**UART**

UART (Universal Asynchronous Receiver and Transmitter) unit provides two independent asynchronous serial I/O (SIO) ports, each of which can operate in interrupt-based or DMA-based mode. In other words, UART can generate an interrupt or DMA request to transfer data between CPU and UART. It can support bit rates of up to 115.2K bps. Each UART channel contains two 16-byte FIFOs for receive and transmit (Depends on Datasheet).

There are two primary forms of serial transmission: Synchronous and Asynchronous. Depending on the modes that are supported by the hardware, the name of the communication sub-system will usually include a A if it supports Asynchronous communications, and a S if it supports Synchronous communications. Both forms are described below.

Some common acronyms are:

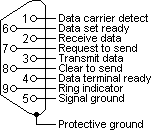
UART Universal Asynchronous Receiver/Transmitter

USART Universal Synchronous-Asynchronous Receiver/Transmitter

**Synchronous serial transmission** requires that the sender and receiver share a clock with one another, or that the sender provide a strobe or other timing signal so that the receiver knows when to “read” the next bit of the data. In most forms of serial Synchronous communication, if there is no data available at a given instant to transmit, a fill character must be sent instead so that data is always being transmitted.

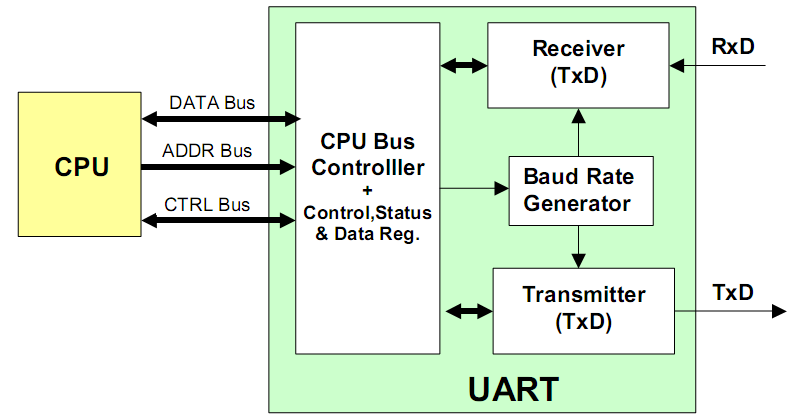
**Asynchronous transmission** allows data to be transmitted without the sender having to send a clock signal to the receiver. Instead, the sender and receiver must agree on timing parameters in advance and special bits are added to each word which are used to synchronize the sending and receiving units.When a word is given to the UART for Asynchronous transmissions, a bit called the "Start Bit" is added to the beginning of each word that is to be transmitted. The Start Bit is used to alert the receiver that a word of data is about to be sent, and to force the clock in the receiver into synchronization with the clock in the transmitter. These two clocks must be accurate enough to not have the frequency drift by more than 10% during the transmission of the remaining bits in the word.

**RS232 DB9 pinout**



**UART Architecture**

The UART circuit enables a computing processing unit (CPU) serial access to the external peripheral. The interface between the CPU and the UART is usually byte parallel and can be synchronous (i.e. Register Map interface). The transmission properties of the UART, such as parity check, number of symbol bits, number of stop bits etc., can be programmed via a control register which is part of the UART circuitry. The CPU can configure the UART by writing the specific control bits via the parallel interface.



The URAT consists of the following four main function blocks:

1. CPU Bus Controller
2. Baud rate generator
3. Receiver
4. Transmitter

**CPU Bus Controller**

The CPU Bus Controller provides the parallel data I/O interface to the local processor bus. It generates the necessary control signal to enable the uP (or CPU) to access onto the data, status and control register of the UART circuit. Furthermore, it generates the local control signal within the UART circuit according to the control register content and updates the status register content according to the local status signal values generated by the other function blocks. It also accommodates the transmitter and receiver buffer (Hold Register or FIFO) that is essential for asynchronous transmission.

