Ajit Serial Driver Documentation

Harshal Kalyane

May 3, 2016

Contents

1 Introduction

Driver is software which is used to do hardware specific operation, and create abstraction between OS and Hardware.
There are mainly three types of driver in Linux:
1)Network Driver
2) Block Driver
3) Char driver
Network driver deals with networks packets and network hardware. Block driver deals with storage devices and file system. Char driver is mainly used for those devices which are not part of the above two categories.

2 Char driver

As name suggests, these types of drivers communicate with hardware by character (byte). Protocol of communication is device dependant. Serial char driver can be developed from scratch or from already built char driver. To reduce complexity and developing time Linux mainline kernel has tty driver. TTY driver gives standard abstract layer between OS and low level serial driver.

TTY core and TTY line discipline is provided by kernel itself. Developer needs to design only low level serial driver. Low level level driver has to do only hardware specific operation and need to fill data in TTY core data structure. Low level serial does not need to worry about user space interaction, it is handles by TTY core. Every driver has to do following things:

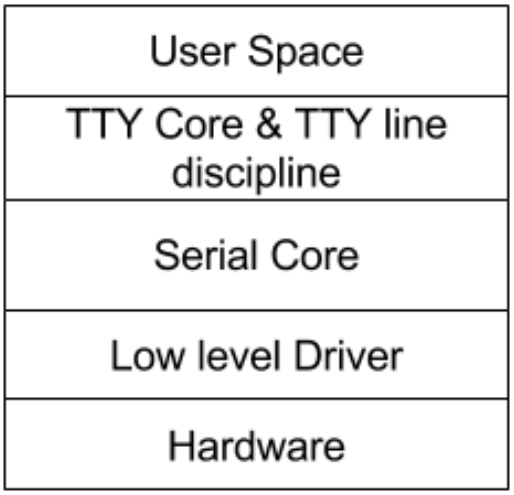

Figure 1: Block Diagram of Serial Driver

1)__init <function_name>: This get loaded on serial driver initialization. It registers itself and respective port in kernel for its hardware.
2) __exit <function_name>: This get called at time unloading of module or driver.

3 AJIT Serial Driver

This driver is written for Ajit serial device.
int __init Ajit_init(void) : This function is get called at time loading driver. module_init(Ajit_init) is used to mark a function to be used as the entry-point of a Linux device-driver. This function first register itself in kernel by calling this function uart_register_driver(&Ajit_serial_driver). It adds it's port in kernel for Ajit serial device.
module_exit(Ajit_exit): This function is get called at time unloading driver.It unregistere serial driver from kernel entry and frees resources by calling function uart_unregister_driver(&Ajit_serial_driver).
Ajit_serial_driver has structure of uart_driver. It has following fields.
struct **uart_driver** Ajit_serial_driver = {
.owner = THIS_MODULE,
.driver_name = AJIT_DRIVER_NAME,
.dev_name = AJIT_SERIAL_NAME,
.major = AJIT_SERIAL_MAJOR,
.minor = AJIT_SERIAL_MINORS,
.nr = AJIT_UART_NR,
#ifdef CONFIG_SERIAL_AJIT_CONSOLE
.cons = &Ajit_console,
#endif
};
.owner: this fields tells who is owner of this module.
.driver_name : name of the driver in his case "Ajit_serial".
.dev_name: string of this field is used to create dev file under /dev/ in this case, it is /dev/ttyS0.
.major: it is major no for driver. OS use this at time of system call.
.minor: The minor number is used by the kernel to determine exactly which device is being referred to.
.nr : the maximum number of ports in this case it is 1.
.cons: Default, it is NULL. For console it needs console structure.

