A Linux Implementation of HIP

Brian Vanderpool Zak Smith

Project Report
ECE750 Fall 1998

Prof. Parmesh Ramanathan

Dept. of Electrical and Computer Engineering
University of Wisconsin - Madison

December 15, 1998

Abstract

As network traffic becomes more demanding, the overhead of invoking a hardware interrupt for each packet becomes high. For regular, frequent, traffic, such as video or audio streaming, a hybrid interrupt polling scheme (HIP) can be used to decrease this overhead. When packets arrive regularly and frequently enough, HIP disables network interrupts and polls the network interface card (NIC) during regular kernel context-switches (1ms resolution). Less overhead allows higher application CPU utilization and higher network bandwidth.

We have implemented HIP in the Linux 2.0.35 kernel for the 3Com 3c590 "Vortex" NIC (10BaseT), and have collected rudimentary performance information. For high packet traffic, HIP can reduce overhead up to 20%. HIP increases packet latency except when packets are very regular, and throughput for a normal application such as FTP is increased by 8.7%.

Contents

1 Introduction

1.1 Motivation

In modern computers, most external, irregular events are triggered by hardware interrupts. Interrupts allow the CPU to run other tasks until it is signalled that work for an external event must be done. At this point, the CPU saves state, jumps into the kernel, does whatever processing is necessary, restores state, and then resumes the original task. This is fine for rare events such as keyboard or mouse input which might only happen a few times a second.

However, for events which happen hundreds or thousands of times per second, the overhead involved can become large compared to the regular task load. This can limit the processing time available for other tasks as well as limit the number of external events which can be handled.

As modern computers are used more and more for network-intensive tasks such as video or audio streaming, the performance of the network interface card (NIC) becomes an issue. It is possible to *poll* for network packets instead of waiting for an interrupt to arrive if it is possible to reasonably accurately predict when the next packet will arrive. NIC polling can then happen during a regularly-scheduled kernel context-switch instead of a costly hardware interrupt. If polling can be done in an intelligent manner, it should be able to do two things:

1. Increase CPU available for other tasks
2. Increase bandwidth available

1.2 Algorithm

Constantinos Dovrolis and Brad Thayer have proposed an algorithm called HIP which automatically switches between interrupt mode and polling mode, and adaptively adjusts the polling parameters based on network traffic characteristics.[1]

The algorithm keeps an average inter-arrival time

$$\overline{I} = \alpha \overline{I} + (1 - \alpha) D$$

where α $(0 \leq \alpha \leq 1)$ is the average inter-arrival weight factor: if $\alpha = 0.8$, then the last inter-arrival time will account for 20% of $\overline{I}$. D is the last inter-arrival time.

In addition, the packet inter-arrival variance σ_I^2 is estimated as

$$\sigma_I^2 = \beta \sigma_I^2 + (1 - \beta) \left(\overline{I} - D\right)^2$$

where β $(0 \leq \beta \leq 1)$ is the inter-arrival variance weight factor, which works like α.

If packets are arriving at widely spaced random intervals, it is best to stay in interrupt mode, but if they are arriving frequently — incurring high interrupt overhead — and regularly enough (the next packet arrival time can be predicted), it is best to use polling mode.

[1] HIP: Hybrid Interrupt-Polling for the Network Interface, Constantinos Dovrolis and Brad Thayer, Dept. of Electrical and Computer Engr, Dept. of Computer Science, University of Wisconsin, Madison

HIP switches to polling mode if

$$\frac{\sigma_I}{\overline{I}} < \gamma$$

AND

$$\overline{I} < T_P^{MAX}$$

where γ is the prediction threshold: reducing γ requires more predictable packet inter-arrivals before switching to polling mode. T_P^{MAX} is the largest allowed polling interval.

Conversely, HIP switches to interrupt mode if

$$\frac{\sigma_I}{\overline{I}} \geq \gamma$$

OR

$$\overline{I} \geq T_P^{MAX}$$

OR

$$P_U \geq P_U^{MAX}$$

where P_U is the number of number of unsuccessful polls, and P_U^{MAX} is the maximum number of unsuccessful polls allowed before switching.

The polling interval T_P is adjusted with the following algorithm:

$$T_P = \overline{I} + \phi\sigma_I$$

where ϕ is the slack parameter, which adjusts how pessimistic the estimation is. A large ϕ value will increase the polling interval and reduce the number of unsuccessful polls. If the resulting T_P is not in the range T_P^{MIN} to T_P^{MAX}, it is adjusted to fix in that range.

1.3 Purpose

In the original HIP paper, the algorithm was studied in a simulator using short network traces. Our purpose for this project was twofold:

1. Prove that HIP is implementable in a modern UNIX kernel

2. Gather rudimentary performance information using more realistic traffic

2 Implementation

The Linux kernel supports both device interrupts and kernel timers through its interrupt handling structure. When a device asserts an interrupt, the processor disables future interrupts, switches contexts to kernel mode, saves processor flags, and examines the interrupt for its source. Possible sources may be a keyboard, hard drive, network card, or a system timer chip. During this time, no other interrupts can be handled, and nothing else in the system can be run. Therefore it is desirable to return from interrupt mode as soon as possible and re-enable system interrupts. The Linux kernel has mechanisms, called bottom half handlers, to allow deferral of work to a later time. There are separate bottom half handlers for devices and timers.

The source is then translated to an offset into the `irq_action vector` vector. When the device driver's interrupt handling routine is called, it may elect do queue some of the processing for a later time. To do this, the interrupt routine can mark the bottom half handler for that interrupt as active. When the kernel processes the task queues, the rest of the interrupt work will be handled.

This section discusses the interaction between device drivers, system timers, and the 3COM "Vortex" network card device driver in particular.

2.1 Device Interrupt Handling

When a device driver initializes, it registers its interrupt handling routine with the Linux kernel. There are two interrupt priorities it can choose from, fast or normal, determined by if the `SA_INTERRUPT` flag passed to the `request_irq` register routine. Fast routines run with all other interrupts disabled, whereas normal routines set up signal handling mechanisms and run with interrupts enabled. To do this, the device driver uses a set of Linux kernel services to request an interrupt, enable it and to disable it. It is important for the device driver to re-enable interrupts when the interrupt has been processed so that future work can be completed.

For HIP to function correctly it is necessary to prevent the network card from raising an interrupt. To do this, the kernel must send a command to the network card, disabling interrupt generation on the card. It isn't sufficient for the kernel to simply ignore interrupts from the network card when in polling mode because the interrupt overhead has already been occurred.

2.2 Timer Modification

Modern day microprocessors have programmable interval timer that periodically interrupts the processor. This interrupt is known as a system clock tick and allows the kernel to interrupt tasks at well defined intervals.

In the Linux kernel, time is measured in clock ticks since the system booted. This time is known as `jiffies` within the Linux kernel. In the unmodified Linux kernel, the `jiffies` counter is incremented every 10 ms.

The Linux kernel has two timer mechanisms. One consists of a static array of 32 pointer to `timer_struct` data structures, with a corresponding mask of which of these are active.

Timers are entered into this array primarily at system initialization time.

To provide support for additional, dynamic timers, the Linux kernel also maintains a linked list of `timer_list` data structures in ascending order of when the timer will expire.

Both methods use jiffies as their time measurement. Every system clock tick, the timer bottom half handler is marked as active so that next time the scheduler runs, the timer queues will be processed. When the timer queues are processed, the expire time in each `timer_list` and `time_struct` data structure is compared against the system `jiffies` counter. If the expire time is less than the system jiffies count, the timer routine is called. If the timer is static, the active bit is then cleared. In the case of a dynamic timer, it is removed from the list.

HIP requires high speed timers for a high polling rate in heavy traffic. The timer interrupt mechanism was modified to increment the `jiffies` counter every 1ms instead of the default 10ms.

2.3 3Com 3c509 "Vortex" NIC

The device driver for the 3c509 "Vortex" network card was modified to support the HIP mechanism. This required maintaining the state of the network driver, either interrupt or polling mode, as well as the necessary variables: the average inter-arrival time ($\overline{I}$), and the average inter-arrival time variance (σ_I^2). In addition, when in polling mode, the number of unsuccessful polls (P_U) also has to be counted.

The 3c509 driver was compiled as a loadable module to facilitate testing and tuning the HIP mechanism. On initialization of the module, the routine `vortex_open` is called. In this routine, a timer is added to the dynamic timer queue set to the default polling interval. Initialization of the network card then continues as in the unmodified driver. After the network card has been configured, a copy of the device information structure is stored away for easy reference from the timer function.

In both interrupt and polling mode, packets are read from the network card using the `vortex_interrupt` routine. This requires a few state checks within the routine to insure proper function. However, the latency of transferring packets from the network card into the kernel wasn't affected due to placing the switching calculations at the end of the interrupt routine, executed just before returning control to other system tasks.

2.3.1 Switching to Polling Mode

Every time the interrupt handling routine is executed in interrupt mode, all packets are first transferred from the network card into the kernel. Then the time since the last packet arrival is computed and $\overline{I}$ and σ_I^2 are updated. If the conditions for switching to polling mode are met, the state is changed to polling mode, interrupts are disabled on the network card, and the polling timer is added to the system timer list for execution at T_P jiffies in the future.

2.3.2 Switching to Interrupt Mode

Every time the polling routine is executed, the parameters $\overline{I}$ and σ_I^2 are first examined. The `vortex_interrupt` routine is called and packets are transferred from the network card into the kernel. After the transfer, the conditions for switching back to interrupt mode are examined. If any of the switching conditions are met, interrupts on the network card are reenabled and the state is switched back into interrupt mode. Otherwise, a new timer is scheduled to fire T_P jiffies in the future.

3 Methodology

The desired result of HIP is to reduce overhead of receiving packets to that of polling, as well as maintain the low latency offered by interrupts. The test setup consisted of a Pentium Pro 180Mhz system with 64MB RAM running Linux 2.0.35 using the modified HIP network driver (the server), connected to a Pentium II 266Mhz system (the client) over an isolated 10 Mbit ethernet network.

To quantify the effects of the HIP mechanism on the 3c590 "Vortex" driver, the following three tests were used.

3.1 Overhead measurement

To measure the overhead associated with interrupt, polling, and the HIP mechanism, a program was written to compute as many square roots as possible in a fixed amount of time. A server program on the HIP machine received packets sent out in variable rates from a client machine. The interarrival time of packets was varied by the client and the number of *sqrt* iterations completed by the server were recorded.

3.2 Round trip latency measurement

To measure the latency of receiving, processing, and returning network packets, the client program was modified to send packets in random intervals to the UDP echo port on the HIP machine. The round trip time if a packet can be broken down into this equation.

$$T_{tot} = 2(T_{send} + T_{trans}) + T_{rcv_{intr}} + T_{rcv_{HIP}}$$

The total round-trip time of a packet consists of two nearly constant send latencies, 2 constant transfer times, the nearly constant receive latency in interrupt mode, and the varied receive latency of HIP mode. Because the average latency of a interrupt is nearly constant on an otherwise idle machine, by measuring total round trip time, the effect of polling in HIP can be measured.

3.3 File Transfer Rate

To measure raw throughput of the HIP interface, a series of file transfers over ftp was conducted. A 16MB file was transferred several times to have the entire file cached in server memory, and then the best value of T_P^{MAX} was determined.

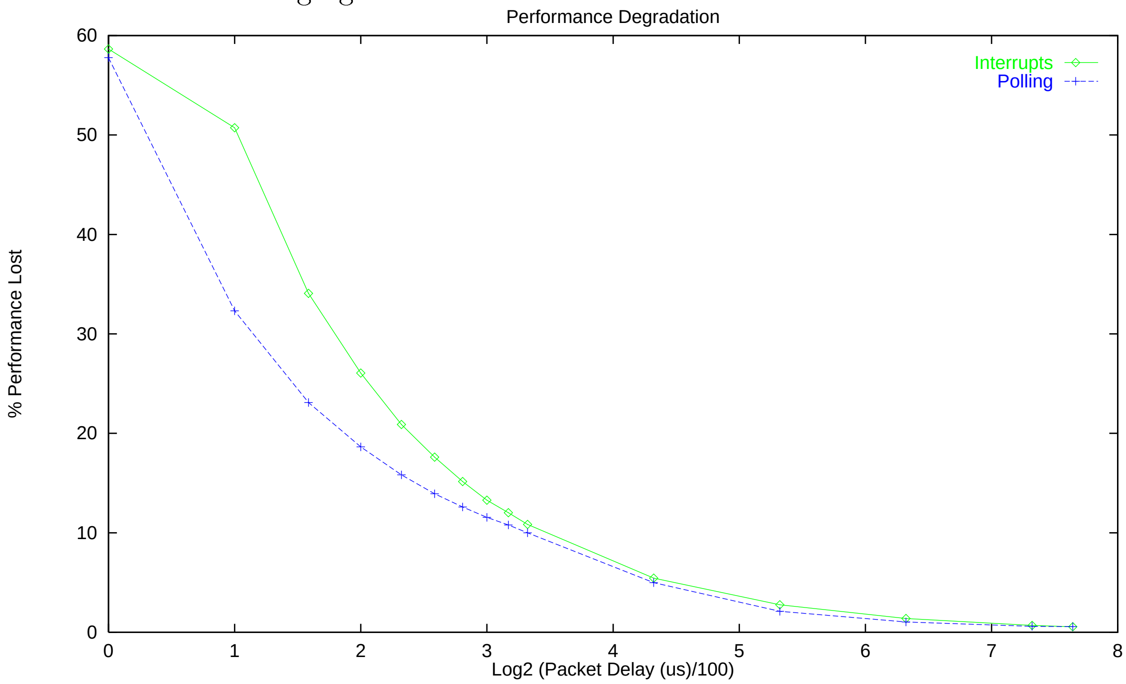
4 Results

For the following experiments, the following parameter values were used:

α	0.3
β	0.5
γ	5.0
σ	2.5
T_P^{MIN}	1ms

4.1 Overhead

Overhead was measured by running a CPU-intensive application on the HIP machine and measuring its performance degradation when the machine was receiving packets at different inter-arrival times ranging from 100us to 20ms.



The graph above shows the CPU overhead as a function of inter-arrival time compared to a baseline (100%) with no network traffic. To better illustrate the difference, the X-axis is log-based: the first data-point (0) corresponds to 100us, 1 to 200us, 2 to 400us, ..., and 8 to 25.6ms.

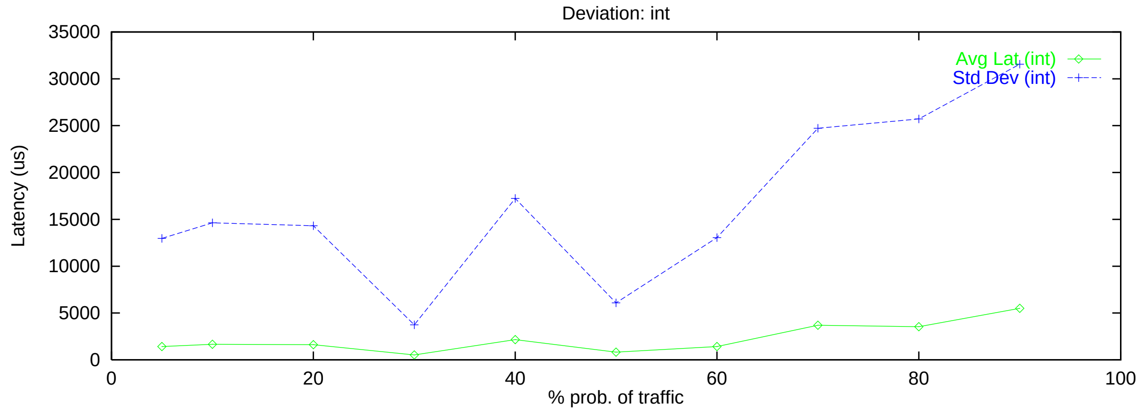
It is clear that polling *always* incurs less overhead than interrupts, and when the inter-arrival times are less than 1ms, polling incurs *much* less overhead than interrupts, up to about a 20% improvement.

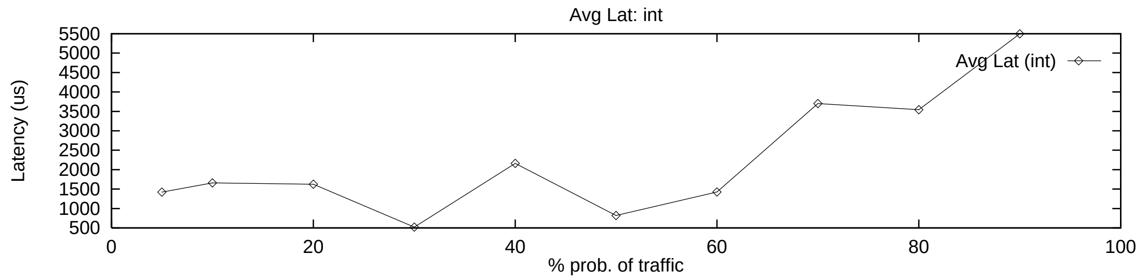
4.2 Latency Measurements

The latency numbers are for total round-trip time based on the following network traffic model: every 200 us, a decision is made to send a packet based on a traffic threshold X. If

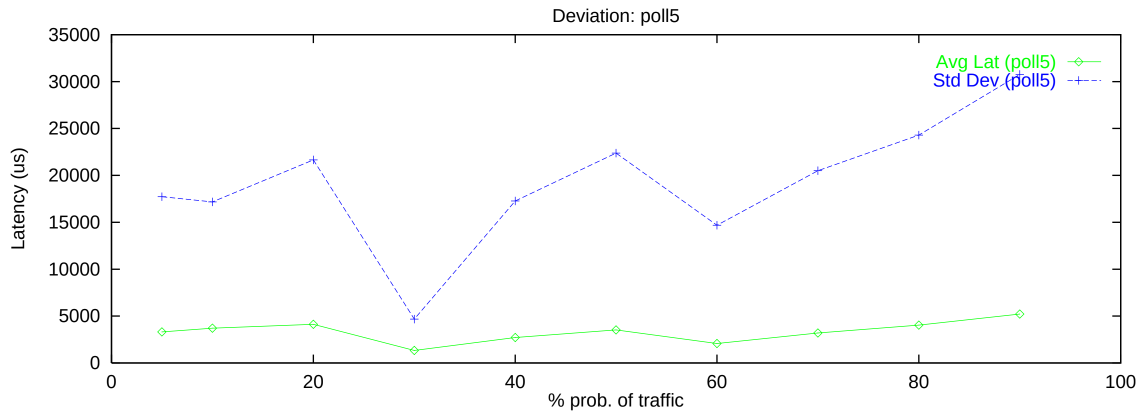
$X = 60$, then 60% of the decisions made at 200 us intervals will be to send packets, and 40% will be to wait.

