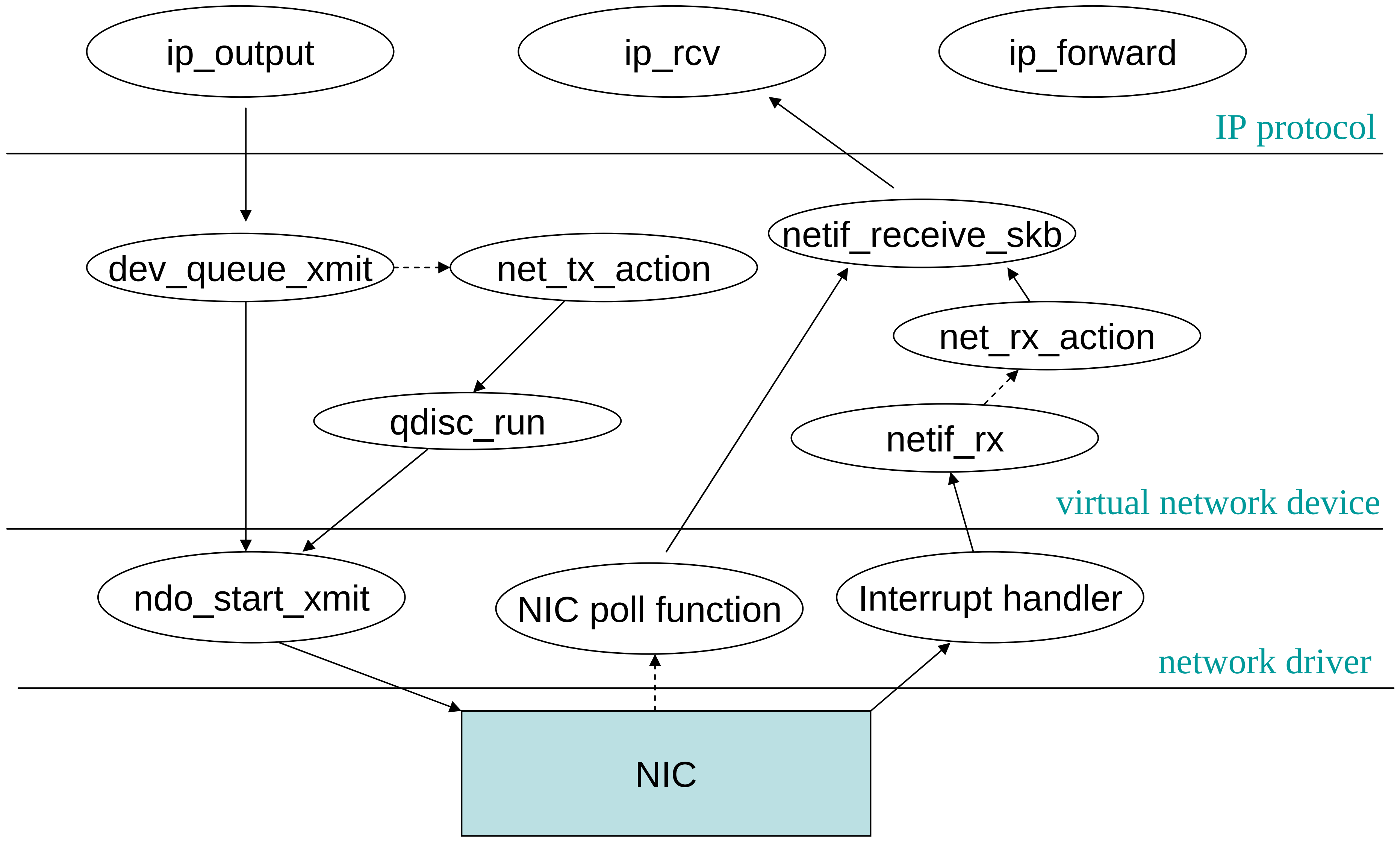
# Architecture Overview



```c
/* sis900.c: A SiS 900/7016 PCI Fast Ethernet driver for Linux.
   Copyright 1999 Silicon Integrated System Corporation
   Revision:	1.08.06 Sep. 24 2002

   Modified from the driver which is originally written by Donald Becker.

   This software may be used and distributed according to the terms
   of the GNU General Public License (GPL), incorporated herein by reference.
   Drivers based on this skeleton fall under the GPL and must retain
   the authorship (implicit copyright) notice.

   References:
   SiS 7016 Fast Ethernet PCI Bus 10/100 Mbps LAN Controller with OnNow Support,
   preliminary Rev. 1.0 Jan. 14, 1998
   SiS 900 Fast Ethernet PCI Bus 10/100 Mbps LAN Single Chip with OnNow Support,
   preliminary Rev. 1.0 Nov. 10, 1998
   SiS 7014 Single Chip 100BASE-TX/10BASE-T Physical Layer Solution,
   preliminary Rev. 1.0 Jan. 18, 1998
   http://www.sis.com.tw/support/databook.htm

   Rev 1.08.07 Nov.  2 2003 Daniele Venzano <webvenza@libero.it> add suspend/resume support
   Rev 1.08.06 Sep. 24 2002 Mufasa Yang bug fix for Tx timeout & add SiS963 support
   Rev 1.08.05 Jun.  6 2002 Mufasa Yang bug fix for read_eeprom & Tx descriptor over-boundary
   Rev 1.08.04 Apr. 25 2002 Mufasa Yang <mufasa@sis.com.tw> added SiS962 support
   Rev 1.08.03 Feb.  1 2002 Matt Domsch <Matt_Domsch@dell.com> update to use library crc32 function
   Rev 1.08.02 Nov. 30 2001 Hui-Fen Hsu workaround for EDB & bug fix for dhcp problem
   Rev 1.08.01 Aug. 25 2001 Hui-Fen Hsu update for 630ET & workaround for ICS1893 PHY
   Rev 1.08.00 Jun. 11 2001 Hui-Fen Hsu workaround for RTL8201 PHY and some bug fix
   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
   Rev 1.07.10 Mar.  1 2001 Hui-Fen Hsu <hfhsu@sis.com.tw> some bug fix & 635M/B support
   Rev 1.07.09 Feb.  9 2001 Dave Jones <davej@suse.de> PCI enable cleanup
   Rev 1.07.08 Jan.  8 2001 Lei-Chun Chang added RTL8201 PHY support
   Rev 1.07.07 Nov. 29 2000 Lei-Chun Chang added kernel-doc extractable documentation and 630 workaround fix
   Rev 1.07.06 Nov.  7 2000 Jeff Garzik <jgarzik@pobox.com> some bug fix and cleaning
   Rev 1.07.05 Nov.  6 2000 metapirat<metapirat@gmx.de> contribute media type select by ifconfig
   Rev 1.07.04 Sep.  6 2000 Lei-Chun Chang added ICS1893 PHY support
   Rev 1.07.03 Aug. 24 2000 Lei-Chun Chang (lcchang@sis.com.tw) modified 630E eqaulizer workaround rule
   Rev 1.07.01 Aug. 08 2000 Ollie Lho minor update for SiS 630E and SiS 630E A1
   Rev 1.07    Mar. 07 2000 Ollie Lho bug fix in Rx buffer ring
   Rev 1.06.04 Feb. 11 2000 Jeff Garzik <jgarzik@pobox.com> softnet and init for kernel 2.4
   Rev 1.06.03 Dec. 23 1999 Ollie Lho Third release
   Rev 1.06.02 Nov. 23 1999 Ollie Lho bug in mac probing fixed
   Rev 1.06.01 Nov. 16 1999 Ollie Lho CRC calculation provide by Joseph Zbiciak (im14u2c@primenet.com)
   Rev 1.06 Nov. 4 1999 Ollie Lho (ollie@sis.com.tw) Second release
   Rev 1.05.05 Oct. 29 1999 Ollie Lho (ollie@sis.com.tw) Single buffer Tx/Rx
   Chin-Shan Li (lcs@sis.com.tw) Added AMD Am79c901 HomePNA PHY support
   Rev 1.05 Aug. 7 1999 Jim Huang (cmhuang@sis.com.tw) Initial release
*/

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/timer.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#include <linux/slab.h>
#include <linux/interrupt.h>
#include <linux/pci.h>
#include <linux/netdevice.h>
#include <linux/init.h>
#include <linux/mii.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/delay.h>
#include <linux/ethtool.h>
#include <linux/crc32.h>
#include <linux/bitops.h>

#include <asm/processor.h>      /* Processor type for cache alignment. */
#include <asm/io.h>
#include <asm/uaccess.h>    /* User space memory access functions */

#include "sis900.h"

#define SIS900_MODULE_NAME "sis900"
#define SIS900_DRV_VERSION "v1.08.07 11/02/2003"

static char version[] __devinitdata =
KERN_INFO "sis900.c: " SIS900_DRV_VERSION "\n";

```

```c
static int max_interrupt_work = 40;
static int multicast_filter_limit = 128;

#define sis900_debug debug
static int sis900_debug;

/* Time in jiffies before concluding the transmitter is hung. */
#define TX_TIMEOUT  (4*HZ)
/* SiS 900 is capable of 32 bits BM DMA */
#define SIS900_DMA_MASK 0xffffffff

enum {
	SIS_900 = 0,
	SIS_7016
};
static char * card_names[] = {
	"SiS 900 PCI Fast Ethernet",
	"SiS 7016 PCI Fast Ethernet"
};
static struct pci_device_id sis900_pci_tbl [] = {
	{PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_900,
	 PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_900},
	{PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_7016,
	 PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_7016},
	{0,}
};
MODULE_DEVICE_TABLE (pci, sis900_pci_tbl);

static void sis900_read_mode(struct net_device *net_dev, int *speed, int *duplex);

static struct mii_chip_info {
	const char * name;
	u16 phy_id0;
	u16 phy_id1;
	u8  phy_types;
#define	HOME 	0x0001
#define LAN	0x0002
#define MIX	0x0003
#define UNKNOWN	0x0
} mii_chip_table[] = {
	{ "SiS 900 Internal MII PHY", 		0x001d, 0x8000, LAN },
	{ "SiS 7014 Physical Layer Solution", 	0x0016, 0xf830, LAN },
	{ "Altimata AC101LF PHY", 		0x0022, 0x5520, LAN },
	{ "AMD 79C901 10BASE-T PHY",  		0x0000, 0x6B70, LAN },
	{ "AMD 79C901 HomePNA PHY",		0x0000, 0x6B90, HOME},
	{ "ICS LAN PHY",			0x0015, 0xF440, LAN },
	{ "NS 83851 PHY",			0x2000, 0x5C20, MIX },
	{ "NS 83847 PHY",                       0x2000, 0x5C30, MIX },
	{ "Realtek RTL8201 PHY",		0x0000, 0x8200, LAN },
	{ "VIA 6103 PHY",			0x0101, 0x8f20, LAN },
	{NULL,},
};

struct mii_phy {
	struct mii_phy * next;
	int phy_addr;
	u16 phy_id0;
	u16 phy_id1;
	u16 status;
	u8  phy_types;
};

typedef struct _BufferDesc {
	u32 link;
	u32 cmdsts;
	u32 bufptr;
} BufferDesc;

struct sis900_private {
	struct net_device_stats stats;
	struct pci_dev * pci_dev;

	spinlock_t lock;

	struct mii_phy * mii;
	struct mii_phy * first_mii; /* record the first mii structure */
	unsigned int cur_phy;

	struct timer_list timer; /* Link status detection timer. */
	u8 autong_complete; /* 1: auto-negotiate complete  */

