```c
/*
 * linux/drivers/char/core.c
 *
 * Driver core for serial ports
 *
 * Based on drivers/char/serial.c, by Linus Torvalds, Theodore Ts'o.
 *
 * Copyright 1999 ARM Limited
 * Copyright (C) 2000-2001 Deep Blue Solutions Ltd.
 *
 * 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., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/tty.h>
#include <linux/slab.h>
#include <linux/init.h>
#include <linux/console.h>
#include <linux/serial_core.h>
#include <linux/smp_lock.h>
#include <linux/device.h>
#include <linux/serial.h> /* for serial_state and serial_icounter_struct */
#include <linux/delay.h>

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

#undef DEBUG
#ifdef DEBUG
#define DPRINTK(x...)           printk(x)
#else
#define DPRINTK(x...)           do { } while (0)
#endif

/*
 * This is used to lock changes in serial line configuration.
 */
static DECLARE_MUTEX(port_sem);

#define HIGH_BITS_OFFSET                ((sizeof(long)-sizeof(int))*8)

#define uart_users(state)               ((state)->count + ((state)->info ? (state)->info->blocked_open : 0))

#ifdef CONFIG_SERIAL_CORE_CONSOLE
#define uart_console(port)              ((port)->cons && (port)->cons->index == (port)->line)
#else
#define uart_console(port)              (0)
#endif

static void uart_change_speed(struct uart_state *state, struct termios *old_termios);
static void uart_wait_until_sent(struct tty_struct *tty, int timeout);
static void uart_change_pm(struct uart_state *state, int pm_state);

/*
 * This routine is used by the interrupt handler to schedule processing in
 * the software interrupt portion of the driver.
 */
void uart_write_wakeup(struct uart_port *port)
{
        struct uart_info *info = port->info;
        tasklet_schedule(&info->tlet);
}

static void uart_stop(struct tty_struct *tty)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        port->ops->stop_tx(port, 1);
        spin_unlock_irqrestore(&port->lock, flags);
}

static void __uart_start(struct tty_struct *tty)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;

        if (!uart_circ_empty(&state->info->xmit) && state->info->xmit.buf &&
            !tty->stopped && !tty->hw_stopped)
                port->ops->start_tx(port, 1);
}

static void uart_start(struct tty_struct *tty)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        unsigned long flags;

        spin_lock_irqsave(&port->lock, flags);
        __uart_start(tty);
        spin_unlock_irqrestore(&port->lock, flags);
}

static void uart_tasklet_action(unsigned long data)
{
        struct uart_state *state = (struct uart_state *)data;
        tty_wakeup(state->info->tty);
}

static inline void
uart_update_mctrl(struct uart_port *port, unsigned int set, unsigned int clear)
{
        unsigned long flags;
        unsigned int old;

        spin_lock_irqsave(&port->lock, flags);
        old = port->mctrl;
        port->mctrl = (old & ~clear) | set;
        if (old != port->mctrl)
                port->ops->set_mctrl(port, port->mctrl);
        spin_unlock_irqrestore(&port->lock, flags);
}
```

```
#define uart_set_mctrl(port,set)				uart_update_mctrl(port,set,0)
#define uart_clear_mctrl(port,clear)			uart_update_mctrl(port,0,clear)

/*
 * Startup the port. This will be called once per open. All calls
 * will be serialised by the per-port semaphore.
 */
static int uart_startup(struct uart_state *state, int init_hw)
{
	struct uart_info *info = state->info;
	struct uart_port *port = state->port;
	unsigned long page;
	int retval = 0;

	if (info->flags & UIF_INITIALIZED)
		return 0;

	/*
	 * Set the TTY IO error marker - we will only clear this
	 * once we have successfully opened the port. Also set
	 * up the tty->alt_speed kludge
	 */
	if (info->tty)
		set_bit(TTY_IO_ERROR, &info->tty->flags);

	if (port->type == PORT_UNKNOWN)
		return 0;

	/*
	 * Initialise and allocate the transmit and temporary
	 * buffer.
	 */
	if (!info->xmit.buf) {
		page = get_zeroed_page(GFP_KERNEL);
		if (!page)
			return -ENOMEM;

		info->xmit.buf = (unsigned char *) page;
		uart_circ_clear(&info->xmit);
	}

	retval = port->ops->startup(port);
	if (retval == 0) {
		if (init_hw) {
			/*
			 * Initialise the hardware port settings.
			 */
			uart_change_speed(state, NULL);

			/*
			 * Setup the RTS and DTR signals once the
			 * port is open and ready to respond.
			 */
			if (info->tty->termios->c_cflag & CBAUD)
				uart_set_mctrl(port, TIOCM_RTS | TIOCM_DTR);
		}

		info->flags |= UIF_INITIALIZED;

		clear_bit(TTY_IO_ERROR, &info->tty->flags);
	}
	if (retval && capable(CAP_SYS_ADMIN))
		retval = 0;

	return retval;
}

/*
 * This routine will shutdown a serial port; interrupts are disabled, and
 * DTR is dropped if the hangup on close termio flag is on. Calls to
 * uart_shutdown are serialised by the per-port semaphore.
 */
static void uart_shutdown(struct uart_state *state)
{
	struct uart_info *info = state->info;
	struct uart_port *port = state->port;

	if (!(info->flags & UIF_INITIALIZED))
		return;

	/*
	 * Turn off DTR and RTS early.
	 */
	if (!info->tty || (info->tty->termios->c_cflag & HUPCL))
		uart_clear_mctrl(port, TIOCM_DTR | TIOCM_RTS);

	/*
	 * clear delta_msr_wait queue to avoid mem leaks: we may free
	 * the irq here so the queue might never be woken up. Note
	 * that we won't end up waiting on delta_msr_wait again since
	 * any outstanding file descriptors should be pointing at
	 * hung_up_tty_fops now.
	 */
	wake_up_interruptible(&info->delta_msr_wait);

	/*
	 * Free the IRQ and disable the port.
	 */
	port->ops->shutdown(port);

	/*
	 * Ensure that the IRQ handler isn't running on another CPU.
	 */
	synchronize_irq(port->irq);

	/*
	 * Free the transmit buffer page.
	 */
	if (info->xmit.buf) {
		free_page((unsigned long)info->xmit.buf);
		info->xmit.buf = NULL;
	}

	/*
	 * kill off our tasklet
	 */
	tasklet_kill(&info->tlet);
	if (info->tty)
		set_bit(TTY_IO_ERROR, &info->tty->flags);

	info->flags &= ~UIF_INITIALIZED;
}

/**
```

```
 *	uart_update_timeout - update per-port FIFO timeout.
 *	@port: uart_port structure describing the port
 *	@cflag: termios cflag value
 *	@baud: speed of the port
 *
 *	Set the port FIFO timeout value. The @cflag value should
 *	reflect the actual hardware settings.
 */
void
uart_update_timeout(struct uart_port *port, unsigned int cflag,
		    unsigned int baud)
{
	unsigned int bits;

	/* byte size and parity */
	switch (cflag & CSIZE) {
	case CS5:
		bits = 7;
		break;
	case CS6:
		bits = 8;
		break;
	case CS7:
		bits = 9;
		break;
	default:
		bits = 10;
		break; // CS8
	}

	if (cflag & CSTOPB)
		bits++;
	if (cflag & PARENB)
		bits++;

	/*
	 * The total number of bits to be transmitted in the fifo.
	 */
	bits = bits * port->fifosize;

	/*
	 * Figure the timeout to send the above number of bits.
	 * Add .02 seconds of slop
	 */
	port->timeout = (HZ * bits) / baud + HZ/50;
}

EXPORT_SYMBOL(uart_update_timeout);

/**
 *	uart_get_baud_rate - return baud rate for a particular port
 *	@port: uart_port structure describing the port in question.
 *	@termios: desired termios settings.
 *	@old: old termios (or NULL)
 *	@min: minimum acceptable baud rate
 *	@max: maximum acceptable baud rate
 *
 *	Decode the termios structure into a numeric baud rate,
 *	taking account of the magic 38400 baud rate (with spd_*
 *	flags), and mapping the %B0 rate to 9600 baud.
 *
 *	If the new baud rate is invalid, try the old termios setting.
 *	If it's still invalid, we try 9600 baud.
 *
 *	Update the @termios structure to reflect the baud rate
 *	we're actually going to be using.
