```c
/*
 * Rocketport device driver for Linux
 *
 * Written by Theodore Ts'o, 1995, 1996.
 *
 * Copyright (C) 1995, 1996 by Comtrol, Inc.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of the
 * License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 * Minor number schema:
 *
 * +-------------------------------+
 * | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
 * +---+-------+-------+-----------+
 * | C | Board |  AIOP | Port #    |
 * +---+-------+-------+-----------+
 *
 * C=0 implements normal POSIX tty.
 * C=1 is reserved for the callout device.
 *
 * Board can only be 0 (we only support one board at this time).
 *
 * Normally, the user won't have to worry about the AIOP; as far as
 * the user is concerned, the lower 5 bits of the minor number address
 * the ports on a particular board (from 0 up to 32).
 */

/* Kernel includes */

#include <linux/config.h>
#include <linux/version.h>

#if (defined(CONFIG_PCI) && (LINUX_VERSION_CODE >= 131072))
#define ENABLE_PCI
#endif

#if (LINUX_VERSION_CODE > 66304)
#define NEW_MODULES
#define MODULE
#endif

#ifdef NEW_MODULES
#ifdef MODVERSIONS
#include <linux/modversions.h>
#endif
#include <linux/module.h>
```

```
#include <linux/config.h>
#else /* !NEW_MODULES */
#ifdef MODVERSIONS
#define MODULE
#endif
#include <linux/config.h>
#include <linux/module.h>
#endif /* NEW_MODULES */

#include <linux/errno.h>
#include <linux/major.h>
#include <linux/kernel.h>
#include <linux/signal.h>
#include <linux/malloc.h>
#include <linux/mm.h>

#include <linux/sched.h>
#include <linux/timer.h>
#include <linux/interrupt.h>
#include <linux/tty.h>
#include <linux/tty_flip.h>
#include <linux/string.h>
#include <linux/fcntl.h>
#include <linux/ptrace.h>
#include <linux/major.h>
#include <linux/ioport.h>
#ifdef ENABLE_PCI
#include <linux/bios32.h>
#include <linux/pci.h>
#endif

#include "rcktldef.h"
#include "rckt.h"
#include "linux.h"
#include "rocket.h"
#include "version.h"

#define ROCKET_PARANOIA_CHECK

#undef ROCKET_DEBUG_OPEN
#undef ROCKET_DEBUG_INTR
#undef ROCKET_DEBUG_WRITE
#undef ROCKET_DEBUG_FLOW
#undef ROCKET_DEBUG_THROTTLE

/*
 * CAUTION!!!!!  The TIME_STAT Function relies on the Pentium 64 bit
 * register.  For various reasons related to 1.2.13, the test for this
 * register is omitted from this driver.  If you are going to enable
 * this option, make sure you are running a Pentium CPU and that a cat
 * of /proc/cpuinfo shows ability TS Counters as Yes.  Warning part
 * done, don't cry to me if you enable this options and things won't
 * work.  If it gives you any problems, then disable the option.  The
 * code in this function is pretty straight forward, if it breaks on
 * your CPU, there is probably something funny about your CPU.
 */

/*
 * For performing timing statistics on driver.  Produces printks, one
 * every TIME_COUNTER loops, eats some of your CPU time.  Good for
```

```
 * testing or other checking, otherwise, leave it undefed.  By Doug
 * Ledford
 */
#undef TIME_STAT

/*
 * This needs to be set to your processor speed For example, 100Mhz
 * CPU, set this to 100
 */
#define TIME_STAT_CPU 100

/*
 * This is how many iterations to run before performing the printk
 * statements.  6000 = 1 minute, 360000 = 1 hour, etc.  Since
 * time_stat is long long, this Can be really high if you want :)
 */
#define TIME_COUNTER 180000

/* Undef this if you want a terse log message. */
#undef TIME_STAT_VERBOSE

#define _INLINE_ inline

/*
 * Until we get a formal timer assignment
 */
#define COMTROL_TIMER 15

#ifndef NEW_MODULES
/*
 * NB. we must include the kernel idenfication string in to install
 * the module.
 */
#include <linux/version.h>
/*static*/ char kernel_version[] = UTS_RELEASE;
#endif

static struct r_port *rp_table[MAX_RP_PORTS];
static struct tty_struct *rocket_table[MAX_RP_PORTS];
static unsigned int xmit_flags[NUM_BOARDS];
static struct termios *rocket_termios[MAX_RP_PORTS];
static struct termios *rocket_termios_locked[MAX_RP_PORTS];

static struct tty_driver rocket_driver, callout_driver;
static int rocket_refcount = 0;

static int rp_num_ports_open = 0;

unsigned long board1 = 0;
unsigned long board2 = 0;
unsigned long board3 = 0;
unsigned long board4 = 0;
unsigned long controller = 0;
static unsigned long rcktpt_io_addr[NUM_BOARDS];
static int max_board;
#ifdef TIME_STAT
static unsigned long long time_stat = 0;
static unsigned long time_stat_short = 0;
static unsigned long time_stat_long = 0;
static unsigned long time_counter = 0;
```

```
#endif

/*
 * tmp_buf is used as a temporary buffer by rp_write.  We need to
 * lock it in case the memcpy_fromfs blocks while swapping in a page,
 * and some other program tries to do a serial write at the same time.
 * Since the lock will only come under contention when the system is
 * swapping and available memory is low, it makes sense to share one
 * buffer across all the serial ports, since it significantly saves
 * memory if large numbers of serial ports are open.
 */
static unsigned char *tmp_buf = 0;
static struct semaphore tmp_buf_sem = MUTEX;

static void rp_start(struct tty_struct *tty);

static inline int rocket_paranoia_check(struct r_port *info,
                                        dev_t device, const char *routine)
{
#ifdef ROCKET_PARANOIA_CHECK
        static const char *badmagic =
                "Warning: bad magic number for rocketport struct (%d, %d) in
%s\n";
        if (!info)
                return 1;
        if (info->magic != RPORT_MAGIC) {
                printk(badmagic, MAJOR(device), MINOR(device), routine);
                return 1;
        }
#endif
        return 0;
}

/*
 * Here begins the interrupt/polling routine for the Rocketport!
 */
static _INLINE_ void rp_do_receive(struct r_port *info,
                                   struct tty_struct *tty,
                                   CHANNEL_t *cp,
                                   unsigned int ChanStatus)
{
        unsigned int CharNStat;
        int ToRecv, wRecv, space;

        ToRecv= sGetRxCnt(cp);
        space = TTY_FLIPBUF_SIZE - tty->flip.count;
#ifdef ROCKET_DEBUG_INTR
        printk("rp_do_receive(%d, %d)...", ToRecv, space);
#endif
        if (ToRecv == 0 || (space <= 0))
                return;

        /*
         * determine how many we can actually read in.  If we can't
         * read any in then we have a software overrun condition.
         */
        if (ToRecv > space)
                ToRecv = space;

        /*
```

```
         * if status indicates there are errored characters in the
         * FIFO, then enter status mode (a word in FIFO holds
         * character and status).
         */
        if (ChanStatus & (RXFOVERFL | RXBREAK | RXFRAME | RXPARITY)) {
                if (!(ChanStatus & STATMODE)) {
                        ChanStatus |= STATMODE;
                        sEnRxStatusMode(cp);
                }
        }

        /*
         * if we previously entered status mode, then read down the
         * FIFO one word at a time, pulling apart the character and
         * the status.  Update error counters depending on status
         */
        if (ChanStatus & STATMODE) {
                while (ToRecv) {
                        CharNStat= sInW(sGetTxRxDataIO(cp));

                        if (CharNStat & STMBREAK) {
                                *tty->flip.flag_buf_ptr++ = TTY_BREAK;
#if 0
                                if (info->flags & ROCKET_SAK)
                                        do_SAK(tty);
#endif
                        } else if (CharNStat & STMPARITYH)
                                *tty->flip.flag_buf_ptr++ = TTY_PARITY;
                        else if (CharNStat & STMFRAMEH)
                                *tty->flip.flag_buf_ptr++ = TTY_FRAME;
                        else if (CharNStat & STMRCVROVRH)
                                *tty->flip.flag_buf_ptr++ =TTY_OVERRUN;
                        else
                                *tty->flip.flag_buf_ptr++ = 0;
                        *tty->flip.char_buf_ptr++ = CharNStat & 0xff;
                        tty->flip.count++;
                        ToRecv--;
                }

                /*
                 * after we’ve emptied the FIFO in status mode, turn
                 * status mode back off
                 */
                if (sGetRxCnt(cp) == 0)
                        sDisRxStatusMode(cp);
        } else {
                /*
                 * we aren’t in status mode, so read down the FIFO two
                 * characters at time by doing repeated word IO
                 * transfer.
