev_addr) + i);
01016:         outl((i << RFADDR_shift), loaddr + rfcr);
01017:         outl(w, loaddr + rfd);
01018:     }

```

```

01019:         if (sis900_debug > 2) {
01020:             printk(KERN_INFO "%s: Receive Filter Addrss[%d]=%x\n",
01021:                  net_dev->name, i, inl(loaddr + rfd));
01022:         }
01023:     }
01024:
01025:     /* enable packet filtering */
01026:     outl(rfcrSave | RFEN, rfcr + loaddr);
01027: } ? end sis900_init_rxfilter ?
01028:
01029: /**
01030:  * sis900_init_tx_ring - Initialize the Tx descriptor ring
01031:  * @net_dev: the net device to initialize for
01032:  *
01033:  * Initialize the Tx descriptor ring,
01034:  */
01035:
01036: static void
01037: sis900_init_tx_ring(struct net_device *net_dev)
01038: {
01039:     struct sis900_private *sis_priv = net_dev->priv;
01040:     long loaddr = net_dev->base_addr;
01041:     int i;
01042:
01043:     sis_priv->tx_full = 0;
01044:     sis_priv->dirty_tx = sis_priv->cur_tx = 0;
01045:
01046:     for (i = 0; i < NUM_TX_DESC; i++) {
01047:         sis_priv->tx_skbuff[i] = NULL;
01048:
01049:         sis_priv->tx_ring[i].link = sis_priv->tx_ring_dma +
01050:             ((i+1)%NUM_TX_DESC)*sizeof(BufferDesc);
01051:         sis_priv->tx_ring[i].cmdsts = 0;
01052:         sis_priv->tx_ring[i].bufptr = 0;
01053:     }
01054:
01055:     /* load Transmit Descriptor Register */
01056:     outl(sis_priv->tx_ring_dma, loaddr + txdp);
01057:     if (sis900_debug > 2)
01058:         printk(KERN_INFO "%s: TX descriptor register loaded with: %8.8x\n",
01059:              net_dev->name, inl(loaddr + txdp));
01060: } ? end sis900_init_tx_ring ?
01061:
01062: /**
01063:  * sis900_init_rx_ring - Initialize the Rx descriptor ring
01064:  * @net_dev: the net device to initialize for
01065:  *
01066:  * Initialize the Rx descriptor ring,
01067:  * and pre-allocate receive buffers (socket buffer)
01068:  */
01069:
01070: static void
01071: sis900_init_rx_ring(struct net_device *net_dev)
01072: {
01073:     struct sis900_private *sis_priv = net_dev->priv;
01074:     long loaddr = net_dev->base_addr;
01075:     int i;
01076:
01077:     sis_priv->cur_rx = 0;
01078:     sis_priv->dirty_rx = 0;
01079:
01080:     /* init RX descriptor */
01081:     for (i = 0; i < NUM_RX_DESC; i++) {
01082:         sis_priv->rx_skbuff[i] = NULL;
01083:
01084:         sis_priv->rx_ring[i].link = sis_priv->rx_ring_dma +
01085:             ((i+1)%NUM_RX_DESC)*sizeof(BufferDesc);
01086:         sis_priv->rx_ring[i].cmdsts = 0;
01087:         sis_priv->rx_ring[i].bufptr = 0;
01088:     }
01089:
01090:     /* allocate sock buffers */
01091:     for (i = 0; i < NUM_RX_DESC; i++) {
01092:         struct sk_buff *skb;
01093:
01094:         if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
01095:             /* not enough memory for skbuff, this makes a "hole"
01096:                on the buffer ring, it is not clear how the
01097:                hardware will react to this kind of degenerated
01098:                buffer */

```



```

01099:         break;
01100:     }
01101:     skb->dev = net_dev;
01102:     sis_priv->rx_skbuff[i] = skb;
01103:     sis_priv->rx_ring[i].cmdsts = RX_BUF_SIZE;
01104:     sis_priv->rx_ring[i].bufptr = pci_map_single(sis_priv->pci_dev,
01105:         skb->tail, RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
01106: }
01107: sis_priv->dirty_rx = (unsigned int) (i - NUM_RX_DESC);
01108:
01109: /* Load Receive Descriptor Register */
01110: outl(sis_priv->rx_ring_dma, loadaddr + rxdp);
01111: if (sis900_debug > 2)
01112:     printk(KERN_INFO "%s: RX descriptor register loaded with: %8.8x\n",
01113:         net_dev->name, inl(loadaddr + rxdp));
01114: } ? end sis900_init_rx_ring ?
01115:
01116: /**
01117:  * sis630_set_eq - set phy equalizer value for 630 LAN
01118:  * @net_dev: the net device to set equalizer value
01119:  * @revision: 630 LAN revision number
01120:  *
01121:  * 630E equalizer workaround rule(Cyrus Huang 08/15)
01122:  * PHY register 14h(Test)
01123:  * Bit 14: 0 -- Automatically detect (default)
01124:  *          1 -- Manually set Equalizer filter
01125:  * Bit 13: 0 -- (Default)
01126:  *          1 -- Speed up convergence of equalizer setting
01127:  * Bit 9 : 0 -- (Default)
01128:  *          1 -- Disable Baseline Wander
01129:  * Bit 3~7 -- Equalizer filter setting
01130:  * Link ON: Set Bit 9, 13 to 1, Bit 14 to 0
01131:  * Then calculate equalizer value
01132:  * Then set equalizer value, and set Bit 14 to 1, Bit 9 to 0
01133:  * Link Off: Set Bit 13 to 1, Bit 14 to 0
01134:  * Calculate Equalizer value:
01135:  * When Link is ON and Bit 14 is 0, SIS900PHY will auto- detect proper equalizer value.
01136:  * When the equalizer is stable, this value is not a fixed value. It will be within
01137:  * a small range(eg. 7~9). Then we get a minimum and a maximum value(eg. min=7, max=9)
01138:  * 0 <= max <= 4 --> set equalizer to max
01139:  * 5 <= max <= 14 --> set equalizer to max+1 or set equalizer to max+2 if max == min
01140:  * max >= 15 --> set equalizer to max+5 or set equalizer to max+6 if max == min
01141:  */
01142:
01143: static void sis630_set_eq(struct net_device *net_dev, u8 revision)
01144: {
01145:     struct sis900_private *sis_priv = net_dev->priv;
01146:     u16 reg14h, eq_value=0, max_value=0, min_value=0;
01147:     int i, maxcount=10;
01148:
01149:     if (! (revision == SIS630E_900_REV || revision == SIS630EA1_900_REV ||
01150:         revision == SIS630A_900_REV || revision == SIS630ET_900_REV) )
01151:         return;
01152:
01153:     if (netif_carrier_ok(net_dev)) {
01154:         reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
01155:         mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
01156:             (0x2200 | reg14h) & 0xBFFF);
01157:         for (i=0; i < maxcount; i++) {
01158:             eq_value = (0x00F8 & mdio_read(net_dev,
01159:                 sis_priv->cur_phy, MII_RESV)) >> 3;
01160:             if (i == 0)
01161:                 max_value=min_value=eq_value;
01162:             max_value = (eq_value > max_value) ?
01163:                 eq_value : max_value;
01164:             min_value = (eq_value < min_value) ?
01165:                 eq_value : min_value;
01166:         }
01167:         /* 630E rule to determine the equalizer value */
01168:         if (revision == SIS630E_900_REV || revision == SIS630EA1_900_REV ||
01169:             revision == SIS630ET_900_REV) {
01170:             if (max_value < 5)
01171:                 eq_value = max_value;
01172:             else if (max_value >= 5 && max_value < 15)
01173:                 eq_value = (max_value == min_value) ?
01174:                     max_value+2 : max_value+1;
01175:             else if (max_value >= 15)
01176:                 eq_value=(max_value == min_value) ?
01177:                     max_value+6 : max_value+5;
01178:         }
01179:         /* 630B0&B1 rule to determine the equalizer value */

```