 */
unsigned int
uart_get_baud_rate(struct uart_port *port, struct termios *termios,
		   struct termios *old, unsigned int min, unsigned int max)
{
	unsigned int try, baud, altbaud = 38400;
	unsigned int flags = port->flags & UPF_SPD_MASK;

	if (flags == UPF_SPD_HI)
		altbaud = 57600;
	if (flags == UPF_SPD_VHI)
		altbaud = 115200;
	if (flags == UPF_SPD_SHI)
		altbaud = 230400;
	if (flags == UPF_SPD_WARP)
		altbaud = 460800;

	for (try = 0; try < 2; try++) {
		baud = tty_termios_baud_rate(termios);

		/*
		 * The spd_hi, spd_vhi, spd_shi, spd_warp kludge...
		 * Die! Die! Die!
		 */
		if (baud == 38400)
			baud = altbaud;

		/*
		 * Special case: B0 rate.
		 */
		if (baud == 0)
			baud = 9600;

		if (baud >= min && baud <= max)
			return baud;

		/*
		 * Oops, the quotient was zero. Try again with
		 * the old baud rate if possible.
		 */
		termios->c_cflag &= ~CBAUD;
		if (old) {
			termios->c_cflag |= old->c_cflag & CBAUD;
			old = NULL;
			continue;
		}

		/*
		 * As a last resort, if the quotient is zero,
		 * default to 9600 bps
		 */
		termios->c_cflag |= B9600;
	}

	return 0;
}

EXPORT_SYMBOL(uart_get_baud_rate);

/**
```

```
 *	uart_get_divisor - return uart clock divisor
 *	@port: uart_port structure describing the port.
 *	@baud: desired baud rate
 *
 *	Calculate the uart clock divisor for the port.
 */
unsigned int
uart_get_divisor(struct uart_port *port, unsigned int baud)
{
	unsigned int quot;

	/*
	 * Old custom speed handling.
	 */
	if (baud == 38400 && (port->flags & UPF_SPD_MASK) == UPF_SPD_CUST)
		quot = port->custom_divisor;
	else
		quot = (port->uartclk + (8 * baud)) / (16 * baud);

	return quot;
}

EXPORT_SYMBOL(uart_get_divisor);

static void
uart_change_speed(struct uart_state *state, struct termios *old_termios)
{
	struct tty_struct *tty = state->info->tty;
	struct uart_port *port = state->port;
	struct termios *termios;

	/*
	 * If we have no tty, termios, or the port does not exist,
	 * then we can't set the parameters for this port.
	 */
	if (!tty || !tty->termios || port->type == PORT_UNKNOWN)
		return;

	termios = tty->termios;

	/*
	 * Set flags based on termios cflag
	 */
	if (termios->c_cflag & CRTSCTS)
		state->info->flags |= UIF_CTS_FLOW;
	else
		state->info->flags &= ~UIF_CTS_FLOW;

	if (termios->c_cflag & CLOCAL)
		state->info->flags &= ~UIF_CHECK_CD;
	else
		state->info->flags |= UIF_CHECK_CD;

	port->ops->set_termios(port, termios, old_termios);
}

static inline void
__uart_put_char(struct uart_port *port, struct circ_buf *circ, unsigned char c)
{
	unsigned long flags;

	if (!circ->buf)
		return;
```

```
        spin_lock_irqsave(&port->lock, flags);
        if (uart_circ_chars_free(circ) != 0) {
                circ->buf[circ->head] = c;
                circ->head = (circ->head + 1) & (UART_XMIT_SIZE - 1);
        }
        spin_unlock_irqrestore(&port->lock, flags);
}

static void uart_put_char(struct tty_struct *tty, unsigned char ch)
{
        struct uart_state *state = tty->driver_data;

        __uart_put_char(state->port, &state->info->xmit, ch);
}

static void uart_flush_chars(struct tty_struct *tty)
{
        uart_start(tty);
}

static int
uart_write(struct tty_struct *tty, const unsigned char * buf, int count)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        struct circ_buf *circ = &state->info->xmit;
        unsigned long flags;
        int c, ret = 0;

        if (!circ->buf)
                return 0;

        spin_lock_irqsave(&port->lock, flags);
        while (1) {
                c = CIRC_SPACE_TO_END(circ->head, circ->tail, UART_XMIT_SIZE);
                if (count < c)
                        c = count;
                if (c <= 0)
                        break;
                memcpy(circ->buf + circ->head, buf, c);
                circ->head = (circ->head + c) & (UART_XMIT_SIZE - 1);
                buf += c;
                count -= c;
                ret += c;
        }
        spin_unlock_irqrestore(&port->lock, flags);

        uart_start(tty);
        return ret;
}

static int uart_write_room(struct tty_struct *tty)
{
        struct uart_state *state = tty->driver_data;

        return uart_circ_chars_free(&state->info->xmit);
}

static int uart_chars_in_buffer(struct tty_struct *tty)
{
        struct uart_state *state = tty->driver_data;
```

```
	return uart_circ_chars_pending(&state->info->xmit);
}

static void uart_flush_buffer(struct tty_struct *tty)
{
	struct uart_state *state = tty->driver_data;
	struct uart_port *port = state->port;
	unsigned long flags;

	DPRINTK("uart_flush_buffer(%d) called\n", tty->index);

	spin_lock_irqsave(&port->lock, flags);
	uart_circ_clear(&state->info->xmit);
	spin_unlock_irqrestore(&port->lock, flags);
	tty_wakeup(tty);
}

/*
 * This function is used to send a high-priority XON/XOFF character to
 * the device
 */
static void uart_send_xchar(struct tty_struct *tty, char ch)
{
	struct uart_state *state = tty->driver_data;
	struct uart_port *port = state->port;
	unsigned long flags;

	if (port->ops->send_xchar)
		port->ops->send_xchar(port, ch);
	else {
		port->x_char = ch;
		if (ch) {
			spin_lock_irqsave(&port->lock, flags);
			port->ops->start_tx(port, 0);
			spin_unlock_irqrestore(&port->lock, flags);
		}
	}
}

static void uart_throttle(struct tty_struct *tty)
{
	struct uart_state *state = tty->driver_data;

	if (I_IXOFF(tty))
		uart_send_xchar(tty, STOP_CHAR(tty));

	if (tty->termios->c_cflag & CRTSCTS)
		uart_clear_mctrl(state->port, TIOCM_RTS);
}

static void uart_unthrottle(struct tty_struct *tty)
{
	struct uart_state *state = tty->driver_data;
	struct uart_port *port = state->port;

	if (I_IXOFF(tty)) {
		if (port->x_char)
			port->x_char = 0;
		else
			uart_send_xchar(tty, START_CHAR(tty));
	}

	if (tty->termios->c_cflag & CRTSCTS)
```

```
                uart_set_mctrl(port, TIOCM_RTS);
}

static int uart_get_info(struct uart_state *state,
                         struct serial_struct __user *retinfo)
{
        struct uart_port *port = state->port;
        struct serial_struct tmp;

        memset(&tmp, 0, sizeof(tmp));
        tmp.type            = port->type;
        tmp.line            = port->line;
        tmp.port            = port->iobase;
        if (HIGH_BITS_OFFSET)
                tmp.port_high = (long) port->iobase >> HIGH_BITS_OFFSET;
        tmp.irq             = port->irq;
        tmp.flags           = port->flags;
        tmp.xmit_fifo_size  = port->fifosize;
        tmp.baud_base       = port->uartclk / 16;
        tmp.close_delay     = state->close_delay / 10;
        tmp.closing_wait    = state->closing_wait == USF_CLOSING_WAIT_NONE ?
                                ASYNC_CLOSING_WAIT_NONE :
                                state->closing_wait / 10;
        tmp.custom_divisor = port->custom_divisor;
        tmp.hub6            = port->hub6;
        tmp.io_type         = port->iotype;
        tmp.iomem_reg_shift = port->regshift;
        tmp.iomem_base      = (void *)port->mapbase;

        if (copy_to_user(retinfo, &tmp, sizeof(*retinfo)))
                return -EFAULT;
        return 0;
}

static int uart_set_info(struct uart_state *state,
                         struct serial_struct __user *newinfo)
{
        struct serial_struct new_serial;
        struct uart_port *port = state->port;
        unsigned long new_port;
        unsigned int change_irq, change_port, old_flags, closing_wait;
        unsigned int old_custom_divisor, close_delay;
        int retval = 0;

        if (copy_from_user(&new_serial, newinfo, sizeof(new_serial)))
                return -EFAULT;

        new_port = new_serial.port;
        if (HIGH_BITS_OFFSET)
                new_port += (unsigned long) new_serial.port_high << HIGH_BITS_OFFSET;

        new_serial.irq = irq_canonicalize(new_serial.irq);
        close_delay = new_serial.close_delay * 10;
        closing_wait = new_serial.closing_wait == ASYNC_CLOSING_WAIT_NONE ?
