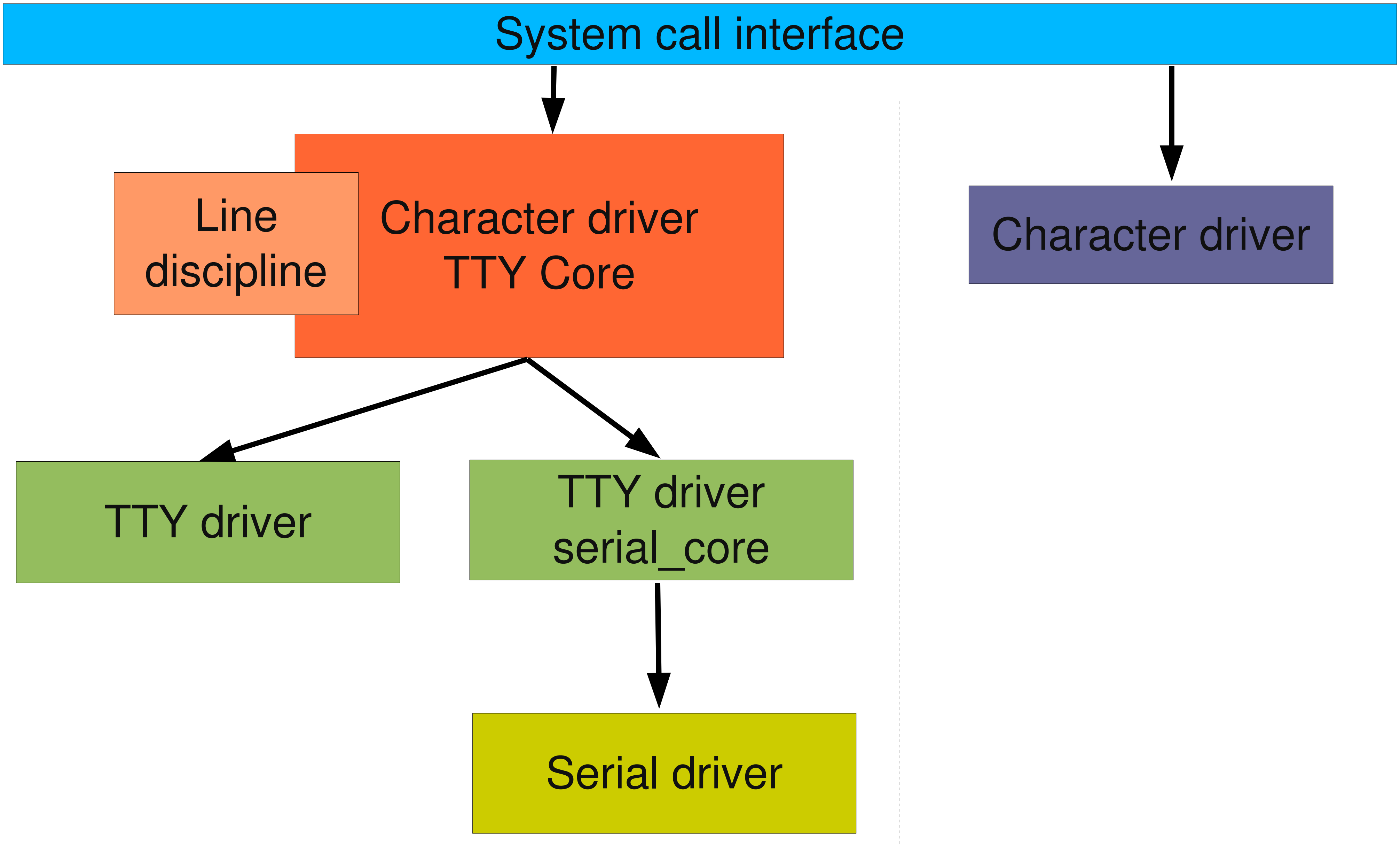
Serial drivers

Thomas Petazzoni
Free Electrons

Latest update: Dec 20, 2010,
Document sources, updates and translations:
http://free-electrons.com/docs/serial-drivers
Corrections, suggestions, contributions and translations are welcome!