```

```
        unsigned int cur_rx, dirty_rx; / * producer/ comsumer pointers for Tx/ Rx ring */
        unsigned int cur_tx, dirty_tx;

        / * The saved address of a sent/ receive- in- place packet buffer */
        struct sk_buff *tx_skbuff[NUM_TX_DESC];
        struct sk_buff *rx_skbuff[NUM_RX_DESC];
        BufferDesc *tx_ring;
        BufferDesc *rx_ring;

        dma_addr_t tx_ring_dma;
        dma_addr_t rx_ring_dma;

        unsigned int tx_full; / * The Tx queue is full. */
        u8 host_bridge_rev;
} ? end sis900_private ? ;

MODULE_AUTHOR("Jim Huang <cmhuang@sis.com.tw>, Ollie Lho <ollie@sis.com.tw>");
MODULE_DESCRIPTION("SiS 900 PCI Fast Ethernet driver");
MODULE_LICENSE("GPL");

module_param(multicast_filter_limit, int, 0444);
module_param(max_interrupt_work, int, 0444);
module_param(debug, int, 0444);
MODULE_PARM_DESC(multicast_filter_limit,
"SiS 900/ 7016 maximum number of filtered multicast addresses");
MODULE_PARM_DESC(max_interrupt_work, "SiS 900/ 7016 maximum events handled per interrupt");
MODULE_PARM_DESC(debug, "SiS 900/ 7016 debug level (2- 4)");

static int sis900_open(struct net_device *net_dev);
static int sis900_mii_probe (struct net_device * net_dev);
static void sis900_init_rxfilter (struct net_device * net_dev);
static u16 read_eeprom(long ioaddr, int location);
static u16 mdio_read(struct net_device *net_dev, int phy_id, int location);
static void mdio_write(struct net_device *net_dev, int phy_id, int location, int val);
static void sis900_timer(unsigned long data);
static void sis900_check_mode (struct net_device *net_dev, struct mii_phy *mii_phy);
static void sis900_tx_timeout(struct net_device *net_dev);
static void sis900_init_tx_ring(struct net_device *net_dev);
static void sis900_init_rx_ring(struct net_device *net_dev);
static int sis900_start_xmit(struct sk_buff *skb, struct net_device *net_dev);
static int sis900_rx(struct net_device *net_dev);
static void sis900_finish_xmit (struct net_device *net_dev);
static irqreturn_t sis900_interrupt(int irq, void *dev_instance, struct pt_regs *regs);
static int sis900_close(struct net_device *net_dev);
static int mii_ioctl(struct net_device *net_dev, struct ifreq *rq, int cmd);
static struct net_device_stats *sis900_get_stats(struct net_device *net_dev);
static u16 sis900_mcast_bitnr(u8 *addr, u8 revision);
static void set_rx_mode(struct net_device *net_dev);
static void sis900_reset(struct net_device *net_dev);
static void sis630_set_eq(struct net_device *net_dev, u8 revision);
static int sis900_set_config(struct net_device *dev, struct ifmap *map);
static u16 sis900_default_phy(struct net_device * net_dev);
static void sis900_set_capability( struct net_device *net_dev ,struct mii_phy *phy);
static u16 sis900_reset_phy(struct net_device *net_dev, int phy_addr);
static void sis900_auto_negotiate(struct net_device *net_dev, int phy_addr);
static void sis900_set_mode (long ioaddr, int speed, int duplex);
static struct ethtool_ops sis900_ethtool_ops;

/ **
 *      sis900_get_mac_addr - Get MAC address for stand alone SiS900 model
 *      @pci_dev: the sis900 pci device
 *      @net_dev: the net device to get address for
 *
 *      Older SiS900 and friends, use EEPROM to store MAC address.
 *      MAC address is read from read_eeprom() into @net_dev- >dev_addr.
 */

```

```
static int __devinit sis900_get_mac_addr(struct pci_dev * pci_dev, struct net_device *net_dev)
{
        long ioaddr = pci_resource_start(pci_dev, 0);
        u16 signature;
        int i;

        / * check to see if we have sane EEPROM */
        signature = (u16) read_eeprom(ioaddr, EEPROMSignature);
        if (signature == 0xffff || signature == 0x0000) {
                printk (KERN_INFO "%s: Error EERPOM read %x\n",
                        net_dev- >name, signature);
                return 0;
        }

        / * get MAC address from EEPROM */
        for (i = 0; i < 3; i++)
                ((u16 *)(net_dev- >dev_addr))[i] = read_eeprom(ioaddr, i+EEPROMMACAddr);

        return 1;
} ? end sis900_get_mac_addr ?

/ **
 *      sis630e_get_mac_addr - Get MAC address for SiS630E model
 *      @pci_dev: the sis900 pci device
 *      @net_dev: the net device to get address for
 *
 *      SiS630E model, use APC CMOS RAM to store MAC address.
 *      APC CMOS RAM is accessed through ISA bridge.
 *      MAC address is read into @net_dev- >dev_addr.
 */

static int __devinit sis630e_get_mac_addr(struct pci_dev * pci_dev,
                                        struct net_device *net_dev)
{
        struct pci_dev *isa_bridge = NULL;
        u8 reg;
        int i;

        isa_bridge = pci_get_device(PCI_VENDOR_ID_SI, 0x0008, isa_bridge);
        if (!isa_bridge)
                isa_bridge = pci_get_device(PCI_VENDOR_ID_SI, 0x0018, isa_bridge);
        if (!isa_bridge) {
                printk("%s: Can not find ISA bridge\n", net_dev- >name);
                return 0;
        }
        pci_read_config_byte(isa_bridge, 0x48, &reg);
        pci_write_config_byte(isa_bridge, 0x48, reg | 0x40);

        for (i = 0; i < 6; i++) {
                outb(0x09 + i, 0x70);
                ((u8 *)(net_dev- >dev_addr))[i] = inb(0x71);
        }
        pci_write_config_byte(isa_bridge, 0x48, reg & ~0x40);
        pci_dev_put(isa_bridge);

        return 1;
} ? end sis630e_get_mac_addr ?


/ **
 *      sis635_get_mac_addr - Get MAC address for SIS635 model
 *      @pci_dev: the sis900 pci device
 *      @net_dev: the net device to get address for
 *
 *      SiS635 model, set MAC Reload Bit to load Mac address from APC
 *      to rfdr. rfdr is accessed through rfcr. MAC address is read into
 *      @net_dev- >dev_addr.
 */

static int __devinit sis635_get_mac_addr(struct pci_dev * pci_dev,
                                        struct net_device *net_dev)
{
        long ioaddr = net_dev- >base_addr;
        u32 rfcrSave;
        u32 i;

        rfcrSave = inl(rfcr + ioaddr);

        outl(rfcrSave | RELOAD, ioaddr + cr);
        outl(0, ioaddr + cr);

```

```c
        /* disable packet filtering before setting filter */
        outl(rfcrSave & ~RFEN, rfcr + ioaddr);

        /* load MAC addr to filter data register */
        for (i = 0 ; i < 3 ; i++) {
                outl((i << RFADDR_shift), ioaddr + rfcr);
                *( ((u16 *)net_dev->dev_addr) + i) = inw(ioaddr + rfdr);
        }

        /* enable packet filtering */
        outl(rfcrSave | RFEN, rfcr + ioaddr);

        return 1;
} ? end sis635_get_mac_addr ?

/**
 *      sis96x_get_mac_addr - Get MAC address for SiS962 or SiS963 model
 *      @pci_dev: the sis900 pci device
 *      @net_dev: the net device to get address for
 *
 *      SiS962 or SiS963 model, use EEPROM to store MAC address. And EEPROM
 *      is shared by
 *      LAN and 1394. When access EEPROM, send EEREQ signal to hardware first
 *      and wait for EEGNT. If EEGNT is ON, EEPROM is permitted to be access
 *      by LAN, otherwise is not. After MAC address is read from EEPROM, send
 *      EEDONE signal to refuse EEPROM access by LAN.
 *      The EEPROM map of SiS962 or SiS963 is different to SiS900.
 *      The signature field in SiS962 or SiS963 spec is meaningless.
 *      MAC address is read into @net_dev->dev_addr.
 */

static int __devinit sis96x_get_mac_addr(struct pci_dev * pci_dev,
                                        struct net_device *net_dev)
{
        long ioaddr = net_dev->base_addr;
        long ee_addr = ioaddr + mear;
        u32 waittime = 0;
        int i;

        outl(EEREQ, ee_addr);
        while(waittime < 2000) {
                if(inl(ee_addr) & EEGNT) {

                        /* get MAC address from EEPROM */
                        for (i = 0; i < 3; i++)
                                ((u16 *)(net_dev->dev_addr))[i] = read_eeprom(ioaddr, i+EEPROMMACAddr);

                        outl(EEDONE, ee_addr);
                        return 1;
                } else {
                        udelay(1);
                        waittime ++;
                }
        }
        outl(EEDONE, ee_addr);
        return 0;
} ? end sis96x_get_mac_addr ?

/**
 *      sis900_probe - Probe for sis900 device
 *      @pci_dev: the sis900 pci device
 *      @pci_id: the pci device ID
 *
 *      Check and probe sis900 net device for @pci_dev.
 *      Get mac address according to the chip revision,
 *      and assign SiS900-specific entries in the device structure.
 *      ie: sis900_open(), sis900_start_xmit(), sis900_close(), etc.
 */

```

```c
static int __devinit sis900_probe(struct pci_dev *pci_dev,
                                const struct pci_device_id *pci_id)
{
        struct sis900_private *sis_priv;
        struct net_device *net_dev;
        struct pci_dev *dev;
        dma_addr_t ring_dma;
        void *ring_space;
        long ioaddr;
        int i, ret;
        u8 revision;
        char *card_name = card_names[pci_id->driver_data];

/* when built into the kernel, we only print version if device is found */
#ifndef MODULE
        static int printed_version;
        if (!printed_version++)
                printk(version);
#endif

        /* setup various bits in PCI command register */
        ret = pci_enable_device(pci_dev);
        if(ret) return ret;

        i = pci_set_dma_mask(pci_dev, SIS900_DMA_MASK);
        if(i){
                printk(KERN_ERR "sis900.c: architecture does not support"
                        "32bit PCI busmaster DMA\n");
                return i;
        }

        pci_set_master(pci_dev);

        net_dev = alloc_etherdev(sizeof(struct sis900_private));
        if (!net_dev)
                return -ENOMEM;
        SET_MODULE_OWNER(net_dev);
        SET_NETDEV_DEV(net_dev, &pci_dev->dev);

        /* We do a request_region() to register /proc/ioports info. */
        ioaddr = pci_resource_start(pci_dev, 0);
        ret = pci_request_regions(pci_dev, "sis900");
        if (ret)
                goto ↓err_out;

        sis_priv = net_dev->priv;
        net_dev->base_addr = ioaddr;
        net_dev->irq = pci_dev->irq;
        sis_priv->pci_dev = pci_dev;
        spin_lock_init(&sis_priv->lock);

        pci_set_drvdata(pci_dev, net_dev);

        ring_space = pci_alloc_consistent(pci_dev, TX_TOTAL_SIZE, &ring_dma);
        if (!ring_space) {
                ret = -ENOMEM;
                goto ↓err_out_cleardev;
        }
        sis_priv->tx_ring = (BufferDesc *)ring_space;
        sis_priv->tx_ring_dma = ring_dma;

        ring_space = pci_alloc_consistent(pci_dev, RX_TOTAL_SIZE, &ring_dma);
        if (!ring_space) {
                ret = -ENOMEM;
                goto ↓err_unmap_tx;
        }
        sis_priv->rx_ring = (BufferDesc *)ring_space;
        sis_priv->rx_ring_dma = ring_dma;

        /* The SiS900-specific entries in the device structure. */
        net_dev->open = &sis900_open;
        net_dev->hard_start_xmit = &sis900_start_xmit;
        net_dev->stop = &sis900_close;
        net_dev->get_stats = &sis900_get_stats;
        net_dev->set_config = &sis900_set_config;
        net_dev->set_multicast_list = &set_rx_mode;
        net_dev->do_ioctl = &mii_ioctl;
        net_dev->tx_timeout = sis900_tx_timeout;
        net_dev->watchdog_timeo = TX_TIMEOUT;
        net_dev->ethtool_ops = &sis900_ethtool_ops;

