```
/* ne2k-pci.c: A NE2000 clone on PCI bus driver for Linux. */
/*
        A Linux device driver for PCI NE2000 clones.

        Authors and other copyright holders:
        1992-2000 by Donald Becker, NE2000 core and various modifications.
        1995-1998 by Paul Gortmaker, core modifications and PCI support.
        Copyright 1993 assigned to the United States Government as represented
        by the Director, National Security Agency.

        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 or derived from this code fall under the GPL and must
        retain the authorship, copyright and license notice.  This file is not
        a complete program and may only be used when the entire operating
        system is licensed under the GPL.

        The author may be reached as becker@scyld.com, or C/O
        Scyld Computing Corporation
        410 Severn Ave., Suite 210
        Annapolis MD 21403

        Issues remaining:
        People are making PCI ne2000 clones! Oh the horror, the horror...
        Limited full-duplex support.
*/

#define DRV_NAME        "ne2k-pci"
#define DRV_VERSION     "1.03"
#define DRV_RELDATE     "9/22/2003"


/* The user-configurable values.
   These may be modified when a driver module is loaded.*/

static int debug = 1;                   /* 1 normal messages, 0 quiet .. 7 verbo
se. */

#define MAX_UNITS 8                             /* More are supported, limit onl
y on options */
/* Used to pass the full-duplex flag, etc. */
static int full_duplex[MAX_UNITS];
static int options[MAX_UNITS];

/* Force a non std. amount of memory.  Units are 256 byte pages. */
/* #define PACKETBUF_MEMSIZE    0x40 */


#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/ethtool.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>

#include <asm/system.h>
#include <asm/io.h>
#include <asm/irq.h>
#include <asm/uaccess.h>

#include "8390.h"

/* These identify the driver base version and may not be removed. */
static char version[] __devinitdata =
KERN_INFO DRV_NAME ".c:v" DRV_VERSION " " DRV_RELDATE " D. Becker/P. Gortmaker\n"
KERN_INFO " http://www.scyld.com/network/ne2k-pci.html\n";
```

```

#if defined(__powerpc__)
#define inl_le(addr)  le32_to_cpu(inl(addr))
#define inw_le(addr)  le16_to_cpu(inw(addr))
#endif

#define PFX DRV_NAME ": "

MODULE_AUTHOR("Donald Becker / Paul Gortmaker");
MODULE_DESCRIPTION("PCI NE2000 clone driver");
MODULE_LICENSE("GPL");

MODULE_PARM(debug, "i");
MODULE_PARM(options, "1-" __MODULE_STRING(MAX_UNITS) "i");
MODULE_PARM(full_duplex, "1-" __MODULE_STRING(MAX_UNITS) "i");
MODULE_PARM_DESC(debug, "debug level (1-2)");
MODULE_PARM_DESC(options, "Bit 5: full duplex");
MODULE_PARM_DESC(full_duplex, "full duplex setting(s) (1)");

/* Some defines that people can play with if so inclined. */

/* Use 32 bit data-movement operations instead of 16 bit. */
#define USE_LONGIO

/* Do we implement the read before write bugfix ? */
/* #define NE_RW_BUGFIX */

/* Flags.  We rename an existing ei_status field to store flags! */
/* Thus only the low 8 bits are usable for non-init-time flags. */
#define ne2k_flags reg0
enum {
        ONLY_16BIT_IO=8, ONLY_32BIT_IO=4,       /* Chip can do only 16/32-bit xf
ers. */
        FORCE_FDX=0x20,                                         /* User override
. */
        REALTEK_FDX=0x40, HOLTEK_FDX=0x80,
        STOP_PG_0x60=0x100,
};

enum ne2k_pci_chipsets {
        CH_RealTek_RTL_8029 = 0,
        CH_Winbond_89C940,
        CH_Compex_RL2000,
        CH_KTI_ET32P2,
        CH_NetVin_NV5000SC,
        CH_Via_86C926,
        CH_SureCom_NE34,