                 */
                wRecv= ToRecv >> 1;
                if (wRecv)
                        sInStrW(sGetTxRxDataIO(cp), tty->flip.char_buf_ptr,
                                wRecv);
                if (ToRecv & 1)
                        tty->flip.char_buf_ptr[ToRecv-1] =
                                sInB(sGetTxRxDataIO(cp));
                memset(tty->flip.flag_buf_ptr, 0, ToRecv);
                tty->flip.char_buf_ptr += ToRecv;
```

```
                tty->flip.flag_buf_ptr += ToRecv;
                tty->flip.count += ToRecv;
        }
        queue_task_irq_off(&tty->flip.tqueue, &tq_timer);
}

/*
 * This routine is called when a transmit interrupt is found.  It's
 * responsible for pushing data found in the transmit buffer out to
 * the serial card.
 */
static _INLINE_ void rp_do_transmit(struct r_port *info)
{
        int     c;
        CHANNEL_t *cp = &info->channel;
        struct tty_struct *tty;

#ifdef ROCKET_DEBUG_INTR
        printk("rp_do_transmit ");
#endif
        if (!info)
                return;
        if (!info->tty) {
                printk("rp: WARNING rp_do_transmit called with "
                       "info->tty==NULL\n");
                xmit_flags[info->line >> 5] &= ~(1 << (info->line & 0x1f));
                return;
        }
        tty = info->tty;
        info->xmit_fifo_room = TXFIFO_SIZE - sGetTxCnt(cp);
        while (1) {
                c = MIN(info->xmit_fifo_room,
                        MIN(info->xmit_cnt,
                            XMIT_BUF_SIZE - info->xmit_tail));
                if (c <= 0 || info->xmit_fifo_room <= 0)
                        break;
                sOutStrW(sGetTxRxDataIO(cp),
                         info->xmit_buf + info->xmit_tail, c/2);
                if (c & 1)
                        sOutB(sGetTxRxDataIO(cp),
                              info->xmit_buf[info->xmit_tail + c -
                                             1]);
                info->xmit_tail += c;
                info->xmit_tail &= XMIT_BUF_SIZE-1;
                info->xmit_cnt -= c;
                info->xmit_fifo_room -= c;
#ifdef ROCKET_DEBUG_INTR
                printk("tx %d chars...", c);
#endif
        }
        if (info->xmit_cnt == 0)
                xmit_flags[info->line >> 5] &= ~(1 << (info->line & 0x1f));
        if (info->xmit_cnt < WAKEUP_CHARS) {
                if ((tty->flags & (1 << TTY_DO_WRITE_WAKEUP)) &&
                    tty->ldisc.write_wakeup)
                        (tty->ldisc.write_wakeup)(tty);
                wake_up_interruptible(&tty->write_wait);
        }
#ifdef ROCKET_DEBUG_INTR
        printk("(%d,%d,%d,%d)...", info->xmit_cnt, info->xmit_head,
```

```
                 info->xmit_tail, info->xmit_fifo_room);
#endif
}

/*
 * This function is called for each port which has signalled an
 * interrupt.  It checks what interrupts are pending and services
 * them.
 */
static _INLINE_ void rp_handle_port(struct r_port *info)
{
        CHANNEL_t *cp;
        struct tty_struct *tty;
        unsigned int IntMask, ChanStatus;

        if (!info)
                return;
        if ( (info->flags & ROCKET_INITIALIZED) == 0 ) {
                printk("rp: WARNING: rp_handle_port called with "
                       "info->flags & NOT_INIT\n");
                return;
        }
        if (!info->tty) {
                printk("rp: WARNING: rp_handle_port called with "
                       "info->tty==NULL\n");
                return;
        }
        cp = &info->channel;
        tty = info->tty;

        IntMask = sGetChanIntID(cp) & info->intmask;
#ifdef ROCKET_DEBUG_INTR
        printk("rp_interrupt %02x...", IntMask);
#endif
        ChanStatus= sGetChanStatus(cp);
        if (IntMask & RXF_TRIG) { /* Rx FIFO trigger level */
                rp_do_receive(info, tty, cp, ChanStatus);
        }
#if 0
        if (IntMask & SRC_INT) {  /* Special receive condition */
        }
#endif
        if (IntMask & DELTA_CD) { /* CD change  */
#if (defined(ROCKET_DEBUG_OPEN) || defined(ROCKET_DEBUG_INTR))
                printk("ttyR%d CD now %s...", info->line,
                       (ChanStatus & CD_ACT) ? "on" : "off");
#endif
                if (!(ChanStatus & CD_ACT) &&
                    info->cd_status &&
                    !((info->flags & ROCKET_CALLOUT_ACTIVE) &&
                      (info->flags & ROCKET_CALLOUT_NOHUP)))
                        tty_hangup(tty);
                info->cd_status = (ChanStatus & CD_ACT) ? 1 : 0;
                wake_up_interruptible(&info->open_wait);
        }
#ifdef ROCKET_DEBUG_INTR
        if (IntMask & DELTA_CTS) {        /* CTS change */
                printk("CTS change...\n");
        }
        if (IntMask & DELTA_DSR) {        /* DSR change */
```

```
                printk("DSR change...\n");
        }
#endif
}

/*
 * The top level polling routine.
 */
static void rp_do_poll()
{
        CONTROLLER_t *ctlp;
        int ctrl, aiop, ch, line;
        unsigned int xmitmask;
        unsigned char CtlMask, AiopMask;

#ifdef TIME_STAT
        unsigned long low=0, high=0, loop_time;
        unsigned long long time_stat_tmp=0, time_stat_tmp2=0;

        __asm__(".byte 0x0f,0x31"
                :"=a" (low), "=d" (high));
        time_stat_tmp = high;
        time_stat_tmp <<= 32;
        time_stat_tmp += low;
#endif /* TIME_STAT */


        for (ctrl=0; ctrl <= max_board; ctrl++) {
                if (rcktpt_io_addr[ctrl] <= 0)
                        continue;
                ctlp= sCtlNumToCtlPtr(ctrl);

#ifdef ENABLE_PCI
                if(ctlp->BusType == isPCI)
                        CtlMask= sPCIGetControllerIntStatus(ctlp);
                else
#endif
                        CtlMask= sGetControllerIntStatus(ctlp);
                for (aiop=0; CtlMask; CtlMask >>= 1, aiop++) {
                        if (CtlMask & 1) {
                                AiopMask= sGetAiopIntStatus(ctlp, aiop);
                                for (ch=0; AiopMask; AiopMask >>= 1, ch++) {
                                        if (AiopMask & 1) {
                                                line = (ctrl << 5) |
                                                        (aiop << 3) | ch;

rp_handle_port(rp_table[line]);
                                        }
                                }
                        }
                }
                xmitmask = xmit_flags[ctrl];
                for (line = ctrl << 5; xmitmask; xmitmask >>= 1, line++) {
                        if (xmitmask & 1)
                                rp_do_transmit(rp_table[line]);
                }
        }

        /*
         * Reset the timer so we get called at the next clock tick.
