```c
/* skeleton.c: A network driver outline for linux.
 *
 *      Written 1993-94 by Donald Becker.
 *
 *      Copyright 1993 United States Government as represented by the
 *      Director, National Security Agency.
 *
 *      This software may be used and distributed according to the terms
 *      of the GNU Public License, incorporated herein by reference.
 *
 *      The author may be reached as becker@CESDIS.gsfc.nasa.gov, or C/O
 *      Center of Excellence in Space Data and Information Sciences
 *         Code 930.5, Goddard Space Flight Center, Greenbelt MD 20771
 *
 *      This file is an outline for writing a network device driver for the
 *      the Linux operating system.
 *
 *      To write (or understand) a driver, have a look at the
 *      "loopback.c" file to get a feel of what is going on, and then
 *      use the code below as a skeleton for the new driver.
 */

static const char *version =
        "skeleton.c:v1.51 9/24/94 Donald Becker\n";

/*
 *  Sources:
 *      List your sources of programming information to document that
 *      the driver is your own creation, and give due credit to others
 *      that contributed to the work. Remember that GNU project code
 *      cannot use proprietary or trade secret information. Interface
 *      definitions are generally considered non-copyrightable to the
 *      extent that the same names and structures must be used to be
 *      compatible.
 *
 *      Finally, keep in mind that the Linux kernel is has an API, not
 *      ABI. Proprietary object-code-only distributions are not permitted
 *      under the GPL.
 */

#include <linux/module.h>

#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/types.h>
#include <linux/fcntl.h>
#include <linux/interrupt.h>
#include <linux/ptrace.h>
#include <linux/ioport.h>
#include <linux/in.h>
#include <linux/malloc.h>
#include <linux/string.h>
#include <asm/system.h>
#include <asm/bitops.h>
#include <asm/io.h>
#include <asm/dma.h>
#include <linux/errno.h>

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

```
#include <linux/skbuff.h>

/*
 * The name of the card. Is used for messages and in the requests for
 * io regions, irqs and dma channels
 */
static const char* cardname = "netcard";

/* First, a few definitions that the brave might change. */

/* A zero-terminated list of I/O addresses to be probed. */
static unsigned int netcard_portlist[] =
   { 0x200, 0x240, 0x280, 0x2C0, 0x300, 0x320, 0x340, 0};

/* use 0 for production, 1 for verification, >2 for debug */
#ifndef NET_DEBUG
#define NET_DEBUG 2
#endif
static unsigned int net_debug = NET_DEBUG;

/* The number of low I/O ports used by the ethercard. */
#define NETCARD_IO_EXTENT       32

/* Information that need to be kept for each board. */
struct net_local {
        struct enet_statistics stats;
        long open_time;                 /* Useless example local info. */
};

/* The station (ethernet) address prefix, used for IDing the board. */
#define SA_ADDR0 0x00
#define SA_ADDR1 0x42
#define SA_ADDR2 0x65

/* Index to functions, as function prototypes. */

extern int netcard_probe(struct device *dev);

static int netcard_probe1(struct device *dev, int ioaddr);
static int net_open(struct device *dev);
static int      net_send_packet(struct sk_buff *skb, struct device *dev);
static void net_interrupt(int irq, void *dev_id, struct pt_regs *regs);
static void net_rx(struct device *dev);
static int net_close(struct device *dev);
static struct enet_statistics *net_get_stats(struct device *dev);
static void set_multicast_list(struct device *dev);

/* Example routines you must write ;->. */
#define tx_done(dev) 1
extern void     hardware_send_packet(short ioaddr, char *buf, int length);
extern void chipset_init(struct device *dev, int startp);

/*
 * Check for a network adaptor of this type, and return '0' iff one exists.
 * If dev->base_addr == 0, probe all likely locations.
 * If dev->base_addr == 1, always return failure.
 * If dev->base_addr == 2, allocate space for the device and return success
 * (detachable devices only).
 */
#ifdef HAVE_DEVLIST
```