        CH_Winbond_W89C940F,
        CH_Holtek_HT80232,
        CH_Holtek_HT80229,
        CH_Winbond_89C940_8c4a,
};


static struct {
        char *name;
        int flags;
} pci_clone_list[] __devinitdata = {
        {"RealTek RTL-8029", REALTEK_FDX},
        {"Winbond 89C940", 0},
        {"Compex RL2000", 0},
        {"KTI ET32P2", 0},
        {"NetVin NV5000SC", 0},
        {"Via 86C926", ONLY_16BIT_IO},
        {"SureCom NE34", 0},
        {"Winbond W89C940F", 0},
        {"Holtek HT80232", ONLY_16BIT_IO | HOLTEK_FDX},
        {"Holtek HT80229", ONLY_32BIT_IO | HOLTEK_FDX | STOP_PG_0x60 },
        {"Winbond W89C940(misprogrammed)", 0},
```

```
        {NULL,}
};


static struct pci_device_id ne2k_pci_tbl[] = {
        { 0x10ec, 0x8029, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_RealTek_RTL_8029 },
        { 0x1050, 0x0940, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_89C940 },
        { 0x11f6, 0x1401, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Compex_RL2000 },
        { 0x8e2e, 0x3000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_KTI_ET32P2 },
        { 0x4a14, 0x5000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_NetVin_NV5000SC },
        { 0x1106, 0x0926, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Via_86C926 },
        { 0x10bd, 0x0e34, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_SureCom_NE34 },
        { 0x1050, 0x5a5a, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_W89C940F },
        { 0x12c3, 0x0058, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80232 },
        { 0x12c3, 0x5598, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80229 },
        { 0x8c4a, 0x1980, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_89C940_8c4a }
,
        { 0, }
};
MODULE_DEVICE_TABLE(pci, ne2k_pci_tbl);


/* ---- No user-serviceable parts below ---- */

#define NE_BASE  (dev→base_addr)
#define NE_CMD          0x00
#define NE_DATAPORT     0x10    /* NatSemi-defined port window offset. */
#define NE_RESET        0x1f    /* Issue a read to reset, a write to clear. */
#define NE_IO_EXTENT    0x20

#define NESM_START_PG   0x40    /* First page of TX buffer */
#define NESM_STOP_PG    0x80    /* Last page +1 of RX ring */


static int ne2k_pci_open(struct net_device *dev);
static int ne2k_pci_close(struct net_device *dev);

static void ne2k_pci_reset_8390(struct net_device *dev);
static void ne2k_pci_get_8390_hdr(struct net_device *dev, struct e8390_pkt_hdr *
hdr,
                          int ring_page);
static void ne2k_pci_block_input(struct net_device *dev, int count,
                          struct sk_buff *skb, int ring_offset);
static void ne2k_pci_block_output(struct net_device *dev, const int count,
                const unsigned char *buf, const int start_page);
static struct ethtool_ops ne2k_pci_ethtool_ops;
```

```

/* There is no room in the standard 8390 structure for extra info we need,
   so we build a meta/outer-wrapper structure.. */
struct ne2k_pci_card {
        struct net_device *dev;
        struct pci_dev *pci_dev;
};



/*
  NEx000-clone boards have a Station Address (SA) PROM (SAPROM) in the packet
  buffer memory space.  By-the-spec NE2000 clones have 0x57,0x57 in bytes
  0x0e,0x0f of the SAPROM, while other supposed NE2000 clones must be
  detected by their SA prefix.

  Reading the SAPROM from a word-wide card with the 8390 set in byte-wide
  mode results in doubled values, which can be detected and compensated for.

  The probe is also responsible for initializing the card and filling
  in the 'dev' and 'ei_status' structures.