3.1 Struct uart_port

```
static struct uart_port Ajit_port = {
        .fifosize=1,
        .flags    = UPF_BOOT_AUTOCONF,
        .iotype   = UPIO_MEM,
        .mapbase  = ADDR_SERIAL_CONTROL_REGISTER,
        .membase  = NULL,
        .ops      = &Ajit_serial_ops,
        .irq      = AJIT_SERIAL_IRQ,
};
```

This function stores information regarding serial port. It is used at time of registration of port. Required fields are shown as above for Ajit serial driver, remaining fields are default NULL. For more information please see serial_core.h.
.fifosize: Tx fifo size
.flags: This fields tells about serial port. UPF_BOOT_AUTOCONF means "The exact UART type is known and should not be probed".
.iotype: Tells about io access style.
.mapbase: This field contains base address of memory map location of serial device.
.membase: Initially it is NULL. It stores virtual address returned by ioremap() after memory mapping.
.ops: This fields store structure, which stores pointers to the different functions.
.irq: It stores IRQ number. In this case it is 12.

3.2 Struct uart_ops

```
static struct uart_ops Ajit_serial_ops = {
        .tx_empty       = Ajit_Tx_empty_locking,
        .set_mctrl      = Ajit_set_mctrl,
        .get_mctrl      = Ajit_get_mctrl,
        .stop_tx        = Ajit_stop_Tx,
        .start_tx       = Ajit_start_Tx,
        .stop_rx        = Ajit_stop_Rx,
        .enable_ms      = Ajit_enable_ms,
        .break_ctl      = Ajit_break_ctl,
        .startup        = Ajit_startup,
        .shutdown       = Ajit_shutdown,
        .set_termios    = Ajit_set_termios,
        .type           = Ajit_type,
        .release_port   = Ajit_release_port,
        .request_port   = Ajit_request_port,
        .config_port    = Ajit_config_port,
        .verify_port    = Ajit_verify_port,
```

```
};
```

This structure stores pointers to functions. It maps user space functions call to the driver's functions.

3.3 Struct console Ajit _ console

```
static struct console Ajit_console = {
        .name    = AJIT_SERIAL_NAME,
        .write   = Ajit_console_write,
        .device  = uart_console_device,
        .flags   = CON_PRINTBUFFER,
        .index   = -1,
        .data    = &Ajit_serial_driver,
};
```

3.4 I/O Functions

3.4.1 Struct Ajit_register_ops

```
struct Ajit_register_ops
{
        u8 (*in8)(void __iomem *addr);
        void (*out8)(u8 val, void __iomem *addr);
        u32 (*in32)(void __iomem *addr);
        void (*out32)(u32 val, void __iomem *addr);
};

static struct Ajit_register_ops Ajit_reg_ops =
{
        .in8    = Ajit_in8,
        .out8   = Ajit_out8,
        .in32   = Ajit_in32,
        .out32  = Ajit_out32,
};
```

This structure is used to point to the i/o functions.

```
//Routines for reads/writes to device registers
static u8 Ajit_in8(void __iomem *addr)
{
        return ioread8(addr);
}

static void Ajit_out8(u8 val, void __iomem *addr)
{
```

```
        iowrite8(val, addr);
}

static u32 Ajit_in32(void __iomem *addr)
{
        return ioread32be(addr);
}

static void Ajit_out32(u32 val, void __iomem *addr)
{
        iowrite32be(val, addr);
}
```

3.4.2 uart_out8

This function is used write 8 bit data to the device. It takes 8 value, offset and port as an input argument and uses port's private data structure to point i/o functions.

3.4.3 uart_in8

This function is used read 8 bit data from the device. It return 8 bit value and takes offset and port as an input argument.

3.4.4 uart_out32

This function is used write 32 bit data to the device. It takes 32 value, offset and port as an input argument and uses port's private data structure to point i/o functions.

3.4.5 uart_in32

This function is used read 32 bit data from the device. It return 32 bit value and takes offset and port as an input argument.

3.5 Tx Functions

3.5.1 Ajit_enable_Tx

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Set Tx enable by doing Bit wise OR of Return value of uart_in32 and MASK_TX_EN
- Write back this value to the device using uart_out32.

3.5.2 Ajit_stop_Tx

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Clear Tx enable by doing Bit wise AND of Return value of uart_in32 and not of MASK_TX_EN
- Write back this value to the device using uart_out32.

3.5.3 Ajit_Tx_empty

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Check full bit by doing Bit wise AND of Return value of uart_in32 and not of MASK_TX_FULL
- Return 0 if result of above step is true else 1.

3.5.4 Ajit_Tx_empty_locking

This function takes uart_port as input argument.