                        USF_CLOSING_WAIT_NONE : new_serial.closing_wait * 10;

        /*
         * This semaphore protects state->count. It is also
         * very useful to prevent opens. Also, take the
         * port configuration semaphore to make sure that a
         * module insertion/removal doesn't change anything
         * under us.
         */
```

```
	down(&state->sem);

	change_irq = new_serial.irq != port->irq;

	/*
	 * Since changing the 'type' of the port changes its resource
	 * allocations, we should treat type changes the same as
	 * IO port changes.
	 */
	change_port = new_port != port->iobase ||
		      (unsigned long)new_serial.iomem_base != port->mapbase ||
		      new_serial.hub6 != port->hub6 ||
		      new_serial.io_type != port->iotype ||
		      new_serial.iomem_reg_shift != port->regshift ||
		      new_serial.type != port->type;

	old_flags = port->flags;
	old_custom_divisor = port->custom_divisor;

	if (!capable(CAP_SYS_ADMIN)) {
		retval = -EPERM;
		if (change_irq || change_port ||
		    (new_serial.baud_base != port->uartclk / 16) ||
		    (close_delay != state->close_delay) ||
		    (closing_wait != state->closing_wait) ||
		    (new_serial.xmit_fifo_size != port->fifosize) ||
		    (((new_serial.flags ^ old_flags) & ~UPF_USR_MASK) != 0))
			goto exit;
		port->flags = ((port->flags & ~UPF_USR_MASK) |
			       (new_serial.flags & UPF_USR_MASK));
		port->custom_divisor = new_serial.custom_divisor;
		goto check_and_exit;
	}

	/*
	 * Ask the low level driver to verify the settings.
	 */
	if (port->ops->verify_port)
		retval = port->ops->verify_port(port, &new_serial);

	if ((new_serial.irq >= NR_IRQS) || (new_serial.irq < 0) ||
	    (new_serial.baud_base < 9600))
		retval = -EINVAL;

	if (retval)
		goto exit;

	if (change_port || change_irq) {
		retval = -EBUSY;

		/*
		 * Make sure that we are the sole user of this port.
		 */
		if (uart_users(state) > 1)
			goto exit;

		/*
		 * We need to shutdown the serial port at the old
		 * port/type/irq combination.
		 */
		uart_shutdown(state);
	}
```

```
        if (change_port) {
                unsigned long old_iobase, old_mapbase;
                unsigned int old_type, old_iotype, old_hub6, old_shift;

                old_iobase = port->iobase;
                old_mapbase = port->mapbase;
                old_type = port->type;
                old_hub6 = port->hub6;
                old_iotype = port->iotype;
                old_shift = port->regshift;

                /*
                 * Free and release old regions
                 */
                if (old_type != PORT_UNKNOWN)
                        port->ops->release_port(port);

                port->iobase = new_port;
                port->type = new_serial.type;
                port->hub6 = new_serial.hub6;
                port->iotype = new_serial.io_type;
                port->regshift = new_serial.iomem_reg_shift;
                port->mapbase = (unsigned long)new_serial.iomem_base;

                /*
                 * Claim and map the new regions
                 */
                if (port->type != PORT_UNKNOWN) {
                        retval = port->ops->request_port(port);
                } else {
                        /* Always success - Jean II */
                        retval = 0;
                }

                /*
                 * If we fail to request resources for the
                 * new port, try to restore the old settings.
                 */
                if (retval && old_type != PORT_UNKNOWN) {
                        port->iobase = old_iobase;
                        port->type = old_type;
                        port->hub6 = old_hub6;
                        port->iotype = old_iotype;
                        port->regshift = old_shift;
                        port->mapbase = old_mapbase;
                        retval = port->ops->request_port(port);
                        /*
                         * If we failed to restore the old settings,
                         * we fail like this.
                         */
                        if (retval)
                                port->type = PORT_UNKNOWN;

                        /*
                         * We failed anyway.
                         */
                        retval = -EBUSY;
                }
        }

        port->irq              = new_serial.irq;
        port->uartclk          = new_serial.baud_base * 16;
        port->flags            = (port->flags & ~UPF_CHANGE_MASK) |
                                        (new_serial.flags & UPF_CHANGE_MASK);
        port->custom_divisor            = new_serial.custom_divisor;
        state->close_delay              = close_delay;
        state->closing_wait             = closing_wait;
        port->fifosize                  = new_serial.xmit_fifo_size;
        if (state->info->tty)
                state->info->tty->low_latency =
                        (port->flags & UPF_LOW_LATENCY) ? 1 : 0;

 check_and_exit:
        retval = 0;
        if (port->type == PORT_UNKNOWN)
                goto exit;
        if (state->info->flags & UIF_INITIALIZED) {
                if (((old_flags ^ port->flags) & UPF_SPD_MASK) ||
                    old_custom_divisor != port->custom_divisor) {
                        /*
                         * If they're setting up a custom divisor or speed,
                         * instead of clearing it, then bitch about it. No
                         * need to rate-limit; it's CAP_SYS_ADMIN only.
                         */
                        if (port->flags & UPF_SPD_MASK) {
                                char buf[64];
                                printk(KERN_NOTICE
                                       "%s sets custom speed on %s. This "
                                       "is deprecated.\n", current->comm,
                                       tty_name(state->info->tty, buf));
                        }
                        uart_change_speed(state, NULL);
                }
        } else
                retval = uart_startup(state, 1);
 exit:
        up(&state->sem);
        return retval;
}


/*
 * uart_get_lsr_info - get line status register info.
 * Note: uart_ioctl protects us against hangups.
 */
static int uart_get_lsr_info(struct uart_state *state,
                             unsigned int __user *value)
{
        struct uart_port *port = state->port;
        unsigned int result;

        result = port->ops->tx_empty(port);

        /*
         * If we're about to load something into the transmit
         * register, we'll pretend the transmitter isn't empty to
         * avoid a race condition (depending on when the transmit
         * interrupt happens).
         */
        if (port->x_char ||
            ((uart_circ_chars_pending(&state->info->xmit) > 0) &&
             !state->info->tty->stopped && !state->info->tty->hw_stopped))
                result &= ~TIOCSER_TEMT;

        return put_user(result, value);
}
```

```
static int uart_tiocmget(struct tty_struct *tty, struct file *file)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        int result = -EIO;

        down(&state->sem);
        if ((!file || !tty_hung_up_p(file)) &&
            !(tty->flags & (1 << TTY_IO_ERROR))) {
                result = port->mctrl;
                result |= port->ops->get_mctrl(port);
        }
        up(&state->sem);

        return result;
}

static int
uart_tiocmset(struct tty_struct *tty, struct file *file,
              unsigned int set, unsigned int clear)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        int ret = -EIO;

        down(&state->sem);
        if ((!file || !tty_hung_up_p(file)) &&
            !(tty->flags & (1 << TTY_IO_ERROR))) {
                uart_update_mctrl(port, set, clear);
                ret = 0;
        }
        up(&state->sem);
        return ret;
}

static void uart_break_ctl(struct tty_struct *tty, int break_state)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;

        BUG_ON(!kernel_locked());

        down(&state->sem);

        if (port->type != PORT_UNKNOWN)
                port->ops->break_ctl(port, break_state);

        up(&state->sem);
}

static int uart_do_autoconfig(struct uart_state *state)
{
        struct uart_port *port = state->port;
        int flags, ret;

        if (!capable(CAP_SYS_ADMIN))
                return -EPERM;

        /*
         * Take the per-port semaphore. This prevents count from
         * changing, and hence any extra opens of the port while
         * we're auto-configuring.
         */
        if (down_interruptible(&state->sem))
                return -ERESTARTSYS;

        ret = -EBUSY;
        if (uart_users(state) == 1) {
                uart_shutdown(state);

                /*
                 * If we already have a port type configured,
                 * we must release its resources.
                 */
                if (port->type != PORT_UNKNOWN)
                        port->ops->release_port(port);

                flags = UART_CONFIG_TYPE;
                if (port->flags & UPF_AUTO_IRQ)
                        flags |= UART_CONFIG_IRQ;

                /*
                 * This will claim the ports resources if
                 * a port is found.
