```
/*
 * RDC R6040 Fast Ethernet MAC support
 *
 * Explanations added in red by Michael Opdenacker <michael@free-electrons.com>
 * See all our technical docs on http://free-electrons.com/docs/
 * Original file: drivers/net/r6040.c in Linux 2.6.27
 *
 * We advise you to read this file starting from the module init and exit
 * functions at the bottom, and progressively going up to lower level functions.
 *
 * Copyright (C) 2004 Sten Wang <sten.wang@rdc.com.tw>
 * Copyright (C) 2007
 *     Daniel Gimpelevich <daniel@gimpelevich.san-francisco.ca.us>
 *     Florian Fainelli <florian@openwrt.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the
 * Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
 * Boston, MA  02110-1301, USA.
*/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.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/etherdevice.h>
#include <linux/skbuff.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/mii.h>
#include <linux/ethtool.h>
#include <linux/crc32.h>
#include <linux/spinlock.h>
#include <linux/bitops.h>
#include <linux/io.h>
#include <linux/irq.h>
#include <linux/uaccess.h>

#include <asm/processor.h>

#define DRV_NAME      "r6040"
#define DRV_VERSION   "0.18"
#define DRV_RELDATE   "13Jul2008"

/* PHY CHIP Address */
#define PHY1_ADDR     1       /* For MAC1 */
#define PHY2_ADDR     2       /* For MAC2 */
#define PHY_MODE      0x3100  /* PHY CHIP Register 0 */
#define PHY_CAP               0x01E1  /* PHY CHIP Register 4 */

/* Time in jiffies before concluding the transmitter is hung. */
#define TX_TIMEOUT    (6000 * HZ / 1000)
```

```
/* RDC MAC I/O Size */
#define R6040_IO_SIZE 256

/* MAX RDC MAC */
#define MAX_MAC             2

/* MAC registers */
#define MCR0           0x00    /* Control register 0 */
#define MCR1           0x04    /* Control register 1 */
#define  MAC_RST       0x0001  /* Reset the MAC */
#define MBCR           0x08    /* Bus control */
#define MT_ICR         0x0C    /* TX interrupt control */
#define MR_ICR         0x10    /* RX interrupt control */
#define MTPR           0x14    /* TX poll command register */
#define MR_BSR         0x18    /* RX buffer size */
#define MR_DCR         0x1A    /* RX descriptor control */
#define MLSR           0x1C    /* Last status */
#define MMDIO          0x20    /* MDIO control register */
#define  MDIO_WRITE    0x4000  /* MDIO write */
#define  MDIO_READ     0x2000  /* MDIO read */
#define MMRD           0x24    /* MDIO read data register */
#define MMWD           0x28    /* MDIO write data register */
#define MTD_SA0                0x2C    /* TX descriptor start address 0 */
#define MTD_SA1                0x30    /* TX descriptor start address 1 */
#define MRD_SA0                0x34    /* RX descriptor start address 0 */
#define MRD_SA1                0x38    /* RX descriptor start address 1 */
#define MISR           0x3C    /* Status register */
#define MIER           0x40    /* INT enable register */
#define  MSK_INT       0x0000  /* Mask off interrupts */
#define  RX_FINISH     0x0001   /* RX finished */
#define  RX_NO_DESC    0x0002   /* No RX descriptor available */
#define  RX_FIFO_FULL  0x0004   /* RX FIFO full */
#define  RX_EARLY      0x0008   /* RX early */
#define  TX_FINISH     0x0010   /* TX finished */
#define  TX_EARLY      0x0080   /* TX early */
#define  EVENT_OVRFL   0x0100   /* Event counter overflow */
#define  LINK_CHANGED  0x0200   /* PHY link changed */
#define ME_CISR                0x44    /* Event counter INT status */
#define ME_CIER                0x48    /* Event counter INT enable  */
#define MR_CNT         0x50    /* Successfully received packet counter */
#define ME_CNT0                0x52    /* Event counter 0 */
#define ME_CNT1                0x54    /* Event counter 1 */
#define ME_CNT2                0x56    /* Event counter 2 */
#define ME_CNT3                0x58    /* Event counter 3 */
#define MT_CNT         0x5A    /* Successfully transmit packet counter */
#define ME_CNT4                0x5C    /* Event counter 4 */
#define MP_CNT         0x5E    /* Pause frame counter register */
#define MAR0           0x60    /* Hash table 0 */
#define MAR1           0x62    /* Hash table 1 */
#define MAR2           0x64    /* Hash table 2 */
#define MAR3           0x66    /* Hash table 3 */
#define MID_0L         0x68    /* Multicast address MID0 Low */
#define MID_0M         0x6A    /* Multicast address MID0 Medium */
#define MID_0H         0x6C    /* Multicast address MID0 High */
#define MID_1L         0x70    /* MID1 Low */
#define MID_1M         0x72    /* MID1 Medium */
#define MID_1H         0x74    /* MID1 High */
#define MID_2L         0x78    /* MID2 Low */
#define MID_2M         0x7A    /* MID2 Medium */
#define MID_2H         0x7C    /* MID2 High */
#define MID_3L         0x80    /* MID3 Low */
#define MID_3M         0x82    /* MID3 Medium */
#define MID_3H         0x84    /* MID3 High */
#define PHY_CC         0x88    /* PHY status change configuration register */
#define PHY_ST         0x8A    /* PHY status register */
#define MAC_SM         0xAC    /* MAC status machine */
#define MAC_ID         0xBE    /* Identifier register */

#define TX_DCNT                0x80    /* TX descriptor count */
#define RX_DCNT                0x80    /* RX descriptor count */
```

```
#define MAX_BUF_SIZE  0x600
#define RX_DESC_SIZE  (RX_DCNT * sizeof(struct r6040_descriptor))