```

```c
	ret = register_netdev(net_dev);
	if (ret)
		goto ↓err_unmap_rx;

	/* Get Mac address according to the chip revision */
	pci_read_config_byte(pci_dev, PCI_CLASS_REVISION, &revision);
	ret = 0;

	if (revision == SIS630E_900_REV)
		ret = sis630e_get_mac_addr(pci_dev, net_dev);
	else if ((revision > 0x81) && (revision <= 0x90) )
		ret = sis635_get_mac_addr(pci_dev, net_dev);
	else if (revision == SIS96x_900_REV)
		ret = sis96x_get_mac_addr(pci_dev, net_dev);
	else
		ret = sis900_get_mac_addr(pci_dev, net_dev);

	if (ret == 0) {
		ret = -ENODEV;
		goto ↓err_out_unregister;
	}

	/* 630ET : set the mii access mode as software-mode */
	if (revision == SIS630ET_900_REV)
		outl(ACCESSMODE | inl(ioaddr + cr), ioaddr + cr);

	/* probe for mii transceiver */
	if (sis900_mii_probe(net_dev) == 0) {
		ret = -ENODEV;
		goto ↓err_out_unregister;
	}

	/* save our host bridge revision */
	dev = pci_get_device(PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_630, NULL);
	if (dev) {
		pci_read_config_byte(dev, PCI_CLASS_REVISION, &sis_priv->host_bridge_rev);
		pci_dev_put(dev);
	}

	/* print some information about our NIC */
	printk(KERN_INFO "%s: %s at %#lx, IRQ %d, ", net_dev->name,
	       card_name, ioaddr, net_dev->irq);
	for (i = 0; i < 5; i++)
		printk("%2.2x:", (u8)net_dev->dev_addr[i]);
	printk("%2.2x.\n", net_dev->dev_addr[i]);

	return 0;

 err_out_unregister:
	unregister_netdev(net_dev);
 err_unmap_rx:
	pci_free_consistent(pci_dev, RX_TOTAL_SIZE, sis_priv->rx_ring,
		sis_priv->rx_ring_dma);
 err_unmap_tx:
	pci_free_consistent(pci_dev, TX_TOTAL_SIZE, sis_priv->tx_ring,
		sis_priv->tx_ring_dma);
 err_out_cleardev:
	pci_set_drvdata(pci_dev, NULL);
	pci_release_regions(pci_dev);
 err_out:
	free_netdev(net_dev);
	return ret;
} ? end sis900_probe ?

/**
 *	sis900_mii_probe - Probe MII PHY for sis900
 *	@net_dev: the net device to probe for
 *
 *	Search for total of 32 possible mii phy addresses.
 *	Identify and set current phy if found one,
 *	return error if it failed to found.
 */

static int __init sis900_mii_probe(struct net_device * net_dev)
{
	struct sis900_private * sis_priv = net_dev->priv;
	u16 poll_bit = MII_STAT_LINK, status = 0;
	unsigned long timeout = jiffies + 5 * HZ;
	int phy_addr;
	u8 revision;

```

```c
	sis_priv->mii = NULL;

	/* search for total of 32 possible mii phy addresses */
	for (phy_addr = 0; phy_addr < 32; phy_addr++) {
		struct mii_phy * mii_phy = NULL;
		u16 mii_status;
		int i;

		mii_phy = NULL;
		for(i = 0; i < 2; i++)
			mii_status = mdio_read(net_dev, phy_addr, MII_STATUS);

		if (mii_status == 0xffff || mii_status == 0x0000)
			/* the mii is not accessible, try next one */
			continue;

		if ((mii_phy = kmalloc(sizeof(struct mii_phy), GFP_KERNEL)) == NULL) {
			printk(KERN_INFO "Cannot allocate mem for struct mii_phy\n");
			mii_phy = sis_priv->first_mii;
			while (mii_phy) {
				struct mii_phy *phy;
				phy = mii_phy;
				mii_phy = mii_phy->next;
				kfree(phy);
			}
			return 0;
		}

		mii_phy->phy_id0 = mdio_read(net_dev, phy_addr, MII_PHY_ID0);
		mii_phy->phy_id1 = mdio_read(net_dev, phy_addr, MII_PHY_ID1);
		mii_phy->phy_addr = phy_addr;
		mii_phy->status = mii_status;
		mii_phy->next = sis_priv->mii;
		sis_priv->mii = mii_phy;
		sis_priv->first_mii = mii_phy;

		for (i = 0; mii_chip_table[i].phy_id1; i++)
			if ((mii_phy->phy_id0 == mii_chip_table[i].phy_id0 ) &&
			    ((mii_phy->phy_id1 & 0xFFF0) == mii_chip_table[i].phy_id1)){
				mii_phy->phy_types = mii_chip_table[i].phy_types;
				if (mii_chip_table[i].phy_types == MIX)
					mii_phy->phy_types =
					   (mii_status & (MII_STAT_CAN_TX_FDX | MII_STAT_CAN_TX)) ? LAN : HOME;
				printk(KERN_INFO "%s: %s transceiver found at address %d.\n",
					net_dev->name, mii_chip_table[i].name,
					phy_addr);
				break;
			}

		if( !mii_chip_table[i].phy_id1 ) {
			printk(KERN_INFO "%s: Unknown PHY transceiver found at address %d.\n",
				net_dev->name, phy_addr);
			mii_phy->phy_types = UNKNOWN;
		}
	} ? end for phy_addr=0;phy_addr<3... ?

	if (sis_priv->mii == NULL) {
		printk(KERN_INFO "%s: No MII transceivers found! \n",
			net_dev->name);
		return 0;
	}

	/* select default PHY for mac */
	sis_priv->mii = NULL;
	sis900_default_phy( net_dev );

	/* Reset phy if default phy is internal sis900 */
	 if ((sis_priv->mii->phy_id0 == 0x001D) &&
	    ((sis_priv->mii->phy_id1&0xFFF0) == 0x8000))
		status = sis900_reset_phy(net_dev, sis_priv->cur_phy);

	 /* workaround for ICS1893 PHY */
	 if ((sis_priv->mii->phy_id0 == 0x0015) &&
	     ((sis_priv->mii->phy_id1&0xFFF0) == 0xF440))
		mdio_write(net_dev, sis_priv->cur_phy, 0x0018, 0xD200);

	if(status & MII_STAT_LINK){
		while (poll_bit) {
			yield();

```

```c
            poll_bit ^= (mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS) & poll_bit);
            if (time_after_eq(jiffies, timeout)) {
                printk(KERN_WARNING "%s: reset phy and link down now\n",
                    net_dev->name);
                return -ETIME;
            }
        }
    }

    pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
    if (revision == SIS630E_900_REV) {
        /* SiS 630E has some bugs on default value of PHY registers */
        mdio_write(net_dev, sis_priv->cur_phy, MII_ANADV, 0x05e1);
        mdio_write(net_dev, sis_priv->cur_phy, MII_CONFIG1, 0x22);
        mdio_write(net_dev, sis_priv->cur_phy, MII_CONFIG2, 0xff00);
        mdio_write(net_dev, sis_priv->cur_phy, MII_MASK, 0xffc0);
        //mdio_write(net_dev, sis_priv->cur_phy, MII_CONTROL, 0x1000);
    }

    if (sis_priv->mii->status & MII_STAT_LINK)
        netif_carrier_on(net_dev);
    else
        netif_carrier_off(net_dev);

    return 1;
} ? end sis900_mii_probe ?

/**
 *  sis900_default_phy - Select default PHY for sis900 mac.
 *  @net_dev: the net device to probe for
 *
 *  Select first detected PHY with link as default.
 *  If no one is link on, select PHY whose types is HOME as default.
 *  If HOME doesn't exist, select LAN.
 */

static u16 sis900_default_phy(struct net_device * net_dev)
{
    struct sis900_private * sis_priv = net_dev->priv;
    struct mii_phy *phy = NULL, *phy_home = NULL,
        *default_phy = NULL, *phy_lan = NULL;
    u16 status;

    for (phy=sis_priv->first_mii; phy; phy=phy->next) {
        status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
        status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);

        /* Link ON & Not select default PHY & not ghost PHY */
        if ((status & MII_STAT_LINK) && !default_phy &&
                    (phy->phy_types != UNKNOWN))
            default_phy = phy;
        else {
            status = mdio_read(net_dev, phy->phy_addr, MII_CONTROL);
            mdio_write(net_dev, phy->phy_addr, MII_CONTROL,
                status | MII_CNTL_AUTO | MII_CNTL_ISOLATE);
            if (phy->phy_types == HOME)
                phy_home = phy;
            else if(phy->phy_types == LAN)
                phy_lan = phy;
        }
    }

    if (!default_phy && phy_home)
        default_phy = phy_home;
    else if (!default_phy && phy_lan)
        default_phy = phy_lan;
    else if (!default_phy)
        default_phy = sis_priv->first_mii;

    if (sis_priv->mii != default_phy) {
        sis_priv->mii = default_phy;
        sis_priv->cur_phy = default_phy->phy_addr;
        printk(KERN_INFO "%s: Using transceiver found at address %d as default\n",
                    net_dev->name,sis_priv->cur_phy);
    }

    status = mdio_read(net_dev, sis_priv->cur_phy, MII_CONTROL);
    status &= (~MII_CNTL_ISOLATE);