```

01180:         if (revision == SIS630A_900_REV &&
01181:             (sis_priv->host_bridge_rev == SIS630B0 ||
01182:              sis_priv->host_bridge_rev == SIS630B1)) {
01183:             if (max_value == 0)
01184:                 eq_value = 3;
01185:             else
01186:                 eq_value = (max_value + min_value + 1) / 2;
01187:         }
01188:         /* write equalizer value and setting */
01189:         reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
01190:         reg14h = (reg14h & 0xFF07) | ((eq_value << 3) & 0x00F8);
01191:         reg14h = (reg14h | 0x6000) & 0xFDFE;
01192:         mdio_write(net_dev, sis_priv->cur_phy, MII_RESV, reg14h);
01193:     } else {
01194:         reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
01195:         if (revision == SIS630A_900_REV &&
01196:             (sis_priv->host_bridge_rev == SIS630B0 ||
01197:              sis_priv->host_bridge_rev == SIS630B1))
01198:             mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
01199:                 (reg14h | 0x2200) & 0xBFFF);
01200:         else
01201:             mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
01202:                 (reg14h | 0x2000) & 0xBFFF);
01203:     }
01204:     return;
01205: } ? end sis630_set_eq ?
01206:
01207: /**
01208:  * sis900_timer - sis900 timer routine
01209:  * @data: pointer to sis900 net device
01210:  *
01211:  * On each timer ticks we check two things,
01212:  * link status (ON/ OFF) and link mode (10/ 100/ Full/ Half)
01213:  */
01214:
01215: static void sis900_timer(unsigned long data)
01216: {
01217:     struct net_device *net_dev = (struct net_device *)data;
01218:     struct sis900_private *sis_priv = net_dev->priv;
01219:     struct mii_phy *mii_phy = sis_priv->mii;
01220:     static int next_tick = 5*HZ;
01221:     u16 status;
01222:     u8 revision;
01223:
01224:     if (! sis_priv->autong_complete){
01225:         int speed, duplex = 0;
01226:
01227:         sis900_read_mode(net_dev, &speed, &duplex);
01228:         if (duplex){
01229:             sis900_set_mode(net_dev->base_addr, speed, duplex);
01230:             pci_read_config_byte(sis_priv->pci_dev,
01231:                 PCI_CLASS_REVISION, &revision);
01232:             sis630_set_eq(net_dev, revision);
01233:             netif_start_queue(net_dev);
01234:         }
01235:
01236:         sis_priv->timer.expires = jiffies + HZ;
01237:         add_timer(&sis_priv->timer);
01238:         return;
01239:     }
01240:
01241:     status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
01242:     status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
01243:
01244:     /* Link OFF -> ON */
01245:     if (! netif_carrier_ok(net_dev)) {
01246:         LookForLink:
01247:         /* Search for new PHY */
01248:         status = sis900_default_phy(net_dev);
01249:         mii_phy = sis_priv->mii;
01250:
01251:         if (status & MII_STAT_LINK){
01252:             sis900_check_mode(net_dev, mii_phy);
01253:             netif_carrier_on(net_dev);
01254:         }
01255:     } else {
01256:         /* Link ON -> OFF */

```

```

01257:     if (! (status & MII_STAT_LINK)){
01258:         netif_carrier_off(net_dev);
01259:         printk(KERN_INFO "%s: Media Link Off\n", net_dev->name);
01260:
01261:         /* Change mode issue */
01262:         if ((mii_phy->phy_id0 == 0x001D) &&
01263:             ((mii_phy->phy_id1 & 0xFFFF0) == 0x8000))
01264:             sis900_reset_phy(net_dev, sis_priv->cur_phy);
01265:
01266:         pci_read_config_byte(sis_priv->pci_dev,
01267:                             PCI_CLASS_REVISION, &revision);
01268:         sis630_set_eq(net_dev, revision);
01269:
01270:         goto ↑LookForLink;
01271:     }
01272: }
01273:
01274:     sis_priv->timer.expires = jiffies + next_tick;
01275:     add_timer(&sis_priv->timer);
01276: } ? end sis900_timer ?
01277:
01278: /**
01279:  * sis900_check_mode - check the media mode for sis900
01280:  * @net_dev: the net device to be checked
01281:  * @mii_phy: the mii phy
01282:  *
01283:  * Older driver gets the media mode from mii status output
01284:  * register. Now we set our media capability and auto-negotiate
01285:  * to get the upper bound of speed and duplex between two ends.
01286:  * If the types of mii phy is HOME, it doesn't need to auto-negotiate
01287:  * and autong_complete should be set to 1.
01288:  */
01289:
01290: static void sis900_check_mode(struct net_device *net_dev, struct mii_phy *mii_phy)
01291: {
01292:     struct sis900_private *sis_priv = net_dev->priv;
01293:     long ioaddr = net_dev->base_addr;
01294:     int speed, duplex;
01295:
01296:     if (mii_phy->phy_types == LAN) {
01297:         outl(~EXD & inl(ioaddr + cfg), ioaddr + cfg);
01298:         sis900_set_capability(net_dev, mii_phy);
01299:         sis900_auto_negotiate(net_dev, sis_priv->cur_phy);
01300:     } else {
01301:         outl(EXD | inl(ioaddr + cfg), ioaddr + cfg);
01302:         speed = HW_SPEED_HOME;
01303:         duplex = FDX_CAPABLE_HALF_SELECTED;
01304:         sis900_set_mode(ioaddr, speed, duplex);
01305:         sis_priv->autong_complete = 1;
01306:     }
01307: }
01308:
01309: /**
01310:  * sis900_set_mode - Set the media mode of mac register.
01311:  * @ioaddr: the address of the device
01312:  * @speed : the transmit speed to be determined
01313:  * @duplex: the duplex mode to be determined
01314:  *
01315:  * Set the media mode of mac register txcfg/rxcfg according to
01316:  * speed and duplex of phy. Bit EDB_MASTER_EN indicates the EDB
01317:  * bus is used instead of PCI bus. When this bit is set 1, the
01318:  * Max DMA Burst Size for TX/RX DMA should be no larger than 16
01319:  * double words.
01320:  */
01321:
01322: static void sis900_set_mode(long ioaddr, int speed, int duplex)
01323: {
01324:     u32 tx_flags = 0, rx_flags = 0;
01325:
01326:     if (inl(ioaddr + cfg) & EDB_MASTER_EN) {
01327:         tx_flags = TxATP | (DMA_BURST_64 << TxMXDMA_shift) |
01328:             (TX_FILL_THRESH << TxFILLT_shift);
01329:         rx_flags = DMA_BURST_64 << RxMXDMA_shift;
01330:     } else {
01331:         tx_flags = TxATP | (DMA_BURST_512 << TxMXDMA_shift) |
01332:             (TX_FILL_THRESH << TxFILLT_shift);
01333:         rx_flags = DMA_BURST_512 << RxMXDMA_shift;
01334:     }
01335: }

```