```

```
	 */
	if (rp_num_ports_open) {
		timer_active |= 1 << COMTROL_TIMER;
	}
#ifdef TIME_STAT
	__asm__(".byte 0x0f,0x31"
		:"=a" (low), "=d" (high));
	time_stat_tmp2 = high;
	time_stat_tmp2 <<= 32;
	time_stat_tmp2 += low;
	time_stat_tmp2 -= time_stat_tmp;
	time_stat += time_stat_tmp2;
	if (time_counter == 0)
		time_stat_short = time_stat_long = time_stat_tmp2;
	else {
		if ( time_stat_tmp2 < time_stat_short )
			time_stat_short = time_stat_tmp2;
		else if ( time_stat_tmp2 > time_stat_long )
			time_stat_long = time_stat_tmp2;
	}
	if ( ++time_counter == TIME_COUNTER ) {
		loop_time = (unsigned long)
			( ((unsigned long)(time_stat >> 32) *
			   ( (unsigned long)(0xffffffff)/
			    (TIME_STAT_CPU * TIME_COUNTER) ) ) +
			 ((unsigned long)time_stat/
			  (TIME_STAT_CPU*TIME_COUNTER)));
#ifdef TIME_STAT_VERBOSE
		printk("rp_do_poll: Interrupt Timings\n");
		printk("     %5ld iterations; %ld us min,\n",
		       (long)TIME_COUNTER, (time_stat_short/TIME_STAT_CPU));
		printk("     %5ld us max, %ld us average per iteration.\n",
		       (time_stat_long/TIME_STAT_CPU), loop_time);
		printk("We want to use < 5,000 us for an iteration.\n");
#else /* TIME_STAT_VERBOSE */
		printk("rp: %ld loops: %ld min, %ld max, %ld us/loop.\n",
		       (long)TIME_COUNTER, (time_stat_short/TIME_STAT_CPU),
		       (time_stat_long/TIME_STAT_CPU), loop_time);
#endif /* TIME_STAT_VERBOSE */
		time_counter = time_stat = 0;
		time_stat_short = time_stat_long = 0;
	}
#endif /* TIME_STAT */
}
/*
 * Here ends the interrupt/polling routine.
 */


/*
 * This function initializes the r_port structure, as well as enabling
 * the port on the RocketPort board.
 */
static void init_r_port(int board, int aiop, int chan)
{
	struct r_port *info;
	int line;
	CONTROLLER_T *ctlp;
	CHANNEL_t	*cp;

```

```
        line = (board << 5) | (aiop << 3) | chan;

        ctlp= sCtlNumToCtlPtr(board);

        info = kmalloc(sizeof(struct r_port), GFP_KERNEL);
        if (!info) {
                printk("Couldn’t allocate info struct for line #%d\n", line);
                return;
        }
        memset(info, 0, sizeof(struct r_port));

        info->magic = RPORT_MAGIC;
        info->line = line;
        info->ctlp = ctlp;
        info->board = board;
        info->aiop = aiop;
        info->chan = chan;
        info->closing_wait = 3000;
        info->close_delay = 50;
        info->callout_termios =callout_driver.init_termios;
        info->normal_termios = rocket_driver.init_termios;

        info->intmask = RXF_TRIG | TXFIFO_MT | SRC_INT | DELTA_CD |
                DELTA_CTS | DELTA_DSR;
        if (sInitChan(ctlp, &info->channel, aiop, chan) == 0) {
                printk("Rocketport sInitChan(%d, %d, %d) failed!\n",
                       board, aiop, chan);
                kfree(info);
                return;
        }
        cp = &info->channel;
        rp_table[line] = info;
}

static int baud_table[] = {
        0, BRD50, BRD75, BRD110, BRD134, BRD150, BRD200, BRD300,
        BRD600, BRD1200, BRD1800, BRD2400, BRD4800, BRD9600, BRD19200,
        BRD38400, BRD57600, BRD115200, BRD230400, BRD460800, 0 };

/*
 * This routine configures a rocketport port so according to its
 * termio settings.
 */
static void configure_r_port(struct r_port *info)
{
        unsigned cflag;
        int     i;
        CHANNEL_t       *cp;

        if (!info->tty || !info->tty->termios)
                return;
        cp = &info->channel;
        cflag = info->tty->termios->c_cflag;
        i = cflag & CBAUD;
        if (i & CBAUDEX) {
                i &= ~CBAUDEX;
                if (i < 1 || i > 4)
                        info->tty->termios->c_cflag &= ~CBAUDEX;
                else
                        i += 15;
```

```
        }
        if (i == 15) {
                if ((info->flags & ROCKET_SPD_MASK) == ROCKET_SPD_HI)
                        i += 1;
                if ((info->flags & ROCKET_SPD_MASK) == ROCKET_SPD_VHI)
                        i += 2;
                if ((info->flags & ROCKET_SPD_MASK) == ROCKET_SPD_SHI)
                        i += 3;
                if ((info->flags & ROCKET_SPD_MASK) == ROCKET_SPD_WARP)
                        i += 4;
        }
        if (i) {
                sSetDTR(cp);
                sSetBaud(cp, baud_table[i]);
        } else {
                /*
                 * speed is 0, hang up the line
                 */
                sClrDTR(cp);
        }
        if ((cflag & CSIZE) == CS8) {
                sSetData8(cp);
        } else {
                sSetData7(cp);
        }
        if (cflag & CSTOPB) {
                sSetStop2(cp);
        } else {
                sSetStop1(cp);
        }

        if (cflag & PARENB) {
                sEnParity(cp);
                if (cflag & PARODD) {
                        sSetOddParity(cp);
                } else {
                        sSetEvenParity(cp);
                }
        } else {
                sDisParity(cp);
        }
        if (cflag & CRTSCTS) {
                info->intmask |= DELTA_CTS;
                sEnCTSFlowCtl(cp);
        } else {
                info->intmask &= ~DELTA_CTS;
                sDisCTSFlowCtl(cp);
        }
        sSetRTS(&info->channel);
        if (cflag & CLOCAL)
                info->intmask &= ~DELTA_CD;
        else
                info->intmask |= DELTA_CD;
}

static int block_til_ready(struct tty_struct *tty, struct file * filp,
                           struct r_port *info)
{
        struct wait_queue wait = { current, NULL };
        int             retval;
```

```
	int		do_clocal = 0;
	unsigned long	flags;

	/*
	 * If the device is in the middle of being closed, then block
	 * until it’s done, and then try again.
	 */
	if (tty_hung_up_p(filp))
		return ((info->flags & ROCKET_HUP_NOTIFY) ?
			-EAGAIN : -ERESTARTSYS);
	if (info->flags & ROCKET_CLOSING) {
		interruptible_sleep_on(&info->close_wait);
		return ((info->flags & ROCKET_HUP_NOTIFY) ?
			-EAGAIN : -ERESTARTSYS);
	}

	/*
	 * If this is a callout device, then just make sure the normal
	 * device isn’t being used.
	 */
	if (tty->driver.subtype == SERIAL_TYPE_CALLOUT) {
		if (info->flags & ROCKET_NORMAL_ACTIVE)
			return -EBUSY;
		if ((info->flags & ROCKET_CALLOUT_ACTIVE) &&
		    (info->flags & ROCKET_SESSION_LOCKOUT) &&
		    (info->session != current->session))
		    return -EBUSY;
		if ((info->flags & ROCKET_CALLOUT_ACTIVE) &&
		    (info->flags & ROCKET_PGRP_LOCKOUT) &&
		    (info->pgrp != current->pgrp))
		    return -EBUSY;
		info->flags |= ROCKET_CALLOUT_ACTIVE;
		return 0;
	}

	/*
	 * If non-blocking mode is set, or the port is not enabled,
	 * then make the check up front and then exit.
	 */
	if ((filp->f_flags & O_NONBLOCK) ||
	    (tty->flags & (1 << TTY_IO_ERROR))) {
		if (info->flags & ROCKET_CALLOUT_ACTIVE)
			return -EBUSY;
		info->flags |= ROCKET_NORMAL_ACTIVE;
		return 0;
	}

	if (info->flags & ROCKET_CALLOUT_ACTIVE) {
		if (info->normal_termios.c_cflag & CLOCAL)
			do_clocal = 1;
	} else {
		if (tty->termios->c_cflag & CLOCAL)
			do_clocal = 1;
	}

	/*
	 * Block waiting for the carrier detect and the line to become
	 * free (i.e., not in use by the callout).  While we are in
	 * this loop, info->count is dropped by one, so that
	 * rp_close() knows when to free things.  We restore it upon
```

```
	 * exit, either normal or abnormal.