Architecture (1)



Architecture (2)

- To be properly integrated in a Linux system, serial ports must be visible as TTY devices from userspace applications
- Therefore, the serial driver must be part of the kernel TTY subsystem
- Until 2.6, serial drivers were implemented directly behind the TTY core
  - A lot of complexity was involved
- Since 2.6, a specialized TTY driver, *serial_core*, eases the development of serial drivers
  - See `include/linux/serial_core.h` for the main definitions of the serial_core infrastructure
- The line discipline that cooks the data exchanged with the tty driver. For normal serial ports, N_TTY is used.

Data structures

- A data structure representing a driver : `uart_driver`
  - Single instance for each driver
  - `uart_register_driver()` and `uart_unregister_driver()`
- A data structure representing a port : `uart_port`
  - One instance for each port (several per driver are possible)
  - `uart_add_one_port()` and `uart_remove_one_port()`
- A data structure containing the pointers to the operations : `uart_ops`
  - Linked from `uart_port` through the `ops` field

uart_driver

- Usually
  - Defined statically in the driver
  - Registered in `module_init()`
  - Unregistered in `module_cleanup()`
- Contains
  - `owner`, usually set to `THIS_MODULE`
  - `driver_name`
  - `dev_name`, the device name prefix, usually “ttyS”
  - `major` and `minor`
    - Use `TTY_MAJOR` and `64` to get the normal numbers. But they might conflict with the 8250-reserved numbers
  - `nr`, the maximum number of ports
  - `cons`, pointer to the console device (covered later)

uart_driver code example (1)

```
static struct uart_driver atmel_uart = {
        .owner          = THIS_MODULE,
        .driver_name    = "atmel_serial",
        .dev_name       = ATMEL_DEVICENAME,
        .major          = SERIAL_ATMEL_MAJOR,
        .minor          = MINOR_START,
        .nr             = ATMEL_MAX_UART,
        .cons           = ATMEL_CONSOLE_DEVICE,
};

static struct platform_driver atmel_serial_driver = {
        .probe          = atmel_serial_probe,
        .remove         = __devexit_p(atmel_serial_remove),
        .suspend        = atmel_serial_suspend,
        .resume         = atmel_serial_resume,
        .driver         = {
                .name   = "atmel_usart",
                .owner  = THIS_MODULE,
        },
};
```

Example code from `drivers/serial/atmel_serial.c`

uart_driver code example (2)

```
static int __init atmel_serial_init(void)
{
        uart_register_driver(&atmel_uart);
        platform_driver_register(&atmel_serial_driver);
        return 0;
}

static void __exit atmel_serial_exit(void)
{
        platform_driver_unregister(&atmel_serial_driver);
        uart_unregister_driver(&atmel_uart);
}

module_init(atmel_serial_init);
module_exit(atmel_serial_exit);
```

Warning: error management removed!

uart_port

- Can be allocated statically or dynamically
- Usually registered at `probe()` time and unregistered at `remove()` time
- Most important fields
  - `iotype`, type of I/O access, usually `UPIO_MEM` for memory-mapped devices
  - `mapbase`, physical address of the registers
  - `irq`, the IRQ channel number
  - `membase`, the virtual address of the registers
  - `uartclk`, the clock rate
  - `ops`, pointer to the operations
  - `dev`, pointer to the device (platform_device or other)

uart_port code example (1)

```
static int __devinit atmel_serial_probe(struct platform_device *pdev)
{
        struct atmel_uart_port *port;

        port = &atmel_ports[pdev->id];
        port->backup_imr = 0;

        atmel_init_port(port, pdev);

        uart_add_one_port(&atmel_uart, &port->uart);

        platform_set_drvdata(pdev, port);

        return 0;
}

static int __devexit atmel_serial_remove(struct platform_device *pdev)
{
        struct uart_port *port = platform_get_drvdata(pdev);

        platform_set_drvdata(pdev, NULL);
        uart_remove_one_port(&atmel_uart, port);

        return 0;
}
```