- Get spin lock by calling function spin_lock_irqsave.
- Call function Ajit_Tx_empty.
- Release spin lock by calling function spin_lock_irqsave.
- Return 0 if result of step 2 is 0 else 1.

3.5.5 Ajit_Tx_full

This function takes uart_port as input argument.

- Call function Ajit_Tx_empty.
- If it's Return value is 1 then return 0 else 1.

3.5.6 Ajit_send_Tx

This function takes 8 bit data and uart_port as input arguments. This is a blocking function.

- It waits till transmitter get empty.
- Write 8 bit data to device by calling function uart_out8.

3.5.7 Ajit_Tx_chars

Please see section Transmit Char From User Space to Device.

3.5.8 Ajit_start_Tx

It calls Ajit_Tx_chars.

3.6 Rx Functions

3.6.1 Ajit_enable_Rx

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Set Rx enable bit by doing Bit wise OR of Return value of uart_in32 and MASK_RX_EN
- Enable receive interrupt by doing Bit wise OR of result of above step and MASK_RX_INT_EN
- Write back this value to the device using uart_out32.

3.6.2 Ajit_stop_Rx

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Clear Rx enable bit by doing Bit wise AND of Return value of uart_in32 and Not of MASK_RX_EN
- Enable receive interrupt by doing Bit wise AND of result of above step and Not of MASK_RX_INT_EN
- Write back this value to the device using uart_out32.

3.6.3 Ajit_Rx_full

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Check full bit by doing Bit wise AND of Return value of uart_in32 and not of MASK_RX_FULL
- Return 0 if result of above step is 0 else 1.

3.6.4 Ajit_Rx_empty

This function takes uart_port as input argument.

- Read control register by passing offset and port structure to the uart_in32.
- Check full bit by doing Bit wise AND of Return value of uart_in32 and not of MASK_RX_FULL
- Return 0 if result of above step is 1 else 0.

3.6.5 Ajit_receive

This function takes uart_port as input argument.

- Call Ajit_Rx_empty(port) to check rx buffer of device is empty or not. Return 0 if it is empty.
- Increment rx count.
- Get 8 bit data from device and store it in local variable.
- Insert data from local variable to flip buffer.
- return 1.

3.6.6 Ajit_isr

Please see section Receive Char From Device to User Space.

4 AJIT Serial Device

4.1 Register Addresses

Control Register (32b) : 0x3200
Tx Register (8b) : 0x3210
Rx Register (8b) : 0x3220