	 */
	retval = 0;
	add_wait_queue(&info->open_wait, &wait);
#ifdef ROCKET_DEBUG_OPEN
	printk("block_til_ready before block: ttyR%d, count = %d\n",
	       info->line, info->count);
#endif
	save_flags(flags); cli();
	if (!tty_hung_up_p(filp))
		info->count--;
	restore_flags(flags);
	info->blocked_open++;
	while (1) {
		if (!(info->flags & ROCKET_CALLOUT_ACTIVE)) {
			sSetDTR(&info->channel);
			sSetRTS(&info->channel);
		}
		current->state = TASK_INTERRUPTIBLE;
		if (tty_hung_up_p(filp) ||
		    !(info->flags & ROCKET_INITIALIZED)) {
			if (info->flags & ROCKET_HUP_NOTIFY)
				retval = -EAGAIN;
			else
				retval = -ERESTARTSYS;
			break;
		}
		if (!(info->flags & ROCKET_CALLOUT_ACTIVE) &&
		    !(info->flags & ROCKET_CLOSING) &&
		    (do_clocal || (sGetChanStatusLo(&info->channel) &
				   CD_ACT)))
			break;
		if (current->signal & ~current->blocked) {
			retval = -ERESTARTSYS;
			break;
		}
#ifdef ROCKET_DEBUG_OPEN
		printk("block_til_ready blocking: ttyR%d, count = %d\n",
		       info->line, info->count);
#endif
		schedule();
	}
	current->state = TASK_RUNNING;
	remove_wait_queue(&info->open_wait, &wait);
	cli();
	if (!tty_hung_up_p(filp))
		info->count++;
	restore_flags(flags);
	info->blocked_open--;
#ifdef ROCKET_DEBUG_OPEN
	printk("block_til_ready after blocking: ttyR%d, count = %d\n",
	       info->line, info->count);
#endif
	if (retval)
		return retval;
	info->flags |= ROCKET_NORMAL_ACTIVE;
	return 0;
}

/*
```

```
 * This routine is called whenever a rocketport board is opened.
 */
static int rp_open(struct tty_struct *tty, struct file * filp)
{
        struct r_port *info;
        int     line, retval;
        CHANNEL_t       *cp;
        unsigned long page;

        line = MINOR(tty->device) - tty->driver.minor_start;
        if ((line < 0) || (line >= MAX_RP_PORTS))
                return -ENODEV;
        if (!tmp_buf) {
                page = get_free_page(GFP_KERNEL);
                if (!page)
                        return -ENOMEM;
                if (tmp_buf)
                        free_page(page);
                else
                        tmp_buf = (unsigned char *) page;
        }
        page = get_free_page(GFP_KERNEL);
        if (!page)
                return -ENOMEM;
        /*
         * We must not sleep from here until the port is marked fully
         * in use./
         */
        if (rp_table[line] == NULL) {
                tty->flags = (1 << TTY_IO_ERROR);
                free_page(page);
                return 0;
        }
        tty->driver_data = info = rp_table[line];
        if (!info) {
                printk("rp_open: rp_table[%d] is NULL!\n", line);
                free_page(page);
                return -EIO;
        }
        if (info->xmit_buf)
                free_page(page);
        else
                info->xmit_buf = (unsigned char *) page;
        info->tty = tty;

        if (info->count++ == 0) {
                MOD_INC_USE_COUNT;
                rp_num_ports_open++;
#ifdef ROCKET_DEBUG_OPEN
                printk("rocket mod++ = %d...", rp_num_ports_open);
#endif
        }
#ifdef ROCKET_DEBUG_OPEN
        printk("rp_open ttyR%d, count=%d\n", info->line, info->count);
#endif
        /*
         * Info->count is now 1; so it’s safe to sleep now.
         */
        info->session = current->session;
        info->pgrp = current->pgrp;
```

```
        cp = &info->channel;
        sSetRxTrigger(cp, TRIG_1);
        if (sGetChanStatus(cp) & CD_ACT)
                info->cd_status = 1;
        else
                info->cd_status = 0;
        sDisRxStatusMode(cp);
        sFlushRxFIFO(cp);
        sFlushTxFIFO(cp);

        sEnInterrupts(cp, (TXINT_EN|MCINT_EN|RXINT_EN|SRCINT_EN|CHANINT_EN));
        sSetRxTrigger(cp, TRIG_1);

        sGetChanStatus(cp);
        sDisRxStatusMode(cp);
        sClrTxXOFF(cp);

        sDisCTSFlowCtl(cp);
        sDisTxSoftFlowCtl(cp);

        sEnRxFIFO(cp);
        sEnTransmit(cp);

        info->flags |= ROCKET_INITIALIZED;

        configure_r_port(info);
        timer_active |= 1 << COMTROL_TIMER;

        retval = block_til_ready(tty, filp, info);
        if (retval) {
#ifdef ROCKET_DEBUG_OPEN
                printk("rp_open returning after block_til_ready with %d\n",
                       retval);
#endif
                return retval;
        }

        if ((info->count == 1) && (info->flags & ROCKET_SPLIT_TERMIOS)) {
                if (tty->driver.subtype == SERIAL_TYPE_NORMAL)
                        *tty->termios = info->normal_termios;
                else
                        *tty->termios = info->callout_termios;
                configure_r_port(info);
        }

        return 0;
}

static void rp_close(struct tty_struct *tty, struct file * filp)
{
        struct r_port * info = (struct r_port *)tty->driver_data;
        unsigned long flags, timeout;
        CHANNEL_t       *cp;

        if (rocket_paranoia_check(info, tty->device, "rp_close"))
                return;

#ifdef ROCKET_DEBUG_OPEN
        printk("rp_close ttyR%d, count = %d\n", info->line, info->count);
```

```
#endif

        save_flags(flags); cli();

        if (tty_hung_up_p(filp)) {
                restore_flags(flags);
                return;
        }
        if ((tty->count == 1) && (info->count != 1)) {
                /*
                 * Uh, oh.  tty->count is 1, which means that the tty
                 * structure will be freed.  Info->count should always
                 * be one in these conditions.  If it's greater than
                 * one, we've got real problems, since it means the
                 * serial port won't be shutdown.
                 */
                printk("rp_close: bad serial port count; tty->count is 1, "
                       "info->count is %d\n", info->count);
                info->count = 1;
        }
        if (--info->count < 0) {
                printk("rp_close: bad serial port count for ttyR%d: %d\n",
                       info->line, info->count);
                info->count = 0;
        }
        if (info->count) {
                restore_flags(flags);
                return;
        }
        info->flags |= ROCKET_CLOSING;
        /*
         * Save the termios structure, since this port may have
         * separate termios for callout and dialin.
         */
        if (info->flags & ROCKET_NORMAL_ACTIVE)
                info->normal_termios = *tty->termios;
        if (info->flags & ROCKET_CALLOUT_ACTIVE)
                info->callout_termios = *tty->termios;

        cp = &info->channel;

        /*
         * Now we wait for the transmit buffer to clear; and we notify
         * the line discipline to only process XON/XOFF characters.
         */
        tty->closing = 1;
        if (info->closing_wait != ROCKET_CLOSING_WAIT_NONE)
                tty_wait_until_sent(tty, info->closing_wait);
        /*
         * Before we drop DTR, make sure the UART transmitter
         * has completely drained; this is especially
         * important if there is a transmit FIFO!