*/


static int __devinit ne2k_pci_init_one (struct pci_dev *pdev,
                                     const struct pci_device_id *ent)
{
        struct net_device *dev;
        int i;
        unsigned char SA_prom[32];
        int start_page, stop_page;
        int irq, reg0, chip_idx = ent→driver_data;
        static unsigned int fnd_cnt;
        long ioaddr;
        int flags = pci_clone_list[chip_idx].flags;

/* 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

        fnd_cnt++;

        i = pci_enable_device (pdev);
        if (i)
                return i;

        ioaddr = pci_resource_start (pdev, 0);
        irq = pdev→irq;

        if (¬ioaddr ∨ ((pci_resource_flags (pdev, 0) & IORESOURCE_IO) ≡ 0)) {
                printk (KERN_ERR PFX "no I/O resource at PCI BAR #0\n");
                return -ENODEV;
        }

        if (request_region (ioaddr, NE_IO_EXTENT, DRV_NAME) ≡ NULL) {
                printk (KERN_ERR PFX "I/O resource 0x%x @ 0x%lx busy\n",
                        NE_IO_EXTENT, ioaddr);
                return -EBUSY;
        }

        reg0 = inb(ioaddr);
        if (reg0 ≡ 0xFF)
                goto err_out_free_res;

        /* Do a preliminary verification that we have a 8390. */
        {
```

```
                int regd;
                outb(E8390_NODMA+E8390_PAGE1+E8390_STOP, ioaddr + E8390_CMD);
                regd = inb(ioaddr + 0x0d);
                outb(0xff, ioaddr + 0x0d);
                outb(E8390_NODMA+E8390_PAGE0, ioaddr + E8390_CMD);
                inb(ioaddr + EN0_COUNTER0); /* Clear the counter by reading. */
                if (inb(ioaddr + EN0_COUNTER0) ≠ 0) {
                        outb(reg0, ioaddr);
                        outb(regd, ioaddr + 0x0d);      /* Restore the old value
s. */
                        goto err_out_free_res;
                }
        }

        /* Allocate net_device, dev->priv; fill in 8390 specific dev fields. */
        dev = alloc_ei_netdev();
        if (¬dev) {
                printk (KERN_ERR PFX "cannot allocate ethernet device\n");
                goto err_out_free_res;
        }
        SET_MODULE_OWNER(dev);
        SET_NETDEV_DEV(dev, &pdev→dev);

        /* Reset card. Who knows what dain-bramaged state it was left in. */
        {
                unsigned long reset_start_time = jiffies;

                outb(inb(ioaddr + NE_RESET), ioaddr + NE_RESET);

                /* This looks like a horrible timing loop, but it should never t
ake
                   more than a few cycles.
                */
                while ((inb(ioaddr + EN0_ISR) & ENISR_RESET) ≡ 0)
                        /* Limit wait: '2' avoids jiffy roll-over. */
                        if (jiffies - reset_start_time > 2) {
                                printk(KERN_ERR PFX "Card failure (no reset ack).\n");
                                goto err_out_free_netdev;
                        }

                outb(0xff, ioaddr + EN0_ISR);           /* Ack all intr. */
        }

        /* Read the 16 bytes of station address PROM.
           We must first initialize registers, similar to NS8390_init(eifdev, 0)
.
           We can't reliably read the SAPROM address without this.
           (I learned the hard way!). */
        {
                struct {unsigned char value, offset; } program_seq[] = {
                        {E8390_NODMA+E8390_PAGE0+E8390_STOP, E8390_CMD}, /* Sele
ct page 0*/
                        {0x49,  EN0_DCFG},      /* Set word-wide access. */
                        {0x00,  EN0_RCNTLO},    /* Clear the count regs. */
                        {0x00,  EN0_RCNTHI},
                        {0x00,  EN0_IMR},       /* Mask completion irq. */
                        {0xFF,  EN0_ISR},
                        {E8390_RXOFF, EN0_RXCR},        /* 0x20  Set to monitor
*/
                        {E8390_TXOFF, EN0_TXCR},        /* 0x02  and loopback mo
de. */
                        {32,    EN0_RCNTLO},
                        {0x00,  EN0_RCNTHI},
                        {0x00,  EN0_RSARLO},    /* DMA starting at 0x0000. */
                        {0x00,  EN0_RSARHI},
                        {E8390_RREAD+E8390_START, E8390_CMD},
                };
                for (i = 0; i < sizeof(program_seq)/sizeof(program_seq[0]); i++)
                        outb(program_seq[i].value, ioaddr + program_seq[i].offse
```

```
t);

        }

        /* Note: all PCI cards have at least 16 bit access, so we don't have
           to check for 8 bit cards.  Most cards permit 32 bit access. */