                 */
                port->ops->config_port(port, flags);

                ret = uart_startup(state, 1);
        }
        up(&state->sem);
        return ret;
}

/*
 * Wait for any of the 4 modem inputs (DCD,RI,DSR,CTS) to change
 * - mask passed in arg for lines of interest
 *       (use |'ed TIOCM_RNG/DSR/CD/CTS for masking)
 * Caller should use TIOCGICOUNT to see which one it was
 */
static int
uart_wait_modem_status(struct uart_state *state, unsigned long arg)
{
        struct uart_port *port = state->port;
        DECLARE_WAITQUEUE(wait, current);
        struct uart_icount cprev, cnow;
        int ret;

        /*
         * note the counters on entry
         */
        spin_lock_irq(&port->lock);
        memcpy(&cprev, &port->icount, sizeof(struct uart_icount));

        /*
         * Force modem status interrupts on
         */
        port->ops->enable_ms(port);
        spin_unlock_irq(&port->lock);

        add_wait_queue(&state->info->delta_msr_wait, &wait);
        for (;;) {
                spin_lock_irq(&port->lock);
                memcpy(&cnow, &port->icount, sizeof(struct uart_icount));
                spin_unlock_irq(&port->lock);

```

```
                        set_current_state(TASK_INTERRUPTIBLE);

                        if (((arg & TIOCM_RNG) && (cnow.rng != cprev.rng)) ||
                            ((arg & TIOCM_DSR) && (cnow.dsr != cprev.dsr)) ||
                            ((arg & TIOCM_CD) && (cnow.dcd != cprev.dcd)) ||
                            ((arg & TIOCM_CTS) && (cnow.cts != cprev.cts))) {
                                ret = 0;
                                break;
                        }

                        schedule();

                        /* see if a signal did it */
                        if (signal_pending(current)) {
                                ret = -ERESTARTSYS;
                                break;
                        }

                        cprev = cnow;
                }

                current->state = TASK_RUNNING;
                remove_wait_queue(&state->info->delta_msr_wait, &wait);

                return ret;
}

/*
 * Get counter of input serial line interrupts (DCD,RI,DSR,CTS)
 * Return: write counters to the user passed counter struct
 * NB: both 1->0 and 0->1 transitions are counted except for
 *         RI where only 0->1 is counted.
 */
static int uart_get_count(struct uart_state *state,
                          struct serial_icounter_struct __user *icnt)
{
        struct serial_icounter_struct icount;
        struct uart_icount cnow;
        struct uart_port *port = state->port;

        spin_lock_irq(&port->lock);
        memcpy(&cnow, &port->icount, sizeof(struct uart_icount));
        spin_unlock_irq(&port->lock);

        icount.cts         = cnow.cts;
        icount.dsr         = cnow.dsr;
        icount.rng         = cnow.rng;
        icount.dcd         = cnow.dcd;
        icount.rx          = cnow.rx;
        icount.tx          = cnow.tx;
        icount.frame       = cnow.frame;
        icount.overrun     = cnow.overrun;
        icount.parity      = cnow.parity;
        icount.brk         = cnow.brk;
        icount.buf_overrun = cnow.buf_overrun;

        return copy_to_user(icnt, &icount, sizeof(icount)) ? -EFAULT : 0;
}

/*
 * Called via sys_ioctl under the BKL. We can use spin_lock_irq() here.
 */
static int
```

```c
uart_ioctl(struct tty_struct *tty, struct file *filp, unsigned int cmd,
	   unsigned long arg)
{
	struct uart_state *state = tty->driver_data;
	void __user *uarg = (void __user *)arg;
	int ret = -ENOIOCTLCMD;

	BUG_ON(!kernel_locked());

	/*
	 * These ioctls don't rely on the hardware to be present.
	 */
	switch (cmd) {
	case TIOCGSERIAL:
		ret = uart_get_info(state, uarg);
		break;

	case TIOCSSERIAL:
		ret = uart_set_info(state, uarg);
		break;

	case TIOCSERCONFIG:
		ret = uart_do_autoconfig(state);
		break;

	case TIOCSERGWILD: /* obsolete */
	case TIOCSERSWILD: /* obsolete */
		ret = 0;
		break;
	}

	if (ret != -ENOIOCTLCMD)
		goto out;

	if (tty->flags & (1 << TTY_IO_ERROR)) {
		ret = -EIO;
		goto out;
	}

	/*
	 * The following should only be used when hardware is present.
	 */
	switch (cmd) {
	case TIOCMIWAIT:
		ret = uart_wait_modem_status(state, arg);
		break;

	case TIOCGICOUNT:
		ret = uart_get_count(state, uarg);
		break;
	}

	if (ret != -ENOIOCTLCMD)
		goto out;

	down(&state->sem);

	if (tty_hung_up_p(filp)) {
		ret = -EIO;
		goto out_up;
	}

	/*
```

```
         * All these rely on hardware being present and need to be
         * protected against the tty being hung up.
         */
        switch (cmd) {
        case TIOCSERGETLSR: /* Get line status register */
                ret = uart_get_lsr_info(state, uarg);
                break;

        default: {
                struct uart_port *port = state->port;
                if (port->ops->ioctl)
                        ret = port->ops->ioctl(port, cmd, arg);
                break;
        }
        }
 out_up:
        up(&state->sem);
 out:
        return ret;
}

static void uart_set_termios(struct tty_struct *tty, struct termios *old_termios)
{
        struct uart_state *state = tty->driver_data;
        unsigned long flags;
        unsigned int cflag = tty->termios->c_cflag;

        BUG_ON(!kernel_locked());

        /*
         * These are the bits that are used to setup various
         * flags in the low level driver.
         */
#define RELEVANT_IFLAG(iflag)          ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))

        if ((cflag ^ old_termios->c_cflag) == 0 &&
            RELEVANT_IFLAG(tty->termios->c_iflag ^ old_termios->c_iflag) == 0)
                return;

        uart_change_speed(state, old_termios);

        /* Handle transition to B0 status */
        if ((old_termios->c_cflag & CBAUD) && !(cflag & CBAUD))
                uart_clear_mctrl(state->port, TIOCM_RTS | TIOCM_DTR);

        /* Handle transition away from B0 status */
        if (!(old_termios->c_cflag & CBAUD) && (cflag & CBAUD)) {
                unsigned int mask = TIOCM_DTR;
                if (!(cflag & CRTSCTS) ||
                    !test_bit(TTY_THROTTLED, &tty->flags))
                        mask |= TIOCM_RTS;
                uart_set_mctrl(state->port, mask);
        }

        /* Handle turning off CRTSCTS */
        if ((old_termios->c_cflag & CRTSCTS) && !(cflag & CRTSCTS)) {
                spin_lock_irqsave(&state->port->lock, flags);
                tty->hw_stopped = 0;
                __uart_start(tty);
                spin_unlock_irqrestore(&state->port->lock, flags);
        }

#if 0
```

```c
        /*
         * No need to wake up processes in open wait, since they
         * sample the CLOCAL flag once, and don't recheck it.
         * XXX It's not clear whether the current behavior is correct
         * or not. Hence, this may change.....
         */
        if (!(old_termios->c_cflag & CLOCAL) &&
            (tty->termios->c_cflag & CLOCAL))
                wake_up_interruptible(&state->info->open_wait);
#endif
}

/*
 * In 2.4.5, calls to this will be serialized via the BKL in
 * linux/drivers/char/tty_io.c:tty_release()
 * linux/drivers/char/tty_io.c:do_tty_handup()
 */
static void uart_close(struct tty_struct *tty, struct file *filp)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port;

        BUG_ON(!kernel_locked());

        if (!state || !state->port)
                return;

        port = state->port;

        DPRINTK("uart_close(%d) called\n", port->line);

        down(&state->sem);

        if (tty_hung_up_p(filp))
                goto done;

        if ((tty->count == 1) && (state->count != 1)) {
                /*
                 * Uh, oh. tty->count is 1, which means that the tty
                 * structure will be freed. state->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(KERN_ERR "uart_close: bad serial port count; tty->count is 1, "
                       "state->count is %d\n", state->count);
                state->count = 1;
        }
        if (--state->count < 0) {
                printk(KERN_ERR "uart_close: bad serial port count for %s: %d\n",
                       tty->name, state->count);
                state->count = 0;
        }
        if (state->count)
                goto done;

        /*
         * Now we wait for the transmit buffer to clear; and we notify
         * the line discipline to only process XON/XOFF characters by
         * setting tty->closing.
         */
        tty->closing = 1;
        if (state->closing_wait != USF_CLOSING_WAIT_NONE)
                tty_wait_until_sent(tty, msecs_to_jiffies(state->closing_wait));

        /*
         * At this point, we stop accepting input. To do this, we
         * disable the receive line status interrupts.