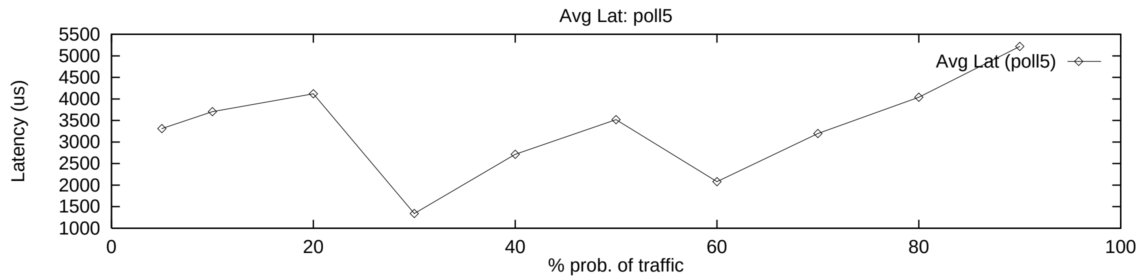
4.2.1 Interrupts Only



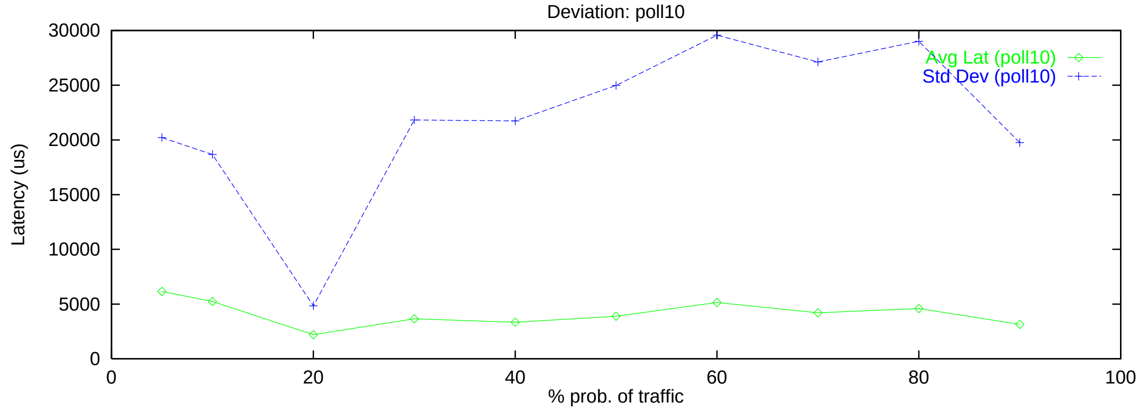


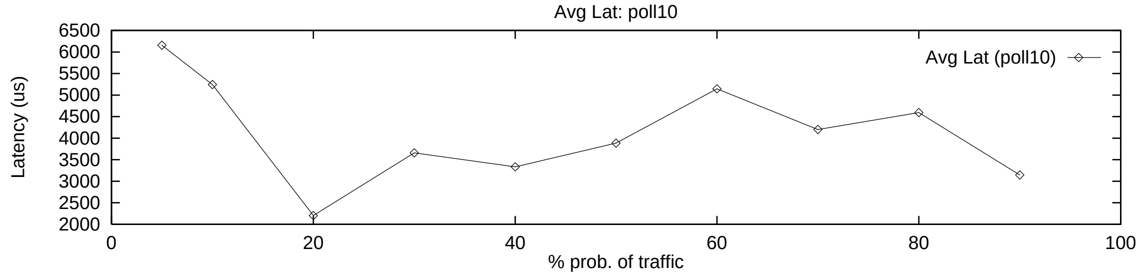
4.2.2 HIP, $T_P^{MAX} = 5$



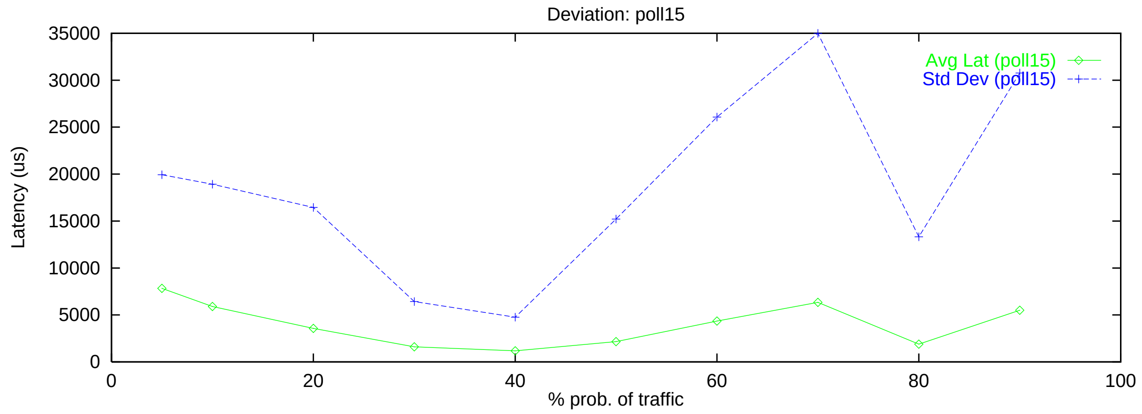


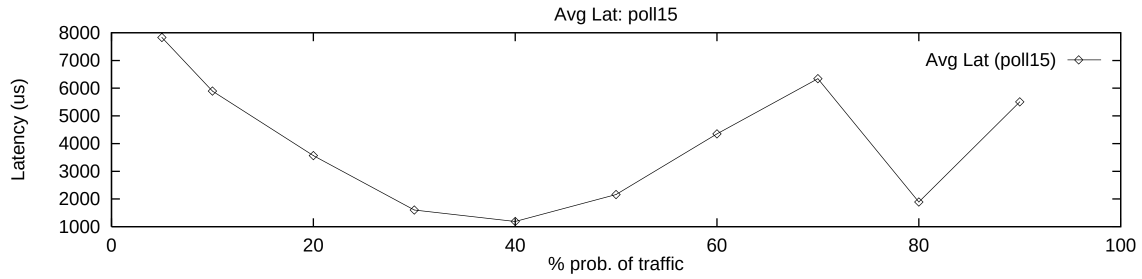
4.2.3 HIP, $T_P^{MAX} = 10$



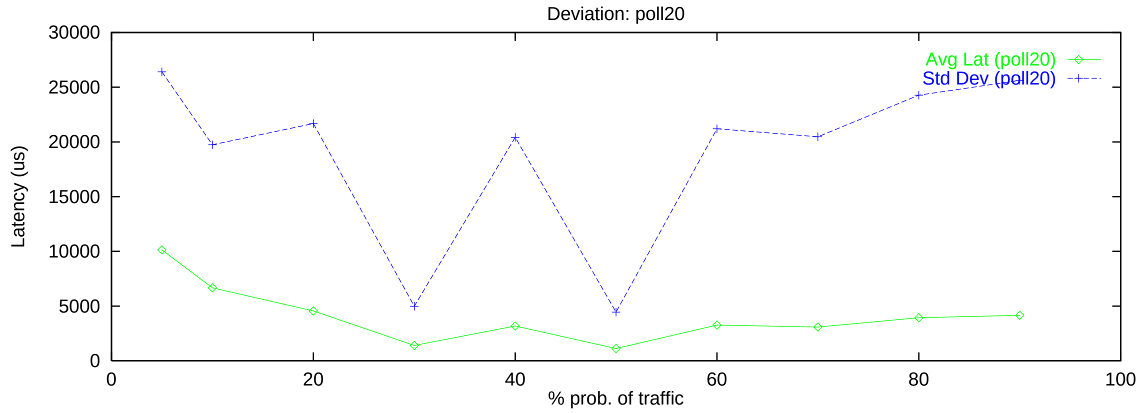


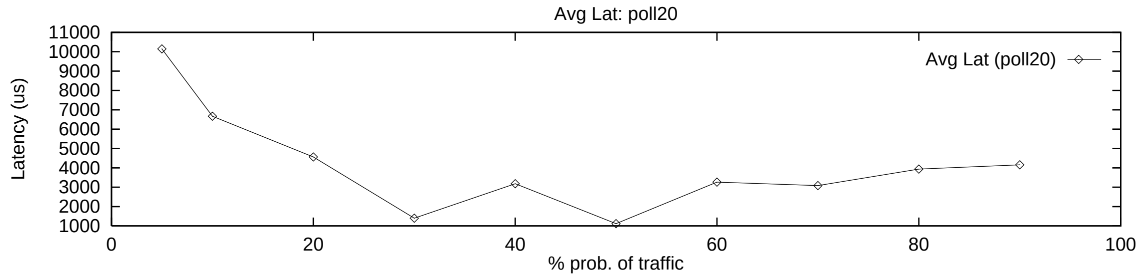
4.2.4 HIP, $T_P^{MAX} = 15$



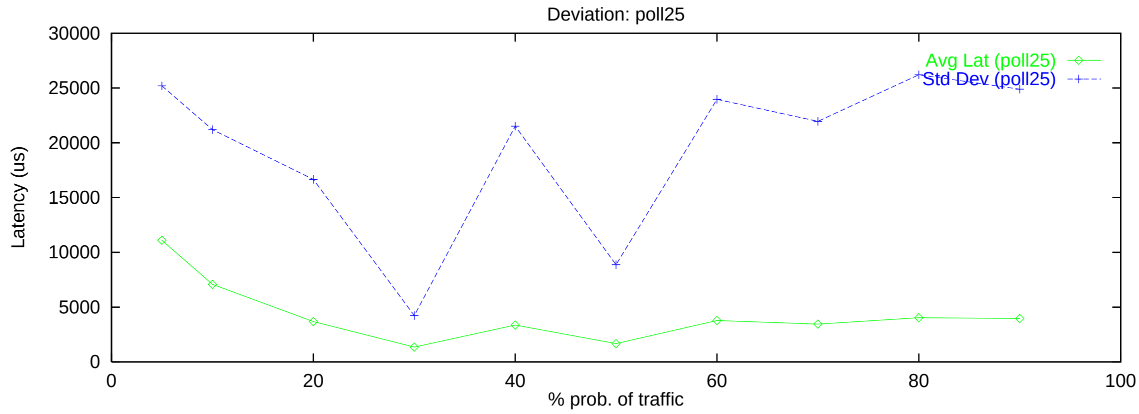


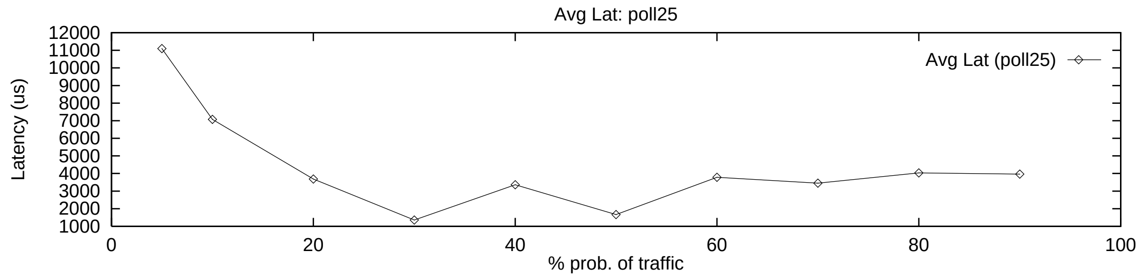
4.2.5 HIP, $T_P^{MAX} = 20$



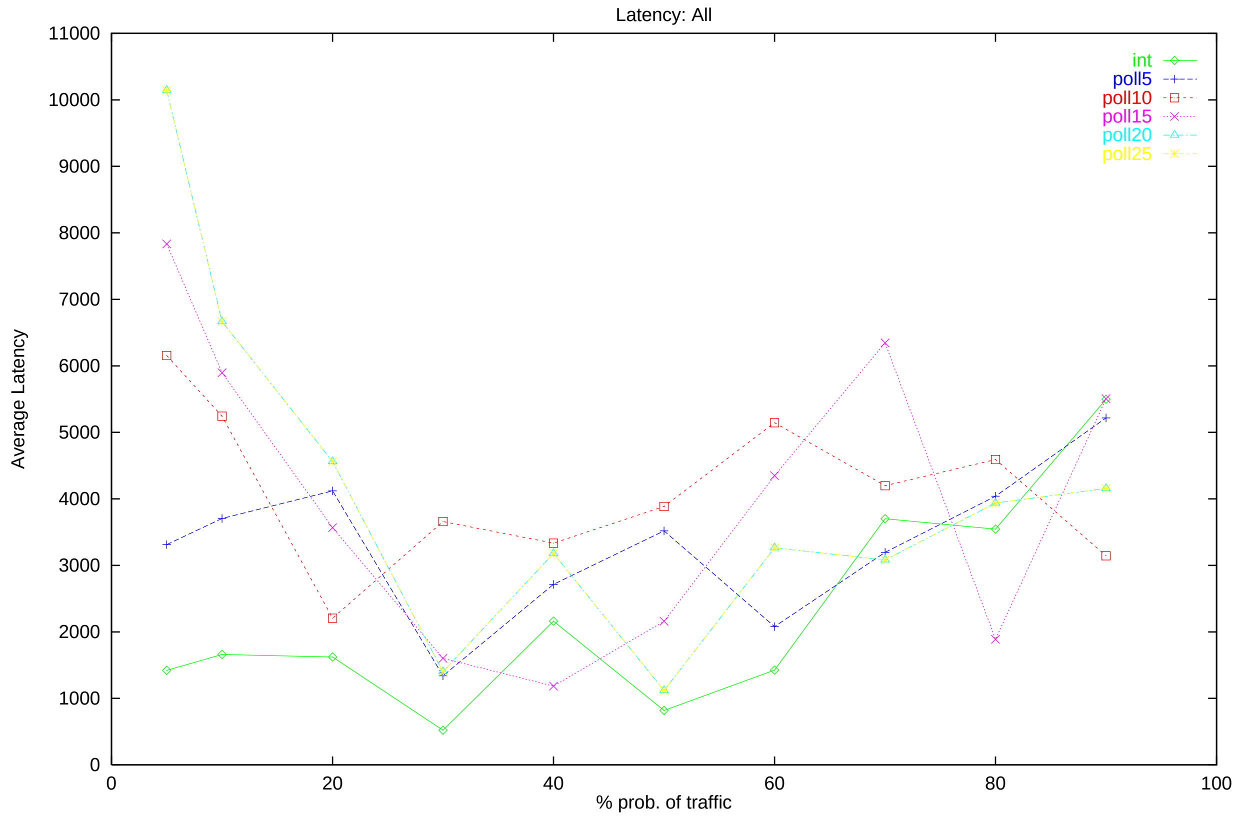


4.2.6 HIP, $T_P^{MAX} = 25$





4.2.7 Average Latency

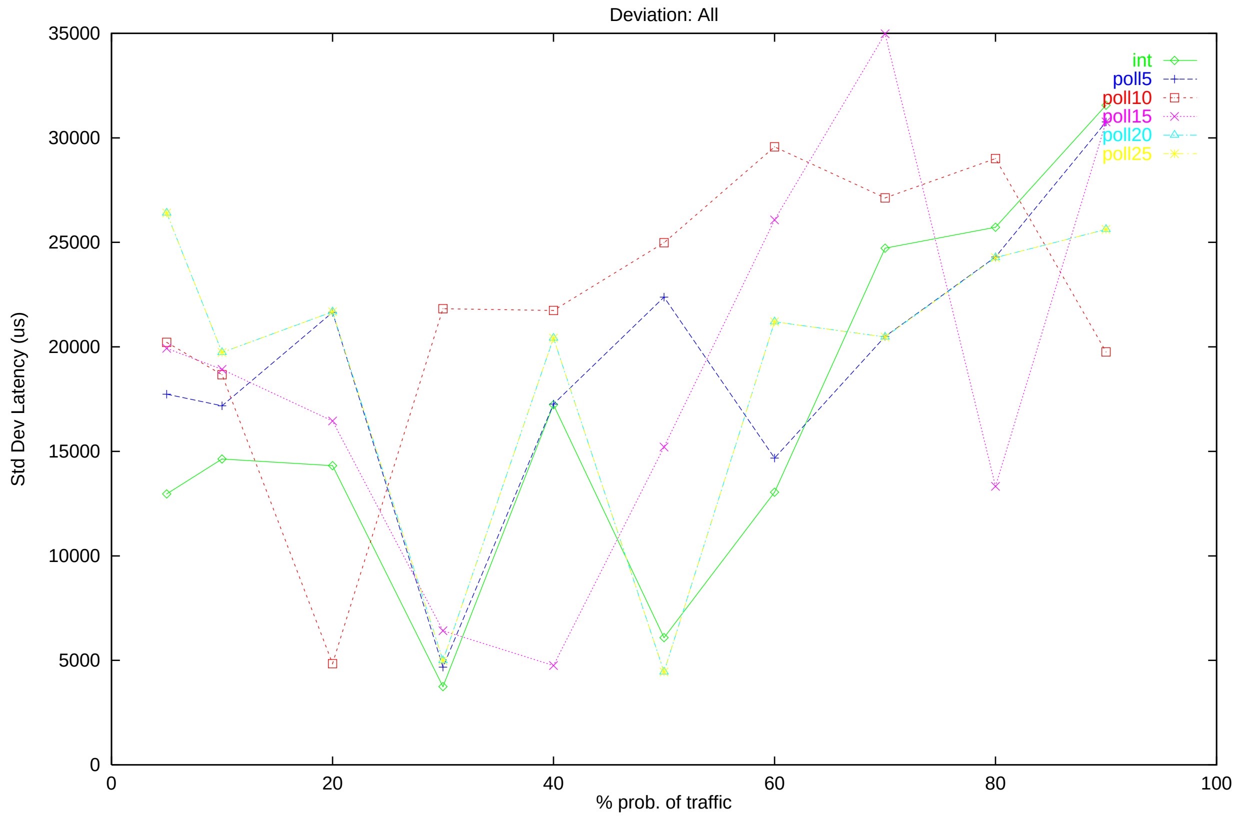


This graph illustrates several important points. First, for large inter-arrival times (low 200us probability of sending — the left side of the graph), interrupt mode has, on average, lower latency than the polling modes. This happens because, in interrupt mode, immediately after a packet arrives, the CPU is interrupted. In polling mode, the NIC must wait to be polled at the next interval; this increases the average latency.