        if (flags & ONLY_32BIT_IO) {
                for (i = 0; i < 4 ; i++)
                        ((u32 *)SA_prom)[i] = le32_to_cpu(inl(ioaddr + NE_DATAPO
RT));
        } else
                for(i = 0; i < 32 /*sizeof(SA_prom)*/; i++)
                        SA_prom[i] = inb(ioaddr + NE_DATAPORT);

        /* We always set the 8390 registers for word mode. */
        outb(0x49, ioaddr + EN0_DCFG);
        start_page = NESM_START_PG;

        stop_page = flags & STOP_PG_0x60 ? 0x60 : NESM_STOP_PG;

        /* Set up the rest of the parameters. */
        dev→irq = irq;
        dev→base_addr = ioaddr;
        pci_set_drvdata(pdev, dev);

        ei_status.name = pci_clone_list[chip_idx].name;
        ei_status.tx_start_page = start_page;
        ei_status.stop_page = stop_page;
        ei_status.word16 = 1;
        ei_status.ne2k_flags = flags;
        if (fnd_cnt < MAX_UNITS) {
                if (full_duplex[fnd_cnt] > 0  ∨  (options[fnd_cnt] & FORCE_FDX)
)
                        ei_status.ne2k_flags |= FORCE_FDX;
        }

        ei_status.rx_start_page = start_page + TX_PAGES;
#ifdef PACKETBUF_MEMSIZE
        /* Allow the packet buffer size to be overridden by know-it-alls. */
        ei_status.stop_page = ei_status.tx_start_page + PACKETBUF_MEMSIZE;
#endif

        ei_status.reset_8390 = &ne2k_pci_reset_8390;
        ei_status.block_input = &ne2k_pci_block_input;
        ei_status.block_output = &ne2k_pci_block_output;
        ei_status.get_8390_hdr = &ne2k_pci_get_8390_hdr;
        ei_status.priv = (unsigned long) pdev;
        dev→open = &ne2k_pci_open;
        dev→stop = &ne2k_pci_close;
        dev→ethtool_ops = &ne2k_pci_ethtool_ops;
#ifdef CONFIG_NET_POLL_CONTROLLER
        dev→poll_controller = ei_poll;
#endif
        NS8390_init(dev, 0);

        i = register_netdev(dev);
        if (i)
                goto err_out_free_netdev;

        printk("%s: %s found at %#lx, IRQ %d, ",
                   dev→name, pci_clone_list[chip_idx].name, ioaddr, dev→irq);
        for(i = 0; i < 6; i++) {
                printk("%2.2X%s", SA_prom[i], i ≡ 5 ? ".\n": ":");
                dev→dev_addr[i] = SA_prom[i];
        }

        return 0;

err_out_free_netdev:
```

```
        free_netdev (dev);
err_out_free_res:
        release_region (ioaddr, NE_IO_EXTENT);
        pci_set_drvdata (pdev, NULL);
        return -ENODEV;

}

/*
 * Magic incantation sequence for full duplex on the supported cards.
 */
static inline int set_realtek_fdx(struct net_device *dev)
{
        long ioaddr = dev→base_addr;

        outb(0xC0 + E8390_NODMA, ioaddr + NE_CMD); /* Page 3 */
        outb(0xC0, ioaddr + 0x01); /* Enable writes to CONFIG3 */
        outb(0x40, ioaddr + 0x06); /* Enable full duplex */
        outb(0x00, ioaddr + 0x01); /* Disable writes to CONFIG3 */
        outb(E8390_PAGE0 + E8390_NODMA, ioaddr + NE_CMD); /* Page 0 */
        return 0;
}

static inline int set_holtek_fdx(struct net_device *dev)
{
        long ioaddr = dev→base_addr;

        outb(inb(ioaddr + 0x20) | 0x80, ioaddr + 0x20);
        return 0;
}

static int ne2k_pci_set_fdx(struct net_device *dev)
{
        if (ei_status.ne2k_flags & REALTEK_FDX)
                return set_realtek_fdx(dev);
        else if (ei_status.ne2k_flags & HOLTEK_FDX)
                return set_holtek_fdx(dev);

        return -EOPNOTSUPP;
}

static int ne2k_pci_open(struct net_device *dev)
{
        int ret = request_irq(dev→irq, ei_interrupt, SA_SHIRQ, dev→name, dev);
        if (ret)
                return ret;

        if (ei_status.ne2k_flags & FORCE_FDX)
                ne2k_pci_set_fdx(dev);

        ei_open(dev);
        return 0;
}

static int ne2k_pci_close(struct net_device *dev)
{
        ei_close(dev);
        free_irq(dev→irq, dev);
        return 0;
}

/* Hard reset the card.  This used to pause for the same period that a
   8390 reset command required, but that shouldn't be necessary. */
static void ne2k_pci_reset_8390(struct net_device *dev)
{
        unsigned long reset_start_time = jiffies;

        if (debug > 1) printk("%s: Resetting the 8390 t=%ld...",
                                                  dev→name, jiffies);
```

```

        outb(inb(NE_BASE + NE_RESET), NE_BASE + NE_RESET);

        ei_status.txing = 0;
        ei_status.dmaing = 0;

        /* This check _should_not_ be necessary, omit eventually. */
        while ((inb(NE_BASE+EN0_ISR) & ENISR_RESET) ≡ 0)
                if (jiffies - reset_start_time > 2) {