         */
        timeout = jiffies+2*HZ;
        while (sGetTxCnt(cp)) {
                current->state = TASK_INTERRUPTIBLE;
                current->timeout = jiffies + HZ/5;
                schedule();
                if (jiffies > timeout)
                        break;
```

```
        }

        xmit_flags[info->line >> 5] &= ~(1 << (info->line & 0x1f));
        sDisTransmit(cp);
        sDisInterrupts(cp,
(TXINT_EN|MCINT_EN|RXINT_EN|SRCINT_EN|CHANINT_EN));
        sDisCTSFlowCtl(cp);
        sDisTxSoftFlowCtl(cp);
        sClrTxXOFF(cp);
        sFlushRxFIFO(cp);
        sFlushTxFIFO(cp);
        sClrRTS(cp);
        if (C_HUPCL(tty)) {
                sClrDTR(cp);
        }
        if (tty->driver.flush_buffer)
                tty->driver.flush_buffer(tty);
        if (tty->ldisc.flush_buffer)
                tty->ldisc.flush_buffer(tty);

        xmit_flags[info->line >> 5] &= ~(1 << (info->line & 0x1f));
        if (info->blocked_open) {
                if (info->close_delay) {
                        current->state = TASK_INTERRUPTIBLE;
                        current->timeout = jiffies + info->close_delay;
                        schedule();
                }
                wake_up_interruptible(&info->open_wait);
        } else {
                if (info->xmit_buf) {
                        free_page((unsigned long) info->xmit_buf);
                        info->xmit_buf = 0;
                }
        }
        info->flags &= ~(ROCKET_INITIALIZED | ROCKET_CLOSING |
                         ROCKET_CALLOUT_ACTIVE | ROCKET_NORMAL_ACTIVE);
        tty->closing = 0;
        wake_up_interruptible(&info->close_wait);

        MOD_DEC_USE_COUNT;
        rp_num_ports_open--;
#ifdef ROCKET_DEBUG_OPEN
        printk("rocket mod-- = %d...", rp_num_ports_open);
#endif
        restore_flags(flags);

#ifdef ROCKET_DEBUG_OPEN
        printk("rp_close ttyR%d complete shutdown\n", info->line);
#endif

}

static void rp_set_termios(struct tty_struct *tty,
                           struct termios *old_termios)
{
        struct r_port * info = (struct r_port *)tty->driver_data;

        if (rocket_paranoia_check(info, tty->device, "rp_set_termios"))
                return;

```

```
        if (tty->termios->c_cflag == old_termios->c_cflag)
                return;

        configure_r_port(info);

        if ((old_termios->c_cflag & CRTSCTS) &&
            !(tty->termios->c_cflag & CRTSCTS)) {
                tty->hw_stopped = 0;
                rp_start(tty);
        }
}

/*
 * Here are the routines used by rp_ioctl
 */

static void send_break(      struct r_port * info, int duration)
{
        current->state = TASK_INTERRUPTIBLE;
        current->timeout = jiffies + duration;
        cli();
        sSendBreak(&info->channel);
        schedule();
        sClrBreak(&info->channel);
        sti();
}

static int get_modem_info(struct r_port * info, unsigned int *value)
{
        unsigned int control, result, ChanStatus;

        ChanStatus = sGetChanStatusLo(&info->channel);

        control = info->channel.TxControl[3];
        result =  ((control & SET_RTS) ? TIOCM_RTS : 0)
                | ((control & SET_DTR) ? TIOCM_DTR : 0)
                | ((ChanStatus  & CD_ACT) ? TIOCM_CAR : 0)
                        /* TIOCM_RNG not supported */
                | ((ChanStatus  & DSR_ACT) ? TIOCM_DSR : 0)
                | ((ChanStatus  & CTS_ACT) ? TIOCM_CTS : 0);
        put_fs_long(result,(unsigned long *) value);
        return 0;
}

static int set_modem_info(struct r_port * info, unsigned int cmd,
                          unsigned int *value)
{
        int error;
        unsigned int arg;

        error = verify_area(VERIFY_READ, value, sizeof(int));
        if (error)
                return error;
        arg = get_fs_long((unsigned long *) value);
        switch (cmd) {
        case TIOCMBIS:
                if (arg & TIOCM_RTS)
                        info->channel.TxControl[3] |= SET_RTS;
                if (arg & TIOCM_DTR)
                        info->channel.TxControl[3] |= SET_DTR;
```

```
                break;
        case TIOCMBIC:
                if (arg & TIOCM_RTS)
                        info->channel.TxControl[3] &= ~SET_RTS;
                if (arg & TIOCM_DTR)
                        info->channel.TxControl[3] &= ~SET_DTR;
                break;
        case TIOCMSET:
                info->channel.TxControl[3] =
                        ((info->channel.TxControl[3] & ~(SET_RTS | SET_DTR))
                         | ((arg & TIOCM_RTS) ? SET_RTS : 0)
                         | ((arg & TIOCM_DTR) ? SET_DTR : 0));
                break;
        default:
                return -EINVAL;
        }

        sOutDW(info->channel.IndexAddr,
               *(DWord_t *) &(info->channel.TxControl[0]));

        return 0;
}

static int get_config(struct r_port * info,
                      struct rocket_config * retinfo)
{
        struct rocket_config tmp;

        if (!retinfo)
                return -EFAULT;
        memset(&tmp, 0, sizeof(tmp));
        tmp.line = info->line;
        tmp.flags = info->flags;
        tmp.close_delay = info->close_delay;
        tmp.closing_wait = info->closing_wait;
        tmp.port = rcktpt_io_addr[(info->line >> 5) & 3];
        memcpy_tofs(retinfo,&tmp,sizeof(*retinfo));
        return 0;
}

static int set_config(struct r_port * info,
                      struct rocket_config * new_info)
{
        struct rocket_config new_serial;

        if (!new_info)
                return -EFAULT;
        memcpy_fromfs(&new_serial,new_info,sizeof(new_serial));
        if (!suser()) {
                if ((new_serial.flags & ~ROCKET_USR_MASK) !=
                    (info->flags & ~ROCKET_USR_MASK))
                        return -EPERM;
                info->flags = ((info->flags & ~ROCKET_USR_MASK) |
                               (new_serial.flags & ROCKET_USR_MASK));
                configure_r_port(info);
                return 0;
        }

        info->flags = ((info->flags & ~ROCKET_FLAGS) |
                        (new_serial.flags & ROCKET_FLAGS));
```

```
        info->close_delay = new_serial.close_delay;
        info->closing_wait = new_serial.closing_wait;
        configure_r_port(info);

        return 0;
}

static int get_ports(struct r_port * info,
                     struct rocket_ports * retports)
{
        struct rocket_ports tmp;
        int     board, port, index;

        if (!retports)
                return -EFAULT;
        memset(&tmp, 0, sizeof(tmp));
        tmp.tty_major = rocket_driver.major;
        tmp.callout_major = callout_driver.major;
        for (board = 0; board < 4; board++) {
                index = board << 5;
                for (port = 0; port < 32; port++, index++) {
                        if (rp_table[index])
                                tmp.port_bitmap[board] |= 1 << port;
                }
        }
        memcpy_tofs(retports,&tmp,sizeof(*retports));
        return 0;
}

static int rp_ioctl(struct tty_struct *tty, struct file * file,
                    unsigned int cmd, unsigned long arg)
{
        int error;
        struct r_port * info = (struct r_port *)tty->driver_data;
        int retval;

        if (cmd != RCKP_GET_PORTS &&
            rocket_paranoia_check(info, tty->device, "rp_ioctl"))
                return -ENODEV;

        switch (cmd) {
                case TCSBRK:    /* SVID version: non-zero arg --> no break */
                        retval = tty_check_change(tty);
                        if (retval)
                                return retval;
                        tty_wait_until_sent(tty, 0);