4.2 Control Register

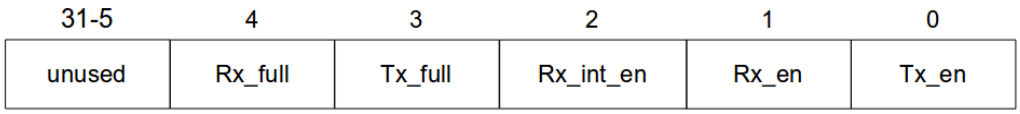

Figure 2: Control Register

4.3 Tx states

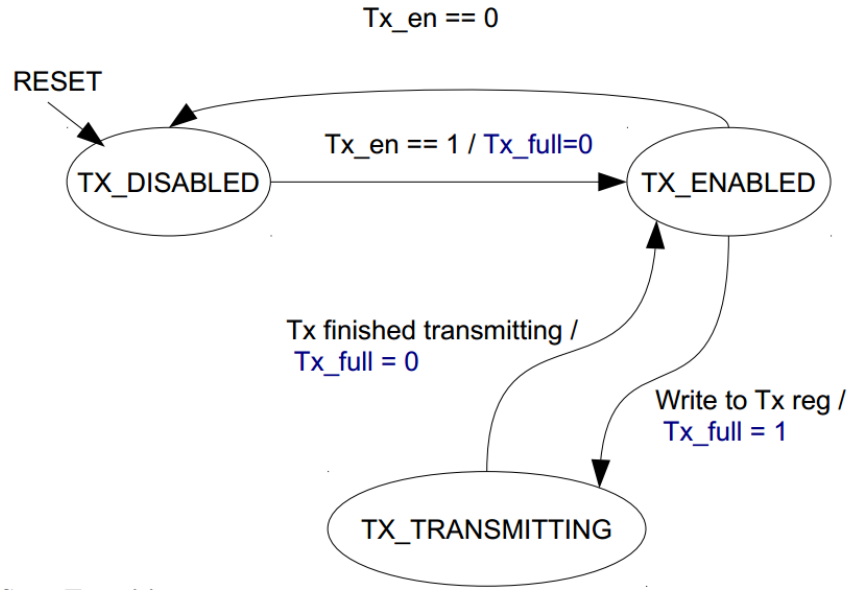

Figure 3: Tx state

4.4 Rx states

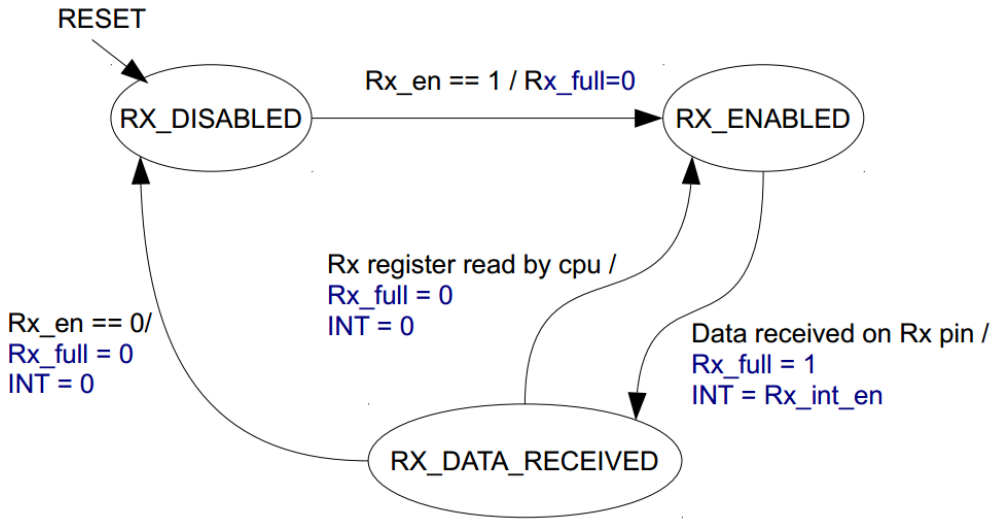

Figure 4: Rx state

5 Control & Data Flow In Ajit Serial Driver

5.1 Serial Driver Module Loading

At time of module loading following functions are executed:

- __init Ajit_init(void).
  - uart_register_driver(&Ajit_serial_driver))
  - uart_add_one_port(&Ajit_serial_driver, &Ajit_port)
  - If uart_add_one_port() fails then unregister serial driver by calling uart_unregister_driver(&Ajit_serial_driver)
  - end

5.2 Opening of Port

- Ajit_startup(struct uart_port *port)
  - Request irq of no mentioned in port->irq.
    Call Ajit_build_device_irq(NULL, port->irq).
    If fails return error else continue.
  - call request_irq(irq, Ajit_isr, IRQF_SHARED, AJIT_DRIVER_NAME, port) to request irq mentioned in port->irq.
    If fails return error else continue. For definition of irq please see Appendix
  - Ajit_enable_Tx(port) Enable Tx.
  - Ajit_enable_Rx(port) Enable Rx.
  - return 0.
- int Ajit_request_port(struct uart_port *port)
  - Request for memory to map device register in memory. it calls **request_mem_region** (port->mapbase, AJIT_REGION, AJIT_DRIVER_NAME)
    AJIT_REGION has value of base address of control register of serial device.
    AJIT_REGION has size of memory to be mapped.
  - map physical address to the virtual address and store it in port->mambase.
    It calls ioremap(port->mapbase, AJIT_REGION)
    If port->mambase is NULL then release memory and return -EBUSY
    else continue.