uart_port code example (2)

```
static void __devinit atmel_init_port(struct atmel_uart_port *atmel_port,
                                      struct platform_device *pdev)
{
        struct uart_port *port = &atmel_port->uart;
        struct atmel_uart_data *data = pdev->dev.platform_data;

        port->iotype    = UPIO_MEM;
        port->flags     = UPF_BOOT_AUTOCONF;
        port->ops       = &atmel_pops;
        port->fifosize  = 1;
        port->line      = pdev->id;
        port->dev       = &pdev->dev;

        port->mapbase   = pdev->resource[0].start;
        port->irq       = pdev->resource[1].start;

        tasklet_init(&atmel_port->tasklet, atmel_tasklet_func,
                        (unsigned long)port);

[... see next page ...]
```

uart_port code example (3)

```
[... continued from previous page ...]

        if (data->regs)
                /* Already mapped by setup code */
                port->membase = data->regs;
        else {
                port->flags     |= UPF_IOREMAP;
                port->membase   = NULL;
        }

        /* for console, the clock could already be configured */
        if (!atmel_port->clk) {
                atmel_port->clk = clk_get(&pdev->dev, "usart");
                clk_enable(atmel_port->clk);
                port->uartclk = clk_get_rate(atmel_port->clk);
                clk_disable(atmel_port->clk);
                /* only enable clock when USART is in use */
        }
}
```

uart_ops

- Important operations
  - `tx_empty()`, tells whether the transmission FIFO is empty or not
  - `set_mctrl()` and `get_mctrl()`, allow to set and get the modem control parameters (RTS, DTR, LOOP, etc.)
  - `start_tx()` and `stop_tx()`, to start and stop the transmission
  - `stop_rx()`, to stop the reception
  - `startup()` and `shutdown()`, called when the port is opened/closed
  - `request_port()` and `release_port()`, request/release I/O or memory regions
  - `set_termios()`, change port parameters
- See the detailed description in `Documentation/serial/driver`

Implementing transmission

- The `start_tx()` method should start transmitting characters over the serial port
- The characters to transmit are stored in a circular buffer, implemented by a `struct uart_circ` structure. It contains
  - `buf[]`, the buffer of characters
  - `tail`, the index of the next character to transmit. After transmit, tail must be updated using `tail = tail & (UART_XMIT_SIZE - 1)`
- Utility functions on `uart_circ`
  - `uart_circ_empty()`, tells whether the circular buffer is empty
  - `uart_circ_chars_pending()`, returns the number of characters left to transmit
- From an `uart_port` pointer, this structure can be reached using `port->info->xmit`

Polled-mode transmission

```
foo_uart_putc(struct uart_port *port, unsigned char c) {
   while(__raw_readl(port->membase + UART_REG1) & UART_TX_FULL)
      cpu_relax();
   __raw_writel(c, port->membase + UART_REG2);
}

foo_uart_start_tx(struct uart_port *port) {
   struct circ_buf *xmit = &port->info->xmit;

   while (!uart_circ_empty(xmit)) {
      foo_uart_putc(port, xmit->buf[xmit->tail]);
      xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
      port->icount.tx++;
   }
}
```

Transmission with interrupts (1)

```
foo_uart_interrupt(int irq, void *dev_id) {
   [...]
   if (interrupt_cause & END_OF_TRANSMISSION)
      foo_uart_handle_transmit(port);
   [...]
}

foo_uart_start_tx(struct uart_port *port) {
   enable_interrupt_on_txrdy();
}
```

Transmission with interrupts (2)

```
foo_uart_handle_transmit(port) {

    struct circ_buf *xmit = &port->info->xmit;
    if (uart_circ_empty(xmit) || uart_tx_stopped(port)) {
        disable_interrupt_on_txrdy();
        return;
    }

    while (! uart_circ_empty(xmit)) {
        if (! (__raw_readl(port->membase + UART_REG1) &
                UART_TX_FULL))
            Break;
        __raw_writel(xmit->buf[xmit->tail],
                    port->membase + UART_REG2);
        xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);
        port->icount.tx++;
    }

    if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
        uart_write_wakeup(port);
}
```

