```
/* tulip.c: A DEC 21040 family ethernet driver for Linux. */
/*
      Written/copyright 1994-2003 by Donald Becker.

      This software may be used and distributed according to the terms of
      the GNU General Public License (GPL), incorporated herein by reference.
      Drivers based on or derived from this code fall under the GPL and must
      retain the authorship, copyright and license notice.  This file is not
      a complete program and may only be used when the entire operating
      system is licensed under the GPL.

      This driver is for the Digital "Tulip" Ethernet adapter interface.
      It should work with most DEC 21*4*-based chips/ethercards, as well as
      with work-alike chips from Lite-On (PNIC) and Macronix (MXIC) and ASIX.

      The author may be reached as becker@scyld.com, or C/O
      Scyld Computing Corporation
      914 Bay Ridge Road, Suite 220
      Annapolis MD 21403

      Support and updates available at
      http://www.scyld.com/network/tulip.html

      10/26/04 (STMicro):  Added line 1577 to clear EEPROM Chip Select
*/

/* These identify the driver base version and may not be removed. */
static const char version1[] =
"tulip.c:v0.97 7/22/2003  Written by Donald Becker <becker@scyld.com>\n";
static const char version2[] =
"  http://www.scyld.com/network/tulip.html\n";

#define SMP_CHECK

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

static int debug = 2;               /* Message enable: 0..31 = no..all messages.
*/

/* Maximum events (Rx packets, etc.) to handle at each interrupt. */
static int max_interrupt_work = 25;

#define MAX_UNITS 8
/* Used to pass the full-duplex flag, etc. */
static int full_duplex[MAX_UNITS] = {0, };
static int options[MAX_UNITS] = {0, };
static int mtu[MAX_UNITS] = {0, };              /* Jumbo MTU for interfaces. */

/*  The possible media types that can be set in options[] are: */
#define MEDIA_MASK 31
static const char * const medianame[32] = {
      "10baseT", "10base2", "AUI", "100baseTx",
      "10baseT-FDX", "100baseTx-FDX", "100baseT4", "100baseFx",
      "100baseFx-FDX", "MII 10baseT", "MII 10baseT-FDX", "MII",
      "10baseT(forced)", "MII 100baseTx", "MII 100baseTx-FDX", "MII 100baseT4",
      "MII 100baseFx-HDX", "MII 100baseFx-FDX", "Home-PNA 1Mbps", "Invalid-19",
```

```
      "","","","", "","","","",  "","","","Transceiver reset",
};

/* Set if the PCI BIOS detects the chips on a multiport board backwards. */
#ifdef REVERSE_PROBE_ORDER
static int reverse_probe = 1;
#else
static int reverse_probe = 0;
#endif

/* Set the copy breakpoint for the copy-only-tiny-buffer Rx structure. */
#ifdef __alpha__                        /* Always copy to aligned IP headers. */
static int rx_copybreak = 1518;
#else
static int rx_copybreak = 100;
#endif

/*
  Set the bus performance register.
      Typical: Set 16 longword cache alignment, no burst limit.
      Cache alignment bits 15:14         Burst length 13:8
            0000  No alignment  0x00000000 unlimited        0800 8 longwords
            4000  8  longwords            0100 1 longword         1000 16
longwords
            8000  16 longwords            0200 2 longwords  2000 32 longwords
            C000  32  longwords           0400 4 longwords
      Warning: many older 486 systems are broken and require setting 0x00A04800
         8 longword cache alignment, 8 longword burst.
      ToDo: Non-Intel setting could be better.
*/

#if defined(__alpha__) || defined(__x86_64) || defined(__ia64)
static int csr0 = 0x01A00000 | 0xE000;
#elif defined(__i386__) || defined(__powerpc__) || defined(__sparc__)
/* Do *not* rely on hardware endian correction for big-endian machines! */
static int csr0 = 0x01A00000 | 0x8000;
#else
#warning Processor architecture undefined!
static int csr0 = 0x00A00000 | 0x4800;
#endif

/* Maximum number of multicast addresses to filter (vs. rx-all-multicast).
   Typical is a 64 element hash table based on the Ethernet CRC.
   This value does not apply to the 512 bit table chips.
*/
static int multicast_filter_limit = 32;

/* Operational parameters that are set at compile time. */

/* Keep the descriptor ring sizes a power of two for efficiency.
   The Tx queue length limits transmit packets to a portion of the available
   ring entries.  It should be at least one element less to allow multicast
   filter setup frames to be queued.  It must be at least four for hysteresis.
   Making the Tx queue too long decreases the effectiveness of channel
   bonding and packet priority.
   Large receive rings waste memory and confound network buffer limits.
   These values have been carefully studied: changing these might mask a
```

```
   problem, it won't fix it.
*/
#define TX_RING_SIZE    16
#define TX_QUEUE_LEN    10
#define RX_RING_SIZE    32

/* Operational parameters that usually are not changed. */
/* Time in jiffies before concluding the transmitter is hung. */
#define TX_TIMEOUT  (6*HZ)
/* Preferred skbuff allocation size. */
#define PKT_BUF_SZ            1536
/* This is a mysterious value that can be written to CSR11 in the 21040 (only)
   to support a pre-NWay full-duplex signaling mechanism using short frames.
   No one knows what it should be, but if left at its default value some
   10base2(!) packets trigger a full-duplex-request interrupt. */
#define FULL_DUPLEX_MAGIC     0x6969

/* The include file section.  We start by doing checks and fix-ups for
   missing compile flags. */
#ifndef __KERNEL__
#define __KERNEL__
#endif
#if !defined(__OPTIMIZE__)
#warning  You must compile this file with the correct options!
#warning  See the last lines of the source file.
#error You must compile this driver with "-O".
#endif

#include <linux/config.h>
#if defined(CONFIG_SMP)  &&  ! defined(__SMP__)
#define __SMP__
#endif
#if defined(CONFIG_MODVERSIONS) && defined(MODULE) && ! defined(MODVERSIONS)
#define MODVERSIONS
#endif

#include <linux/version.h>
#if defined(MODVERSIONS)
#include <linux/modversions.h>
#endif
#include <linux/module.h>


#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/timer.h>
#include <linux/errno.h>
#include <linux/ioport.h>
#if LINUX_VERSION_CODE >= 0x20400
#include <linux/slab.h>
#else
#include <linux/malloc.h>
#endif
#include <linux/interrupt.h>
#include <linux/pci.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
```

```
#include <linux/skbuff.h>
#include <asm/processor.h>          /* Processor type for cache alignment. */
#include <asm/bitops.h>
#include <asm/io.h>
#include <asm/unaligned.h>

#ifdef INLINE_PCISCAN
#include "k_compat.h"
#else
#include "pci-scan.h"
#include "kern_compat.h"
#endif

/* Condensed operations for readability. */
#define virt_to_le32desc(addr)  cpu_to_le32(virt_to_bus(addr))

#if (LINUX_VERSION_CODE >= 0x20100)  &&  defined(MODULE)
char kernel_version[] = UTS_RELEASE;
#endif

MODULE_AUTHOR("Donald Becker <becker@scyld.com>");
MODULE_DESCRIPTION("Digital 21*4* Tulip ethernet driver");
MODULE_LICENSE("GPL");
MODULE_PARM(debug, "i");
MODULE_PARM(max_interrupt_work, "i");
MODULE_PARM(reverse_probe, "i");
MODULE_PARM(rx_copybreak, "i");
MODULE_PARM(csr0, "i");
MODULE_PARM(options, "1-" __MODULE_STRING(MAX_UNITS) "i");
MODULE_PARM(full_duplex, "1-" __MODULE_STRING(MAX_UNITS) "i");
MODULE_PARM(multicast_filter_limit, "i");
#ifdef MODULE_PARM_DESC
MODULE_PARM_DESC(debug, "Tulip driver message level (0-31)");
MODULE_PARM_DESC(options,
                         "Tulip: force transceiver type or fixed speed+duplex");
MODULE_PARM_DESC(max_interrupt_work,
                         "Tulip driver maximum events handled per interrupt");
MODULE_PARM_DESC(full_duplex, "Tulip: non-zero to set forced full duplex.");
MODULE_PARM_DESC(rx_copybreak,
                         "Tulip breakpoint in bytes for copy-only-tiny-frames");
MODULE_PARM_DESC(multicast_filter_limit,
                         "Tulip breakpoint for switching to Rx-all-multicast");
MODULE_PARM_DESC(reverse_probe, "Search PCI devices in reverse order to work "
                         "around misordered multiport NICS.");
MODULE_PARM_DESC(csr0, "Special setting for the CSR0 PCI bus parameter "
                         "register.");
#endif

/* This driver was originally written to use I/O space access, but now
   uses memory space by default. Override this this with -DUSE_IO_OPS. */
#if (LINUX_VERSION_CODE < 0x20100)  ||  ! defined(MODULE)
#define USE_IO_OPS
#endif
#ifndef USE_IO_OPS
#undef inb
#undef inw
#undef inl
```

```
#undef outb
#undef outw
#undef outl
#define inb readb
#define inw readw
#define inl readl
#define outb writeb
#define outw writew
#define outl writel
#endif

/*
                        Theory of Operation

I. Board Compatibility

This device driver is designed for the DECchip "Tulip", Digital's
single-chip ethernet controllers for PCI.  Supported members of the family
are the 21040, 21041, 21140, 21140A, 21142, and 21143.  Similar work-alike
chips from Lite-On, Macronics, ASIX, Compex and other listed below are also
supported.

These chips are used on at least 140 unique PCI board designs.  The great
number of chips and board designs supported is the reason for the
driver size and complexity.  Almost of the increasing complexity is in the
board configuration and media selection code.  There is very little
increasing in the operational critical path length.

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 preferably should assign the
PCI INTA signal to an otherwise unused system IRQ line.

Some boards have EEPROMs tables with default media entry.  The factory default
is usually "autoselect".  This should only be overridden when using
transceiver connections without link beat e.g. 10base2 or AUI, or (rarely!)
for forcing full-duplex when used with old link partners that do not do
autonegotiation.

III. Driver operation

IIIa. Ring buffers

The Tulip can use either ring buffers or lists of Tx and Rx descriptors.
This driver uses statically allocated rings of Rx and Tx descriptors, set at
compile time by RX/TX_RING_SIZE.  This version of the driver allocates skbuffs
for the Rx ring buffers at open() time and passes the skb->data field to the
Tulip as receive data buffers.  When an incoming frame is less than
RX_COPYBREAK bytes long, a fresh skbuff is allocated and the frame is
copied to the new skbuff.  When the incoming frame is larger, the skbuff is
passed directly up the protocol stack and replaced by a newly allocated
skbuff.

The RX_COPYBREAK value is chosen to trade-off the memory wasted by
using a full-sized skbuff for small frames vs. the copying costs of larger
frames.  For small frames the copying cost is negligible (esp. considering
```

that we are pre-loading the cache with immediately useful header information). For large frames the copying cost is non-trivial, and the larger copy might flush the cache of useful data. A subtle aspect of this choice is that the Tulip only receives into longword aligned buffers, thus the IP header at offset 14 is not longword aligned for further processing. Copied frames are put into the new skbuff at an offset of "+2", thus copying has the beneficial effect of aligning the IP header and preloading the cache.

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.

The send packet thread has partial control over the Tx ring and 'dev->tbusy' flag. It sets the tbusy flag whenever it is queuing a Tx packet. If the next queue slot is empty, it clears the tbusy flag when finished otherwise it sets the 'tp->tx_full' flag.

The interrupt handler has exclusive control over the Rx ring and records stats from the Tx ring. (The Tx-done interrupt can not be selectively turned off, so we cannot avoid the interrupt overhead by having the Tx routine reap the Tx stats.) After reaping the stats, it marks the queue entry as empty by setting the 'base' to zero. Iff the 'tp->tx_full' flag is set, it clears both the tx_full and tbusy flags.

IV. Notes

Thanks to Duke Kamstra of SMC for long ago providing an EtherPower board.
Greg LaPolla at Linksys provided PNIC and other Linksys boards.
Znyx provided a four-port card for testing.

IVb. References

http://scyld.com/expert/NWay.html
http://www.digital.com (search for current 21*4* datasheets and "21X4 SROM")
http://www.national.com/pf/DP/DP83840A.html
http://www.asix.com.tw/pmac.htm
http://www.admtek.com.tw/

IVc. Errata

The old DEC databooks were light on details.
The 21040 databook claims that CSR13, CSR14, and CSR15 should each be the last register of the set CSR12-15 written. Hmmm, now how is that possible?

The DEC SROM format is very badly designed not precisely defined, leading to part of the media selection junkheap below. Some boards do not have EEPROM media tables and need to be patched up. Worse, other boards use the DEC design kit media table when it is not correct for their design.

We cannot use MII interrupts because there is no defined GPIO pin to attach them. The MII transceiver status is polled using an kernel timer.

*/

```
static void *tulip_probe1(struct pci_dev *pdev, void *init_dev,
                                      long ioaddr, int irq, int chip_idx, int
find_cnt);
static int tulip_pwr_event(void *dev_instance, int event);

#ifdef USE_IO_OPS
#define TULIP_IOTYPE  PCI_USES_MASTER | PCI_USES_IO | PCI_ADDR0
#define TULIP_SIZE 0x80
#define TULIP_SIZE1 0x100
#else
#define TULIP_IOTYPE  PCI_USES_MASTER | PCI_USES_MEM | PCI_ADDR1
#define TULIP_SIZE   0x400          /* New PCI v2.1 recommends 4K min mem size.
*/
#define TULIP_SIZE1     0x400       /* New PCI v2.1 recommends 4K min mem size.
*/
#endif

/* This much match tulip_tbl[]!  Note 21142 == 21143. */
enum tulip_chips {
      DC21040=0, DC21041=1, DC21140=2, DC21142=3, DC21143=3,
      LC82C168, MX98713, MX98715, MX98725, AX88141, AX88140, PNIC2, COMET,
      COMPEX9881, I21145, XIRCOM, CONEXANT,
      /* These flags may be added to the chip type. */
      HAS_VLAN=0x100,
};

static struct pci_id_info pci_id_tbl[] = {
      { "Digital DC21040 Tulip", { 0x00021011, 0xffffffff },
        TULIP_IOTYPE, 0x80, DC21040 },
      { "Digital DC21041 Tulip", { 0x00141011, 0xffffffff },
        TULIP_IOTYPE, 0x80, DC21041 },
      { "Digital DS21140A Tulip", { 0x00091011, 0xffffffff, 0,0, 0x20,0xf0 },
        TULIP_IOTYPE, 0x80, DC21140 },
      { "Digital DS21140 Tulip", { 0x00091011, 0xffffffff },
        TULIP_IOTYPE, 0x80, DC21140 },
      { "Digital DS21143-xD Tulip", { 0x00191011, 0xffffffff, 0,0, 0x40,0xf0 },
        TULIP_IOTYPE, TULIP_SIZE, DC21142 | HAS_VLAN },
      { "Digital DS21143-xC Tulip", { 0x00191011, 0xffffffff, 0,0, 0x30,0xf0 },
        TULIP_IOTYPE, TULIP_SIZE, DC21142 },
      { "Digital DS21142 Tulip", { 0x00191011, 0xffffffff },
        TULIP_IOTYPE, TULIP_SIZE, DC21142 },
      { "Kingston KNE110tx (PNIC)",
        { 0x000211AD, 0xffffffff, 0xf0022646, 0xffffffff },
        TULIP_IOTYPE, 256, LC82C168 },
      { "Linksys LNE100TX (82c168 PNIC)",                    /*  w/SYM */
        { 0x000211AD, 0xffffffff, 0xffff11ad, 0xffffffff, 17,0xff },
        TULIP_IOTYPE, 256, LC82C168 },
      { "Linksys LNE100TX (82c169 PNIC)",                    /* w/ MII */
        { 0x000211AD, 0xffffffff, 0xf00311ad, 0xffffffff, 32,0xff },
        TULIP_IOTYPE, 256, LC82C168 },
      { "Lite-On 82c168 PNIC", { 0x000211AD, 0xffffffff },
        TULIP_IOTYPE, 256, LC82C168 },
      { "Macronix 98713 PMAC", { 0x051210d9, 0xffffffff },
        TULIP_IOTYPE, 256, MX98713 },
      { "Macronix 98715 PMAC", { 0x053110d9, 0xffffffff },
        TULIP_IOTYPE, 256, MX98715 },
      { "Macronix 98725 PMAC", { 0x053110d9, 0xffffffff },
```

```
  TULIP_IOTYPE, 256, MX98725 },
{ "ASIX AX88141", { 0x1400125B, 0xffffffff, 0,0, 0x10, 0xf0 },
  TULIP_IOTYPE, 128, AX88141 },
{ "ASIX AX88140", { 0x1400125B, 0xffffffff },
  TULIP_IOTYPE, 128, AX88140 },
{ "Lite-On LC82C115 PNIC-II", { 0xc11511AD, 0xffffffff },
  TULIP_IOTYPE, 256, PNIC2 },
{ "ADMtek AN981 Comet", { 0x09811317, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-P", { 0x09851317, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C", { 0x19851317, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "D-Link DFE-680TXD v1.0 (ADMtek Centaur-C)", { 0x15411186, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C (Linksys v2)", { 0xab0213d1, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C (Linksys)", { 0xab0313d1, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C (Linksys)", { 0xab0813d1, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C (Linksys PCM200 v3)", { 0xab081737, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Centaur-C (Linksys PCM200 v3)", { 0xab091737, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "STMicro STE10/100 Comet", { 0x0981104a, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "STMicro STE10/100A Comet", { 0x2774104a, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Comet-II", { 0x95111317, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Comet-II (9513)", { 0x95131317, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "SMC1255TX (ADMtek Comet)",
  { 0x12161113, 0xffffffff, 0x125510b8, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "Accton EN1217/EN2242 (ADMtek Comet)", { 0x12161113, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "SMC1255TX (ADMtek Comet-II)", { 0x125510b8, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "ADMtek Comet-II (model 1020)", { 0x1020111a, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "Allied Telesyn A120 (ADMtek Comet)", { 0xa1201259, 0xffffffff },