**Baud Rate Generator**

The Baud Rate Generator is a programmable transmit and receive bit timing device. Given the programmed value, it generates a periodic pulse, which determines the baud rate of the UART transmission. This pulse is used by the receiver and transmitter circuit to generate a sampling pulse for sampling the received serial data and to determine the bit width of the transmit data.

**Receiver section**

The Receiver block detects the start bit of an incoming serial data and samples the data bits, bit by bit, according to the baud clock of the baud rate generator. It completes the receive process of a single symbol of 6,7 or 8 bits with the detection of the stop bit (the stop bit can be 1, 1.5 or 2 bits width). Parity check of the received symbol ensures that the data has been received correctly. In the case of invalid stop bit or parity check error, the status signals parity error or frame error will be set. Finally the receiver writes the received symbol data onto the local data bus, which connected to the CPU Bus controller, and sets a signal to indicate “Receiver data Write”. This signal initiates that the CPU Bus controller informs the CPU via an interrupt about the data arrival.

***Explanation with registers***

The receiver section contains an 8-bit Receive Shift Register (RSR) and 64 bytes of FIFO which includes a byte-wide Receive Holding Register (RHR). The RSR uses the 16X/8X/4X clock (DLD [5:4]) for timing. It verifies and validates every bit on the incoming character in the middle of each data bit. On the falling edge of a start or false start bit, an internal receiver counter starts counting at the 16X/8X/4X clock rate. After 8 clocks (or 4 if 8X or 2 if 4X) the start bit period should be at the center of the start bit. At this time the start bit is sampled and if it is still a logic 0 it is validated. Evaluating the start bit in this manner prevents the receiver from assembling a false character. The rest of the data bits and stop bits are sampled and validated in this same manner to prevent false framing. If there were any error(s), they are reported in the LSR register bits 2-4. Upon unloading the receive data byte from RHR, the receive FIFO pointer is bumped and the error tags are immediately updated to reflect the status of the data byte in RHR register. RHR can generate a receive data ready interrupt upon receiving a character or delay until it reaches the FIFO trigger level. Furthermore, data delivery to the host is guaranteed by a receive data ready time-out interrupt when data is not received for 4 word lengths as defined by LCR[1:0] plus 12 bits time. This is equivalent to 3.7-4.6 character times. The RHR interrupt is enabled by IER bit-0.

**Receive Holding Register (RHR) - Read-Only**

The Receive Holding Register is an 8-bit register that holds a receive data byte from the Receive Shift Register. It provides the receive data interface to the host processor. The RHR register is part of the receive FIFO of 64 bytes by 11-bits wide, the 3 extra bits are for the 3 error tags to be reported in LSR register. When the FIFO is enabled by FCR bit-0, the RHR contains the first data character received by the FIFO. After the RHR is read, the next character byte is loaded into the RHR and the errors associated with the current data byte are immediately updated in the LSR bits 2-4.

**Transmitter Section**

The transmitter block is responsible for the serial transmitting of the data, which is written by the uP (CPU) onto the TxD Hold register (or FIFO) at the CPU Bus controller block. First the transmitter detects whether the UART transmitter buffer (FIFO or TxD Hold Register) contains data for transmission. If it does, it loads the data onto the transmit register at the transmitter circuit via the local data bus (which connects the CPU Bus controller and the transmitter) and sets a signal to indicate “Transmit data Read”. This signal initiates a sequence where the CPU Bus controller informs the CPU via an interrupt about the transmitted data, so that the CPU can load a new value. Synchronous to the baud clock which is generated by the baud rate generator, the transmitter sets the start bit on the TxD signal line to initiate the start of a frame and then bit by bit the symbol data. It finally completes the transmission by sending a parity bit that represents the parity of transmitted data and completes the frame with the final stop bit. The procedure will be repeated for another symbol, if the transmitter buffer contains another symbol, else the transmitter goes to an idle mode, transmitting “Marks”.