Reception

- On reception, usually in an interrupt handler, the driver must
  - Increment `port->icount.rx`
  - Call `uart_handle_break()` if a BRK has been received, and if it returns TRUE, skip to the next character
  - If an error occurred, increment `port->icount.parity`, `port->icount.frame`, `port->icount.overrun` depending on the error type
  - Call `uart_handle_sysrq_char()` with the received character, and if it returns TRUE, skip to the next character
  - Call `uart_insert_char()` with the received character and a status
    - Status is `TTY_NORMAL` is everything is OK, or `TTY_BREAK`, `TTY_PARITY`, `TTY_FRAME` in case of error
  - Call `tty_flip_buffer_push()` to push data to the TTY later

Understanding Sysrq

- Part of the reception work is dedicated to handling Sysrq
  - Sysrq are special commands that can be sent to the kernel to make it reboot, unmount filesystems, dump the task state, nice real-time tasks, etc.
  - These commands are implemented at the lowest possible level so that even if the system is locked, you can recover it.
  - Through serial port: send a BRK character, send the character of the Sysrq command
  - See `Documentation/sysrq.txt`
- In the driver
  - `uart_handle_break()` saves the current time + 5 seconds in a variable
  - `uart_handle_sysrq_char()` will test if the current time is below the saved time, and if so, will trigger the execution of the Sysrq command

Reception code sample (1)

```
foo_receive_chars(struct uart_port *port) {
    int limit = 256;

    while (limit-- > 0) {
        status = __raw_readl(port->membase + REG_STATUS);
        ch = __raw_readl(port->membase + REG_DATA);
        flag = TTY_NORMAL;

        if (status & BREAK) {
            port->icount.break++;
            if (uart_handle_break(port))
                Continue;
        }
        else if (status & PARITY)
            port->icount.parity++;
        else if (status & FRAME)
            port->icount.frame++;
        else if (status & OVERRUN)
            port->icount.overrun++;

        [...]
```

Reception code sample (2)

```
        [...]
        status &= port->read_status_mask;

        if (status & BREAK)
            flag = TTY_BREAK;
        else if (status & PARITY)
            flag = TTY_PARITY;
        else if (status & FRAME)
            flag = TTY_FRAME;

        if (uart_handle_sysrq_char(port, ch))
            continue;

        uart_insert_char(port, status, OVERRUN, ch, flag);
    }

    spin_unlock(& port->lock);
    tty_flip_buffer_push(port->info->port.tty);
    spin_lock(& port->lock);
}
```

Modem control lines

- Set using the `set_mctrl()` operation
  - The `mctrl` argument can be a mask of `TIOCM_RTS` (request to send), `TIOCM_DTR` (Data Terminal Ready), `TIOCM_OUT1`, `TIOCM_OUT2`, `TIOCM_LOOP` (enable loop mode)
  - If a bit is set in `mctrl`, the signal must be driven active, if the bit is cleared, the signal must be driven inactive
- Status using the `get_mctrl()` operation
  - Must return read hardware status and return a combination of `TIOCM_CD` (Carrier Detect), `TIOCM_CTS` (Clear to Send), `TIOCM_DSR` (Data Set Ready) and `TIOCM_RI` (Ring Indicator)

set_mctrl() example

```
foo_set_mctrl(struct uart_port *uart, u_int mctrl) {
    unsigned int control = 0, mode = 0;

    if (mctrl & TIOCM_RTS)
        control |= ATMEL_US_RTSEN;
    else
        control |= ATMEL_US_RTSDIS;

    if (mctrl & TIOCM_DTS)
        control |= ATMEL_US_DTREN;
    else
        control |= ATMEL_US_DTRDIS;

    __raw_writel(port->membase + REG_CTRL, control);

    if (mctrl & TIOCM_LOOP)
        mode |= ATMEL_US_CHMODE_LOC_LOOP;
    else
        mode |= ATMEL_US_CHMODE_NORMAL;

    __raw_writel(port->membase + REG_MODE, mode);
}
```