```
/*
 * Support for a alternate probe manager,
 * which will eliminate the boilerplate below.
 */
struct netdev_entry netcard_drv =
{cardname, netcard_probe1, NETCARD_IO_EXTENT, netcard_portlist};
#else
int
netcard_probe(struct device *dev)
{
        int i;
        int base_addr = dev ? dev->base_addr : 0;

        if (base_addr > 0x1ff)    /* Check a single specified location. */
                return netcard_probe1(dev, base_addr);
        else if (base_addr != 0)  /* Don't probe at all. */
                return -ENXIO;

        for (i = 0; netcard_portlist[i]; i++) {
                int ioaddr = netcard_portlist[i];
                if (check_region(ioaddr, NETCARD_IO_EXTENT))
                        continue;
                if (netcard_probe1(dev, ioaddr) == 0)
                        return 0;
        }

        return -ENODEV;
}
#endif

/*
 * This is the real probe routine. Linux has a history of friendly
 * device probes on the ISA bus. A good device probes avoids doing
 * writes, and verifies that the correct device exists and functions.
 */
static int netcard_probe1(struct device *dev, int ioaddr)
{
        static unsigned version_printed = 0;
        int i;

        /*
         * For ethernet adaptors the first three octets of the station
         * address contains the manufacturer's unique code. That might
         * be a good probe method. Ideally you would add additional
         * checks.
         */
        if (inb(ioaddr + 0) != SA_ADDR0
                ||       inb(ioaddr + 1) != SA_ADDR1
                ||       inb(ioaddr + 2) != SA_ADDR2) {
                return -ENODEV;
        }

        /* Allocate a new 'dev' if needed. */
        if (dev == NULL) {
                /*
                 * Don't allocate the private data here, it is done
                 * later This makes it easier to free the memory when
                 * this driver is used as a module.
                 */
                dev = init_etherdev(0, 0);
```

```
                if (dev == NULL)
                        return -ENOMEM;
        }

        if (net_debug  &&  version_printed++ == 0)
                printk(KERN_DEBUG "%s", version);

        printk(KERN_INFO "%s: %s found at %#3x, ", dev->name,
               cardname, ioaddr);

        /* Fill in the ’dev’ fields. */
        dev->base_addr = ioaddr;

        /* Retrieve and print the ethernet address. */
        for (i = 0; i < 6; i++)
                printk(" %2.2x", dev->dev_addr[i] = inb(ioaddr + i));

#ifdef jumpered_interrupts
        /*
         * If this board has jumpered interrupts, allocate the
         * interrupt vector now. There is no point in waiting since no
         * other device can use the interrupt, and this marks the irq
         * as busy. Jumpered interrupts are typically not reported by
         * the boards, and we must used autoIRQ to find them.
         */

        if (dev->irq == -1)
                ;       /* Do nothing: a user-level program will set it. */
        else if (dev->irq < 2) {        /* "Auto-IRQ" */
                autoirq_setup(0);
                /* Trigger an interrupt here. */

                dev->irq = autoirq_report(0);
                if (net_debug >= 2)
                        printk(" autoirq is %d", dev->irq);
        } else if (dev->irq == 2)
                /*
                 * Fixup for users that don’t know that IRQ 2 is really
                 * IRQ9, or don’t know which one to set.
                 */
                dev->irq = 9;

        {
                int irqval = request_irq(dev->irq, &net_interrupt,
                                         0, cardname, NULL);
                if (irqval) {
                        printk("%s: unable to get IRQ %d (irqval=%d).\n",
                                      dev->name, dev->irq, irqval);
                        return -EAGAIN;
                }
        }
#endif  /* jumpered interrupt */
#ifdef jumpered_dma
        /*
         * If we use a jumpered DMA channel, that should be probed for
         * and allocated here as well. See lance.c for an example.
         */
        if (dev->dma == 0) {
                if (request_dma(dev->dma, cardname)) {
                        printk("DMA %d allocation failed.\n", dev->dma);
```

```
				return -EAGAIN;
			} else
				printk(", assigned DMA %d.\n", dev->dma);
		} else {
			short dma_status, new_dma_status;