```

01336:     if (speed == HW_SPEED_HOME | | speed == HW_SPEED_10_MBPS) {
01337:         rx_flags |= (RxDRNT_10 << RxDRNT_shift);
01338:         tx_flags |= (TxDRNT_10 << TxDRNT_shift);
01339:     } else {
01340:         rx_flags |= (RxDRNT_100 << RxDRNT_shift);
01341:         tx_flags |= (TxDRNT_100 << TxDRNT_shift);
01342:     }
01343:
01344:     if (duplex == FDX_CAPABLE_FULL_SELECTED) {
01345:         tx_flags |= (TxCSI | TxHBI);
01346:         rx_flags |= RxATX;
01347:     }
01348:
01349:     outl(tx_flags, ioaddr + txcfg);
01350:     outl(rx_flags, ioaddr + rxcfg);
01351: } ? end sis900_set_mode ?
01352:
01353: /**
01354:  * sis900_auto_negotiate - Set the Auto-Negotiation Enable/Reset bit.
01355:  * @net_dev: the net device to read mode for
01356:  * @phy_addr: mii phy address
01357:  *
01358:  * If the adapter is link-on, set the auto-negotiate enable/reset bit.
01359:  * autong_complete should be set to 0 when starting auto-negotiation.
01360:  * autong_complete should be set to 1 if we didn't start auto-negotiation.
01361:  * sis900_timer will wait for link on again if autong_complete = 0.
01362:  */
01363:
01364: static void sis900_auto_negotiate(struct net_device *net_dev, int phy_addr)
01365: {
01366:     struct sis900_private *sis_priv = net_dev->priv;
01367:     int i = 0;
01368:     u32 status;
01369:
01370:     while (i++ < 2)
01371:         status = mdio_read(net_dev, phy_addr, MII_STATUS);
01372:
01373:     if (! (status & MII_STAT_LINK)){
01374:         printk(KERN_INFO "%s: Media Link Off\n", net_dev->name);
01375:         sis_priv->autong_complete = 1;
01376:         netif_carrier_off(net_dev);
01377:         return;
01378:     }
01379:
01380:     /* (Re)start AutoNegotiate */
01381:     mdio_write(net_dev, phy_addr, MII_CONTROL,
01382:               MII_CNTL_AUTO | MII_CNTL_RST_AUTO);
01383:     sis_priv->autong_complete = 0;
01384: } ? end sis900_auto_negotiate ?
01385:
01386:
01387: /**
01388:  * sis900_read_mode - read media mode for sis900 internal phy
01389:  * @net_dev: the net device to read mode for
01390:  * @speed : the transmit speed to be determined
01391:  * @duplex : the duplex mode to be determined
01392:  *
01393:  * The capability of remote end will be put in mii register autorec
01394:  * after auto-negotiation. Use AND operation to get the upper bound
01395:  * of speed and duplex between two ends.
01396:  */
01397:
01398: static void sis900_read_mode(struct net_device *net_dev, int *speed, int *duplex)
01399: {
01400:     struct sis900_private *sis_priv = net_dev->priv;
01401:     struct mii_phy *phy = sis_priv->mii;
01402:     int phy_addr = sis_priv->cur_phy;
01403:     u32 status;
01404:     u16 autoadv, autorec;
01405:     int i = 0;
01406:
01407:     while (i++ < 2)
01408:         status = mdio_read(net_dev, phy_addr, MII_STATUS);
01409:
01410:     if (! (status & MII_STAT_LINK))
01411:         return;
01412:
01413:     /* AutoNegotiate completed */

```

```

01414:     autoadv = mdio_read(net_dev, phy_addr, MII_ANADV);
01415:     autorec = mdio_read(net_dev, phy_addr, MII_ANLPAR);
01416:     status = autoadv & autorec;
01417:
01418:     *speed = HW_SPEED_10_MBPS;
01419:     *duplex = FDX_CAPABLE_HALF_SELECTED;
01420:
01421:     if (status & (MII_NWAY_TX | MII_NWAY_TX_FDX))
01422:         *speed = HW_SPEED_100_MBPS;
01423:     if (status & (MII_NWAY_TX_FDX | MII_NWAY_T_FDX))
01424:         *duplex = FDX_CAPABLE_FULL_SELECTED;
01425:
01426:     sis_priv->autong_complete = 1;
01427:
01428:     /* Workaround for Realtek RTL8201 PHY issue */
01429:     if ((phy->phy_id0 == 0x0000) && ((phy->phy_id1 & 0xFFFF0) == 0x8200)) {
01430:         if (mdio_read(net_dev, phy_addr, MII_CONTROL) & MII_CNTL_FDX)
01431:             *duplex = FDX_CAPABLE_FULL_SELECTED;
01432:         if (mdio_read(net_dev, phy_addr, 0x0019) & 0x01)
01433:             *speed = HW_SPEED_100_MBPS;
01434:     }
01435:
01436:     printk(KERN_INFO "%s: Media Link On %s %s- duplex \n",
01437:            net_dev->name,
01438:            *speed == HW_SPEED_100_MBPS ?
01439:            "100mbps" : "10mbps",
01440:            *duplex == FDX_CAPABLE_FULL_SELECTED ?
01441:            "full" : "half");
01442: } ? end sis900_read_mode ?
01443:
01444: /**
01445:  * sis900_tx_timeout - sis900 transmit timeout routine
01446:  * @net_dev: the net device to transmit
01447:  *
01448:  * print transmit timeout status
01449:  * disable interrupts and do some tasks
01450:  */
01451:
01452: static void sis900_tx_timeout(struct net_device *net_dev)
01453: {
01454:     struct sis900_private *sis_priv = net_dev->priv;
01455:     long loaddr = net_dev->base_addr;
01456:     unsigned long flags;
01457:     int i;
01458:
01459:     printk(KERN_INFO "%s: Transmit timeout, status %8.8x %8.8x \n",
01460:            net_dev->name, inl(loaddr + cr), inl(loaddr + isr));
01461:
01462:     /* Disable interrupts by clearing the interrupt mask. */
01463:     outl(0x0000, loaddr + imr);
01464:
01465:     /* use spinlock to prevent interrupt handler accessing buffer ring */
01466:     spin_lock_irqsave(&sis_priv->lock, flags);
01467:
01468:     /* discard unsent packets */
01469:     sis_priv->dirty_tx = sis_priv->cur_tx = 0;
01470:     for (i = 0; i < NUM_TX_DESC; i++) {
01471:         struct sk_buff *skb = sis_priv->tx_skbuff[i];
01472:
01473:         if (skb) {
01474:             pci_unmap_single(sis_priv->pci_dev,
01475:                             sis_priv->tx_ring[i].bufptr, skb->len,
01476:                             PCI_DMA_TODEVICE);
01477:             dev_kfree_skb_irq(skb);
01478:             sis_priv->tx_skbuff[i] = NULL;
01479:             sis_priv->tx_ring[i].cmdsts = 0;
01480:             sis_priv->tx_ring[i].bufptr = 0;
01481:             sis_priv->stats.tx_dropped++;
01482:         }
01483:     }
01484:     sis_priv->tx_full = 0;
01485:     netif_wake_queue(net_dev);
01486:
01487:     spin_unlock_irqrestore(&sis_priv->lock, flags);
01488:
01489:     net_dev->trans_start = jiffies;
01490:
01491:     /* load Transmit Descriptor Register */
01492:     outl(sis_priv->tx_ring_dma, loaddr + txdp);
01493:
01494:     /* Enable all known interrupts by setting the interrupt mask. */

```

