Low Level Serial API

This document is meant as a brief overview of some aspects of the new serial driver. It is not complete, any questions you have should be directed to <rmk@arm.linux.org.uk>

The reference implementation is contained within amba_pl011.c.

Low Level Serial Hardware Driver

The low level serial hardware driver is responsible for supplying port information (defined by uart_port) and a set of control methods (defined by uart_ops) to the core serial driver. The low level driver is also responsible for handling interrupts for the port, and providing any console support.

Console Support

The serial core provides a few helper functions. This includes identifing the correct port structure (via uart_get_console) and decoding command line arguments (uart_parse_options).

There is also a helper function (uart_write_console) which performs a character by character write, translating newlines to CRLF sequences. Driver writers are recommended to use this function rather than implementing their own version.

Locking

It is the responsibility of the low level hardware driver to perform the necessary locking using port->lock. There are some exceptions (which are described in the uart_ops listing below.)

There are three locks. A per-port spinlock, a per-port tmpbuf semaphore, and an overall semaphore.

From the core driver perspective, the port->lock locks the following data:

```
port->mctrl
port->icount
info->xmit.head (circ->head)
info->xmit.tail (circ->tail)
```

The low level driver is free to use this lock to provide any additional locking.

The core driver uses the info->tmpbuf_sem lock to prevent multi-threaded access to the info->tmpbuf bouncebuffer used for port writes.