- map generic platform data pointer to the private data.
  port->private_data = &Ajit_reg_ops

5.3 Transmit Char From User Space to Device

- Driver calls Ajit_start_Tx(struct uart_port *port). This function calls int Ajit_Tx_chars(struct uart_port *port).
- creates pointer to point TTY core's circular buffer.
- It enables transmission by calling function Ajit_enable_Tx(port).
- If port->x_char is true then it sends data to device, increment tx count and clears port->x_char.
- return 0 if circular buffer is empty or serial device stopped transmitting.
- sends all 8bit data to the serial device buffer by calling function Ajit_send_Tx(xmit->buf[xmit->tail], port).
  Here xmit->tail points to the circular buffer of TTY core. It increment pointer by 1 at each iteration. Also it increments tx count.
- if still there is data in TTY core circular buffer less than WAKEUP_CHARS then it sends wakeup signal to those process which are sleeping on lock of port.
- if circular buffer empty then sends stop signal to serial Tx.
- return 1 as success.

5.4 Receive Char From Device to User Space

- Serial hardware generate interrupt when it has data.
- Receive interrupt occurs on interrupt no 12.
- This interrupt calls Ajit_isr(int irq, void *dev_id).
  - It disables all interrupt by calling Ajit_write_IRC_control_word(0x00).
  - It checks, is this interrupt generated by serial device or by any other device by calling Ajit_receive(port). Increment value of n at each iteration.
  - It enables interrupt by calling Ajit_write_IRC_control_word(0x01).
  - if n >1 then it returns IRQ_HANDLED else IRQ_NONE
  - if n >1 it flushes receive buffer.

5.5 Closing of Port

- At time closing it first calls void Ajit_release_port(struct uart_port *port). This function is used to release memory mapped region of serial device and unmapped Virtual Address from physical address.
- Then tty core calls Ajit_shutdown(struct uart_port *port)
  - It sends stop tx signal to serial device by using function Ajit_stop_Tx(port).
  - It send stop rx signal to serial device by using function Ajit_stop_Rx(port).
  - then it frees registered irq by calling function free_irq(port->irq, port).

5.6 Serial Driver Module Unloading

At time of module unloading following functions are executed:

- __exit Ajit_exit(void).
  - uart_unregister_driver(&Ajit_serial_driver))
  - end

6 Appendix

6.1 Linux Inbuilt Functions

6.1.1 ioread8

unsigned int ioread8(void __iomem *addr)
__iomem annotation used to mark pointers to I/O memory.
This function is used to read 8 bit data from given memory location. Memory address is return value of ioremap().

6.1.2 ioread32be

unsigned int ioread32be(void __iomem *addr)
__iomem annotation used to mark pointers to I/O memory.
This function is used to read 32 bit data from given memory location. Memory address is return value of ioremap().

6.1.3 iowrite8

It is is used to write 8 bit data to device.

6.1.4 iowrite32be

It is is used to write 32 bit data to device.

6.1.5 request_irq

int **request_irq**(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_regs *), unsigned long irqflags, const char *devname, void *dev_id)
This function is used to register ISR in kernel.

6.2 Definition of Function From uart_ops

6.2.1 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.

6.2.2 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_OUT2OUT2 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.

6.2.3 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.

6.2.4 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

6.2.5 start_tx(port)

Start transmitting characters.

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

6.2.6 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.

6.2.7 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.

6.2.8 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

6.2.9 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

6.2.10 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.

6.2.11 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.

6.2.12 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

6.2.13 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):

character received, marked as TTY_NORMAL
Parity error:n/a INPCK:0 IGNPAR:n/a

character received, marked as TTY_NORMAL
Parity error:None INPCK:1 IGNPAR: n/a

character received, marked as TTY_PARITY
Parity error:Yes INPCK:1 IGNPAR:0

character discarded
Parity error:Yes INPCK:1 IGNPAR:1
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

6.2.14 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.

6.2.15 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.

6.2.16 release_port(port)

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

Locking: none.
Interrupts: caller dependent.

6.2.17 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.

6.2.18 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 auto-configuration of the interrupt signal, which should be probed using standard kernel auto-probing 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.

6.2.19 verify_port(port,serinfo)

Verify the new serial port information contained within serinfo is suitable for this port type.

Locking: none.
Interrupts: caller dependent.

6.2.20 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.

6.2.21 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.

6.2.22 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

6.2.23 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.