Second, for small inter-arrival times (low 200us probability of sending — the right side of the graph), the polling modes have less latency than interrupt mode. This occurs primarily because the packet arrival times are more regular (only one out of 10 is skipped, on average) so polling mode can accurately predict the next arrival time, and process the packet immediately when it arrives. The overhead for polling mode is much more pronounced with high packet rates (§4.1).

One reason why, on average, they all increase as the traffic increases is because the ethernet segment is more busy and thus more collisions occur.

4.2.8 Latency Deviation



The average latencies are somewhat misleading: most packets have small latencies compared to the average, but there are packets which have perhaps 10–20x the latency of "most packets." Besides increasing the average latencies in the previous graph, we can see this effect in the graph above, which shows the standard deviation of packet latencies as a function of traffic model for each of the modes tested. On the whole, the trend is that the standard deviation increases as the traffic increases. One reason this happens is because there are many more collisions on the ethernet segment, which causes some packets to be delayed for much longer than those which are not delayed.

4.3 Transfer Rates

To measure throughput, we used FTP to copy a large file from another machine to the machine running HIP. To reduce non-network-related effects, we made sure the file was buffered in RAM before copying the file.

4.3.1 Interrupt Mode

Rates of 830.66, 801.19, 816.50, 803.97 KB/s were achieved, giving an average of 813.0 KB/s, and a deviation of 13.3.

4.3.2 Polling Mode

Using a maximum polling period of 10ms, rates of 889.8, 890.0, 877.0, and 879.0 KB/s were achieved, giving an average of 884.0 KB/s and deviation of 6.9. This peak transfer rate was achieved with $T_P^{MAX} = 10$ms.

Thus polling mode was about 8.7% better than interrupt mode.

5 Summary

5.1 Conclusions

In answer to the original purpose, we showed that

1. HIP is implementable in a modern UNIX kernel.

2. HIP can increase network performance.

For short inter-arrival times, the overhead of HIP is much — up to 20% — less than interrupt-only mode (§4.1). As expected, for most of the inter-send times tested, latency increased with HIP compared to interrupt-only. When traffic was very regular and very often, however, HIP had less latency because it could accurately predict the next arrival time *and* there was less overhead which increases latency (§4.2.7). Finally, for streaming data — in our case FTP — higher throughput is possible using HIP (§4.3).

5.2 Future Work

- Implement a `/proc` file-system interface for tuning parameters without reloading the network driver module, and to more easily get performance information.

- Tune α, β, and γ.

- Tune T_P^{MAX}.

- Evaluate switching algorithm.

- Use 100BaseT ethernet. Our study saw effects of ethernet saturation (§4.2.7, §4.2.8), namely the effect on packet latency. Using a faster network interface would remove this effect.

- Measure latencies on a finer level. Our study measured only end-to-end packet latency. A better measure would be to measure the delay from the time the packet arrives at the NIC to the time is arrives in the application. This requires some sort of accurate time-stamping to be done by the NIC, which the Vortex card does not do reliably.