***Explanation with registers***

The transmitter section comprises of an 8-bit Transmit Shift Register (TSR) and 64 bytes of FIFO which includes a byte-wide Transmit Holding Register (THR). TSR shifts out every data bit with the 16X/8X/4X internal clock. A bit time is 16 (8 if 8X or 4 if 4X) clock periods (see DLD[5:4]). The transmitter sends the start-bit followed by the number of data bits, inserts the proper parity-bit if enabled, and adds the stop-bit(s). The status of the FIFO and TSR are reported in the Line Status Register (LSR[6:5]).

**Transmit Holding Register (THR) - Write Only**

The transmit holding register is an 8-bit register providing a data interface to the host processor. The host writes transmit data byte to the THR to be converted into a serial data stream including start-bit, data bits, parity-bit and stop-bit(s). The least-significant-bit (Bit-0) becomes first data bit to go out. The THR is the input register to the transmit FIFO of 64 bytes when FIFO operation is enabled by FCR bit-0. Every time a write operation is made to the THR, the FIFO data pointer is automatically bumped to the next sequential data location.

**Transmitter Operation in non-FIFO Mode**

The host loads transmit data to THR one character at a time. The THR empty flag (LSR bit-5) is set when the data byte is transferred to TSR. THR flag can generate a transmit empty interrupt (ISR bit-1) when it is enabled by IER bit-1. The TSR flag (LSR bit-6) is set when TSR becomes completely empty.

**Transmitter Operation in FIFO Mode**

The host may fill the transmit FIFO with up to 64 bytes of transmit data. The THR empty flag (LSR bit-5) is set whenever the FIFO is empty. The THR empty flag can generate a transmit empty interrupt (ISR bit-1) when the amount of data in the FIFO falls below its programmed trigger level. The transmit empty interrupt is enabled by IER bit-1. The TSR flag (LSR bit-6) is set when TSR/FIFO becomes empty.

**How baud rate generates:**

**Some of the common registers of uart(16c550) are below**

**Linux Kernel with UART**

Uart is generally relies on tty driver along with serial core file. The Linux serial driver is tightly coupled with the TTY subsystem. The TTY layer is a separate class of character driver. On embedded systems having a serial port, the TTY layer is used for providing access to the low-level serial port.

A user process does not talk to the serial driver directly. TTY presents a stack of software over the driver and exports the entire functionality via TTY devices.

The TTY subsystem is split into three layers as shown

Application

In kernel

Application

Line discipline

TTY I/O LAYER

tty\_io.c

Serial\_core.c

Low level driver

S3c2410.c/driver

Hardware

So writing a new driver of any external uart,some important structures and functions are available I tty\_io.c & serial\_core.c.

**structures**

**struct uart\_driver**: This data structure contains information about the name, major and minor numbers, and number of ports of this driver.

**struct uart\_port**: This data structure contains all the configuration data of the low-level hardware.

**struct uart\_ops**: This data structure contains the pointers to functions that operate on the hardware.

struct uart\_driver {

struct module \*owner;

const char \*driver\_name;

const char \*dev\_name;

int major;

int minor;

int nr;

};

struct uart\_port {

unsigned int iobase; /\* in/out[bwl] \*/

unsigned char \_\_iomem \*membase; /\* read/write[bwl] \*/

unsigned int irq; /\* irq number \*/

unsigned int uartclk; /\* base uart clock \*/

unsigned int fifosize; /\* tx fifo size \*/

unsigned char x\_char; /\* xon/xoff char \*/

unsigned char regshift; /\* reg offset shift \*/

unsigned char iotype; /\* io access style \*/

unsigned int line; /\* port index \*/

resource\_size\_t mapbase; /\* for ioremap \*/

};

struct uart\_ops {

unsigned int (\*tx\_empty)(struct uart\_port \*);

void (\*set\_mctrl)(struct uart\_port \*, unsigned int mctrl);

unsigned int (\*get\_mctrl)(struct uart\_port \*);