The port_sem semaphore is used to protect against ports being added/ removed or reconfigured at inappropriate times. Since v2.6.27, this semaphore has been the 'mutex' member of the tty_port struct, and commonly referred to as the port mutex (or port->mutex).

```
uart_ops
--------

The uart_ops structure is the main interface between serial_core and the
hardware specific driver.  It contains all the methods to control the
hardware.

  tx_empty(port)
        This function tests whether the transmitter fifo and shifter
        for the port described by 'port' is empty.  If it is empty,
        this function should return TIOCSER_TEMT, otherwise return 0.
        If the port does not support this operation, then it should
        return TIOCSER_TEMT.

        Locking: none.
        Interrupts: caller dependent.
        This call must not sleep

  set_mctrl(port, mctrl)
        This function sets the modem control lines for port described
        by 'port' to the state described by mctrl.  The relevant bits
        of mctrl are:
                - TIOCM_RTS     RTS signal.
                - TIOCM_DTR     DTR signal.
                - TIOCM_OUT1    OUT1 signal.
                - TIOCM_OUT2    OUT2 signal.
                - TIOCM_LOOP    Set the port into loopback mode.
        If the appropriate bit is set, the signal should be driven
        active.  If the bit is clear, the signal should be driven
        inactive.

        Locking: port->lock taken.
        Interrupts: locally disabled.
        This call must not sleep

  get_mctrl(port)
        Returns the current state of modem control inputs.  The state
        of the outputs should not be returned, since the core keeps
        track of their state.  The state information should include:
                - TIOCM_CAR     state of DCD signal
                - TIOCM_CTS     state of CTS signal
                - TIOCM_DSR     state of DSR signal
                - TIOCM_RI      state of RI signal
        The bit is set if the signal is currently driven active.  If
        the port does not support CTS, DCD or DSR, the driver should
        indicate that the signal is permanently active.  If RI is
        not available, the signal should not be indicated as active.

        Locking: port->lock taken.
        Interrupts: locally disabled.
        This call must not sleep

  stop_tx(port)
        Stop transmitting characters.  This might be due to the CTS
        line becoming inactive or the tty layer indicating we want
        to stop transmission due to an XOFF character.

        The driver should stop transmitting characters as soon as
        possible.

        Locking: port->lock taken.
        Interrupts: locally disabled.
        This call must not sleep
```

```
    start_tx(port)
          Start transmitting characters.

          Locking: port->lock taken.
          Interrupts: locally disabled.
          This call must not sleep

    send_xchar(port,ch)
          Transmit a high priority character, even if the port is stopped.
          This is used to implement XON/XOFF flow control and tcflow().  If
          the serial driver does not implement this function, the tty core
          will append the character to the circular buffer and then call
          start_tx() / stop_tx() to flush the data out.

          Do not transmit if ch == '\0' (__DISABLED_CHAR).

          Locking: none.
          Interrupts: caller dependent.

    stop_rx(port)
          Stop receiving characters; the port is in the process of
          being closed.

          Locking: port->lock taken.
          Interrupts: locally disabled.
          This call must not sleep

    enable_ms(port)
          Enable the modem status interrupts.

          This method may be called multiple times.  Modem status
          interrupts should be disabled when the shutdown method is
          called.

          Locking: port->lock taken.
          Interrupts: locally disabled.
          This call must not sleep

    break_ctl(port,ctl)
          Control the transmission of a break signal.  If ctl is
          nonzero, the break signal should be transmitted.  The signal
          should be terminated when another call is made with a zero
          ctl.

          Locking: none.
          Interrupts: caller dependent.
          This call must not sleep

    startup(port)
          Grab any interrupt resources and initialise any low level driver
          state.  Enable the port for reception.  It should not activate
          RTS nor DTR; this will be done via a separate call to set_mctrl.

          This method will only be called when the port is initially opened.

          Locking: port_sem taken.
          Interrupts: globally disabled.

    shutdown(port)
          Disable the port, disable any break condition that may be in
          effect, and free any interrupt resources.  It should not disable
          RTS nor DTR; this will have already been done via a separate
          call to set_mctrl.
```

```
        Drivers must not access port->info once this call has completed.

        This method will only be called when there are no more users of
        this port.

        Locking: port_sem taken.
        Interrupts: caller dependent.

    flush_buffer(port)
        Flush any write buffers, reset any DMA state and stop any
        ongoing DMA transfers.

        This will be called whenever the port->info->xmit circular
        buffer is cleared.

        Locking: port->lock taken.
        Interrupts: locally disabled.
        This call must not sleep

    set_termios(port,termios,oldtermios)
        Change the port parameters, including word length, parity, stop
        bits.  Update read_status_mask and ignore_status_mask to indicate
        the types of events we are interested in receiving.  Relevant
        termios->c_cflag bits are:
                CSIZE   - word size
                CSTOPB  - 2 stop bits
                PARENB  - parity enable
                PARODD  - odd parity (when PARENB is in force)
                CREAD   - enable reception of characters (if not set,
                          still receive characters from the port, but
                          throw them away.
                CRTSCTS - if set, enable CTS status change reporting
                CLOCAL  - if not set, enable modem status change
                          reporting.
        Relevant termios->c_iflag bits are:
                INPCK   - enable frame and parity error events to be
                          passed to the TTY layer.
                BRKINT
                PARMRK  - both of these enable break events to be
                          passed to the TTY layer.

                IGNPAR  - ignore parity and framing errors
                IGNBRK  - ignore break errors,  If IGNPAR is also
                          set, ignore overrun errors as well.
        The interaction of the iflag bits is as follows (parity error
        given as an example):
        Parity error    INPCK   IGNPAR
        n/a             0       n/a     character received, marked as
                                        TTY_NORMAL
        None            1       n/a     character received, marked as