                        printk("%s: ne2k_pci_reset_8390() did not complete.\n", dev→name);
                        break;
                }
        outb(ENISR_RESET, NE_BASE + EN0_ISR);   /* Ack intr. */
}

/* Grab the 8390 specific header. Similar to the block_input routine, but
   we don't need to be concerned with ring wrap as the header will be at
   the start of a page, so we optimize accordingly. */

static void ne2k_pci_get_8390_hdr(struct net_device *dev, struct e8390_pkt_hdr *
hdr, int ring_page)
{

        long nic_base = dev→base_addr;

        /* This *shouldn't* happen. If it does, it's the last thing you'll see *
/
        if (ei_status.dmaing) {
                printk("%s: DMAing conflict in ne2k_pci_get_8390_hdr "
                           "[DMAstat:%d][irqlock:%d].\n",
                           dev→name, ei_status.dmaing, ei_status.irqlock);
                return;
        }

        ei_status.dmaing |= 0x01;
        outb(E8390_NODMA+E8390_PAGE0+E8390_START, nic_base+ NE_CMD);
        outb(sizeof(struct e8390_pkt_hdr), nic_base + EN0_RCNTLO);
        outb(0, nic_base + EN0_RCNTHI);
        outb(0, nic_base + EN0_RSARLO);         /* On page boundary */
        outb(ring_page, nic_base + EN0_RSARHI);
        outb(E8390_RREAD+E8390_START, nic_base + NE_CMD);

        if (ei_status.ne2k_flags & ONLY_16BIT_IO) {
                insw(NE_BASE + NE_DATAPORT, hdr, sizeof(struct e8390_pkt_hdr)>>1
);
        } else {
                *(u32*)hdr = le32_to_cpu(inl(NE_BASE + NE_DATAPORT));
                le16_to_cpus(&hdr→count);
        }

        outb(ENISR_RDC, nic_base + EN0_ISR);    /* Ack intr. */
        ei_status.dmaing &= ~0x01;
}

/* Block input and output, similar to the Crynwr packet driver.  If you
   are porting to a new ethercard, look at the packet driver source for hints.
   The NEx000 doesn't share the on-board packet memory -- you have to put
   the packet out through the "remote DMA" dataport using outb. */

static void ne2k_pci_block_input(struct net_device *dev, int count,
                                 struct sk_buff *skb, int ring_offset)
{
        long nic_base = dev→base_addr;
        char *buf = skb→data;

        /* This *shouldn't* happen. If it does, it's the last thing you'll see *
/
        if (ei_status.dmaing) {
                printk("%s: DMAing conflict in ne2k_pci_block_input "
```

```
                           "[DMAstat:%d][irqlock:%d].\n",
                           dev→name, ei_status.dmaing, ei_status.irqlock);
                return;
        }
        ei_status.dmaing |= 0x01;
        if (ei_status.ne2k_flags & ONLY_32BIT_IO)
                count = (count + 3) & 0xFFFC;
        outb(E8390_NODMA+E8390_PAGE0+E8390_START, nic_base+ NE_CMD);
        outb(count & 0xff, nic_base + EN0_RCNTLO);
        outb(count >> 8, nic_base + EN0_RCNTHI);
        outb(ring_offset & 0xff, nic_base + EN0_RSARLO);
        outb(ring_offset >> 8, nic_base + EN0_RSARHI);
        outb(E8390_RREAD+E8390_START, nic_base + NE_CMD);

        if (ei_status.ne2k_flags & ONLY_16BIT_IO) {
                insw(NE_BASE + NE_DATAPORT,buf,count>>1);
                if (count & 0x01) {
                        buf[count-1] = inb(NE_BASE + NE_DATAPORT);
                }
        } else {
                insl(NE_BASE + NE_DATAPORT, buf, count>>2);
                if (count & 3) {
                        buf += count & ~3;
                        if (count & 2) {
                                u16 *b = (u16 *)buf;

                                *b++ = le16_to_cpu(inw(NE_BASE + NE_DATAPORT));
                                buf = (char *)b;
                        }
                        if (count & 1)
                                *buf = inb(NE_BASE + NE_DATAPORT);
                }
        }

        outb(ENISR_RDC, nic_base + EN0_ISR);    /* Ack intr. */
        ei_status.dmaing &= ~0x01;
}

static void ne2k_pci_block_output(struct net_device *dev, int count,
                                  const unsigned char *buf, const int start_page
)
{
        long nic_base = NE_BASE;
        unsigned long dma_start;

        /* On little-endian it’s always safe to round the count up for
           word writes. */
        if (ei_status.ne2k_flags & ONLY_32BIT_IO)
                count = (count + 3) & 0xFFFC;
        else
                if (count & 0x01)
                        count++;

        /* This *shouldn’t* happen. If it does, it’s the last thing you’ll see *
/
        if (ei_status.dmaing) {
                printk("%s: DMAing conflict in ne2k_pci_block_output."