         */
        if (state->info->flags & UIF_INITIALIZED) {
                unsigned long flags;
                spin_lock_irqsave(&port->lock, flags);
                port->ops->stop_rx(port);
                spin_unlock_irqrestore(&port->lock, flags);
                /*
                 * Before we drop DTR, make sure the UART transmitter
                 * has completely drained; this is especially
                 * important if there is a transmit FIFO!
                 */
                uart_wait_until_sent(tty, port->timeout);
        }

        uart_shutdown(state);
        uart_flush_buffer(tty);

        tty_ldisc_flush(tty);

        tty->closing = 0;
        state->info->tty = NULL;

        if (state->info->blocked_open) {
                if (state->close_delay)
                        msleep_interruptible(state->close_delay);
        } else if (!uart_console(port)) {
                uart_change_pm(state, 3);
        }

        /*
         * Wake up anyone trying to open this port.
         */
        state->info->flags &= ~UIF_NORMAL_ACTIVE;
        wake_up_interruptible(&state->info->open_wait);

 done:
        up(&state->sem);
}

static void uart_wait_until_sent(struct tty_struct *tty, int timeout)
{
        struct uart_state *state = tty->driver_data;
        struct uart_port *port = state->port;
        unsigned long char_time, expire;

        BUG_ON(!kernel_locked());

        if (port->type == PORT_UNKNOWN || port->fifosize == 0)
                return;

        /*
         * Set the check interval to be 1/5 of the estimated time to
         * send a single character, and make it at least 1. The check
         * interval should also be less than the timeout.
         *
         * Note: we have to use pretty tight timings here to satisfy
         * the NIST-PCTS.
```

```
	 */
	char_time = (port->timeout - HZ/50) / port->fifosize;
	char_time = char_time / 5;
	if (char_time == 0)
		char_time = 1;
	if (timeout && timeout < char_time)
		char_time = timeout;

	/*
	 * If the transmitter hasn't cleared in twice the approximate
	 * amount of time to send the entire FIFO, it probably won't
	 * ever clear. This assumes the UART isn't doing flow
	 * control, which is currently the case. Hence, if it ever
	 * takes longer than port->timeout, this is probably due to a
	 * UART bug of some kind. So, we clamp the timeout parameter at
	 * 2*port->timeout.
	 */
	if (timeout == 0 || timeout > 2 * port->timeout)
		timeout = 2 * port->timeout;

	expire = jiffies + timeout;

	DPRINTK("uart_wait_until_sent(%d), jiffies=%lu, expire=%lu...\n",
		port->line, jiffies, expire);

	/*
	 * Check whether the transmitter is empty every 'char_time'.
	 * 'timeout' / 'expire' give us the maximum amount of time
	 * we wait.
	 */
	while (!port->ops->tx_empty(port)) {
		msleep_interruptible(jiffies_to_msecs(char_time));
		if (signal_pending(current))
			break;
		if (time_after(jiffies, expire))
			break;
	}
	set_current_state(TASK_RUNNING); /* might not be needed */
}

/*
 * This is called with the BKL held in
 * linux/drivers/char/tty_io.c:do_tty_hangup()
 * We're called from the eventd thread, so we can sleep for
 * a _short_ time only.
 */
static void uart_hangup(struct tty_struct *tty)
{
	struct uart_state *state = tty->driver_data;

	BUG_ON(!kernel_locked());
	DPRINTK("uart_hangup(%d)\n", state->port->line);

	down(&state->sem);
	if (state->info && state->info->flags & UIF_NORMAL_ACTIVE) {
		uart_flush_buffer(tty);
		uart_shutdown(state);
		state->count = 0;
		state->info->flags &= ~UIF_NORMAL_ACTIVE;
		state->info->tty = NULL;
		wake_up_interruptible(&state->info->open_wait);
		wake_up_interruptible(&state->info->delta_msr_wait);
	}
```

```
        up(&state->sem);
}

/*
 * Copy across the serial console cflag setting into the termios settings
 * for the initial open of the port. This allows continuity between the
 * kernel settings, and the settings init adopts when it opens the port
 * for the first time.
 */
static void uart_update_termios(struct uart_state *state)
{
        struct tty_struct *tty = state->info->tty;
        struct uart_port *port = state->port;

        if (uart_console(port) && port->cons->cflag) {
                tty->termios->c_cflag = port->cons->cflag;
                port->cons->cflag = 0;
        }

        /*
         * If the device failed to grab its irq resources,
         * or some other error occurred, don't try to talk
         * to the port hardware.
         */
        if (!(tty->flags & (1 << TTY_IO_ERROR))) {
                /*
                 * Make termios settings take effect.
                 */
                uart_change_speed(state, NULL);

                /*
                 * And finally enable the RTS and DTR signals.
                 */
                if (tty->termios->c_cflag & CBAUD)
                        uart_set_mctrl(port, TIOCM_DTR | TIOCM_RTS);
        }
}

/*
 * Block the open until the port is ready. We must be called with
 * the per-port semaphore held.
 */
static int
uart_block_til_ready(struct file *filp, struct uart_state *state)
{
        DECLARE_WAITQUEUE(wait, current);
        struct uart_info *info = state->info;
        struct uart_port *port = state->port;

        info->blocked_open++;
        state->count--;

        add_wait_queue(&info->open_wait, &wait);
        while (1) {
                set_current_state(TASK_INTERRUPTIBLE);

                /*
                 * If we have been hung up, tell userspace/restart open.
                 */
                if (tty_hung_up_p(filp) || info->tty == NULL)
                        break;

                /*
```

```
		 * If the port has been closed, tell userspace/restart open.
		 */
		if (!(info->flags & UIF_INITIALIZED))
			break;

		/*
		 * If non-blocking mode is set, or CLOCAL mode is set,
		 * we don't want to wait for the modem status lines to
		 * indicate that the port is ready.
		 *
		 * Also, if the port is not enabled/configured, we want
		 * to allow the open to succeed here. Note that we will
		 * have set TTY_IO_ERROR for a non-existant port.
		 */
		if ((filp->f_flags & O_NONBLOCK) ||
		    (info->tty->termios->c_cflag & CLOCAL) ||
		    (info->tty->flags & (1 << TTY_IO_ERROR))) {
			break;
		}

		/*
		 * Set DTR to allow modem to know we're waiting. Do
		 * not set RTS here - we want to make sure we catch
		 * the data from the modem.
		 */
		if (info->tty->termios->c_cflag & CBAUD)
			uart_set_mctrl(port, TIOCM_DTR);

		/*
		 * and wait for the carrier to indicate that the
		 * modem is ready for us.
		 */
		if (port->ops->get_mctrl(port) & TIOCM_CAR)
			break;

		up(&state->sem);
		schedule();
		down(&state->sem);

		if (signal_pending(current))
			break;
	}
	set_current_state(TASK_RUNNING);
	remove_wait_queue(&info->open_wait, &wait);

	state->count++;
	info->blocked_open--;

	if (signal_pending(current))
		return -ERESTARTSYS;

	if (!info->tty || tty_hung_up_p(filp))
		return -EAGAIN;

	return 0;
}

static struct uart_state *uart_get(struct uart_driver *drv, int line)
{
	struct uart_state *state;

	down(&port_sem);
	state = drv->state + line;
	if (down_interruptible(&state->sem)) {
		state = ERR_PTR(-ERESTARTSYS);
		goto out;
	}

	state->count++;
	if (!state->port) {
		state->count--;
		up(&state->sem);
		state = ERR_PTR(-ENXIO);
		goto out;
	}

	if (!state->info) {
		state->info = kmalloc(sizeof(struct uart_info), GFP_KERNEL);
		if (state->info) {
			memset(state->info, 0, sizeof(struct uart_info));
			init_waitqueue_head(&state->info->open_wait);
			init_waitqueue_head(&state->info->delta_msr_wait);

			/*
			 * Link the info into the other structures.
			 */
			state->port->info = state->info;

			tasklet_init(&state->info->tlet, uart_tasklet_action,
				     (unsigned long)state);
		} else {
			state->count--;
			up(&state->sem);
			state = ERR_PTR(-ENOMEM);
		}
	}

 out:
	up(&port_sem);
	return state;
}

/*
 * In 2.4.5, calls to uart_open are serialised by the BKL in
 *   linux/fs/devices.c:chrdev_open()
 * Note that if this fails, then uart_close() _will_ be called.