get_mctrl() example

```
foo_get_mctrl(struct uart_port *uart, u_int mctrl) {
        unsigned int status, ret = 0;

        status = __raw_readl(port->membase + REG_STATUS);

        /*
         * The control signals are active low.
         */
        if (!(status & ATMEL_US_DCD))
                ret |= TIOCM_CD;
        if (!(status & ATMEL_US_CTS))
                ret |= TIOCM_CTS;
        if (!(status & ATMEL_US_DSR))
                ret |= TIOCM_DSR;
        if (!(status & ATMEL_US_RI))
                ret |= TIOCM_RI;

        return ret;
}
```

termios

- "The termios functions describe a general terminal interface that is provided to control asynchronous communications ports"
- A mechanism to control from userspace serial port parameters such as
  - Speed
  - Parity
  - Byte size
  - Stop bit
  - Hardware handshake
  - Etc.
- See termios(3) for details

set_termios()

- The `set_termios()` operation must
  - apply configuration changes according to the arguments
  - update `port->read_config_mask` and `port->ignore_config_mask` to indicate the events we are interested in receiving
- `static void set_termios(struct uart_port *port, struct ktermios *termios, struct ktermios *old)`
  - `port`, the port, `termios`, the new values and `old`, the old values
- Relevant `ktermios` structure fields are
  - `c_cflag` with word size, stop bits, parity, reception enable, CTS status change reporting, enable modem status change reporting
  - `c_iflag` with frame and parity errors reporting, break event reporting

set_termios() example (1)

```
static void atmel_set_termios(struct uart_port *port, struct ktermios *termios,
                              struct ktermios *old)
{
        unsigned long flags;
        unsigned int mode, imr, quot, baud;

        mode = __raw_readl(port->membase + REG_MODE);
        baud = uart_get_baud_rate(port, termios, old, 0, port->uartclk / 16);
        quot = uart_get_divisor(port, baud);

        switch (termios->c_cflag & CSIZE) {
        case CS5:
                mode |= ATMEL_US_CHRL_5;
                break;
        case CS6:
                mode |= ATMEL_US_CHRL_6;
                break;
        case CS7:
                mode |= ATMEL_US_CHRL_7;
                break;
        default:
                mode |= ATMEL_US_CHRL_8;
                break;
        }

         [...]
```

Read current configuration

Compute the mode modification for the byte size parameter

set_termios() example (2)

```
[...]

if (termios->c_cflag & CSTOPB)
        mode |= ATMEL_US_NBSTOP_2;

if (termios->c_cflag & PARENB) {
        /* Mark or Space parity */
        if (termios->c_cflag & CMSPAR) {
                if (termios->c_cflag & PARODD)
                        mode |= ATMEL_US_PAR_MARK;
                else
                        mode |= ATMEL_US_PAR_SPACE;
        } else if (termios->c_cflag & PARODD)
                mode |= ATMEL_US_PAR_ODD;
        else
                mode |= ATMEL_US_PAR_EVEN;
} else
        mode |= ATMEL_US_PAR_NONE;

if (termios->c_cflag & CRTSCTS)
        mode |= ATMEL_US_USMODE_HWHS;
else
        mode |= ATMEL_US_USMODE_NORMAL;

[...]
```

Compute the mode modification for
- the stop bit
- Parity
- CTS reporting

set_termios() example (3)

Compute the read_status_mask and ignore_status_mask according to the events we're interested in. These values are used in the interrupt handler.

```
 [...]

port->read_status_mask = ATMEL_US_OVRE;
if (termios->c_iflag & INPCK)
        port->read_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & (BRKINT | PARMRK))
        port->read_status_mask |= ATMEL_US_RXBRK;

port->ignore_status_mask = 0;
if (termios->c_iflag & IGNPAR)
        port->ignore_status_mask |= (ATMEL_US_FRAME | ATMEL_US_PARE);
if (termios->c_iflag & IGNBRK) {
        port->ignore_status_mask |= ATMEL_US_RXBRK;
        if (termios->c_iflag & IGNPAR)
                port->ignore_status_mask |= ATMEL_US_OVRE;
}

uart_update_timeout(port, termios->c_cflag, baud);

 [...]
```

The serial_core maintains a timeout that corresponds to the duration it takes to send the full transmit FIFO. This timeout has to be updated.

set_termios() example (4)