                           "[DMAstat:%d][irqlock:%d]\n",
                           dev→name, ei_status.dmaing, ei_status.irqlock);
                return;
        }
        ei_status.dmaing |= 0x01;
        /* We should already be in page 0, but to be safe... */
        outb(E8390_PAGE0+E8390_START+E8390_NODMA, nic_base + NE_CMD);

#ifdef NE8390_RW_BUGFIX
        /* Handle the read-before-write bug the same way as the
           Crynwr packet driver -- the NatSemi method doesn’t work.
           Actually this doesn’t always work either, but if you have
```

```
           problems with your NEx000 this is better than nothing! */
        outb(0x42, nic_base + EN0_RCNTLO);
        outb(0x00, nic_base + EN0_RCNTHI);
        outb(0x42, nic_base + EN0_RSARLO);
        outb(0x00, nic_base + EN0_RSARHI);
        outb(E8390_RREAD+E8390_START, nic_base + NE_CMD);
#endif
        outb(ENISR_RDC, nic_base + EN0_ISR);

   /* Now the normal output. */
        outb(count & 0xff, nic_base + EN0_RCNTLO);
        outb(count >> 8,   nic_base + EN0_RCNTHI);
        outb(0x00, nic_base + EN0_RSARLO);
        outb(start_page, nic_base + EN0_RSARHI);
        outb(E8390_RWRITE+E8390_START, nic_base + NE_CMD);
        if (ei_status.ne2k_flags & ONLY_16BIT_IO) {
                outsw(NE_BASE + NE_DATAPORT, buf, count>>1);
        } else {
                outsl(NE_BASE + NE_DATAPORT, buf, count>>2);
                if (count & 3) {
                        buf += count & ~3;
                        if (count & 2) {
                                u16 *b = (u16 *)buf;

                                outw(cpu_to_le16(*b++), NE_BASE + NE_DATAPORT);
                                buf = (char *)b;
                        }
                }
        }

        dma_start = jiffies;

        while ((inb(nic_base + EN0_ISR) & ENISR_RDC) ≡ 0)
                if (jiffies - dma_start > 2) {                  /* Avoid clock r
oll-over. */
                        printk(KERN_WARNING "%s: timeout waiting for Tx RDC.\n", dev→nam
e);
                        ne2k_pci_reset_8390(dev);
                        NS8390_init(dev,1);
                        break;
                }

        outb(ENISR_RDC, nic_base + EN0_ISR);    /* Ack intr. */
        ei_status.dmaing &= ~0x01;
        return;
}

static void ne2k_pci_get_drvinfo(struct net_device *dev,
                                 struct ethtool_drvinfo *info)
{
        struct ei_device *ei = dev→priv;
        struct pci_dev *pci_dev = (struct pci_dev *) ei→priv;

        strcpy(info→driver, DRV_NAME);
        strcpy(info→version, DRV_VERSION);
        strcpy(info→bus_info, pci_name(pci_dev));
}

static struct ethtool_ops ne2k_pci_ethtool_ops = {
        .get_drvinfo            = ne2k_pci_get_drvinfo,
        .get_tx_csum            = ethtool_op_get_tx_csum,
        .get_sg                 = ethtool_op_get_sg,
};

static void __devexit ne2k_pci_remove_one (struct pci_dev *pdev)
{
        struct net_device *dev = pci_get_drvdata(pdev);

        if (¬dev)
```

```
                BUG();

        unregister_netdev(dev);
        release_region(dev->base_addr, NE_IO_EXTENT);
        free_netdev(dev);
        pci_disable_device(pdev);
        pci_set_drvdata(pdev, NULL);
}


static struct pci_driver ne2k_driver = {
        .name           = DRV_NAME,
        .probe          = ne2k_pci_init_one,
        .remove         = __devexit_p(ne2k_pci_remove_one),
        .id_table       = ne2k_pci_tbl,
};


static int __init ne2k_pci_init(void)
{
/* when a module, this is printed whether or not devices are found in probe */
#ifdef MODULE
        printk(version);
#endif
        return pci_module_init (&ne2k_driver);
}


static void __exit ne2k_pci_cleanup(void)
{
        pci_unregister_driver (&ne2k_driver);
}

module_init(ne2k_pci_init);
module_exit(ne2k_pci_cleanup);
```