			/* Read the DMA channel status registers. */
			dma_status = ((inb(DMA1_STAT_REG) >> 4) & 0x0f) |
				(inb(DMA2_STAT_REG) & 0xf0);
			/* Trigger a DMA request, perhaps pause a bit. */
			outw(0x1234, ioaddr + 8);
			/* Re-read the DMA status registers. */
			new_dma_status = ((inb(DMA1_STAT_REG) >> 4) & 0x0f) |
				(inb(DMA2_STAT_REG) & 0xf0);
			/*
			 * Eliminate the old and floating requests,
			 * and DMA4 the cascade.
			 */
			new_dma_status ^= dma_status;
			new_dma_status &= ~0x10;
			for (i = 7; i > 0; i--)
				if (test_bit(i, &new_dma_status)) {
					dev->dma = i;
					break;
				}
			if (i <= 0) {
				printk("DMA probe failed.\n");
				return -EAGAIN;
			}
			if (request_dma(dev->dma, cardname)) {
				printk("probed DMA %d allocation failed.\n",
				       dev->dma);
				return -EAGAIN;
			}
		}
#endif	/* jumpered DMA */

		/* Initialize the device structure. */
		if (dev->priv == NULL) {
			dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
			if (dev->priv == NULL)
				return -ENOMEM;
		}

		memset(dev->priv, 0, sizeof(struct net_local));

		/*
		 * Grab the region so that no one else tries to probe
		 * our ioports.
		 */
		request_region(ioaddr, NETCARD_IO_EXTENT, cardname);

		dev->open		= net_open;
		dev->stop		= net_close;
		dev->hard_start_xmit = net_send_packet;
		dev->get_stats	= net_get_stats;
		dev->set_multicast_list = &set_multicast_list;

		/*
		 * Fill in the fields of the device structure with
```

```
         * ethernet values.
         */
        ether_setup(dev);

        return 0;
}

/*
 * Open/initialize the board. This is called (in the current kernel)
 * sometime after booting when the 'ifconfig' program is run.
 *
 * This routine should set everything up anew at each open, even
 * registers that "should" only need to be set once at boot, so that
 * there is non-reboot way to recover if something goes wrong.
 */
static int
net_open(struct device *dev)
{
        struct net_local *lp = (struct net_local *)dev->priv;
        int ioaddr = dev->base_addr;
        /*
         * This is used if the interrupt line can turned off (shared).
         * See 3c503.c for an example of selecting the IRQ at config-time.
         */
        if (request_irq(dev->irq, &net_interrupt, 0, cardname, NULL)) {
                return -EAGAIN;
        }
        /*
         * Always allocate the DMA channel after the IRQ,
         * and clean up on failure.
         */
        if (request_dma(dev->dma, cardname)) {
                free_irq(dev->irq, NULL);
                return -EAGAIN;
        }
        irq2dev_map[dev->irq] = dev;

        /*
         * Reset the hardware here. Don't forget to set the
         * station address.
         */
        /*chipset_init(dev, 1);*/
        outb(0x00, ioaddr);
        lp->open_time = jiffies;

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

        MOD_INC_USE_COUNT;

        return 0;
}

static int
net_send_packet(struct sk_buff *skb, struct device *dev)
{
        struct net_local *lp = (struct net_local *)dev->priv;
        int ioaddr = dev->base_addr;

```