```

01495:     outl((RxSOVR| RxORN| RxERR| RxOK| TxURN| TxERR| TxIDLE), loaddr + imr);
01496:     return;
01497: } ? end sis900_tx_timeout ?
01498:
01499: /**
01500:  * sis900_start_xmit - sis900 start transmit routine
01501:  * @skb: socket buffer pointer to put the data being transmitted
01502:  * @net_dev: the net device to transmit with
01503:  *
01504:  * Set the transmit buffer descriptor,
01505:  * and write TxENA to enable transmit state machine.
01506:  * tell upper layer if the buffer is full
01507:  */
01508:
01509: static int
01510: sis900_start_xmit(struct sk_buff *skb, struct net_device *net_dev)
01511: {
01512:     struct sis900_private *sis_priv = net_dev->priv;
01513:     long loaddr = net_dev->base_addr;
01514:     unsigned int entry;
01515:     unsigned long flags;
01516:     unsigned int index_cur_tx, index_dirty_tx;
01517:     unsigned int count_dirty_tx;
01518:
01519:     /* Don't transmit data before the complete of auto-negotiation */
01520:     if(! sis_priv->autong_complete){
01521:         netif_stop_queue(net_dev);
01522:         return 1;
01523:     }
01524:
01525:     spin_lock_irqsave(&sis_priv->lock, flags);
01526:
01527:     /* Calculate the next Tx descriptor entry. */
01528:     entry = sis_priv->cur_tx % NUM_TX_DESC;
01529:     sis_priv->tx_skbuff[entry] = skb;
01530:
01531:     /* set the transmit buffer descriptor and enable Transmit State Machine */
01532:     sis_priv->tx_ring[entry].bufptr = pci_map_single(sis_priv->pci_dev,
01533:             skb->data, skb->len, PCI_DMA_TODEVICE);
01534:     sis_priv->tx_ring[entry].cmdsts = (OWN | skb->len);
01535:     outl(TxENA | inl(loaddr + cr), loaddr + cr);
01536:
01537:     sis_priv->cur_tx++;
01538:     index_cur_tx = sis_priv->cur_tx;
01539:     index_dirty_tx = sis_priv->dirty_tx;
01540:
01541:     for (count_dirty_tx = 0; index_cur_tx != index_dirty_tx; index_dirty_tx++)
01542:         count_dirty_tx++;
01543:
01544:     if (index_cur_tx == index_dirty_tx) {
01545:         /* dirty_tx is met in the cycle of cur_tx, buffer full */
01546:         sis_priv->tx_full = 1;
01547:         netif_stop_queue(net_dev);
01548:     } else if (count_dirty_tx < NUM_TX_DESC) {
01549:         /* Typical path, tell upper layer that more transmission is possible */
01550:         netif_start_queue(net_dev);
01551:     } else {
01552:         /* buffer full, tell upper layer no more transmission */
01553:         sis_priv->tx_full = 1;
01554:         netif_stop_queue(net_dev);
01555:     }
01556:
01557:     spin_unlock_irqrestore(&sis_priv->lock, flags);
01558:
01559:     net_dev->trans_start = jiffies;
01560:
01561:     if (sis900_debug > 3)
01562:         printk(KERN_INFO "%s: Queued Tx packet at %p size %d "
01563:                "to slot %d.\n",
01564:                net_dev->name, skb->data, (int)skb->len, entry);
01565:
01566:     return 0;
01567: } ? end sis900_start_xmit ?
01568:
01569: /**
01570:  * sis900_interrupt - sis900 interrupt handler
01571:  * @irq: the irq number
01572:  * @dev_instance: the client data object
01573:  * @regs: snapshot of processor context
01574:  *
01575:  * The interrupt handler does all of the Rx thread work,

```

```

01576:  *   and cleans up after the Tx thread
01577:  */
01578:
01579: static irqreturn_t sis900_interrupt(int irq, void *dev_instance, struct pt_regs *regs)
01580: {
01581:     struct net_device *net_dev = dev_instance;
01582:     struct sis900_private *sis_priv = net_dev->priv;
01583:     int boguscnt = max_interrupt_work;
01584:     long loaddr = net_dev->base_addr;
01585:     u32 status;
01586:     unsigned int handled = 0;
01587:
01588:     spin_lock (&sis_priv->lock);
01589:
01590:     do {
01591:         status = inl(loaddr + isr);
01592:
01593:         if ((status & (HIBERR|TxURN|TxERR|TxIDLE|RxORN|RxEERR|RxOK)) == 0)
01594:             /* nothing intresting happened */
01595:             break;
01596:         handled = 1;
01597:
01598:         /* why don't we break after Tx/Rx case ?? keyword: full-duplex */
01599:         if (status & (RxORN | RxERR | RxOK))
01600:             /* Rx interrupt */
01601:             sis900_rx(net_dev);
01602:
01603:         if (status & (TxURN | TxERR | TxIDLE))
01604:             /* Tx interrupt */
01605:             sis900_finish_xmit(net_dev);
01606:
01607:         /* something strange happened !!! */
01608:         if (status & HIBERR) {
01609:             printk(KERN_INFO "%s: Abnormal interrupt, "
01610:                 "status %#8.8x.\n", net_dev->name, status);
01611:             break;
01612:         }
01613:         if (--boguscnt < 0) {
01614:             printk(KERN_INFO "%s: Too much work at interrupt, "
01615:                 "interrupt status = %#8.8x.\n",
01616:                 net_dev->name, status);
01617:             break;
01618:         }
01619:     } ? end do ? while (1);
01620:
01621:     if (sis900_debug > 3)
01622:         printk(KERN_INFO "%s: exiting interrupt, "
01623:             "interrupt status = 0x%#8.8x.\n",
01624:             net_dev->name, inl(loaddr + isr));
01625:
01626:     spin_unlock (&sis_priv->lock);
01627:     return IRQ_RETVAL(handled);
01628: } ? end sis900_interrupt ?
01629:
01630: /**
01631:  * sis900_rx - sis900 receive routine
01632:  * @net_dev: the net device which receives data
01633:  *
01634:  * Process receive interrupt events,
01635:  * put buffer to higher layer and refill buffer pool
01636:  * Note: This fuction is called by interrupt handler,
01637:  * don't do "too much" work here
01638:  */
01639:
01640: static int sis900_rx(struct net_device *net_dev)
01641: {
01642:     struct sis900_private *sis_priv = net_dev->priv;
01643:     long loaddr = net_dev->base_addr;
01644:     unsigned int entry = sis_priv->cur_rx % NUM_RX_DESC;
01645:     u32 rx_status = sis_priv->rx_ring[entry].cmdsts;
01646:
01647:     if (sis900_debug > 3)
01648:         printk(KERN_INFO "sis900_rx, cur_rx:%4.4d, dirty_rx:%4.4d "
01649:             "status:0x%8.8x\n",
01650:             sis_priv->cur_rx, sis_priv->dirty_rx, rx_status);
01651:
01652:     while (rx_status & OWN) {
01653:         unsigned int rx_size;
01654:
01655:         rx_size = (rx_status & DSIZE) - CRC_SIZE;
01656:

```