A drivers/net/3c59x.c

```
/*
Written 1996-1998 by Donald Becker.

     This software may be used and distributed according to the terms
     of the GNU Public License, incorporated herein by reference.

     This driver is for the 3Com "Vortex" and "Boomerang" series ethercards.
     Members of the series include Fast EtherLink 3c590/3c592/3c595/3c597
     and the EtherLink XL 3c900 and 3c905 cards.

     The author may be reached as becker@CESDIS.gsfc.nasa.gov, or C/O
     Center of Excellence in Space Data and Information Sciences
        Code 930.5, Goddard Space Flight Center, Greenbelt MD 20771
*/

static char *version =
"3c59x.c:v0.99E 5/12/98 Donald Becker http://cesdis.gsfc.nasa.gov/linux/drivers/vortex.html\n";

#define HIP_DEBUG 0


/* "Knobs" that adjust features and parameters. */
/* Set the copy breakpoint for the copy-only-tiny-frames scheme.
   Setting to > 1512 effectively disables this feature. */
static const int rx_copybreak = 200;
/* Allow setting MTU to a larger size, bypassing the normal ethernet setup. */
static const int mtu = 1500;
/* Maximum events (Rx packets, etc.) to handle at each interrupt. */
static int max_interrupt_work = 20; /* was 20 */

#define VORTEX_DEBUG 1
/* Put out somewhat more debugging messages. (0: no msg, 1 minimal .. 6). */
#ifdef VORTEX_DEBUG
static int vortex_debug = VORTEX_DEBUG;
#else
static int vortex_debug = 1;
#endif

/* Some values here only for performance evaluation and path-coverage
   debugging. */
static int rx_nocopy = 0, rx_copy = 0, queued_packet = 0, rx_csumhits;

/* Enable the automatic media selection code -- usually set. */
#define AUTOMEDIA 1

/* Allow the use of fragment bus master transfers instead of only
   programmed-I/O for Vortex cards.  Full-bus-master transfers are always
   enabled by default on Boomerang cards.  If VORTEX_BUS_MASTER is defined,
   the feature may be turned on using 'options'. */
#if YOU_ARE_BRAVER_THAN_ME
#define VORTEX_BUS_MASTER
#endif

/* A few values that may be tweaked. */
/* Time in jiffies before concluding the transmitter is hung. */
#define TX_TIMEOUT  ((400*HZ)/1000)

/* Keep the ring sizes a power of two for efficiency. */
#define TX_RING_SIZE     16
#define RX_RING_SIZE     32
#define PKT_BUF_SZ          1536               /* Size of each temporary Rx buffer.*/

#include <linux/config.h>
#ifdef MODULE
#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif
#include <linux/module.h>
#include <linux/version.h>
#else
#define MOD_INC_USE_COUNT
#define MOD_DEC_USE_COUNT
#endif

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/ptrace.h>
#include <linux/errno.h>
#include <linux/in.h>
#include <linux/ioport.h>
#include <linux/malloc.h>
#include <linux/interrupt.h>
#include <linux/pci.h>
#include <linux/bios32.h>
#include <linux/timer.h>
#include <asm/irq.h>              /* For NR_IRQS only. */
#include <asm/bitops.h>
```

```
#include <asm/io.h>

#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/skbuff.h>

/* Kernel compatibility defines, common to David Hind's PCMCIA package.
   This is only in the support-all-kernels source code. */
#ifndef LINUX_VERSION_CODE
#include <linux/version.h>          /* Redundant above, here for easy clean-up. */
#endif
#if LINUX_VERSION_CODE < 0x10300
#define RUN_AT(x) (x)               /* What to put in timer->expires.  */
#define DEV_ALLOC_SKB(len) alloc_skb(len, GFP_ATOMIC)
#if defined(__alpha)
#error "The Alpha architecture is only support with kernel version 2.0."
#endif
#define virt_to_bus(addr)  ((unsigned long)addr)
#define bus_to_virt(addr) ((void*)addr)
#define NR_IRQS 16
#else  /* 1.3.0 and later */
#define RUN_AT(x) (jiffies + (x))
#define DEV_ALLOC_SKB(len) dev_alloc_skb(len)
#endif
#if LINUX_VERSION_CODE < 0x20159
#define DEV_FREE_SKB(skb) dev_kfree_skb (skb, FREE_WRITE);
#else  /* Grrr, unneeded incompatible change. */
#define DEV_FREE_SKB(skb) dev_kfree_skb(skb);
#endif

#ifdef SA_SHIRQ
#define FREE_IRQ(irqnum, dev) free_irq(irqnum, dev)
#define REQUEST_IRQ(i,h,f,n, instance) request_irq(i,h,f,n, instance)
#define IRQ(irq, dev_id, pt_regs) (irq, dev_id, pt_regs)
#else
#define FREE_IRQ(irqnum, dev) free_irq(irqnum)
#define REQUEST_IRQ(i,h,f,n, instance) request_irq(i,h,f,n)
#define IRQ(irq, dev_id, pt_regs) (irq, pt_regs)
#endif

#if (LINUX_VERSION_CODE >= 0x10344)
#define NEW_MULTICAST
#include <linux/delay.h>
#else
#define udelay(microsec)     do { int _i = 4*microsec; while (--_i > 0) { __SLOW_DOWN_IO; }} while (0)
#endif

#if LINUX_VERSION_CODE < 0x20138
#define test_and_set_bit(val, addr) set_bit(val, addr)
#endif
#if defined(MODULE) && (LINUX_VERSION_CODE >= 0x20115)
MODULE_AUTHOR("Donald Becker <becker@cesdis.gsfc.nasa.gov>");
MODULE_DESCRIPTION("3Com 3c590/3c900 series Vortex/Boomerang driver");
MODULE_PARM(debug, "i");
MODULE_PARM(options, "1-" __MODULE_STRING(8) "i");
MODULE_PARM(full_duplex, "1-" __MODULE_STRING(8) "i");
MODULE_PARM(rx_copybreak, "i");
MODULE_PARM(max_interrupt_work, "i");
MODULE_PARM(compaq_ioaddr, "i");
MODULE_PARM(compaq_irq, "i");
MODULE_PARM(compaq_prod_id, "i");
#endif

/* Operational parameter that usually are not changed. */

/* The Vortex size is twice that of the original EtherLinkIII series: the
   runtime register window, window 1, is now always mapped in.
   The Boomerang size is twice as large as the Vortex -- it has additional
   bus master control registers. */
#define VORTEX_TOTAL_SIZE 0x20
#define BOOMERANG_TOTAL_SIZE 0x40

#ifdef HAVE_DEVLIST
struct netdev_entry tc59x_drv =
{"Vortex", vortex_pci_probe, VORTEX_TOTAL_SIZE, NULL};
#endif

/* Set iff a MII transceiver on any interface requires mdio preamble.
   This only set with the original DP83840 on older 3c905 boards, so the extra
   code size of a per-interface flag is not worthwhile. */
static char mii_preamble_required = 0;

/* Caution!  These entries must be consistent. */
static const int product_ids[] = {
     0x5900, 0x5920, 0x5970, 0x5950, 0x5951, 0x5952, 0x9000, 0x9001,
     0x9050, 0x9051, 0x9055,     0x5057, 0 };
static const char *product_names[] = {
     "3c590 Vortex 10Mbps",
     "3c592 EISA 10mbps Demon/Vortex",
     "3c597 EISA Fast Demon/Vortex",
     "3c595 Vortex 100baseTX",
     "3c595 Vortex 100baseT4",
```

```
    "3c595 Vortex 100base-MII",
    "3c900 Boomerang 10baseT",
    "3c900 Boomerang 10Mbps/Combo",
    "3c905 Boomerang 100baseTx",
    "3c905 Boomerang 100baseT4",
    "3c905B Cyclone 100baseTx",
    "3c575",                              /* Cardbus Boomerang */
};


/*
                    Theory of Operation

I. Board Compatibility

This device driver is designed for the 3Com FastEtherLink and FastEtherLink
XL, 3Com's PCI to 10/100baseT adapters.  It also works with the 10Mbs
versions of the FastEtherLink cards.  The supported product IDs are
  3c590, 3c592, 3c595, 3c597, 3c900, 3c905

The ISA 3c515 is supported with a separate driver, 3c515.c, included with
the kernel source or available from
    cesdis.gsfc.nasa.gov:/pub/linux/drivers/3c515.html

II. Board-specific settings

PCI bus devices are configured by the system at boot time, so no jumpers
need to be set on the board.  The system BIOS should be set to assign the
PCI INTA signal to an otherwise unused system IRQ line.  While it's
physically possible to shared PCI interrupt lines, the 1.2.0 kernel doesn't
support it.

III. Driver operation

The 3c59x series use an interface that's very similar to the previous 3c5x9
series.  The primary interface is two programmed-I/O FIFOs, with an
alternate single-contiguous-region bus-master transfer (see next).

The 3c900 "Boomerang" series uses a full-bus-master interface with separate
lists of transmit and receive descriptors, similar to the AMD LANCE/PCnet,
DEC Tulip and Intel Speedo3.  The first chip version retains a compatible
programmed-I/O interface that will be removed in the 'B' and subsequent
revisions.

One extension that is advertised in a very large font is that the adapters
are capable of being bus masters.  On the Vortex chip this capability was
only for a single contiguous region making it far less useful than the full
bus master capability.  There is a significant performance impact of taking
an extra interrupt or polling for the completion of each transfer, as well
as difficulty sharing the single transfer engine between the transmit and
receive threads.  Using DMA transfers is a win only with large blocks or
with the flawed versions of the Intel Orion motherboard PCI controller.

The Boomerang chip's full-bus-master interface is useful, and has the
currently-unused advantages over other similar chips that queued transmit
packets may be reordered and receive buffer groups are associated with a
single frame.

With full-bus-master support, this driver uses a "RX_COPYBREAK" scheme.
Tather than a fixed intermed  te receive buffer, this scheme allocates
full-sized skbuffs as receive buffers.  The value RX_COPYBREAK is used as
the copying breakpoint: it is chosen to trade-off the memory wasted by
passing the full-sized skbuff to the queue layer for all frames vs. the
copying cost of copying a frame to a correctly-sized skbuff.


IIIC. Synchronization
The driver runs as two independent, single-threaded flows of control.  One
is the send-packet routine, which enforces single-threaded use by the
dev->tbusy flag.  The other thread is the interrupt handler, which is single
threaded by the hardware and other software.

IV. Notes

Thanks to Cameron Spitzer and Terry Murphy of 3Com for providing development
3c590, 3c595, and 3c900 boards.
The name "Vortex" is the internal 3Com project name for the PCI ASIC, and
the EISA version is called "Demon".  According to Terry these names come
from rides at the local amusement park.

The new chips support both ethernet (1.5K) and FDDI (4.5K) packet sizes!
This driver only supports ethernet packets because of the skbuff allocation
limit of 4K.
*/

#define TCOM_VENDOR_ID     0x10B7          /* 3Com's manufacturer's ID. */

/* Operational definitions.
   These are not used by other compilation units and thus are not
   exported in a ".h" file.
```

```
   First the windows.  There are eight register windows, with the command
   and status registers available in each.
   */
#define EL3WINDOW(win_num) outw(SelectWindow + (win_num), ioaddr + EL3_CMD)
#define EL3_CMD 0x0e
#define EL3_STATUS 0x0e

/* The top five bits written to EL3_CMD are a command, the lower
   11 bits are the parameter, if applicable.
   Note that 11 parameters bits was fine for ethernet, but the new chip
   can handle FDDI length frames (~4500 octets) and now parameters count
   32-bit 'Dwords' rather than octets. */

enum vortex_cmd {
     TotalReset = 0<<11, SelectWindow = 1<<11, StartCoax = 2<<11,
     RxDisable = 3<<11, RxEnable = 4<<11, RxReset = 5<<11,
     UpStall = 6<<11, UpUnstall = (6<<11)+1,
     DownStall = (6<<11)+2, DownUnstall = (6<<11)+3,
     RxDiscard = 8<<11, TxEnable = 9<<11, TxDisable = 10<<11, TxReset = 11<<11,
     FakeIntr = 12<<11, AckIntr = 13<<11, SetIntrEnb = 14<<11,
     SetStatusEnb = 15<<11, SetRxFilter = 16<<11, SetRxThreshold = 17<<11,
     SetTxThreshold = 18<<11, SetTxStart = 19<<11,
     StartDMAUp = 20<<11, StartDMADown = (20<<11)+1, StatsEnable = 21<<11,
     StatsDisable = 22<<11, StopCoax = 23<<11,};

/* The SetRxFilter command accepts the following classes: */
enum RxFilter {
     RxStation = 1, RxMulticast = 2, RxBroadcast = 4, RxProm = 8 };

/* Bits in the general status register. */
enum vortex_status {
     IntLatch = 0x0001, HostError = 0x0002, TxComplete = 0x0004,
     TxAvailable = 0x0008, RxComplete = 0x0010, RxEarly = 0x0020,
     IntReq = 0x0040, StatsFull = 0x0080,
     DMADone = 1<<8, DownComplete = 1<<9, UpComplete = 1<<10,
     DMAInProgress = 1<<11,               /* DMA controller is still busy.*/
     CmdInProgress = 1<<12,               /* EL3_CMD is still busy.*/
};

/* Register window 1 offsets, the window used in normal operation.
   On the Vortex this window is always mapped at offsets 0x10-0x1f. */
enum Window1 {
     TX_FIFO = 0x10,  RX_FIFO = 0x10,  RxErrors = 0x14,
     RxStatus = 0x18,  Timer=0x1A, TxStatus = 0x1B,
     TxFree = 0x1C, /* Remaining free bytes in Tx buffer. */
};
enum Window0 {
     Wn0EepromCmd = 10,          /* Window 0: EEPROM command register. */
     Wn0EepromData = 12,          /* Window 0: EEPROM results register. */
     IntrStatus=0x0E,          /* Valid in all windows. */
};
enum Win0_EEPROM_bits {
     EEPROM_Read = 0x80, EEPROM_WRITE = 0x40, EEPROM_ERASE = 0xC0,
     EEPROM_EWENB = 0x30,          /* Enable erasing/writing for 10 msec. */
     EEPROM_EWDIS = 0x00,          /* Disable EWENB before 10 msec timeout. */
};
/* EEPROM locations. */
enum eeprom_offset {
     PhysAddr01=0, PhysAddr23=1, PhysAddr45=2, ModelID=3,
     EtherLink3ID=7, IFXcvrIO=8, IRQLine=9,
     NodeAddr01=10, NodeAddr23=11, NodeAddr45=12,
     DriverTune=13, Checksum=15};

enum Window3 {               /* Window 3: MAC/config bits. */
     Wn3_Config=0, Wn3_MAC_Ctrl=6, Wn3_Options=8,
};
union wn3_config {
     int i;
     struct w3_config_fields {
          unsigned int ram_size:3, ram_width:1, ram_speed:2, rom_size:2;
          int pad8:8;
          unsigned int ram_split:2, pad18:2, xcvr:4, autoselect:1;
          int pad24:7;
     } u;
};

enum Window4 {          /* Window 4: Xcvr/media bits. */
     Wn4_FIFODiag = 4, Wn4_NetDiag = 6, Wn4_PhysicalMgmt=8, Wn4_Media = 10,
};
enum Win4_Media_bits {
     Media_SQE = 0x0008,          /* Enable SQE error counting for AUI. */
     Media_10TP = 0x00C0,     /* Enable link beat and jabber for 10baseT. */
     Media_Lnk = 0x0080,          /* Enable just link beat for 100TX/100FX. */
     Media_LnkBeat = 0x0800,
};
enum Window7 {                          /* Window 7: Bus Master control. */
     Wn7_MasterAddr = 0, Wn7_MasterLen = 6, Wn7_MasterStatus = 12,
};
/* Boomerang bus master control registers. */
enum MasterCtrl {
     PktStatus = 0x20, DownListPtr = 0x24, FragAddr = 0x28, FragLen = 0x2c,
     TxFreeThreshold = 0x2f, UpPktStatus = 0x30, UpListPtr = 0x38,
```

```
};

/* The Rx and Tx descriptor lists.
   Caution Alpha hackers: these types are 32 bits!  Note also the 8 byte
   alignment contraint on tx_ring[] and rx_ring[]. */
#define LAST_FRAG  0x80000000               /* Last Addr/Len pair in descriptor. */
struct boom_rx_desc {
     u32 next;                         /* Last entry points to 0.   */
     s32 status;
     u32 addr;                         /* Up to 63 addr/len pairs possible. */
     s32 length;                         /* Set LAST_FRAG to indicate last pair. */
};
/* Values for the Rx status entry. */
enum rx_desc_status {
     RxDComplete=0x00008000, RxDError=0x4000,
     /* See boomerang_rx() for actual error bits */
     IPChksumErr=1<<25, TCPChksumErr=1<<26, UDPChksumErr=1<<27,
     IPChksumValid=1<<29, TCPChksumValid=1<<30, UDPChksumValid=1<<31,
};

struct boom_tx_desc {
     u32 next;                         /* Last entry points to 0.   */
     s32 status;                         /* bits 0:12 length, others see below.  */
     u32 addr;
     s32 length;
};

/* Values for the Tx status entry. */
enum tx_desc_status {
     CRCDisable=0x2000, TxDComplete=0x8000,
     AddIPChksum=0x02000000, AddTCPChksum=0x04000000, AddUDPChksum=0x08000000,
     TxIntrUploaded=0x80000000,          /* IRQ when in FIFO, but maybe not sent. */
};

/* Chip features we care about in vp->capabilities, read from the EEPROM. */
enum ChipCaps { CapBusMaster=0x20 };

struct vortex_private {
     char devname[8];               /* "ethN" string, also for kernel debug. */
     const char *product_name;
     struct device *next_module;
     /* The Rx and Tx rings are here to keep them quad-word-aligned. */
     struct boom_rx_desc rx_ring[RX_RING_SIZE];
     struct boom_tx_desc tx_ring[TX_RING_SIZE];
     /* The addresses of transmit- and receive-in-place skbuffs. */
     struct sk_buff* rx_skbuff[RX_RING_SIZE];
     struct sk_buff* tx_skbuff[TX_RING_SIZE];
     unsigned int cur_rx, cur_tx;          /* The next free ring entry */
     unsigned int dirty_rx, dirty_tx;     /* The ring entries to be free()ed. */
     struct enet_statistics stats;
     struct sk_buff *tx_skb;          /* Packet being eaten by bus master ctrl.  */

     /* PCI configuration space information. */
     u8 pci_bus, pci_dev_fn;          /* PCI bus location, for power management. */
     u16 pci_device_id;

     /* The remainder are related to chip state, mostly media selection. */
     int in_interrupt;
     struct timer_list timer;     /* Media selection timer. */
     int options;                    /* User-settable misc. driver options. */
     unsigned int
       media_override:3,                /* Passed-in media type. */
       default_media:3,                    /* Read from the EEPROM/Wn3_Config. */
       full_duplex:1, autoselect:1,
       bus_master:1,                    /* Vortex can only do a fragment bus-m. */
       full_bus_master_tx:1, full_bus_master_rx:2, /* Boomerang  */
       hw_csums:1,                    /* Has hardware checksums. */
      tx_full:1;
     u16 status_enable;
     u16 available_media;                    /* From Wn3_Options. */
     u16 capabilities, info1, info2;          /* Various, from EEPROM. */
     u16 advertising;                          /* NWay media advertisement */
     unsigned char phys[2];                    /* MII device addresses. */
};

/* The action to take with a media selection timer tick.
   Note that we deviate from the 3Com order by checking 10base2 before AUI.
 */
enum xcvr_types {
     XCVR_10baseT=0, XCVR_AUI, XCVR_10baseTOnly, XCVR_10base2, XCVR_100baseTx,
     XCVR_100baseFx, XCVR_MII=6, XCVR_NWAY=8, XCVR_ExtMII=9, XCVR_Default=10,
};

static struct media_table {
     char *name;
     unsigned int media_bits:16,          /* Bits to set in Wn4_Media register. */
          mask:8,                    /* The transceiver-present bit in Wn3_Config.*/
          next:8;                    /* The media type to try next. */
     int wait;               /* Time before we check media status. */
} media_tbl[] = {
  {     "10baseT",   Media_10TP,0x08, XCVR_10base2, (14*HZ)/10},
  { "10Mbs AUI", Media_SQE, 0x20, XCVR_Default, (1*HZ)/10},
```

```
  { "undefined", 0,               0x80, XCVR_10baseT, 10000},
  { "10base2",   0,               0x10, XCVR_AUI,          (1*HZ)/10},
  { "100baseTX", Media_Lnk, 0x02, XCVR_100baseFx, (14*HZ)/10},
  { "100baseFX", Media_Lnk, 0x04, XCVR_MII,          (14*HZ)/10},
  { "MII",           0,               0x41, XCVR_10baseT, 3*HZ },
  { "undefined", 0,               0x01, XCVR_10baseT, 10000},