```
        if (dev->tbusy) {
                /*
                 * If we get here, some higher level has decided we
                 * are broken.  There should really be a "kick me"
                 * function call instead.
                 */
                int tickssofar = jiffies - dev->trans_start;
                if (tickssofar < 5)
                        return 1;
                printk(KERN_WARNING "%s: transmit timed out, %s?\n",
                       dev->name, (tx_done(dev) ? "IRQ conflict" :
                                   "network cable problem"));
                /* Try to restart the adaptor. */
                chipset_init(dev, 1);
                dev->tbusy=0;
                dev->trans_start = jiffies;
        }
        /*
         * If some higher layer thinks we've missed an tx-done
         * interrupt we are passed NULL. Caution: dev_tint() handles
         * the cli()/sti() itself.
         */
        if (skb == NULL) {
                dev_tint(dev);
                return 0;
        }
        /*
         * Block a timer-based transmit from overlapping. This could
         * better be done with atomic_swap(1, dev->tbusy), but
         * set_bit() works as well.
         */
        if (set_bit(0, (void*)&dev->tbusy) != 0)
                printk(KERN_WARNING "%s: Transmitter access conflict.\n",
                       dev->name);
        else {
                short length = ETH_ZLEN < skb->len ? skb->len : ETH_ZLEN;
                unsigned char *buf = skb->data;

                hardware_send_packet(ioaddr, buf, length);
                dev->trans_start = jiffies;
        }
        dev_kfree_skb (skb, FREE_WRITE);

        /* You might need to clean up and record Tx statistics here. */
        if (inw(ioaddr) == /*RU*/81)
                lp->stats.tx_aborted_errors++;

        return 0;
}

/*
 * The typical workload of the driver:
 *   Handle the network interface interrupts.
 */
static void
net_interrupt(int irq, void *dev_id, struct pt_regs * regs)
{
        struct device *dev = (struct device *)(irq2dev_map[irq]);
        struct net_local *lp;
        int ioaddr, status, boguscount = 0;
```

```
	if (dev == NULL) {
		printk(KERN_WARNING "%s: irq %d for unknown device.\n",
		       cardname, irq);
		return;
	}
	dev->interrupt = 1;

	ioaddr = dev->base_addr;
	lp = (struct net_local *)dev->priv;
	status = inw(ioaddr + 0);

	do {
		if (status /*& RX_INTR*/) {
			/* Got a packet(s). */
			net_rx(dev);
		}
		if (status /*& TX_INTR*/) {
			lp->stats.tx_packets++;
			dev->tbusy = 0;
			mark_bh(NET_BH);	/* Inform upper layers. */
		}
		if (status /*& COUNTERS_INTR*/) {
			/* Increment the appropriate 'localstats' field. */
			lp->stats.tx_window_errors++;
		}
	} while (++boguscount < 20) ;

	dev->interrupt = 0;
	return;
}

/* We have a good packet(s), get it/them out of the buffers. */
static void
net_rx(struct device *dev)
{
	struct net_local *lp = (struct net_local *)dev->priv;
	int ioaddr = dev->base_addr;
	int boguscount = 10;

	do {
		int status = inw(ioaddr);
		int pkt_len = inw(ioaddr);

		if (pkt_len == 0)		/* Read all the frames? */
			break;			/* Done for now */

		if (status & 0x40) {	/* There was an error. */
			lp->stats.rx_errors++;
			if (status & 0x20) lp->stats.rx_frame_errors++;
			if (status & 0x10) lp->stats.rx_over_errors++;
			if (status & 0x08) lp->stats.rx_crc_errors++;
			if (status & 0x04) lp->stats.rx_fifo_errors++;
		} else {
			/* Malloc up new buffer. */
			struct sk_buff *skb;

			skb = dev_alloc_skb(pkt_len);
			if (skb == NULL) {
				printk(KERN_NOTICE "%s: Memory squeeze, "
```

```
                                        "dropping packet.\n", dev->name);
                                lp->stats.rx_dropped++;
                                break;
                        }
                        skb->dev = dev;

                        /*
                         * 'skb->data' points to the start of sk_buff
                         * data area.
                         */
                        memcpy(skb_put(skb,pkt_len),
                               (void*)dev->rmem_start, pkt_len);
                        /* or */
                        insw(ioaddr, skb->data, (pkt_len + 1) >> 1);

                        netif_rx(skb);
                        lp->stats.rx_packets++;
                }
        } while (--boguscount);

        /*
         * If any worth-while packets have been received, dev_rint()
         * has done a mark_bh(NET_BH) for us and will work on them
         * when we get to the bottom-half routine.
         */
        return;
}