```

```c
    mdio_write(net_dev, sis_priv->cur_phy, MII_CONTROL, status);
    status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
    status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);

    return status;
} ? end sis900_default_phy ?


/**
 *  sis900_set_capability - set the media capability of network adapter.
 *  @net_dev : the net device to probe for
 *  @phy : default PHY
 *
 *  Set the media capability of network adapter according to
 *  mii status register. It's necessary before auto-negotiate.
 */

static void sis900_set_capability(struct net_device *net_dev, struct mii_phy *phy)
{
    u16 cap;
    u16 status;

    status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);
    status = mdio_read(net_dev, phy->phy_addr, MII_STATUS);

    cap = MII_NWAY_CSMA_CD |
        ((phy->status & MII_STAT_CAN_TX_FDX)? MII_NWAY_TX_FDX:0) |
        ((phy->status & MII_STAT_CAN_TX)    ? MII_NWAY_TX:0) |
        ((phy->status & MII_STAT_CAN_T_FDX) ? MII_NWAY_T_FDX:0)|
        ((phy->status & MII_STAT_CAN_T)     ? MII_NWAY_T:0);

    mdio_write(net_dev, phy->phy_addr, MII_ANADV, cap);
}


/* Delay between EEPROM clock transitions. */
#define eeprom_delay()  inl(ee_addr)

/**
 *  read_eeprom - Read Serial EEPROM
 *  @ioaddr: base i/o address
 *  @location: the EEPROM location to read
 *
 *  Read Serial EEPROM through EEPROM Access Register.
 *  Note that location is in word (16 bits) unit
 */

static u16 __devinit read_eeprom(long ioaddr, int location)
{
    int i;
    u16 retval = 0;
    long ee_addr = ioaddr + mear;
    u32 read_cmd = location | EEread;

    outl(0, ee_addr);
    eeprom_delay();
    outl(EECS, ee_addr);
    eeprom_delay();

    /* Shift the read command (9) bits out. */
    for (i = 8; i >= 0; i--) {
        u32 dataval = (read_cmd & (1 << i)) ? EEDI | EECS : EECS;
        outl(dataval, ee_addr);
        eeprom_delay();
        outl(dataval | EECLK, ee_addr);
        eeprom_delay();
    }
    outl(EECS, ee_addr);
    eeprom_delay();

    /* read the 16-bits data in */
    for (i = 16; i > 0; i--) {
        outl(EECS, ee_addr);
        eeprom_delay();
        outl(EECS | EECLK, ee_addr);
        eeprom_delay();
        retval = (retval << 1) | ((inl(ee_addr) & EEDO) ? 1 : 0);
        eeprom_delay();
    }

    /* Terminate the EEPROM access. */
```

```
        outl(0, ee_addr);
        eeprom_delay();

        return (retval);
} ? end read_eeprom ?

/* Read and write the MII management registers using software-generated
   serial MDIO protocol. Note that the command bits and data bits are
   send out separately */
#define mdio_delay()    inl(mdio_addr)

static void mdio_idle(long mdio_addr)
{
        outl(MDIO | MDDIR, mdio_addr);
        mdio_delay();
        outl(MDIO | MDDIR | MDC, mdio_addr);
}

/* Syncronize the MII management interface by shifting 32 one bits out. */
static void mdio_reset(long mdio_addr)
{
        int i;

        for (i = 31; i >= 0; i--) {
                outl(MDDIR | MDIO, mdio_addr);
                mdio_delay();
                outl(MDDIR | MDIO | MDC, mdio_addr);
                mdio_delay();
        }
        return;
}

/**
 *      mdio_read - read MII PHY register
 *      @net_dev: the net device to read
 *      @phy_id: the phy address to read
 *      @location: the phy regiester id to read
 *
 *      Read MII registers through MDIO and MDC
 *      using MDIO management frame structure and protocol(defined by ISO/IEC).
 *      Please see SiS7014 or ICS spec
 */

static u16 mdio_read(struct net_device *net_dev, int phy_id, int location)
{
        long mdio_addr = net_dev->base_addr + mear;
        int mii_cmd = MIIread|(phy_id<<MIIpmdShift)|(location<<MIIregShift);
        u16 retval = 0;
        int i;

        mdio_reset(mdio_addr);
        mdio_idle(mdio_addr);

        for (i = 15; i >= 0; i--) {
                int dataval = (mii_cmd & (1 << i)) ? MDDIR | MDIO : MDDIR;
                outl(dataval, mdio_addr);
                mdio_delay();
                outl(dataval | MDC, mdio_addr);
                mdio_delay();
        }

        /* Read the 16 data bits. */
        for (i = 16; i > 0; i--) {
                outl(0, mdio_addr);
                mdio_delay();
                retval = (retval << 1) | ((inl(mdio_addr) & MDIO) ? 1 : 0);
                outl(MDC, mdio_addr);
                mdio_delay();
        }
        outl(0x00, mdio_addr);

        return retval;
} ? end mdio_read ?

/**
 *      mdio_write - write MII PHY register
 *      @net_dev: the net device to write
 *      @phy_id: the phy address to write
 *      @location: the phy regiester id to write
 *      @value: the register value to write with
 *
```

```
 *      Write MII registers with @value through MDIO and MDC
 *      using MDIO management frame structure and protocol(defined by ISO/IEC)
 *      please see SiS7014 or ICS spec
 */

static void mdio_write(struct net_device *net_dev, int phy_id, int location,
                        int value)
{
        long mdio_addr = net_dev->base_addr + mear;
        int mii_cmd = MIIwrite|(phy_id<<MIIpmdShift)|(location<<MIIregShift);
        int i;

        mdio_reset(mdio_addr);
        mdio_idle(mdio_addr);

        /* Shift the command bits out. */
        for (i = 15; i >= 0; i--) {
                int dataval = (mii_cmd & (1 << i)) ? MDDIR | MDIO : MDDIR;
                outb(dataval, mdio_addr);
                mdio_delay();
                outb(dataval | MDC, mdio_addr);
                mdio_delay();
        }
        mdio_delay();

        /* Shift the value bits out. */
        for (i = 15; i >= 0; i--) {
                int dataval = (value & (1 << i)) ? MDDIR | MDIO : MDDIR;
                outl(dataval, mdio_addr);
                mdio_delay();
                outl(dataval | MDC, mdio_addr);
                mdio_delay();
        }
        mdio_delay();

        /* Clear out extra bits. */
        for (i = 2; i > 0; i--) {
                outb(0, mdio_addr);
                mdio_delay();
                outb(MDC, mdio_addr);
                mdio_delay();
        }
        outl(0x00, mdio_addr);

        return;
} ? end mdio_write ?


/**
 *      sis900_reset_phy - reset sis900 mii phy.
 *      @net_dev: the net device to write
 *      @phy_addr: default phy address
 *
 *      Some specific phy can't work properly without reset.
 *      This function will be called during initialization and
 *      link status change from ON to DOWN.
 */

static u16 sis900_reset_phy(struct net_device *net_dev, int phy_addr)
{
        int i = 0;
        u16 status;

        while (i++ < 2)
                status = mdio_read(net_dev, phy_addr, MII_STATUS);

        mdio_write( net_dev, phy_addr, MII_CONTROL, MII_CNTL_RESET );

        return status;
}

/**
 *      sis900_open - open sis900 device
 *      @net_dev: the net device to open
 *
 *      Do some initialization and start net interface.
 *      enable interrupts and set sis900 timer.
 */

static int
```

```c
sis900_open(struct net_device *net_dev)
{
        struct sis900_private *sis_priv = net_dev- >priv;
        long ioaddr = net_dev- >base_addr;
        u8 revision;
        int ret;

        / * Soft reset the chip. */
        sis900_reset(net_dev);

        / * Equalizer workaround Rule */
        pci_read_config_byte(sis_priv- >pci_dev, PCI_CLASS_REVISION, &revision);
        sis630_set_eq(net_dev, revision);

        ret = request_irq(net_dev- >irq, &sis900_interrupt, SA_SHIRQ,
                                        net_dev- >name, net_dev);
        if (ret)
                return ret;

        sis900_init_rxfilter(net_dev);

        sis900_init_tx_ring(net_dev);
        sis900_init_rx_ring(net_dev);

        set_rx_mode(net_dev);

        netif_start_queue(net_dev);

        / * Workaround for EDB */
        sis900_set_mode(ioaddr, HW_SPEED_10_MBPS, FDX_CAPABLE_HALF_SELECTED);

        / * Enable all known interrupts by setting the interrupt mask. */
        outl((RxSOVR|RxORN|RxERR|RxOK|TxURN|TxERR|TxIDLE), ioaddr + imr);
        outl(RxENA | inl(ioaddr + cr), ioaddr + cr);
        outl(IE, ioaddr + ier);

        sis900_check_mode(net_dev, sis_priv- >mii);

        / * Set the timer to switch to check for link beat and perhaps switch
           to an alternate media type. */
        init_timer(&sis_priv- >timer);
        sis_priv- >timer.expires = jiffies + HZ;
        sis_priv- >timer.data = (unsigned long)net_dev;
        sis_priv- >timer.function = &sis900_timer;
        add_timer(&sis_priv- >timer);

        return 0;
} ? end sis900_open ?

/ **
 *      sis900_init_rxfilter - Initialize the Rx filter
 *      @net_dev: the net device to initialize for
 *
 *      Set receive filter address to our MAC address
 *      and enable packet filtering.
 */

static void
sis900_init_rxfilter (struct net_device * net_dev)
{
        long ioaddr = net_dev- >base_addr;
        u32 rfcrSave;
        u32 i;

        rfcrSave = inl(rfcr + ioaddr);

        / * disable packet filtering before setting filter */
        outl(rfcrSave & ~RFEN, rfcr + ioaddr);

        / * load MAC addr to filter data register */
        for (i = 0 ; i < 3 ; i++) {
                u32 w;

                w = (u32) *((u16 *)(net_dev- >dev_addr)+i);
                outl((i << RFADDR_shift), ioaddr + rfcr);
                outl(w, ioaddr + rfdr);