                        if (!arg)
                                send_break(info, HZ/4);   /* 1/4 second */
                        return 0;
                case TCSBRKP:   /* support for POSIX tcsendbreak() */
                        retval = tty_check_change(tty);
                        if (retval)
                                return retval;
                        tty_wait_until_sent(tty, 0);
                        send_break(info, arg ? arg*(HZ/10) : HZ/4);
                        return 0;
                case TIOCGSOFTCAR:
                        error = verify_area(VERIFY_WRITE, (void *)
arg,sizeof(long));
                        if (error)
```

```
                                return error;
                        put_fs_long(C_CLOCAL(tty) ? 1 : 0,
                                    (unsigned long *) arg);
                        return 0;
                case TIOCSSOFTCAR:
                        arg = get_fs_long((unsigned long *) arg);
                        tty->termios->c_cflag =
                                ((tty->termios->c_cflag & ~CLOCAL) |
                                 (arg ? CLOCAL : 0));
                        return 0;
                case TIOCMGET:
                        error = verify_area(VERIFY_WRITE, (void *) arg,
                                sizeof(unsigned int));
                        if (error)
                                return error;
                        return get_modem_info(info, (unsigned int *) arg);
                case TIOCMBIS:
                case TIOCMBIC:
                case TIOCMSET:
                        return set_modem_info(info, cmd, (unsigned int *)
arg);
                case RCKP_GET_STRUCT:
                        error = verify_area(VERIFY_WRITE, (void *) arg,
                                                sizeof(struct r_port));
                        if (error)
                                return error;
                        memcpy_tofs((struct r_port *) arg,
                                    info, sizeof(struct r_port));
                        return 0;

                case RCKP_GET_CONFIG:
                        error = verify_area(VERIFY_WRITE, (void *) arg,
                                                sizeof(struct
rocket_config));
                        if (error)
                                return error;
                        return get_config(info, (struct rocket_config *)
arg);
                case RCKP_SET_CONFIG:
                        return set_config(info, (struct rocket_config *)
arg);

                case RCKP_GET_PORTS:
                        error = verify_area(VERIFY_WRITE, (void *) arg,
                                                sizeof(struct rocket_ports));
                        if (error)
                                return error;
                        return get_ports(info, (struct rocket_ports *) arg);
                default:
                        return -ENOIOCTLCMD;
                }
        return 0;
}

#if (defined(ROCKET_DEBUG_FLOW) || defined(ROCKET_DEBUG_THROTTLE))
static char *rp_tty_name(struct tty_struct *tty, char *buf)
{
        if (tty)
                sprintf(buf, "%s%d", tty->driver.name,
                        MINOR(tty->device) - tty->driver.minor_start +
```

```
                                tty->driver.name_base);
        else
                strcpy(buf, "NULL tty");
        return buf;
}
#endif

static void rp_throttle(struct tty_struct * tty)
{
        struct r_port *info = (struct r_port *)tty->driver_data;
        CHANNEL_t *cp;
#ifdef ROCKET_DEBUG_THROTTLE
        char    buf[64];

        printk("throttle %s: %d....\n", rp_tty_name(tty, buf),
               tty->ldisc.chars_in_buffer(tty));
#endif

        if (rocket_paranoia_check(info, tty->device, "rp_throttle"))
                return;

        cp = &info->channel;
        if (I_IXOFF(tty)) {
                if (sGetTxCnt(cp))
                        sWriteTxPrioByte(cp, STOP_CHAR(tty));
                else
                        sWriteTxByte(sGetTxRxDataIO(cp), STOP_CHAR(tty));
        }

        sClrRTS(&info->channel);
}

static void rp_unthrottle(struct tty_struct * tty)
{
        struct r_port *info = (struct r_port *)tty->driver_data;
        CHANNEL_t *cp;
#ifdef ROCKET_DEBUG_THROTTLE
        char    buf[64];

        printk("unthrottle %s: %d....\n", rp_tty_name(tty, buf),
               tty->ldisc.chars_in_buffer(tty));
#endif

        if (rocket_paranoia_check(info, tty->device, "rp_throttle"))
                return;

        cp = &info->channel;
        if (I_IXOFF(tty)) {
                if (sGetTxCnt(cp))
                        sWriteTxPrioByte(cp, START_CHAR(tty));
                else
                        sWriteTxByte(sGetTxRxDataIO(cp), START_CHAR(tty));
        }
        sSetRTS(&info->channel);
}

/*
 * ------------------------------------------------------------
 * rp_stop() and rp_start()
 *
```

```
 * This routines are called before setting or resetting tty->stopped.
 * They enable or disable transmitter interrupts, as necessary.
 * ------------------------------------------------------------
 */
static void rp_stop(struct tty_struct *tty)
{
        struct r_port * info = (struct r_port *)tty->driver_data;
#ifdef ROCKET_DEBUG_FLOW
        char    buf[64];

        printk("stop %s: %d %d....\n", rp_tty_name(tty, buf),
               info->xmit_cnt, info->xmit_fifo_room);
#endif

        if (rocket_paranoia_check(info, tty->device, "rp_stop"))
                return;

        sDisTransmit(&info->channel);
}

static void rp_start(struct tty_struct *tty)
{
        struct r_port * info = (struct r_port *)tty->driver_data;
#ifdef ROCKET_DEBUG_FLOW
        char    buf[64];

        printk("start %s: %d %d....\n", rp_tty_name(tty, buf),
               info->xmit_cnt, info->xmit_fifo_room);
#endif

        if (rocket_paranoia_check(info, tty->device, "rp_stop"))
                return;

        sEnTransmit(&info->channel);
}

/*
 * rp_hangup() --- called by tty_hangup() when a hangup is signaled.
 */
static void rp_hangup(struct tty_struct *tty)
{
        CHANNEL_t       *cp;
        struct r_port * info = (struct r_port *)tty->driver_data;

        if (rocket_paranoia_check(info, tty->device, "rp_hangup"))
                return;

#ifdef ROCKET_DEBUG_OPEN
        printk("rp_hangup of ttyR%d...", info->line);
#endif
        /*
         * If the port is in the process of being closed, just force
         * the transmit buffer to be empty, and let rp_close handle
         * the clean up.
         */
        if (info->flags & ROCKET_CLOSING) {
                cli();
                info->xmit_cnt = info->xmit_head = info->xmit_tail = 0;
                sti();
                wake_up_interruptible(&tty->write_wait);
```

```
			return;
		}
		if (info->count) {
			MOD_DEC_USE_COUNT;
			rp_num_ports_open--;
		}

		xmit_flags[info->line >> 5] &= ~(1 << (info->line & 0x1f));
		info->count = 0;
		info->flags &= ~(ROCKET_NORMAL_ACTIVE|ROCKET_CALLOUT_ACTIVE);
		info->tty = 0;

		cp = &info->channel;
		sDisRxFIFO(cp);
		sDisTransmit(cp);
		sDisInterrupts(cp,
(TXINT_EN|MCINT_EN|RXINT_EN|SRCINT_EN|CHANINT_EN));
		sDisCTSFlowCtl(cp);
		sDisTxSoftFlowCtl(cp);
		sClrTxXOFF(cp);
		info->flags &= ~ROCKET_INITIALIZED;

		wake_up_interruptible(&info->open_wait);
}

/*
 * The Rocketport write routines.  The Rocketport driver uses a
 * double-buffering strategy, with the twist that if the in-memory CPU
 * buffer is empty, and there's space in the transmit FIFO, the
 * writing routines will write directly to transmit FIFO.
 *
 * This gets a little tricky, but I'm pretty sure I got it all right.
 */
static void rp_put_char(struct tty_struct *tty, unsigned char ch)
{
		struct r_port * info = (struct r_port *)tty->driver_data;
		CHANNEL_t	*cp;

		if (rocket_paranoia_check(info, tty->device, "rp_put_char"))
			return;

#ifdef ROCKET_DEBUG_WRITE
		printk("rp_put_char %c...", ch);
#endif

		cp = &info->channel;

		if (!tty->stopped && !tty->hw_stopped && info->xmit_fifo_room == 0)
			info->xmit_fifo_room = TXFIFO_SIZE - sGetTxCnt(cp);

		if (tty->stopped || tty->hw_stopped ||
		    info->xmit_fifo_room == 0 || info->xmit_cnt != 0) {
			info->xmit_buf[info->xmit_head++] = ch;
			info->xmit_head &= XMIT_BUF_SIZE-1;
			info->xmit_cnt++;
			xmit_flags[info->line >> 5] |= (1 << (info->line & 0x1f));
		} else {
			sOutB(sGetTxRxDataIO(cp), ch);
			info->xmit_fifo_room--;
		}
```

```
}

#if (LINUX_VERSION_CODE > 66304)
static int rp_write(struct tty_struct * tty, int from_user,
                    const unsigned char *buf, int count)
#else
static int rp_write(struct tty_struct * tty, int from_user,
                    unsigned char *buf, int count)
#endif
{
        struct r_port * info = (struct r_port *)tty->driver_data;
        CHANNEL_t       *cp;
        const unsigned char     *b;
        int             error, c, written;
        unsigned long   flags;

        if (count <= 0 || rocket_paranoia_check(info, tty->device,
"rp_write"))
                return 0;

#ifdef ROCKET_DEBUG_WRITE
        printk("rp_write %d chars...", count);
#endif
        if (from_user) {
                error = verify_area(VERIFY_READ, buf, count);
                if (error)
                        return error;
        }
        cp = &info->channel;
        written = 0;

        if (!tty->stopped && !tty->hw_stopped && info->xmit_fifo_room == 0)
                info->xmit_fifo_room = TXFIFO_SIZE - sGetTxCnt(cp);

        if (!tty->stopped && !tty->hw_stopped && info->xmit_cnt == 0
            && info->xmit_fifo_room >= 0) {
                c = MIN(count, info->xmit_fifo_room);
                b = buf;
                if (from_user) {
                        down(&tmp_buf_sem);
                        memcpy_fromfs(tmp_buf, buf, c);
                        b = tmp_buf;
                        up(&tmp_buf_sem);
                        /* In case we got pre-empted */
                        if (info->tty == 0)
                                goto end;
                        c = MIN(c, info->xmit_fifo_room);
                }
                sOutStrW(sGetTxRxDataIO(cp), b, c/2);
                if (c & 1)
                        sOutB(sGetTxRxDataIO(cp), b[c-1]);
                written += c;
                buf += c;
                count -= c;
                info->xmit_fifo_room -= c;
        }
        if (!count)
                goto end;

        save_flags(flags);
```

```
	while (1) {
		cli();
		if (info->tty == 0) {
			restore_flags(flags);
			goto end;
		}
		c = MIN(count, MIN(XMIT_BUF_SIZE - info->xmit_cnt - 1,
				   XMIT_BUF_SIZE - info->xmit_head));
		if (c <= 0)
			break;

		b = buf;
		if (from_user) {
			down(&tmp_buf_sem);
			memcpy_fromfs(tmp_buf, buf, c);
			b = tmp_buf;
			up(&tmp_buf_sem);
			/* In case we got pre-empted */
			if (info->tty == 0) {
				restore_flags(flags);
				goto end;
			}
			c = MIN(c, MIN(XMIT_BUF_SIZE - info->xmit_cnt - 1,
				       XMIT_BUF_SIZE - info->xmit_head));

		}
		memcpy(info->xmit_buf + info->xmit_head, b, c);
		info->xmit_head = (info->xmit_head + c) & (XMIT_BUF_SIZE-1);
		info->xmit_cnt += c;
		restore_flags(flags);
		buf += c;
		count -= c;
		written += c;
	}
	xmit_flags[info->line >> 5] |= (1 << (info->line & 0x1f));
	restore_flags(flags);
end:
	if (info->xmit_cnt < WAKEUP_CHARS) {
		if ((tty->flags & (1 << TTY_DO_WRITE_WAKEUP)) &&
		    tty->ldisc.write_wakeup)
			(tty->ldisc.write_wakeup)(tty);
		wake_up_interruptible(&tty->write_wait);
	}
	return written;
}

/*
 * Return the number of characters that can be sent.  We estimate
 * only using the in-memory transmit buffer only, and ignore the
 * potential space in the transmit FIFO.
 */
static int rp_write_room(struct tty_struct *tty)
{
	struct r_port * info = (struct r_port *)tty->driver_data;
	int	ret;

	if (rocket_paranoia_check(info, tty->device, "rp_write_room"))
		return 0;

	ret = XMIT_BUF_SIZE - info->xmit_cnt - 1;
```

```
        if (ret < 0)
                ret = 0;
#ifdef ROCKET_DEBUG_WRITE
        printk("rp_write_room returns %d...", ret);
#endif
        return ret;
}

/*
 * Return the number of characters in the buffer.  Again, this only
 * counts those characters in the in-memory transmit buffer.
 */
static int rp_chars_in_buffer(struct tty_struct *tty)
{
        struct r_port * info = (struct r_port *)tty->driver_data;
        CHANNEL_t       *cp;

        if (rocket_paranoia_check(info, tty->device, "rp_chars_in_buffer"))
                return 0;

        cp = &info->channel;

#ifdef ROCKET_DEBUG_WRITE
        printk("rp_chars_in_buffer returns %d...", info->xmit_cnt);
#endif
        return info->xmit_cnt;
}

static void rp_flush_buffer(struct tty_struct *tty)
{
        struct r_port * info = (struct r_port *)tty->driver_data;
        CHANNEL_t       *cp;

        if (rocket_paranoia_check(info, tty->device, "rp_flush_buffer"))
                return;

        cli();
        info->xmit_cnt = info->xmit_head = info->xmit_tail = 0;
        sti();
        wake_up_interruptible(&tty->write_wait);
        if ((tty->flags & (1 << TTY_DO_WRITE_WAKEUP)) &&
            tty->ldisc.write_wakeup)
                (tty->ldisc.write_wakeup)(tty);

        cp = &info->channel;

        sFlushTxFIFO(cp);
}

#ifdef ENABLE_PCI
int register_PCI(int i, char bus, char device_fn)
{
        int     num_aiops, aiop, max_num_aiops, num_chan, chan;
        unsigned int    aiopio[MAX_AIOPS_PER_BOARD];
        CONTROLLER_t    *ctlp;
        unsigned short  vendor_id, device_id;
        int     ret, error;
        unsigned int port;

        printk("Bus: %d, device_fn: %d at index %d\n", bus, device_fn, i);
```

```
	error = pcibios_read_config_word(bus, device_fn, PCI_VENDOR_ID,
		&vendor_id);
	printk("vendor_id is %x\n", vendor_id);
	ret = pcibios_read_config_word(bus, device_fn, PCI_DEVICE_ID,
		&device_id);
	printk("device_id is %x\n", device_id);
	if (error == 0)
		error = ret;
	ret = pcibios_read_config_dword(bus, device_fn, PCI_BASE_ADDRESS_0,
		&port);
	rcktpt_io_addr[i] = (unsigned long) port;
	printk("Base I/O is %lx\n", rcktpt_io_addr[i]);
	if (error == 0)
		error = ret;

	if (error) {
		printk("PCI RocketPort error: %s not initializing due to
error"
		       "reading configuration space\n",
		       pcibios_strerror(error));
		return(0);
	}

	--rcktpt_io_addr[i];
	switch(device_id) {
	case PCI_DEVICE_ID_RP4QUAD:
	case PCI_DEVICE_ID_RP8OCTA:
	case PCI_DEVICE_ID_RP8INTF:
	case PCI_DEVICE_ID_RP8J:
		max_num_aiops = 1;
		break;
	case PCI_DEVICE_ID_RP16INTF:
		max_num_aiops = 2;
		break;
	case PCI_DEVICE_ID_RP32INTF:
		max_num_aiops = 4;
		break;
	default:
		max_num_aiops = 0;
		break;
	}
	for(aiop=0;aiop < max_num_aiops;aiop++)
		aiopio[aiop] = rcktpt_io_addr[i] + (aiop * 0x40);
	ctlp = sCtlNumToCtlPtr(i);
	num_aiops = sPCIInitController(ctlp, i,
					aiopio, max_num_aiops, 0,
					FREQ_DIS, 0);
	if(num_aiops <= 0) {
		rcktpt_io_addr[i] = 0;
		return(0);
	}
	for(aiop = 0;aiop < num_aiops; aiop++) {
		sResetAiopByNum(ctlp, aiop);
		sEnAiop(ctlp, aiop);
		num_chan = sGetAiopNumChan(ctlp, aiop);
		for(chan=0;chan < num_chan; chan++)
			init_r_port(i, aiop, chan);
	}
	return(1);
}
```

```
static int init_PCI(int boards_found)
{
        unsigned char   bus, device_fn;
        int     i, count = 0;

        for(i=0; i < (NUM_BOARDS - boards_found); i++) {
                if(!pcibios_find_device(PCI_VENDOR_ID_RP,
PCI_DEVICE_ID_RP8OCTA,
                        i, &bus, &device_fn))
                        if(register_PCI(count+boards_found, bus, device_fn))
                                count++;
                if(!pcibios_find_device(PCI_VENDOR_ID_RP,
PCI_DEVICE_ID_RP8INTF,
                        i, &bus, &device_fn))
                        if(register_PCI(count+boards_found, bus, device_fn))
                                count++;
                if(!pcibios_find_device(PCI_VENDOR_ID_RP,
PCI_DEVICE_ID_RP16INTF,
                        i, &bus, &device_fn))
                        if(register_PCI(count+boards_found, bus, device_fn))
                                count++;
                if(!pcibios_find_device(PCI_VENDOR_ID_RP,
PCI_DEVICE_ID_RP32INTF,
                        i, &bus, &device_fn))
                        if(register_PCI(count+boards_found, bus, device_fn))
                                count++;
        }
        return(count);
}
#endif

static int init_ISA(int i, int *reserved_controller)
{
        int     num_aiops, num_chan;
        int     aiop, chan;
        unsigned int    aiopio[MAX_AIOPS_PER_BOARD];
        CONTROLLER_t    *ctlp;

        if (rcktpt_io_addr[i] == 0)
                return(0);

        if (check_region(rcktpt_io_addr[i],64)) {
                printk("RocketPort board address 0x%lx in use...\n",
                        rcktpt_io_addr[i]);
                rcktpt_io_addr[i] = 0;
                return(0);
        }

        for (aiop=0; aiop<MAX_AIOPS_PER_BOARD; aiop++)
                aiopio[aiop]= rcktpt_io_addr[i] + (aiop * 0x400);
        ctlp= sCtlNumToCtlPtr(i);
        num_aiops = sInitController(ctlp, i, controller + (i*0x400),
                                    aiopio, MAX_AIOPS_PER_BOARD, 0,
                                    FREQ_DIS, 0);
        if (num_aiops <= 0) {
                rcktpt_io_addr[i] = 0;
                return(0);
        }
        for (aiop = 0; aiop < num_aiops; aiop++) {
```

```
                sResetAiopByNum(ctlp, aiop);
                sEnAiop(ctlp, aiop);
                num_chan = sGetAiopNumChan(ctlp,aiop);
                for (chan=0; chan < num_chan; chan++)
                        init_r_port(i, aiop, chan);
        }
        printk("Rocketport controller #%d found at 0x%lx, "
               "%d AIOPs\n", i, rcktpt_io_addr[i],
               num_aiops);
        if (rcktpt_io_addr[i] + 0x40 == controller) {
                *reserved_controller = 1;
                request_region(rcktpt_io_addr[i], 68,
                                       "Comtrol Rocketport");
        } else {
                request_region(rcktpt_io_addr[i], 64,
                               "Comtrol Rocketport");
        }
        return(1);
}


/*
 * The module "startup" routine; it's run when the module is loaded.