 *
 * In time, we want to scrap the "opening nonpresent ports"
 * behaviour and implement an alternative way for setserial
 * to set base addresses/ports/types.  This will allow us to
 * get rid of a certain amount of extra tests.
 */
static int uart_open(struct tty_struct *tty, struct file *filp)
{
	struct uart_driver *drv = (struct uart_driver *)tty->driver->driver_state;
	struct uart_state *state;
	int retval, line = tty->index;

	BUG_ON(!kernel_locked());
	DPRINTK("uart_open(%d) called\n", line);

	/*
	 * tty->driver->num won't change, so we won't fail here with
	 * tty->driver_data set to something non-NULL (and therefore
	 * we won't get caught by uart_close()).
	 */
```

```
        retval = -ENODEV;
        if (line >= tty->driver->num)
                goto fail;

        /*
         * We take the semaphore inside uart_get to guarantee that we won't
         * be re-entered while allocating the info structure, or while we
         * request any IRQs that the driver may need. This also has the nice
         * side-effect that it delays the action of uart_hangup, so we can
         * guarantee that info->tty will always contain something reasonable.
         */
        state = uart_get(drv, line);
        if (IS_ERR(state)) {
                retval = PTR_ERR(state);
                goto fail;
        }

        /*
         * Once we set tty->driver_data here, we are guaranteed that
         * uart_close() will decrement the driver module use count.
         * Any failures from here onwards should not touch the count.
         */
        tty->driver_data = state;
        tty->low_latency = (state->port->flags & UPF_LOW_LATENCY) ? 1 : 0;
        tty->alt_speed = 0;
        state->info->tty = tty;

        /*
         * If the port is in the middle of closing, bail out now.
         */
        if (tty_hung_up_p(filp)) {
                retval = -EAGAIN;
                state->count--;
                up(&state->sem);
                goto fail;
        }

        /*
         * Make sure the device is in D0 state.
         */
        if (state->count == 1)
                uart_change_pm(state, 0);

        /*
         * Start up the serial port.
         */
        retval = uart_startup(state, 0);

        /*
         * If we succeeded, wait until the port is ready.
         */
        if (retval == 0)
                retval = uart_block_til_ready(filp, state);
        up(&state->sem);

        /*
         * If this is the first open to succeed, adjust things to suit.
         */
        if (retval == 0 && !(state->info->flags & UIF_NORMAL_ACTIVE)) {
                state->info->flags |= UIF_NORMAL_ACTIVE;

                uart_update_termios(state);
        }
```

```
 fail:
        return retval;
}

static const char *uart_type(struct uart_port *port)
{
        const char *str = NULL;

        if (port->ops->type)
                str = port->ops->type(port);

        if (!str)
                str = "unknown";

        return str;
}

#ifdef CONFIG_PROC_FS

static int uart_line_info(char *buf, struct uart_driver *drv, int i)
{
        struct uart_state *state = drv->state + i;
        struct uart_port *port = state->port;
        char stat_buf[32];
        unsigned int status;
        int ret;

        if (!port)
                return 0;

        ret = sprintf(buf, "%d: uart:%s %s%08lX irq:%d",
                        port->line, uart_type(port),
                        port->iotype == UPIO_MEM ? "mmio:0x" : "port:",
                        port->iotype == UPIO_MEM ? port->mapbase :
                                                (unsigned long) port->iobase,
                        port->irq);

        if (port->type == PORT_UNKNOWN) {
                strcat(buf, "\n");
                return ret + 1;
        }

        if(capable(CAP_SYS_ADMIN))
        {
                status = port->ops->get_mctrl(port);

                ret += sprintf(buf + ret, " tx:%d rx:%d",
                                port->icount.tx, port->icount.rx);
                if (port->icount.frame)
                        ret += sprintf(buf + ret, " fe:%d",
                                port->icount.frame);
                if (port->icount.parity)
                        ret += sprintf(buf + ret, " pe:%d",
                                port->icount.parity);
                if (port->icount.brk)
                        ret += sprintf(buf + ret, " brk:%d",
                                port->icount.brk);
                if (port->icount.overrun)
                        ret += sprintf(buf + ret, " oe:%d",
                                port->icount.overrun);

#define INFOBIT(bit,str) \
        if (port->mctrl & (bit)) \
                strncat(stat_buf, (str), sizeof(stat_buf) - \
                        strlen(stat_buf) - 2)
#define STATBIT(bit,str) \
        if (status & (bit)) \
                strncat(stat_buf, (str), sizeof(stat_buf) - \
                       strlen(stat_buf) - 2)

                stat_buf[0] = '\0';
                stat_buf[1] = '\0';
                INFOBIT(TIOCM_RTS, "|RTS");
                STATBIT(TIOCM_CTS, "|CTS");
                INFOBIT(TIOCM_DTR, "|DTR");
                STATBIT(TIOCM_DSR, "|DSR");
                STATBIT(TIOCM_CAR, "|CD");
                STATBIT(TIOCM_RNG, "|RI");
                if (stat_buf[0])
                        stat_buf[0] = ' ';
                strcat(stat_buf, "\n");

                ret += sprintf(buf + ret, stat_buf);
        } else {
                strcat(buf, "\n");
                ret++;
        }
#undef STATBIT
#undef INFOBIT
        return ret;
}

static int uart_read_proc(char *page, char **start, off_t off,
                          int count, int *eof, void *data)
{
        struct tty_driver *ttydrv = data;
        struct uart_driver *drv = ttydrv->driver_state;
        int i, len = 0, l;
        off_t begin = 0;

        len += sprintf(page, "serinfo:1.0 driver%s%s revision:%s\n",
                        "", "", "");
        for (i = 0; i <         drv->nr && len < PAGE_SIZE - 96; i++) {
                l = uart_line_info(page + len, drv, i);
                len += l;
                if (len + begin > off + count)
                        goto done;
                if (len         + begin < off) {
                        begin += len;
                        len = 0;
                }
        }
        *eof = 1;
 done:
        if (off >= len + begin)
                return 0;
        *start = page + (off - begin);
        return (count < begin + len - off) ? count : (begin + len - off);
}
#endif

#ifdef CONFIG_SERIAL_CORE_CONSOLE
/*
 *      Check whether an invalid uart number has been specified, and
 *      if so, search for the first available port that does have
 *	console support.
 */
struct uart_port * __init
uart_get_console(struct uart_port *ports, int nr, struct console *co)
{
	int idx = co->index;

	if (idx < 0 || idx >= nr || (ports[idx].iobase == 0 &&
				     ports[idx].membase == NULL))
		for (idx = 0; idx < nr; idx++)
			if (ports[idx].iobase != 0 ||
			    ports[idx].membase != NULL)
				break;

	co->index = idx;

	return ports + idx;
}

/**
 *	uart_parse_options - Parse serial port baud/parity/bits/flow contro.
 *	@options: pointer to option string
 *	@baud: pointer to an 'int' variable for the baud rate.
 *	@parity: pointer to an 'int' variable for the parity.
 *	@bits: pointer to an 'int' variable for the number of data bits.
 *	@flow: pointer to an 'int' variable for the flow control character.
 *
 *	uart_parse_options decodes a string containing the serial console
 *	options. The format of the string is <baud><parity><bits><flow>,
 *	eg: 115200n8r
 */
void __init
uart_parse_options(char *options, int *baud, int *parity, int *bits, int *flow)
{
	char *s = options;

	*baud = simple_strtoul(s, NULL, 10);
	while (*s >= '0' && *s <= '9')
		s++;
	if (*s)
		*parity = *s++;
	if (*s)
		*bits = *s++ - '0';
	if (*s)
		*flow = *s;
}

struct baud_rates {
	unsigned int rate;
	unsigned int cflag;
};

static struct baud_rates baud_rates[] = {
	{ 921600, B921600 },
	{ 460800, B460800 },
	{ 230400, B230400 },
	{ 115200, B115200 },
	{  57600, B57600  },
	{  38400, B38400  },
	{  19200, B19200  },
	{   9600, B9600   },
	{   4800, B4800   },
	{   2400, B2400   },
	{	1200, B1200	},
	{	   0, B38400	}
};

/**
 *	uart_set_options - setup the serial console parameters
 *	@port: pointer to the serial ports uart_port structure
 *	@co: console pointer
 *	@baud: baud rate
 *	@parity: parity character - 'n' (none), 'o' (odd), 'e' (even)
 *	@bits: number of data bits
 *	@flow: flow control character - 'r' (rts)
 */
int __init
uart_set_options(struct uart_port *port, struct console *co,
		 int baud, int parity, int bits, int flow)
{
	struct termios termios;
	int i;

	memset(&termios, 0, sizeof(struct termios));

	termios.c_cflag = CREAD | HUPCL | CLOCAL;

	/*
	 * Construct a cflag setting.
	 */
	for (i = 0; baud_rates[i].rate; i++)
		if (baud_rates[i].rate <= baud)
			break;

	termios.c_cflag |= baud_rates[i].cflag;

	if (bits == 7)
		termios.c_cflag |= CS7;
	else
		termios.c_cflag |= CS8;

	switch (parity) {
	case 'o': case 'O':
		termios.c_cflag |= PARODD;
		/*fall through*/
	case 'e': case 'E':
		termios.c_cflag |= PARENB;
		break;
	}

	if (flow == 'r')
		termios.c_cflag |= CRTSCTS;

	port->ops->set_termios(port, &termios, NULL);
	co->cflag = termios.c_cflag;

	return 0;
}
#endif /* CONFIG_SERIAL_CORE_CONSOLE */

static void uart_change_pm(struct uart_state *state, int pm_state)
{
	struct uart_port *port = state->port;
	if (port->ops->pm)
		port->ops->pm(port, pm_state, state->pm_state);
	state->pm_state = pm_state;
```

```
}

int uart_suspend_port(struct uart_driver *drv, struct uart_port *port)
{
	struct uart_state *state = drv->state + port->line;

	down(&state->sem);

	if (state->info && state->info->flags & UIF_INITIALIZED) {
		struct uart_ops *ops = port->ops;

		spin_lock_irq(&port->lock);
		ops->stop_tx(port, 0);
		ops->set_mctrl(port, 0);
		ops->stop_rx(port);
		spin_unlock_irq(&port->lock);

		/*
		 * Wait for the transmitter to empty.
		 */
		while (!ops->tx_empty(port)) {
			msleep(10);
		}

		ops->shutdown(port);
	}

	/*
	 * Disable the console device before suspending.
	 */
	if (uart_console(port))
		console_stop(port->cons);

	uart_change_pm(state, 3);

	up(&state->sem);

	return 0;
}

int uart_resume_port(struct uart_driver *drv, struct uart_port *port)
{
	struct uart_state *state = drv->state + port->line;

	down(&state->sem);

	uart_change_pm(state, 0);

	/*
	 * Re-enable the console device after suspending.
	 */
	if (uart_console(port)) {
		struct termios termios;

		/*
		 * First try to use the console cflag setting.
		 */
		memset(&termios, 0, sizeof(struct termios));
		termios.c_cflag = port->cons->cflag;

		/*
		 * If that's unset, use the tty termios setting.
		 */
		if (state->info && state->info->tty && termios.c_cflag == 0)
			termios = *state->info->tty->termios;

		port->ops->set_termios(port, &termios, NULL);
		console_start(port->cons);
	}

	if (state->info && state->info->flags & UIF_INITIALIZED) {
		struct uart_ops *ops = port->ops;

		ops->set_mctrl(port, 0);
		ops->startup(port);
		uart_change_speed(state, NULL);
		spin_lock_irq(&port->lock);
		ops->set_mctrl(port, port->mctrl);
		ops->start_tx(port, 0);
		spin_unlock_irq(&port->lock);
	}

	up(&state->sem);

	return 0;
}

static inline void
uart_report_port(struct uart_driver *drv, struct uart_port *port)
{
	printk("%s%d", drv->dev_name, port->line);
	printk(" at ");
	switch (port->iotype) {
	case UPIO_PORT:
		printk("I/O 0x%x", port->iobase);
		break;
	case UPIO_HUB6:
		printk("I/O 0x%x offset 0x%x", port->iobase, port->hub6);
		break;
	case UPIO_MEM:
	case UPIO_MEM32:
		printk("MMIO 0x%lx", port->mapbase);
		break;
	}
	printk(" (irq = %d) is a %s\n", port->irq, uart_type(port));
}

static void
uart_configure_port(struct uart_driver *drv, struct uart_state *state,
		    struct uart_port *port)
{
	unsigned int flags;

	/*
	 * If there isn't a port here, don't do anything further.
	 */
	if (!port->iobase && !port->mapbase && !port->membase)
		return;

	/*
	 * Now do the auto configuration stuff. Note that config_port
	 * is expected to claim the resources and map the port for us.
	 */
	flags = UART_CONFIG_TYPE;
	if (port->flags & UPF_AUTO_IRQ)
		flags |= UART_CONFIG_IRQ;
```

```
        if (port->flags & UPF_BOOT_AUTOCONF) {
                port->type = PORT_UNKNOWN;
                port->ops->config_port(port, flags);
        }

        if (port->type != PORT_UNKNOWN) {
                unsigned long flags;

                uart_report_port(drv, port);

                /*
                 * Ensure that the modem control lines are de-activated.
                 * We probably don't need a spinlock around this, but
                 */
                spin_lock_irqsave(&port->lock, flags);
                port->ops->set_mctrl(port, 0);
                spin_unlock_irqrestore(&port->lock, flags);

                /*
                 * Power down all ports by default, except the
                 * console if we have one.
                 */
                if (!uart_console(port))
                        uart_change_pm(state, 3);
        }
}

/*
 * This reverses the effects of uart_configure_port, hanging up the
 * port before removal.
 */
static void
uart_unconfigure_port(struct uart_driver *drv, struct uart_state *state)
{
        struct uart_port *port = state->port;
        struct uart_info *info = state->info;

        if (info && info->tty)
                tty_vhangup(info->tty);

        down(&state->sem);

        state->info = NULL;

        /*
         * Free the port IO and memory resources, if any.
         */
        if (port->type != PORT_UNKNOWN)
                port->ops->release_port(port);

        /*
         * Indicate that there isn't a port here anymore.
         */
        port->type = PORT_UNKNOWN;

        /*
         * Kill the tasklet, and free resources.
         */
        if (info) {
                tasklet_kill(&info->tlet);
                kfree(info);
        }
```

```
		up(&state->sem);
}

static struct tty_operations uart_ops = {
		.open			= uart_open,
		.close			= uart_close,
		.write			= uart_write,
		.put_char		= uart_put_char,
		.flush_chars		= uart_flush_chars,
		.write_room		= uart_write_room,
		.chars_in_buffer= uart_chars_in_buffer,
		.flush_buffer		= uart_flush_buffer,
		.ioctl			= uart_ioctl,
		.throttle		= uart_throttle,
		.unthrottle		= uart_unthrottle,
		.send_xchar		= uart_send_xchar,
		.set_termios		= uart_set_termios,
		.stop			= uart_stop,
		.start			= uart_start,
		.hangup			= uart_hangup,
		.break_ctl		= uart_break_ctl,
		.wait_until_sent= uart_wait_until_sent,
#ifdef CONFIG_PROC_FS
		.read_proc		= uart_read_proc,
#endif
		.tiocmget		= uart_tiocmget,
		.tiocmset		= uart_tiocmset,
};

/**
 *		uart_register_driver - register a driver with the uart core layer
 *		@drv: low level driver structure
 *
 *		Register a uart driver with the core driver. We in turn register
 *		with the tty layer, and initialise the core driver per-port state.
 *
 *		We have a proc file in /proc/tty/driver which is named after the
 *		normal driver.
 *
 *		drv->port should be NULL, and the per-port structures should be
 *		registered using uart_add_one_port after this call has succeeded.
 */
int uart_register_driver(struct uart_driver *drv)
{
		struct tty_driver *normal = NULL;
		int i, retval;

		BUG_ON(drv->state);

		/*
		 * Maybe we should be using a slab cache for this, especially if
		 * we have a large number of ports to handle.
		 */
		drv->state = kmalloc(sizeof(struct uart_state) * drv->nr, GFP_KERNEL);
		retval = -ENOMEM;
		if (!drv->state)
				goto out;

		memset(drv->state, 0, sizeof(struct uart_state) * drv->nr);

		normal = alloc_tty_driver(drv->nr);
		if (!normal)
				goto out;
	drv->tty_driver = normal;

	normal->owner                = drv->owner;
	normal->driver_name          = drv->driver_name;
	normal->devfs_name           = drv->devfs_name;
	normal->name                 = drv->dev_name;
	normal->major                = drv->major;
	normal->minor_start          = drv->minor;
	normal->type                 = TTY_DRIVER_TYPE_SERIAL;
	normal->subtype              = SERIAL_TYPE_NORMAL;
	normal->init_termios         = tty_std_termios;
	normal->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
	normal->flags                = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS;
	normal->driver_state         = drv;
	tty_set_operations(normal, &uart_ops);

	/*
	 * Initialise the UART state(s).