  TULIP_IOTYPE, TULIP_SIZE1, COMET },
{ "Compex RL100-TX", { 0x988111F6, 0xffffffff },
  TULIP_IOTYPE, 128, COMPEX9881 },
{ "Intel 21145 Tulip", { 0x00398086, 0xffffffff },
  TULIP_IOTYPE, 128, I21145 },
{ "Xircom Tulip clone", { 0x0003115d, 0xffffffff },
  TULIP_IOTYPE, 128, XIRCOM },
{ "Davicom DM9102", { 0x91021282, 0xffffffff },
  TULIP_IOTYPE, 0x80, DC21140 },
{ "Davicom DM9100", { 0x91001282, 0xffffffff },
  TULIP_IOTYPE, 0x80, DC21140 },
{ "Macronix mxic-98715 (EN1217)", { 0x12171113, 0xffffffff },
  TULIP_IOTYPE, 256, MX98715 },
{ "Conexant LANfinity", { 0x180314f1, 0xffffffff },
```

```
        TULIP_IOTYPE, TULIP_SIZE1, CONEXANT },
      { "3Com 3cSOHO100B-TX (ADMtek Centaur)", { 0x930010b7, 0xffffffff },
        TULIP_IOTYPE, TULIP_SIZE1, COMET },
      { 0},
};

struct drv_id_info tulip_drv_id = {
      "tulip", PCI_HOTSWAP, PCI_CLASS_NETWORK_ETHERNET<<8, pci_id_tbl,
      tulip_probe1, tulip_pwr_event };

/* This table is used during operation for capabilities and media timer. */

static void tulip_timer(unsigned long data);
static void nway_timer(unsigned long data);
static void mxic_timer(unsigned long data);
static void pnic_timer(unsigned long data);
static void comet_timer(unsigned long data);

enum tbl_flag {
      HAS_MII=1, HAS_MEDIA_TABLE=2, CSR12_IN_SROM=4, ALWAYS_CHECK_MII=8,
      HAS_PWRDWN=0x10, MC_HASH_ONLY=0x20, /* Hash-only multicast filter. */
      HAS_PNICNWAY=0x80, HAS_NWAY=0x40,   /* Uses internal NWay xcvr. */
      HAS_INTR_MITIGATION=0x100, IS_ASIX=0x200, HAS_8023X=0x400,
      COMET_MAC_ADDR=0x0800,
};

/* Note: this table must match  enum tulip_chips  above. */
static struct tulip_chip_table {
      char *chip_name;
      int io_size;                        /* Unused */
      int valid_intrs;              /* CSR7 interrupt enable settings */
      int flags;
      void (*media_timer)(unsigned long data);
} tulip_tbl[] = {
  { "Digital DC21040 Tulip", 128, 0x0001ebef, 0, tulip_timer },
  { "Digital DC21041 Tulip", 128, 0x0001ebff,
      HAS_MEDIA_TABLE | HAS_NWAY, tulip_timer },
  { "Digital DS21140 Tulip", 128, 0x0001ebef,
      HAS_MII | HAS_MEDIA_TABLE | CSR12_IN_SROM, tulip_timer },
  { "Digital DS21143 Tulip", 128, 0x0801fbff,
      HAS_MII | HAS_MEDIA_TABLE | ALWAYS_CHECK_MII | HAS_PWRDWN | HAS_NWAY
      | HAS_INTR_MITIGATION, nway_timer },
  { "Lite-On 82c168 PNIC", 256, 0x0001ebef,
      HAS_MII | HAS_PNICNWAY, pnic_timer },
  { "Macronix 98713 PMAC", 128, 0x0001ebef,
      HAS_MII | HAS_MEDIA_TABLE | CSR12_IN_SROM, mxic_timer },
  { "Macronix 98715 PMAC", 256, 0x0001ebef,
      HAS_MEDIA_TABLE, mxic_timer },
  { "Macronix 98725 PMAC", 256, 0x0001ebef,
      HAS_MEDIA_TABLE, mxic_timer },
  { "ASIX AX88140", 128, 0x0001fbff,
      HAS_MII | HAS_MEDIA_TABLE | CSR12_IN_SROM | MC_HASH_ONLY | IS_ASIX,
tulip_timer },
  { "ASIX AX88141", 128, 0x0001fbff,
      HAS_MII | HAS_MEDIA_TABLE | CSR12_IN_SROM | MC_HASH_ONLY | IS_ASIX,
tulip_timer },
  { "Lite-On PNIC-II", 256, 0x0801fbff,
```

```
      HAS_MII | HAS_NWAY | HAS_8023X, nway_timer },
  { "ADMtek Comet", 256, 0x0001abef,
      HAS_MII | MC_HASH_ONLY | COMET_MAC_ADDR, comet_timer },
  { "Compex 9881 PMAC", 128, 0x0001ebef,
      HAS_MII | HAS_MEDIA_TABLE | CSR12_IN_SROM, mxic_timer },
  { "Intel DS21145 Tulip", 128, 0x0801fbff,
      HAS_MII | HAS_MEDIA_TABLE | ALWAYS_CHECK_MII | HAS_PWRDWN | HAS_NWAY,
      nway_timer },
  { "Xircom tulip work-alike", 128, 0x0801fbff,
      HAS_MII | HAS_MEDIA_TABLE | ALWAYS_CHECK_MII | HAS_PWRDWN | HAS_NWAY,
      nway_timer },
  { "Conexant LANfinity", 256, 0x0001ebef,
      HAS_MII | HAS_PWRDWN, tulip_timer },
  {0},
};

/* A full-duplex map for media types. */
enum MediaIs {
      MediaIsFD = 1, MediaAlwaysFD=2, MediaIsMII=4, MediaIsFx=8,
      MediaIs100=16};
static const char media_cap[32] =
{0,0,0,16,  3,19,16,24,  27,4,7,5, 0,20,23,20,  28,31,0,0, };
static u8 t21040_csr13[] = {2,0x0C,8,4,  4,0,0,0, 0,0,0,0, 4,0,0,0};

/* 21041 transceiver register settings: 10-T, 10-2, AUI, 10-T, 10T-FD*/
static u16 t21041_csr13[] = { 0xEF01, 0xEF09, 0xEF09, 0xEF01, 0xEF09, };
static u16 t21041_csr14[] = { 0xFFFF, 0xF7FD, 0xF7FD, 0x6F3F, 0x6F3D, };
static u16 t21041_csr15[] = { 0x0008, 0x0006, 0x000E, 0x0008, 0x0008, };

static u16 t21142_csr13[] = { 0x0001, 0x0009, 0x0009, 0x0000, 0x0001, };
static u16 t21142_csr14[] = { 0xFFFF, 0x0705, 0x0705, 0x0000, 0x7F3D, };
static u16 t21142_csr15[] = { 0x0008, 0x0006, 0x000E, 0x0008, 0x0008, };

/* Offsets to the Command and Status Registers, "CSRs".  All accesses
   must be longword instructions and quadword aligned. */
enum tulip_offsets {
      CSR0=0,    CSR1=0x08, CSR2=0x10, CSR3=0x18, CSR4=0x20, CSR5=0x28,
      CSR6=0x30, CSR7=0x38, CSR8=0x40, CSR9=0x48, CSR10=0x50, CSR11=0x58,
      CSR12=0x60, CSR13=0x68, CSR14=0x70, CSR15=0x78 };

/* The bits in the CSR5 status registers, mostly interrupt sources. */
enum status_bits {
      TimerInt=0x800, TPLnkFail=0x1000, TPLnkPass=0x10,
      NormalIntr=0x10000, AbnormalIntr=0x8000, PCIBusError=0x2000,
      RxJabber=0x200, RxStopped=0x100, RxNoBuf=0x80, RxIntr=0x40,
      TxFIFOUnderflow=0x20, TxJabber=0x08, TxNoBuf=0x04, TxDied=0x02,
TxIntr=0x01,
};

/* The configuration bits in CSR6. */
enum csr6_mode_bits {
      TxOn=0x2000, RxOn=0x0002, FullDuplex=0x0200,
      AcceptBroadcast=0x0100, AcceptAllMulticast=0x0080,
      AcceptAllPhys=0x0040, AcceptRunt=0x0008,
};
```

```
/* The Tulip Rx and Tx buffer descriptors. */
struct tulip_rx_desc {
      s32 status;
      s32 length;
      u32 buffer1, buffer2;
};

struct tulip_tx_desc {
      s32 status;
      s32 length;
      u32 buffer1, buffer2;                     /* We use only buffer 1.  */
};

enum desc_status_bits {
      DescOwned=0x80000000, RxDescFatalErr=0x8000, RxWholePkt=0x0300,
};

/* Ring-wrap flag in length field, use for last ring entry.
      0x01000000 means chain on buffer2 address,
      0x02000000 means use the ring start address in CSR2/3.
   Note: Some work-alike chips do not function correctly in chained mode.
   The ASIX chip works only in chained mode.
   Thus we indicates ring mode, but always write the 'next' field for
   chained mode as well.
*/
#define DESC_RING_WRAP 0x02000000

#define EEPROM_SIZE 512       /* support 256*16 EEPROMs */

struct medialeaf {
      u8 type;
      u8 media;
      unsigned char *leafdata;
};

struct mediatable {
      u16 defaultmedia;
      u8 leafcount, csr12dir;                   /* General purpose pin
directions. */
      unsigned has_mii:1, has_nonmii:1, has_reset:6;
      u32 csr15dir, csr15val;                   /* 21143 NWay setting. */
      struct medialeaf mleaf[0];
};

struct mediainfo {
      struct mediainfo *next;
      int info_type;
      int index;
      unsigned char *info;
};

#define PRIV_ALIGN      15    /* Required alignment mask */
struct tulip_private {
      struct tulip_rx_desc rx_ring[RX_RING_SIZE];
      struct tulip_tx_desc tx_ring[TX_RING_SIZE];
      /* The saved addresses of Rx/Tx-in-place packet buffers. */
      struct sk_buff* tx_skbuff[TX_RING_SIZE];
```

```
      struct sk_buff* rx_skbuff[RX_RING_SIZE];
      struct net_device *next_module;
      void *priv_addr;              /* Unaligned address of dev->priv for kfree
*/
      /* Multicast filter control. */
      u16 setup_frame[96];          /* Pseudo-Tx frame to init address table. */
      u32 mc_filter[2];             /* Multicast hash filter */
      int multicast_filter_limit;
      struct pci_dev *pci_dev;
      int chip_id, revision;
      int flags;
      int max_interrupt_work;
      int msg_level;
      unsigned int csr0, csr6;                  /* Current CSR0, CSR6 settings.
*/
      /* Note: cache line pairing and isolation of Rx vs. Tx indicies. */
      unsigned int cur_rx, dirty_rx;            /* Producer/consumer ring
indices */
      unsigned int rx_buf_sz;                   /* Based on MTU+slack. */
      int rx_copybreak;
      unsigned int rx_dead:1;                   /* We have no Rx buffers. */

      struct net_device_stats stats;
      unsigned int cur_tx, dirty_tx;
      unsigned int tx_full:1;                   /* The Tx queue is full. */

      /* Media selection state. */
      unsigned int full_duplex:1;               /* Full-duplex operation
requested. */
      unsigned int full_duplex_lock:1;
      unsigned int fake_addr:1;                 /* Multiport board faked
address. */
      unsigned int media2:4;                    /* Secondary monitored media
port. */
      unsigned int medialock:1;                 /* Do not sense media type. */
      unsigned int mediasense:1;                /* Media sensing in progress. */
      unsigned int nway:1, nwayset:1;           /* 21143 internal NWay. */
      unsigned int default_port;                /* Last dev->if_port value. */
      unsigned char eeprom[EEPROM_SIZE];  /* Serial EEPROM contents. */
      struct timer_list timer;                  /* Media selection timer. */
      void (*link_change)(struct net_device *dev, int csr5);
      u16 lpar;                                     /* 21143 Link partner
ability. */
      u16 sym_advertise, mii_advertise;   /* NWay to-advertise. */
      u16 advertising[4];                           /* MII advertise, from
SROM table. */
      signed char phys[4], mii_cnt;       /* MII device addresses. */
      spinlock_t mii_lock;
      struct mediatable *mtable;
      int cur_index;                                /* Current media index.
*/
      int saved_if_port;
};

static void start_link(struct net_device *dev);
static void parse_eeprom(struct net_device *dev);
static int read_eeprom(long ioaddr, int location, int addr_len);
```

```
static int mdio_read(struct net_device *dev, int phy_id, int location);
static void mdio_write(struct net_device *dev, int phy_id, int location, int
value);
static int tulip_open(struct net_device *dev);
/* Chip-specific media selection (timer functions prototyped above). */
static int  check_duplex(struct net_device *dev);
static void select_media(struct net_device *dev, int startup);
static void init_media(struct net_device *dev);
static void nway_lnk_change(struct net_device *dev, int csr5);
static void nway_start(struct net_device *dev);
static void pnic_lnk_change(struct net_device *dev, int csr5);
static void pnic_do_nway(struct net_device *dev);

static void tulip_tx_timeout(struct net_device *dev);
static void tulip_init_ring(struct net_device *dev);
static int tulip_start_xmit(struct sk_buff *skb, struct net_device *dev);
static int tulip_rx(struct net_device *dev);
static void tulip_interrupt(int irq, void *dev_instance, struct pt_regs *regs);
static int tulip_close(struct net_device *dev);
static struct net_device_stats *tulip_get_stats(struct net_device *dev);
#ifdef HAVE_PRIVATE_IOCTL
static int private_ioctl(struct net_device *dev, struct ifreq *rq, int cmd);
#endif
static void set_rx_mode(struct net_device *dev);
```

```
/* A list of all installed Tulip devices. */
static struct net_device *root_tulip_dev = NULL;

static void *tulip_probe1(struct pci_dev *pdev, void *init_dev,
                                        long ioaddr, int irq, int pci_tbl_idx, int
find_cnt)
{
      struct net_device *dev;
      struct tulip_private *tp;
      void *priv_mem;
      /* See note below on the multiport cards. */
      static unsigned char last_phys_addr[6] = {0x00, 'L', 'i', 'n', 'u', 'x'};
      static int last_irq = 0;
      static int multiport_cnt = 0;       /* For four-port boards w/one EEPROM
*/
      u8 chip_rev;
      int i, chip_idx = pci_id_tbl[pci_tbl_idx].drv_flags & 0xff;
      unsigned short sum;
      u8 ee_data[EEPROM_SIZE];

      /* Bring the 21041/21143 out of sleep mode.
         Caution: Snooze mode does not work with some boards! */
      if (tulip_tbl[chip_idx].flags & HAS_PWRDWN)
            pci_write_config_dword(pdev, 0x40, 0x00000000);

      if (inl(ioaddr + CSR5) == 0xffffffff) {
            printk(KERN_ERR "The Tulip chip at %#lx is not functioning.\n",
ioaddr);
            return 0;
      }

      dev = init_etherdev(init_dev, 0);
      if (!dev)
            return NULL;

      /* Make certain the data structures are quadword aligned. */
      priv_mem = kmalloc(sizeof(*tp) + PRIV_ALIGN, GFP_KERNEL);
      /* Check for the very unlikely case of no memory. */
      if (priv_mem == NULL)
            return NULL;
      dev->priv = tp = (void *)(((long)priv_mem + PRIV_ALIGN) & ~PRIV_ALIGN);
      memset(tp, 0, sizeof(*tp));
      tp->mii_lock = (spinlock_t) SPIN_LOCK_UNLOCKED;
      tp->priv_addr = priv_mem;

      tp->next_module = root_tulip_dev;
      root_tulip_dev = dev;

      pci_read_config_byte(pdev, PCI_REVISION_ID, &chip_rev);

      printk(KERN_INFO "%s: %s rev %d at %#3lx,",
               dev->name, pci_id_tbl[pci_tbl_idx].name, chip_rev, ioaddr);

      /* Stop the Tx and Rx processes. */
      outl(inl(ioaddr + CSR6) & ~TxOn & ~RxOn, ioaddr + CSR6);
```

```
	/* Clear the missed-packet counter. */
	inl(ioaddr + CSR8);

	if (chip_idx == DC21041  &&  inl(ioaddr + CSR9) & 0x8000) {
		printk(" 21040 compatible mode,");
		chip_idx = DC21040;
	}

	/* The SROM/EEPROM interface varies dramatically. */
	sum = 0;
	if (chip_idx == DC21040) {
		outl(0, ioaddr + CSR9);		/* Reset the pointer with a dummy
write. */
		for (i = 0; i < 6; i++) {
			int value, boguscnt = 100000;
			do
				value = inl(ioaddr + CSR9);
			while (value < 0  && --boguscnt > 0);
			dev->dev_addr[i] = value;
			sum += value & 0xff;
		}
	} else if (chip_idx == LC82C168) {
		for (i = 0; i < 3; i++) {
			int value, boguscnt = 100000;
			outl(0x600 | i, ioaddr + 0x98);
			do
				value = inl(ioaddr + CSR9);
			while (value < 0  && --boguscnt > 0);
			put_unaligned(le16_to_cpu(value), ((u16*)dev->dev_addr) + i);
			sum += value & 0xffff;
		}
	} else if (chip_idx == COMET) {
		/* No need to read the EEPROM. */
		put_unaligned(le32_to_cpu(inl(ioaddr + 0xA4)), (u32
*)dev->dev_addr);
		put_unaligned(le16_to_cpu(inl(ioaddr + 0xA8)),
				(u16 *)(dev->dev_addr + 4));
		for (i = 0; i < 6; i ++)
			sum += dev->dev_addr[i];
	} else {
		/* A serial EEPROM interface, we read now and sort it out later. */
		int sa_offset = 0;
		int ee_addr_size = read_eeprom(ioaddr, 0xff, 8) & 0x40000 ? 8 : 6;
		int eeprom_word_cnt = 1 << ee_addr_size;

		for (i = 0; i < eeprom_word_cnt; i++)
			((u16 *)ee_data)[i] =
				le16_to_cpu(read_eeprom(ioaddr, i, ee_addr_size));

		/* DEC now has a specification (see Notes) but early board makers
		   just put the address in the first EEPROM locations. */
		/* This does  memcmp(eedata, eedata+16, 8) */
		for (i = 0; i < 8; i ++)
			if (ee_data[i] != ee_data[16+i])
				sa_offset = 20;
		if (chip_idx == CONEXANT) {
			/* Check that the tuple type and length is correct. */
```

```
                if (ee_data[0x198] == 0x04  &&  ee_data[0x199] == 6)
                        sa_offset = 0x19A;
            } else if (ee_data[0] == 0xff  &&  ee_data[1] == 0xff &&
                           ee_data[2] == 0) {
                sa_offset = 2;          /* Grrr, damn Matrox boards. */
                multiport_cnt = 4;
            }
            for (i = 0; i < 6; i ++) {
                dev->dev_addr[i] = ee_data[i + sa_offset];
                sum += ee_data[i + sa_offset];
            }
        }
        /* Lite-On boards have the address byte-swapped. */
        if ((dev->dev_addr[0] == 0xA0  ||  dev->dev_addr[0] == 0xC0)
            &&  dev->dev_addr[1] == 0x00)
            for (i = 0; i < 6; i+=2) {
                char tmp = dev->dev_addr[i];
                dev->dev_addr[i] = dev->dev_addr[i+1];
                dev->dev_addr[i+1] = tmp;
            }
        /* On the Zynx 315 Etherarray and other multiport boards only the
           first Tulip has an EEPROM.
           The addresses of the subsequent ports are derived from the first.
           Many PCI BIOSes also incorrectly report the IRQ line, so we correct
           that here as well. */
        if (sum == 0  || sum == 6*0xff) {
            printk(" EEPROM not present,");
            for (i = 0; i < 5; i++)
                dev->dev_addr[i] = last_phys_addr[i];
            dev->dev_addr[i] = last_phys_addr[i] + 1;
#if defined(__i386__)           /* Patch up x86 BIOS bug. */
            if (last_irq)
                irq = last_irq;
#endif
        }

        for (i = 0; i < 6; i++)
            printk("%c%2.2X", i ? ':' : ' ', last_phys_addr[i] =
dev->dev_addr[i]);
        printk(", IRQ %d.\n", irq);
        last_irq = irq;

#ifdef USE_IO_OPS
        /* We do a request_region() to register /proc/ioports info. */
        request_region(ioaddr, pci_id_tbl[chip_idx].io_size, dev->name);
#endif

        dev->base_addr = ioaddr;
        dev->irq = irq;

        tp->pci_dev = pdev;
        tp->msg_level = (1 << debug) - 1;
        tp->chip_id = chip_idx;
        tp->revision = chip_rev;
        tp->flags = tulip_tbl[chip_idx].flags
            | (pci_id_tbl[pci_tbl_idx].drv_flags & 0xffffff00);
        tp->rx_copybreak = rx_copybreak;
```

```
	tp->max_interrupt_work = max_interrupt_work;
	tp->multicast_filter_limit = multicast_filter_limit;
	tp->csr0 = csr0;

	/* BugFixes: The 21143-TD hangs with PCI Write-and-Invalidate cycles.