#define TX_DESC_SIZE  (TX_DCNT * sizeof(struct r6040_descriptor))
#define MBCR_DEFAULT  0x012A  /* MAC Bus Control Register */
#define MCAST_MAX     4       /* Max number multicast addresses to filter */

/* Descriptor status */
#define DSC_OWNER_MAC 0x8000  /* MAC is the owner of this descriptor */
#define DSC_RX_OK     0x4000  /* RX was successful */
#define DSC_RX_ERR    0x0800  /* RX PHY error */
#define DSC_RX_ERR_DRI0x0400  /* RX dribble packet */
#define DSC_RX_ERR_BUF0x0200  /* RX length exceeds buffer size */
#define DSC_RX_ERR_LONG       0x0100  /* RX length > maximum packet length */
#define DSC_RX_ERR_RUNT       0x0080  /* RX packet length < 64 byte */
#define DSC_RX_ERR_CRC0x0040  /* RX CRC error */
#define DSC_RX_BCAST  0x0020  /* RX broadcast (no error) */
#define DSC_RX_MCAST  0x0010  /* RX multicast (no error) */
#define DSC_RX_MCH_HIT0x0008  /* RX multicast hit in hash table (no error) */
#define DSC_RX_MIDH_HIT       0x0004  /* RX MID table hit (no error) */
#define DSC_RX_IDX_MID_MASK 3 /* RX mask for the index of matched MIDx */

/* PHY settings */
#define ICPLUS_PHY_ID 0x0243

MODULE_AUTHOR("Sten Wang <sten.wang@rdc.com.tw>,"
       "Daniel Gimpelevich <daniel@gimpelevich.san-francisco.ca.us>,"
       "Florian Fainelli <florian@openwrt.org>");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("RDC R6040 NAPI PCI FastEthernet driver");

/* RX and TX interrupts that we handle */
#define RX_INTS                       (RX_FIFO_FULL | RX_NO_DESC | RX_FINISH)
#define TX_INTS                       (TX_FINISH)
#define INT_MASK              (RX_INTS | TX_INTS)

struct r6040_descriptor {
       u16    status, len;          /* 0-3 */
       __le32 buf;                  /* 4-7 */
       __le32 ndesc;                /* 8-B */
       u32    rev1;                 /* C-F */
       char   *vbufp;               /* 10-13 */
       struct r6040_descriptor *vndescp;    /* 14-17 */
       struct sk_buff *skb_ptr;     /* 18-1B */
       u32    rev2;                 /* 1C-1F */
} __attribute__((aligned(32)));

struct r6040_private {
       spinlock_t lock;             /* driver lock */
       struct timer_list timer;
       struct pci_dev *pdev;
       struct r6040_descriptor *rx_insert_ptr;
       struct r6040_descriptor *rx_remove_ptr;
       struct r6040_descriptor *tx_insert_ptr;
       struct r6040_descriptor *tx_remove_ptr;
       struct r6040_descriptor *rx_ring;
       struct r6040_descriptor *tx_ring;
       dma_addr_t rx_ring_dma;
       dma_addr_t tx_ring_dma;
       u16    tx_free_desc, phy_addr, phy_mode;
       u16    mcr0, mcr1;
       u16    switch_sig;
       struct net_device *dev;
       struct mii_if_info mii_if;
       struct napi_struct napi;
       void __iomem *base;
};

static char version[] __devinitdata = KERN_INFO DRV_NAME
       ": RDC R6040 NAPI net driver,"
       "version "DRV_VERSION " (" DRV_RELDATE ")\n";
```

```
static int phy_table[] = { PHY1_ADDR, PHY2_ADDR };

/* Read a word data from PHY Chip */
static int r6040_phy_read(void __iomem *ioaddr, int phy_addr, int reg)
{
	int limit = 2048;
	u16 cmd;

	iowrite16(MDIO_READ + reg + (phy_addr << 8), ioaddr + MMDIO);
	/* Wait for the read bit to be cleared */
	while (limit--) {
		cmd = ioread16(ioaddr + MMDIO);
		if (cmd & MDIO_READ)
			break;
	}

	return ioread16(ioaddr + MMRD);
}

/* Write a word data from PHY Chip */
static void r6040_phy_write(void __iomem *ioaddr, int phy_addr, int reg, u16 val)
{
	int limit = 2048;
	u16 cmd;

	iowrite16(val, ioaddr + MMWD);
	/* Write the command to the MDIO bus */
	iowrite16(MDIO_WRITE + reg + (phy_addr << 8), ioaddr + MMDIO);
	/* Wait for the write bit to be cleared */
	while (limit--) {
		cmd = ioread16(ioaddr + MMDIO);
		if (cmd & MDIO_WRITE)
			break;
	}
}

static int r6040_mdio_read(struct net_device *dev, int mii_id, int reg)
{
	struct r6040_private *lp = netdev_priv(dev);
	void __iomem *ioaddr = lp->base;

	return (r6040_phy_read(ioaddr, lp->phy_addr, reg));
}

static void r6040_mdio_write(struct net_device *dev, int mii_id, int reg, int val)
{
	struct r6040_private *lp = netdev_priv(dev);
	void __iomem *ioaddr = lp->base;

	r6040_phy_write(ioaddr, lp->phy_addr, reg, val);
}

static void r6040_free_txbufs(struct net_device *dev)
{
	struct r6040_private *lp = netdev_priv(dev);
	int i;

	for (i = 0; i < TX_DCNT; i++) {
		if (lp->tx_insert_ptr->skb_ptr) {
			pci_unmap_single(lp->pdev,
				le32_to_cpu(lp->tx_insert_ptr->buf),
				MAX_BUF_SIZE, PCI_DMA_TODEVICE);
			dev_kfree_skb(lp->tx_insert_ptr->skb_ptr);
			lp->rx_insert_ptr->skb_ptr = NULL;
		}
		lp->tx_insert_ptr = lp->tx_insert_ptr->vndescp;
	}
}
```

```
static void r6040_free_rxbufs(struct net_device *dev)
{
	struct r6040_private *lp = netdev_priv(dev);
	int i;

	for (i = 0; i < RX_DCNT; i++) {
		if (lp->rx_insert_ptr->skb_ptr) {
			pci_unmap_single(lp->pdev,
				le32_to_cpu(lp->rx_insert_ptr->buf),
				MAX_BUF_SIZE, PCI_DMA_FROMDEVICE);
			dev_kfree_skb(lp->rx_insert_ptr->skb_ptr);
			lp->rx_insert_ptr->skb_ptr = NULL;
		}
		lp->rx_insert_ptr = lp->rx_insert_ptr->vndescp;
	}
}

static void r6040_init_ring_desc(struct r6040_descriptor *desc_ring,
				 dma_addr_t desc_dma, int size)
{
	struct r6040_descriptor *desc = desc_ring;
	dma_addr_t mapping = desc_dma;

	while (size-- > 0) {
		mapping += sizeof(*desc);
		desc->ndesc = cpu_to_le32(mapping);
		desc->vndescp = desc + 1;