                                        TTY_NORMAL
        Yes             1       0       character received, marked as
                                        TTY_PARITY
        Yes             1       1       character discarded

        Other flags may be used (eg, xon/xoff characters) if your
        hardware supports hardware "soft" flow control.

        Locking: caller holds port->mutex
        Interrupts: caller dependent.
        This call must not sleep

    pm(port,state,oldstate)
        Perform any power management related activities on the specified
        port.  State indicates the new state (defined by
```

```
      enum uart_pm_state), oldstate indicates the previous state.

      This function should not be used to grab any resources.

      This will be called when the port is initially opened and finally
      closed, except when the port is also the system console.  This
      will occur even if CONFIG_PM is not set.

      Locking: none.
      Interrupts: caller dependent.

  type(port)
      Return a pointer to a string constant describing the specified
      port, or return NULL, in which case the string 'unknown' is
      substituted.

      Locking: none.
      Interrupts: caller dependent.

  release_port(port)
      Release any memory and IO region resources currently in use by
      the port.

      Locking: none.
      Interrupts: caller dependent.

  request_port(port)
      Request any memory and IO region resources required by the port.
      If any fail, no resources should be registered when this function
      returns, and it should return -EBUSY on failure.

      Locking: none.
      Interrupts: caller dependent.

  config_port(port,type)
      Perform any autoconfiguration steps required for the port.  `type`
      contains a bit mask of the required configuration.  UART_CONFIG_TYPE
      indicates that the port requires detection and identification.
      port->type should be set to the type found, or PORT_UNKNOWN if
      no port was detected.

      UART_CONFIG_IRQ indicates autoconfiguration of the interrupt signal,
      which should be probed using standard kernel autoprobing techniques.
      This is not necessary on platforms where ports have interrupts
      internally hard wired (eg, system on a chip implementations).

      Locking: none.
      Interrupts: caller dependent.

  verify_port(port,serinfo)
      Verify the new serial port information contained within serinfo is
      suitable for this port type.

      Locking: none.
      Interrupts: caller dependent.

  ioctl(port,cmd,arg)
      Perform any port specific IOCTLs.  IOCTL commands must be defined
      using the standard numbering system found in <asm/ioctl.h>

      Locking: none.
      Interrupts: caller dependent.

  poll_init(port)
      Called by kgdb to perform the minimal hardware initialization needed
```

```
        to support poll_put_char() and poll_get_char().  Unlike ->startup()
        this should not request interrupts.

        Locking: tty_mutex and tty_port->mutex taken.
        Interrupts: n/a.

  poll_put_char(port,ch)
        Called by kgdb to write a single character directly to the serial
        port.  It can and should block until there is space in the TX FIFO.

        Locking: none.
        Interrupts: caller dependent.
        This call must not sleep

  poll_get_char(port)
        Called by kgdb to read a single character directly from the serial
        port.  If data is available, it should be returned; otherwise
        the function should return NO_POLL_CHAR immediately.

        Locking: none.
        Interrupts: caller dependent.
        This call must not sleep

Other functions
---------------

uart_update_timeout(port,cflag,baud)
        Update the FIFO drain timeout, port->timeout, according to the
        number of bits, parity, stop bits and baud rate.

        Locking: caller is expected to take port->lock
        Interrupts: n/a

uart_get_baud_rate(port,termios,old,min,max)
        Return the numeric baud rate for the specified termios, taking
        account of the special 38400 baud "kludge".  The B0 baud rate
        is mapped to 9600 baud.

        If the baud rate is not within min..max, then if old is non-NULL,
        the original baud rate will be tried.  If that exceeds the
        min..max constraint, 9600 baud will be returned.  termios will
        be updated to the baud rate in use.

        Note: min..max must always allow 9600 baud to be selected.

        Locking: caller dependent.
        Interrupts: n/a

uart_get_divisor(port,baud)
        Return the divsor (baud_base / baud) for the specified baud
        rate, appropriately rounded.

        If 38400 baud and custom divisor is selected, return the
        custom divisor instead.

        Locking: caller dependent.
        Interrupts: n/a

uart_match_port(port1,port2)
        This utility function can be used to determine whether two
        uart_port structures describe the same port.

        Locking: n/a
        Interrupts: n/a
```

```
uart_write_wakeup(port)
        A driver is expected to call this function when the number of
        characters in the transmit buffer have dropped below a threshold.

        Locking: port->lock should be held.
        Interrupts: n/a

uart_register_driver(drv)
        Register a uart driver with the core driver.  We in turn register
        with the tty layer, and initialise the core driver per-port state.

        drv->port should be NULL, and the per-port structures should be
        registered using uart_add_one_port after this call has succeeded.

        Locking: none
        Interrupts: enabled

uart_unregister_driver()
        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().

        Locking: none
        Interrupts: enabled

uart_suspend_port()

uart_resume_port()

uart_add_one_port()

uart_remove_one_port()

Other notes
-----------

It is intended some day to drop the 'unused' entries from uart_port, and
allow low level drivers to register their own individual uart_port's with
the core.  This will allow drivers to use uart_port as a pointer to a
structure containing both the uart_port entry with their own extensions,
thus:

        struct my_port {
                struct uart_port        port;
                int                     my_stuff;
        };

Modem control lines via GPIO
----------------------------

Some helpers are provided in order to set/get modem control lines via GPIO.

mctrl_gpio_init(dev, idx):
        This will get the {cts,rts,...}-gpios from device tree if they are
        present and request them, set direction etc, and return an
        allocated structure. devm_* functions are used, so there's no need
        to call mctrl_gpio_free().

mctrl_gpio_free(dev, gpios):
        This will free the requested gpios in mctrl_gpio_init().
        As devm_* function are used, there's generally no need to call
        this function.

mctrl_gpio_to_gpiod(gpios, gidx)
        This returns the gpio structure associated to the modem line index.
```

```
mctrl_gpio_set(gpios, mctrl):
        This will sets the gpios according to the mctrl state.

mctrl_gpio_get(gpios, mctrl):
        This will update mctrl with the gpios values.
```