```

01657:     if (rx_status & (ABORT|OVERRUN|TOOLONG|RUNT|RXISERR|CRCERR|FAERR)) {
01658:         /* corrupted packet received */
01659:         if (sis900_debug > 3)
01660:             printk(KERN_INFO "%s: Corrupted packet "
01661:                 "received, buffer status = 0x%8.8x.\n",
01662:                 net_dev->name, rx_status);
01663:         sis_priv->stats.rx_errors++;
01664:         if (rx_status & OVERRUN)
01665:             sis_priv->stats.rx_over_errors++;
01666:         if (rx_status & (TOOLONG|RUNT))
01667:             sis_priv->stats.rx_length_errors++;
01668:         if (rx_status & (RXISERR | FAERR))
01669:             sis_priv->stats.rx_frame_errors++;
01670:         if (rx_status & CRCERR)
01671:             sis_priv->stats.rx_crc_errors++;
01672:         /* reset buffer descriptor state */
01673:         sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
01674:     } else {
01675:         struct sk_buff *skb;
01676:
01677:         /* This situation should never happen, but due to
01678:            some unknow bugs, it is possible that
01679:            we are working on NULL sk_buff :- ( */
01680:         if (sis_priv->rx_skbuff[entry] == NULL) {
01681:             printk(KERN_INFO "%s: NULL pointer "
01682:                 "encountered in Rx ring, skipping\n",
01683:                 net_dev->name);
01684:             break;
01685:         }
01686:
01687:         pci_unmap_single(sis_priv->pci_dev,
01688:             sis_priv->rx_ring[entry].bufptr, RX_BUF_SIZE,
01689:             PCI_DMA_FROMDEVICE);
01690:         /* give the socket buffer to upper layers */
01691:         skb = sis_priv->rx_skbuff[entry];
01692:         skb_put(skb, rx_size);
01693:         skb->protocol = eth_type_trans(skb, net_dev);
01694:         netif_rx(skb);
01695:
01696:         /* some network statistics */
01697:         if ((rx_status & BCAST) == MCAST)
01698:             sis_priv->stats.multicast++;
01699:         net_dev->last_rx = jiffies;
01700:         sis_priv->stats.rx_bytes += rx_size;
01701:         sis_priv->stats.rx_packets++;
01702:
01703:         /* refill the Rx buffer, what if there is not enough
01704:            * memory for new socket buffer ?? */
01705:         if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
01706:             /* not enough memory for skbuff, this makes a
01707:                * "hole" on the buffer ring, it is not clear
01708:                * how the hardware will react to this kind
01709:                * of degenerated buffer */
01710:             printk(KERN_INFO "%s: Memory squeeze, "
01711:                 "deferring packet.\n",
01712:                 net_dev->name);
01713:             sis_priv->rx_skbuff[entry] = NULL;
01714:             /* reset buffer descriptor state */
01715:             sis_priv->rx_ring[entry].cmdsts = 0;
01716:             sis_priv->rx_ring[entry].bufptr = 0;
01717:             sis_priv->stats.rx_dropped++;
01718:             break;
01719:         }
01720:         skb->dev = net_dev;
01721:         sis_priv->rx_skbuff[entry] = skb;
01722:         sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
01723:         sis_priv->rx_ring[entry].bufptr =
01724:             pci_map_single(sis_priv->pci_dev, skb->tail,
01725:                 RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
01726:         sis_priv->dirty_rx++;
01727:     } ? end else ?
01728:     sis_priv->cur_rx++;
01729:     entry = sis_priv->cur_rx % NUM_RX_DESC;
01730:     rx_status = sis_priv->rx_ring[entry].cmdsts;
01731: } ? end while rx_status&OWN ? // while
01732:
01733: /* refill the Rx buffer, what if the rate of refilling is slower
01734:  * than consuming ?? */
01735: for (; sis_priv->cur_rx - sis_priv->dirty_rx > 0; sis_priv->dirty_rx++) {
01736:     struct sk_buff *skb;
01737:

```

```

01738:     entry = sis_priv->dirty_rx % NUM_RX_DESC;
01739:
01740:     if (sis_priv->rx_skbuff[entry] == NULL) {
01741:         if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
01742:             /* not enough memory for skbuff, this makes a
01743:              * "hole" on the buffer ring, it is not clear
01744:              * how the hardware will react to this kind
01745:              * of degenerated buffer */
01746:             printk(KERN_INFO "%s: Memory squeeze,
01747:              \"deferring packet.\n\",
01748:              net_dev->name);
01749:             sis_priv->stats.rx_dropped++;
01750:             break;
01751:         }
01752:         skb->dev = net_dev;
01753:         sis_priv->rx_skbuff[entry] = skb;
01754:         sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
01755:         sis_priv->rx_ring[entry].bufptr =
01756:             pci_map_single(sis_priv->pci_dev, skb->tail,
01757:                 RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
01758:     }
01759: } ? end for ; sis_priv->cur_rx->sis... ?
01760: /* re-enable the potentially idle receive state machine */
01761: outl(RxENA | inl(loaddr + cr), loaddr + cr);
01762:
01763: return 0;
01764: } ? end sis900_rx ?
01765:
01766: /**
01767:  * sis900_finish_xmit - finish up transmission of packets
01768:  * @net_dev: the net device to be transmitted on
01769:  *
01770:  * Check for error condition and free socket buffer etc
01771:  * schedule for more transmission as needed
01772:  * Note: This function is called by interrupt handler,
01773:  * don't do "too much" work here
01774:  */
01775:
01776: static void sis900_finish_xmit (struct net_device *net_dev)
01777: {
01778:     struct sis900_private *sis_priv = net_dev->priv;
01779:
01780:     for (; sis_priv->dirty_tx != sis_priv->cur_tx; sis_priv->dirty_tx++) {
01781:         struct sk_buff *skb;
01782:         unsigned int entry;
01783:         u32 tx_status;
01784:
01785:         entry = sis_priv->dirty_tx % NUM_TX_DESC;
01786:         tx_status = sis_priv->tx_ring[entry].cmdsts;
01787:
01788:         if (tx_status & OWN) {
01789:             /* The packet is not transmitted yet (owned by hardware) !
01790:              * Note: the interrupt is generated only when Tx Machine
01791:              * is idle, so this is an almost impossible case */
01792:             break;
01793:         }
01794:
01795:         if (tx_status & (ABORT | UNDERRUN | OWCOLL)) {
01796:             /* packet unsuccessfully transmitted */
01797:             if (sis900_debug > 3)
01798:                 printk(KERN_INFO "%s: Transmit \"
01799:                 \"error, Tx status %8.8x.\n\",
01800:                 net_dev->name, tx_status);
01801:             sis_priv->stats.tx_errors++;
01802:             if (tx_status & UNDERRUN)
01803:                 sis_priv->stats.tx_fifo_errors++;
01804:             if (tx_status & ABORT)
01805:                 sis_priv->stats.tx_aborted_errors++;
01806:             if (tx_status & NOCARRIER)
01807:                 sis_priv->stats.tx_carrier_errors++;
01808:             if (tx_status & OWCOLL)
01809:                 sis_priv->stats.tx_window_errors++;
01810:         } else {
01811:             /* packet successfully transmitted */
01812:             sis_priv->stats.collisions += (tx_status & COLCNT) >> 16;
01813:             sis_priv->stats.tx_bytes += tx_status & DSIZE;
01814:             sis_priv->stats.tx_packets++;
01815:         }
01816:         /* Free the original skb. */

```