		desc++;
	}
	desc--;
	desc->ndesc = cpu_to_le32(desc_dma);
	desc->vndescp = desc_ring;
}

static void r6040_init_txbufs(struct net_device *dev)
{
	struct r6040_private *lp = netdev_priv(dev);

	lp->tx_free_desc = TX_DCNT;

	lp->tx_remove_ptr = lp->tx_insert_ptr = lp->tx_ring;
	r6040_init_ring_desc(lp->tx_ring, lp->tx_ring_dma, TX_DCNT);
}

static int r6040_alloc_rxbufs(struct net_device *dev)
{
	struct r6040_private *lp = netdev_priv(dev);
	struct r6040_descriptor *desc;
	struct sk_buff *skb;
	int rc;

	lp->rx_remove_ptr = lp->rx_insert_ptr = lp->rx_ring;
	r6040_init_ring_desc(lp->rx_ring, lp->rx_ring_dma, RX_DCNT);

	/* Allocate skbs for the rx descriptors */
	desc = lp->rx_ring;
	do {
		skb = netdev_alloc_skb(dev, MAX_BUF_SIZE);
		if (!skb) {
			printk(KERN_ERR "%s: failed to alloc skb for rx\n", dev->name);
			rc = -ENOMEM;
			goto err_exit;
		}
		desc->skb_ptr = skb;

		// Give ownership of the buffers to the device (for DMA)
		desc->buf = cpu_to_le32(pci_map_single(lp->pdev,
					desc->skb_ptr->data,
					MAX_BUF_SIZE, PCI_DMA_FROMDEVICE));
		desc->status = DSC_OWNER_MAC;
```

```
                desc = desc->vndescp;
        } while (desc != lp->rx_ring);

        return 0;

err_exit:
        /* Deallocate all previously allocated skbs */
        r6040_free_rxbufs(dev);
        return rc;
}

static void r6040_init_mac_regs(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        int limit = 2048;
        u16 cmd;

        /* Mask Off Interrupt */
        iowrite16(MSK_INT, ioaddr + MIER);

        /* Reset RDC MAC */
        iowrite16(MAC_RST, ioaddr + MCR1);
        while (limit--) {
                cmd = ioread16(ioaddr + MCR1);
                if (cmd & 0x1)
                        break;
        }
        /* Reset internal state machine */
        iowrite16(2, ioaddr + MAC_SM);
        iowrite16(0, ioaddr + MAC_SM);
        udelay(5000);

        /* MAC Bus Control Register */
        iowrite16(MBCR_DEFAULT, ioaddr + MBCR);

        /* Buffer Size Register */
        iowrite16(MAX_BUF_SIZE, ioaddr + MR_BSR);

        /* Write TX ring start address */
        iowrite16(lp->tx_ring_dma, ioaddr + MTD_SA0);
        iowrite16(lp->tx_ring_dma >> 16, ioaddr + MTD_SA1);

        /* Write RX ring start address */
        iowrite16(lp->rx_ring_dma, ioaddr + MRD_SA0);
        iowrite16(lp->rx_ring_dma >> 16, ioaddr + MRD_SA1);

        /* Set interrupt waiting time and packet numbers */
        iowrite16(0, ioaddr + MT_ICR);
        iowrite16(0, ioaddr + MR_ICR);

        /* Enable interrupts */
        iowrite16(INT_MASK, ioaddr + MIER);

        /* Enable TX and RX */
        iowrite16(lp->mcr0 | 0x0002, ioaddr);

        /* Let TX poll the descriptors
         * we may got called by r6040_tx_timeout which has left
         * some unsent tx buffers */
        iowrite16(0x01, ioaddr + MTPR);
}

static void r6040_tx_timeout(struct net_device *dev)
{
        struct r6040_private *priv = netdev_priv(dev);
        void __iomem *ioaddr = priv->base;

        printk(KERN_WARNING "%s: transmit timed out, int enable %4.4x "
                "status %4.4x, PHY status %4.4x\n",
```

```
                dev->name, ioread16(ioaddr + MIER),
                ioread16(ioaddr + MISR),
                r6040_mdio_read(dev, priv->mii_if.phy_id, MII_BMSR));

        dev->stats.tx_errors++;

        /* Reset MAC and re-init all registers */
        r6040_init_mac_regs(dev);
}

static struct net_device_stats *r6040_get_stats(struct net_device *dev)
{
        struct r6040_private *priv = netdev_priv(dev);
        void __iomem *ioaddr = priv->base;
        unsigned long flags;

        spin_lock_irqsave(&priv->lock, flags);
        dev->stats.rx_crc_errors += ioread8(ioaddr + ME_CNT1);
        dev->stats.multicast += ioread8(ioaddr + ME_CNT0);
        spin_unlock_irqrestore(&priv->lock, flags);

        return &dev->stats;
}

/* Stop RDC MAC and Free the allocated resource */
static void r6040_down(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        struct pci_dev *pdev = lp->pdev;
        int limit = 2048;
        u16 *adrp;
        u16 cmd;

        /* Stop MAC */
        iowrite16(MSK_INT, ioaddr + MIER);    /* Mask Off Interrupt */
        iowrite16(MAC_RST, ioaddr + MCR1);    /* Reset RDC MAC */
        while (limit--) {
                cmd = ioread16(ioaddr + MCR1);
                if (cmd & 0x1)
                        break;
        }

        /* Restore MAC Address to MIDx */
        adrp = (u16 *) dev->dev_addr;
        iowrite16(adrp[0], ioaddr + MID_0L);
        iowrite16(adrp[1], ioaddr + MID_0M);
        iowrite16(adrp[2], ioaddr + MID_0H);
        free_irq(dev->irq, dev);

        /* Free RX buffer */
        r6040_free_rxbufs(dev);

        /* Free TX buffer */
        r6040_free_txbufs(dev);

        /* Free Descriptor memory */
        pci_free_consistent(pdev, RX_DESC_SIZE, lp->rx_ring, lp->rx_ring_dma);
        pci_free_consistent(pdev, TX_DESC_SIZE, lp->tx_ring, lp->tx_ring_dma);
}

// Called when the ethx device is closed
static int r6040_close(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);

        /* deleted timer */
        del_timer_sync(&lp->timer);

        spin_lock_irq(&lp->lock);
```

```
        // Wait for any ongoing work to complete, stop napi
        napi_disable(&lp->napi);
        // Stop accepting work from the protocol stack.
