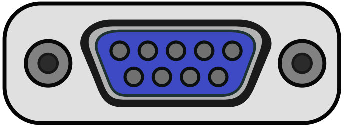
Linux serial drivers

This file is an old chapter of Free Electrons' embedded Linux kernel and driver development training materials (http://free-electrons.com/training/kernel/), which has been removed and is no longer maintained.

PDF version and sources are available on http://free-electrons.com/doc/legacy/serial-drivers/



Source: http://openclipart.org

Rights to copy

© Copyright 2004-2017, Free Electrons
License: Creative Commons Attribution - Share Alike 3.0
`http://creativecommons.org/licenses/by-sa/3.0/legalcode`

You are free:

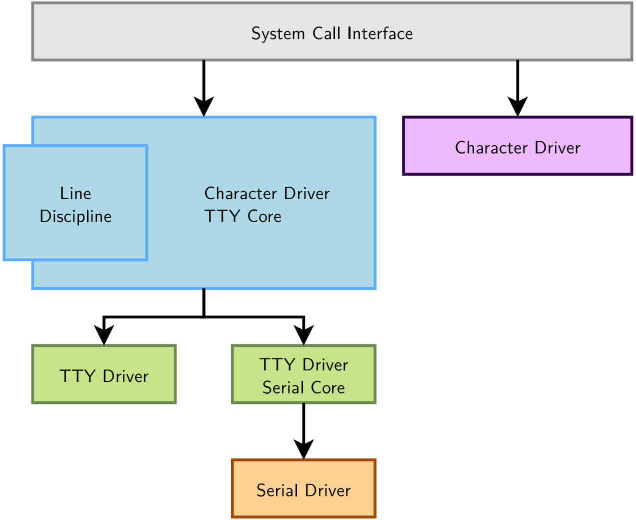
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

- **Attribution**. You must give the original author credit.
- **Share Alike**. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

Architecture (1)



Architecture (2)

- To be properly integrated in a Linux system, serial ports must be visible as TTY devices from user space 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)

```c
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 = 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)
{
    /* Warning: Error management removed */
    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);
```

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 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 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 atmel_init_port(
    struct atmel_uart_port *atmel_port,
    struct platform_device *pdev)
{
    struct uart_port *port = &atmelt_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);
```

uart_port Code Example (3)

```
    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->state->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->state->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->state->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 layer

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->state->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 communication ports*
- A mechanism to control from user space 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);
    /* Read current configuration */
    quot = uart_get_divisor(port, baud);

    /* Compute the mode modification for the byte size parameter */
    switch (termios->c_cflag & CSIZE) {
    case CS5:
        mode |= ATMEL_US_CHRL_5;
        break;
    case CS6:
        mode |= ATMEL_US_CHRL_6;
        break;
    [...]
    default:
        mode |= ATMEL_US_CHRL_8;
        break
```

set_termios() example (2)

```
/* Compute the mode modification for the stop bit */
if (termios->c_cflag & CSTOPB)
    mode |= ATMEL_US_NBSTOP_2;

/* Compute the mode modification for parity */
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;

/* Compute the mode modification for CTS reporting */
if (termios->c_cflag & CRTSCTS)
    mode |= ATMEL_US_USMODE_HWHS;
else
```

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;
}
/* 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. */
uart_update_timeout(port, termios->c_cflag, baud);
```

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 interrupts */
    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,
    /* Helper function from the serial_core layer */
    .device = uart_console_device,
    .setup = serial_txx9_console_setup,
    /* Ask for the kernel messages buffered during
     * boot to be printed to the console when activated */
    .flags = CON_PRINTBUFFER,
    .index = -1,
    .data = &serial_txx9_reg,
};

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

/* This will make sure the function is called early during the boot process.
 * start_kernel() calls console_init() that calls our function */
console_initcall(serial_txx9_console_init);
```

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;

    /* Function shared with the normal serial driver */
    serial_txx9_initialize(&up->port);

    if (options)
        /* Helper function from serial_core that parses the console= string */
        uart_parse_options(options, &baud, &parity, &bits, &flow);

    /* Helper function from serial_core that calls the ->set_termios() */
    /* operation with the proper arguments to configure the port */
    return uart_set_options(port, co, baud, parity, bits, flow);
}
```

Console: Write

```
static void serial_txx9_console_putchar(struct uart_port *port, int ch)
{
^^Istruct uart_txx9_port *up = (struct uart_txx9_port *)port;
^^I/* Busy-wait for transmitter ready and output a single character. */
^^Iwait_for_xmitr(up);
^^Isio_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);

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