```

```c
                if (sis900_debug > 2) {
                        printk(KERN_INFO "%s: Receive Filter Addrss[%d]=%x\n",
                                net_dev- >name, i, inl(ioaddr + rfdr));
                }
        }

        / * enable packet filtering */
        outl(rfcrSave | RFEN, rfcr + ioaddr);
} ? end sis900_init_rxfilter ?

/ **
 *      sis900_init_tx_ring - Initialize the Tx descriptor ring
 *      @net_dev: the net device to initialize for
 *
 *      Initialize the Tx descriptor ring,
 */

static void
sis900_init_tx_ring(struct net_device *net_dev)
{
        struct sis900_private *sis_priv = net_dev- >priv;
        long ioaddr = net_dev- >base_addr;
        int i;

        sis_priv- >tx_full = 0;
        sis_priv- >dirty_tx = sis_priv- >cur_tx = 0;

        for (i = 0; i < NUM_TX_DESC; i++) {
                sis_priv- >tx_skbuff[i] = NULL;

                sis_priv- >tx_ring[i].link = sis_priv- >tx_ring_dma +
                        ((i+1)%NUM_TX_DESC)*sizeof(BufferDesc);
                sis_priv- >tx_ring[i].cmdsts = 0;
                sis_priv- >tx_ring[i].bufptr = 0;
        }

        / * load Transmit Descriptor Register */
        outl(sis_priv- >tx_ring_dma, ioaddr + txdp);
        if (sis900_debug > 2)
                printk(KERN_INFO "%s: TX descriptor register loaded with: %8.8x\n",
                        net_dev- >name, inl(ioaddr + txdp));
} ? end sis900_init_tx_ring ?

/ **
 *      sis900_init_rx_ring - Initialize the Rx descriptor ring
 *      @net_dev: the net device to initialize for
 *
 *      Initialize the Rx descriptor ring,
 *      and pre- allocate recevie buffers (socket buffer)
 */

static void
sis900_init_rx_ring(struct net_device *net_dev)
{
        struct sis900_private *sis_priv = net_dev- >priv;
        long ioaddr = net_dev- >base_addr;
        int i;

        sis_priv- >cur_rx = 0;
        sis_priv- >dirty_rx = 0;

        / * init RX descriptor */
        for (i = 0; i < NUM_RX_DESC; i++) {
                sis_priv- >rx_skbuff[i] = NULL;

                sis_priv- >rx_ring[i].link = sis_priv- >rx_ring_dma +
                        ((i+1)%NUM_RX_DESC)*sizeof(BufferDesc);
                sis_priv- >rx_ring[i].cmdsts = 0;
                sis_priv- >rx_ring[i].bufptr = 0;
        }

        / * allocate sock buffers */
        for (i = 0; i < NUM_RX_DESC; i++) {
                struct sk_buff *skb;

                if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
                        / * not enough memory for skbuff, this makes a "hole"
                          on the buffer ring, it is not clear how the
                          hardware will react to this kind of degenerated
                          buffer */
                        break;
                }
                skb->dev = net_dev;
                sis_priv->rx_skbuff[i] = skb;
                sis_priv->rx_ring[i].cmdsts = RX_BUF_SIZE;
                 sis_priv->rx_ring[i].bufptr = pci_map_single(sis_priv->pci_dev,
                        skb->tail, RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
        }
        sis_priv->dirty_rx = (unsigned int) (i - NUM_RX_DESC);

        /* load Receive Descriptor Register */
        outl(sis_priv->rx_ring_dma, ioaddr + rxdp);
        if (sis900_debug > 2)
                printk(KERN_INFO "%s: RX descriptor register loaded with: %8.8x\n",
                        net_dev->name, inl(ioaddr + rxdp));
} ? end sis900_init_rx_ring ?

/**
 *      sis630_set_eq - set phy equalizer value for 630 LAN
 *      @net_dev: the net device to set equalizer value
 *      @revision: 630 LAN revision number
 *
 *      630E equalizer workaround rule(Cyrus Huang 08/15)
 *      PHY register 14h(Test)
 *      Bit 14: 0 -- Automatically dectect (default)
 *              1 -- Manually set Equalizer filter
 *      Bit 13: 0 -- (Default)
 *              1 -- Speed up convergence of equalizer setting
 *      Bit 9 : 0 -- (Default)
 *              1 -- Disable Baseline Wander
 *      Bit 3~7   -- Equalizer filter setting
 *      Link ON: Set Bit 9, 13 to 1, Bit 14 to 0
 *      Then calculate equalizer value
 *      Then set equalizer value, and set Bit 14 to 1, Bit 9 to 0
 *      Link Off:Set Bit 13 to 1, Bit 14 to 0
 *      Calculate Equalizer value:
 *      When Link is ON and Bit 14 is 0, SIS900PHY will auto-dectect proper equalizer value.
 *      When the equalizer is stable, this value is not a fixed value. It will be within
 *      a small range(eg. 7~9). Then we get a minimum and a maximum value(eg. min=7, max=9)
 *      0 <= max <= 4  --> set equalizer to max
 *      5 <= max <= 14 --> set equalizer to max+1 or set equalizer to max+2 if max == min
 *      max >= 15      --> set equalizer to max+5 or set equalizer to max+6 if max == min
 */

static void sis630_set_eq(struct net_device *net_dev, u8 revision)
{
        struct sis900_private *sis_priv = net_dev->priv;
        u16 reg14h, eq_value=0, max_value=0, min_value=0;
        int i, maxcount=10;

        if ( !(revision == SIS630E_900_REV || revision == SIS630EA1_900_REV ||
               revision == SIS630A_900_REV || revision ==  SIS630ET_900_REV) )
                return;

        if (netif_carrier_ok(net_dev)) {
                reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
                mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
                                (0x2200 | reg14h) & 0xBFFF);
                for (i=0; i < maxcount; i++) {
                        eq_value = (0x00F8 & mdio_read(net_dev,
                                sis_priv->cur_phy, MII_RESV)) >> 3;
                        if (i == 0)
                                max_value=min_value=eq_value;
                        max_value = (eq_value > max_value) ?
                                        eq_value : max_value;
                        min_value = (eq_value < min_value) ?
                                        eq_value : min_value;
                }
                /* 630E rule to determine the equalizer value */
                if (revision == SIS630E_900_REV || revision == SIS630EA1_900_REV ||
                    revision == SIS630ET_900_REV) {
                        if (max_value < 5)
                                eq_value = max_value;
                        else if (max_value >= 5 && max_value < 15)
                                eq_value = (max_value == min_value) ?
                                        max_value+2 : max_value+1;
                        else if (max_value >= 15)
                                eq_value=(max_value == min_value) ?
                                        max_value+6 : max_value+5;
                }
                /* 630B0&B1 rule to determine the equalizer value */
```

```
                if (revision == SIS630A_900_REV &&
                    (sis_priv->host_bridge_rev == SIS630B0 ||
                     sis_priv->host_bridge_rev == SIS630B1)) {
                        if (max_value == 0)
                                eq_value = 3;
                        else
                                eq_value = (max_value + min_value + 1)/2;
                }
                /* write equalizer value and setting */
                reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
                reg14h = (reg14h & 0xFF07) | ((eq_value << 3) & 0x00F8);
                reg14h = (reg14h | 0x6000) & 0xFDFF;
                mdio_write(net_dev, sis_priv->cur_phy, MII_RESV, reg14h);
        } else {
                reg14h = mdio_read(net_dev, sis_priv->cur_phy, MII_RESV);
                if (revision == SIS630A_900_REV &&
                    (sis_priv->host_bridge_rev == SIS630B0 ||
                     sis_priv->host_bridge_rev == SIS630B1))
                        mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
                                (reg14h | 0x2200) & 0xBFFF);
                else
                        mdio_write(net_dev, sis_priv->cur_phy, MII_RESV,
                                (reg14h | 0x2000) & 0xBFFF);
        }
        return;
} ? end sis630_set_eq ?

/**
 *      sis900_timer - sis900 timer routine
 *      @data: pointer to sis900 net device
 *
 *      On each timer ticks we check two things,
 *      link status (ON/OFF) and link mode (10/100/Full/Half)
 */

static void sis900_timer(unsigned long data)
{
        struct net_device *net_dev = (struct net_device *)data;
        struct sis900_private *sis_priv = net_dev->priv;
        struct mii_phy *mii_phy = sis_priv->mii;
        static int next_tick = 5*HZ;
        u16 status;
        u8 revision;

        if (!sis_priv->autong_complete){
                int speed, duplex = 0;

                sis900_read_mode(net_dev, &speed, &duplex);
                if (duplex){
                        sis900_set_mode(net_dev->base_addr, speed, duplex);
                        pci_read_config_byte(sis_priv->pci_dev,
                                        PCI_CLASS_REVISION, &revision);
                        sis630_set_eq(net_dev, revision);
                        netif_start_queue(net_dev);
                }

                sis_priv->timer.expires = jiffies + HZ;
                add_timer(&sis_priv->timer);
                return;
        }

        status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);
        status = mdio_read(net_dev, sis_priv->cur_phy, MII_STATUS);

        /* Link OFF -> ON */
        if (!netif_carrier_ok(net_dev)) {
        LookForLink:
                /* Search for new PHY */
                status = sis900_default_phy(net_dev);
                mii_phy = sis_priv->mii;

                if (status & MII_STAT_LINK){
                        sis900_check_mode(net_dev, mii_phy);
                        netif_carrier_on(net_dev);
                }
        } else {
        /* Link ON -> OFF */
```

```c
		if (!(status & MII_STAT_LINK)){
			netif_carrier_off(net_dev);
			printk(KERN_INFO "%s: Media Link Off\n", net_dev->name);

			/* Change mode issue */
			if ((mii_phy->phy_id0 == 0x001D) &&
			    ((mii_phy->phy_id1 & 0xFFF0) == 0x8000))
					sis900_reset_phy(net_dev,  sis_priv->cur_phy);

			pci_read_config_byte(sis_priv->pci_dev,
					PCI_CLASS_REVISION, &revision);
			sis630_set_eq(net_dev, revision);

			goto ↑LookForLink;
		}
	}

	sis_priv->timer.expires = jiffies + next_tick;
	add_timer(&sis_priv->timer);
} ? end sis900_timer ?

/**
 *	sis900_check_mode - check the media mode for sis900
 *	@net_dev: the net device to be checked
 *	@mii_phy: the mii phy
 *
 *	Older driver gets the media mode from mii status output
 *	register. Now we set our media capability and auto-negotiate
 *	to get the upper bound of speed and duplex between two ends.
 *	If the types of mii phy is HOME, it doesn't need to auto-negotiate
 *	and autong_complete should be set to 1.
 */

static void sis900_check_mode(struct net_device *net_dev, struct mii_phy *mii_phy)
{
	struct sis900_private *sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;
	int speed, duplex;

	if (mii_phy->phy_types == LAN) {
		outl(~EXD & inl(ioaddr + cfg), ioaddr + cfg);
		sis900_set_capability(net_dev , mii_phy);
		sis900_auto_negotiate(net_dev, sis_priv->cur_phy);
	} else {
		outl(EXD | inl(ioaddr + cfg), ioaddr + cfg);
		speed = HW_SPEED_HOME;
		duplex = FDX_CAPABLE_HALF_SELECTED;
		sis900_set_mode(ioaddr, speed, duplex);
		sis_priv->autong_complete = 1;
	}
}

/**
 *	sis900_set_mode - Set the media mode of mac register.
 *	@ioaddr: the address of the device
 *	@speed : the transmit speed to be determined
 *	@duplex: the duplex mode to be determined
 *
 *	Set the media mode of mac register txcfg/rxcfg according to
 *	speed and duplex of phy. Bit EDB_MASTER_EN indicates the EDB
 *	bus is used instead of PCI bus. When this bit is set 1, the
 *	Max DMA Burst Size for TX/RX DMA should be no larger than 16
 *	double words.
 */

static void sis900_set_mode (long ioaddr, int speed, int duplex)
{
	u32 tx_flags = 0, rx_flags = 0;

	if (inl(ioaddr + cfg) & EDB_MASTER_EN) {
		tx_flags = TxATP | (DMA_BURST_64 << TxMXDMA_shift) |
				(TX_FILL_THRESH << TxFILLT_shift);
		rx_flags = DMA_BURST_64 << RxMXDMA_shift;
	} else {
		tx_flags = TxATP | (DMA_BURST_512 << TxMXDMA_shift) |
				(TX_FILL_THRESH << TxFILLT_shift);
		rx_flags = DMA_BURST_512 << RxMXDMA_shift;
	}

```

```c
	if (speed == HW_SPEED_HOME || speed == HW_SPEED_10_MBPS) {
		rx_flags |= (RxDRNT_10 << RxDRNT_shift);
		tx_flags |= (TxDRNT_10 << TxDRNT_shift);
	} else {
		rx_flags |= (RxDRNT_100 << RxDRNT_shift);
		tx_flags |= (TxDRNT_100 << TxDRNT_shift);
	}

	if (duplex == FDX_CAPABLE_FULL_SELECTED) {
		tx_flags |= (TxCSI | TxHBI);
		rx_flags |= RxATX;
	}

	outl (tx_flags, ioaddr + txcfg);
	outl (rx_flags, ioaddr + rxcfg);
} ? end sis900_set_mode ?

/**
 *	sis900_auto_negotiate - Set the Auto-Negotiation Enable/Reset bit.
 *	@net_dev: the net device to read mode for
 *	@phy_addr: mii phy address
 *
 *	If the adapter is link-on, set the auto-negotiate enable/reset bit.
 *	autong_complete should be set to 0 when starting auto-negotiation.
 *	autong_complete should be set to 1 if we didn't start auto-negotiation.
 *	sis900_timer will wait for link on again if autong_complete = 0.
 */

static void sis900_auto_negotiate(struct net_device *net_dev, int phy_addr)
{
	struct sis900_private *sis_priv = net_dev->priv;
	int i = 0;
	u32 status;

	while (i++ < 2)
		status = mdio_read(net_dev, phy_addr, MII_STATUS);

	if (!(status & MII_STAT_LINK)){
		printk(KERN_INFO "%s: Media Link Off\n", net_dev->name);
		sis_priv->autong_complete = 1;
		netif_carrier_off(net_dev);
		return;
	}

	/* (Re)start AutoNegotiate */
	mdio_write(net_dev, phy_addr, MII_CONTROL,
		   MII_CNTL_AUTO | MII_CNTL_RST_AUTO);
	sis_priv->autong_complete = 0;
} ? end sis900_auto_negotiate ?


/**
 *	sis900_read_mode - read media mode for sis900 internal phy
 *	@net_dev: the net device to read mode for
 *	@speed  : the transmit speed to be determined
 *	@duplex : the duplex mode to be determined
 *
 *	The capability of remote end will be put in mii register autorec
 *	after auto-negotiation. Use AND operation to get the upper bound
 *	of speed and duplex between two ends.
 */

static void sis900_read_mode(struct net_device *net_dev, int *speed, int *duplex)
{
	struct sis900_private *sis_priv = net_dev->priv;
	struct mii_phy *phy = sis_priv->mii;
	int phy_addr = sis_priv->cur_phy;
	u32 status;
	u16 autoadv, autorec;
	int i = 0;

	while (i++ < 2)
		status = mdio_read(net_dev, phy_addr, MII_STATUS);

	if (!(status & MII_STAT_LINK))
		return;

	/* AutoNegotiate completed */
	autoadv = mdio_read(net_dev, phy_addr, MII_ANADV);
	autorec = mdio_read(net_dev, phy_addr, MII_ANLPAR);
	status = autoadv & autorec;

	*speed = HW_SPEED_10_MBPS;
	*duplex = FDX_CAPABLE_HALF_SELECTED;

	if (status & (MII_NWAY_TX | MII_NWAY_TX_FDX))
		*speed = HW_SPEED_100_MBPS;
	if (status & ( MII_NWAY_TX_FDX | MII_NWAY_T_FDX))
		*duplex = FDX_CAPABLE_FULL_SELECTED;

	sis_priv->autong_complete = 1;

	/* Workaround for Realtek RTL8201 PHY issue */
	if ((phy->phy_id0 == 0x0000) && ((phy->phy_id1 & 0xFFF0) == 0x8200)) {
		if (mdio_read(net_dev, phy_addr, MII_CONTROL) & MII_CNTL_FDX)
			*duplex = FDX_CAPABLE_FULL_SELECTED;
		if (mdio_read(net_dev, phy_addr, 0x0019) & 0x01)
			*speed = HW_SPEED_100_MBPS;
	}

	printk(KERN_INFO "%s: Media Link On %s %s-duplex \n",
					net_dev->name,
					*speed == HW_SPEED_100_MBPS ?
						"100mbps" : "10mbps",
					*duplex == FDX_CAPABLE_FULL_SELECTED ?
						"full" : "half");
} ? end sis900_read_mode ?

/**
 *	sis900_tx_timeout - sis900 transmit timeout routine
 *	@net_dev: the net device to transmit
 *
 *	print transmit timeout status
 *	disable interrupts and do some tasks
 */

static void sis900_tx_timeout(struct net_device *net_dev)
{
	struct sis900_private *sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;
	unsigned long flags;
	int i;

	printk(KERN_INFO "%s: Transmit timeout, status %8.8x %8.8x \n",
		net_dev->name, inl(ioaddr + cr), inl(ioaddr + isr));

	/* Disable interrupts by clearing the interrupt mask. */
	outl(0x0000, ioaddr + imr);

	/* use spinlock to prevent interrupt handler accessing buffer ring */
	spin_lock_irqsave(&sis_priv->lock, flags);

	/* discard unsent packets */
	sis_priv->dirty_tx = sis_priv->cur_tx = 0;
	for (i = 0; i < NUM_TX_DESC; i++) {
		struct sk_buff *skb = sis_priv->tx_skbuff[i];

		if (skb) {
			pci_unmap_single(sis_priv->pci_dev,
				sis_priv->tx_ring[i].bufptr, skb->len,
				PCI_DMA_TODEVICE);
			dev_kfree_skb_irq(skb);
			sis_priv->tx_skbuff[i] = NULL;
			sis_priv->tx_ring[i].cmdsts = 0;
			sis_priv->tx_ring[i].bufptr = 0;
			sis_priv->stats.tx_dropped++;
		}
	}
	sis_priv->tx_full = 0;
	netif_wake_queue(net_dev);

	spin_unlock_irqrestore(&sis_priv->lock, flags);

	net_dev->trans_start = jiffies;

	/* load Transmit Descriptor Register */
	outl(sis_priv->tx_ring_dma, ioaddr + txdp);

	/* Enable all known interrupts by setting the interrupt mask. */
```

```
	outl((RxSOVR|RxORN|RxERR|RxOK|TxURN|TxERR|TxIDLE), ioaddr + imr);
	return;
} ? end sis900_tx_timeout ?

/**
 *	sis900_start_xmit - sis900 start transmit routine
 *	@skb: socket buffer pointer to put the data being transmitted
 *	@net_dev: the net device to transmit with
 *
 *	Set the transmit buffer descriptor,
 *	and write TxENA to enable transmit state machine.
 *	tell upper layer if the buffer is full
 */

static int
sis900_start_xmit(struct sk_buff *skb, struct net_device *net_dev)
{
	struct sis900_private *sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;
	unsigned int  entry;
	unsigned long flags;
	unsigned int  index_cur_tx, index_dirty_tx;
	unsigned int  count_dirty_tx;

	/* Don't transmit data before the complete of auto-negotiation */
	if(!sis_priv->autong_complete){
		netif_stop_queue(net_dev);
		return 1;
	}

	spin_lock_irqsave(&sis_priv->lock, flags);

	/* Calculate the next Tx descriptor entry. */
	entry = sis_priv->cur_tx % NUM_TX_DESC;
	sis_priv->tx_skbuff[entry] = skb;

	/* set the transmit buffer descriptor and enable Transmit State Machine */
	sis_priv->tx_ring[entry].bufptr = pci_map_single(sis_priv->pci_dev,
		skb->data, skb->len, PCI_DMA_TODEVICE);
	sis_priv->tx_ring[entry].cmdsts = (OWN | skb->len);
	outl(TxENA | inl(ioaddr + cr), ioaddr + cr);

	sis_priv->cur_tx ++;
	index_cur_tx = sis_priv->cur_tx;
	index_dirty_tx = sis_priv->dirty_tx;

	for (count_dirty_tx = 0; index_cur_tx != index_dirty_tx; index_dirty_tx++)
		count_dirty_tx ++;

	if (index_cur_tx == index_dirty_tx) {
		/* dirty_tx is met in the cycle of cur_tx, buffer full */
		sis_priv->tx_full = 1;
		netif_stop_queue(net_dev);
	} else if (count_dirty_tx < NUM_TX_DESC) {
		/* Typical path, tell upper layer that more transmission is possible */
		netif_start_queue(net_dev);
	} else {
		/* buffer full, tell upper layer no more transmission */
		sis_priv->tx_full = 1;
		netif_stop_queue(net_dev);
	}

	spin_unlock_irqrestore(&sis_priv->lock, flags);

	net_dev->trans_start = jiffies;

	if (sis900_debug > 3)
		printk(KERN_INFO "%s: Queued Tx packet at %p size %d "
			"to slot %d.\n",
			net_dev->name, skb->data, (int)skb->len, entry);

	return 0;
} ? end sis900_start_xmit ?

/**
 *	sis900_interrupt - sis900 interrupt handler
 *	@irq: the irq number
 *	@dev_instance: the client data object
 *	@regs: snapshot of processor context
 *
 *	The interrupt handler does all of the Rx thread work,
```

```c
 *	and cleans up after the Tx thread
 */

static irqreturn_t sis900_interrupt(int irq, void *dev_instance, struct pt_regs *regs)
{
	struct net_device *net_dev = dev_instance;
	struct sis900_private *sis_priv = net_dev->priv;
	int boguscnt = max_interrupt_work;
	long ioaddr = net_dev->base_addr;
	u32 status;
	unsigned int handled = 0;

	spin_lock (&sis_priv->lock);

	do {
		status = inl(ioaddr + isr);

		if ((status & (HIBERR|TxURN|TxERR|TxIDLE|RxORN|RxERR|RxOK)) == 0)
			/* nothing intresting happened */
			break;
		handled = 1;

		/* why dow't we break after Tx/Rx case ?? keyword: full-duplex */
		if (status & (RxORN | RxERR | RxOK))
			/* Rx interrupt */
			sis900_rx(net_dev);

		if (status & (TxURN | TxERR | TxIDLE))
			/* Tx interrupt */
			sis900_finish_xmit(net_dev);

		/* something strange happened !!! */
		if (status & HIBERR) {
			printk(KERN_INFO "%s: Abnormal interrupt,"
				"status %#8.8x.\n", net_dev->name, status);
			break;
		}
		if (--boguscnt < 0) {
			printk(KERN_INFO "%s: Too much work at interrupt, "
				"interrupt status = %#8.8x.\n",
				net_dev->name, status);
			break;
		}
	} ? end do ? while (1);

	if (sis900_debug > 3)
		printk(KERN_INFO "%s: exiting interrupt, "
			"interrupt status = 0x%#8.8x.\n",
			net_dev->name, inl(ioaddr + isr));

	spin_unlock (&sis_priv->lock);
	return IRQ_RETVAL(handled);
} ? end sis900_interrupt ?

/**
 *	sis900_rx - sis900 receive routine
 *	@net_dev: the net device which receives data
 *
 *	Process receive interrupt events,
 *	put buffer to higher layer and refill buffer pool
 *	Note: This fucntion is called by interrupt handler,
 *	don't do "too much" work here
 */

static int sis900_rx(struct net_device *net_dev)
{
	struct sis900_private *sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;
	unsigned int entry = sis_priv->cur_rx % NUM_RX_DESC;
	u32 rx_status = sis_priv->rx_ring[entry].cmdsts;

	if (sis900_debug > 3)
		printk(KERN_INFO "sis900_rx, cur_rx:%4.4d, dirty_rx:%4.4d "
			"status:0x%8.8x\n",
			sis_priv->cur_rx, sis_priv->dirty_rx, rx_status);

	while (rx_status & OWN) {
		unsigned int rx_size;

		rx_size = (rx_status & DSIZE) - CRC_SIZE;

```

```c
		if (rx_status & (ABORT|OVERRUN|TOOLONG|RUNT|RXISERR|CRCERR|FAERR)) {
			/* corrupted packet received */
			if (sis900_debug > 3)
				printk(KERN_INFO "%s: Corrupted packet "
					"received, buffer status = 0x%8.8x.\n",
					net_dev->name, rx_status);
			sis_priv->stats.rx_errors++;
			if (rx_status & OVERRUN)
				sis_priv->stats.rx_over_errors++;
			if (rx_status & (TOOLONG|RUNT))
				sis_priv->stats.rx_length_errors++;
			if (rx_status & (RXISERR | FAERR))
				sis_priv->stats.rx_frame_errors++;
			if (rx_status & CRCERR)
				sis_priv->stats.rx_crc_errors++;
			/* reset buffer descriptor state */
			sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
		} else {
			struct sk_buff * skb;

			/* This situation should never happen, but due to
			   some unknow bugs, it is possible that
			   we are working on NULL sk_buff :-( */
			if (sis_priv->rx_skbuff[entry] == NULL) {
				printk(KERN_INFO "%s: NULL pointer "
					"encountered in Rx ring, skipping\n",
					net_dev->name);
				break;
			}

			pci_unmap_single(sis_priv->pci_dev,
				sis_priv->rx_ring[entry].bufptr, RX_BUF_SIZE,
				PCI_DMA_FROMDEVICE);
			/* give the socket buffer to upper layers */
			skb = sis_priv->rx_skbuff[entry];
			skb_put(skb, rx_size);
			skb->protocol = eth_type_trans(skb, net_dev);
			netif_rx(skb);

			/* some network statistics */
			if ((rx_status & BCAST) == MCAST)
				sis_priv->stats.multicast++;
			net_dev->last_rx = jiffies;
			sis_priv->stats.rx_bytes += rx_size;
			sis_priv->stats.rx_packets++;

			/* refill the Rx buffer, what if there is not enought
			 * memory for new socket buffer ?? */
			if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
				/* not enough memory for skbuff, this makes a
				 * "hole" on the buffer ring, it is not clear
				 * how the hardware will react to this kind
				 * of degenerated buffer */
				printk(KERN_INFO "%s: Memory squeeze,"
					"deferring packet.\n",
					net_dev->name);
				sis_priv->rx_skbuff[entry] = NULL;
				/* reset buffer descriptor state */
				sis_priv->rx_ring[entry].cmdsts = 0;
				sis_priv->rx_ring[entry].bufptr = 0;
				sis_priv->stats.rx_dropped++;
				break;
			}
			skb->dev = net_dev;
			sis_priv->rx_skbuff[entry] = skb;
			sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
			sis_priv->rx_ring[entry].bufptr =
				pci_map_single(sis_priv->pci_dev, skb->tail,
					RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
			sis_priv->dirty_rx++;
		} ? end else ?
		sis_priv->cur_rx++;
		entry = sis_priv->cur_rx % NUM_RX_DESC;
		rx_status = sis_priv->rx_ring[entry].cmdsts;
	} ? end while rx_status&OWN ? // while

	/* refill the Rx buffer, what if the rate of refilling is slower
	 * than consuming ?? */
	for (;sis_priv->cur_rx - sis_priv->dirty_rx > 0; sis_priv->dirty_rx++) {
		struct sk_buff *skb;

```

```c
                entry = sis_priv->dirty_rx % NUM_RX_DESC;

                if (sis_priv->rx_skbuff[entry] == NULL) {
                        if ((skb = dev_alloc_skb(RX_BUF_SIZE)) == NULL) {
                                /* not enough memory for skbuff, this makes a
                                 * "hole" on the buffer ring, it is not clear
                                 * how the hardware will react to this kind
                                 * of degenerated buffer */
                                printk(KERN_INFO "%s: Memory squeeze,"
                                        "deferring packet.\n",
                                        net_dev->name);
                                sis_priv->stats.rx_dropped++;
                                break;
                        }
                        skb->dev = net_dev;
                        sis_priv->rx_skbuff[entry] = skb;
                        sis_priv->rx_ring[entry].cmdsts = RX_BUF_SIZE;
                        sis_priv->rx_ring[entry].bufptr =
                                pci_map_single(sis_priv->pci_dev, skb->tail,
                                        RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
                }
        } ? end for ;sis_priv->cur_rx-sis... ?
        /* re-enable the potentially idle receive state matchine */
        outl(RxENA | inl(ioaddr + cr), ioaddr + cr );

        return 0;
} ? end sis900_rx ?

/**
 *      sis900_finish_xmit - finish up transmission of packets
 *      @net_dev: the net device to be transmitted on
 *
 *      Check for error condition and free socket buffer etc
 *      schedule for more transmission as needed
 *      Note: This fucntion is called by interrupt handler,
 *      don't do "too much" work here
 */

static void sis900_finish_xmit (struct net_device *net_dev)
{
        struct sis900_private *sis_priv = net_dev->priv;

        for (; sis_priv->dirty_tx != sis_priv->cur_tx; sis_priv->dirty_tx++) {
                struct sk_buff *skb;
                unsigned int entry;
                u32 tx_status;

                entry = sis_priv->dirty_tx % NUM_TX_DESC;
                tx_status = sis_priv->tx_ring[entry].cmdsts;

                if (tx_status & OWN) {
                        /* The packet is not transmitted yet (owned by hardware) !
                         * Note: the interrupt is generated only when Tx Machine
                         * is idle, so this is an almost impossible case */
                        break;
                }

                if (tx_status & (ABORT | UNDERRUN | OWCOLL)) {
                        /* packet unsuccessfully transmitted */
                        if (sis900_debug > 3)
                                printk(KERN_INFO "%s: Transmit "
                                        "error, Tx status %8.8x.\n",
                                        net_dev->name, tx_status);
                        sis_priv->stats.tx_errors++;
                        if (tx_status & UNDERRUN)
                                sis_priv->stats.tx_fifo_errors++;
                        if (tx_status & ABORT)
                                sis_priv->stats.tx_aborted_errors++;
                        if (tx_status & NOCARRIER)
                                sis_priv->stats.tx_carrier_errors++;
                        if (tx_status & OWCOLL)
                                sis_priv->stats.tx_window_errors++;
                } else {
                        /* packet successfully transmitted */
                        sis_priv->stats.collisions += (tx_status & COLCNT) >> 16;
                        sis_priv->stats.tx_bytes += tx_status & DSIZE;
                        sis_priv->stats.tx_packets++;
                }
                /* Free the original skb. */
```

```c
                skb = sis_priv->tx_skbuff[entry];
                pci_unmap_single(sis_priv->pci_dev,
                        sis_priv->tx_ring[entry].bufptr, skb->len,
                        PCI_DMA_TODEVICE);
                dev_kfree_skb_irq(skb);
                sis_priv->tx_skbuff[entry] = NULL;
                sis_priv->tx_ring[entry].bufptr = 0;
                sis_priv->tx_ring[entry].cmdsts = 0;
        } ? end for ;sis_priv->dirty_tx!=... ?

        if (sis_priv->tx_full && netif_queue_stopped(net_dev) &&
            sis_priv->cur_tx - sis_priv->dirty_tx < NUM_TX_DESC - 4) {
                /* The ring is no longer full, clear tx_full and schedule
                 * more transmission by netif_wake_queue(net_dev) */
                sis_priv->tx_full = 0;
                netif_wake_queue (net_dev);
        }
} ? end sis900_finish_xmit ?

/**
 *      sis900_close - close sis900 device
 *      @net_dev: the net device to be closed
 *
 *      Disable interrupts, stop the Tx and Rx Status Machine
 *      free Tx and RX socket buffer
 */

static int sis900_close(struct net_device *net_dev)
{
        long ioaddr = net_dev->base_addr;
        struct sis900_private *sis_priv = net_dev->priv;
        struct sk_buff *skb;
        int i;

        netif_stop_queue(net_dev);

        /* Disable interrupts by clearing the interrupt mask. */
        outl(0x0000, ioaddr + imr);
        outl(0x0000, ioaddr + ier);

        /* Stop the chip's Tx and Rx Status Machine */
        outl(RxDIS | TxDIS | inl(ioaddr + cr), ioaddr + cr);

        del_timer(&sis_priv->timer);

        free_irq(net_dev->irq, net_dev);

        /* Free Tx and RX skbuff */
        for (i = 0; i < NUM_RX_DESC; i++) {
                skb = sis_priv->rx_skbuff[i];
                if (skb) {
                        pci_unmap_single(sis_priv->pci_dev,
                                sis_priv->rx_ring[i].bufptr,
                                RX_BUF_SIZE, PCI_DMA_FROMDEVICE);
                        dev_kfree_skb(skb);
                        sis_priv->rx_skbuff[i] = NULL;
                }
        }
        for (i = 0; i < NUM_TX_DESC; i++) {
                skb = sis_priv->tx_skbuff[i];
                if (skb) {
                        pci_unmap_single(sis_priv->pci_dev,
                                sis_priv->tx_ring[i].bufptr, skb->len,
                                PCI_DMA_TODEVICE);
                        dev_kfree_skb(skb);
                        sis_priv->tx_skbuff[i] = NULL;
                }
        }

        /* Green! Put the chip in low-power mode. */

        return 0;
} ? end sis900_close ?

/**
 *      sis900_get_drvinfo - Return information about driver
 *      @net_dev: the net device to probe
 *      @info: container for info returned
 *
 *      Process ethtool command such as "ethtool -i" to show information
 */

```

```c
static void sis900_get_drvinfo(struct net_device *net_dev,
                               struct ethtool_drvinfo *info)
{
        struct sis900_private *sis_priv = net_dev->priv;

        strcpy (info->driver, SIS900_MODULE_NAME);
        strcpy (info->version, SIS900_DRV_VERSION);
        strcpy (info->bus_info, pci_name(sis_priv->pci_dev));
}

static struct ethtool_ops sis900_ethtool_ops = {
        .get_drvinfo =          sis900_get_drvinfo,
};

/**
 *      mii_ioctl - process MII i/o control command
 *      @net_dev: the net device to command for
 *      @rq: parameter for command
 *      @cmd: the i/o command
 *
 *      Process MII command like read/write MII register
 */

static int mii_ioctl(struct net_device *net_dev, struct ifreq *rq, int cmd)
{
        struct sis900_private *sis_priv = net_dev->priv;
        struct mii_ioctl_data *data = if_mii(rq);

        switch(cmd) {
        case SIOCGMIIPHY:               /* Get address of MII PHY in use. */
                data->phy_id = sis_priv->mii->phy_addr;
                /* Fall Through */

        case SIOCGMIIREG:               /* Read MII PHY register. */
                data->val_out = mdio_read(net_dev, data->phy_id & 0x1f, data->reg_num & 0x1f);
                return 0;

        case SIOCSMIIREG:               /* Write MII PHY register. */
                if (!capable(CAP_NET_ADMIN))
                        return -EPERM;
                mdio_write(net_dev, data->phy_id & 0x1f, data->reg_num & 0x1f, data->val_in);
                return 0;
        default:
                return -EOPNOTSUPP;
        }
} ? end mii_ioctl ?

/**
 *      sis900_get_stats - Get sis900 read/write statistics
 *      @net_dev: the net device to get statistics for
 *
 *      get tx/rx statistics for sis900
 */

static struct net_device_stats *
sis900_get_stats(struct net_device *net_dev)
{
        struct sis900_private *sis_priv = net_dev->priv;

        return &sis_priv->stats;
}

/**
 *      sis900_set_config - Set media type by net_device.set_config
 *      @dev: the net device for media type change
 *      @map: ifmap passed by ifconfig
 *
 *      Set media type to 10baseT, 100baseT or 0(for auto) by ifconfig
 *      we support only port changes. All other runtime configuration
 *      changes will be ignored
 */

static int sis900_set_config(struct net_device *dev, struct ifmap *map)
{
        struct sis900_private *sis_priv = dev->priv;
        struct mii_phy *mii_phy = sis_priv->mii;

        u16 status;

```

```c
        if ((map->port != (u_char)(-1)) && (map->port != dev->if_port)) {
                /* we switch on the ifmap->port field. I couldn't find anything
                 * like a definition or standard for the values of that field.
                 * I think the meaning of those values is device specific. But
                 * since I would like to change the media type via the ifconfig
                 * command I use the definition from linux/netdevice.h
                 * (which seems to be different from the ifport(pcmcia) definition) */
                switch(map->port){
                case IF_PORT_UNKNOWN: /* use auto here */
                        dev->if_port = map->port;
                        /* we are going to change the media type, so the Link
                         * will be temporary down and we need to reflect that
                         * here. When the Link comes up again, it will be
                         * sensed by the sis_timer procedure, which also does
                         * all the rest for us */
                        netif_carrier_off(dev);

                        /* read current state */
                        status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);

                        /* enable auto negotiation and reset the negotioation
                         * (I don't really know what the auto negatiotiation
                         * reset really means, but it sounds for me right to
                         * do one here) */
                        mdio_write(dev, mii_phy->phy_addr,
                                MII_CONTROL, status | MII_CNTL_AUTO | MII_CNTL_RST_AUTO);

                        break;

                case IF_PORT_10BASET: /* 10BaseT */
                        dev->if_port = map->port;

                        /* we are going to change the media type, so the Link
                         * will be temporary down and we need to reflect that
                         * here. When the Link comes up again, it will be
                         * sensed by the sis_timer procedure, which also does
                         * all the rest for us */
                        netif_carrier_off(dev);

                        /* set Speed to 10Mbps */
                        /* read current state */
                        status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);

                        /* disable auto negotiation and force 10MBit mode*/
                        mdio_write(dev, mii_phy->phy_addr,
                                MII_CONTROL, status & ~(MII_CNTL_SPEED |
                                        MII_CNTL_AUTO));
                        break;

                case IF_PORT_100BASET: /* 100BaseT */
                case IF_PORT_100BASETX: /* 100BaseTx */
                        dev->if_port = map->port;

                        /* we are going to change the media type, so the Link
                         * will be temporary down and we need to reflect that
                         * here. When the Link comes up again, it will be
                         * sensed by the sis_timer procedure, which also does
                         * all the rest for us */
                        netif_carrier_off(dev);

                        /* set Speed to 100Mbps */
                        /* disable auto negotiation and enable 100MBit Mode */
                        status = mdio_read(dev, mii_phy->phy_addr, MII_CONTROL);
                        mdio_write(dev, mii_phy->phy_addr,
                                MII_CONTROL, (status & ~MII_CNTL_SPEED) |
                                MII_CNTL_SPEED);

                        break;

                case IF_PORT_10BASE2: /* 10Base2 */
                case IF_PORT_AUI: /* AUI */
                case IF_PORT_100BASEFX: /* 100BaseFx */
                        /* These Modes are not supported (are they?)*/
                        printk(KERN_INFO "Not supported");
                        return -EOPNOTSUPP;
                        break;

                default:
```

```c
			printk(KERN_INFO "Invalid");
			return -EINVAL;
		} ? end switch map->port ?
	} ? end if (map->port!=(u_char)(... ?
	return 0;
} ? end sis900_set_config ?

/**
 *	sis900_mcast_bitnr - compute hashtable index
 *	@addr: multicast address
 *	@revision: revision id of chip
 *
 *	SiS 900 uses the most sigificant 7 bits to index a 128 bits multicast
 *	hash table, which makes this function a little bit different from other drivers
 *	SiS 900 B0 & 635 M/B uses the most significat 8 bits to index 256 bits
 *	multicast hash table.
 */

static inline u16 sis900_mcast_bitnr(u8 *addr, u8 revision)
{

	u32 crc = ether_crc(6, addr);

	/* leave 8 or 7 most siginifant bits */
	if ((revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV))
		return ((int)(crc >> 24));
	else
		return ((int)(crc >> 25));
}

/**
 *	set_rx_mode - Set SiS900 receive mode
 *	@net_dev: the net device to be set
 *
 *	Set SiS900 receive mode for promiscuous, multicast, or broadcast mode.
 *	And set the appropriate multicast filter.
 *	Multicast hash table changes from 128 to 256 bits for 635M/B & 900B0.
 */

static void set_rx_mode(struct net_device *net_dev)
{
	long ioaddr = net_dev->base_addr;
	struct sis900_private * sis_priv = net_dev->priv;
	u16 mc_filter[16] = {0};	/* 256/128 bits multicast hash table */
	int i, table_entries;
	u32 rx_mode;
	u8 revision;

	/* 635 Hash Table entires = 256(2^16) */
	pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
	if((revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV))
		table_entries = 16;
	else
		table_entries = 8;

	if (net_dev->flags & IFF_PROMISC) {
		/* Accept any kinds of packets */
		rx_mode = RFPromiscuous;
		for (i = 0; i < table_entries; i++)
			mc_filter[i] = 0xffff;
	} else if ((net_dev->mc_count > multicast_filter_limit) ||
		  (net_dev->flags & IFF_ALLMULTI)) {
		/* too many multicast addresses or accept all multicast packet */
		rx_mode = RFAAB | RFAAM;
		for (i = 0; i < table_entries; i++)
			mc_filter[i] = 0xffff;
	} else {
		/* Accept Broadcast packet, destination address matchs our
		 * MAC address, use Receive Filter to reject unwanted MCAST
		 * packets */
		struct dev_mc_list *mclist;
		rx_mode = RFAAB;
		for (i = 0, mclist = net_dev->mc_list;
			mclist && i < net_dev->mc_count;
			i++, mclist = mclist->next) {
			unsigned int bit_nr =
				sis900_mcast_bitnr(mclist->dmi_addr, revision);
			mc_filter[bit_nr >> 4] |= (1 << (bit_nr & 0xf));
		}
	}

```

```c
	/* update Multicast Hash Table in Receive Filter */
	for (i = 0; i < table_entries; i++) {
		/* why plus 0x04 ??, That makes the correct value for hash table. */
		outl((u32)(0x00000004+i) << RFADDR_shift, ioaddr + rfcr);
		outl(mc_filter[i], ioaddr + rfdr);
	}

	outl(RFEN | rx_mode, ioaddr + rfcr);

	/* sis900 is capable of looping back packets at MAC level for
	 * debugging purpose */
	if (net_dev->flags & IFF_LOOPBACK) {
		u32 cr_saved;
		/* We must disable Tx/Rx before setting loopback mode */
		cr_saved = inl(ioaddr + cr);
		outl(cr_saved | TxDIS | RxDIS, ioaddr + cr);
		/* enable loopback */
		outl(inl(ioaddr + txcfg) | TxMLB, ioaddr + txcfg);
		outl(inl(ioaddr + rxcfg) | RxATX, ioaddr + rxcfg);
		/* restore cr */
		outl(cr_saved, ioaddr + cr);
	}

	return;
} ? end set_rx_mode ?

/**
 *	sis900_reset - Reset sis900 MAC
 *	@net_dev: the net device to reset
 *
 *	reset sis900 MAC and wait until finished
 *	reset through command register
 *	change backoff algorithm for 900B0 & 635 M/B
 */

static void sis900_reset(struct net_device *net_dev)
{
	struct sis900_private * sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;
	int i = 0;
	u32 status = TxRCMP | RxRCMP;
	u8  revision;

	outl(0, ioaddr + ier);
	outl(0, ioaddr + imr);
	outl(0, ioaddr + rfcr);

	outl(RxRESET | TxRESET | RESET | inl(ioaddr + cr), ioaddr + cr);

	/* Check that the chip has finished the reset. */
	while (status && (i++ < 1000)) {
		status ^= (inl(isr + ioaddr) & status);
	}

	pci_read_config_byte(sis_priv->pci_dev, PCI_CLASS_REVISION, &revision);
	if( (revision >= SIS635A_900_REV) || (revision == SIS900B_900_REV) )
		outl(PESEL | RND_CNT, ioaddr + cfg);
	else
		outl(PESEL, ioaddr + cfg);
} ? end sis900_reset ?

/**
 *	sis900_remove - Remove sis900 device
 *	@pci_dev: the pci device to be removed
 *
 *	remove and release SiS900 net device
 */

static void __devexit sis900_remove(struct pci_dev *pci_dev)
{
	struct net_device *net_dev = pci_get_drvdata(pci_dev);
	struct sis900_private * sis_priv = net_dev->priv;
	struct mii_phy *phy = NULL;

	while (sis_priv->first_mii) {
		phy = sis_priv->first_mii;
		sis_priv->first_mii = phy->next;
		kfree(phy);
	}

	pci_free_consistent(pci_dev, RX_TOTAL_SIZE, sis_priv->rx_ring,
		sis_priv->rx_ring_dma);
	pci_free_consistent(pci_dev, TX_TOTAL_SIZE, sis_priv->tx_ring,
		sis_priv->tx_ring_dma);
	unregister_netdev(net_dev);
	free_netdev(net_dev);
	pci_release_regions(pci_dev);
	pci_set_drvdata(pci_dev, NULL);
} ? end sis900_remove ?

#ifdef CONFIG_PM

static int sis900_suspend(struct pci_dev *pci_dev, u32 state)
{
	struct net_device *net_dev = pci_get_drvdata(pci_dev);
	long ioaddr = net_dev->base_addr;

	if(!netif_running(net_dev))
		return 0;

	netif_stop_queue(net_dev);
	netif_device_detach(net_dev);

	/* Stop the chip's Tx and Rx Status Machine */
	outl(RxDIS | TxDIS | inl(ioaddr + cr), ioaddr + cr);

	pci_set_power_state(pci_dev, PCI_D3hot);
	pci_save_state(pci_dev);

	return 0;
}

static int sis900_resume(struct pci_dev *pci_dev)
{
	struct net_device *net_dev = pci_get_drvdata(pci_dev);
	struct sis900_private *sis_priv = net_dev->priv;
	long ioaddr = net_dev->base_addr;

	if(!netif_running(net_dev))
		return 0;
	pci_restore_state(pci_dev);
	pci_set_power_state(pci_dev, PCI_D0);

	sis900_init_rxfilter(net_dev);

	sis900_init_tx_ring(net_dev);
	sis900_init_rx_ring(net_dev);

	set_rx_mode(net_dev);

	netif_device_attach(net_dev);
	netif_start_queue(net_dev);

	/* Workaround for EDB */
	sis900_set_mode(ioaddr, HW_SPEED_10_MBPS, FDX_CAPABLE_HALF_SELECTED);

	/* Enable all known interrupts by setting the interrupt mask. */
	outl((RxSOVR|RxORN|RxERR|RxOK|TxURN|TxERR|TxIDLE), ioaddr + imr);
	outl(RxENA | inl(ioaddr + cr), ioaddr + cr);
	outl(IE, ioaddr + ier);

	sis900_check_mode(net_dev, sis_priv->mii);

	return 0;
} ? end sis900_resume ?
#endif /* CONFIG_PM */

static struct pci_driver sis900_pci_driver = {
	.name		= SIS900_MODULE_NAME,
	.id_table	= sis900_pci_tbl,
	.probe		= sis900_probe,
	.remove		= __devexit_p(sis900_remove),
#ifdef CONFIG_PM
	.suspend	= sis900_suspend,
	.resume		= sis900_resume,
#endif /* CONFIG_PM */
};

static int __init sis900_init_module(void)
{
/* when a module, this is printed whether or not devices are found in probe */
```

```
#ifdef MODULE
	printk(version);
#endif

	return pci_module_init(&sis900_pci_driver);
}

static void __exit sis900_cleanup_module(void)
{
	pci_unregister_driver(&sis900_pci_driver);
}

module_init(sis900_init_module);
module_exit(sis900_cleanup_module);

```