  { "Autonegotiate", 0,          0x41, XCVR_10baseT, 3*HZ},
  { "MII-External",      0,          0x41, XCVR_10baseT, 3*HZ },
  { "Default",      0,               0xFF, XCVR_10baseT, 10000},
};

static int vortex_scan(struct device *dev);
static struct device *vortex_found_device(struct device *dev, int ioaddr,
                                                   int irq, int device_id,
                                                   int options, int card_idx);
static int vortex_probe1(struct device *dev);
static int vortex_open(struct device *dev);
static void mdio_sync(int ioaddr, int bits);
static int mdio_read(int ioaddr, int phy_id, int location);
#ifdef HAVE_PRIVATE_IOCTL
static void mdio_write(int ioaddr, int phy_id, int location, int value);
#endif
static void vortex_timer(unsigned long arg);
static int vortex_start_xmit(struct sk_buff *skb, struct device *dev);
static int boomerang_start_xmit(struct sk_buff *skb, struct device *dev);
static int vortex_rx(struct device *dev);
static int boomerang_rx(struct device *dev);
static void vortex_interrupt IRQ(int irq, void *dev_id, struct pt_regs *regs);
static int vortex_close(struct device *dev);
static void update_stats(int addr, struct device *dev);
static struct enet_statistics *vortex_get_stats(struct device *dev);
static void set_rx_mode(struct device *dev);
#ifdef HAVE_PRIVATE_IOCTL
static int vortex_ioctl(struct device *dev, struct ifreq *rq, int cmd);
#endif
#ifndef NEW_MULTICAST
static void set_multicast_list(struct device *dev, int num_addrs, void *addrs);
#endif

/* ZAK */

/* CONSTANT - tunable */

char *z_flight_recorder;
int z_flight_pos;
int z_flight_length;

struct timeval z_tv;

unsigned int z_max_unsucc_polls = 10;

/* weight of the current time w.r.t previous average */
float z_alpha = 0.3;

/* weight of the current variance wrt previous average variance */
float z_beta = 0.5;

/* prediction threshold - smaller allows more predictable */
float z_gamma = 5.0; /* 0 .. 20 */

/* min and max polling period - in 1ms units */
unsigned int z_min_poll_interval = 1;
unsigned int z_max_poll_interval = 25;

/* phi - the slack */
float z_slack = 2.5;


/* machine variables */

unsigned long z_total_int_latency = 0;
unsigned long z_total_poll_latency = 0;
unsigned long z_total_num_interrupts = 0;
unsigned long z_total_num_polls = 0;
unsigned long z_total_num_unsucc_polls = 0;

/* how often interrupt polling happens */
unsigned int z_polling_interval = 2;


/* time in jiffies when we go the last net interrupt */
volatile unsigned int z_last_net_int = 0;

/* current number of unsuccessful polls */
volatile unsigned int z_unsucc_polls = 0;


/* average interarrival time */
float z_avg_ia = 5.0;

/* average interarrival variance */
float z_avg_iv = 5.0;
```

```
/* the timer which keeps track of our routine */
struct timer_list z_timer;

/* our copy of the ptr to the dev struct */
struct device *z_dev = NULL;

/* for printing things out a modulo intervals */
unsigned int z_count = 0;

/* state machine variable */
volatile int z_polling_mode = 0;


static void write_fr(char *s)
{
     int len = strlen(s);
     if (len > z_flight_length - z_flight_pos)
          z_flight_pos = 0;
     memcpy(z_flight_pos + z_flight_recorder, s, len);
}

static void my_timer_function(unsigned long __data)
     {
    /* unsigned long my_jiffies = __data; */

     int case1, case2, case3;
     unsigned int comp_value;
     struct vortex_private *vp;

     /* printk("In my timer function jiffies - myjiffies = %lu\n",
        jiffies-my_jiffies); */
     init_timer(&z_timer);

     z_timer.data = (unsigned long)jiffies;
     z_timer.function = my_timer_function;


     if (z_polling_mode)
          {
          float sqrt_sigma = z_avg_iv/4.0;
          /* iterate a few times to get a better value */
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
          if (z_avg_iv < 0.0001)
               sqrt_sigma = 0.0;

          /* now this should be pretty good */
     /* SQRT */
/*          printk("Sqrt Check %lu = sqrt(%lu)\n", (unsigned long)(sqrt_sigma*100.0),
               (unsigned long)(z_avg_iv*100));
*/


          if ( (int) ((z_avg_ia/1000.0 + z_slack*sqrt_sigma/1000.0)) != z_polling_interval)
          {
               z_polling_interval = (int) (z_avg_ia/1000.0 + z_slack*sqrt_sigma/1000.0);
               /* printk("polling interval adjusted to %d \n", z_polling_interval); */
          }

          if (z_polling_interval < z_min_poll_interval ) z_polling_interval = z_min_poll_interval;
          else if (z_polling_interval > z_max_poll_interval ) z_polling_interval = z_max_poll_interval;


          cli();
          if (z_dev)
               {
               vortex_interrupt IRQ(z_dev->irq, z_dev, 0);
               }
          sti();

          case1 = case2 = case3 = 0;
          comp_value = (unsigned int)(z_avg_ia/1000.0); /* Convert from us to ms */
          /* test for switching to int mode */
          if (
               (case1 =  ((sqrt_sigma / z_avg_ia ) > z_gamma)  )
               ||
               (case2 = ( comp_value >= z_max_poll_interval ))
               ||
               (case3 = ( z_unsucc_polls >= z_max_unsucc_polls ))
               )
               {
#if HIP_DEBUG==1
               printk("JIFFIES: %lu  : moving to interrupt mode z_avg_ia=%lu(us), sigma = %lu, unsucc_polls=%u CASE:[%d%d%d]\n",
                           jiffies, (unsigned long)(z_avg_ia),  (unsigned long)sqrt_sigma,
                          z_unsucc_polls  , case1, case2, case3 );
#endif
```

```
                if (z_unsucc_polls >= z_max_unsucc_polls)
                {
                     z_unsucc_polls = 0;
                     z_avg_iv = 5000.0;
                     z_avg_ia = 5000.0;
                }
                if (z_total_num_interrupts != 0 && z_total_num_polls != 0)
                {
#if HIP_DEBUG==1
                printk("STAT: Time:%lu, num_inter=%lu, num_poll=%lu, num_unsucc_poll=%lu, AIL=%lu, APL=%lu, AL=%lu %d\n",
                          jiffies, z_total_num_interrupts, z_total_num_polls,
                          z_total_num_unsucc_polls, z_total_int_latency*100/z_total_num_interrupts,
                          z_total_poll_latency*100/z_total_num_polls,
                          (z_total_poll_latency+z_total_int_latency)*100/
                               (z_total_num_polls + z_total_num_interrupts),
                               z_polling_interval);
#endif
                }

                z_polling_mode = 0;

                /* enable interrupts on the board */

                if (z_dev) outw(SetIntrEnb | IntLatch | TxAvailable | RxComplete | StatsFull
                                    | HostError | TxComplete | UpComplete | DownComplete,
                                    z_dev->base_addr + EL3_CMD);

                }
           else /* We should reschedule ourselves for the next interval */
                {
                z_timer.expires = jiffies + z_polling_interval;
                add_timer(&z_timer);
                }
           }
      else
           { /* in interrupt mode */
           }
      }


/* Unlike the other PCI cards the 59x cards don't need a large contiguous
   memory region, so making the driver a loadable module is feasible.

   Unfortunately maximizing the shared code between the integrated and
   module version of the driver results in a complicated set of initialization
   procedures.
   init_module() -- modules /  tc59x_init()  -- built-in
          The wrappers for vortex_scan()
   vortex_scan()              The common routine that scans for PCI and EISA cards
   vortex_found_device() Allocate a device structure when we find a card.
                          Different versions exist for modules and built-in.
   vortex_probe1()           Fill in the device structure -- this is separated
                          so that the modules code can put it in dev->init.
*/
/* This driver uses 'options' to pass the media type, full-duplex flag, etc. */
/* Note: this is the only limit on the number of cards supported!! */
static int options[8] = { -1, -1, -1, -1, -1, -1, -1, -1,};
static int full_duplex[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
/* A list of all installed Vortex devices, for removing the driver module. */
static struct device *root_vortex_dev = NULL;

#ifdef MODULE
/* Variables to work-around the Compaq PCI BIOS32 problem. */
static int compaq_ioaddr = 0, compaq_irq = 0, compaq_device_id = 0x5900;

static int debug = -1;

#ifdef CARDBUS

#include <pcmcia/driver_ops.h>

static dev_node_t *vortex_attach(dev_locator_t *loc)
{
     u16 dev_id;
     u32 io;
     u8 bus, devfn, irq;
     struct device *dev;

     if (loc->bus != LOC_PCI) return NULL;
     bus = loc->b.pci.bus; devfn = loc->b.pci.devfn;
     printk(KERN_INFO "vortex_attach(bus %d, function %d)\n", bus, devfn);
     pcibios_read_config_dword(bus, devfn, PCI_BASE_ADDRESS_0, &io);
     pcibios_read_config_byte(bus, devfn, PCI_INTERRUPT_LINE, &irq);
     pcibios_read_config_word(bus, devfn, PCI_DEVICE_ID, &dev_id);
     io &= ~3;
     dev = vortex_found_device(NULL, io, irq, dev_id, 0, -1);
     if (dev) {
          dev_node_t *node = kmalloc(sizeof(dev_node_t), GFP_KERNEL);
          strcpy(node->dev_name, dev->name);
          node->major = node->minor = 0;
          node->next = NULL;
```

```
          MOD_INC_USE_COUNT;
          return node;
     }
     return NULL;
}

static void vortex_detach(dev_node_t *node)
{
     struct device **devp, **next;
     printk(KERN_INFO "vortex_detach(%s)\n", node->dev_name);
     for (devp = &root_vortex_dev; *devp; devp = next) {
          next = &((struct vortex_private *)(*devp)->priv)->next_module;
          if (strcmp((*devp)->name, node->dev_name) == 0) break;
     }
     if (*devp) {
          struct device *dev = *devp;
          if (dev->flags & IFF_UP)
               vortex_close(dev);
          dev->flags &= ~(IFF_UP|IFF_RUNNING);
          unregister_netdev(dev);
          kfree(dev);
          *devp = *next;
          kfree(node);
          MOD_DEC_USE_COUNT;
     }
}

struct driver_operations vortex_ops = {
     "3c59x_cb", vortex_attach, NULL, NULL, vortex_detach
};

#endif  /* Cardbus support */


int
init_module(void)
{
     if (debug >= 0)
          vortex_debug = debug;

/*     if (vortex_debug)
          printk(version); */

     /* ZAK */
     z_flight_recorder = kmalloc(8192,  GFP_KERNEL);
     z_flight_pos = 0;
     z_flight_length = 8192;


     printk("3COM 3c590 HIP driver.  alpha = %lu, beta = %lu, gamma = %lu, Tpmax = %u\n",
               (unsigned long) (z_alpha*100.0),
               (unsigned long) (z_beta*100.0),
               (unsigned long)(z_gamma*100.0),
               z_max_poll_interval);
     printk("HIP flight recorder address = %x, length = %d\n", z_flight_recorder, z_flight_length);
     write_fr("HIP FLIGHT RECORDER\n");
     do_gettimeofday(&z_tv);

     root_vortex_dev = NULL;
#ifdef CARDBUS
     register_driver(&vortex_ops);
     return 0;
#else
     {
          int cards_found = vortex_scan(0);
          if (cards_found == 0)
               printk("No 3Com Vortex/Boomerang cards found.\n");
          return cards_found ? 0 : -ENODEV;
     }
#endif
}

#else
int tc59x_probe(struct device *dev)
{
     int cards_found = 0;

     cards_found = vortex_scan(dev);

     if (vortex_debug > 0  &&  cards_found)
          printk(version);

     return cards_found ? 0 : -ENODEV;
}
#endif  /* not MODULE */

static int vortex_scan(struct device *dev)
{
     int cards_found = 0;

     /* Allow an EISA-only driver. */
#if defined(CONFIG_PCI) || (defined(MODULE) && !defined(NO_PCI))
```

```
    /* Ideally we would detect all cards in slot order.  That would
       be best done a central PCI probe dispatch, which wouldn't work
       well with the current structure.  So instead we detect 3Com cards
       in slot order. */
    if (pcibios_present()) {
         static int pci_index = 0;
         unsigned char pci_bus, pci_device_fn;

         for (;pci_index < 0xff; pci_index++) {
              u8 pci_latency;
              u16 pci_command, new_command, vendor, device;
              int irq;
              long ioaddr;

              if (pcibios_find_class (PCI_CLASS_NETWORK_ETHERNET << 8,
                                      pci_index, &pci_bus, &pci_device_fn)
                   != PCIBIOS_SUCCESSFUL)
                   break;
              pcibios_read_config_word(pci_bus, pci_device_fn,
                                       PCI_VENDOR_ID, &vendor);
              pcibios_read_config_word(pci_bus, pci_device_fn,
                                       PCI_DEVICE_ID, &device);
              pcibios_read_config_word(pci_bus, pci_device_fn,
                                       PCI_COMMAND, &pci_command);
              {
#if LINUX_VERSION_CODE >= 0x20155
                   struct pci_dev *pdev = pci_find_slot(pci_bus, pci_device_fn);
                   ioaddr = pdev->base_address[0];
                   irq = pdev->irq;
#else
                   u32 pci_ioaddr;
                   u8 pci_irq_line;
                   pcibios_read_config_byte(pci_bus, pci_device_fn,
                                            PCI_INTERRUPT_LINE, &pci_irq_line);
                   pcibios_read_config_dword(pci_bus, pci_device_fn,
                                             PCI_BASE_ADDRESS_0, &pci_ioaddr);
                   ioaddr = pci_ioaddr;
                   irq = pci_irq_line;
#endif
              }
              /* Remove I/O space marker in bit 0. */
              ioaddr &= ~3;

              if (vendor != TCOM_VENDOR_ID)
                   continue;

              if (ioaddr == 0) {
                   printk(KERN_WARNING "  A 3Com network adapter has been found, "
                          "however it has not been assigned an I/O address.\n"
                          "  You may need to power-cycle the machine for this "
                          "device to work!\n");
                   continue;
              }

              if (check_region(ioaddr, VORTEX_TOTAL_SIZE))
                   continue;

              /* Activate the card. */
              new_command = pci_command | PCI_COMMAND_MASTER|PCI_COMMAND_IO;
              if (pci_command != new_command) {
                   printk(KERN_INFO "  The PCI BIOS has not enabled this"
                          " device!  Updating PCI command %4.4x->%4.4x.\n",
                          pci_command, new_command);
                   pcibios_write_config_word(pci_bus, pci_device_fn,
                                             PCI_COMMAND, new_command);
              }

              dev = vortex_found_device(dev, ioaddr, irq,
                                        device, dev && dev->mem_start
                                        ? dev->mem_start : options[cards_found],
                                        cards_found);

              if (dev) {
                   struct vortex_private *vp = (struct vortex_private *)dev->priv;
                   /* Get and check the latency values.  On the 3c590 series
                      the latency timer must be set to the maximum value to avoid
                      data corruption that occurs when the timer expires during
                      a transfer -- a bug in the Vortex chip only. */
                   u8 new_latency = (device&0xff00) == 0x5900 ? 248 : 32;
                   vp->pci_bus = pci_bus;
                   vp->pci_dev_fn = pci_device_fn;
                   vp->pci_device_id = device;

                   pcibios_read_config_byte(pci_bus, pci_device_fn,
                                            PCI_LATENCY_TIMER, &pci_latency);
                   if (pci_latency < new_latency) {
                        printk(KERN_INFO "%s: Overriding PCI latency"
                               " timer (CFLT) setting of %d, new value is %d.\n",
                               dev->name, pci_latency, new_latency);
                        pcibios_write_config_byte(pci_bus, pci_device_fn,
                                                  PCI_LATENCY_TIMER, new_latency);
                   }
```

```
                    dev = 0;
                    cards_found++;
               }
          }
     }
#endif /* NO_PCI */

     /* Now check all slots of the EISA bus. */
     if (EISA_bus) {
          static int ioaddr = 0x1000;
          for ( ; ioaddr < 0x9000; ioaddr += 0x1000) {
               int device_id;
               if (check_region(ioaddr, VORTEX_TOTAL_SIZE))
                    continue;
               /* Check the standard EISA ID register for an encoded '3Com'. */
               if (inw(ioaddr + 0xC80) != 0x6d50)
                    continue;
               /* Check for a product that we support, 3c59{2,7} any rev. */
               device_id = (inb(ioaddr + 0xC82)<<8) + inb(ioaddr + 0xC83);
               if ((device_id & 0xFF00) != 0x5900)
                    continue;
               vortex_found_device(dev, ioaddr, inw(ioaddr + 0xC88) >> 12,
                                   device_id,  dev && dev->mem_start
                                   ? dev->mem_start : options[cards_found],
                                   cards_found);
               dev = 0;
               cards_found++;
          }
     }

#ifdef MODULE
     /* Special code to work-around the Compaq PCI BIOS32 problem. */
     if (compaq_ioaddr) {
          vortex_found_device(dev, compaq_ioaddr, compaq_irq, compaq_device_id,
                              dev && dev->mem_start ? dev->mem_start
                              : options[cards_found], cards_found);
          cards_found++;
          dev = 0;
     }
#endif

     /* 3c515 cards are now supported by the 3c515.c driver. */

     return cards_found;
}

static struct device *
vortex_found_device(struct device *dev, int ioaddr, int irq,
                    int device_id, int option, int card_idx)
{
     struct vortex_private *vp;
     const char *product_name;
     int board_index = 0;

     for (board_index = 0; product_ids[board_index]; board_index++) {
          if (device_id == product_ids[board_index])
               break;
     }
     /* Handle products we don't recognize, but might still work with. */
     if (product_ids[board_index])
          product_name = product_names[board_index];
     else if ((device_id & 0xff00) == 0x5900)
          product_name = "3c590 Vortex";
     else if ((device_id & 0xfff0) == 0x9000)
          product_name = "3c900";
     else if ((device_id & 0xfff0) == 0x9050)
          product_name = "3c905";
     else {