```

01817:         skb = sis_priv->tx_skbuff[entry];
01818:         pci_unmap_single(sis_priv->pci_dev,
01819:             sis_priv->tx_ring[entry].bufptr, skb->len,
01820:             PCI_DMA_TODEVICE);
01821:         dev_kfree_skb_irq(skb);
01822:         sis_priv->tx_skbuff[entry] = NULL;
01823:         sis_priv->tx_ring[entry].bufptr = 0;
01824:         sis_priv->tx_ring[entry].cmdsts = 0;
01825:     } ? end for ; sis_priv->dirty_tx != ... ?
01826:
01827:     if (sis_priv->tx_full && netif_queue_stopped(net_dev) &&
01828:         sis_priv->cur_tx - sis_priv->dirty_tx < NUM_TX_DESC - 4) {
01829:         /* The ring is no longer full, clear tx_full and schedule
01830:          * more transmission by netif_wake_queue(net_dev) */
01831:         sis_priv->tx_full = 0;
01832:         netif_wake_queue(net_dev);
01833:     }
01834: } ? end sis900_finish_xmit ?
01835:
01836: /**
01837:  * sis900_close - close sis900 device
01838:  * @net_dev: the net device to be closed
01839:  *
01840:  * Disable interrupts, stop the Tx and Rx Status Machine
01841:  * free Tx and RX socket buffer
01842:  */
01843:
01844: static int sis900_close(struct net_device *net_dev)
01845: {
01846:     long loaddr = net_dev->base_addr;
01847:     struct sis900_private *sis_priv = net_dev->priv;
01848:     struct sk_buff *skb;
01849:     int i;
01850:
01851:     netif_stop_queue(net_dev);
01852:
01853:     /* Disable interrupts by clearing the interrupt mask. */
01854:     outl(0x0000, loaddr + lmr);
01855:     outl(0x0000, loaddr + lerr);
01856:
01857:     /* Stop the chip's Tx and Rx Status Machine */
01858:     outl(RxDIS | TxDIS | inl(loaddr + cr), loaddr + cr);
01859:
01860:     del_timer(&sis_priv->timer);
01861:
01862:     free_irq(net_dev->irq, net_dev);
01863:
01864:     /* Free Tx and RX skbuff */
01865:     for (i = 0; i < NUM_RX_DESC; i++) {
01866:         skb = sis_priv->rx_skbuff[i];
01867:         if (skb) {
01868:             pci_unmap_single(sis_priv->pci_dev,
01869:                 sis_priv->rx_ring[i].bufptr,
01870:                 RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
01871:             dev_kfree_skb(skb);
01872:             sis_priv->rx_skbuff[i] = NULL;
01873:         }
01874:     }
01875:     for (i = 0; i < NUM_TX_DESC; i++) {
01876:         skb = sis_priv->tx_skbuff[i];
01877:         if (skb) {
01878:             pci_unmap_single(sis_priv->pci_dev,
01879:                 sis_priv->tx_ring[i].bufptr, skb->len,
01880:                 PCI_DMA_TODEVICE);
01881:             dev_kfree_skb(skb);
01882:             sis_priv->tx_skbuff[i] = NULL;
01883:         }
01884:     }
01885:
01886:     /* Green! Put the chip in low-power mode. */
01887:
01888:     return 0;
01889: } ? end sis900_close ?
01890:
01891: /**
01892:  * sis900_get_drvinfo - Return information about driver
01893:  * @net_dev: the net device to probe
01894:  * @info: container for info returned
01895:  *
01896:  * Process ethtool command such as "ethtool -i" to show information
01897:  */
01898:

```



```

01899: static void sis900_get_drvinfo(struct net_device *net_dev,
01900:                                struct ethtool_drvinfo *info)
01901: {
01902:     struct sis900_private *sis_priv = net_dev->priv;
01903:
01904:     strcopy (info->driver, SIS900_MODULE_NAME);
01905:     strcopy (info->version, SIS900_DRV_VERSION);
01906:     strcopy (info->bus_info, pci_name(sis_priv->pci_dev));
01907: }
01908:
01909: static struct ethtool_ops sis900_ethtool_ops = {
01910:     .get_drvinfo = sis900_get_drvinfo,
01911: };
01912:
01913: /**
01914:  * mii_ioctl - process MII i/o control command
01915:  * @net_dev: the net device to command for
01916:  * @rq: parameter for command
01917:  * @cmd: the i/o command
01918:  *
01919:  * Process MII command like read/write MII register
01920:  */
01921:
01922: static int mii_ioctl(struct net_device *net_dev, struct ifreq *rq, int cmd)
01923: {
01924:     struct sis900_private *sis_priv = net_dev->priv;
01925:     struct mii_ioctl_data *data = if_mii(rq);
01926:
01927:     switch(cmd) {
01928:     case SIOCGMIIPHY: /* Get address of MII PHY in use. */
01929:         data->phy_id = sis_priv->mii->phy_addr;
01930:         /* Fall Through */
01931:
01932:     case SIOCGMIIREG: /* Read MII PHY register. */
01933:         data->val_out = mdio_read(net_dev, data->phy_id & 0x1f, data->reg_num & 0x1f);
01934:         return 0;
01935:
01936:     case SIOCSMIIREG: /* Write MII PHY register. */
01937:         if (!capable(CAP_NET_ADMIN))
01938:             return -EPERM;
01939:         mdio_write(net_dev, data->phy_id & 0x1f, data->reg_num & 0x1f, data->val_in);
01940:         return 0;
01941:     default:
01942:         return -EOPNOTSUPP;
01943:     }
01944: } ? end mii_ioctl ?
01945:
01946: /**
01947:  * sis900_get_stats - Get sis900 read/write statistics
01948:  * @net_dev: the net device to get statistics for
01949:  *
01950:  * get tx/rx statistics for sis900
01951:  */
01952:
01953: static struct net_device_stats *
01954: sis900_get_stats(struct net_device *net_dev)
01955: {
01956:     struct sis900_private *sis_priv = net_dev->priv;
01957:
01958:     return &sis_priv->stats;
01959: }
01960:
01961: /**
01962:  * sis900_set_config - Set media type by net_device.set_config
01963:  * @dev: the net device for media type change
01964:  * @map: ifmap passed by ifconfig
01965:  *
01966:  * Set media type to 10baseT, 100baseT or 0(for auto) by ifconfig
01967:  * we support only port changes. All other runtime configuration
01968:  * changes will be ignored
01969:  */
01970:
01971: static int sis900_set_config(struct net_device *dev, struct ifmap *map)
01972: {
01973:     struct sis900_private *sis_priv = dev->priv;
01974:     struct mii_phy *mii_phy = sis_priv->mii;
01975:
01976:     u16 status;
01977:

```

```

01978: if ((map->port != (u_char)(-1)) && (map->port != dev->if_port)) {
01979:     /* we switch on the ifmap->port field. I couldn't find anything
01980:      * like a definition or standard for the values of that field.
01981:      * I think the meaning of those values is device specific. But
01982:      * since I would like to change the media type via the ifconfig
01983:      * command I use the definition from linux/netdevice.h
01984:      * (which seems to be different from the ifport(pcmcia) definition) */
01985:     switch(map->port){
01986:     case IF_PORT_UNKNOWN: /* use auto here */
01987:         dev->if_port = map->port;
01988:         /* we are going to change the media type, so the Link
01989:          * will be temporary down and we need to reflect that
01990:          * here. When the Link comes up again, it will be
01991:          * sensed by the sis_timer procedure, which also does
01992:          * all the rest for us */
01993:         netif_carrier_off(dev);
01994:
01995:         /* read current state */
01996:         status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);
01997:
01998:         /* enable auto negotiation and reset the negotiation
01999:          * (I don't really know what the auto negotiation
02000:          * reset really means, but it sounds for me right to
02001:          * do one here) */
02002:         mdio_write(dev, mii_phy->phy_addr,
02003:                    MII_CONTROL, status | MII_CNTL_AUTO | MII_CNTL_RST_AUTO);
02004:
02005:         break;
02006:
02007:     case IF_PORT_10BASET: /* 10BaseT */
02008:         dev->if_port = map->port;
02009:
02010:         /* we are going to change the media type, so the Link
02011:          * will be temporary down and we need to reflect that
02012:          * here. When the Link comes up again, it will be
02013:          * sensed by the sis_timer procedure, which also does
02014:          * all the rest for us */
02015:         netif_carrier_off(dev);
02016:
02017:         /* set Speed to 10Mbps */
02018:         /* read current state */
02019:         status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);
02020:
02021:         /* disable auto negotiation and force 10Mbit mode*/
02022:         mdio_write(dev, mii_phy->phy_addr,
02023:                    MII_CONTROL, status & ~(MII_CNTL_SPEED |
02024:                    MII_CNTL_AUTO));
02025:         break;
02026:
02027:     case IF_PORT_100BASET: /* 100BaseT */
02028:     case IF_PORT_100BASETX: /* 100BaseTx */
02029:         dev->if_port = map->port;
02030:
02031:         /* we are going to change the media type, so the Link
02032:          * will be temporary down and we need to reflect that
02033:          * here. When the Link comes up again, it will be
02034:          * sensed by the sis_timer procedure, which also does
02035:          * all the rest for us */
02036:         netif_carrier_off(dev);
02037:
02038:         /* set Speed to 100Mbps */
02039:         /* disable auto negotiation and enable 100Mbit Mode */
02040:         status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);
02041:         mdio_write(dev, mii_phy->phy_addr,
02042:                    MII_CONTROL, (status & ~MII_CNTL_SPEED) |
02043:                    MII_CNTL_SPEED);
02044:
02045:         break;
02046:
02047:     case IF_PORT_10BASE2: /* 10Base2 */
02048:     case IF_PORT_AUI: /* AUI */
02049:     case IF_PORT_100BASEFX: /* 100BaseFx */
02050:         /* These Modes are not supported (are they?) */
02051:         printk(KERN_INFO "Not supported");
02052:         return -EOPNOTSUPP;
02053:         break;
02054:
02055:     default:

```



```

02056:         printk(KERN_INFO "Invalid");
02057:         return -EINVAL;
02058:     } ? end switch map->port ?
02059: } ? end if (map->port! =(u_char){... ?
02060: return 0;
02061: } ? end sis900_set_config ?
02062:
02063: /**
02064:  * sis900_mcast_bitnr - compute hashtable index
02065:  * @addr: multicast address
02066:  * @revision: revision id of chip
02067:  *
02068:  * SIS 900 uses the most significant 7 bits to index a 128 bits multicast
02069:  * hash table, which makes this function a little bit different from other drivers
02070:  * SIS 900 B0 & 635 M/B uses the most significant 8 bits to index 256 bits
02071:  * multicast hash table.
02072:  */
02073:
02074: static inline u16 sis900_mcast_bitnr(u8 *addr, u8 revision)
02075: {
02076:
02077:     u32 crc = ether_crc(6, addr);
02078:
02079:     /* leave 8 or 7 most significant bits */
02080:     if ((revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV))
02081:         return ((int)(crc >> 24));
02082:     else
02083:         return ((int)(crc >> 25));
02084: }
02085:
02086: /**
02087:  * set_rx_mode - Set SIS900 receive mode
02088:  * @net_dev: the net device to be set
02089:  *
02090:  * Set SIS900 receive mode for promiscuous, multicast, or broadcast mode.
02091:  * And set the appropriate multicast filter.
02092:  * Multicast hash table changes from 128 to 256 bits for 635M/B & 900B0.
02093:  */
02094:
02095: static void set_rx_mode(struct net_device *net_dev)
02096: {
02097:     long loaddr = net_dev->base_addr;
02098:     struct sis900_private *sis_priv = net_dev->priv;
02099:     u16 mc_filter[16] = {0}; /* 256/128 bits multicast hash table */
02100:     int i, table_entries;
02101:     u32 rx_mode;
02102:     u8 revision;
02103:
02104:     /* 635 Hash Table entries = 256(2^16) */
02105:     pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
02106:     if((revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV))
02107:         table_entries = 16;
02108:     else
02109:         table_entries = 8;
02110:
02111:     if (net_dev->flags & IFF_PROMISC) {
02112:         /* Accept any kinds of packets */
02113:         rx_mode = RFPromiscuous;
02114:         for (i = 0; i < table_entries; i++)
02115:             mc_filter[i] = 0xffff;
02116:     } else if ((net_dev->mc_count > multicast_filter_limit) ||
02117:                (net_dev->flags & IFF_ALLMULTI)) {
02118:         /* too many multicast addresses or accept all multicast packet */
02119:         rx_mode = RFAAB | RFAAM;
02120:         for (i = 0; i < table_entries; i++)
02121:             mc_filter[i] = 0xffff;
02122:     } else {
02123:         /* Accept Broadcast packet, destination address matchs our
02124:          * MAC address, use Receive Filter to reject unwanted MCAST
02125:          * packets */
02126:         struct dev_mc_list *mclist;
02127:         rx_mode = RFAAB;
02128:         for (i = 0, mclist = net_dev->mc_list;
02129:              mclist && i < net_dev->mc_count;
02130:              i++, mclist = mclist->next) {
02131:             unsigned int bit_nr =
02132:                 sis900_mcast_bitnr(mclist->dmi_addr, revision);
02133:             mc_filter[bit_nr >> 4] |= (1 << (bit_nr & 0xf));
02134:         }
02135:     }
02136: }

```