 */
int
init_module( void) {
        int i, retval, pci_boards_found, isa_boards_found;
        int     reserved_controller = 0;

        printk("Rocketport device driver module, version %s, %s\n",
               ROCKET_VERSION, ROCKET_DATE);

        /*
         * Set up the timer channel.  If it is already in use by
         * some other driver, give up.
         */
        if (timer_table[COMTROL_TIMER].fn) {
                printk("rocket.o: Timer channel %d already in use!\n",
                       COMTROL_TIMER);
                return -EBUSY;
        }
        timer_table[COMTROL_TIMER].fn = rp_do_poll;
        timer_table[COMTROL_TIMER].expires = 0;

        /*
         * Initialize the array of pointers to our own internal state
         * structures.
         */
        memset(rp_table, 0, sizeof(rp_table));
        memset(xmit_flags, 0, sizeof(xmit_flags));

        if (board1 == 0)
                board1 = 0x180;
        if (controller == 0)
                controller = board1 + 0x40;

        if (check_region(controller, 4)) {
                printk("Controller IO addresses in use, unloading
driver.\n");
                return -EBUSY;
```

```
	}

	rcktpt_io_addr[0] = board1;
	rcktpt_io_addr[1] = board2;
	rcktpt_io_addr[2] = board3;
	rcktpt_io_addr[3] = board4;

	/*
	 * OK, let’s probe each of the controllers looking for boards.
	 */
	isa_boards_found = 0;
	pci_boards_found = 0;
	for (i=0; i < NUM_BOARDS; i++) {
		if(init_ISA(i, &reserved_controller))
			isa_boards_found++;
	}
#ifdef ENABLE_PCI
	if (pcibios_present()) {
		if(isa_boards_found < NUM_BOARDS)
			pci_boards_found += init_PCI(isa_boards_found);
		printk("%d PCI boards found\n", pci_boards_found);
	} else {
		printk("No PCI BIOS found\n");
	}
#endif
	max_board = pci_boards_found + isa_boards_found;

	if (max_board == 0) {
		printk("No rocketport ports found; unloading driver.\n");
		timer_table[COMTROL_TIMER].fn = 0;
		return -ENODEV;
	}

	if (reserved_controller == 0)
		request_region(controller, 4, "Comtrol Rocketport");

	/*
	 * Set up the tty driver structure and then register this
	 * driver with the tty layer.
	 */
	memset(&rocket_driver, 0, sizeof(struct tty_driver));
	rocket_driver.magic = TTY_DRIVER_MAGIC;
	rocket_driver.name = "ttyR";
	rocket_driver.major = TTY_ROCKET_MAJOR;
	rocket_driver.minor_start = 0;
	rocket_driver.num = MAX_RP_PORTS;
	rocket_driver.type = TTY_DRIVER_TYPE_SERIAL;
	rocket_driver.subtype = SERIAL_TYPE_NORMAL;
	rocket_driver.init_termios = tty_std_termios;
	rocket_driver.init_termios.c_cflag =
		B9600 | CS8 | CREAD | HUPCL | CLOCAL;
	rocket_driver.flags = TTY_DRIVER_REAL_RAW;
	rocket_driver.refcount = &rocket_refcount;
	rocket_driver.table = rocket_table;
	rocket_driver.termios = rocket_termios;
	rocket_driver.termios_locked = rocket_termios_locked;

	rocket_driver.open = rp_open;
	rocket_driver.close = rp_close;
	rocket_driver.write = rp_write;
```

```
        rocket_driver.put_char = rp_put_char;
        rocket_driver.write_room = rp_write_room;
        rocket_driver.chars_in_buffer = rp_chars_in_buffer;
        rocket_driver.flush_buffer = rp_flush_buffer;
        rocket_driver.ioctl = rp_ioctl;
        rocket_driver.throttle = rp_throttle;
        rocket_driver.unthrottle = rp_unthrottle;
        rocket_driver.set_termios = rp_set_termios;
        rocket_driver.stop = rp_stop;
        rocket_driver.start = rp_start;
        rocket_driver.hangup = rp_hangup;

        /*
         * The callout device is just like normal device except for
         * the minor number and the subtype code.
         */
        callout_driver = rocket_driver;
        callout_driver.name = "cur";
        callout_driver.major = CUA_ROCKET_MAJOR;
        callout_driver.minor_start = 0;
        callout_driver.subtype = SERIAL_TYPE_CALLOUT;

        retval = tty_register_driver(&callout_driver);
        if (retval < 0) {
                printk("Couldn't install Rocketport callout driver "
                       "(error %d)\n", -retval);
                return -1;
        }

        retval = tty_register_driver(&rocket_driver);
        if (retval < 0) {
                printk("Couldn't install tty Rocketport driver "
                       "(error %d)\n", -retval);
                return -1;
        }
#ifdef ROCKET_DEBUG_OPEN
        printk("Rocketport driver is major %d, callout is %d\n",
               rocket_driver.major, callout_driver.major);
#endif

        return 0;
}

void
cleanup_module( void) {
        int     retval;
        int     i;
        int     released_controller = 0;

        retval = tty_unregister_driver(&callout_driver);
        if (retval) {
                printk("Error %d while trying to unregister "
                       "rocketport callout driver\n", -retval);
        }
        retval = tty_unregister_driver(&rocket_driver);
        if (retval) {
                printk("Error %d while trying to unregister "
                       "rocketport driver\n", -retval);
        }
        for (i = 0; i < MAX_RP_PORTS; i++) {
```

```
                if (rp_table[i])
                        kfree(rp_table[i]);
        }
        for (i=0; i < NUM_BOARDS; i++) {
                if (rcktpt_io_addr[i] <= 0)
                        continue;
                if (rcktpt_io_addr[i] + 0x40 == controller) {
                        released_controller++;
                        release_region(rcktpt_io_addr[i], 68);
                } else
                        release_region(rcktpt_io_addr[i], 64);
                if (released_controller == 0)
                        release_region(controller, 4);
        }
        if (tmp_buf)
                free_page((unsigned long) tmp_buf);
        timer_table[COMTROL_TIMER].fn = 0;
}
```