          printk(KERN_WARNING "Unknown 3Com PCI ethernet adapter type %4.4x detected:"
                 " not configured.\n", device_id);
          return 0;
     }

#ifdef MODULE
     /* Allocate and fill new device structure. */
     {
          int dev_size = sizeof(struct device) +
               sizeof(struct vortex_private) + 15;          /* Pad for alignment */

          dev = (struct device *) kmalloc(dev_size, GFP_KERNEL);
          memset(dev, 0, dev_size);
     }
     /* Align the Rx and Tx ring entries.  */
     dev->priv = (void *)(((long)dev + sizeof(struct device) + 15) & ~15);
     vp = (struct vortex_private *)dev->priv;
     dev->name = vp->devname; /* An empty string. */
     dev->base_addr = ioaddr;
     dev->irq = irq;
     dev->init = vortex_probe1;
     vp->product_name = product_name;
     vp->options = option;
     if (card_idx >= 0) {
```

```
		if (full_duplex[card_idx] >= 0)
			vp->full_duplex = full_duplex[card_idx];
	} else
		vp->full_duplex = (option > 0 && (option & 0x10) ? 1 : 0);

	if (option > 0) {
		vp->media_override = ((option & 7) == XCVR_10baseTOnly) ?
			XCVR_10baseT  :  option & 7;
		vp->bus_master = (option & 16) ? 1 : 0;
	} else {
		vp->media_override = 7;
		vp->bus_master = 0;
	}
	ether_setup(dev);
	vp->next_module = root_vortex_dev;
	root_vortex_dev = dev;
	if (register_netdev(dev) != 0)
		return 0;
#else  /* not a MODULE */
	if (dev) {
		/* Caution: quad-word alignment required for rings! */
		dev->priv = kmalloc(sizeof (struct vortex_private), GFP_KERNEL);
		memset(dev->priv, 0, sizeof (struct vortex_private));
	}
	dev = init_etherdev(dev, sizeof(struct vortex_private));
	dev->base_addr = ioaddr;
	dev->irq = irq;
	dev->mtu = mtu;

	vp  = (struct vortex_private *)dev->priv;
	vp->product_name = product_name;
	vp->options = option;
	if (option >= 0) {
		vp->media_override = ((option & 7) == 2)  ?  0  :  option & 7;
		vp->full_duplex = (option & 8) ? 1 : 0;
		vp->bus_master = (option & 16) ? 1 : 0;
	} else {
		vp->media_override = 7;
		vp->full_duplex = 0;
		vp->bus_master = 0;
	}

	vortex_probe1(dev);
#endif /* MODULE */
	return dev;
}

static int vortex_probe1(struct device *dev)
{
	int ioaddr = dev->base_addr;
	struct vortex_private *vp = (struct vortex_private *)dev->priv;
	u16 *ether_addr = (u16 *)dev->dev_addr;
	unsigned int eeprom[0x40], checksum = 0;		/* EEPROM contents */
	int i;

	printk(KERN_INFO "%s: 3Com %s at %#3x,",
		   dev->name, vp->product_name, ioaddr);

	/* Read the station address from the EEPROM. */
	EL3WINDOW(0);
	for (i = 0; i < 0x40; i++) {
		int timer;
#ifdef CARDBUS
		outw(0x230 + i, ioaddr + Wn0EepromCmd);
#else
		outw(EEPROM_Read + i, ioaddr + Wn0EepromCmd);
#endif
		/* Pause for at least 162 us. for the read to take place. */
		for (timer = 10; timer >= 0; timer--) {
			udelay(162);
			if ((inw(ioaddr + Wn0EepromCmd) & 0x8000) == 0)
				break;
		}
		eeprom[i] = inw(ioaddr + Wn0EepromData);
	}
	for (i = 0; i < 0x18; i++)
		checksum ^= eeprom[i];
	checksum = (checksum ^ (checksum >> 8)) & 0xff;
	if (checksum != 0x00) {		/* Grrr, needless incompatible change 3Com. */
		while (i < 0x21)
			checksum ^= eeprom[i++];
		checksum = (checksum ^ (checksum >> 8)) & 0xff;
	}
	if (checksum != 0x00)
		printk(" ***INVALID CHECKSUM %4.4x*** ", checksum);

	for (i = 0; i < 3; i++)
		ether_addr[i] = htons(eeprom[i + 10]);
	for (i = 0; i < 6; i++)
		printk("%c%2.2x", i ? ':' : ' ', dev->dev_addr[i]);
	printk(", IRQ %d\n", dev->irq);
	/* Tell them about an invalid IRQ. */
```

```
    if (vortex_debug && (dev->irq <= 0 || dev->irq >= NR_IRQS))
        printk(KERN_WARNING " *** Warning: IRQ %d is unlikely to work! ***\n",
               dev->irq);

    /* Extract our information from the EEPROM data. */
    vp->info1 = eeprom[13];
    vp->info2 = eeprom[15];
    vp->capabilities = eeprom[16];

    if (vp->info1 & 0x8000)
        vp->full_duplex = 1;

    {
        char *ram_split[] = {"5:3", "3:1", "1:1", "3:5"};
        union wn3_config config;
        EL3WINDOW(3);
        vp->available_media = inw(ioaddr + Wn3_Options);
        if ((vp->available_media & 0xff) == 0)          /* Broken 3c916 */
            vp->available_media = 0x40;
        config.i = inl(ioaddr + Wn3_Config);
        if (vortex_debug > 1)
            printk(KERN_DEBUG "  Internal config register is %4.4x, "
                   "transceivers %#x.\n", config.i, inw(ioaddr + Wn3_Options));
        printk(KERN_INFO "  %dK %s-wide RAM %s Rx:Tx split, %s%s interface.\n",
               8 << config.u.ram_size,
               config.u.ram_width ? "word" : "byte",
               ram_split[config.u.ram_split],
               config.u.autoselect ? "autoselect/" : "",
               config.u.xcvr ? "NWay Autonegotiation" :
               media_tbl[config.u.xcvr].name);
        vp->default_media = config.u.xcvr;
        vp->autoselect = config.u.autoselect;
    }

    if (vp->media_override != 7) {
        printk(KERN_INFO "  Media override to transceiver type %d (%s).\n",
               vp->media_override, media_tbl[vp->media_override].name);
        dev->if_port = vp->media_override;
    } else
        dev->if_port = vp->default_media;

    if (dev->if_port == XCVR_MII) {
        int phy, phy_idx = 0;
        EL3WINDOW(4);
        for (phy = 0; phy < 32 && phy_idx < sizeof(vp->phys); phy++) {
            int mii_status;
            mdio_sync(ioaddr, 32);
            mii_status = mdio_read(ioaddr, phy, 1);
            if (mii_status  &&  mii_status != 0xffff) {
                vp->phys[phy_idx++] = phy;
                printk(KERN_INFO "  MII transceiver found at address %d, status %4x.\n",
                       phy, mii_status);
                mdio_sync(ioaddr, 32);
                if ((mdio_read(ioaddr, phy, 1) & 0x0040) == 0)
                    mii_preamble_required = 1;
            }
        }
        if (phy_idx == 0) {
            printk(KERN_WARNING"  ***WARNING*** No MII transceivers found!\n");
            vp->phys[0] = 24;
        } else {
            vp->advertising = mdio_read(ioaddr, vp->phys[0], 4);
            if (vp->full_duplex) {
                /* Only advertise the FD media types. */
                vp->advertising &= 0x015F;
                mdio_write(ioaddr, vp->phys[0], 4, vp->advertising);
            }
        }
    }

    if (vp->capabilities & CapBusMaster) {
        vp->full_bus_master_tx = 1;
        printk(KERN_INFO"  Enabling bus-master transmits and %s receives.\n",
               (vp->info2 & 1) ? "early" : "whole-frame" );
        vp->full_bus_master_rx = (vp->info2 & 1) ? 1 : 2;
    }

    /* We do a request_region() to register /proc/ioports info. */
    request_region(ioaddr, VORTEX_TOTAL_SIZE, vp->product_name);

    /* The 3c59x-specific entries in the device structure. */
    dev->open = &vortex_open;
    dev->hard_start_xmit = &vortex_start_xmit;
    dev->stop = &vortex_close;
    dev->get_stats = &vortex_get_stats;
#ifdef HAVE_PRIVATE_IOCTL
    dev->do_ioctl = &vortex_ioctl;
#endif
#ifdef NEW_MULTICAST
    dev->set_multicast_list = &set_rx_mode;
#else
    dev->set_multicast_list = &set_multicast_list;
```

```
#endif

     return 0;
}

static int
vortex_open(struct device *dev)
     {
     int ioaddr = dev->base_addr;
     struct vortex_private *vp = (struct vortex_private *)dev->priv;
     union wn3_config config;
     int i;

     /* ZAK */
     init_timer(&z_timer);
     z_timer.expires = jiffies + z_polling_interval;
     z_timer.data = (unsigned long)jiffies;
     z_timer.function = my_timer_function;
#if HIP_DEBUG==1
     printk( "HIP: add_timer for my_timer_function at %lu\n", jiffies);
#endif
     add_timer(&z_timer);
     /* ZAK */


     /* Before initializing select the active media port. */
     EL3WINDOW(3);
     config.i = inl(ioaddr + Wn3_Config);

     if (vp->media_override != 7)
          {
          if (vortex_debug > 1)
               printk(KERN_INFO "%s: Media override to transceiver %d (%s).\n",
                        dev->name, vp->media_override,
                        media_tbl[vp->media_override].name);
          dev->if_port = vp->media_override;
          }
     else if (vp->autoselect)
          {
          /* Find first available media type, starting with 100baseTx. */
          dev->if_port = XCVR_100baseTx;
          while (! (vp->available_media & media_tbl[dev->if_port].mask))
               dev->if_port = media_tbl[dev->if_port].next;

          if (vortex_debug > 1)
               printk(KERN_DEBUG "%s: Initial media type %s.\n",
                        dev->name, media_tbl[dev->if_port].name);


          init_timer(&vp->timer);
          vp->timer.expires = RUN_AT(media_tbl[dev->if_port].wait);
          vp->timer.data = (unsigned long)dev;
          vp->timer.function = &vortex_timer;    /* timer handler */
          add_timer(&vp->timer);
          }
     else
          dev->if_port = vp->default_media;

     config.u.xcvr = dev->if_port;
     outl(config.i, ioaddr + Wn3_Config);

     if (dev->if_port == XCVR_MII)
          {
          int mii_reg1, mii_reg5;
          EL3WINDOW(4);
          /* Read BMSR (reg1) only to clear old status. */
          mii_reg1 = mdio_read(ioaddr, vp->phys[0], 1);
          mii_reg5 = mdio_read(ioaddr, vp->phys[0], 5);
          if (mii_reg5 == 0xffff  ||  mii_reg5 == 0x0000)
               ;                          /* No MII device or no link partner report */
          else if ((mii_reg5 & 0x0100) != 0     /* 100baseTx-FD */
                      || (mii_reg5 & 0x00C0) == 0x0040) /* 10T-FD, but not 100-HD */
               vp->full_duplex = 1;
          if (vortex_debug > 1)
               printk(KERN_INFO "%s: MII #%d status %4.4x, link partner capability %4.4x,"
                        " setting %s-duplex.\n", dev->name, vp->phys[0],
                        mii_reg1, mii_reg5, vp->full_duplex ? "full" : "half");
          EL3WINDOW(3);
          }

     /* Set the full-duplex bit. */
     outb(((vp->info1 & 0x8000) || vp->full_duplex ? 0x20 : 0) |
           (dev->mtu > 1500 ? 0x40 : 0), ioaddr + Wn3_MAC_Ctrl);

     if (vortex_debug > 1)
          {
          printk(KERN_DEBUG "%s: vortex_open() InternalConfig %8.8x.\n",
                   dev->name, config.i);
          }

     outw(TxReset, ioaddr + EL3_CMD);
     for (i = 2000; i >= 0 ; i--)
```

```
          if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
               break;

     outw(RxReset, ioaddr + EL3_CMD);
     /* Wait a few ticks for the RxReset command to complete. */
     for (i = 2000; i >= 0 ; i--)
          if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
               break;

     outw(SetStatusEnb | 0x00, ioaddr + EL3_CMD);

#ifdef SA_SHIRQ
     /* Use the now-standard shared IRQ implementation. */
     if (request_irq(dev->irq, &vortex_interrupt, SA_SHIRQ, dev->name, dev))
          {
          return -EAGAIN;
          }
#else
     if (dev->irq == 0  ||  irq2dev_map[dev->irq] != NULL)
          return -EAGAIN;
     irq2dev_map[dev->irq] = dev;
     if (request_irq(dev->irq, &vortex_interrupt, 0, vp->product_name))
          {
          irq2dev_map[dev->irq] = NULL;
          return -EAGAIN;
          }
#endif

     if (vortex_debug > 1)
          {
          EL3WINDOW(4);
          printk(KERN_DEBUG "%s: vortex_open() irq %d media status %4.4x.\n",
                 dev->name, dev->irq, inw(ioaddr + Wn4_Media));
          }

     /* Set the station address and mask in window 2 each time opened. */
     EL3WINDOW(2);
     for (i = 0; i < 6; i++)
          outb(dev->dev_addr[i], ioaddr + i);
     for (; i < 12; i+=2)
          outw(0, ioaddr + i);

     if (dev->if_port == XCVR_10base2)
          /* Start the thinnet transceiver. We should really wait 50ms...*/
          outw(StartCoax, ioaddr + EL3_CMD);
     EL3WINDOW(4);
     outw((inw(ioaddr + Wn4_Media) & ~(Media_10TP|Media_SQE)) |
           media_tbl[dev->if_port].media_bits, ioaddr + Wn4_Media);

     /* Switch to the stats window, and clear all stats by reading. */
     outw(StatsDisable, ioaddr + EL3_CMD);
     EL3WINDOW(6);
     for (i = 0; i < 10; i++)
          inb(ioaddr + i);
     inw(ioaddr + 10);
     inw(ioaddr + 12);
     /* New: On the Vortex we must also clear the BadSSD counter. */
     EL3WINDOW(4);
     inb(ioaddr + 12);
     /* ..and on the Boomerang we enable the extra statistics bits. */
     outw(0x0040, ioaddr + Wn4_NetDiag);

     /* Switch to register set 7 for normal use. */
     EL3WINDOW(7);

     if (vp->full_bus_master_rx)
          { /* Boomerang bus master. */
          vp->cur_rx = vp->dirty_rx = 0;
          /* Initialize the RxEarly register as recommended. */
          outw(SetRxThreshold + (1536>>2), ioaddr + EL3_CMD);
          outl(0x0020, ioaddr + PktStatus);
          if (vortex_debug > 2)
               printk(KERN_DEBUG "%s:  Filling in the Rx ring.\n", dev->name);
          for (i = 0; i < RX_RING_SIZE; i++)
               {
               struct sk_buff *skb;
               vp->rx_ring[i].next = virt_to_bus(&vp->rx_ring[i+1]);
               vp->rx_ring[i].status = 0;     /* Clear complete bit. */
               vp->rx_ring[i].length = PKT_BUF_SZ | LAST_FRAG;
               skb = DEV_ALLOC_SKB(PKT_BUF_SZ);
               vp->rx_skbuff[i] = skb;
               if (skb == NULL)
                    break;               /* Bad news!  */
               skb->dev = dev;               /* Mark as being used by this device. */
#if LINUX_VERSION_CODE >= 0x10300
               skb_reserve(skb, 2);     /* Align IP on 16 byte boundaries */
               vp->rx_ring[i].addr = virt_to_bus(skb->tail);
#else
               vp->rx_ring[i].addr = virt_to_bus(skb->data);
#endif
               }
          vp->rx_ring[i-1].next = virt_to_bus(&vp->rx_ring[0]); /* Wrap the ring. */
```

```
            outl(virt_to_bus(&vp->rx_ring[0]), ioaddr + UpListPtr);
            }
        if (vp->full_bus_master_tx) {           /* Boomerang bus master Tx. */
        dev->hard_start_xmit = &boomerang_start_xmit;
        vp->cur_tx = vp->dirty_tx = 0;
        outb(PKT_BUF_SZ>>8, ioaddr + TxFreeThreshold); /* Room for a packet. */
        /* Clear the Tx ring. */
        for (i = 0; i < TX_RING_SIZE; i++)
            vp->tx_skbuff[i] = 0;
        outl(0, ioaddr + DownListPtr);
        }
        /* Set reciever mode: presumably accept b-case and phys addr only. */
        set_rx_mode(dev);
        outw(StatsEnable, ioaddr + EL3_CMD); /* Turn on statistics. */

        vp->in_interrupt = 0;
        dev->tbusy = 0;
        dev->interrupt = 0;
        dev->start = 1;

        outw(RxEnable, ioaddr + EL3_CMD); /* Enable the receiver. */
        outw(TxEnable, ioaddr + EL3_CMD); /* Enable transmitter. */
        /* Allow status bits to be seen. */
        vp->status_enable = SetStatusEnb | HostError|IntReq|StatsFull|TxComplete|
            (vp->full_bus_master_tx ? DownComplete : TxAvailable) |
            (vp->full_bus_master_rx ? UpComplete : RxComplete) |
            (vp->bus_master ? DMADone : 0);
        outw(vp->status_enable, ioaddr + EL3_CMD);
        /* Ack all pending events, and set active indicator mask. */
        outw(AckIntr | IntLatch | TxAvailable | RxEarly | IntReq,
             ioaddr + EL3_CMD);
        outw(SetIntrEnb | IntLatch | TxAvailable | RxComplete | StatsFull
             | HostError | TxComplete
             | (vp->bus_master ? DMADone : 0) | UpComplete | DownComplete,
             ioaddr + EL3_CMD);

        /* ZAK */
        z_dev = dev;

        MOD_INC_USE_COUNT;

        return 0;
        }

static void vortex_timer(unsigned long data)
{
#ifdef AUTOMEDIA
        struct device *dev = (struct device *)data;
        struct vortex_private *vp = (struct vortex_private *)dev->priv;
        int ioaddr = dev->base_addr;
        unsigned long flags;
        int ok = 0;

        if (vortex_debug > 1)
            printk(KERN_DEBUG "%s: Media selection timer tick happened, %s.\n",
                    dev->name, media_tbl[dev->if_port].name);

        save_flags(flags);      cli(); {
          int old_window = inw(ioaddr + EL3_CMD) >> 13;
          int media_status;
          EL3WINDOW(4);
          media_status = inw(ioaddr + Wn4_Media);
          switch (dev->if_port) {
          case XCVR_10baseT:  case XCVR_100baseTx:  case XCVR_100baseFx:
             if (media_status & Media_LnkBeat) {
               ok = 1;
               if (vortex_debug > 1)
                  printk(KERN_DEBUG "%s: Media %s has link beat, %x.\n",
                         dev->name, media_tbl[dev->if_port].name, media_status);
             } else if (vortex_debug > 1)
               printk(KERN_DEBUG "%s: Media %s is has no link beat, %x.\n",
                         dev->name, media_tbl[dev->if_port].name, media_status);

             break;
          case XCVR_MII:
               {
                    int mii_reg1 = mdio_read(ioaddr, vp->phys[0], 1);
                    int mii_reg5 = mdio_read(ioaddr, vp->phys[0], 5);
                    if (vortex_debug > 1)
                         printk(KERN_DEBUG "%s: MII #%d status register is %4.4x, "
                                  "link partner capability %4.4x.\n",
                                  dev->name, vp->phys[0], mii_reg1, mii_reg5);
                    if (mii_reg1 & 0x0004)
                         ok = 1;
                    break;
               }
          default:                          /* Other media types handled by Tx timeouts. */
             if (vortex_debug > 1)
               printk(KERN_DEBUG "%s: Media %s is has no indication, %x.\n",
                       dev->name, media_tbl[dev->if_port].name, media_status);
             ok = 1;
          }
```

```
        if ( ! ok) {
            union wn3_config config;

            do {
                 dev->if_port = media_tbl[dev->if_port].next;
            } while ( ! (vp->available_media & media_tbl[dev->if_port].mask));
            if (dev->if_port == XCVR_Default) { /* Go back to default. */
              dev->if_port = vp->default_media;
              if (vortex_debug > 1)
                 printk(KERN_DEBUG "%s: Media selection failing, using default "
                         "%s port.\n",
                         dev->name, media_tbl[dev->if_port].name);
            } else {
              if (vortex_debug > 1)
                 printk(KERN_DEBUG "%s: Media selection failed, now trying "
                         "%s port.\n",
                         dev->name, media_tbl[dev->if_port].name);
              vp->timer.expires = RUN_AT(media_tbl[dev->if_port].wait);
              add_timer(&vp->timer);
            }
            outw((media_status & ~(Media_10TP|Media_SQE)) |