        netif_stop_queue(dev);
        // Disable the hardware and release resources
        r6040_down(dev);
        spin_unlock_irq(&lp->lock);

        return 0;
}

/* Status of PHY CHIP */
static int r6040_phy_mode_chk(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        int phy_dat;

        /* PHY Link Status Check */
        phy_dat = r6040_phy_read(ioaddr, lp->phy_addr, 1);
        if (!(phy_dat & 0x4))
                phy_dat = 0x8000;     /* Link Failed, full duplex */

        /* PHY Chip Auto-Negotiation Status */
        phy_dat = r6040_phy_read(ioaddr, lp->phy_addr, 1);
        if (phy_dat & 0x0020) {
                /* Auto Negotiation Mode */
                phy_dat = r6040_phy_read(ioaddr, lp->phy_addr, 5);
                phy_dat &= r6040_phy_read(ioaddr, lp->phy_addr, 4);
                if (phy_dat & 0x140)
                        /* Force full duplex */
                        phy_dat = 0x8000;
                else
                        phy_dat = 0;
        } else {
                /* Force Mode */
                phy_dat = r6040_phy_read(ioaddr, lp->phy_addr, 0);
                if (phy_dat & 0x100)
                        phy_dat = 0x8000;
                else
                        phy_dat = 0x0000;
        }

        return phy_dat;
};

static void r6040_set_carrier(struct mii_if_info *mii)
{
        if (r6040_phy_mode_chk(mii->dev)) {
                /* autoneg is off: Link is always assumed to be up */
                if (!netif_carrier_ok(mii->dev))
                        netif_carrier_on(mii->dev);
        } else
                r6040_phy_mode_chk(mii->dev);
}

// ioctl to change mii settings

static int r6040_ioctl(struct net_device *dev, struct ifreq *rq, int cmd)
{
        struct r6040_private *lp = netdev_priv(dev);
        struct mii_ioctl_data *data = if_mii(rq);
        int rc;

        if (!netif_running(dev))
                return -EINVAL;
        spin_lock_irq(&lp->lock);
        rc = generic_mii_ioctl(&lp->mii_if, data, cmd, NULL);
        spin_unlock_irq(&lp->lock);
        r6040_set_carrier(&lp->mii_if);
```

```
        return rc;
}

// Function reading incoming packets and forwarding
// them to the protocol layer

static int r6040_rx(struct net_device *dev, int limit)
{
        struct r6040_private *priv = netdev_priv(dev);
        struct r6040_descriptor *descptr = priv->rx_remove_ptr;
        struct sk_buff *skb_ptr, *new_skb;
        int count = 0;
        u16 err;

        /* Limit not reached and the descriptor belongs to the CPU */
        // Iterate over rx descriptors
        while (count < limit && !(descptr->status & DSC_OWNER_MAC)) {
                /* Read the descriptor status */
                err = descptr->status;
                /* Global error status set */
                if (err & DSC_RX_ERR) {
                        /* RX dribble */
                        if (err & DSC_RX_ERR_DRI)
                                dev->stats.rx_frame_errors++;
                        /* Buffer length exceeded */
                        if (err & DSC_RX_ERR_BUF)
                                dev->stats.rx_length_errors++;
                        /* Packet too long */
                        if (err & DSC_RX_ERR_LONG)
                                dev->stats.rx_length_errors++;
                        /* Packet < 64 bytes */
                        if (err & DSC_RX_ERR_RUNT)
                                dev->stats.rx_length_errors++;
                        /* CRC error */
                        if (err & DSC_RX_ERR_CRC) {
                                spin_lock(&priv->lock);
                                dev->stats.rx_crc_errors++;
                                spin_unlock(&priv->lock);
                        }
                        goto next_descr;
                }

                /* Packet successfully received */

                // We're going to pass the descriptor's skb
                // to the protocol layer. Allocate a new skb
                // that will then be attached to the descriptor.

                new_skb = netdev_alloc_skb(dev, MAX_BUF_SIZE);
                if (!new_skb) {
                        dev->stats.rx_dropped++;
                        goto next_descr;
                }

                // incoming skb that will be passed to the protocol layer.
                skb_ptr = descptr->skb_ptr;
                skb_ptr->dev = priv->dev;

                /* Do not count the CRC */
                // Add data to the skb, by moving the skb's tail pointer
                skb_put(skb_ptr, descptr->len - 4);
                // Return ownership of the descriptor buffer to the CPU (after DMA),
                // needed to access the packet data
                pci_unmap_single(priv->pdev, le32_to_cpu(descptr->buf),
                                 MAX_BUF_SIZE, PCI_DMA_FROMDEVICE);
                skb_ptr->protocol = eth_type_trans(skb_ptr, priv->dev);

                /* Send to upper layer */
                netif_receive_skb(skb_ptr);
```

```
                // Update device stats
                dev->last_rx = jiffies;       // Current time (in timer ticks)
                dev->stats.rx_packets++;
                dev->stats.rx_bytes += descptr->len - 4;

                /* put new skb into descriptor */
                // to make this descriptor available again for new incoming packets.
                descptr->skb_ptr = new_skb;

                // Given ownership of the new descriptor buffer to the device (for DMA)
                descptr->buf = cpu_to_le32(pci_map_single(priv->pdev,
                                        descptr->skb_ptr->data,
                                MAX_BUF_SIZE, PCI_DMA_FROMDEVICE));

next_descr:
                /* put the descriptor back to the MAC */
                descptr->status = DSC_OWNER_MAC;
                descptr = descptr->vndescp;
                count++;
        }
        priv->rx_remove_ptr = descptr;

        return count;
}

// Called at the completion of a transmit operation.
// Need to check for transfer errors

static void r6040_tx(struct net_device *dev)
{
        struct r6040_private *priv = netdev_priv(dev);
        struct r6040_descriptor *descptr;
        void __iomem *ioaddr = priv->base;
        struct sk_buff *skb_ptr;
        u16 err;

        spin_lock(&priv->lock);
        descptr = priv->tx_remove_ptr;

        while (priv->tx_free_desc < TX_DCNT) {
                /* Check for errors */
                err = ioread16(ioaddr + MLSR);

                if (err & 0x0200)
                        dev->stats.rx_fifo_errors++;
                if (err & (0x2000 | 0x4000))
                        dev->stats.tx_carrier_errors++;

                if (descptr->status & DSC_OWNER_MAC)
                        break; /* Not complete */
                skb_ptr = descptr->skb_ptr;

                // Unmap the DMA streaming buffer. DMA done.
                pci_unmap_single(priv->pdev, le32_to_cpu(descptr->buf),
                        skb_ptr->len, PCI_DMA_TODEVICE);
                /* Free buffer */
                // Release the tx descriptor and make it available for future transfers
                dev_kfree_skb_irq(skb_ptr);
                descptr->skb_ptr = NULL;
                /* To next descriptor */
                descptr = descptr->vndescp;
                priv->tx_free_desc++;
        }
        priv->tx_remove_ptr = descptr;

        if (priv->tx_free_desc)
                // Accept more tx requests if descriptors available
                netif_wake_queue(dev);
        spin_unlock(&priv->lock);
}
```

```
// Function reading received packets in polled mode.
static int r6040_poll(struct napi_struct *napi, int budget)
{
        struct r6040_private *priv =
                container_of(napi, struct r6040_private, napi);
        struct net_device *dev = priv->dev;
        void __iomem *ioaddr = priv->base;
        int work_done;

        // Receive / consume packets
        work_done = r6040_rx(dev, budget);

        if (work_done < budget) {
                // All packets consumed. Switch back to interrupt mode.
                netif_rx_complete(dev, napi);
                /* Enable RX interrupt */
                iowrite16(ioread16(ioaddr + MIER) | RX_INTS, ioaddr + MIER);
        }
        return work_done;
}

/* The RDC interrupt handler. */
static irqreturn_t r6040_interrupt(int irq, void *dev_id)
{
        struct net_device *dev = dev_id;
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        u16 status;

        /* Mask off RDC MAC interrupt */
        iowrite16(MSK_INT, ioaddr + MIER);
        /* Read MISR status and clear */
        status = ioread16(ioaddr + MISR);

        // Interrupt not for our device, ignored
        if (status == 0x0000 || status == 0xffff)
                return IRQ_NONE;

        /* RX interrupt request */
        // If this was a rx event
        if (status & RX_INTS) {
                if (status & RX_NO_DESC) {
                        /* RX descriptor unavailable */
                        dev->stats.rx_dropped++;
                        dev->stats.rx_missed_errors++;
                }
                if (status & RX_FIFO_FULL)
                        dev->stats.rx_fifo_errors++;

                /* Mask off RX interrupt */
                // We disable the RX interrupt to switch to polled mode.
                iowrite16(ioread16(ioaddr + MIER) & ~RX_INTS, ioaddr + MIER);
                // We add the device to a polled list
                // This is a softirq to offload the work of posting
                // received packets to the protocol stack.
                netif_rx_schedule(dev, &lp->napi);
                // The r6040_poll function will be called. It was
                // attached to lp->napi by the netif_napi_add() call
                // in the probe() function.
        }

        /* TX interrupt request */
        // Much simpler! Signals the completion
        // of a transmit operation.
        if (status & TX_INTS)
                r6040_tx(dev);

        return IRQ_HANDLED;
}
```

```
#ifdef CONFIG_NET_POLL_CONTROLLER
static void r6040_poll_controller(struct net_device *dev)
{
	disable_irq(dev->irq);
	r6040_interrupt(dev->irq, dev);
	enable_irq(dev->irq);
}
#endif

/* Init RDC MAC */
// Called in the open function - Gets the device ready to tx and rx

static int r6040_up(struct net_device *dev)
{
	struct r6040_private *lp = netdev_priv(dev);
	void __iomem *ioaddr = lp->base;
	int ret;

	/* Initialise and alloc RX/TX buffers */
	r6040_init_txbufs(dev);
	ret = r6040_alloc_rxbufs(dev);
	if (ret)
		return ret;

	/* Read the PHY ID */
	lp->switch_sig = r6040_phy_read(ioaddr, 0, 2);

	if (lp->switch_sig  == ICPLUS_PHY_ID) {
		r6040_phy_write(ioaddr, 29, 31, 0x175C); /* Enable registers */
		lp->phy_mode = 0x8000;
	} else {
		/* PHY Mode Check */
		r6040_phy_write(ioaddr, lp->phy_addr, 4, PHY_CAP);
		r6040_phy_write(ioaddr, lp->phy_addr, 0, PHY_MODE);

		if (PHY_MODE == 0x3100)
			lp->phy_mode = r6040_phy_mode_chk(dev);
		else
			lp->phy_mode = (PHY_MODE & 0x0100) ? 0x8000:0x0;
	}

	/* Set duplex mode */
	lp->mcr0 |= lp->phy_mode;

	/* improve performance (by RDC guys) */
	r6040_phy_write(ioaddr, 30, 17, (r6040_phy_read(ioaddr, 30, 17) | 0x4000));
	r6040_phy_write(ioaddr, 30, 17, ~((~r6040_phy_read(ioaddr, 30, 17)) | 0x2000));
	r6040_phy_write(ioaddr, 0, 19, 0x0000);
	r6040_phy_write(ioaddr, 0, 30, 0x01F0);

	/* Initialize all MAC registers */
	r6040_init_mac_regs(dev);

	return 0;
}

/*
  A periodic timer routine
	Polling PHY Chip Link Status
*/