/* The inverse routine to net_open(). */
static int
net_close(struct device *dev)
{
        struct net_local *lp = (struct net_local *)dev->priv;
        int ioaddr = dev->base_addr;

        lp->open_time = 0;

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

        /* Flush the Tx and disable Rx here. */

        disable_dma(dev->dma);

        /* If not IRQ or DMA jumpered, free up the line. */
        outw(0x00, ioaddr+0);   /* Release the physical interrupt line. */

        free_irq(dev->irq, NULL);
        free_dma(dev->dma);

        irq2dev_map[dev->irq] = 0;

        /* Update the statistics here. */

        MOD_DEC_USE_COUNT;

        return 0;

}
```

```
/*
 * Get the current statistics.
 * This may be called with the card open or closed.
 */
static struct enet_statistics *
net_get_stats(struct device *dev)
{
        struct net_local *lp = (struct net_local *)dev->priv;
        short ioaddr = dev->base_addr;

        cli();
        /* Update the statistics from the device registers. */
        lp->stats.rx_missed_errors = inw(ioaddr+1);
        sti();

        return &lp->stats;
}

/*
 * Set or clear the multicast filter for this adaptor.
 * num_addrs == -1      Promiscuous mode, receive all packets
 * num_addrs == 0       Normal mode, clear multicast list
 * num_addrs > 0        Multicast mode, receive normal and MC packets,
 *                      and do best-effort filtering.
 */
static void
set_multicast_list(struct device *dev)
{
        short ioaddr = dev->base_addr;
        if (dev->flags&IFF_PROMISC)
        {
                /* Enable promiscuous mode */
                outw(MULTICAST|PROMISC, ioaddr);
        }
        else if((dev->flags&IFF_ALLMULTI) || dev->mc_count > HW_MAX_ADDRS)
        {
                /* Disable promiscuous mode, use normal mode. */
                hardware_set_filter(NULL);

                outw(MULTICAST, ioaddr);
        }
        else if(dev->mc_count)
        {
                /* Walk the address list, and load the filter */
                hardware_set_filter(dev->mc_list);

                outw(MULTICAST, ioaddr);
        }
        else
                outw(0, ioaddr);
}

#ifdef MODULE

static char devicename[9] = { 0, };
static struct device this_device = {
        devicename, /* will be inserted by linux/drivers/net/net_init.c */
        0, 0, 0, 0,
        0, 0,  /* I/O address, IRQ */
        0, 0, 0, NULL, netcard_probe };
```

```
static int io = 0x300;
static int irq = 0;
static int dma = 0;
static int mem = 0;

int init_module(void)
{
        int result;

        if (io == 0)
                printk(KERN_WARNING "%s: You shouldn’t use "
                       "auto-probing with insmod!\n", cardname);

        /* Copy the parameters from insmod into the device structure. */
        this_device.base_addr = io;
        this_device.irq       = irq;
        this_device.dma       = dma;
        this_device.mem_start = mem;

        if ((result = register_netdev(&this_device)) != 0)
                return result;

        return 0;
}

void
cleanup_module(void)
{
        /* No need to check MOD_IN_USE, as sys_delete_module() checks. */
        unregister_netdev(&this_device);
        /*
         * If we don’t do this, we can’t re-insmod it later.
         * Release irq/dma here, when you have jumpered versions and
         * allocate them in net_probe1().
         */
        /*
           free_irq(this_device.irq, NULL);
           free_dma(this_device.dma);
        */
        release_region(this_device.base_addr, NETCARD_IO_EXTENT);

        if (this_device.priv)
                kfree_s(this_device.priv, sizeof(struct net_local));
}

#endif /* MODULE */

/*
 * Local variables:
 *  compile-command:
 *      gcc -D__KERNEL__ -Wall -Wstrict-prototypes -Wwrite-strings
 *      -Wredundant-decls -O2 -m486 -c skeleton.c
 *  version-control: t
 *  kept-new-versions: 5
 *  tab-width: 4
 *  c-indent-level: 4
 * End:
 */
```