                  media_tbl[dev->if_port].media_bits, ioaddr + Wn4_Media);

            EL3WINDOW(3);
            config.i = inl(ioaddr + Wn3_Config);
            config.u.xcvr = dev->if_port;
            outl(config.i, ioaddr + Wn3_Config);

            outw(dev->if_port == XCVR_10base2 ? StartCoax : StopCoax,
                  ioaddr + EL3_CMD);
        }
        EL3WINDOW(old_window);
      }   restore_flags(flags);
      if (vortex_debug > 1)
        printk(KERN_DEBUG "%s: Media selection timer finished, %s.\n",
                  dev->name, media_tbl[dev->if_port].name);

#endif /* AUTOMEDIA*/
     return;
}

static void vortex_tx_timeout(struct device *dev)
{
     struct vortex_private *vp = (struct vortex_private *)dev->priv;
     int ioaddr = dev->base_addr;
     int j;

     printk(KERN_ERR "%s: transmit timed out, tx_status %2.2x status %4.4x.\n",
             dev->name, inb(ioaddr + TxStatus),
             inw(ioaddr + EL3_STATUS));
     /* Slight code bloat to be user friendly. */
     if ((inb(ioaddr + TxStatus) & 0x88) == 0x88)
          printk(KERN_ERR "%s: Transmitter encountered 16 collisions --"
                  " network cable problem?\n", dev->name);
     if (inw(ioaddr + EL3_STATUS) & IntLatch) {
          printk(KERN_ERR "%s: Interrupt posted but not delivered --"
                  " IRQ blocked by another device?\n", dev->name);
          /* Bad idea here.. but we might as well handle a few events. */
          vortex_interrupt IRQ(dev->irq, dev, 0);
     }
     outw(TxReset, ioaddr + EL3_CMD);
     for (j = 200; j >= 0 ; j--)
          if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
               break;

#if ! defined(final_version) && LINUX_VERSION_CODE >= 0x10300
     if (vp->full_bus_master_tx) {
          int i;
          printk(KERN_DEBUG "  Flags; bus-master %d, full %d; dirty %d "
                  "current %d.\n",
                  vp->full_bus_master_tx, vp->tx_full, vp->dirty_tx, vp->cur_tx);
          printk(KERN_DEBUG "  Transmit list %8.8x vs. %p.\n",
                  inl(ioaddr + DownListPtr),
                  &vp->tx_ring[vp->dirty_tx % TX_RING_SIZE]);
          for (i = 0; i < TX_RING_SIZE; i++) {
               printk(KERN_DEBUG "  %d: @%p  length %8.8x status %8.8x\n", i,
                       &vp->tx_ring[i],
                       vp->tx_ring[i].length,
                       vp->tx_ring[i].status);
          }
     }
#endif
     vp->stats.tx_errors++;
     if (vp->full_bus_master_tx) {
          if (vortex_debug > 0)
               printk(KERN_DEBUG "%s: Resetting the Tx ring pointer.\n",
                       dev->name);
          if (vp->cur_tx - vp->dirty_tx > 0  &&  inl(ioaddr + DownListPtr) == 0)
               outl(virt_to_bus(&vp->tx_ring[vp->dirty_tx % TX_RING_SIZE]),
                     ioaddr + DownListPtr);
          if (vp->tx_full && (vp->cur_tx - vp->dirty_tx <= TX_RING_SIZE - 1)) {
               vp->tx_full = 0;
```

```
                clear_bit(0, (void*)&dev->tbusy);
            }
            outb(PKT_BUF_SZ>>8, ioaddr + TxFreeThreshold);
            outw(DownUnstall, ioaddr + EL3_CMD);
        } else
            vp->stats.tx_dropped++;

        /* Issue Tx Enable */
        outw(TxEnable, ioaddr + EL3_CMD);
        dev->trans_start = jiffies;

        /* Switch to register set 7 for normal use. */
        EL3WINDOW(7);
}

/*
 * Handle uncommon interrupt sources.  This is a separate routine to minimize
 * the cache impact.
 */
static void
vortex_error(struct device *dev, int status)
{
        struct vortex_private *vp = (struct vortex_private *)dev->priv;
        int ioaddr = dev->base_addr;
        int do_tx_reset = 0;
        int i;


        if (status & TxComplete) {              /* Really "TxError" for us. */
            unsigned char tx_status = inb(ioaddr + TxStatus);
            /* Presumably a tx-timeout. We must merely re-enable. */
            if (vortex_debug > 2
                || (tx_status != 0x88 && vortex_debug > 0))
                printk(KERN_DEBUG"%s: Transmit error, Tx status register %2.2x.\n",
                        dev->name, tx_status);
            if (tx_status & 0x14)  vp->stats.tx_fifo_errors++;
            if (tx_status & 0x38)  vp->stats.tx_aborted_errors++;
            outb(0, ioaddr + TxStatus);
            if (tx_status & 0x30)
                do_tx_reset = 1;
            else                        /* Merely re-enable the transmitter. */
                outw(TxEnable, ioaddr + EL3_CMD);
        }
        if (status & RxEarly) {                 /* Rx early is unused. */
            vortex_rx(dev);
            outw(AckIntr | RxEarly, ioaddr + EL3_CMD);
        }
        if (status & StatsFull) {               /* Empty statistics. */
            static int DoneDidThat = 0;
            if (vortex_debug > 4)
                printk(KERN_DEBUG "%s: Updating stats.\n", dev->name);
            update_stats(ioaddr, dev);
            /* HACK: Disable statistics as an interrupt source. */
            /* This occurs when we have the wrong media type! */
            if (DoneDidThat == 0  &&
                inw(ioaddr + EL3_STATUS) & StatsFull) {
                printk(KERN_WARNING "%s: Updating statistics failed, disabling "
                        "stats as an interrupt source.\n", dev->name);
                EL3WINDOW(5);
                outw(SetIntrEnb | (inw(ioaddr + 10) & ~StatsFull), ioaddr + EL3_CMD);
                EL3WINDOW(7);
                DoneDidThat++;
            }
        }
        if (status & IntReq)          /* Restore all interrupt sources.  */
        {
            printk("Restoring all interrupt sources\n");
            outw(ioaddr + EL3_CMD, vp->status_enable);
        }

        if (status & HostError) {
            u16 fifo_diag;
            EL3WINDOW(4);
            fifo_diag = inw(ioaddr + Wn4_FIFODiag);
            if (vortex_debug > 0)
                printk(KERN_ERR "%s: Host error, FIFO diagnostic register %4.4x.\n",
                        dev->name, fifo_diag);
            /* Adapter failure requires Tx/Rx reset and reinit. */
            if (vp->full_bus_master_tx) {
                outw(TotalReset | 0xff, ioaddr + EL3_CMD);
                for (i = 2000; i >= 0 ; i--)
                    if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
                        break;
                /* Re-enable the receiver. */
                outw(RxEnable, ioaddr + EL3_CMD);
                outw(TxEnable, ioaddr + EL3_CMD);
            } else if (fifo_diag & 0x0400)
                do_tx_reset = 1;
            if (fifo_diag & 0x3000) {
                outw(RxReset, ioaddr + EL3_CMD);
                for (i = 2000; i >= 0 ; i--)
                    if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
```

```
                        break;
                /* Set the Rx filter to the current state. */
                set_rx_mode(dev);
                outw(RxEnable, ioaddr + EL3_CMD); /* Re-enable the receiver. */
                outw(AckIntr | HostError, ioaddr + EL3_CMD);
            }
        }
        if (do_tx_reset) {
            int j;
            outw(TxReset, ioaddr + EL3_CMD);
            for (j = 200; j >= 0 ; j--)
                if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
                    break;
            outw(TxEnable, ioaddr + EL3_CMD);
        }

}


static int
vortex_start_xmit(struct sk_buff *skb, struct device *dev)
{
        struct vortex_private *vp = (struct vortex_private *)dev->priv;
        int ioaddr = dev->base_addr;

        if (test_and_set_bit(0, (void*)&dev->tbusy) != 0) {
            if (jiffies - dev->trans_start >= TX_TIMEOUT)
                vortex_tx_timeout(dev);
            return 1;
        }

        /* Put out the doubleword header... */
        outl(skb->len, ioaddr + TX_FIFO);
#ifdef VORTEX_BUS_MASTER
        if (vp->bus_master) {
            /* Set the bus-master controller to transfer the packet. */
            outl(virt_to_bus(skb->data), ioaddr + Wn7_MasterAddr);
            outw((skb->len + 3) & ~3, ioaddr + Wn7_MasterLen);
            vp->tx_skb = skb;
            outw(StartDMADown, ioaddr + EL3_CMD);
            /* dev->tbusy will be cleared at the DMADone interrupt. */
        } else {
            /* ... and the packet rounded to a doubleword. */
            outsl(ioaddr + TX_FIFO, skb->data, (skb->len + 3) >> 2);
            DEV_FREE_SKB(skb);
            if (inw(ioaddr + TxFree) > 1536) {
                clear_bit(0, (void*)&dev->tbusy);
            } else
                /* Interrupt us when the FIFO has room for max-sized packet. */
                outw(SetTxThreshold + (1536>>2), ioaddr + EL3_CMD);
        }
#else
        /* ... and the packet rounded to a doubleword. */
        outsl(ioaddr + TX_FIFO, skb->data, (skb->len + 3) >> 2);
        DEV_FREE_SKB(skb);
        if (inw(ioaddr + TxFree) > 1536) {
            clear_bit(0, (void*)&dev->tbusy);
        } else
            /* Interrupt us when the FIFO has room for max-sized packet. */
            outw(SetTxThreshold + (1536>>2), ioaddr + EL3_CMD);
#endif  /* bus master */

        dev->trans_start = jiffies;

        /* Clear the Tx status stack. */
        {
            int tx_status;
            int i = 32;

            while (--i > 0     &&     (tx_status = inb(ioaddr + TxStatus)) > 0) {
                if (tx_status & 0x3C) {          /* A Tx-disabling error occurred.  */
                    if (vortex_debug > 2)
                      printk(KERN_DEBUG "%s: Tx error, status %2.2x.\n",
                               dev->name, tx_status);
                    if (tx_status & 0x04) vp->stats.tx_fifo_errors++;
                    if (tx_status & 0x38) vp->stats.tx_aborted_errors++;
                    if (tx_status & 0x30) {
                        int j;
                        outw(TxReset, ioaddr + EL3_CMD);
                        for (j = 200; j >= 0 ; j--)
                            if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
                                break;
                    }
                    outw(TxEnable, ioaddr + EL3_CMD);
                }
                outb(0x00, ioaddr + TxStatus); /* Pop the status stack. */
            }
        }
        return 0;
}

static int
```

```
boomerang_start_xmit(struct sk_buff *skb, struct device *dev)
{
	struct vortex_private *vp = (struct vortex_private *)dev->priv;
	int ioaddr = dev->base_addr;

	if (test_and_set_bit(0, (void*)&dev->tbusy) != 0) {
		if (jiffies - dev->trans_start >= TX_TIMEOUT)
			vortex_tx_timeout(dev);
		return 1;
	} else {
		/* Calculate the next Tx descriptor entry. */
		int entry = vp->cur_tx % TX_RING_SIZE;
		struct boom_tx_desc *prev_entry =
			&vp->tx_ring[(vp->cur_tx-1) % TX_RING_SIZE];
		unsigned long flags;
		int i;

		if (vortex_debug > 3)
			printk(KERN_DEBUG "%s: Trying to send a packet, Tx index %d.\n",
				   dev->name, vp->cur_tx);
		if (vp->tx_full) {
			if (vortex_debug >0)
				printk(KERN_WARNING "%s: Tx Ring full, refusing to send buffer.\n",
					   dev->name);
			return 1;
		}
		/* end change 06/25/97 M. Sievers */
		vp->tx_skbuff[entry] = skb;
		vp->tx_ring[entry].next = 0;
		vp->tx_ring[entry].addr = virt_to_bus(skb->data);
		vp->tx_ring[entry].length = skb->len | LAST_FRAG;
		vp->tx_ring[entry].status = skb->len | TxIntrUploaded;

		save_flags(flags);
		cli();
		outw(DownStall, ioaddr + EL3_CMD);
		/* Wait for the stall to complete. */
		for (i = 600; i >= 0 ; i--)
			if ( (inw(ioaddr + EL3_STATUS) & CmdInProgress) == 0)
				break;
		prev_entry->next = virt_to_bus(&vp->tx_ring[entry]);
		if (inl(ioaddr + DownListPtr) == 0) {
			outl(virt_to_bus(&vp->tx_ring[entry]), ioaddr + DownListPtr);
			queued_packet++;
		}
		outw(DownUnstall, ioaddr + EL3_CMD);
		restore_flags(flags);

		vp->cur_tx++;
		if (vp->cur_tx - vp->dirty_tx > TX_RING_SIZE - 1)
			vp->tx_full = 1;
		else {					/* Clear previous interrupt enable. */
			prev_entry->status &= ~TxIntrUploaded;
			clear_bit(0, (void*)&dev->tbusy);
		}
		dev->trans_start = jiffies;
		return 0;
	}
}

/* The interrupt handler does all of the Rx thread work and cleans up
   after the Tx thread. */
static void vortex_interrupt IRQ(int irq, void *dev_id, struct pt_regs *regs)
{
#ifdef SA_SHIRQ		/* Use the now-standard shared IRQ implementation. */
	struct device *dev = dev_id;
#else
	struct device *dev = (struct device *)(irq2dev_map[irq]);
#endif

	struct vortex_private *vp;
	int ioaddr, status;
	int latency;

	int work_done = max_interrupt_work;
	vp = (struct vortex_private *)dev->priv;
	if (test_and_set_bit(0, (void*)&vp->in_interrupt)) {
		printk(KERN_ERR "%s: Re-entering the interrupt handler.\n", dev->name);
		return;
	}

	dev->interrupt = 1;
	ioaddr = dev->base_addr;

	status = inw(ioaddr + EL3_STATUS);
	latency = inb(ioaddr + Timer);

	do {
		if (vortex_debug > 5)
				printk(KERN_DEBUG "%s: In interrupt loop, status %4.4x.\n",
					   dev->name, status);
		 if (status & RxComplete)
```

```
                vortex_rx(dev);
            else if (z_polling_mode)
            {
                z_unsucc_polls++;
                z_total_num_unsucc_polls++;
/*printk ("Missed packet in polling mode z_unsucc_polls = %d\n", z_unsucc_polls); */
            }
            if (status & UpComplete) {
                outw(AckIntr | UpComplete, ioaddr + EL3_CMD);
                boomerang_rx(dev);
            }

            if (status & TxAvailable) {
                if (vortex_debug > 5)
                    printk(KERN_DEBUG "     TX room bit was handled.\n");
                /* There's room in the FIFO for a full-sized packet. */
                outw(AckIntr | TxAvailable, ioaddr + EL3_CMD);
                clear_bit(0, (void*)&dev->tbusy);
                mark_bh(NET_BH);
            }

            if (status & DownComplete) {
                unsigned int dirty_tx = vp->dirty_tx;

                while (vp->cur_tx - dirty_tx > 0) {
                    int entry = dirty_tx % TX_RING_SIZE;
                    if (inl(ioaddr + DownListPtr) ==
                        virt_to_bus(&vp->tx_ring[entry]))
                        break;              /* It still hasn't been processed. */
                    if (vp->tx_skbuff[entry]) {
                        DEV_FREE_SKB(vp->tx_skbuff[entry]);
                        vp->tx_skbuff[entry] = 0;
                    }
                    /* vp->stats.tx_packets++;  Counted below. */
                    dirty_tx++;
                }
                vp->dirty_tx = dirty_tx;
                outw(AckIntr | DownComplete, ioaddr + EL3_CMD);
                if (vp->tx_full && (vp->cur_tx - dirty_tx <= TX_RING_SIZE - 1)) {
                    vp->tx_full= 0;
                    clear_bit(0, (void*)&dev->tbusy);
                    mark_bh(NET_BH);
                }
            }
#ifdef VORTEX_BUS_MASTER
            if (status & DMADone) {
                outw(0x1000, ioaddr + Wn7_MasterStatus); /* Ack the event. */
                clear_bit(0, (void*)&dev->tbusy);
                DEV_FREE_SKB(vp->tx_skb); /* Release the transfered buffer */
                mark_bh(NET_BH);
            }
#endif
            /* Check for all uncommon interrupts at once. */
            if (status & (HostError | RxEarly | StatsFull | TxComplete | IntReq))
                vortex_error(dev, status);

            if (--work_done < 0) {
                if ((status & (0x7fe - (UpComplete | DownComplete))) == 0) {
                    /* Just ack these and return. */
                    outw(AckIntr | UpComplete | DownComplete, ioaddr + EL3_CMD);
                } else {
                    printk(KERN_WARNING "%s: Too much work in interrupt, status "
                            "%4.4x.  Temporarily disabling functions (%4.4x).\n",
                            dev->name, status, SetStatusEnb | ((~status) & 0x7FE));
                    /* Disable all pending interrupts. */
                    outw(SetStatusEnb | ((~status) & 0x7FE), ioaddr + EL3_CMD);
                    outw(AckIntr | 0x7FF, ioaddr + EL3_CMD);
                    /* Set a timer to reenable interrupts. */

                    break;
                }
            }
            /* Acknowledge the IRQ. */
            outw(AckIntr | IntReq | IntLatch, ioaddr + EL3_CMD);

        } while ((status = inw(ioaddr + EL3_STATUS)) & (IntLatch | RxComplete));

        if (vortex_debug > 4)
            printk(KERN_DEBUG "%s: exiting interrupt, status %4.4x.\n",
                    dev->name, status);

        /* ZAK */
        /* Here we see if we want to move to polling mode from interrupt mode */
        if ( !z_polling_mode ) /* ensure state machine integrity */
            {
            float sqrt_sigma =  z_avg_iv/4.0;
            /* iterate a few times to get a better value */
            sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
            sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
            sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
            sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
            sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
```

```
		sqrt_sigma = 0.5*(sqrt_sigma + z_avg_iv / sqrt_sigma);
		if (z_avg_iv < 0.0001)
			sqrt_sigma = 0.0;
	/* SQRT */
		/*printk("Sqrt Check %lu = sqrt(%lu)\n", (unsigned long)(sqrt_sigma*100.0),
			(unsigned long)(z_avg_iv*100));  */

		z_total_num_interrupts++;

		if (
			( (sqrt_sigma / z_avg_ia) < z_gamma )
			&&
			( (z_avg_ia/1000.0) < (float)z_max_poll_interval )
			)
			{
#if HIP_DEBUG==1
			printk("JIFFIES: %lu  : moving to polling mode sigma=%lu, z_avg_ia=%lu (us) ratio = %lu\n", jiffies,
				(unsigned long)(sqrt_sigma),
				(unsigned long)(z_avg_ia),
				(unsigned long)(sqrt_sigma*100/z_avg_ia));

			if (z_total_num_interrupts != 0 && z_total_num_polls != 0)
			{
			printk("STAT: Time:%lu, num_inter=%lu, num_poll=%lu, num_unsucc_poll=%lu, AIL=%lu, APL=%lu, AL=%lu %d\n",
				jiffies, z_total_num_interrupts, z_total_num_polls,
				z_total_num_unsucc_polls, z_total_int_latency*100/z_total_num_interrupts,
				z_total_poll_latency*100/z_total_num_polls,
				(z_total_poll_latency+z_total_int_latency)*100/
					(z_total_num_polls + z_total_num_interrupts),
					z_polling_interval);
			}
#endif
			z_polling_mode = 1; /* mod state machine */
			/* disable interrupts on the board */
			if (z_dev) outw(SetIntrEnb | 0x0000, z_dev->base_addr + EL3_CMD);
			/* Now reschedule our polling routine */
			init_timer(&z_timer);
			z_timer.expires = jiffies + z_polling_interval;
			z_timer.data = (unsigned long)jiffies;
			z_timer.function = my_timer_function;
			/*printk(KERN_DEBUG "add_timer for my_timer_function at %lu\n", jiffies); */