Finally, apply the mode and baud rate modifications. Interrupts, transmission and reception are disabled when the modifications are made.

```
        [...]

        /* Save and disable interupts */
       imr = UART_GET_IMR(port);
       UART_PUT_IDR(port, -1);

       /* disable receiver and transmitter */
       UART_PUT_CR(port, ATMEL_US_TXDIS | ATMEL_US_RXDIS);

       /* set the parity, stop bits and data size */
       UART_PUT_MR(port, mode);

       /* set the baud rate */
       UART_PUT_BRGR(port, quot);
       UART_PUT_CR(port, ATMEL_US_RSTSTA | ATMEL_US_RSTRX);
       UART_PUT_CR(port, ATMEL_US_TXEN | ATMEL_US_RXEN);

       /* restore interrupts */
       UART_PUT_IER(port, imr);

       /* CTS flow-control and modem-status interrupts */
       if (UART_ENABLE_MS(port, termios->c_cflag))
               port->ops->enable_ms(port);
}
```

Console

- To allows early boot messages to be printed, the kernel provides a separate but related facility: console
  - This console can be enabled using the `console=` kernel argument
- The driver developer must
  - Implement a `console_write()` operation, called to print characters on the console
  - Implement a `console_setup()` operation, called to parse the `console=` argument
  - Declare a `struct console` structure
  - Register the console using a `console_initcall()` function

Console: registration

```
static struct console serial_txx9_console = {
        .name           = TXX9_TTY_NAME,
        .write          = serial_txx9_console_write,
        .device         = uart_console_device,
        .setup          = serial_txx9_console_setup,
        .flags          = CON_PRINTBUFFER,
        .index          = -1,
        .data           = &serial_txx9_reg,
};

static int __init serial_txx9_console_init(void)
{
        register_console(&serial_txx9_console);
        return 0;
}
console_initcall(serial_txx9_console_init);
```

Helper function from the serial_core layer

Ask for the kernel messages buffered during boot to be printed to the console when activated

This will make sure the function is called early during the boot process.

start_kernel() calls console_init() that calls our function

Console : setup

```
static int __init serial_txx9_console_setup(struct console *co, char *options)
{
        struct uart_port *port;
        struct uart_txx9_port *up;
        int baud = 9600;
        int bits = 8;
        int parity = 'n';
        int flow = 'n';

        if (co->index >= UART_NR)
                co->index = 0;
        up = &serial_txx9_ports[co->index];
        port = &up->port;
        if (!port->ops)
                return -ENODEV;

        serial_txx9_initialize(&up->port);

        if (options)
                uart_parse_options(options, &baud, &parity, &bits, &flow);

        return uart_set_options(port, co, baud, parity, bits, flow);
}
```

Function shared with the normal serial driver

Helper function from serial_core that parses the console= string

Helper function from serial_core that calls the ->set_termios() operation with the proper arguments to configure the port