static void r6040_timer(unsigned long data)
{
	struct net_device *dev = (struct net_device *)data;
	struct r6040_private *lp = netdev_priv(dev);
	void __iomem *ioaddr = lp->base;
	u16 phy_mode;

	/* Polling PHY Chip Status */
	if (PHY_MODE == 0x3100)
```

```
                phy_mode = r6040_phy_mode_chk(dev);
        else
                phy_mode = (PHY_MODE & 0x0100) ? 0x8000:0x0;

        if (phy_mode != lp->phy_mode) {
                lp->phy_mode = phy_mode;
                lp->mcr0 = (lp->mcr0 & 0x7fff) | phy_mode;
                iowrite16(lp->mcr0, ioaddr);
                printk(KERN_INFO "Link Change %x \n", ioread16(ioaddr));
        }

        /* Timer active again */
        mod_timer(&lp->timer, round_jiffies(jiffies + HZ));
}

/* Read/set MAC address routines */
static void r6040_mac_address(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        u16 *adrp;

        /* MAC operation register */
        iowrite16(0x01, ioaddr + MCR1); /* Reset MAC */
        iowrite16(2, ioaddr + MAC_SM); /* Reset internal state machine */
        iowrite16(0, ioaddr + MAC_SM);
        udelay(5000);

        /* Restore MAC Address */
        adrp = (u16 *) dev->dev_addr;
        iowrite16(adrp[0], ioaddr + MID_0L);
        iowrite16(adrp[1], ioaddr + MID_0M);
        iowrite16(adrp[2], ioaddr + MID_0H);
}

// Open function - Called with ethx is opened

static int r6040_open(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        int ret;

        /* Request IRQ and Register interrupt handler */
        // Register the IRQ handler at net device open time.
        // Not needed earlier!

        ret = request_irq(dev->irq, &r6040_interrupt,
                IRQF_SHARED, dev->name, dev);
        if (ret)
                return ret;

        /* Set MAC address */
        r6040_mac_address(dev);

        /* Allocate Descriptor memory */
        // rx descriptor ring
        // allocated bus address received in &lp->rx_ring_dma
        lp->rx_ring =
                pci_alloc_consistent(lp->pdev, RX_DESC_SIZE, &lp->rx_ring_dma);
        if (!lp->rx_ring)
                return -ENOMEM;

        // tx descriptor ring
        lp->tx_ring =
                pci_alloc_consistent(lp->pdev, TX_DESC_SIZE, &lp->tx_ring_dma);
        if (!lp->tx_ring) {
                // Failed. Free the rx ring before exiting
                pci_free_consistent(lp->pdev, RX_DESC_SIZE, lp->rx_ring,
                                    lp->rx_ring_dma);
                return -ENOMEM;
```

```
        }

        // Initialize MAC, alloc and initialize tx and rx skbs
        ret = r6040_up(dev);
        if (ret) {
                pci_free_consistent(lp->pdev, TX_DESC_SIZE, lp->tx_ring,
                                                        lp->tx_ring_dma);
                pci_free_consistent(lp->pdev, RX_DESC_SIZE, lp->rx_ring,
                                                        lp->rx_ring_dma);
                return ret;
        }

        napi_enable(&lp->napi); // Enable NAPI mode (polling mode when busy, intr mode otherwise)
        netif_start_queue(dev); // Start accepting to rx / tx packets

        /* set and active a timer process */
        // Periodic time to check the status of the phy link
        setup_timer(&lp->timer, r6040_timer, (unsigned long) dev);
        if (lp->switch_sig != ICPLUS_PHY_ID)
                mod_timer(&lp->timer, jiffies + HZ);
        return 0;
}

// Transmit function - Called by the protocol stack.
// Declared by attaching to the netdev structure

static int r6040_start_xmit(struct sk_buff *skb, struct net_device *dev)

        // skb: socket buffer - data received from the protocol stack
        void __iomem *ioaddr = lp->base;
        unsigned long flags;
{
        struct r6040_private *lp = netdev_priv(dev);
        struct r6040_descriptor *descptr;
        int ret = NETDEV_TX_OK;

        /* Critical Section */
        // Prevent concurrent access to netdev private data
        spin_lock_irqsave(&lp->lock, flags);

        /* TX resource check */
        if (!lp->tx_free_desc) {
                // No free tx descriptor left
                spin_unlock_irqrestore(&lp->lock, flags);
                // Stop accepting packets
                netif_stop_queue(dev);
                printk(KERN_ERR DRV_NAME ": no tx descriptor\n");
                ret = NETDEV_TX_BUSY;
                return ret;
        }

        /* Statistic Counter */

        // Increment tx stats for ifconfig
        dev->stats.tx_packets++;
        dev->stats.tx_bytes += skb->len;

        /* Set TX descriptor & Transmit it */
        lp->tx_free_desc--;
        descptr = lp->tx_insert_ptr;
        if (skb->len < MISR)
                descptr->len = MISR;
        else
                descptr->len = skb->len;

        descptr->skb_ptr = skb;

        // Map a tx descriptor ready for DMA
        descptr->buf = cpu_to_le32(pci_map_single(lp->pdev,
                skb->data, skb->len, PCI_DMA_TODEVICE));
```

```
        descptr->status = DSC_OWNER_MAC;

        /* Trigger the MAC to check the TX descriptor */
        // This means, start the DMA from skb to device
        iowrite16(0x01, ioaddr + MTPR);
        lp->tx_insert_ptr = descptr->vndescp;

        /* If no tx resource, stop */
        // Stop accepting to transmit packets

        if (!lp->tx_free_desc)
                netif_stop_queue(dev);

        dev->trans_start = jiffies;
        spin_unlock_irqrestore(&lp->lock, flags);
        return ret;
}

// Multicast support - Not detailed here.