	   And the ASIX must have a burst limit or horrible things happen. */
	if (chip_idx == DC21143  &&  chip_rev == 65)
		tp->csr0 &= ~0x01000000;
	else if (tp->flags & IS_ASIX)
		tp->csr0 |= 0x2000;

	/* We support a zillion ways to set the media type. */
#ifdef TULIP_FULL_DUPLEX
	tp->full_duplex = 1;
	tp->full_duplex_lock = 1;
#endif
#ifdef TULIP_DEFAULT_MEDIA
	tp->default_port = TULIP_DEFAULT_MEDIA;
#endif
#ifdef TULIP_NO_MEDIA_SWITCH
	tp->medialock = 1;
#endif

	/* The lower four bits are the media type. */
	if (find_cnt >= 0  &&  find_cnt < MAX_UNITS) {
		if (options[find_cnt] & 0x1f)
			tp->default_port = options[find_cnt] & 0x1f;
		if ((options[find_cnt] & 0x200) || full_duplex[find_cnt] > 0)
			tp->full_duplex = 1;
		if (mtu[find_cnt] > 0)
			dev->mtu = mtu[find_cnt];
	}
	if (dev->mem_start)
		tp->default_port = dev->mem_start & 0x1f;
	if (tp->default_port) {
		printk(KERN_INFO "%s: Transceiver selection forced to %s.\n",
			   dev->name, medianame[tp->default_port & MEDIA_MASK]);
		tp->medialock = 1;
		if (media_cap[tp->default_port] & MediaAlwaysFD)
			tp->full_duplex = 1;
	}
	if (tp->full_duplex)
		tp->full_duplex_lock = 1;

	if (media_cap[tp->default_port] & MediaIsMII) {
		u16 media2advert[] = { 0x20, 0x40, 0x03e0, 0x60, 0x80, 0x100, 0x200
};
		tp->mii_advertise = media2advert[tp->default_port - 9];
		tp->mii_advertise |= (tp->flags & HAS_8023X); /* Matching bits! */
	}

	/* This is logically part of probe1(), but too complex to write inline. */
	if (tp->flags & HAS_MEDIA_TABLE) {
		memcpy(tp->eeprom, ee_data, sizeof(tp->eeprom));
		parse_eeprom(dev);
	}
```

```
      /* The Tulip-specific entries in the device structure. */
      dev->open = &tulip_open;
      dev->hard_start_xmit = &tulip_start_xmit;
      dev->stop = &tulip_close;
      dev->get_stats = &tulip_get_stats;
#ifdef HAVE_PRIVATE_IOCTL
      dev->do_ioctl = &private_ioctl;
#endif
#ifdef HAVE_MULTICAST
      dev->set_multicast_list = &set_rx_mode;
#endif

      if (tp->flags & HAS_NWAY)
            tp->link_change = nway_lnk_change;
      else if (tp->flags & HAS_PNICNWAY)
            tp->link_change = pnic_lnk_change;
      start_link(dev);
      if (chip_idx == COMET) {
            /* Set the Comet LED configuration. */
            outl(0xf0000000, ioaddr + CSR9);
      }

      return dev;
}

/* Start the link, typically called at probe1() time but sometimes later with
   multiport cards. */
static void start_link(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int i;

      if ((tp->flags & ALWAYS_CHECK_MII) ||
            (tp->mtable  &&  tp->mtable->has_mii) ||
            ( ! tp->mtable  &&  (tp->flags & HAS_MII))) {
            int phyn, phy_idx = 0;
            if (tp->mtable  &&  tp->mtable->has_mii) {
                  for (i = 0; i < tp->mtable->leafcount; i++)
                        if (tp->mtable->mleaf[i].media == 11) {
                              tp->cur_index = i;
                              tp->saved_if_port = dev->if_port;
                              select_media(dev, 2);
                              dev->if_port = tp->saved_if_port;
                              break;
                        }
            }
            /* Find the connected MII xcvrs.
               Doing this in open() would allow detecting external xcvrs later,
               but takes much time. */
            for (phyn = 1; phyn <= 32 && phy_idx < sizeof(tp->phys); phyn++) {
                  int phy = phyn & 0x1f;
                  int mii_status = mdio_read(dev, phy, 1);
                  if ((mii_status & 0x8301) == 0x8001 ||
                        ((mii_status & 0x8000) == 0  && (mii_status & 0x7800) !=
0)) {
                        int mii_reg0 = mdio_read(dev, phy, 0);
```

```
                        int mii_advert = mdio_read(dev, phy, 4);
                        int to_advert;

                        if (tp->mii_advertise)
                              to_advert = tp->mii_advertise;
                        else if (tp->advertising[phy_idx])
                              to_advert = tp->advertising[phy_idx];
                        else                  /* Leave unchanged. */
                              tp->mii_advertise = to_advert = mii_advert;

                        tp->phys[phy_idx++] = phy;
                        printk(KERN_INFO "%s:  MII transceiver #%d "
                                 "config %4.4x status %4.4x advertising
%4.4x.\n",
                                 dev->name, phy, mii_reg0, mii_status,
mii_advert);
                        /* Fixup for DLink with miswired PHY. */
                        if (mii_advert != to_advert) {
                              printk(KERN_DEBUG "%s:  Advertising %4.4x on PHY
%d,"
                                         " previously advertising %4.4x.\n",
                                         dev->name, to_advert, phy, mii_advert);
                              mdio_write(dev, phy, 4, to_advert);
                        }
                        /* Enable autonegotiation: some boards default to off.
*/
                        mdio_write(dev, phy, 0, (mii_reg0 & ~0x3000) |
                                         (tp->full_duplex ? 0x0100 : 0x0000) |
                                         ((media_cap[tp->default_port] &
MediaIs100) ?
                                            0x2000 : 0x1000));
                  }
            }
            tp->mii_cnt = phy_idx;
            if (tp->mtable  &&  tp->mtable->has_mii  &&  phy_idx == 0) {
                  printk(KERN_INFO "%s: ***WARNING***: No MII transceiver
found!\n",
                           dev->name);
                  tp->phys[0] = 1;
            }
      }

      /* Reset the xcvr interface and turn on heartbeat. */
      switch (tp->chip_id) {
      case DC21040:
            outl(0x00000000, ioaddr + CSR13);
            outl(0x00000004, ioaddr + CSR13);
            break;
      case DC21041:
            /* This is nway_start(). */
            if (tp->sym_advertise == 0)
                  tp->sym_advertise = 0x0061;
            outl(0x00000000, ioaddr + CSR13);
            outl(0xFFFFFFFF, ioaddr + CSR14);
            outl(0x00000008, ioaddr + CSR15); /* Listen on AUI also. */
            outl(inl(ioaddr + CSR6) | FullDuplex, ioaddr + CSR6);
            outl(0x0000EF01, ioaddr + CSR13);
```

```
            break;
      case DC21140: default:
            if (tp->mtable)
                  outl(tp->mtable->csr12dir | 0x100, ioaddr + CSR12);
            break;
      case DC21142:
      case PNIC2:
            if (tp->mii_cnt  ||  media_cap[dev->if_port] & MediaIsMII) {
                  outl(0x82020000, ioaddr + CSR6);
                  outl(0x0000, ioaddr + CSR13);
                  outl(0x0000, ioaddr + CSR14);
                  outl(0x820E0000, ioaddr + CSR6);
            } else
                  nway_start(dev);
            break;
      case LC82C168:
            if ( ! tp->mii_cnt) {
                  tp->nway = 1;
                  tp->nwayset = 0;
                  outl(0x00420000, ioaddr + CSR6);
                  outl(0x30, ioaddr + CSR12);
                  outl(0x0001F078, ioaddr + 0xB8);
                  outl(0x0201F078, ioaddr + 0xB8); /* Turn on autonegotiation.
*/
            }
            break;
      case COMPEX9881:
            outl(0x00000000, ioaddr + CSR6);
            outl(0x000711C0, ioaddr + CSR14); /* Turn on NWay. */
            outl(0x00000001, ioaddr + CSR13);
            break;
      case MX98713: case MX98715: case MX98725:
            outl(0x01a80000, ioaddr + CSR6);
            outl(0xFFFFFFFF, ioaddr + CSR14);
            outl(0x00001000, ioaddr + CSR12);
            break;
      case COMET:
            break;
      }

      if (tp->flags & HAS_PWRDWN)
            pci_write_config_dword(tp->pci_dev, 0x40, 0x40000000);
}
```

```
/* Serial EEPROM section. */
/* The main routine to parse the very complicated SROM structure.
   Search www.digital.com for "21X4 SROM" to get details.
   This code is very complex, and will require changes to support
   additional cards, so I will be verbose about what is going on.
   */

/* Known cards that have old-style EEPROMs.
   Writing this table is described at
   http://www.scyld.com/network/tulip-media.html
*/
static struct fixups {
  char *name;
  unsigned char addr0, addr1, addr2;
  u16 newtable[32];                     /* Max length below. */
} eeprom_fixups[] = {
  {"Asante", 0, 0, 0x94, {0x1e00, 0x0000, 0x0800, 0x0100, 0x018c,
                                    0x0000, 0x0000, 0xe078, 0x0001, 0x0050,
0x0018 }},
  {"SMC9332DST", 0, 0, 0xC0, { 0x1e00, 0x0000, 0x0800, 0x041f,
                                          0x0000, 0x009E, /* 10baseT */
                                          0x0004, 0x009E, /* 10baseT-FD */
                                          0x0903, 0x006D, /* 100baseTx */
                                          0x0905, 0x006D, /* 100baseTx-FD */
}},
  {"Cogent EM100", 0, 0, 0x92, { 0x1e00, 0x0000, 0x0800, 0x063f,
                                              0x0107, 0x8021, /* 100baseFx
*/
                                              0x0108, 0x8021, /*
100baseFx-FD */
                                              0x0100, 0x009E, /* 10baseT */
                                              0x0104, 0x009E, /* 10baseT-FD
*/
                                              0x0103, 0x006D, /* 100baseTx
*/
                                              0x0105, 0x006D, /*
100baseTx-FD */ }},
  {"Maxtech NX-110", 0, 0, 0xE8, { 0x1e00, 0x0000, 0x0800, 0x0513,
                                              0x1001, 0x009E, /* 10base2, CSR12
0x10*/
                                              0x0000, 0x009E, /* 10baseT */
                                              0x0004, 0x009E, /* 10baseT-FD */
                                              0x0303, 0x006D, /* 100baseTx,
CSR12 0x03 */
                                              0x0305, 0x006D, /* 100baseTx-FD
CSR12 0x03 */}},
  {"Accton EN1207", 0, 0, 0xE8, { 0x1e00, 0x0000, 0x0800, 0x051F,
                                                0x1B01, 0x0000, /* 10base2,
CSR12 0x1B */
                                                0x0B00, 0x009E, /* 10baseT,
CSR12 0x0B */
                                                0x0B04, 0x009E, /*
10baseT-FD,CSR12 0x0B */
                                                0x1B03, 0x006D, /* 100baseTx,
CSR12 0x1B */
                                                0x1B05, 0x006D, /*
```

```
100baseTx-FD CSR12 0x1B */
   }},
  {0, 0, 0, 0, {}}};

static const char * block_name[] = {"21140 non-MII", "21140 MII PHY",
 "21142 Serial PHY", "21142 MII PHY", "21143 SYM PHY", "21143 reset method"};

#if defined(__i386__)               /* AKA get_unaligned() */
#define get_u16(ptr) (*(u16 *)(ptr))
#else
#define get_u16(ptr) (((u8*)(ptr))[0] + (((u8*)(ptr))[1]<<8))
#endif

static void parse_eeprom(struct net_device *dev)
{
      /* The last media info list parsed, for multiport boards.  */
      static struct mediatable *last_mediatable = NULL;
      static unsigned char *last_ee_data = NULL;
      static int controller_index = 0;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      unsigned char *p, *ee_data = tp->eeprom;
      int new_advertise = 0;
      int i;

      tp->mtable = 0;
      /* Detect an old-style (SA only) EEPROM layout:
         memcmp(eedata, eedata+16, 8). */
      for (i = 0; i < 8; i ++)
            if (ee_data[i] != ee_data[16+i])
                  break;
      if (i >= 8) {
            if (ee_data[0] == 0xff) {
                  if (last_mediatable) {
                        controller_index++;
                        printk(KERN_INFO "%s:  Controller %d of multiport
board.\n",
                                    dev->name, controller_index);
                        tp->mtable = last_mediatable;
                        ee_data = last_ee_data;
                        goto subsequent_board;
                  } else
                        printk(KERN_INFO "%s:  Missing EEPROM, this interface
may "
                                    "not work correctly!\n",
                                    dev->name);
                  return;
            }
            /* Do a fix-up based on the vendor half of the station address. */
            for (i = 0; eeprom_fixups[i].name; i++) {
                  if (dev->dev_addr[0] == eeprom_fixups[i].addr0
                        &&  dev->dev_addr[1] == eeprom_fixups[i].addr1
                        &&  dev->dev_addr[2] == eeprom_fixups[i].addr2) {
                        if (dev->dev_addr[2] == 0xE8  &&  ee_data[0x1a] == 0x55)
                              i++;        /* An Accton EN1207, not an outlaw
Maxtech. */
                        memcpy(ee_data + 26, eeprom_fixups[i].newtable,
                                    sizeof(eeprom_fixups[i].newtable));
```

```
                        printk(KERN_INFO "%s: Old format EEPROM on '%s' board.\n"
                                KERN_INFO "%s: Using substitute media control
info.\n",
                                dev->name, eeprom_fixups[i].name, dev->name);
                        break;
                  }
            }
            if (eeprom_fixups[i].name == NULL) { /* No fixup found. */
                  printk(KERN_INFO "%s: Old style EEPROM with no media selection
"
                           "information.\n",
                           dev->name);
                  return;
            }
      }

      controller_index = 0;
      if (ee_data[19] > 1) {
            struct net_device *prev_dev;
            struct tulip_private *otp;
            /* This is a multiport board.  The probe order may be "backwards",
so
               we patch up already found devices. */
            last_ee_data = ee_data;
            for (prev_dev = tp->next_module; prev_dev; prev_dev =
otp->next_module) {
                  otp = (struct tulip_private *)prev_dev->priv;
                  if (otp->eeprom[0] == 0xff  &&  otp->mtable == 0) {
                        parse_eeprom(prev_dev);
                        start_link(prev_dev);
                  } else
                        break;
            }
            controller_index = 0;
      }
subsequent_board:

      p = (void *)ee_data + ee_data[27 + controller_index*3];
      if (ee_data[27] == 0) {       /* No valid media table. */
      } else if (tp->chip_id == DC21041) {
            int media = get_u16(p);
            int count = p[2];
            p += 3;

            printk(KERN_INFO "%s: 21041 Media table, default media %4.4x
(%s).\n",
                     dev->name, media,
                     media & 0x0800 ? "Autosense" : medianame[media &
MEDIA_MASK]);
            for (i = 0; i < count; i++) {
                  unsigned char media_block = *p++;
                  int media_code = media_block & MEDIA_MASK;
                  if (media_block & 0x40)
                        p += 6;
                  switch(media_code) {
                  case 0: new_advertise |= 0x0020; break;
                  case 4: new_advertise |= 0x0040; break;
```

```
                }
                printk(KERN_INFO "%s:  21041 media #%d, %s.\n",
                           dev->name, media_code, medianame[media_code]);
          }
     } else {
          unsigned char csr12dir = 0;
          int count;
          struct mediatable *mtable;
          u16 media = get_u16(p);

          p += 2;
          if (tp->flags & CSR12_IN_SROM)
                csr12dir = *p++;
          count = *p++;
          mtable = (struct mediatable *)
                kmalloc(sizeof(struct mediatable) + count*sizeof(struct
medialeaf),
                                GFP_KERNEL);
          if (mtable == NULL)
                return;                         /* Horrible, impossible
failure. */
          last_mediatable = tp->mtable = mtable;
          mtable->defaultmedia = media;
          mtable->leafcount = count;
          mtable->csr12dir = csr12dir;
          mtable->has_nonmii = mtable->has_mii = mtable->has_reset = 0;
          mtable->csr15dir = mtable->csr15val = 0;

          printk(KERN_INFO "%s:  EEPROM default media type %s.\n", dev->name,
                    media & 0x0800 ? "Autosense" : medianame[media &
MEDIA_MASK]);
          for (i = 0; i < count; i++) {
                struct medialeaf *leaf = &mtable->mleaf[i];

                if ((p[0] & 0x80) == 0) { /* 21140 Compact block. */
                      leaf->type = 0;
                      leaf->media = p[0] & 0x3f;
                      leaf->leafdata = p;
                      if ((p[2] & 0x61) == 0x01)    /* Bogus, but Znyx boards
do it. */
                            mtable->has_mii = 1;
                      p += 4;
                } else {
                      switch(leaf->type = p[1]) {
                      case 5:
                            mtable->has_reset = i + 1; /* Assure non-zero */
                            /* Fall through */
                      case 6:
                            leaf->media = 31;
                            break;
                      case 1: case 3:
                            mtable->has_mii = 1;
                            leaf->media = 11;
                            break;
                      case 2:
                            if ((p[2] & 0x3f) == 0) {
                                  u32 base15 = (p[2] & 0x40) ? get_u16(p + 7)
```

```
: 0x0008;
                                    u16 *p1 = (u16 *)(p + (p[2] & 0x40 ? 9 : 3));
                                    mtable->csr15dir = (get_unaligned(p1 +
0)<<16) + base15;
                                    mtable->csr15val = (get_unaligned(p1 +
1)<<16) + base15;
                              }
                              /* Fall through. */
                        case 0: case 4:
                              mtable->has_nonmii = 1;
                              leaf->media = p[2] & MEDIA_MASK;
                              switch (leaf->media) {
                              case 0: new_advertise |= 0x0020; break;
                              case 4: new_advertise |= 0x0040; break;
                              case 3: new_advertise |= 0x0080; break;
                              case 5: new_advertise |= 0x0100; break;
                              case 6: new_advertise |= 0x0200; break;
                              }
                              break;
                        default:
                              leaf->media = 19;
                        }
                        leaf->leafdata = p + 2;
                        p += (p[0] & 0x3f) + 1;
                  }
                  if ((tp->msg_level & NETIF_MSG_LINK) &&
                        leaf->media == 11) {
                        unsigned char *bp = leaf->leafdata;
                        printk(KERN_INFO "%s:  MII interface PHY %d, setup/reset
"
                                    "sequences %d/%d long, capabilities %2.2x
%2.2x.\n",
                                    dev->name, bp[0], bp[1], bp[2 + bp[1]*2],
                                    bp[5 + bp[2 + bp[1]*2]*2], bp[4 + bp[2 +
bp[1]*2]*2]);
                  }
                  if (tp->msg_level & NETIF_MSG_PROBE)
                        printk(KERN_INFO "%s:  Index #%d - Media %s (#%d)
described "
                                    "by a %s (%d) block.\n",
                                    dev->name, i, medianame[leaf->media],
leaf->media,
                                    leaf->type < 6 ? block_name[leaf->type] :
"UNKNOWN",
                                    leaf->type);
            }
            if (new_advertise)
                  tp->sym_advertise = new_advertise;
      }
}
/* Reading a serial EEPROM is a "bit" grungy, but we work our way through:->.*/

/*  EEPROM_Ctrl bits. */
#define EE_SHIFT_CLK    0x02  /* EEPROM shift clock. */
#define EE_CS                 0x01  /* EEPROM chip select. */
#define EE_DATA_WRITE   0x04  /* Data from the Tulip to EEPROM. */
#define EE_WRITE_0            0x01
```

```
#define EE_WRITE_1            0x05
#define EE_DATA_READ    0x08  /* Data from the EEPROM chip. */
#define EE_ENB                (0x4800 | EE_CS)

/* Delay between EEPROM clock transitions.
   Even at 33Mhz current PCI implementations do not overrun the EEPROM clock.
   We add a bus turn-around to insure that this remains true. */
#define eeprom_delay()  inl(ee_addr)

/* The EEPROM commands include the alway-set leading bit. */
#define EE_READ_CMD           (6)

/* Note: this routine returns extra data bits for size detection. */
static int read_eeprom(long ioaddr, int location, int addr_len)
{
      int i;
      unsigned retval = 0;
      long ee_addr = ioaddr + CSR9;
      int read_cmd = location | (EE_READ_CMD << addr_len);

      outl(EE_ENB & ~EE_CS, ee_addr);
      outl(EE_ENB, ee_addr);

      /* Shift the read command bits out. */
      for (i = 4 + addr_len; i >= 0; i--) {
            short dataval = (read_cmd & (1 << i)) ? EE_DATA_WRITE : 0;
            outl(EE_ENB | dataval, ee_addr);
            eeprom_delay();
            outl(EE_ENB | dataval | EE_SHIFT_CLK, ee_addr);
            eeprom_delay();
            retval = (retval << 1) | ((inl(ee_addr) & EE_DATA_READ) ? 1 : 0);
      }
      outl(EE_ENB, ee_addr);
      eeprom_delay();

      for (i = 16; i > 0; i--) {
            outl(EE_ENB | EE_SHIFT_CLK, ee_addr);
            eeprom_delay();
            retval = (retval << 1) | ((inl(ee_addr) & EE_DATA_READ) ? 1 : 0);
            outl(EE_ENB, ee_addr);
            eeprom_delay();
      }

      /* Terminate the EEPROM access. */
      outl(EE_ENB & ~EE_CS, ee_addr);
      return retval;
}

/* 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 or future 66Mhz PCI. */
#define mdio_delay() inl(mdio_addr)
```

```
/* Read and write the MII registers using software-generated serial
   MDIO protocol.  It is just different enough from the EEPROM protocol
   to not share code.  The maxium data clock rate is 2.5 Mhz. */
#define MDIO_SHIFT_CLK  0x10000
#define MDIO_DATA_WRITE0 0x00000
#define MDIO_DATA_WRITE1 0x20000
#define MDIO_ENB        0x00000          /* Ignore the 0x02000 databook
setting. */
#define MDIO_ENB_IN          0x40000
#define MDIO_DATA_READ  0x80000

static const unsigned char comet_miireg2offset[32] = {
      0xB4, 0xB8, 0xBC, 0xC0,  0xC4, 0xC8, 0xCC, 0,  0,0,0,0,  0,0,0,0,
      0,0xD0,0,0,  0,0,0,0,  0,0,0,0, 0, 0xD4, 0xD8, 0xDC, };

static int mdio_read(struct net_device *dev, int phy_id, int location)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      int i;
      int read_cmd = (0xf6 << 10) | ((phy_id & 0x1f) << 5) | location;
      int retval = 0;
      long ioaddr = dev->base_addr;
      long mdio_addr = ioaddr + CSR9;
      unsigned long flags;

      if (location & ~0x1f)
            return 0xffff;

      if (tp->chip_id == COMET  &&  phy_id == 30) {
            if (comet_miireg2offset[location])
                  return inl(ioaddr + comet_miireg2offset[location]);
            return 0xffff;
      }

      spin_lock_irqsave(&tp->mii_lock, flags);
      if (tp->chip_id == LC82C168) {
            int i = 1000;
            outl(0x60020000 + (phy_id<<23) + (location<<18), ioaddr + 0xA0);
            inl(ioaddr + 0xA0);
            inl(ioaddr + 0xA0);
            inl(ioaddr + 0xA0);
            inl(ioaddr + 0xA0);
            while (--i > 0)
                  if ( ! ((retval = inl(ioaddr + 0xA0)) & 0x80000000))
                        break;
            spin_unlock_irqrestore(&tp->mii_lock, flags);
            return retval & 0xffff;
      }

      /* Establish sync by sending at least 32 logic ones. */
      for (i = 32; i >= 0; i--) {
            outl(MDIO_ENB | MDIO_DATA_WRITE1, mdio_addr);
            mdio_delay();
            outl(MDIO_ENB | MDIO_DATA_WRITE1 | MDIO_SHIFT_CLK, mdio_addr);
            mdio_delay();
      }
```

```
	/* Shift the read command bits out. */
	for (i = 15; i >= 0; i--) {
		int dataval = (read_cmd & (1 << i)) ? MDIO_DATA_WRITE1 : 0;

		outl(MDIO_ENB | dataval, mdio_addr);
		mdio_delay();
		outl(MDIO_ENB | dataval | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	/* Read the two transition, 16 data, and wire-idle bits. */
	for (i = 19; i > 0; i--) {
		outl(MDIO_ENB_IN, mdio_addr);
		mdio_delay();
		retval = (retval << 1) | ((inl(mdio_addr) & MDIO_DATA_READ) ? 1 : 0);
		outl(MDIO_ENB_IN | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	spin_unlock_irqrestore(&tp->mii_lock, flags);
	return (retval>>1) & 0xffff;
}

static void mdio_write(struct net_device *dev, int phy_id, int location, int val)
{
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	int i;
	int cmd = (0x5002 << 16) | (phy_id << 23) | (location<<18) | (val & 0xffff);
	long ioaddr = dev->base_addr;
	long mdio_addr = ioaddr + CSR9;
	unsigned long flags;

	if (location & ~0x1f)
		return;

	if (tp->chip_id == COMET  &&  phy_id == 30) {
		if (comet_miireg2offset[location])
			outl(val, ioaddr + comet_miireg2offset[location]);
		return;
	}

	spin_lock_irqsave(&tp->mii_lock, flags);
	if (tp->chip_id == LC82C168) {
		int i = 1000;
		outl(cmd, ioaddr + 0xA0);
		do
			if ( ! (inl(ioaddr + 0xA0) & 0x80000000))
				break;
		while (--i > 0);
		spin_unlock_irqrestore(&tp->mii_lock, flags);
		return;
	}

	/* Establish sync by sending 32 logic ones. */
	for (i = 32; i >= 0; i--) {
		outl(MDIO_ENB | MDIO_DATA_WRITE1, mdio_addr);
		mdio_delay();
		outl(MDIO_ENB | MDIO_DATA_WRITE1 | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
```

```
	}
	/* Shift the command bits out. */
	for (i = 31; i >= 0; i--) {
		int dataval = (cmd & (1 << i)) ? MDIO_DATA_WRITE1 : 0;
		outl(MDIO_ENB | dataval, mdio_addr);
		mdio_delay();
		outl(MDIO_ENB | dataval | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	/* Clear out extra bits. */
	for (i = 2; i > 0; i--) {
		outl(MDIO_ENB_IN, mdio_addr);
		mdio_delay();
		outl(MDIO_ENB_IN | MDIO_SHIFT_CLK, mdio_addr);
		mdio_delay();
	}
	spin_unlock_irqrestore(&tp->mii_lock, flags);
	return;
}
```

```
static int
tulip_open(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int next_tick = 3*HZ;

      /* Wake the chip from sleep/snooze mode. */
      if (tp->flags & HAS_PWRDWN)
            pci_write_config_dword(tp->pci_dev, 0x40, 0);

      /* On some chip revs we must set the MII/SYM port before the reset!? */
      if (tp->mii_cnt  ||  (tp->mtable  &&  tp->mtable->has_mii))
            outl(0x00040000, ioaddr + CSR6);

      /* Reset the chip, holding bit 0 set at least 50 PCI cycles. */
      outl(0x00000001, ioaddr + CSR0);

      MOD_INC_USE_COUNT;

      /* This would be done after interrupts are initialized, but we do not want
         to frob the transceiver only to fail later. */
      if (request_irq(dev->irq, &tulip_interrupt, SA_SHIRQ, dev->name, dev)) {
            MOD_DEC_USE_COUNT;
            return -EAGAIN;
      }

      /* Deassert reset.
         Wait the specified 50 PCI cycles after a reset by initializing
         Tx and Rx queues and the address filter list. */
      outl(tp->csr0, ioaddr + CSR0);

      if (tp->msg_level & NETIF_MSG_IFUP)
            printk(KERN_DEBUG "%s: tulip_open() irq %d.\n", dev->name,
dev->irq);

      tulip_init_ring(dev);

      if (tp->chip_id == PNIC2) {
            u32 addr_high = (dev->dev_addr[1]<<8) + (dev->dev_addr[0]<<0);
            /* This address setting does not appear to impact chip operation??
*/
            outl((dev->dev_addr[5]<<8) + dev->dev_addr[4] +
                  (dev->dev_addr[3]<<24) + (dev->dev_addr[2]<<16),
                  ioaddr + 0xB0);
            outl(addr_high + (addr_high<<16), ioaddr + 0xB8);
      }
      if (tp->flags & MC_HASH_ONLY) {
            u32 addr_low = cpu_to_le32(get_unaligned((u32 *)dev->dev_addr));
            u32 addr_high = cpu_to_le16(get_unaligned((u16
*)(dev->dev_addr+4)));
            if (tp->flags & IS_ASIX) {
                  outl(0, ioaddr + CSR13);
                  outl(addr_low,  ioaddr + CSR14);
                  outl(1, ioaddr + CSR13);
                  outl(addr_high, ioaddr + CSR14);
```

```
		} else if (tp->flags & COMET_MAC_ADDR) {
			outl(addr_low,  ioaddr + 0xA4);
			outl(addr_high, ioaddr + 0xA8);
			outl(0, ioaddr + 0xAC);
			outl(0, ioaddr + 0xB0);
		}
	}

	outl(virt_to_bus(tp->rx_ring), ioaddr + CSR3);
	outl(virt_to_bus(tp->tx_ring), ioaddr + CSR4);

	if ( ! tp->full_duplex_lock)
		tp->full_duplex = 0;
	init_media(dev);
	if (media_cap[dev->if_port] & MediaIsMII)
		check_duplex(dev);
	set_rx_mode(dev);

	/* Start the Tx to process setup frame. */
	outl(tp->csr6, ioaddr + CSR6);
	outl(tp->csr6 | TxOn, ioaddr + CSR6);

	/* Patch to clear EEPROM Chip Select (lsb of CSR9) */
	outl(0xf0000000, ioaddr + CSR9);

	netif_start_tx_queue(dev);

	/* Enable interrupts by setting the interrupt mask. */
	outl(tulip_tbl[tp->chip_id].valid_intrs, ioaddr + CSR5);
	outl(tulip_tbl[tp->chip_id].valid_intrs, ioaddr + CSR7);
	outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
	outl(0, ioaddr + CSR2);		/* Rx poll demand */

	if (tp->msg_level & NETIF_MSG_IFUP)
		printk(KERN_DEBUG "%s: Done tulip_open(), CSR0 %8.8x, CSR5 %8.8x CSR6
"
			"%8.8x.\n", dev->name, (int)inl(ioaddr + CSR0),
			(int)inl(ioaddr + CSR5), (int)inl(ioaddr + CSR6));

	/* Set the timer to switch to check for link beat and perhaps switch
	   to an alternate media type. */
	init_timer(&tp->timer);
	tp->timer.expires = jiffies + next_tick;
	tp->timer.data = (unsigned long)dev;
	tp->timer.function = tulip_tbl[tp->chip_id].media_timer;
	add_timer(&tp->timer);

	return 0;
}

static void init_media(struct net_device *dev)
{
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	long ioaddr = dev->base_addr;
	int i;

	tp->saved_if_port = dev->if_port;
```

```
	if (dev->if_port == 0)
		dev->if_port = tp->default_port;

	/* Allow selecting a default media. */
	i = 0;
	if (tp->mtable == NULL)
		goto media_picked;
	if (dev->if_port) {
		int looking_for = media_cap[dev->if_port] & MediaIsMII ? 11 :
			(dev->if_port == 12 ? 0 : dev->if_port);
		for (i = 0; i < tp->mtable->leafcount; i++)
			if (tp->mtable->mleaf[i].media == looking_for) {
				printk(KERN_INFO "%s: Using user-specified media %s.\n",
					dev->name, medianame[dev->if_port]);
				goto media_picked;
			}
	}
	if ((tp->mtable->defaultmedia & 0x0800) == 0) {
		int looking_for = tp->mtable->defaultmedia & MEDIA_MASK;
		for (i = 0; i < tp->mtable->leafcount; i++)
			if (tp->mtable->mleaf[i].media == looking_for) {
				printk(KERN_INFO "%s: Using EEPROM-set media %s.\n",
					dev->name, medianame[looking_for]);
				goto media_picked;
			}
	}
	/* Start sensing first non-full-duplex media. */
	for (i = tp->mtable->leafcount - 1;
		 (media_cap[tp->mtable->mleaf[i].media] & MediaAlwaysFD) && i > 0;
i--)
		;
media_picked:

	tp->csr6 = 0;
	tp->cur_index = i;
	tp->nwayset = 0;

	if (dev->if_port) {
		if (tp->chip_id == DC21143  &&
			(media_cap[dev->if_port] & MediaIsMII)) {
			/* We must reset the media CSRs when we force-select MII mode.
*/
			outl(0x0000, ioaddr + CSR13);
			outl(0x0000, ioaddr + CSR14);
			outl(0x0008, ioaddr + CSR15);
		}
		select_media(dev, 1);
		return;
	}
	switch(tp->chip_id) {
	case DC21041:
		/* tp->nway = 1;*/
		nway_start(dev);
		break;
	case DC21142:
		if (tp->mii_cnt) {
			select_media(dev, 1);
```

```
                if (tp->msg_level & NETIF_MSG_LINK)
                      printk(KERN_INFO "%s: Using MII transceiver %d, status
"
                                  "%4.4x.\n",
                                  dev->name, tp->phys[0], mdio_read(dev,
tp->phys[0], 1));
                outl(0x82020000, ioaddr + CSR6);
                tp->csr6 = 0x820E0000;
                dev->if_port = 11;
                outl(0x0000, ioaddr + CSR13);
                outl(0x0000, ioaddr + CSR14);
          } else
                nway_start(dev);
          break;
      case PNIC2:
          nway_start(dev);
          break;
      case LC82C168:
          if (tp->mii_cnt) {
                dev->if_port = 11;
                tp->csr6 = 0x814C0000 | (tp->full_duplex ? FullDuplex : 0);
                outl(0x0001, ioaddr + CSR15);
          } else if (inl(ioaddr + CSR5) & TPLnkPass)
                pnic_do_nway(dev);
          else {
                /* Start with 10mbps to do autonegotiation. */
                outl(0x32, ioaddr + CSR12);
                tp->csr6 = 0x00420000;
                outl(0x0001B078, ioaddr + 0xB8);
                outl(0x0201B078, ioaddr + 0xB8);
          }
          break;
      case MX98713: case COMPEX9881:
          dev->if_port = 0;
          tp->csr6 = 0x01880000 | (tp->full_duplex ? FullDuplex : 0);
          outl(0x0f370000 | inw(ioaddr + 0x80), ioaddr + 0x80);
          break;
      case MX98715: case MX98725:
          /* Provided by BOLO, Macronix - 12/10/1998. */
          dev->if_port = 0;
          tp->csr6 = 0x01a80000 | FullDuplex;
          outl(0x0f370000 | inw(ioaddr + 0x80), ioaddr + 0x80);
          outl(0x11000 | inw(ioaddr + 0xa0), ioaddr + 0xa0);
          break;
      case COMET: case CONEXANT:
          /* Enable automatic Tx underrun recovery. */
          outl(inl(ioaddr + 0x88) | 1, ioaddr + 0x88);
          dev->if_port = tp->mii_cnt ? 11 : 0;
          tp->csr6 = 0x00040000;
          break;
      case AX88140: case AX88141:
          tp->csr6 = tp->mii_cnt ? 0x00040100 : 0x00000100;
          break;
      default:
          select_media(dev, 1);
      }
}
```

```
/* Set up the transceiver control registers for the selected media type.
   STARTUP indicates to reset the transceiver.  It is set to '2' for
   the initial card detection, and '1' during resume or open().
*/
static void select_media(struct net_device *dev, int startup)
{
      long ioaddr = dev->base_addr;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      struct mediatable *mtable = tp->mtable;
      u32 new_csr6;
      int i;

      if (mtable) {
            struct medialeaf *mleaf = &mtable->mleaf[tp->cur_index];
            unsigned char *p = mleaf->leafdata;
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s:  Media table type %d.\n",
                           dev->name, mleaf->type);
            switch (mleaf->type) {
            case 0:                                 /* 21140 non-MII xcvr. */
                  if (tp->msg_level & NETIF_MSG_LINK)
                        printk(KERN_DEBUG "%s: Using a 21140 non-MII
transceiver"
                                     " with control setting %2.2x.\n",
                                     dev->name, p[1]);
                  dev->if_port = p[0];
                  if (startup)
                        outl(mtable->csr12dir | 0x100, ioaddr + CSR12);
                  outl(p[1], ioaddr + CSR12);
                  new_csr6 = 0x02000000 | ((p[2] & 0x71) << 18);
                  break;
            case 2: case 4: {
                  u16 setup[5];
                  u32 csr13val, csr14val, csr15dir, csr15val;
                  for (i = 0; i < 5; i++)
                        setup[i] = get_u16(&p[i*2 + 1]);

                  dev->if_port = p[0] & MEDIA_MASK;
                  if (media_cap[dev->if_port] & MediaAlwaysFD)
                        tp->full_duplex = 1;

                  if (startup && mtable->has_reset) {
                        struct medialeaf *rleaf =
&mtable->mleaf[mtable->has_reset-1];
                        unsigned char *rst = rleaf->leafdata;
                        if (tp->msg_level & NETIF_MSG_LINK)
                              printk(KERN_DEBUG "%s: Resetting the
transceiver.\n",
                                           dev->name);
                        for (i = 0; i < rst[0]; i++)
                              outl(get_u16(rst + 1 + (i<<1)) << 16, ioaddr +
CSR15);
                  }
                  if (tp->msg_level & NETIF_MSG_LINK)
                        printk(KERN_DEBUG "%s: 21143 non-MII %s transceiver
control "
```

```
                              "%4.4x/%4.4x.\n",
                              dev->name, medianame[dev->if_port], setup[0],
setup[1]);
                if (p[0] & 0x40) {      /* SIA (CSR13-15) setup values are
provided. */
                      csr13val = setup[0];
                      csr14val = setup[1];
                      csr15dir = (setup[3]<<16) | setup[2];
                      csr15val = (setup[4]<<16) | setup[2];
                      outl(0, ioaddr + CSR13);
                      outl(csr14val, ioaddr + CSR14);
                      outl(csr15dir, ioaddr + CSR15);     /* Direction */
                      outl(csr15val, ioaddr + CSR15);     /* Data */
                      outl(csr13val, ioaddr + CSR13);
                } else {
                      csr13val = 1;
                      csr14val = 0x0003FFFF;
                      csr15dir = (setup[0]<<16) | 0x0008;
                      csr15val = (setup[1]<<16) | 0x0008;
                      if (dev->if_port <= 4)
                            csr14val = t21142_csr14[dev->if_port];
                      if (startup) {
                            outl(0, ioaddr + CSR13);
                            outl(csr14val, ioaddr + CSR14);
                      }
                      outl(csr15dir, ioaddr + CSR15);     /* Direction */
                      outl(csr15val, ioaddr + CSR15);     /* Data */
                      if (startup) outl(csr13val, ioaddr + CSR13);
                }
                if (tp->msg_level & NETIF_MSG_LINK)
                      printk(KERN_DEBUG "%s:  Setting CSR15 to
%8.8x/%8.8x.\n",
                                  dev->name, csr15dir, csr15val);
                if (mleaf->type == 4)
                      new_csr6 = 0x820A0000 | ((setup[2] & 0x71) << 18);
                else
                      new_csr6 = 0x82420000;
                break;
            }
            case 1: case 3: {
                int phy_num = p[0];
                int init_length = p[1];
                u16 *misc_info;

                dev->if_port = 11;
                new_csr6 = 0x020E0000;
                if (mleaf->type == 3) { /* 21142 */
                      u16 *init_sequence = (u16*)(p+2);
                      u16 *reset_sequence = &((u16*)(p+3))[init_length];
                      int reset_length = p[2 + init_length*2];
                      misc_info = reset_sequence + reset_length;
                      if (startup)
                            for (i = 0; i < reset_length; i++)
                                  outl(get_u16(&reset_sequence[i]) << 16,
ioaddr + CSR15);
                      for (i = 0; i < init_length; i++)
                            outl(get_u16(&init_sequence[i]) << 16, ioaddr +
```

```
CSR15);
                  } else {
                        u8 *init_sequence = p + 2;
                        u8 *reset_sequence = p + 3 + init_length;
                        int reset_length = p[2 + init_length];
                        misc_info = (u16*)(reset_sequence + reset_length);
                        if (startup) {
                              outl(mtable->csr12dir | 0x100, ioaddr + CSR12);
                              for (i = 0; i < reset_length; i++)
                                    outl(reset_sequence[i], ioaddr + CSR12);
                        }
                        for (i = 0; i < init_length; i++)
                              outl(init_sequence[i], ioaddr + CSR12);
                  }
                  tp->advertising[phy_num] = get_u16(&misc_info[1]) | 1;
                  if (startup < 2) {
                        if (tp->mii_advertise == 0)
                              tp->mii_advertise = tp->advertising[phy_num];
                        if (tp->msg_level & NETIF_MSG_LINK)
                              printk(KERN_DEBUG "%s:  Advertising %4.4x on MII
%d.\n",
                                          dev->name, tp->mii_advertise,
tp->phys[phy_num]);
                        mdio_write(dev, tp->phys[phy_num], 4,
tp->mii_advertise);
                  }
                  break;
            }
            default:
                  printk(KERN_DEBUG "%s:  Invalid media table selection %d.\n",
                                  dev->name, mleaf->type);
                  new_csr6 = 0x020E0000;
            }
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s: Using media type %s, CSR12 is %2.2x.\n",
                              dev->name, medianame[dev->if_port],
                              (int)inl(ioaddr + CSR12) & 0xff);
      } else if (tp->chip_id == DC21041) {
            int port = dev->if_port <= 4 ? dev->if_port : 0;
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s: 21041 using media %s, CSR12 is
%4.4x.\n",
                              dev->name, medianame[port == 3 ? 12: port],
                              (int)inl(ioaddr + CSR12));
            outl(0x00000000, ioaddr + CSR13); /* Reset the serial interface */
            outl(t21041_csr14[port], ioaddr + CSR14);
            outl(t21041_csr15[port], ioaddr + CSR15);
            outl(t21041_csr13[port], ioaddr + CSR13);
            new_csr6 = 0x80020000;
      } else if (tp->chip_id == LC82C168) {
            if (startup && ! tp->medialock)
                  dev->if_port = tp->mii_cnt ? 11 : 0;
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s: PNIC PHY status is %3.3x, media %s.\n",
                              dev->name, (int)inl(ioaddr + 0xB8),
                              medianame[dev->if_port]);
            if (tp->mii_cnt) {
```

```
                new_csr6 = 0x810C0000;
                outl(0x0001, ioaddr + CSR15);
                outl(0x0201B07A, ioaddr + 0xB8);
          } else if (startup) {
                /* Start with 10mbps to do autonegotiation. */
                outl(0x32, ioaddr + CSR12);
                new_csr6 = 0x00420000;
                outl(0x0001B078, ioaddr + 0xB8);
                outl(0x0201B078, ioaddr + 0xB8);
          } else if (dev->if_port == 3  ||  dev->if_port == 5) {
                outl(0x33, ioaddr + CSR12);
                new_csr6 = 0x01860000;
                /* Trigger autonegotiation. */
                outl(startup ? 0x0201F868 : 0x0001F868, ioaddr + 0xB8);
          } else {
                outl(0x32, ioaddr + CSR12);
                new_csr6 = 0x00420000;
                outl(0x1F078, ioaddr + 0xB8);
          }
     } else if (tp->chip_id == DC21040) {                                 /*
21040 */
          /* Turn on the xcvr interface. */
          int csr12 = inl(ioaddr + CSR12);
          if (tp->msg_level & NETIF_MSG_LINK)
                printk(KERN_DEBUG "%s: 21040 media type is %s, CSR12 is
%2.2x.\n",
                           dev->name, medianame[dev->if_port], csr12);
          if (media_cap[dev->if_port] & MediaAlwaysFD)
                tp->full_duplex = 1;
          new_csr6 = 0x20000;
          /* Set the full duplux match frame. */
          outl(FULL_DUPLEX_MAGIC, ioaddr + CSR11);
          outl(0x00000000, ioaddr + CSR13); /* Reset the serial interface */
          if (t21040_csr13[dev->if_port] & 8) {
                outl(0x0705, ioaddr + CSR14);
                outl(0x0006, ioaddr + CSR15);
          } else {
                outl(0xffff, ioaddr + CSR14);
                outl(0x0000, ioaddr + CSR15);
          }
          outl(0x8f01 | t21040_csr13[dev->if_port], ioaddr + CSR13);
     } else {                              /* Unknown chip type with no media
table. */
          if (tp->default_port == 0)
                dev->if_port = tp->mii_cnt ? 11 : 3;
          if (media_cap[dev->if_port] & MediaIsMII) {
                new_csr6 = 0x020E0000;
          } else if (media_cap[dev->if_port] & MediaIsFx) {
                new_csr6 = 0x02860000;
          } else
                new_csr6 = 0x038E0000;
          if (tp->msg_level & NETIF_MSG_LINK)
                printk(KERN_DEBUG "%s: No media description table, assuming "
                           "%s transceiver, CSR12 %2.2x.\n",
                           dev->name, medianame[dev->if_port],
                           (int)inl(ioaddr + CSR12));
     }
```

```
	tp->csr6 = new_csr6 | (tp->csr6 & 0xfdff) |
		(tp->full_duplex ? FullDuplex : 0);
	return;
}

/*
  Check the MII negotiated duplex, and change the CSR6 setting if
  required.
  Return 0 if everything is OK.
  Return < 0 if the transceiver is missing or has no link beat.
  */
static int check_duplex(struct net_device *dev)
{
	long ioaddr = dev->base_addr;
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	int mii_reg1, mii_reg5, negotiated, duplex;

	if (tp->full_duplex_lock)
		return 0;
	mii_reg5 = mdio_read(dev, tp->phys[0], 5);
	negotiated = mii_reg5 & tp->mii_advertise;

	if (tp->msg_level & NETIF_MSG_TIMER)
		printk(KERN_INFO "%s: MII link partner %4.4x, negotiated %4.4x.\n",
			dev->name, mii_reg5, negotiated);
	if (mii_reg5 == 0xffff)
		return -2;
	if ((mii_reg5 & 0x4000) == 0  &&                /* No negotiation. */
		((mii_reg1 = mdio_read(dev, tp->phys[0], 1)) & 0x0004) == 0) {
		int new_reg1 = mdio_read(dev, tp->phys[0], 1);
		if ((new_reg1 & 0x0004) == 0) {
			if (tp->msg_level & NETIF_MSG_TIMER)
				printk(KERN_INFO "%s: No link beat on the MII interface,"
					" status %4.4x.\n", dev->name, new_reg1);
			return -1;
		}
	}
	duplex = ((negotiated & 0x0300) == 0x0100
		  || (negotiated & 0x00C0) == 0x0040);
	/* 100baseTx-FD  or  10T-FD, but not 100-HD */
	if (tp->full_duplex != duplex) {
		tp->full_duplex = duplex;
		if (negotiated & 0x0380)      /* 100mbps. */
			tp->csr6 &= ~0x00400000;
		if (tp->full_duplex) tp->csr6 |= FullDuplex;
		else                     tp->csr6 &= ~FullDuplex;
		outl(tp->csr6 | RxOn, ioaddr + CSR6);
		outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
		if (tp->msg_level & NETIF_MSG_LINK)
			printk(KERN_INFO "%s: Setting %s-duplex based on MII "
				"#%d link partner capability of %4.4x.\n",
				dev->name, tp->full_duplex ? "full" : "half",
				tp->phys[0], mii_reg5);
		return 1;
	}
	return 0;
```

```
}

static void tulip_timer(unsigned long data)
{
      struct net_device *dev = (struct net_device *)data;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      u32 csr12 = inl(ioaddr + CSR12);
      int next_tick = 2*HZ;

      if (tp->msg_level & NETIF_MSG_TIMER)
            printk(KERN_DEBUG "%s: Media selection tick, %s, status %8.8x mode"
                     " %8.8x SIA %8.8x %8.8x %8.8x %8.8x.\n",
                     dev->name, medianame[dev->if_port], (int)inl(ioaddr +
CSR5),
                     (int)inl(ioaddr + CSR6), csr12, (int)inl(ioaddr + CSR13),
                     (int)inl(ioaddr + CSR14), (int)inl(ioaddr + CSR15));

      switch (tp->chip_id) {
      case DC21040:
            if (!tp->medialock  &&  (csr12 & 0x0002)) { /* Network error */
                  if (tp->msg_level & NETIF_MSG_TIMER)
                        printk(KERN_INFO "%s: No link beat found.\n",
                                    dev->name);
                  dev->if_port = (dev->if_port == 2 ? 0 : 2);
                  select_media(dev, 0);
                  dev->trans_start = jiffies;
            }
            break;
      case DC21041:
            if (tp->msg_level & NETIF_MSG_TIMER)
                  printk(KERN_DEBUG "%s: 21041 media tick  CSR12 %8.8x.\n",
                           dev->name, csr12);
            if (tp->medialock) break;
            switch (dev->if_port) {
            case 0: case 3: case 4:
              if (csr12 & 0x0004) { /*LnkFail */
                  /* 10baseT is dead.  Check for activity on alternate port. */
                  tp->mediasense = 1;
                  if (csr12 & 0x0200)
                        dev->if_port = 2;
                  else
                        dev->if_port = 1;
                  if (tp->msg_level & NETIF_MSG_LINK)
                        printk(KERN_INFO "%s: No 21041 10baseT link beat, Media
"
                                    "switched to %s.\n",
                                    dev->name, medianame[dev->if_port]);
                  outl(0, ioaddr + CSR13); /* Reset */
                  outl(t21041_csr14[dev->if_port], ioaddr + CSR14);
                  outl(t21041_csr15[dev->if_port], ioaddr + CSR15);
                  outl(t21041_csr13[dev->if_port], ioaddr + CSR13);
                  next_tick = 10*HZ;                  /* 2.4 sec. */
              } else
                  next_tick = 30*HZ;
              break;
            case 1:                               /* 10base2 */
```

```
            case 2:                                 /* AUI */
                  if (csr12 & 0x0100) {
                        next_tick = (30*HZ);                /* 30 sec. */
                        tp->mediasense = 0;
                  } else if ((csr12 & 0x0004) == 0) {
                        if (tp->msg_level & NETIF_MSG_LINK)
                              printk(KERN_INFO "%s: 21041 media switched to
10baseT.\n",
                                          dev->name);
                        dev->if_port = 0;
                        select_media(dev, 0);
                        next_tick = (24*HZ)/10;                  /* 2.4 sec.
*/
                  } else if (tp->mediasense || (csr12 & 0x0002)) {
                        dev->if_port = 3 - dev->if_port; /* Swap ports. */
                        select_media(dev, 0);
                        next_tick = 20*HZ;
                  } else {
                        next_tick = 20*HZ;
                  }
                  break;
            }
            break;
      case DC21140:  case DC21142: case MX98713: case COMPEX9881: default: {
            struct medialeaf *mleaf;
            unsigned char *p;
            if (tp->mtable == NULL) {     /* No EEPROM info, use generic code.
*/
                  /* Not much that can be done.
                     Assume this a generic MII or SYM transceiver. */
                  next_tick = 60*HZ;
                  if (tp->msg_level & NETIF_MSG_TIMER)
                        printk(KERN_DEBUG "%s: network media monitor CSR6 %8.8x
"
                                    "CSR12 0x%2.2x.\n",
                                    dev->name, (int)inl(ioaddr + CSR6), csr12 &
0xff);
                  break;
            }
            mleaf = &tp->mtable->mleaf[tp->cur_index];
            p = mleaf->leafdata;
            switch (mleaf->type) {
            case 0: case 4: {
                  /* Type 0 serial or 4 SYM transceiver.  Check the link beat bit.
*/
                  int offset = mleaf->type == 4 ? 5 : 2;
                  s8 bitnum = p[offset];
                  if (p[offset+1] & 0x80) {
                        if (tp->msg_level & NETIF_MSG_TIMER)
                              printk(KERN_DEBUG"%s: Transceiver monitor tick "
                                          "CSR12=%#2.2x, no media sense.\n",
                                          dev->name, csr12);
                        if (mleaf->type == 4) {
                              if (mleaf->media == 3 && (csr12 & 0x02))
                                    goto select_next_media;
                        }
                        break;
```

```
                }
                if (tp->msg_level & NETIF_MSG_TIMER)
                      printk(KERN_DEBUG "%s: Transceiver monitor tick:
CSR12=%#2.2x"
                                     " bit %d is %d, expecting %d.\n",
                                     dev->name, csr12, (bitnum >> 1) & 7,
                                     (csr12 & (1 << ((bitnum >> 1) & 7))) != 0,
                                     (bitnum >= 0));
                /* Check that the specified bit has the proper value. */
                if ((bitnum < 0) !=
                      ((csr12 & (1 << ((bitnum >> 1) & 7))) != 0)) {
                      if (tp->msg_level & NETIF_MSG_LINK)
                            printk(KERN_DEBUG "%s: Link beat detected for
%s.\n",
                                         dev->name, medianame[mleaf->media &
MEDIA_MASK]);
                      if ((p[2] & 0x61) == 0x01)    /* Bogus Znyx board. */
                            goto actually_mii;
                      break;
                }
                if (tp->medialock)
                      break;
        select_next_media:
                if (--tp->cur_index < 0) {
                      /* We start again, but should instead look for default.
*/
                      tp->cur_index = tp->mtable->leafcount - 1;
                }
                dev->if_port = tp->mtable->mleaf[tp->cur_index].media;
                if (media_cap[dev->if_port] & MediaIsFD)
                      goto select_next_media; /* Skip FD entries. */
                if (tp->msg_level & NETIF_MSG_LINK)
                      printk(KERN_DEBUG "%s: No link beat on media %s,"
                                     " trying transceiver type %s.\n",
                                     dev->name, medianame[mleaf->media &
MEDIA_MASK],

medianame[tp->mtable->mleaf[tp->cur_index].media]);
                select_media(dev, 0);
                /* Restart the transmit process. */
                outl(tp->csr6 | RxOn, ioaddr + CSR6);
                outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
                next_tick = (24*HZ)/10;
                break;
            }
            case 1:  case 3:        /* 21140, 21142 MII */
            actually_mii:
                  check_duplex(dev);
                  next_tick = 60*HZ;
                  break;
            case 2:                                 /* 21142 serial block has no
link beat. */
            default:
                  break;
            }
      }
      break;
```

```
	}
	tp->timer.expires = jiffies + next_tick;
	add_timer(&tp->timer);
}

/* Handle internal NWay transceivers uniquely.
   These exist on the 21041, 21143 (in SYM mode) and the PNIC2.
   */
static void nway_timer(unsigned long data)
{
	struct net_device *dev = (struct net_device *)data;
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	long ioaddr = dev->base_addr;
	int csr12 = inl(ioaddr + CSR12);
	int next_tick = 60*HZ;
	int new_csr6 = 0;

	if (tp->msg_level & NETIF_MSG_TIMER)
		printk(KERN_INFO"%s: N-Way autonegotiation status %8.8x, %s.\n",
			dev->name, csr12, medianame[dev->if_port]);
	if (media_cap[dev->if_port] & MediaIsMII) {
		check_duplex(dev);
	} else if (tp->nwayset) {
		/* Do not screw up a negotiated session! */
		if (tp->msg_level & NETIF_MSG_TIMER)
			printk(KERN_INFO"%s: Using NWay-set %s media, csr12 %8.8x.\n",
				dev->name, medianame[dev->if_port], csr12);
	} else if (tp->medialock) {
			;
	} else if (dev->if_port == 3) {
		if (csr12 & 2) {  /* No 100mbps link beat, revert to 10mbps. */
			if (tp->msg_level & NETIF_MSG_LINK)
				printk(KERN_INFO"%s: No 21143 100baseTx link beat,
%8.8x, "
					"trying NWay.\n", dev->name, csr12);
			nway_start(dev);
			next_tick = 3*HZ;
		}
	} else if ((csr12 & 0x7000) != 0x5000) {
		/* Negotiation failed.  Search media types. */
		if (tp->msg_level & NETIF_MSG_LINK)
			printk(KERN_INFO"%s: 21143 negotiation failed, status
%8.8x.\n",
				dev->name, csr12);
		if (!(csr12 & 4)) {		/* 10mbps link beat good. */
			new_csr6 = 0x82420000;
			dev->if_port = 0;
			outl(0, ioaddr + CSR13);
			outl(0x0003FFFF, ioaddr + CSR14);
			outw(t21142_csr15[dev->if_port], ioaddr + CSR15);
			outl(t21142_csr13[dev->if_port], ioaddr + CSR13);
		} else {
			/* Select 100mbps port to check for link beat. */
			new_csr6 = 0x83860000;
			dev->if_port = 3;
			outl(0, ioaddr + CSR13);
			outl(0x0003FF7F, ioaddr + CSR14);
```

```
                  outw(8, ioaddr + CSR15);
                  outl(1, ioaddr + CSR13);
            }
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_INFO"%s: Testing new 21143 media %s.\n",
                           dev->name, medianame[dev->if_port]);
            if (new_csr6 != (tp->csr6 & ~0x20D7)) {
                  tp->csr6 &= 0x20D7;
                  tp->csr6 |= new_csr6;
                  outl(0x0301, ioaddr + CSR12);
                  outl(tp->csr6 | RxOn, ioaddr + CSR6);
                  outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
            }
            next_tick = 3*HZ;
      }
      if (tp->cur_tx - tp->dirty_tx > 0  &&
            jiffies - dev->trans_start > TX_TIMEOUT) {
            printk(KERN_WARNING "%s: Tx hung, %d vs. %d.\n",
                     dev->name, tp->cur_tx, tp->dirty_tx);
            tulip_tx_timeout(dev);
      }

      tp->timer.expires = jiffies + next_tick;
      add_timer(&tp->timer);
}

static void nway_start(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int csr14 = ((tp->sym_advertise & 0x0780) << 9)  |
            ((tp->sym_advertise&0x0020)<<1) | 0xffbf;

      dev->if_port = 0;
      tp->nway = tp->mediasense = 1;
      tp->nwayset = tp->lpar = 0;
      if (tp->chip_id == PNIC2) {
            tp->csr6 = 0x01000000 | (tp->sym_advertise & 0x0040 ? FullDuplex :
0);
            return;
      }
      if (tp->msg_level & NETIF_MSG_LINK)
            printk(KERN_DEBUG "%s: Restarting internal NWay autonegotiation, "
                     "%8.8x.\n", dev->name, csr14);
      outl(0x0001, ioaddr + CSR13);
      outl(csr14, ioaddr + CSR14);
      tp->csr6 = 0x82420000 | (tp->sym_advertise & 0x0040 ? FullDuplex : 0)
            | (tp->csr6 & 0x20ff);
      outl(tp->csr6, ioaddr + CSR6);
      if (tp->mtable  &&  tp->mtable->csr15dir) {
            outl(tp->mtable->csr15dir, ioaddr + CSR15);
            outl(tp->mtable->csr15val, ioaddr + CSR15);
      } else if (tp->chip_id != PNIC2)
            outw(0x0008, ioaddr + CSR15);
      if (tp->chip_id == DC21041)               /* Trigger NWAY. */
            outl(0xEF01, ioaddr + CSR12);
      else
```

```
            outl(0x1301, ioaddr + CSR12);
}

static void nway_lnk_change(struct net_device *dev, int csr5)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int csr12 = inl(ioaddr + CSR12);

      if (tp->chip_id == PNIC2) {
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_INFO"%s: PNIC-2 link status changed, CSR5/12/14
%8.8x"
                              " %8.8x, %8.8x.\n",
                              dev->name, csr12, csr5, (int)inl(ioaddr + CSR14));
            dev->if_port = 5;
            tp->lpar = csr12 >> 16;
            tp->nwayset = 1;
            tp->csr6 = 0x01000000 | (tp->csr6 & 0xffff);
            outl(tp->csr6, ioaddr + CSR6);
            return;
      }
      if (tp->msg_level & NETIF_MSG_LINK)
            printk(KERN_INFO"%s: 21143 link status interrupt %8.8x, CSR5 %x, "
                    "%8.8x.\n", dev->name, csr12, csr5, (int)inl(ioaddr +
CSR14));

      /* If NWay finished and we have a negotiated partner capability. */
      if (tp->nway  &&  !tp->nwayset  &&  (csr12 & 0x7000) == 0x5000) {
            int setup_done = 0;
            int negotiated = tp->sym_advertise & (csr12 >> 16);
            tp->lpar = csr12 >> 16;
            tp->nwayset = 1;
            if (negotiated & 0x0100)            dev->if_port = 5;
            else if (negotiated & 0x0080) dev->if_port = 3;
            else if (negotiated & 0x0040) dev->if_port = 4;
            else if (negotiated & 0x0020) dev->if_port = 0;
            else {
                  tp->nwayset = 0;
                  if ((csr12 & 2) == 0  &&  (tp->sym_advertise & 0x0180))
                        dev->if_port = 3;
            }
            tp->full_duplex = (media_cap[dev->if_port] & MediaAlwaysFD) ? 1:0;

            if (tp->msg_level & NETIF_MSG_LINK) {
                  if (tp->nwayset)
                        printk(KERN_INFO "%s: Switching to %s based on link "
                                    "negotiation %4.4x & %4.4x = %4.4x.\n",
                                    dev->name, medianame[dev->if_port],
tp->sym_advertise,
                                    tp->lpar, negotiated);
                  else
                        printk(KERN_INFO "%s: Autonegotiation failed, using %s,"
                                    " link beat status %4.4x.\n",
                                    dev->name, medianame[dev->if_port], csr12);
            }
```

```
            if (tp->mtable) {
                  int i;
                  for (i = 0; i < tp->mtable->leafcount; i++)
                        if (tp->mtable->mleaf[i].media == dev->if_port) {
                              tp->cur_index = i;
                              select_media(dev, 0);
                              setup_done = 1;
                              break;
                        }
            }
            if ( ! setup_done) {
                  tp->csr6 = (dev->if_port & 1 ? 0x838E0000 : 0x82420000)
                        | (tp->csr6 & 0x20ff);
                  if (tp->full_duplex)
                        tp->csr6 |= FullDuplex;
                  outl(1, ioaddr + CSR13);
            }
#if 0                                       /* Restart should not be needed. */
            outl(tp->csr6 | 0x0000, ioaddr + CSR6);
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s:  Restarting Tx and Rx, CSR5 is
%8.8x.\n",
                               dev->name, inl(ioaddr + CSR5));
#endif
            outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s:  Setting CSR6 %8.8x/%x CSR12 %8.8x.\n",
                               dev->name, tp->csr6, (int)inl(ioaddr + CSR6),
                               (int)inl(ioaddr + CSR12));
      } else if ((tp->nwayset  &&  (csr5 & 0x08000000)
                        && (dev->if_port == 3  ||  dev->if_port == 5)
                        && (csr12 & 2) == 2) ||
                     (tp->nway && (csr5 & (TPLnkFail)))) {
            /* Link blew? Maybe restart NWay. */
            del_timer(&tp->timer);
            nway_start(dev);
            tp->timer.expires = jiffies + 3*HZ;
            add_timer(&tp->timer);
      } else if (dev->if_port == 3  ||  dev->if_port == 5) {
            if (tp->msg_level & NETIF_MSG_LINK) /* TIMER? */
                  printk(KERN_INFO"%s: 21143 %s link beat %s.\n",
                               dev->name, medianame[dev->if_port],
                               (csr12 & 2) ? "failed" : "good");
            if ((csr12 & 2)  &&  ! tp->medialock) {
                  del_timer(&tp->timer);
                  nway_start(dev);
                  tp->timer.expires = jiffies + 3*HZ;
                  add_timer(&tp->timer);
            } else if (dev->if_port == 5)
                  outl(inl(ioaddr + CSR14) & ~0x080, ioaddr + CSR14);
      } else if (dev->if_port == 0  ||  dev->if_port == 4) {
            if ((csr12 & 4) == 0)
                  printk(KERN_INFO"%s: 21143 10baseT link beat good.\n",
                               dev->name);
      } else if (!(csr12 & 4)) {          /* 10mbps link beat good. */
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_INFO"%s: 21143 10mbps sensed media.\n",
```

```
                        dev->name);
            dev->if_port = 0;
      } else if (tp->nwayset) {
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_INFO"%s: 21143 using NWay-set %s, csr6 %8.8x.\n",
                        dev->name, medianame[dev->if_port], tp->csr6);
      } else {          /* 100mbps link beat good. */
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_INFO"%s: 21143 100baseTx sensed media.\n",
                        dev->name);
            dev->if_port = 3;
            tp->csr6 = 0x838E0000 | (tp->csr6 & 0x20ff);
            outl(0x0003FF7F, ioaddr + CSR14);
            outl(0x0301, ioaddr + CSR12);
            outl(tp->csr6 | RxOn, ioaddr + CSR6);
            outl(tp->csr6 | RxOn | TxOn, ioaddr + CSR6);
      }
}

static void mxic_timer(unsigned long data)
{
      struct net_device *dev = (struct net_device *)data;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int next_tick = 60*HZ;

      if (tp->msg_level & NETIF_MSG_TIMER) {
            printk(KERN_INFO"%s: MXIC negotiation status %8.8x.\n", dev->name,
                  (int)inl(ioaddr + CSR12));
      }
      tp->timer.expires = jiffies + next_tick;
      add_timer(&tp->timer);
}

static void pnic_do_nway(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      u32 phy_reg = inl(ioaddr + 0xB8);
      u32 new_csr6 = tp->csr6 & ~0x40C40200;

      if (phy_reg & 0x78000000) { /* Ignore baseT4 */
            if (phy_reg & 0x20000000)           dev->if_port = 5;
            else if (phy_reg & 0x40000000)      dev->if_port = 3;
            else if (phy_reg & 0x10000000)      dev->if_port = 4;
            else if (phy_reg & 0x08000000)      dev->if_port = 0;
            tp->nwayset = 1;
            new_csr6 = (dev->if_port & 1) ? 0x01860000 : 0x00420000;
            outl(0x32 | (dev->if_port & 1), ioaddr + CSR12);
            if (dev->if_port & 1)
                  outl(0x1F868, ioaddr + 0xB8);
            if (phy_reg & 0x30000000) {
                  tp->full_duplex = 1;
                  new_csr6 |= FullDuplex;
            }
            if (tp->msg_level & NETIF_MSG_LINK)
                  printk(KERN_DEBUG "%s: PNIC autonegotiated status %8.8x,
```

```
%s.\n",
                           dev->name, phy_reg, medianame[dev->if_port]);
            if (tp->csr6 != new_csr6) {
                  tp->csr6 = new_csr6;
                  outl(tp->csr6 | RxOn, ioaddr + CSR6);     /* Restart Tx */
                  outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
                  dev->trans_start = jiffies;
            }
      }
}

static void pnic_lnk_change(struct net_device *dev, int csr5)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int phy_reg = inl(ioaddr + 0xB8);

      if (tp->msg_level & NETIF_MSG_LINK)
            printk(KERN_DEBUG "%s: PNIC link changed state %8.8x, CSR5 %8.8x.\n",
                   dev->name, phy_reg, csr5);
      if (inl(ioaddr + CSR5) & TPLnkFail) {
            outl((inl(ioaddr + CSR7) & ~TPLnkFail) | TPLnkPass, ioaddr + CSR7);
            if (! tp->nwayset  ||  jiffies - dev->trans_start > 1*HZ) {
                  tp->csr6 = 0x00420000 | (tp->csr6 & 0x0000fdff);
                  outl(tp->csr6, ioaddr + CSR6);
                  outl(0x30, ioaddr + CSR12);
                  outl(0x0201F078, ioaddr + 0xB8); /* Turn on autonegotiation.
*/
                  dev->trans_start = jiffies;
            }
      } else if (inl(ioaddr + CSR5) & TPLnkPass) {
            pnic_do_nway(dev);
            outl((inl(ioaddr + CSR7) & ~TPLnkPass) | TPLnkFail, ioaddr + CSR7);
      }
}
static void pnic_timer(unsigned long data)
{
      struct net_device *dev = (struct net_device *)data;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int next_tick = 60*HZ;

      if (media_cap[dev->if_port] & MediaIsMII) {
            if (check_duplex(dev) > 0)
                  next_tick = 3*HZ;
      } else {
            int csr12 = inl(ioaddr + CSR12);
            int new_csr6 = tp->csr6 & ~0x40C40200;
            int phy_reg = inl(ioaddr + 0xB8);
            int csr5 = inl(ioaddr + CSR5);

            if (tp->msg_level & NETIF_MSG_TIMER)
                  printk(KERN_DEBUG "%s: PNIC timer PHY status %8.8x, %s "
                           "CSR5 %8.8x.\n",
                           dev->name, phy_reg, medianame[dev->if_port], csr5);
            if (phy_reg & 0x04000000) {   /* Remote link fault */
                  outl(0x0201F078, ioaddr + 0xB8);
```

```
                next_tick = 1*HZ;
                tp->nwayset = 0;
            } else if (phy_reg & 0x78000000) { /* Ignore baseT4 */
                pnic_do_nway(dev);
                next_tick = 60*HZ;
            } else if (csr5 & TPLnkFail) { /* 100baseTx link beat */
                if (tp->msg_level & NETIF_MSG_LINK)
                    printk(KERN_DEBUG "%s: %s link beat failed, CSR12 %4.4x,
"
                                "CSR5 %8.8x, PHY %3.3x.\n",
                                dev->name, medianame[dev->if_port], csr12,
                                (int)inl(ioaddr + CSR5), (int)inl(ioaddr +
0xB8));
                next_tick = 3*HZ;
                if (tp->medialock) {
                } else if (tp->nwayset  &&  (dev->if_port & 1)) {
                    next_tick = 1*HZ;
                } else if (dev->if_port == 0) {
                    dev->if_port = 3;
                    outl(0x33, ioaddr + CSR12);
                    new_csr6 = 0x01860000;
                    outl(0x1F868, ioaddr + 0xB8);
                } else {
                    dev->if_port = 0;
                    outl(0x32, ioaddr + CSR12);
                    new_csr6 = 0x00420000;
                    outl(0x1F078, ioaddr + 0xB8);
                }
                if (tp->csr6 != new_csr6) {
                    tp->csr6 = new_csr6;
                    outl(tp->csr6 | RxOn, ioaddr + CSR6);     /* Restart
Tx */
                    outl(tp->csr6 | RxOn | TxOn, ioaddr + CSR6);
                    dev->trans_start = jiffies;
                    if (tp->msg_level & NETIF_MSG_LINK)
                        printk(KERN_INFO "%s: Changing PNIC configuration
to %s "
                                "%s-duplex, CSR6 %8.8x.\n",
                                dev->name, medianame[dev->if_port],
                                tp->full_duplex ? "full" : "half",
new_csr6);
                }
            }
        }
        tp->timer.expires = jiffies + next_tick;
        add_timer(&tp->timer);
}

static void comet_timer(unsigned long data)
{
        struct net_device *dev = (struct net_device *)data;
        struct tulip_private *tp = (struct tulip_private *)dev->priv;
        int next_tick = 60*HZ;

        if (tp->msg_level & NETIF_MSG_TIMER)
            printk(KERN_DEBUG "%s: Comet link status %4.4x partner capability "
                    "%4.4x.\n",
```

```
                        dev->name, mdio_read(dev, tp->phys[0], 1),
                        mdio_read(dev, tp->phys[0], 5));
        check_duplex(dev);
        tp->timer.expires = jiffies + next_tick;
        add_timer(&tp->timer);
}

static void tulip_tx_timeout(struct net_device *dev)
{
        struct tulip_private *tp = (struct tulip_private *)dev->priv;
        long ioaddr = dev->base_addr;

        if (media_cap[dev->if_port] & MediaIsMII) {
                /* Do nothing -- the media monitor should handle this. */
                int mii_bmsr = mdio_read(dev, tp->phys[0], 1);
                if (tp->msg_level & NETIF_MSG_LINK)
                        printk(KERN_WARNING "%s: Transmit timeout using MII device,"
                                   " status %4.4x.\n",
                                   dev->name, mii_bmsr);
                if ( ! (mii_bmsr & 0x0004)) {       /* No link beat present */
                        dev->trans_start = jiffies;
                        netif_link_down(dev);
                        return;
                }
        } else switch (tp->chip_id) {
        case DC21040:
                if ( !tp->medialock  &&  inl(ioaddr + CSR12) & 0x0002) {
                        dev->if_port = (dev->if_port == 2 ? 0 : 2);
                        printk(KERN_INFO "%s: transmit timed out, switching to "
                                   "%s.\n",
                                   dev->name, medianame[dev->if_port]);
                        select_media(dev, 0);
                }
                dev->trans_start = jiffies;
                return;                             /* Note: not break! */
        case DC21041: {
                int csr12 = inl(ioaddr + CSR12);

                printk(KERN_WARNING "%s: 21041 transmit timed out, status %8.8x, "
                           "CSR12 %8.8x, CSR13 %8.8x, CSR14 %8.8x, resetting...\n",
                           dev->name, (int)inl(ioaddr + CSR5), csr12,
                           (int)inl(ioaddr + CSR13), (int)inl(ioaddr + CSR14));
                tp->mediasense = 1;
                if ( ! tp->medialock) {
                        if (dev->if_port == 1 || dev->if_port == 2)
                                dev->if_port = (csr12 & 0x0004) ? 2 - dev->if_port : 0;
                        else
                                dev->if_port = 1;
                        select_media(dev, 0);
                }
                break;
        }
        case DC21142:
                if (tp->nwayset) {
                        printk(KERN_WARNING "%s: Transmit timed out, status %8.8x, "
                                   "SIA %8.8x %8.8x %8.8x %8.8x, restarting NWay .\n",
                                   dev->name, (int)inl(ioaddr + CSR5),
```

```
                        (int)inl(ioaddr + CSR12), (int)inl(ioaddr + CSR13),
                        (int)inl(ioaddr + CSR14), (int)inl(ioaddr + CSR15));
                nway_start(dev);
                break;
            }
            /* Fall through. */
      case DC21140: case MX98713: case COMPEX9881:
            printk(KERN_WARNING "%s: %s transmit timed out, status %8.8x, "
                   "SIA %8.8x %8.8x %8.8x %8.8x, resetting...\n",
                   dev->name, tulip_tbl[tp->chip_id].chip_name,
                   (int)inl(ioaddr + CSR5), (int)inl(ioaddr + CSR12),
                   (int)inl(ioaddr + CSR13), (int)inl(ioaddr + CSR14),
                   (int)inl(ioaddr + CSR15));
            if ( ! tp->medialock  &&  tp->mtable) {
                do
                    --tp->cur_index;
                while (tp->cur_index >= 0
                       &&
(media_cap[tp->mtable->mleaf[tp->cur_index].media]
                                & MediaIsFD));
                if (tp->cur_index < 0) {
                    /* We start again, but should instead look for default.
*/
                    tp->cur_index = tp->mtable->leafcount - 1;
                }
                select_media(dev, 0);
                printk(KERN_WARNING "%s: transmit timed out, switching to %s
"
                       "media.\n", dev->name, medianame[dev->if_port]);
            }
            break;
      case PNIC2:
            printk(KERN_WARNING "%s: PNIC2 transmit timed out, status %8.8x, "
                   "CSR6/7 %8.8x / %8.8x CSR12 %8.8x, resetting...\n",
                   dev->name, (int)inl(ioaddr + CSR5), (int)inl(ioaddr +
CSR6),
                   (int)inl(ioaddr + CSR7), (int)inl(ioaddr + CSR12));
            break;
      default:
            printk(KERN_WARNING "%s: Transmit timed out, status %8.8x, CSR12 "
                   "%8.8x, resetting...\n",
                   dev->name, (int)inl(ioaddr + CSR5), (int)inl(ioaddr +
CSR12));
      }

#if defined(way_too_many_messages)  &&  defined(__i386__)
      if (tp->msg_level & NETIF_MSG_TXERR) {
            int i;
            for (i = 0; i < RX_RING_SIZE; i++) {
                u8 *buf = (u8 *)(tp->rx_ring[i].buffer1);
                int j;
                printk(KERN_DEBUG "%2d: %8.8x %8.8x %8.8x %8.8x  "
                       "%2.2x %2.2x %2.2x.\n",
                       i, (unsigned int)tp->rx_ring[i].status,
                       (unsigned int)tp->rx_ring[i].length,
                       (unsigned int)tp->rx_ring[i].buffer1,
                       (unsigned int)tp->rx_ring[i].buffer2,
```

```
                        buf[0], buf[1], buf[2]);
                for (j = 0; buf[j] != 0xee && j < 1600; j++)
                      if (j < 100) printk(" %2.2x", buf[j]);
                printk(" j=%d.\n", j);
          }
          printk(KERN_DEBUG "  Rx ring %8.8x: ", (int)tp->rx_ring);
          for (i = 0; i < RX_RING_SIZE; i++)
                printk(" %8.8x", (unsigned int)tp->rx_ring[i].status);
          printk("\n" KERN_DEBUG "  Tx ring %8.8x: ", (int)tp->tx_ring);
          for (i = 0; i < TX_RING_SIZE; i++)
                printk(" %8.8x", (unsigned int)tp->tx_ring[i].status);
          printk("\n");
    }
#endif

    /* Stop and restart the Tx process.
       The pwr_event approach of empty/init_rings() may be better... */
    outl(tp->csr6 | RxOn, ioaddr + CSR6);
    outl(tp->csr6 | RxOn | TxOn, ioaddr + CSR6);
    /* Trigger an immediate transmit demand. */
    outl(0, ioaddr + CSR1);
    outl(tulip_tbl[tp->chip_id].valid_intrs, ioaddr + CSR7);

    dev->trans_start = jiffies;
    tp->stats.tx_errors++;
    return;
}


/* Initialize the Rx and Tx rings, along with various 'dev' bits. */
static void tulip_init_ring(struct net_device *dev)
{
    struct tulip_private *tp = (struct tulip_private *)dev->priv;
    int i;

    tp->rx_dead = tp->tx_full = 0;
    tp->cur_rx = tp->cur_tx = 0;
    tp->dirty_rx = tp->dirty_tx = 0;

    tp->rx_buf_sz = dev->mtu + 18;
    if (tp->rx_buf_sz < PKT_BUF_SZ)
          tp->rx_buf_sz = PKT_BUF_SZ;

    for (i = 0; i < RX_RING_SIZE; i++) {
          tp->rx_ring[i].status = 0x00000000;
          tp->rx_ring[i].length = cpu_to_le32(tp->rx_buf_sz);
          tp->rx_ring[i].buffer2 = virt_to_le32desc(&tp->rx_ring[i+1]);
          tp->rx_skbuff[i] = NULL;
    }
    /* Mark the last entry as wrapping the ring. */
    tp->rx_ring[i-1].length |= cpu_to_le32(DESC_RING_WRAP);
    tp->rx_ring[i-1].buffer2 = virt_to_le32desc(&tp->rx_ring[0]);

    for (i = 0; i < RX_RING_SIZE; i++) {
          /* Note the receive buffer must be longword aligned.
             dev_alloc_skb() provides 16 byte alignment.  But do *not*
             use skb_reserve() to align the IP header! */
```

```
            struct sk_buff *skb = dev_alloc_skb(tp->rx_buf_sz);
            tp->rx_skbuff[i] = skb;
            if (skb == NULL)
                  break;
            skb->dev = dev;               /* Mark as being used by this device.
*/
            tp->rx_ring[i].status = cpu_to_le32(DescOwned);
            tp->rx_ring[i].buffer1 = virt_to_le32desc(skb->tail);
      }
      tp->dirty_rx = (unsigned int)(i - RX_RING_SIZE);

      /* The Tx buffer descriptor is filled in as needed, but we
         do need to clear the ownership bit. */
      for (i = 0; i < TX_RING_SIZE; i++) {
            tp->tx_skbuff[i] = 0;
            tp->tx_ring[i].status = 0x00000000;
            tp->tx_ring[i].buffer2 = virt_to_le32desc(&tp->tx_ring[i+1]);
      }
      tp->tx_ring[i-1].buffer2 = virt_to_le32desc(&tp->tx_ring[0]);
}

static int
tulip_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      int entry, q_used_cnt;
      u32 flag;

      /* Block a timer-based transmit from overlapping.  This happens when
         packets are presumed lost, and we use this check the Tx status. */
      if (netif_pause_tx_queue(dev) != 0) {
            /* This watchdog code is redundant with the media monitor timer. */
            if (jiffies - dev->trans_start > TX_TIMEOUT)
                  tulip_tx_timeout(dev);
            return 1;
      }

      /* Caution: the write order is important here, set the field
         with the ownership bits last. */

      /* Calculate the next Tx descriptor entry. */
      entry = tp->cur_tx % TX_RING_SIZE;
      q_used_cnt = tp->cur_tx - tp->dirty_tx;

      tp->tx_skbuff[entry] = skb;
      tp->tx_ring[entry].buffer1 = virt_to_le32desc(skb->data);

      if (q_used_cnt < TX_QUEUE_LEN/2) {/* Typical path */
            flag = 0x60000000; /* No interrupt */
      } else if (q_used_cnt == TX_QUEUE_LEN/2) {
            flag = 0xe0000000; /* Tx-done intr. */
      } else if (q_used_cnt < TX_QUEUE_LEN) {
            flag = 0x60000000; /* No Tx-done intr. */
      } else {          /* Leave room for set_rx_mode() to fill entries. */
            tp->tx_full = 1;
            flag = 0xe0000000; /* Tx-done intr. */
      }
```

```
	if (entry == TX_RING_SIZE-1)
		flag = 0xe0000000 | DESC_RING_WRAP;

	tp->tx_ring[entry].length = cpu_to_le32(skb->len | flag);
	tp->tx_ring[entry].status = cpu_to_le32(DescOwned);
	tp->cur_tx++;
	if ( ! tp->tx_full)
		netif_unpause_tx_queue(dev);
	else {
		netif_stop_tx_queue(dev);
		/* Check for a just-cleared queue race.
		   Note that this code path differs from other drivers because we
		   set the tx_full flag early. */
		if ( ! tp->tx_full)
			netif_resume_tx_queue(dev);
	}

	dev->trans_start = jiffies;
	/* Trigger an immediate transmit demand. */
	outl(0, dev->base_addr + CSR1);

	return 0;
}

/* The interrupt handler does all of the Rx thread work and cleans up
   after the Tx thread. */
static void tulip_interrupt(int irq, void *dev_instance, struct pt_regs *regs)
{
	struct net_device *dev = (struct net_device *)dev_instance;
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	long ioaddr = dev->base_addr;
	int csr5, work_budget = tp->max_interrupt_work;

	do {
		csr5 = inl(ioaddr + CSR5);
		if ((csr5 & (NormalIntr|AbnormalIntr)) == 0)
			break;

		if (tp->msg_level & NETIF_MSG_INTR)
			printk(KERN_DEBUG "%s: interrupt  csr5=%#8.8x new
csr5=%#8.8x.\n",
				dev->name, csr5, (int)inl(dev->base_addr + CSR5));
		/* Acknowledge all of the current interrupt sources ASAP. */
		outl(csr5 & 0x0001ffff, ioaddr + CSR5);

		if (csr5 & (RxIntr | RxNoBuf))
			work_budget -= tulip_rx(dev);

		if (csr5 & (TxNoBuf | TxDied | TxIntr)) {
			unsigned int dirty_tx;

			for (dirty_tx = tp->dirty_tx; tp->cur_tx - dirty_tx > 0;
				 dirty_tx++) {
				int entry = dirty_tx % TX_RING_SIZE;
				int status = le32_to_cpu(tp->tx_ring[entry].status);

				if (status < 0)
```

```
                              break;                  /* It still has not been
Txed */
                        /* Check for Rx filter setup frames. */
                        if (tp->tx_skbuff[entry] == NULL)
                          continue;

                        if (status & 0x8000) {
                              /* There was an major error, log it. */
                              if (tp->msg_level & NETIF_MSG_TX_ERR)
                                    printk(KERN_DEBUG "%s: Transmit error, Tx
status %8.8x.\n",
                                                dev->name, status);
                              tp->stats.tx_errors++;
                              if (status & 0x4104)
tp->stats.tx_aborted_errors++;
                              if (status & 0x0C00)
tp->stats.tx_carrier_errors++;
                              if (status & 0x0200)
tp->stats.tx_window_errors++;
                              if (status & 0x0002) tp->stats.tx_fifo_errors++;
                              if ((status & 0x0080) && tp->full_duplex == 0)
                                    tp->stats.tx_heartbeat_errors++;
#ifdef ETHER_STATS
                              if (status & 0x0100) tp->stats.collisions16++;
#endif
                        } else {
                              if (tp->msg_level & NETIF_MSG_TX_DONE)
                                    printk(KERN_DEBUG "%s: Transmit complete,
status "
                                                "%8.8x.\n", dev->name, status);
#ifdef ETHER_STATS
                              if (status & 0x0001) tp->stats.tx_deferred++;
#endif
#if LINUX_VERSION_CODE > 0x20127
                              tp->stats.tx_bytes += tp->tx_skbuff[entry]->len;
#endif
                              tp->stats.collisions += (status >> 3) & 15;
                              tp->stats.tx_packets++;
                        }

                        /* Free the original skb. */
                        dev_free_skb_irq(tp->tx_skbuff[entry]);
                        tp->tx_skbuff[entry] = 0;
                  }

#ifndef final_version
                  if (tp->cur_tx - dirty_tx > TX_RING_SIZE) {
                        printk(KERN_ERR "%s: Out-of-sync dirty pointer, %d vs.
%d, full=%d.\n",
                                   dev->name, dirty_tx, tp->cur_tx, tp->tx_full);
                        dirty_tx += TX_RING_SIZE;
                  }
#endif

                  if (tp->tx_full && tp->cur_tx - dirty_tx  < TX_QUEUE_LEN - 4)
{
                        /* The ring is no longer full, clear tbusy. */
```

```
                        tp->tx_full = 0;
                        netif_resume_tx_queue(dev);
                  }

                  tp->dirty_tx = dirty_tx;
            }

            if (tp->rx_dead) {
                  tulip_rx(dev);
                  if (tp->cur_rx - tp->dirty_rx < RX_RING_SIZE - 3) {
                        printk(KERN_ERR "%s: Restarted Rx at %d / %d.\n",
                                    dev->name, tp->cur_rx, tp->dirty_rx);
                        outl(0, ioaddr + CSR2);       /* Rx poll demand */
                        tp->rx_dead = 0;
                  }
            }

            /* Log errors. */
            if (csr5 & AbnormalIntr) {    /* Abnormal error summary bit. */
                  if (csr5 == 0xffffffff)
                        break;
                  if (csr5 & TxJabber) tp->stats.tx_errors++;
                  if (csr5 & PCIBusError) {
                        printk(KERN_ERR "%s: PCI Fatal Bus Error, %8.8x.\n",
                                    dev->name, csr5);
                  }
                  if (csr5 & TxFIFOUnderflow) {
                        if ((tp->csr6 & 0xC000) != 0xC000)
                              tp->csr6 += 0x4000;     /* Bump up the Tx
threshold */
                        else
                              tp->csr6 |= 0x00200000;  /* Store-n-forward. */
                        if (tp->msg_level & NETIF_MSG_TX_ERR)
                              printk(KERN_WARNING "%s: Tx threshold increased,
"
                                          "new CSR6 %x.\n", dev->name, tp->csr6);
                  }
                  if (csr5 & TxDied) {
                        /* This is normal when changing Tx modes. */
                        if (tp->msg_level & NETIF_MSG_LINK)
                              printk(KERN_WARNING "%s: The transmitter
stopped."
                                          "  CSR5 is %x, CSR6 %x, new CSR6 %x.\n",
                                          dev->name, csr5, (int)inl(ioaddr +
CSR6), tp->csr6);
                  }
                  if (csr5 & (TxDied | TxFIFOUnderflow | PCIBusError)) {
                        /* Restart the transmit process. */
                        outl(tp->csr6 | RxOn, ioaddr + CSR6);
                        outl(tp->csr6 | RxOn | TxOn, ioaddr + CSR6);
                  }
                  if (csr5 & (RxStopped | RxNoBuf)) {
                        /* Missed a Rx frame or mode change. */
                        tp->stats.rx_missed_errors += inl(ioaddr + CSR8) &
0xffff;
                        if (tp->flags & COMET_MAC_ADDR) {
                              outl(tp->mc_filter[0], ioaddr + 0xAC);
```

```
                              outl(tp->mc_filter[1], ioaddr + 0xB0);
                        }
                        tulip_rx(dev);
                        if (csr5 & RxNoBuf)
                              tp->rx_dead = 1;
                        outl(tp->csr6 | RxOn | TxOn, ioaddr + CSR6);
                  }
                  if (csr5 & TimerInt) {
                        if (tp->msg_level & NETIF_MSG_INTR)
                              printk(KERN_ERR "%s: Re-enabling interrupts,
%8.8x.\n",
                                          dev->name, csr5);
                        outl(tulip_tbl[tp->chip_id].valid_intrs, ioaddr +
CSR7);
                  }
                  if (csr5 & (TPLnkPass | TPLnkFail | 0x08000000)) {
                        if (tp->link_change)
                              (tp->link_change)(dev, csr5);
                  }
                  /* Clear all error sources, included undocumented ones! */
                  outl(0x0800f7ba, ioaddr + CSR5);
            }
            if (--work_budget < 0) {
                  if (tp->msg_level & NETIF_MSG_DRV)
                        printk(KERN_WARNING "%s: Too much work during an
interrupt, "
                                    "csr5=0x%8.8x.\n", dev->name, csr5);
                  /* Acknowledge all interrupt sources. */
                  outl(0x8001ffff, ioaddr + CSR5);
                  if (tp->flags & HAS_INTR_MITIGATION) {
                        /* Josip Loncaric at ICASE did extensive experimentation
                           to develop a good interrupt mitigation setting.*/
                        outl(0x8b240000, ioaddr + CSR11);
                  } else {
                        /* Mask all interrupting sources, set timer to re-enable.
*/
                        outl(((~csr5) & 0x0001ebef) | AbnormalIntr | TimerInt,
                               ioaddr + CSR7);
                        outl(0x0012, ioaddr + CSR11);
                  }
                  break;
            }
      } while (1);

      if (tp->msg_level & NETIF_MSG_INTR)
            printk(KERN_DEBUG "%s: exiting interrupt, csr5=%#4.4x.\n",
                     dev->name, (int)inl(ioaddr + CSR5));

      return;
}

static int tulip_rx(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      int entry = tp->cur_rx % RX_RING_SIZE;
      int rx_work_limit = tp->dirty_rx + RX_RING_SIZE - tp->cur_rx;
      int work_done = 0;
```

```
      if (tp->msg_level & NETIF_MSG_RX_STATUS)
            printk(KERN_DEBUG " In tulip_rx(), entry %d %8.8x.\n", entry,
                   tp->rx_ring[entry].status);
      /* If we own the next entry, it is a new packet. Send it up. */
      while ( ! (tp->rx_ring[entry].status & cpu_to_le32(DescOwned))) {
            s32 status = le32_to_cpu(tp->rx_ring[entry].status);

            if (tp->msg_level & NETIF_MSG_RX_STATUS)
                  printk(KERN_DEBUG "%s: In tulip_rx(), entry %d %8.8x.\n",
                         dev->name, entry, status);
            if (--rx_work_limit < 0)
                  break;
            if ((status & 0x38008300) != 0x0300) {
                  if ((status & 0x38000300) != 0x0300) {
                        /* Ingore earlier buffers. */
                        if ((status & 0xffff) != 0x7fff) {
                              if (tp->msg_level & NETIF_MSG_RX_ERR)
                                    printk(KERN_WARNING "%s: Oversized Ethernet
frame "
                                                        "spanned multiple buffers, status
%8.8x!\n",
                                                        dev->name, status);
                              tp->stats.rx_length_errors++;
                        }
                  } else if (status & RxDescFatalErr) {
                        /* There was a fatal error. */
                        if (tp->msg_level & NETIF_MSG_RX_ERR)
                              printk(KERN_DEBUG "%s: Receive error, Rx status
%8.8x.\n",
                                                 dev->name, status);
                        tp->stats.rx_errors++; /* end of a packet.*/
                        if (status & 0x0890) tp->stats.rx_length_errors++;
                        if (status & 0x0004) tp->stats.rx_frame_errors++;
                        if (status & 0x0002) tp->stats.rx_crc_errors++;
                        if (status & 0x0001) tp->stats.rx_fifo_errors++;
                  }
            } else {
                  /* Omit the four octet CRC from the length. */
                  short pkt_len = ((status >> 16) & 0x7ff) - 4;
                  struct sk_buff *skb;

#ifndef final_version
                  if (pkt_len > 1518) {
                        printk(KERN_WARNING "%s: Bogus packet size of %d
(%#x).\n",
                                            dev->name, pkt_len, pkt_len);
                        pkt_len = 1518;
                        tp->stats.rx_length_errors++;
                  }
#endif
                  /* Check if the packet is long enough to accept without copying
                     to a minimally-sized skbuff. */
                  if (pkt_len < tp->rx_copybreak
                        && (skb = dev_alloc_skb(pkt_len + 2)) != NULL) {
                        skb->dev = dev;
                        skb_reserve(skb, 2);    /* 16 byte align the IP header
```

```
*/
#if (LINUX_VERSION_CODE >= 0x20100)
                        eth_copy_and_sum(skb, tp->rx_skbuff[entry]->tail,
pkt_len, 0);
                        skb_put(skb, pkt_len);
#else
                        memcpy(skb_put(skb, pkt_len),
tp->rx_skbuff[entry]->tail,
                                   pkt_len);
#endif
                        work_done++;
                  } else {    /* Pass up the skb already on the Rx ring. */
                        skb_put(skb = tp->rx_skbuff[entry], pkt_len);
                        tp->rx_skbuff[entry] = NULL;
                  }
                  skb->protocol = eth_type_trans(skb, dev);
                  netif_rx(skb);
                  dev->last_rx = jiffies;
                  tp->stats.rx_packets++;
#if LINUX_VERSION_CODE > 0x20127
                  tp->stats.rx_bytes += pkt_len;
#endif
            }
            entry = (++tp->cur_rx) % RX_RING_SIZE;
      }

      /* Refill the Rx ring buffers. */
      for (; tp->cur_rx - tp->dirty_rx > 0; tp->dirty_rx++) {
            entry = tp->dirty_rx % RX_RING_SIZE;
            if (tp->rx_skbuff[entry] == NULL) {
                  struct sk_buff *skb;
                  skb = tp->rx_skbuff[entry] = dev_alloc_skb(tp->rx_buf_sz);
                  if (skb == NULL) {
                        if (tp->cur_rx - tp->dirty_rx == RX_RING_SIZE)
                              printk(KERN_ERR "%s: No kernel memory to allocate
"
                                         "receive buffers.\n", dev->name);
                        break;
                  }
                  skb->dev = dev;               /* Mark as being used by this
device. */
                  tp->rx_ring[entry].buffer1 = virt_to_le32desc(skb->tail);
                  work_done++;
            }
            tp->rx_ring[entry].status = cpu_to_le32(DescOwned);
      }

      return work_done;
}

static void empty_rings(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      int i;

      /* Free all the skbuffs in the Rx queue. */
      for (i = 0; i < RX_RING_SIZE; i++) {
```

```
            struct sk_buff *skb = tp->rx_skbuff[i];
            tp->rx_skbuff[i] = 0;
            tp->rx_ring[i].status = 0;          /* Not owned by Tulip chip. */
            tp->rx_ring[i].length = 0;
            tp->rx_ring[i].buffer1 = 0xBADF00D0; /* An invalid address. */
            if (skb) {
#if LINUX_VERSION_CODE < 0x20100
                  skb->free = 1;
#endif
                  dev_free_skb(skb);
            }
      }
      for (i = 0; i < TX_RING_SIZE; i++) {
            if (tp->tx_skbuff[i])
                  dev_free_skb(tp->tx_skbuff[i]);
            tp->tx_skbuff[i] = 0;
      }
}

static int tulip_close(struct net_device *dev)
{
      long ioaddr = dev->base_addr;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;

      netif_stop_tx_queue(dev);

      if (tp->msg_level & NETIF_MSG_IFDOWN)
            printk(KERN_DEBUG "%s: Shutting down ethercard, status was %2.2x.\n",
                   dev->name, (int)inl(ioaddr + CSR5));

      /* Disable interrupts by clearing the interrupt mask. */
      outl(0x00000000, ioaddr + CSR7);
      /* Stop the Tx and Rx processes. */
      outl(inl(ioaddr + CSR6) & ~TxOn & ~RxOn, ioaddr + CSR6);
      /* 21040 -- Leave the card in 10baseT state. */
      if (tp->chip_id == DC21040)
            outl(0x00000004, ioaddr + CSR13);

      if (inl(ioaddr + CSR6) != 0xffffffff)
            tp->stats.rx_missed_errors += inl(ioaddr + CSR8) & 0xffff;

      del_timer(&tp->timer);

      free_irq(dev->irq, dev);

      dev->if_port = tp->saved_if_port;

      empty_rings(dev);
      /* Leave the driver in snooze, not sleep, mode. */
      if (tp->flags & HAS_PWRDWN)
            pci_write_config_dword(tp->pci_dev, 0x40, 0x40000000);

      MOD_DEC_USE_COUNT;

      return 0;
}
```

```
static struct net_device_stats *tulip_get_stats(struct net_device *dev)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      int csr8 = inl(ioaddr + CSR8);

      if (netif_running(dev)  &&  csr8 != 0xffffffff)
            tp->stats.rx_missed_errors += (u16)csr8;

      return &tp->stats;
}

#ifdef HAVE_PRIVATE_IOCTL
/* Provide ioctl() calls to examine the MII xcvr state.
   We emulate a MII management registers for chips without MII.
   The two numeric constants are because some clueless person
   changed value for the symbolic name.
 */
static int private_ioctl(struct net_device *dev, struct ifreq *rq, int cmd)
{
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      u16 *data = (u16 *)&rq->ifr_data;
      u32 *data32 = (void *)&rq->ifr_data;
      unsigned int phy = tp->phys[0];
      unsigned int regnum = data[1];

      switch(cmd) {
      case 0x8947: case 0x89F0:
            /* SIOCGMIIPHY: Get the address of the PHY in use. */
            if (tp->mii_cnt)
                  data[0] = phy;
            else if (tp->flags & HAS_NWAY)
                  data[0] = 32;
            else if (tp->chip_id == COMET)
                  data[0] = 1;
            else
                  return -ENODEV;
      case 0x8948: case 0x89F1:
            /* SIOCGMIIREG: Read the specified MII register. */
            if (data[0] == 32  &&  (tp->flags & HAS_NWAY)) {
                  int csr12 = inl(ioaddr + CSR12);
                  int csr14 = inl(ioaddr + CSR14);
                  switch (regnum) {
                  case 0:
                        if (((csr14<<5) & 0x1000) ||
                              (dev->if_port == 5 && tp->nwayset))
                              data[3] = 0x1000;
                        else
                              data[3] = (media_cap[dev->if_port]&MediaIs100 ?
0x2000 : 0)
                                    | (media_cap[dev->if_port]&MediaIsFD ?
0x0100 : 0);
                        break;
                  case 1:
                        data[3] = 0x1848 + ((csr12&0x7000) == 0x5000 ? 0x20 : 0)
                              + ((csr12&0x06) == 6 ? 0 : 4);
```

```
                        if (tp->chip_id != DC21041)
                              data[3] |= 0x6048;
                        break;
                  case 4: {
                        /* Advertised value, bogus 10baseTx-FD value from CSR6.
*/
                        data[3] = ((inl(ioaddr +
CSR6)>>3)&0x0040)+((csr14>>1)&0x20)+1;
                        if (tp->chip_id != DC21041)
                               data[3] |= ((csr14>>9)&0x03C0);
                        break;
                  }
                  case 5: data[3] = tp->lpar; break;
                  default: data[3] = 0; break;
                  }
            } else {
                  data[3] = mdio_read(dev, data[0] & 0x1f, regnum);
            }
            return 0;
      case 0x8949: case 0x89F2:
            /* SIOCSMIIREG: Write the specified MII register */
            if (!capable(CAP_NET_ADMIN))
                  return -EPERM;
            if (regnum & ~0x1f)
                  return -EINVAL;
            if (data[0] == phy) {
                  u16 value = data[2];
                  switch (regnum) {
                  case 0: /* Check for autonegotiation on or reset. */
                        tp->full_duplex_lock = (value & 0x9000) ? 0 : 1;
                        if (tp->full_duplex_lock)
                              tp->full_duplex = (value & 0x0100) ? 1 : 0;
                        break;
                  case 4: tp->mii_advertise = data[2]; break;
                  }
            }
            if (data[0] == 32  &&  (tp->flags & HAS_NWAY)) {
                  u16 value = data[2];
                  if (regnum == 0) {
                        if ((value & 0x1200) == 0x1200)
                              nway_start(dev);
                  } else if (regnum == 4)
                        tp->sym_advertise = value;
            } else {
                  mdio_write(dev, data[0] & 0x1f, regnum, data[2]);
            }
            return 0;
      case SIOCGPARAMS:
            data32[0] = tp->msg_level;
            data32[1] = tp->multicast_filter_limit;
            data32[2] = tp->max_interrupt_work;
            data32[3] = tp->rx_copybreak;
            data32[4] = inl(ioaddr + CSR11);
            return 0;
      case SIOCSPARAMS:
            if (!capable(CAP_NET_ADMIN))
                  return -EPERM;
```

```
            tp->msg_level = data32[0];
            tp->multicast_filter_limit = data32[1];
            tp->max_interrupt_work = data32[2];
            tp->rx_copybreak = data32[3];
            if (tp->flags & HAS_INTR_MITIGATION) {
                  u32 *d = (u32 *)&rq->ifr_data;
                  outl(data32[4], ioaddr + CSR11);
                  printk(KERN_NOTICE "%s: Set interrupt mitigate paramters
%8.8x.\n",
                               dev->name, d[0]);
            }
            return 0;
      default:
            return -EOPNOTSUPP;
      }

      return -EOPNOTSUPP;
}
#endif  /* HAVE_PRIVATE_IOCTL */

/* Set or clear the multicast filter for this adaptor.
   Note that we only use exclusion around actually queueing the
   new frame, not around filling tp->setup_frame.  This is non-deterministic
   when re-entered but still correct. */

/* The little-endian AUTODIN32 ethernet CRC calculation.
   N.B. Do not use for bulk data, use a table-based routine instead.
   This is common code and should be moved to net/core/crc.c */
static unsigned const ethernet_polynomial_le = 0xedb88320U;
static inline u32 ether_crc_le(int length, unsigned char *data)
{
      u32 crc = 0xffffffff;   /* Initial value. */
      while(--length >= 0) {
            unsigned char current_octet = *data++;
            int bit;
            for (bit = 8; --bit >= 0; current_octet >>= 1) {
                  if ((crc ^ current_octet) & 1) {
                        crc >>= 1;
                        crc ^= ethernet_polynomial_le;
                  } else
                        crc >>= 1;
            }
      }
      return crc;
}
static unsigned const ethernet_polynomial = 0x04c11db7U;
static inline u32 ether_crc(int length, unsigned char *data)
{
      int crc = -1;

      while(--length >= 0) {
            unsigned char current_octet = *data++;
            int bit;
            for (bit = 0; bit < 8; bit++, current_octet >>= 1)
                  crc = (crc << 1) ^
                        ((crc < 0) ^ (current_octet & 1) ? ethernet_polynomial
: 0);
```

```
	}
	return crc;
}

static void set_rx_mode(struct net_device *dev)
{
	struct tulip_private *tp = (struct tulip_private *)dev->priv;
	long ioaddr = dev->base_addr;
	int csr6 = inl(ioaddr + CSR6) & ~0x00D5;

	tp->csr6 &= ~0x00D5;
	if (dev->flags & IFF_PROMISC) {			/* Set promiscuous. */
		tp->csr6 |= AcceptAllMulticast | AcceptAllPhys;
		csr6 |= AcceptAllMulticast | AcceptAllPhys;
		/* Unconditionally log net taps. */
		printk(KERN_INFO "%s: Promiscuous mode enabled.\n", dev->name);
	} else if ((dev->mc_count > tp->multicast_filter_limit)  ||
			(dev->flags & IFF_ALLMULTI)) {
		/* Too many to filter well -- accept all multicasts. */
		tp->csr6 |= AcceptAllMulticast;
		csr6 |= AcceptAllMulticast;
	} else	if (tp->flags & MC_HASH_ONLY) {
		/* Some work-alikes have only a 64-entry hash filter table. */
		/* Should verify correctness on big-endian/__powerpc__ */
		struct dev_mc_list *mclist;
		int i;
		if (dev->mc_count > tp->multicast_filter_limit) {
			tp->csr6 |= AcceptAllMulticast;
			csr6 |= AcceptAllMulticast;
		} else {
			u32 mc_filter[2] = {0, 0};		 /* Multicast hash filter
*/
			int filterbit;
			for (i = 0, mclist = dev->mc_list; mclist && i < dev->mc_count;
			     i++, mclist = mclist->next) {
				if (tp->flags & COMET_MAC_ADDR)
					filterbit = ether_crc_le(ETH_ALEN,
mclist->dmi_addr);
				else
					filterbit = ether_crc(ETH_ALEN, mclist->dmi_addr)
>> 26;
				filterbit &= 0x3f;
				set_bit(filterbit, mc_filter);
				if (tp->msg_level & NETIF_MSG_RXFILTER)
					printk(KERN_INFO "%s: Added filter for
%2.2x:%2.2x:%2.2x:"
					       "%2.2x:%2.2x:%2.2x  %8.8x bit %d.\n",
dev->name,
					       mclist->dmi_addr[0],
mclist->dmi_addr[1],
					       mclist->dmi_addr[2],
mclist->dmi_addr[3],
					       mclist->dmi_addr[4],
mclist->dmi_addr[5],
					       ether_crc(ETH_ALEN, mclist->dmi_addr),
filterbit);
			}
```

```
                if (mc_filter[0] == tp->mc_filter[0]  &&
                      mc_filter[1] == tp->mc_filter[1])
                      ;                          /* No change. */
                else if (tp->flags & IS_ASIX) {
                      outl(2, ioaddr + CSR13);
                      outl(mc_filter[0], ioaddr + CSR14);
                      outl(3, ioaddr + CSR13);
                      outl(mc_filter[1], ioaddr + CSR14);
                } else if (tp->flags & COMET_MAC_ADDR) {
                      outl(mc_filter[0], ioaddr + 0xAC);
                      outl(mc_filter[1], ioaddr + 0xB0);
                }
                tp->mc_filter[0] = mc_filter[0];
                tp->mc_filter[1] = mc_filter[1];
          }
     } else {
          u16 *eaddrs, *setup_frm = tp->setup_frame;
          struct dev_mc_list *mclist;
          u32 tx_flags = 0x08000000 | 192;
          int i;

          /* Note that only the low-address shortword of setup_frame is valid!
             The values are doubled for big-endian architectures. */
          if (dev->mc_count > 14) { /* Must use a multicast hash table. */
                u16 hash_table[32];
                tx_flags = 0x08400000 | 192;        /* Use hash filter. */
                memset(hash_table, 0, sizeof(hash_table));
                set_bit(255, hash_table);                 /* Broadcast entry
*/
                /* This should work on big-endian machines as well. */
                for (i = 0, mclist = dev->mc_list; mclist && i < dev->mc_count;
                       i++, mclist = mclist->next)
                      set_bit(ether_crc_le(ETH_ALEN, mclist->dmi_addr) &
0x1ff,
                                  hash_table);
                for (i = 0; i < 32; i++) {
                      *setup_frm++ = hash_table[i];
                      *setup_frm++ = hash_table[i];
                }
                setup_frm = &tp->setup_frame[13*6];
          } else {
                /* We have <= 14 addresses so we can use the wonderful
                   16 address perfect filtering of the Tulip. */
                for (i = 0, mclist = dev->mc_list; i < dev->mc_count;
                       i++, mclist = mclist->next) {
                      eaddrs = (u16 *)mclist->dmi_addr;
                      *setup_frm++ = *eaddrs; *setup_frm++ = *eaddrs++;
                      *setup_frm++ = *eaddrs; *setup_frm++ = *eaddrs++;
                      *setup_frm++ = *eaddrs; *setup_frm++ = *eaddrs++;
                }
                /* Fill the unused entries with the broadcast address. */
                memset(setup_frm, 0xff, (15-i)*12);
                setup_frm = &tp->setup_frame[15*6];
          }
          /* Fill the final entry with our physical address. */
          eaddrs = (u16 *)dev->dev_addr;
          *setup_frm++ = eaddrs[0]; *setup_frm++ = eaddrs[0];
```

```
            *setup_frm++ = eaddrs[1]; *setup_frm++ = eaddrs[1];
            *setup_frm++ = eaddrs[2]; *setup_frm++ = eaddrs[2];
            /* Now add this frame to the Tx list. */
            if (tp->cur_tx - tp->dirty_tx > TX_RING_SIZE - 2) {
                  /* Same setup recently queued, we need not add it. */
            } else {
                  unsigned long flags;
                  unsigned int entry;

                  spin_lock_irqsave(&tp->mii_lock, flags);
                  entry = tp->cur_tx++ % TX_RING_SIZE;

                  if (entry != 0) {
                        /* Avoid a chip errata by prefixing a dummy entry. */
                        tp->tx_skbuff[entry] = 0;
                        tp->tx_ring[entry].length =
                              (entry == TX_RING_SIZE-1) ?
cpu_to_le32(DESC_RING_WRAP):0;
                        tp->tx_ring[entry].buffer1 = 0;
                        tp->tx_ring[entry].status = cpu_to_le32(DescOwned);
                        entry = tp->cur_tx++ % TX_RING_SIZE;
                  }

                  tp->tx_skbuff[entry] = 0;
                  /* Put the setup frame on the Tx list. */
                  if (entry == TX_RING_SIZE-1)
                        tx_flags |= DESC_RING_WRAP;         /* Wrap ring. */
                  tp->tx_ring[entry].length = cpu_to_le32(tx_flags);
                  tp->tx_ring[entry].buffer1 =
virt_to_le32desc(tp->setup_frame);
                  tp->tx_ring[entry].status = cpu_to_le32(DescOwned);
                  if (tp->cur_tx - tp->dirty_tx >= TX_RING_SIZE - 2) {
                        netif_stop_tx_queue(dev);
                        tp->tx_full = 1;
                  }
                  spin_unlock_irqrestore(&tp->mii_lock, flags);
                  /* Trigger an immediate transmit demand. */
                  outl(0, ioaddr + CSR1);
            }
      }
      outl(csr6, ioaddr + CSR6);
}
```

```
static int tulip_pwr_event(void *dev_instance, int event)
{
      struct net_device *dev = dev_instance;
      struct tulip_private *tp = (struct tulip_private *)dev->priv;
      long ioaddr = dev->base_addr;
      if (tp->msg_level & NETIF_MSG_LINK)
            printk("%s: Handling power event %d.\n", dev->name, event);
      switch(event) {
      case DRV_ATTACH:
            MOD_INC_USE_COUNT;
            break;
      case DRV_SUSPEND: {
            int csr6 = inl(ioaddr + CSR6);
            /* Disable interrupts, stop the chip, gather stats. */
            if (csr6 != 0xffffffff) {
                  int csr8 = inl(ioaddr + CSR8);
                  outl(0x00000000, ioaddr + CSR7);
                  outl(csr6 & ~TxOn & ~RxOn, ioaddr + CSR6);
                  tp->stats.rx_missed_errors += (unsigned short)csr8;
            }
            empty_rings(dev);
            /* Put the 21143 into sleep mode. */
            if (tp->flags & HAS_PWRDWN)
                  pci_write_config_dword(tp->pci_dev, 0x40,0x80000000);
            break;
      }
      case DRV_RESUME:
            if (tp->flags & HAS_PWRDWN)
                  pci_write_config_dword(tp->pci_dev, 0x40, 0x0000);
            outl(tp->csr0, ioaddr + CSR0);
            tulip_init_ring(dev);
            outl(virt_to_bus(tp->rx_ring), ioaddr + CSR3);
            outl(virt_to_bus(tp->tx_ring), ioaddr + CSR4);
            if (tp->mii_cnt) {
                  dev->if_port = 11;
                  if (tp->mtable  &&  tp->mtable->has_mii)
                        select_media(dev, 1);
                  tp->csr6 = 0x820E0000;
                  dev->if_port = 11;
                  outl(0x0000, ioaddr + CSR13);
                  outl(0x0000, ioaddr + CSR14);
            } else if (! tp->medialock)
                  nway_start(dev);
            else
                  select_media(dev, 1);
            outl(tulip_tbl[tp->chip_id].valid_intrs, ioaddr + CSR7);
            outl(tp->csr6 | TxOn | RxOn, ioaddr + CSR6);
            outl(0, ioaddr + CSR2);       /* Rx poll demand */
            set_rx_mode(dev);
            break;
      case DRV_DETACH: {
            struct net_device **devp, **next;
            if (dev->flags & IFF_UP) {
                  printk(KERN_ERR "%s: Tulip CardBus interface was detached while
"
```

```
                        "still active.\n", dev->name);
                dev_close(dev);
                dev->flags &= ~(IFF_UP|IFF_RUNNING);
            }
            if (tp->msg_level & NETIF_MSG_DRV)
                printk(KERN_DEBUG "%s: Unregistering device.\n", dev->name);
            unregister_netdev(dev);
#ifdef USE_IO_OPS
            release_region(dev->base_addr, pci_id_tbl[tp->chip_id].io_size);
#else
            iounmap((char *)dev->base_addr);
#endif
            for (devp = &root_tulip_dev; *devp; devp = next) {
                next = &((struct tulip_private *)(*devp)->priv)->next_module;
                if (*devp == dev) {
                    *devp = *next;
                    break;
                }
            }
            if (tp->priv_addr)
                kfree(tp->priv_addr);
            kfree(dev);
            MOD_DEC_USE_COUNT;
            break;
        }
        default:
            break;
        }

        return 0;
}

#ifdef CARDBUS

#include <pcmcia/driver_ops.h>

static dev_node_t *tulip_attach(dev_locator_t *loc)
{
        struct net_device *dev;
        long ioaddr;
        struct pci_dev *pdev;
        u8 bus, devfn, irq;
        u32 dev_id;
        u32 pciaddr;
        int i, chip_id = 4;                 /* DC21143 */

        if (loc->bus != LOC_PCI) return NULL;
        bus = loc->b.pci.bus; devfn = loc->b.pci.devfn;
        printk(KERN_INFO "tulip_attach(bus %d, function %d)\n", bus, devfn);
        pdev = pci_find_slot(bus, devfn);
#ifdef USE_IO_OPS
        pci_read_config_dword(pdev, PCI_BASE_ADDRESS_0, &pciaddr);
        ioaddr = pciaddr & PCI_BASE_ADDRESS_IO_MASK;
#else
        pci_read_config_dword(pdev, PCI_BASE_ADDRESS_1, &pciaddr);
        ioaddr = (long)ioremap(pciaddr & PCI_BASE_ADDRESS_MEM_MASK,
                                        pci_id_tbl[DC21142].io_size);
```

```
#endif
      pci_read_config_dword(pdev, 0, &dev_id);
      pci_read_config_byte(pdev, PCI_INTERRUPT_LINE, &irq);
      if (ioaddr == 0 || irq == 0) {
            printk(KERN_ERR "The Tulip CardBus Ethernet interface at %d/%d was
"
                        "not assigned an %s.\n"
                        KERN_ERR "  It will not be activated.\n",
                        bus, devfn, ioaddr == 0 ? "address" : "IRQ");
            return NULL;
      }
      for (i = 0; pci_id_tbl[i].id.pci; i++) {
            if (pci_id_tbl[i].id.pci == (dev_id & pci_id_tbl[i].id.pci_mask)) {
                  chip_id = i; break;
            }
      }
      dev = tulip_probe1(pdev, NULL, ioaddr, irq, chip_id, 0);
      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 tulip_suspend(dev_node_t *node)
{
      struct net_device **devp, **next;
      printk(KERN_INFO "tulip_suspend(%s)\n", node->dev_name);
      for (devp = &root_tulip_dev; *devp; devp = next) {
            next = &((struct tulip_private *)(*devp)->priv)->next_module;
            if (strcmp((*devp)->name, node->dev_name) == 0) {
                  tulip_pwr_event(*devp, DRV_SUSPEND);
                  break;
            }
      }
}

static void tulip_resume(dev_node_t *node)
{
      struct net_device **devp, **next;
      printk(KERN_INFO "tulip_resume(%s)\n", node->dev_name);
      for (devp = &root_tulip_dev; *devp; devp = next) {
            next = &((struct tulip_private *)(*devp)->priv)->next_module;
            if (strcmp((*devp)->name, node->dev_name) == 0) {
                  tulip_pwr_event(*devp, DRV_RESUME);
                  break;
            }
      }
}

static void tulip_detach(dev_node_t *node)
{
      struct net_device **devp, **next;
```

```
      printk(KERN_INFO "tulip_detach(%s)\n", node->dev_name);
      for (devp = &root_tulip_dev; *devp; devp = next) {
            next = &((struct tulip_private *)(*devp)->priv)->next_module;
            if (strcmp((*devp)->name, node->dev_name) == 0) break;
      }
      if (*devp) {
            struct tulip_private *tp = (struct tulip_private *)(*devp)->priv;
            unregister_netdev(*devp);
#ifdef USE_IO_OPS
            release_region((*devp)->base_addr, pci_id_tbl[DC21142].io_size);
#else
            iounmap((char *)(*devp)->base_addr);
#endif
            kfree(*devp);
            if (tp->priv_addr)
                  kfree(tp->priv_addr);
            *devp = *next;
            kfree(node);
            MOD_DEC_USE_COUNT;
      }
}

struct driver_operations tulip_ops = {
      "tulip_cb", tulip_attach, tulip_suspend, tulip_resume, tulip_detach
};

#endif  /* Cardbus support */
```

```
#ifdef MODULE
int init_module(void)
{
      if (debug >= NETIF_MSG_DRV)   /* Emit version even if no cards detected.
*/
            printk(KERN_INFO "%s" KERN_INFO "%s", version1, version2);
#ifdef CARDBUS
      register_driver(&tulip_ops);
      return 0;
#else
      return pci_drv_register(&tulip_drv_id, NULL);
#endif
      reverse_probe = 0;                  /* Not used. */
}

void cleanup_module(void)
{
      struct net_device *next_dev;

#ifdef CARDBUS
      unregister_driver(&tulip_ops);
#else
      pci_drv_unregister(&tulip_drv_id);
#endif

      /* No need to check MOD_IN_USE, as sys_delete_module() checks. */
      while (root_tulip_dev) {
            struct tulip_private *tp = (struct
tulip_private*)root_tulip_dev->priv;
            unregister_netdev(root_tulip_dev);
#ifdef USE_IO_OPS
            release_region(root_tulip_dev->base_addr,
                           pci_id_tbl[tp->chip_id].io_size);
#else
            iounmap((char *)root_tulip_dev->base_addr);
#endif
            next_dev = tp->next_module;
            if (tp->priv_addr)
                  kfree(tp->priv_addr);
            kfree(root_tulip_dev);
            root_tulip_dev = next_dev;
      }
}
#else
int tulip_probe(struct net_device *dev)
{
      if (pci_drv_register(&tulip_drv_id, dev) < 0)
            return -ENODEV;
      printk(KERN_INFO "%s" KERN_INFO "%s", version1, version2);
      return 0;
      reverse_probe = 0;                  /* Not used. */
}
#endif  /* MODULE */
```

```
/*
 * Local variables:
 *  compile-command: "make KERNVER=`uname -r` tulip.o"
 *  compile-cmd: "gcc -DMODULE -Wall -Wstrict-prototypes -O6 -c tulip.c"
 *  cardbus-compile-command: "gcc -DCARDBUS -DMODULE -Wall -Wstrict-prototypes
-O6 -c tulip.c -o tulip_cb.o -I/usr/src/pcmcia/include/"
 *  c-indent-level: 4
 *  c-basic-offset: 4
 *  tab-width: 4
 * End:
 */
```