Console : write

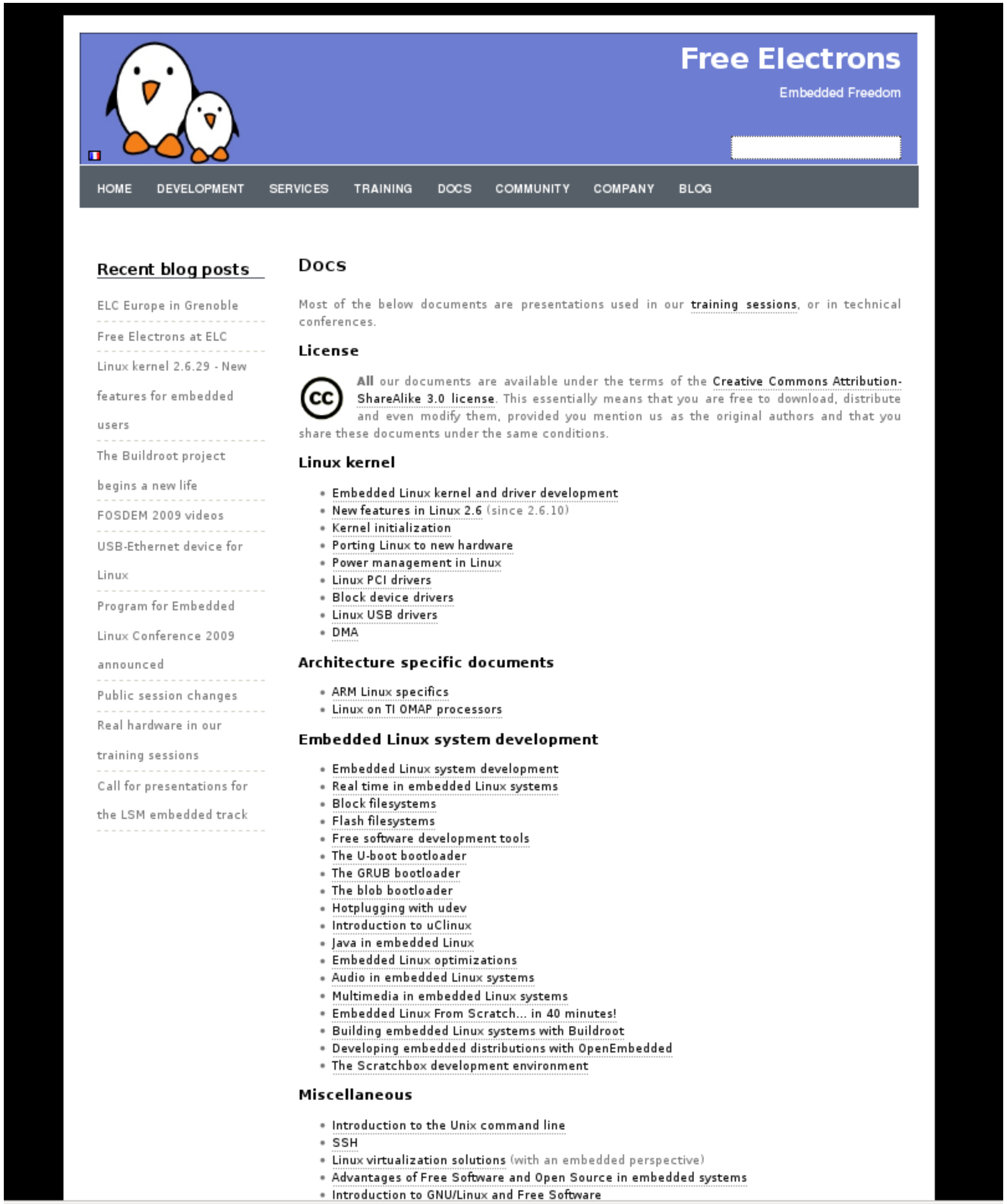
```
static void serial_txx9_console_putchar(struct uart_port *port, int ch)
{
        struct uart_txx9_port *up = (struct uart_txx9_port *)port;
        wait_for_xmitr(up);
        sio_out(up, TXX9_SITFIFO, ch);
}
static void serial_txx9_console_write( struct console *co,
                                       const char *s, unsigned int count)
{
        struct uart_txx9_port *up = &serial_txx9_ports[co->index];
        unsigned int ier, flcr;

         /* Disable interrupts
        ier = sio_in(up, TXX9_SIDICR);
        sio_out(up, TXX9_SIDICR, 0);

         /* Disable flow control */
        flcr = sio_in(up, TXX9_SIFLCR);
        if (!(up->port.flags & UPF_CONS_FLOW) && (flcr & TXX9_SIFLCR_TES))
                sio_out(up, TXX9_SIFLCR, flcr & ~TXX9_SIFLCR_TES);

        uart_console_write(&up->port, s, count, serial_txx9_console_putchar);

         /* Re-enable interrupts */
        wait_for_xmitr(up);
        sio_out(up, TXX9_SIFLCR, flcr);
        sio_out(up, TXX9_SIDICR, ier);
}
```

Busy-wait for transmitter ready and output a single character.

Helper function from *serial_core* that repeatedly calls the given `putchar()` callback

- Improve the character driver of the previous labs to make it a real serial driver

Related documents



All our technical presentations
on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs