Device Driver
for Serial Communication

RTOS Support

- NOT necessarily responsible everything
- RTOS system calls

Interrupt Management
rtl_request_irq
rtl_free_irq
rtl_hard_enable_irq
rtl_hard_disable_irq

Time Management
clock_gethrtime
clock_gettime
clock_settime
gethrtime
nanosleep

Task Management
pthread_create
pthread_setschedparam// pri. sched
pthread_make_periodic_np
pthread_wait_np
pthread_delete_np
pthread_cancel
pthread_join

Task Communication
FIFO
Shared Memory
Signal

Mutual Exclusion
Lock
Semaphore

Device drivers
rt_com
rtsock

Various Devices

- Serial com (UART – RS232)
- Parallel com
- Ethernet card (TCP/IP – socket interface)
- CAN bus
- CDMA communication chip set
- LCD

Device driver (1)

- Simplify the access to devices
  - Hide device specific details as much as possibile
  - Provide a consistent way to access different devices
- ioctl(), read(), write()

Device Driver (2)

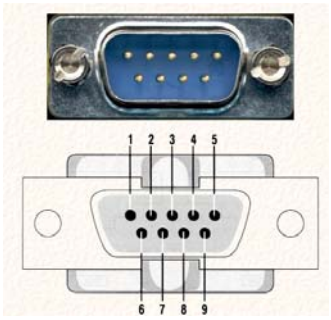
- Writing a device driver
  - Understand basic physical mechanism
  - Understand supporting HW chips
  - Implement standard interface functions
    - Register necessary interrupt handlers
    - Define internal data structure (message buffer)
    - Implement standard interface functions
      - ioctl, read, write

Device Driver (2)

- A device driver USER only needs to know (standard) interface functions without knowledge of physical properties of the device
- A device driver DEVELOPER needs to know physical details and provides the interface functions as specified

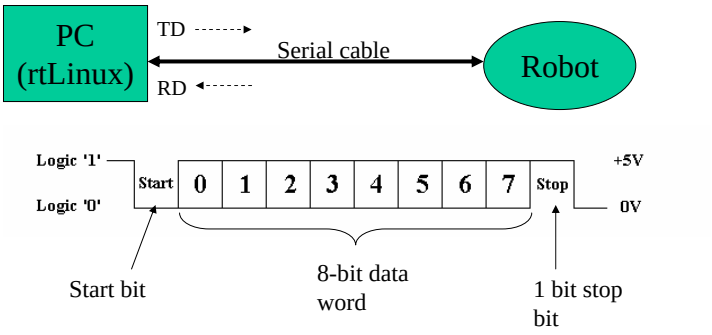
Let's take serial com as an example

- Physical communication mechanism
  - RS-232:widely used industrial standard



pin	name	function
1	DCD	asserted when the modem detects a "carrier"from the other end
2	***RD***	Serial data input
3	***TD***	Serial data output
4	DTR	DTE (UART) ready to establish a link
5	SG	Signal ground
6	***DSR***	DCE (modem) ready to establish a link
7	***RTS***	DTE (UART) ready to exchange data
8	CTS	DCE (modem) ready to exchange data
9	RI	Asserted when modem detects a ringing signal from the PSTN

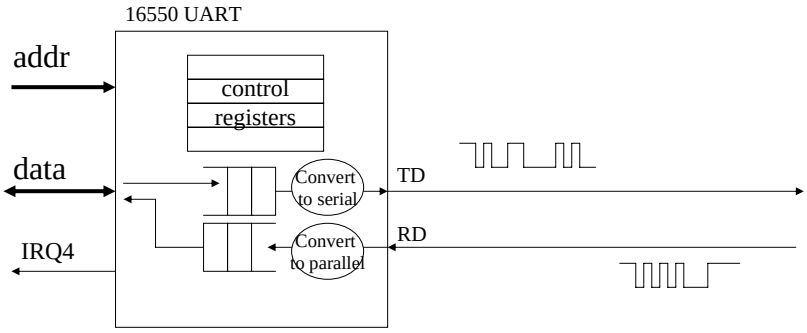
One word communication (8bit data, no parity, 1 stop bit)



- Why we need start bit in asynchronous communication even when the both sides use the same baud rate?

Program controls bit level timing? (think about 115200 bps)

- UART (8250 and compatibles) will help



Serial communication is possible by touching UART registers

- Port Addresses & IRQ's (standard PC)

Name	Address	IRQ
COM 1	3F8	4
COM 2	2F8	3
COM 3	3E8	4
COM 4	2E8	3

- Table of registers

Base Address	DLAB	Read/Write	Abr.	Register Name
+ 0	=0	Write	-	Transmitter Holding Buffer
	=0	Read	-	Receiver Buffer
	=1	Read/Write	-	Divisor Latch Low Byte
+ 1	=0	Read/Write	IER	Interrupt Enable Register
	=1	Read/Write	-	Divisor Latch High Byte
+ 2	-	Read	IIR	Interrupt Identification Register
	-	Write	FCR	FIFO Control Register
+ 3	-	Read/Write	LCR	Line Control Register
+ 4	-	Read/Write	MCR	Modem Control Register
+ 5	-	Read	LSR	Line Status Register
+ 6	-	Read	MSR	Modem Status Register
+ 7	-	Read/Write	-	Scratch Register

Divisor Latch Low/High Bytes (program Baud Rate Generator)

Speed (BPS)	Divisor (Dec)	Divisor Latch High Byte	Divisor Latch Low Byte
50	2304	09h	00h
300	384	01h	80h
600	192	00h	C0h
2400	48	00h	30h
4800	24	00h	18h
9600	12	00h	0Ch
19200	6	00h	06h
38400	3	00h	03h
57600	2	00h	02h
115200	1	00h	01h

Interrupt Enable Register (IER) and Interrupt Identification Register (IIR)

- IER

Bit	Notes
Bit 7	Reserved
Bit 6	Reserved
Bit 5	Enables Low Power Mode (16750)
Bit 4	Enables Sleep Mode (16750)
Bit 3	Enable Modem Status Interrupt
Bit 2	Enable Receiver Line Status Interrupt
Bit 1	Enable Transmitter Holding Register Empty Interrupt
Bit 0	Enable Received Data Available Interrupt

- IIR (read only)

Bit			Notes
Bits 6 and 7	Bit 6	Bit 7	
	0	0	No FIFO
	0	1	FIFO Enabled but Unusable
	1	1	FIFO Enabled
Bit 5	64 Byte Fifo Enabled (16750 only)		
Bit 4	Reserved		
Bit 3	0	Reserved on 8250, 16450	
	1	16550 Time-out Interrupt Pending	
Bits 1 and 2	Bit 2	Bit 1	
	0	0	Modem Status Interrupt
	0	1	Transmitter Holding Register Empty Interrupt
	1	0	Received Data Available Interrupt
	1	1	Receiver Line Status Interrupt
Bit 0	0	Interrupt Pending	
	1	No Interrupt Pending	

FIFO Control Register (FCR)

- FCR (write only)

Bit	Notes		
Bits 6 and 7	Bit 7	Bit 6	Interrupt Trigger Level
	0	0	1 Byte
	0	1	4 Bytes
	1	0	8 Bytes
	1	1	14 Bytes
Bit 5	Enable 64 Byte FIFO (16750 only)		
Bit 4	Reserved		
Bit 3	DMA Mode Select. Change status of RXRDY & TXRDY pins from mode 1 to mode 2.		
Bit 2	Clear Transmit FIFO		
Bit 1	Clear Receive FIFO		
Bit 0	Enable FIFO's		

Line Control Register (LCR)

- LCR

Bit 7	1	Divisor Latch Access Bit		
	0	Access to Receiver buffer, Transmitter buffer & Interrupt Enable Register		
Bit 6	Set Break Enable			
Bits 3, 4 And 5	Bit 5	Bit 4	Bit 3	Parity Select
	X	X	0	No Parity
	0	0	1	Odd Parity
	0	1	1	Even Parity
	1	0	1	High Parity (Sticky)
	1	1	1	Low Parity (Sticky)
Bit 2	Length of Stop Bit			
	0	One Stop Bit		
	1	2 Stop bits for words of length 6,7 or 8 bits or 1.5 Stop Bits for Word lengths of 5 bits.		
Bits 0 And 1	Bit 1	Bit 0	Word Length	
	0	0	5 Bits	
	0	1	6 Bits	
	1	0	7 Bits	
	1	1	8 Bits	

Line Status Register (LSR)

- LSR (read only)

	Bit	Notes
	Bit 7	Error in Received FIFO
	Bit 6	Empty Data Holding Registers
√	Bit 5	Empty Transmitter Holding Register
	Bit 4	Break Interrupt
	Bit 3	Framing Error
	Bit 2	Parity Error
	Bit 1	Overrun Error
√	Bit 0	Data Ready

Modem Control Register (MCR)

- MCR

Bit	Notes
Bit 7	Reserved
Bit 6	Reserved
Bit 5	Autoflow Control Enabled (16750 only)
Bit 4	LoopBack Mode
Bit 3	Aux Output 2
Bit 2	Aux Output 1
Bit 1	Force Request to Send
Bit 0	Force Data Terminal Ready

Modem Status Register (MSR)

- MSR (read only)

Bit	Notes
Bit 7	Carrier Detect
Bit 6	Ring Indicator
Bit 5	Data Set Ready
Bit 4	Clear To Send
Bit 3	Delta Data Carrier Detect
Bit 2	Trailing Edge Ring Indicator
Bit 1	Delta Data Set Ready
Bit 0	Delta Clear to Send

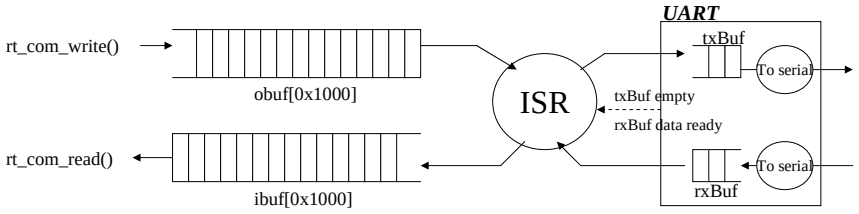
- For more details, see http://www.beyondlogic.org/serial/serial.htm

How the driver should work?

- Three standard interface functions
  - rt_com_setup: configure serial com parameters
  - rt_com read: read data received from COM
    - int rt_com_read(com, char *ptr, byteCountToBeRead);
  - rt_com write: write data to be transmitted to COM
    - void rt_com_write(com, char *ptr, byteCountToBeWritten);

How the driver should work?

- Real-Time response!
  - Non-blocked I/O (cf. Blocked I/O)
    - “Blocking until the read/write is actually done” is not good for real-time.
    - Return immediately even if the operation is partially done and let the rest done whenever UART is ready.
    - This allows RT_Thread make a decision with the partial result.
  - Use internal data structure (message buffer (cf. HW buffer))



Driver Initialization

- Initialize the internal data structure
- Register IRQ4 (COM1) and IRQ3 (COM2) Interrupt Service Routines (ISRs)
  - rtl_request_irq(irq, isr);
  - rtl_hard_enable_irq(irq);
- Setup UART
  - Initial setup
  - Setup can be changed later by Rt thread (rt_com_setup)

Source code

```
struct rt_com_struct rt_com_table[ RT_COM_CNT ] =
{
    { 0, RT_COM_BASE_BAUD, 0x3f8, 4, RT_COM_STD_COM_FLAG, rt_com0_isr, 1, 1 },
/* ttyS0 - COM1 */
    { 0, RT_COM_BASE_BAUD, 0x2f8, 3, RT_COM_STD_COM_FLAG, rt_com1_isr, 1, 1 }
/* ttyS1 - COM2 */
};

int init_module( void )
{
    struct rt_com_struct *p;
    int errorcode = 0, i, j;
    for( i=0; i<RT_COM_CNT; i++ ) {
        p = &( rt_com_table[ i ] );
        if( p->used > 0 ) {
            if(-EBUSY == check_region( p->port, 8 ) ) {
                errorcode = 1;
                break;
            }
            request_region( p->port, 8, "rt_com" );
            rt_com_request_irq( p->irq, p->isr );
            rt_com_setup( i, 0, 0, 0, 0 );
        }
    }
    if( 0 == errorcode ) {
        printk( KERN_INFO "rt_com: RT-Linux serial port driver (version "
                VERSION ") sucessfully loaded.\n"
                KERN_INFO "rt_com: Copyright (C) 1997-2000 Jochen Küpper et al.\n" );
    } else {
        printk( KERN_WARNING "rt_com: cannot request all port regions,\nrt_com: giving up.\n" );
        for( j=0; j<i; j++ ) {
            p = &( rt_com_table[ j ] );
            if( 0 < p->used ) {
                rt_com_free_irq( p->irq );
                release_region( p->port, 8 );
            }
        }
    }
    return( errorcode );
}
```

rt_com_setup

- Input parameters: com, baud, parity, stopbits, wordlength
- Operations
  - Disable all UART interrupts – ***IER***
  - Clear all status registers – ***IIR***, ***LSR***, ***MSR***, ***ReceiverBuffer***
  - Setup baud rate – ***DivisorLatchLow***, ***DivisorLatchHigh***
  - Setup stopbits, parity, wordlength – ***LCR***
  - Assert RS-232 signals DTR and RTS – ***MCR***
  - Interrupt enable for “receive data” and “receive line status change (if LSR changes)” – ***IER***
  - Enable fifo – ***FCR***

Source code

```
        /* Stop everything, set DLAB */
        outb( 0x00, base + RT_COM_IER );
        outb( 0x80, base + RT_COM_LCR );

        /* clear irq */
        inb( base + RT_COM_IIR );
        inb( base + RT_COM_LSR );
        inb( base + RT_COM_RXB );
        inb( base + RT_COM_MSR );

        p->error = 0; //!! init. last error code
        p->callback = 0;

        if( 0 == baud ) {
            /* return */
        } else if( 0 > baud ) {
            MOD_DEC_USE_COUNT;
        } else {
            MOD_INC_USE_COUNT;
            divider = p->baud_base / baud;
            outb( divider % 256, base + RT_COM_DLL );
            outb( divider / 256, base + RT_COM_DLM );
            /* bits 3,4 + 5 determine parity - clear all other bits */
            par &= 0x38;
            /* set transmission parameters and clear DLAB */
            outb( ( wordlength - 5 ) + ( ( stopbits - 1 ) << 2 ) + par, base + R
T_COM_LCR );
            p->mcr = RT_COM_DTR + RT_COM_RTS + RT_COM_Out1 + RT_COM_Out2;
            outb( p->mcr , base + RT_COM_MCR );
            if (p->mode & 0x01) p->ier = 0x05; /* if no hs signals enable only r
eceiver interrupts  */
            else p->ier = 0x0D;                /* else enable receiver and modem
 interrupts */
            outb( p->ier, base + RT_COM_IER );
            rt_com_enable_fifo( base, RT_COM_FIFO_TRIGGER, 0 );
            p->used |= 0x02; /* mark setup done */
        }
```

rt_com_read

- Input parameters: com, dataBuf, byteCountToBeRead
- Operations
  - rt_com_irq_off: enter critical section
  - Copy from “rt_com ibuf” to “dataBuf” up to actualByteCount = min(byteCountToBeRead, byteCountReadyInBuffer)
  - rt_com_irq_leave: leave critical section
  - Return “actualByteCount”

- Note: No blocking inside rt_com_read

Source code

```
int rt_com_read( unsigned int com, char *ptr, int cnt )
{
    int done = 0;
    if( com < RT_COM_CNT ) {
        struct rt_com_struct *p = &( rt_com_table[ com ] );
        struct rt_buf_struct *b = &( p->ibuf );
        long state;
        if( p->used > 0) {
            rt_com_irq_off( state );
            while( ( b->head != b->tail ) && ( --cnt >= 0 ) ) {
                done++;
                *ptr++ = b->buf[ b->tail++ ];
                b->tail &= ( RT_COM_BUF_SIZ - 1 );
            }
            rt_com_irq_on( state );
            if( ( p->mode & 0x02 )
                && ( rt_com_buff_free( b->head, b->tail ) > RT_COM_BUF_HI ) ) {
                /* if hardware flow and enough free space on input buffer
                   then set RTS */
                p->mcr |= 0x02;
                outb( p->mcr, p->port+RT_COM_MCR );
            }
        }
    }
    return( done );
}
```

rt_com_write

- Input parameters: com, dataBuf, byteCountToBeWritten
- Operations
  - rt_com_irq_off: enter critical section
  - Copy data from “dataBuf” to “rt_com obuf” (no of bytes = byteCountToBeWritten)
  - Enable UART interrupt “Transmitter Holding Register Empty” – ***IER***
  - rt_com_irq_leave: leave critical section

- Note: No blocking inside rt_com_write

Source code

```
void rt_com_write( unsigned int com, const char *buffer, int count )
{
    const char *data = buffer;
    if( com < RT_COM_CNT ) {
        struct rt_com_struct *p = &( rt_com_table[ com ] );
        struct rt_buf_struct *b = &( p->obuf );
        long state;
        if( p->used > 0 ) {
            rt_com_irq_off( state );
            while( --count >= 0 ) {
                /* put byte into buffer, move pointers to next elements */
                b->buf[ b->head++ ] = *data++;
                /* if( head == RT_COM_BUF_SIZ ), wrap head to the first buffer e
lement */
                b->head &= ( RT_COM_BUF_SIZ - 1 );
            }
            p->ier |= 0x02;
            outb( p->ier, p->port + RT_COM_IER );
            rt_com_irq_on( state );
        }
    }
}
```

ISR

- Triggered by “line-status-change”, “receive-data-ready”, and “transmit-holding-buffer-empty”
- Operations
  - Check if received data is ready in Rx HW buffer – ***LSR***
    - While there is a byte in Rx HW buffer, copy it to “rt_com ibuf”
  - Check if there is a room in “transmit-holding-buffer” – ***LSR***
    - If rt_com obuf is not empty (data to be transmitted), copy up to 16 (Tx HW buffer size) bytes to Tx HW buffer
    - If rt_com obuf becomes empty (no more data to be transmitted), disable “transmit-holding-buffer-empty” interrupt – ***IER***

Source code

Check if received data is ready

Check if txHW buffer is empty

```
    do {
        /* get available data from port */
        sta = inb( B + RT_COM_LSR );
        if( 0x1e & sta)
            p->error = sta & 0x1e;
        while( RT_COM_DATA_READY & sta ) {
            data = inb( B + RT_COM_RXB );
            rt_com_irq_put( p, data );
            rxd_bytes = 1;
            sta = inb( B + RT_COM_LSR );
            if( 0x1e & sta )
                p->error = sta & 0x1e;
        }
        /* controls on buffer full and RTS clear on hardware flow control */
        buff = rt_com_buff_free( b->head, b->tail );
        if (buff < RT_COM_BUF_FULL)
            p->error = RT_COM_BUFFER_FULL;

        if ( ( p->mode & 0x02 ) && ( buff < RT_COM_BUF_LOW ) ) {
            p->mcr &= ~0x02;
            outb( p->mcr, p->port+RT_COM_MCR );
        }
        /* if possible, put data to port */
        sta = inb( B + RT_COM_MSR );
        if ( ( ( 0x20 & sta )
               && ( ( 0x10 & sta ) || ( 0 == ( p->mode & 0x02 ) ) ) ) || ( p->mo
de & 0x01 ) ) {
            /* (DSR && (CTS || Mode==no hw flow)) or Mode==no hand shake */
            if ( ( sta = inb(B + RT_COM_LSR) ) & 0x20 ) { // if (THRE==1) i.e. t
ransmitter empty
                if( 0 != ( t = rt_com_irq_get( p, &data ) ) ) { /* if data in ou
tput buffer */
                    do {
                        outb( data, B + RT_COM_TXB );
                    } while( ( --toFifo > 0 ) && ( 0 != ( t = rt_com_irq_get( p,
 &data ) ) ) );
                }
                if( ! t ) {
                    /* no more data in output buffer, disable Transmitter Holdin
g Register Empty Interrupt */
                    p->ier &= ~0x02;
                    outb( p->ier, B + RT_COM_IER );
                }
            }
        }

      /* check the low nibble of IIR to check if there is another pending interr
upt */
        /* Note: why is it done at most 4 times ? */
    } while( 1 != ( ( inb( B + RT_COM_IIR ) & 0x0f ) ) && ( --loop > 0 ) );
```

User only needs to know

- Three standard interface functions
  - rt_com_setup: configure serial com parameters
  - rt_com read: read data received from COM
    - int rt_com_read(com, char *ptr, byteCountToBeRead);
  - rt_com write: write data to be transmitted to COM
    - void rt_com_write(com, char *ptr, byteCountToBeWritten);