static void r6040_multicast_list(struct net_device *dev)
{
        struct r6040_private *lp = netdev_priv(dev);
        void __iomem *ioaddr = lp->base;
        u16 *adrp;
        u16 reg;
        unsigned long flags;
        struct dev_mc_list *dmi = dev->mc_list;
        int i;

        /* MAC Address */
        adrp = (u16 *)dev->dev_addr;
        iowrite16(adrp[0], ioaddr + MID_0L);
        iowrite16(adrp[1], ioaddr + MID_0M);
        iowrite16(adrp[2], ioaddr + MID_0H);


        /* Promiscous Mode */
        spin_lock_irqsave(&lp->lock, flags);

        /* Clear AMCP & PROM bits */
        reg = ioread16(ioaddr) & ~0x0120;
        if (dev->flags & IFF_PROMISC) {
                reg |= 0x0020;
                lp->mcr0 |= 0x0020;
        }
        /* Too many multicast addresses
         * accept all traffic */
        else if ((dev->mc_count > MCAST_MAX)
                || (dev->flags & IFF_ALLMULTI))
                reg |= 0x0020;

        iowrite16(reg, ioaddr);
        spin_unlock_irqrestore(&lp->lock, flags);

        /* Build the hash table */
        if (dev->mc_count > MCAST_MAX) {
                u16 hash_table[4];
                u32 crc;

                for (i = 0; i < 4; i++)
                        hash_table[i] = 0;

                for (i = 0; i < dev->mc_count; i++) {
                        char *addrs = dmi->dmi_addr;

                        dmi = dmi->next;

                        if (!(*addrs & 1))
                                continue;
```

```
                        crc = ether_crc_le(6, addrs);
                        crc >>= 26;
                        hash_table[crc >> 4] |= 1 << (15 - (crc & 0xf));
                }
                /* Write the index of the hash table */
                for (i = 0; i < 4; i++)
                        iowrite16(hash_table[i] << 14, ioaddr + MCR1);
                /* Fill the MAC hash tables with their values */
                iowrite16(hash_table[0], ioaddr + MAR0);
                iowrite16(hash_table[1], ioaddr + MAR1);
                iowrite16(hash_table[2], ioaddr + MAR2);
                iowrite16(hash_table[3], ioaddr + MAR3);
        }
        /* Multicast Address 1~4 case */
        for (i = 0, dmi; (i < dev->mc_count) && (i < MCAST_MAX); i++) {
                adrp = (u16 *)dmi->dmi_addr;
                iowrite16(adrp[0], ioaddr + MID_1L + 8*i);
                iowrite16(adrp[1], ioaddr + MID_1M + 8*i);
                iowrite16(adrp[2], ioaddr + MID_1H + 8*i);
                dmi = dmi->next;
        }
        for (i = dev->mc_count; i < MCAST_MAX; i++) {
                iowrite16(0xffff, ioaddr + MID_0L + 8*i);
                iowrite16(0xffff, ioaddr + MID_0M + 8*i);
                iowrite16(0xffff, ioaddr + MID_0H + 8*i);
        }
}

// Ops for configuring ethx from user space with ethtool

static void netdev_get_drvinfo(struct net_device *dev,
                        struct ethtool_drvinfo *info)
{
        struct r6040_private *rp = netdev_priv(dev);

        strcpy(info->driver, DRV_NAME);
        strcpy(info->version, DRV_VERSION);
        strcpy(info->bus_info, pci_name(rp->pdev));
}

static int netdev_get_settings(struct net_device *dev, struct ethtool_cmd *cmd)
{
        struct r6040_private *rp = netdev_priv(dev);
        int rc;

        spin_lock_irq(&rp->lock);
        rc = mii_ethtool_gset(&rp->mii_if, cmd);
        spin_unlock_irq(&rp->lock);

        return rc;
}

static int netdev_set_settings(struct net_device *dev, struct ethtool_cmd *cmd)
{
        struct r6040_private *rp = netdev_priv(dev);
        int rc;

        spin_lock_irq(&rp->lock);
        rc = mii_ethtool_sset(&rp->mii_if, cmd);
        spin_unlock_irq(&rp->lock);
        r6040_set_carrier(&rp->mii_if);

        return rc;
}

static u32 netdev_get_link(struct net_device *dev)
{
        struct r6040_private *rp = netdev_priv(dev);
```

```
        return mii_link_ok(&rp->mii_if);
}

// Ops for configuring ethx from user space with ethtool

static struct ethtool_ops netdev_ethtool_ops = {
        .get_drvinfo            = netdev_get_drvinfo,
        .get_settings           = netdev_get_settings,
        .set_settings           = netdev_set_settings,
        .get_link               = netdev_get_link,
};

static int __devinit r6040_init_one(struct pci_dev *pdev,
                                    const struct pci_device_id *ent)
{

        // Device probe function - Called when the bus finds this device
        struct net_device *dev; // Structure representing the net device (ethx)
        struct r6040_private *lp;
        void __iomem *ioaddr;
        int err, io_size = R6040_IO_SIZE;
        static int card_idx = -1;
        int bar = 0;            // Base address register (on the PCI bus)
        long pioaddr;
        u16 *adrp;

        printk(KERN_INFO "%s\n", version);

        // Enable the PCI device. This makes device configuration accessible.
        // A IRQ line is also assigned then (if not done yet by the BIOS)

        err = pci_enable_device(pdev);
        if (err)
                goto err_out;

        /* this should always be supported */

        // Ask for a 32-bit DMA range

        err = pci_set_dma_mask(pdev, DMA_32BIT_MASK);
        if (err) {
                printk(KERN_ERR DRV_NAME "32-bit PCI DMA addresses"
                                "not supported by the card\n");
                goto err_out;
        }

        // Needed to use DMA consistent buffers that can go above 4GB addresses

        err = pci_set_consistent_dma_mask(pdev, DMA_32BIT_MASK);
        if (err) {
                printk(KERN_ERR DRV_NAME "32-bit PCI DMA addresses"
                                "not supported by the card\n");
                goto err_out;
        }

        /* IO Size check */
        // Check the size of PCI device I/O mapped area described by BAR0
        if (pci_resource_len(pdev, 0) < io_size) {
                printk(KERN_ERR DRV_NAME "Insufficient PCI resources, aborting\n");
                err = -EIO;
                goto err_out;
        }

        pioaddr = pci_resource_start(pdev, 0);      /* IO map base address */

        // Enable the device to become a master on the bus (to do DMA)
        pci_set_master(pdev);

        // Allocate a net device (Ethernet type)
```

```
dev = alloc_etherdev(sizeof(struct r6040_private));
if (!dev) {
        printk(KERN_ERR DRV_NAME "Failed to allocate etherdev\n");
        err = -ENOMEM;
        goto err_out;
}

// Makes pdev->dev become the parent of net (net->net.parent=&pdev->dev)
SET_NETDEV_DEV(dev, &pdev->dev);

// Get network device private data
lp = netdev_priv(dev);

// Reserve all PCI I/O port regions
err = pci_request_regions(pdev, DRV_NAME);

if (err) {
        printk(KERN_ERR DRV_NAME ": Failed to request PCI regions\n");
        goto err_out_free_dev;
}

// Map the physical addresses described by BAR0
// This gives a virtual address that the kernel can
// access (PCI wrapper around ioremap)

ioaddr = pci_iomap(pdev, bar, io_size);
if (!ioaddr) {
        printk(KERN_ERR "ioremap failed for device %s\n",
                pci_name(pdev));
        err = -EIO;
        goto err_out_free_res;
}

/* Init system & device */
// Keep track of the ioaddr in the netdev private data
lp->base = ioaddr;

// Use the IRQ line allocated by the PCI bus
dev->irq = pdev->irq;

// Initialize the netdev private data spinlock
spin_lock_init(&lp->lock);

// Attach the netdev to the PCI dev.
// Useful to retrieve the netdev in the PCI remove hook.
// netdev can't be a global variable: you could have
// several of them (multiple devices)

pci_set_drvdata(pdev, dev);

/* Set MAC address */
card_idx++;

adrp = (u16 *)dev->dev_addr;
adrp[0] = ioread16(ioaddr + MID_0L);
adrp[1] = ioread16(ioaddr + MID_0M);
adrp[2] = ioread16(ioaddr + MID_0H);

/* Link new device into r6040_root_dev */
lp->pdev = pdev;
lp->dev = dev;

/* Init RDC private data */
// RDC: name of the device vendor
lp->mcr0 = 0x1002;
lp->phy_addr = phy_table[card_idx];
lp->switch_sig = 0;

/* The RDC-specific entries in the device structure. */
dev->open = &r6040_open;                    // Called when ethx is opened
```

```
        dev->hard_start_xmit = &r6040_start_xmit;    // Called when a packet is emitted
                                                     // by the network stack
        dev->stop = &r6040_close;                    // Called when ethx is closed
        dev->get_stats = r6040_get_stats;            // Device rx and tx count stats
        dev->set_multicast_list = &r6040_multicast_list;
        dev->do_ioctl = &r6040_ioctl;
        dev->ethtool_ops = &netdev_ethtool_ops;      // Ops for ethtool, allowing to configure
                                                     // the ethx from userspace: set half/full duplex,
                                                     // change MTU...

        dev->tx_timeout = &r6040_tx_timeout;
        dev->watchdog_timeo = TX_TIMEOUT;
#ifdef CONFIG_NET_POLL_CONTROLLER
        dev->poll_controller = r6040_poll_controller;
#endif
        netif_napi_add(dev, &lp->napi, r6040_poll, 64);     // NAPI: New API - Uses polling mode
                                                            // when busy, and gets back to interrupt
                                                            // mode when it's done. This attaches the
                                                            // r6040_poll routine to lp->napi. This
                                                            // function will be used in polled mode.
        lp->mii_if.dev = dev;
        lp->mii_if.mdio_read = r6040_mdio_read;
        lp->mii_if.mdio_write = r6040_mdio_write;
        lp->mii_if.phy_id = lp->phy_addr;
        lp->mii_if.phy_id_mask = 0x1f;
        lp->mii_if.reg_num_mask = 0x1f;

        /* Register net device. After this dev->name assign */

        // The net device is ready - Activate it.
        // Allow it to be opened and accessed.

        err = register_netdev(dev);
        if (err) {
                printk(KERN_ERR DRV_NAME ": Failed to register net device\n");
                goto err_out_unmap;
        }
        return 0;

err_out_unmap:
        pci_iounmap(pdev, ioaddr);
err_out_free_res:
        pci_release_regions(pdev);
err_out_free_dev:
        free_netdev(dev);
err_out:
        return err;
}

static void __devexit r6040_remove_one(struct pci_dev *pdev)
{

        // Device remove function - Run when a remove event is received

        struct net_device *dev = pci_get_drvdata(pdev);

        // See the probe function for details
        unregister_netdev(dev);
        pci_release_regions(pdev);
        free_netdev(dev);
        pci_disable_device(pdev);
        pci_set_drvdata(pdev, NULL);
}


static struct pci_device_id r6040_pci_tbl[] = {
        { PCI_DEVICE(PCI_VENDOR_ID_RDC, 0x6040) }, // Table of supported devices
        { 0 }
};
MODULE_DEVICE_TABLE(pci, r6040_pci_tbl);
```

```
static struct pci_driver r6040_driver = {

        /* PCI Driver hooks */
        .name           = DRV_NAME,
        .id_table       = r6040_pci_tbl,  // Table of supported devices
        .probe          = r6040_init_one, // Loaded when the bus detects a matching device
        .remove         = __devexit_p(r6040_remove_one),
};


static int __init r6040_init(void)
{
        return pci_register_driver(&r6040_driver); // PCI driver hooks defined right above
}


static void __exit r6040_cleanup(void)
{
        pci_unregister_driver(&r6040_driver);
}

module_init(r6040_init);
module_exit(r6040_cleanup);
```