```

02137: /* update Multicast Hash Table in Receive Filter */
02138: for (i = 0; i < table_entries; i++) {
02139:     /* why plus 0x04 ??, That makes the correct value for hash table. */
02140:     outl((u32)(0x00000004+i) << RFADDR_shift, loaddr + rfcr);
02141:     outl(mc_filter[i], loaddr + rldr);
02142: }
02143:
02144: outl(RFEN | rx_mode, loaddr + rfcr);
02145:
02146: /* sis900 is capable of looping back packets at MAC level for
02147:  * debugging purpose */
02148: if (net_dev->flags & IFF_LOOPBACK) {
02149:     u32 cr_saved;
02150:     /* We must disable Tx/Rx before setting loopback mode */
02151:     cr_saved = inl(loaddr + cr);
02152:     outl(cr_saved | TxDIS | RxDIS, loaddr + cr);
02153:     /* enable loopback */
02154:     outl(inl(loaddr + txcfg) | TxMLB, loaddr + txcfg);
02155:     outl(inl(loaddr + rxcfg) | RxATX, loaddr + rxcfg);
02156:     /* restore cr */
02157:     outl(cr_saved, loaddr + cr);
02158: }
02159:
02160: return;
02161: } ? end set_rx_mode ?
02162:
02163: /**
02164:  * sis900_reset - Reset sis900 MAC
02165:  * @net_dev: the net device to reset
02166:  *
02167:  * reset sis900 MAC and wait until finished
02168:  * reset through command register
02169:  * change backoff algorithm for 900B0 & 635 M/B
02170:  */
02171:
02172: static void sis900_reset(struct net_device *net_dev)
02173: {
02174:     struct sis900_private *sis_priv = net_dev->priv;
02175:     long loaddr = net_dev->base_addr;
02176:     int i = 0;
02177:     u32 status = TxRCMP | RxRCMP;
02178:     u8 revision;
02179:
02180:     outl(0, loaddr + ler);
02181:     outl(0, loaddr + lmr);
02182:     outl(0, loaddr + rfcr);
02183:
02184:     outl(RxRESET | TxRESET | RESET | inl(loaddr + cr), loaddr + cr);
02185:
02186:     /* Check that the chip has finished the reset. */
02187:     while (status && (i++ < 1000)) {
02188:         status ^= (inl(isr + loaddr) & status);
02189:     }
02190:
02191:     pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
02192:     if((revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV))
02193:         outl(PESEL | RND_CNT, loaddr + cfg);
02194:     else
02195:         outl(PESEL, loaddr + cfg);
02196: } ? end sis900_reset ?
02197:
02198: /**
02199:  * sis900_remove - Remove sis900 device
02200:  * @pci_dev: the pci device to be removed
02201:  *
02202:  * remove and release SIS900 net device
02203:  */
02204:
02205: static void __devexit sis900_remove(struct pci_dev *pci_dev)
02206: {
02207:     struct net_device *net_dev = pci_get_drvdata(pci_dev);
02208:     struct sis900_private *sis_priv = net_dev->priv;
02209:     struct mii_phy *phy = NULL;
02210:
02211:     while (sis_priv->first_mii) {
02212:         phy = sis_priv->first_mii;
02213:         sis_priv->first_mii = phy->next;
02214:         kfree(phy);
02215:     }
02216: }

```

```

02217:     pci_free_consistent(pci_dev, RX_TOTAL_SIZE, sis_priv->rx_ring,
02218:         sis_priv->rx_ring_dma);
02219:     pci_free_consistent(pci_dev, TX_TOTAL_SIZE, sis_priv->tx_ring,
02220:         sis_priv->tx_ring_dma);
02221:     unregister_netdev(net_dev);
02222:     free_netdev(net_dev);
02223:     pci_release_regions(pci_dev);
02224:     pci_set_drvdata(pci_dev, NULL);
02225: } ? end sis900_remove ?
02226:
02227: #ifdef CONFIG_PM
02228:
02229: static int sis900_suspend(struct pci_dev *pci_dev, u32 state)
02230: {
02231:     struct net_device *net_dev = pci_get_drvdata(pci_dev);
02232:     long ioaddr = net_dev->base_addr;
02233:
02234:     if(! netif_running(net_dev))
02235:         return 0;
02236:
02237:     netif_stop_queue(net_dev);
02238:     netif_device_detach(net_dev);
02239:
02240:     /* Stop the chip's Tx and Rx Status Machine */
02241:     outl(RxDIS | TxDIS | inl(ioaddr + cr), ioaddr + cr);
02242:
02243:     pci_set_power_state(pci_dev, PCI_D3hot);
02244:     pci_save_state(pci_dev);
02245:
02246:     return 0;
02247: }
02248:
02249: static int sis900_resume(struct pci_dev *pci_dev)
02250: {
02251:     struct net_device *net_dev = pci_get_drvdata(pci_dev);
02252:     struct sis900_private *sis_priv = net_dev->priv;
02253:     long ioaddr = net_dev->base_addr;
02254:
02255:     if(! netif_running(net_dev))
02256:         return 0;
02257:     pci_restore_state(pci_dev);
02258:     pci_set_power_state(pci_dev, PCI_D0);
02259:
02260:     sis900_init_rxfilter(net_dev);
02261:
02262:     sis900_init_tx_ring(net_dev);
02263:     sis900_init_rx_ring(net_dev);
02264:
02265:     set_rx_mode(net_dev);
02266:
02267:     netif_device_attach(net_dev);
02268:     netif_start_queue(net_dev);
02269:
02270:     /* Workaround for EDB */
02271:     sis900_set_mode(ioaddr, HW_SPEED_10_MBPS, FDX_CAPABLE_HALF_SELECTED);
02272:
02273:     /* Enable all known interrupts by setting the interrupt mask. */
02274:     outl((RxSOVR| RxORN| RxERR| RxOK| TxURN| TxERR| TxIDLE), ioaddr + imr);
02275:     outl(RxENA | inl(ioaddr + cr), ioaddr + cr);
02276:     outl(IE, ioaddr + ier);
02277:
02278:     sis900_check_mode(net_dev, sis_priv->mii);
02279:
02280:     return 0;
02281: } ? end sis900_resume ?
02282: #endif /* CONFIG_PM */
02283:
02284: static struct pci_driver sis900_pci_driver = {
02285:     .name = SIS900_MODULE_NAME,
02286:     .id_table = sis900_pci_tbl,
02287:     .probe = sis900_probe,
02288:     .remove = __devexit_p(sis900_remove),
02289: #ifdef CONFIG_PM
02290:     .suspend = sis900_suspend,
02291:     .resume = sis900_resume,
02292: #endif /* CONFIG_PM */
02293: };
02294:
02295: static int __init sis900_init_module(void)
02296: {
02297:     /* when a module, this is printed whether or not devices are found in probe */

```

```

02298: #ifdef MODULE
02299:     printk(version);
02300: #endif
02301:
02302:     return pci_module_init(&sis900_pci_driver);
02303: }
02304:
02305: static void __exit sis900_cleanup_module(void)
02306: {
02307:     pci_unregister_driver(&sis900_pci_driver);
02308: }
02309:
02310: module_init(sis900_init_module);
02311: module_exit(sis900_cleanup_module);
02312:

```