	 */
	for (i = 0; i < drv->nr; i++) {
		struct uart_state *state = drv->state + i;

		state->close_delay       = 500;    /* .5 seconds */
		state->closing_wait      = 30000;  /* 30 seconds */

		init_MUTEX(&state->sem);
	}

	retval = tty_register_driver(normal);
 out:
	if (retval < 0) {
		put_tty_driver(normal);
		kfree(drv->state);
	}
	return retval;
}

/**
 *	uart_unregister_driver - remove a driver from the uart core layer
 *	@drv: low level driver structure
 *
 *	Remove all references to a driver from the core driver. The low
 *	level driver must have removed all its ports via the
 *	uart_remove_one_port() if it registered them with uart_add_one_port().
 *	(ie, drv->port == NULL)
 */
void uart_unregister_driver(struct uart_driver *drv)
{
	struct tty_driver *p = drv->tty_driver;
	tty_unregister_driver(p);
	put_tty_driver(p);
	kfree(drv->state);
	drv->tty_driver = NULL;
}

struct tty_driver *uart_console_device(struct console *co, int *index)
{
	struct uart_driver *p = co->data;
	*index = co->index;
	return p->tty_driver;
}
```

```
/**
 *      uart_add_one_port - attach a driver-defined port structure
 *      @drv: pointer to the uart low level driver structure for this port
 *      @port: uart port structure to use for this port.
 *
 *      This allows the driver to register its own uart_port structure
 *      with the core driver. The main purpose is to allow the low
 *      level uart drivers to expand uart_port, rather than having yet
 *      more levels of structures.
 */
int uart_add_one_port(struct uart_driver *drv, struct uart_port *port)
{
        struct uart_state *state;
        int ret = 0;

        BUG_ON(in_interrupt());

        if (port->line >= drv->nr)
                return -EINVAL;

        state = drv->state + port->line;

        down(&port_sem);
        if (state->port) {
                ret = -EINVAL;
                goto out;
        }

        state->port = port;

        spin_lock_init(&port->lock);
        port->cons = drv->cons;
        port->info = state->info;

        uart_configure_port(drv, state, port);

        /*
         * Register the port whether it's detected or not. This allows
         * setserial to be used to alter this ports parameters.
         */
        tty_register_device(drv->tty_driver, port->line, port->dev);

        /*
         * If this driver supports console, and it hasn't been
         * successfully registered yet, try to re-register it.
         * It may be that the port was not available.
         */
        if (port->type != PORT_UNKNOWN &&
            port->cons && !(port->cons->flags & CON_ENABLED))
                register_console(port->cons);

 out:
        up(&port_sem);

        return ret;
}

/**
 *      uart_remove_one_port - detach a driver defined port structure
 *      @drv: pointer to the uart low level driver structure for this port
 *      @port: uart port structure for this port
 *
 *      This unhooks (and hangs up) the specified port structure from the
 *              core driver. No further calls will be made to the low-level code
 *              for this port.
 */
int uart_remove_one_port(struct uart_driver *drv, struct uart_port *port)
{
        struct uart_state *state = drv->state + port->line;

        BUG_ON(in_interrupt());

        if (state->port != port)
                printk(KERN_ALERT "Removing wrong port: %p != %p\n",
                        state->port, port);

        down(&port_sem);

        /*
         * Remove the devices from devfs
         */
        tty_unregister_device(drv->tty_driver, port->line);

        uart_unconfigure_port(drv, state);
        state->port = NULL;
        up(&port_sem);

        return 0;
}

/*
 *              Are the two ports equivalent?
 */
static int uart_match_port(struct uart_port *port1, struct uart_port *port2)
{
        if (port1->iotype != port2->iotype)
                return 0;

        switch (port1->iotype) {
        case UPIO_PORT:
                return (port1->iobase == port2->iobase);
        case UPIO_HUB6:
                return (port1->iobase == port2->iobase) &&
                       (port1->hub6   == port2->hub6);
        case UPIO_MEM:
                return (port1->membase == port2->membase);
        }
        return 0;
}

/*
 *              Try to find an unused uart_state slot for a port.
 */
static struct uart_state *
uart_find_match_or_unused(struct uart_driver *drv, struct uart_port *port)
{
        int i;

        /*
         * First, find a port entry which matches. Note: if we do
         * find a matching entry, and it has a non-zero use count,
         * then we can't register the port.
         */
        for (i = 0; i < drv->nr; i++)
                if (uart_match_port(drv->state[i].port, port))
                        return &drv->state[i];
	/*
	 * We didn't find a matching entry, so look for the first
	 * free entry. We look for one which hasn't been previously
	 * used (indicated by zero iobase).
	 */
	for (i = 0; i < drv->nr; i++)
		if (drv->state[i].port->type == PORT_UNKNOWN &&
		    drv->state[i].port->iobase == 0 &&
		    drv->state[i].count == 0)
			return &drv->state[i];

	/*
	 * That also failed. Last resort is to find any currently
	 * entry which doesn't have a real port associated with it.
	 */
	for (i = 0; i < drv->nr; i++)
		if (drv->state[i].port->type == PORT_UNKNOWN &&
		    drv->state[i].count == 0)
			return &drv->state[i];

	return NULL;
}

/**
 *	uart_register_port: register uart settings with a port
 *	@drv: pointer to the uart low level driver structure for this port
 *	@port: uart port structure describing the port
 *
 *	Register UART settings with the specified low level driver. Detect
 *	the type of the port if UPF_BOOT_AUTOCONF is set, and detect the
 *	IRQ if UPF_AUTO_IRQ is set.
 *
 *	We try to pick the same port for the same IO base address, so that
 *	when a modem is plugged in, unplugged and plugged back in, it gets
 *	allocated the same port.
 *
 *	Returns negative error, or positive line number.
 */
int uart_register_port(struct uart_driver *drv, struct uart_port *port)
{
	struct uart_state *state;
	int ret;

	down(&port_sem);

	state = uart_find_match_or_unused(drv, port);

	if (state) {
		/*
		 * Ok, we've found a line that we can use.
		 *
		 * If we find a port that matches this one, and it appears
		 * to be in-use (even if it doesn't have a type) we shouldn't
		 * alter it underneath itself - the port may be open and
		 * trying to do useful work.
		 */
		if (uart_users(state) != 0) {
			ret = -EBUSY;
			goto out;
		}

		/*
```

```
			 * If the port is already initialised, don't touch it.
			 */
			if (state->port->type == PORT_UNKNOWN) {
				state->port->iobase     = port->iobase;
				state->port->membase    = port->membase;
				state->port->irq        = port->irq;
				state->port->uartclk    = port->uartclk;
				state->port->fifosize   = port->fifosize;
				state->port->regshift   = port->regshift;
				state->port->iotype     = port->iotype;
				state->port->flags      = port->flags;
				state->port->line       = state - drv->state;
				state->port->mapbase    = port->mapbase;

				uart_configure_port(drv, state, state->port);
			}

			ret = state->port->line;
		} else
			ret = -ENOSPC;
 out:
		up(&port_sem);
		return ret;
}

/**
 *	uart_unregister_port - de-allocate a port
 *	@drv: pointer to the uart low level driver structure for this port
 *	@line: line index previously returned from uart_register_port()
 *
 *	Hang up the specified line associated with the low level driver,
 *	and mark the port as unused.
 */
void uart_unregister_port(struct uart_driver *drv, int line)
{
		struct uart_state *state;

		if (line < 0 || line >= drv->nr) {
			printk(KERN_ERR "Attempt to unregister ");
			printk("%s%d", drv->dev_name, line);
			printk("\n");
			return;
		}

		state = drv->state + line;

		down(&port_sem);
		uart_unconfigure_port(drv, state);
		up(&port_sem);
}

EXPORT_SYMBOL(uart_write_wakeup);
EXPORT_SYMBOL(uart_register_driver);
EXPORT_SYMBOL(uart_unregister_driver);
EXPORT_SYMBOL(uart_suspend_port);
EXPORT_SYMBOL(uart_resume_port);
EXPORT_SYMBOL(uart_register_port);
EXPORT_SYMBOL(uart_unregister_port);
EXPORT_SYMBOL(uart_add_one_port);
EXPORT_SYMBOL(uart_remove_one_port);

MODULE_DESCRIPTION("Serial driver core");
MODULE_LICENSE("GPL");
```