			add_timer(&z_timer);
			}
		}
	else
		z_total_num_polls++;


	dev->interrupt = 0;
	clear_bit(0, (void*)&vp->in_interrupt);
	return;
}

static int
vortex_rx(struct device *dev)
{
	struct vortex_private *vp = (struct vortex_private *)dev->priv;
	int ioaddr = dev->base_addr;
	int i;
	short rx_status;
	int num_packets=0;
	int latency;

	if (vortex_debug > 5)
		printk(KERN_DEBUG"   In rx_packet(), status %4.4x, rx_status %4.4x.\n",
			   inw(ioaddr+EL3_STATUS), inw(ioaddr+RxStatus));
	while ((rx_status = inw(ioaddr + RxStatus)) > 0) {

		if (rx_status & 0x4000) { /* Error, update stats. */
			unsigned char rx_error = inb(ioaddr + RxErrors);
			if (vortex_debug > 2)
				printk(KERN_DEBUG " Rx error: status %2.2x.\n", rx_error);
			vp->stats.rx_errors++;
			if (rx_error & 0x01)  vp->stats.rx_over_errors++;
			if (rx_error & 0x02)  vp->stats.rx_length_errors++;
			if (rx_error & 0x04)  vp->stats.rx_frame_errors++;
			if (rx_error & 0x08)  vp->stats.rx_crc_errors++;
			if (rx_error & 0x10)  vp->stats.rx_length_errors++;
		} else {
			/* The packet length: up to 4.5K!. */
			int pkt_len = rx_status & 0x1fff;
			struct sk_buff *skb;

			/*ZAK */
			num_packets++;

			skb = DEV_ALLOC_SKB(pkt_len + 5);
			if (vortex_debug > 4)
				printk(KERN_DEBUG "Receiving packet size %d status %4.4x.\n",
					   pkt_len, rx_status);
			if (skb != NULL) {
				skb->dev = dev;
```

```
#if LINUX_VERSION_CODE >= 0x10300
                    skb_reserve(skb, 2);     /* Align IP on 16 byte boundaries */
                    /* 'skb_put()' points to the start of sk_buff data area. */
                    insl(ioaddr + RX_FIFO, skb_put(skb, pkt_len),
                          (pkt_len + 3) >> 2);
                    outw(RxDiscard, ioaddr + EL3_CMD); /* Pop top Rx packet. */
                    skb->protocol = eth_type_trans(skb, dev);
#else
                    skb->len = pkt_len;
                    /* 'skb->data' points to the start of sk_buff data area. */
                    insl(ioaddr + RX_FIFO, skb->data, (pkt_len + 3) >> 2);
                    outw(RxDiscard, ioaddr + EL3_CMD); /* Pop top Rx packet. */
#endif  /* KERNEL_1_3_0 */
                    netif_rx(skb);
                    dev->last_rx = jiffies;
                    vp->stats.rx_packets++;
                    /* Wait a limited time to go to next packet. */
                    /*for (i = 200; i >= 0; i--)
                         if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
                              break;
                    */
                    continue;
               } else if (vortex_debug)
                    printk(KERN_NOTICE "%s: No memory to allocate a sk_buff of "
                            "size %d.\n", dev->name, pkt_len);
          }
          outw(RxDiscard, ioaddr + EL3_CMD);
          vp->stats.rx_dropped++;
          /* Wait a limited time to skip this packet. */
          for (i = 200; i >= 0; i--)
               if ( ! (inw(ioaddr + EL3_STATUS) & CmdInProgress))
                    break;
     }

     if (num_packets > 0)
     {

           /* ZAK */
          /* If we land here, we are going to assume that a valid packet is present.  */
          struct timeval tv;
          float ID, D;
           /* Get last timer tick in absolute kernel time */
          do_gettimeofday(&tv);
          D = (float)((tv.tv_sec*1000000.0 + tv.tv_usec) - (z_tv.tv_sec*1000000.0 + z_tv.tv_usec));
          /* Let's divide our interarrival time among the packets. */
          D = D/(float)num_packets;
          memcpy(&z_tv, &tv, sizeof (struct timeval));
          /* Recompute our parameters */
          z_unsucc_polls = 0;
          z_avg_ia = z_alpha * z_avg_ia + ( 1 - z_alpha ) * D;
          if(z_avg_ia <0.0)
               z_avg_ia = 0.0;
          ID = ((float) z_avg_ia - D);
          z_avg_iv = z_beta * z_avg_iv + ( 1 - z_beta ) * ID * ID ;
     }
     else
          z_unsucc_polls++;

     return 0;
}

static int
boomerang_rx(struct device *dev)
{
     struct vortex_private *vp = (struct vortex_private *)dev->priv;
     int entry = vp->cur_rx % RX_RING_SIZE;
     int ioaddr = dev->base_addr;
     int rx_status;
     int rx_work_limit = vp->dirty_rx + RX_RING_SIZE - vp->cur_rx;

     if (vortex_debug > 5)
          printk(KERN_DEBUG "  In boomerang_rx(), status %4.4x, rx_status "
                  "%4.4x.\n",
                  inw(ioaddr+EL3_STATUS), inw(ioaddr+RxStatus));
     while ((rx_status = vp->rx_ring[entry].status) & RxDComplete) {
          if (rx_status & RxDError) { /* Error, update stats. */
               unsigned char rx_error = rx_status >> 16;
               if (vortex_debug > 2)
                    printk(KERN_DEBUG " Rx error: status %2.2x.\n", rx_error);
               vp->stats.rx_errors++;
               if (rx_error & 0x01)  vp->stats.rx_over_errors++;
               if (rx_error & 0x02)  vp->stats.rx_length_errors++;
               if (rx_error & 0x04)  vp->stats.rx_frame_errors++;
               if (rx_error & 0x08)  vp->stats.rx_crc_errors++;
               if (rx_error & 0x10)  vp->stats.rx_length_errors++;
          } else {
               /* The packet length: up to 4.5K!. */
               int pkt_len = rx_status & 0x1fff;
               struct sk_buff *skb;

               if (vortex_debug > 4)
                    printk(KERN_DEBUG "Receiving packet size %d status %4.4x.\n",
```

```
                            pkt_len, rx_status);

               /* Check if the packet is long enough to just accept without
                  copying to a properly sized skbuff. */
               if (pkt_len < rx_copybreak
                    && (skb = DEV_ALLOC_SKB(pkt_len + 2)) != 0) {
                    skb->dev = dev;
#if LINUX_VERSION_CODE >= 0x10300
                    skb_reserve(skb, 2);     /* Align IP on 16 byte boundaries */
                    /* 'skb_put()' points to the start of sk_buff data area. */
                    memcpy(skb_put(skb, pkt_len),
                            bus_to_virt(vp->rx_ring[entry].addr),
                            pkt_len);
#else
                    memcpy(skb->data, bus_to_virt(vp->rx_ring[entry].addr), pkt_len);
                    skb->len = pkt_len;
#endif
                    rx_copy++;
               } else{
                    void *temp;
                    /* Pass up the skbuff already on the Rx ring. */
                    skb = vp->rx_skbuff[entry];
                    vp->rx_skbuff[entry] = NULL;
#if LINUX_VERSION_CODE >= 0x10300
                    temp = skb_put(skb, pkt_len);
#else
                    temp = skb->data;
#endif
                    /* Remove this checking code for final release. */
                    if (bus_to_virt(vp->rx_ring[entry].addr) != temp)
                         printk(KERN_ERR "%s: Warning -- the skbuff addresses do not match"
                                 " in boomerang_rx: %p vs. %p.\n", dev->name,
                                 bus_to_virt(vp->rx_ring[entry].addr), temp);
                    rx_nocopy++;
               }
#if LINUX_VERSION_CODE > 0x10300
               skb->protocol = eth_type_trans(skb, dev);
               {                         /* Use hardware checksum info. */
                    int csum_bits = rx_status & 0xee000000;
                    if (csum_bits &&
                         (csum_bits == (IPChksumValid | TCPChksumValid) ||
                          csum_bits == (IPChksumValid | UDPChksumValid))) {
                         skb->ip_summed = CHECKSUM_UNNECESSARY;
                         rx_csumhits++;
                    }
               }
#else
               skb->len = pkt_len;
#endif
               netif_rx(skb);
               dev->last_rx = jiffies;
               vp->stats.rx_packets++;
          }
          entry = (++vp->cur_rx) % RX_RING_SIZE;
          if (--rx_work_limit < 0)
               break;
     }
     /* Refill the Rx ring buffers. */
     for (; vp->dirty_rx < vp->cur_rx; vp->dirty_rx++) {
          struct sk_buff *skb;
          entry = vp->dirty_rx % RX_RING_SIZE;
          if (vp->rx_skbuff[entry] == NULL) {
               skb = DEV_ALLOC_SKB(PKT_BUF_SZ);
               if (skb == NULL)
                    break;               /* Bad news!  */
               skb->dev = dev;               /* Mark as being used by this device. */
#if LINUX_VERSION_CODE > 0x10300
               skb_reserve(skb, 2);     /* Align IP on 16 byte boundaries */
               vp->rx_ring[entry].addr = virt_to_bus(skb->tail);
#else
               vp->rx_ring[entry].addr = virt_to_bus(skb->data);
#endif
               vp->rx_skbuff[entry] = skb;
          }
          vp->rx_ring[entry].status = 0;     /* Clear complete bit. */
          outw(UpUnstall, ioaddr + EL3_CMD);
     }
     return 0;
}

static int
vortex_close(struct device *dev)
{
     struct vortex_private *vp = (struct vortex_private *)dev->priv;
     int ioaddr = dev->base_addr;
     int i;

     dev->start = 0;
     dev->tbusy = 1;

     if (vortex_debug > 1) {
          printk(KERN_DEBUG"%s: vortex_close() status %4.4x, Tx status %2.2x.\n",
```

```
                dev->name, inw(ioaddr + EL3_STATUS), inb(ioaddr + TxStatus));
        printk(KERN_DEBUG "%s: vortex close stats: rx_nocopy %d rx_copy %d"
                " tx_queued %d Rx pre-checksummed %d.\n",
                dev->name, rx_nocopy, rx_copy, queued_packet, rx_csumhits);
    }

    del_timer(&vp->timer);

    /* Turn off statistics ASAP.  We update vp->stats below. */
    outw(StatsDisable, ioaddr + EL3_CMD);

    /* Disable the receiver and transmitter. */
    outw(RxDisable, ioaddr + EL3_CMD);
    outw(TxDisable, ioaddr + EL3_CMD);

    if (dev->if_port == XCVR_10base2)
        /* Turn off thinnet power.  Green! */
        outw(StopCoax, ioaddr + EL3_CMD);

#ifdef SA_SHIRQ
    free_irq(dev->irq, dev);
#else
    free_irq(dev->irq);
    irq2dev_map[dev->irq] = 0;
#endif

    outw(SetIntrEnb | 0x0000, ioaddr + EL3_CMD);

    update_stats(ioaddr, dev);
    if (vp->full_bus_master_rx) { /* Free Boomerang bus master Rx buffers. */
        outl(0, ioaddr + UpListPtr);
        for (i = 0; i < RX_RING_SIZE; i++)
            if (vp->rx_skbuff[i]) {
#if LINUX_VERSION_CODE < 0x20100
                vp->rx_skbuff[i]->free = 1;
#endif
                DEV_FREE_SKB(vp->rx_skbuff[i]);
                vp->rx_skbuff[i] = 0;
            }
    }
    if (vp->full_bus_master_tx) { /* Free Boomerang bus master Tx buffers. */
        outl(0, ioaddr + DownListPtr);
        for (i = 0; i < TX_RING_SIZE; i++)
            if (vp->tx_skbuff[i]) {
                DEV_FREE_SKB(vp->tx_skbuff[i]);
                vp->tx_skbuff[i] = 0;
            }
    }

    MOD_DEC_USE_COUNT;

    return 0;
}

static struct enet_statistics *
vortex_get_stats(struct device *dev)
{
    struct vortex_private *vp = (struct vortex_private *)dev->priv;
    unsigned long flags;

    if (dev->start) {
        save_flags(flags);
        cli();
        update_stats(dev->base_addr, dev);
        restore_flags(flags);
    }
    return &vp->stats;
}

/*  Update statistics.
    Unlike with the EL3 we need not worry about interrupts changing
    the window setting from underneath us, but we must still guard
    against a race condition with a StatsUpdate interrupt updating the
    table.  This is done by checking that the ASM (!) code generated uses
    atomic updates with '+='.
    */
static void update_stats(int ioaddr, struct device *dev)
{
    struct vortex_private *vp = (struct vortex_private *)dev->priv;

    /* Unlike the 3c5x9 we need not turn off stats updates while reading. */
    /* Switch to the stats window, and read everything. */
    EL3WINDOW(6);
    vp->stats.tx_carrier_errors         += inb(ioaddr + 0);
    vp->stats.tx_heartbeat_errors       += inb(ioaddr + 1);
    /* Multiple collisions. */          inb(ioaddr + 2);
    vp->stats.collisions                += inb(ioaddr + 3);
    vp->stats.tx_window_errors          += inb(ioaddr + 4);
    vp->stats.rx_fifo_errors            += inb(ioaddr + 5);
    vp->stats.tx_packets                += inb(ioaddr + 6);
    vp->stats.tx_packets                += (inb(ioaddr + 9)&0x30) << 4;
    /* Rx packets     */                   inb(ioaddr + 7);   /* Must read to clear */
```

```
	/* Tx deferrals */			inb(ioaddr + 8);
	/* Don't bother with register 9, an extension of registers 6&7.
	   If we do use the 6&7 values the atomic update assumption above
	   is invalid. */
	inw(ioaddr + 10);	/* Total Rx and Tx octets. */
	inw(ioaddr + 12);
	/* New: On the Vortex we must also clear the BadSSD counter. */
	EL3WINDOW(4);
	inb(ioaddr + 12);

	/* We change back to window 7 (not 1) with the Vortex. */
	EL3WINDOW(7);
	return;
}

#ifdef HAVE_PRIVATE_IOCTL
static int vortex_ioctl(struct device *dev, struct ifreq *rq, int cmd)
{
	struct vortex_private *vp = (struct vortex_private *)dev->priv;
	int ioaddr = dev->base_addr;
	u16 *data = (u16 *)&rq->ifr_data;
	int phy = vp->phys[0] & 0x1f;

	if (vortex_debug > 2)
		printk(KERN_DEBUG "%s: In ioct(%-.6s, %#4.4x) %4.4x %4.4x %4.4x %4.4x.\n",
			   dev->name, rq->ifr_ifrn.ifrn_name, cmd,
			   data[0], data[1], data[2], data[3]);

    switch(cmd) {
	case SIOCDEVPRIVATE:		/* Get the address of the PHY in use. */
		data[0] = phy;
	case SIOCDEVPRIVATE+1:		/* Read the specified MII register. */
		EL3WINDOW(4);
		data[3] = mdio_read(ioaddr, data[0] & 0x1f, data[1] & 0x1f);
		return 0;
	case SIOCDEVPRIVATE+2:		/* Write the specified MII register */
		if (!suser())
			return -EPERM;
		mdio_write(ioaddr, data[0] & 0x1f, data[1] & 0x1f, data[2]);
		return 0;
	default:
		return -EOPNOTSUPP;
	}
}
#endif  /* HAVE_PRIVATE_IOCTL */

/* This new version of set_rx_mode() supports v1.4 kernels.
   The Vortex chip has no documented multicast filter, so the only
   multicast setting is to receive all multicast frames.  At least
   the chip has a very clean way to set the mode, unlike many others. */
static void
set_rx_mode(struct device *dev)
{
	int ioaddr = dev->base_addr;
	int new_mode;

	if (dev->flags & IFF_PROMISC) {
		if (vortex_debug > 0)
			printk(KERN_NOTICE "%s: Setting promiscuous mode.\n", dev->name);
		new_mode = SetRxFilter|RxStation|RxMulticast|RxBroadcast|RxProm;
	} else	if ((dev->mc_list)  ||  (dev->flags & IFF_ALLMULTI)) {
		new_mode = SetRxFilter|RxStation|RxMulticast|RxBroadcast;
	} else
		new_mode = SetRxFilter | RxStation | RxBroadcast;

	outw(new_mode, ioaddr + EL3_CMD);
}
#ifndef NEW_MULTICAST
/* The old interface to set the Rx mode. */
static void
set_multicast_list(struct device *dev, int num_addrs, void *addrs)
{
	set_rx_mode(dev);
}
#endif

/* MII transceiver control section.
   Read and write the MII registers using software-generated serial
   MDIO protocol.  See the MII specifications or DP83840A data sheet
   for details. */

/* The maximum data clock rate is 2.5 Mhz.  The minimum timing is usually
   met by back-to-back PCI I/O cycles, but we insert a delay to avoid
   "overclocking" issues. */
#define mdio_delay() udelay(1)

#define MDIO_SHIFT_CLK	0x01
#define MDIO_DIR_WRITE	0x04
#define MDIO_DATA_WRITE0 (0x00 | MDIO_DIR_WRITE)
#define MDIO_DATA_WRITE1 (0x02 | MDIO_DIR_WRITE)
#define MDIO_DATA_READ	0x02
#define MDIO_ENB_IN		0x00
```

```
/* Generate the preamble required for initial synchronization and
   a few older transceivers. */
static void mdio_sync(int ioaddr, int bits)
{
	int mdio_addr = ioaddr + Wn4_PhysicalMgmt;

	/* Establish sync by sending at least 32 logic ones. */
	while (-- bits >= 0) {
		outw(MDIO_DATA_WRITE1, mdio_addr);
		mdio_delay();
		outw(MDIO_DATA_WRITE1 | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
}

static int mdio_read(int ioaddr, int phy_id, int location)
{
	int i;
	int read_cmd = (0xf6 << 10) | (phy_id << 5) | location;
	unsigned int retval = 0;
	int mdio_addr = ioaddr + Wn4_PhysicalMgmt;

	if (mii_preamble_required)
		mdio_sync(ioaddr, 32);

	/* Shift the read command bits out. */
	for (i = 14; i >= 0; i--) {
		int dataval = (read_cmd&(1<<i)) ? MDIO_DATA_WRITE1 : MDIO_DATA_WRITE0;
		outw(dataval, mdio_addr);
		mdio_delay();
		outw(dataval | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	/* Read the two transition, 16 data, and wire-idle bits. */
	for (i = 19; i > 0; i--) {
		outw(MDIO_ENB_IN, mdio_addr);
		mdio_delay();
		retval = (retval << 1) | ((inw(mdio_addr) & MDIO_DATA_READ) ? 1 : 0);
		outw(MDIO_ENB_IN | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	return retval>>1 & 0xffff;
}

static void mdio_write(int ioaddr, int phy_id, int location, int value)
{
	int write_cmd = 0x50020000 | (phy_id << 23) | (location << 18) | value;
	int mdio_addr = ioaddr + Wn4_PhysicalMgmt;
	int i;

	if (mii_preamble_required)
		mdio_sync(ioaddr, 32);

	/* Shift the command bits out. */
	for (i = 31; i >= 0; i--) {
		int dataval = (write_cmd&(1<<i)) ? MDIO_DATA_WRITE1 : MDIO_DATA_WRITE0;
		outw(dataval, mdio_addr);
		mdio_delay();
		outw(dataval | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	/* Leave the interface idle. */
	for (i = 1; i >= 0; i--) {
		outw(MDIO_ENB_IN, mdio_addr);
		mdio_delay();
		outw(MDIO_ENB_IN | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}

	return;
}

#ifdef MODULE
void
cleanup_module(void)
{
	struct device *next_dev;

#ifdef CARDBUS
	unregister_driver(&vortex_ops);
#endif
	/*ZAK */
	del_timer(&z_timer);
	printk("3COM 3c590 Saying good by\n");
	if( z_total_num_interrupts != 0 && z_total_num_polls != 0)
		printk("STAT: Time:%lu, num_inter=%lu, num_poll=%lu, num_unsucc_poll=%lu, Avg Int Latency=%lu, Avg Poll Latency=%lu\n",
		jiffies, z_total_num_interrupts, z_total_num_polls,
		z_total_num_unsucc_polls, z_total_int_latency*100/z_total_num_interrupts,
		z_total_poll_latency*100/z_total_num_polls);

	/* No need to check MOD_IN_USE, as sys_delete_module() checks. */
```

```
	while (root_vortex_dev) {
		next_dev = ((struct vortex_private *)root_vortex_dev->priv)->next_module;
		unregister_netdev(root_vortex_dev);
		outw(TotalReset, root_vortex_dev->base_addr + EL3_CMD);
		release_region(root_vortex_dev->base_addr, VORTEX_TOTAL_SIZE);
		kfree(root_vortex_dev);
		root_vortex_dev = next_dev;
	}
}

#endif  /* MODULE */
/*
 * Local variables:
 *  compile-command: "gcc -DMODULE -D__KERNEL__ -Wall -Wstrict-prototypes -O6 -c 3c59x.c `[ -f /usr/include/linux/modversions.h ] && echo -DMODVERSIONS`"
 *  SMP-compile-command: "gcc -D__SMP__ -DMODULE -D__KERNEL__ -Wall -Wstrict-prototypes -O6 -c 3c59x.c"
 *  compile-command-alt1: "gcc -DCARDBUS -DMODULE -D__KERNEL__ -Wall -Wstrict-prototypes -O6 -c 3c59x.c -o 3c59x_cb.o"
 *  c-indent-level: 4
 *  c-basic-offset: 4
 *  tab-width: 4
 * End:
 */
```