void (\*stop\_tx)(struct uart\_port \*);

void (\*start\_tx)(struct uart\_port \*);

void (\*send\_xchar)(struct uart\_port \*, char ch);

void (\*stop\_rx)(struct uart\_port \*);

void (\*enable\_ms)(struct uart\_port \*);

void (\*break\_ctl)(struct uart\_port \*, int ctl);

int (\*startup)(struct uart\_port \*);

void (\*shutdown)(struct uart\_port \*);

void (\*set\_termios)(struct uart\_port \*, struct ktermios \*new,

struct ktermios \*old);

void (\*pm)(struct uart\_port \*, unsigned int state,

unsigned int oldstate);

int (\*set\_wake)(struct uart\_port \*, unsigned int state);

};

The above structures are to be filled while writing a new uart/serial driver.

**Source code flow**

1. Fill the above structures according to the datasheet.
2. In the init function of the driver,pass the uart\_driver structures variables to uart\_register\_driver i.e

uart\_register\_driver (&uart\_driver);

uart\_register\_driver binds the low level driver with the serial CORE which in turn registers with the TTY layer using the tty\_register\_driver() function. Also the uart\_state structures are created (the number of these structures are equivalent to number of hardware ports) and pointer to this array is stored in my\_uart\_driver.

1. Then uart\_add\_one\_port (&uart\_driver, &uart\_port);

this function connects the uart\_state to the uart\_port. Also this function lets the TTY layer know that a device has been added using the function tty\_register\_device().

There is one private structure held by the kernel — uart\_state. The number of uart\_state is equivalent to the number of hardware ports that are accessed via the driver. Each state contains a pointer to the per-port settings uart\_port, which in turn contains the structure uart\_ops holding the routines for accessing the hardware.

1. The uart\_ops structure is being passed in the uart\_port structure.
2. In the uart\_ops structure,

**Start\_up**:

When device is being opened.This also includes registration of IRQ assigned.Like setting up the some basic registers according to datasheet.

**Shutdow**n:

While device is being closed(close(fd)) this is being called and the corresponding irq is being freed.

**Set\_termios**:

While setting up the baud rate,parity,data bits,stop bit etc,this function pointer is being called which is pointing to the corresponding function. Implementing the termios settings is done across various TTY layers; the low-level driver has to bother only about the control options. These options are set using the c\_cflag field of the terminos structure.

**Request\_port/config\_port:**

Here we are passing the memory mapped address provided by the platform device in arch/arm/mach-xxx /devices.c file.These has been done using same request\_mem\_region() & ioremap()..If the device in interfaced with i2c/spi,this may not be necessary.

**Release\_port:**

Releases the memory occupied by above.

**Start\_tx:**

This is being called when transmission is being started.Transmission can also be based on interrupt.If it is so,in this function only transmit bit in IER is being set otherwise data transmission can be implemented here.

**Stop\_tx:**

To stop the transmission.

Before going to data transmission/reception part, lets us talk about interrupt.

* When data tx/rx happens, interrupt generates if set in the IER register(RHR/THR interrupt).
* Then it moves to read the ISR register where it can find about which interrupt generated and according to that it will go to the corresponding function.
* Following is the source code example:

pending = UART\_GET\_IIR(chip->port.line);

if (pending & 0x01) { //this means no interrupt

enable\_irq(IRQ\_EINT3);

return IRQ\_HANDLED;

}

switch (pending & 0x3f) {

case RX\_TIME\_OUT\_INT:

case RHR\_INT:

xrm1172\_rx\_chars(&chip->port);

break;

case THR\_INT:

xrm1172\_tx\_chars(&chip->port);

break;

case MODEM\_INT:

some operations;

break;

* There is a FCR register also which declares the FIFO enable/disable of RX/TX.Also important is the trigger levels based on which the RHR/THR interrupt generates

**Data Transmission**

Transmission starts with the my\_uart\_start\_tx() function; this function is invoked by the line discipline to start transmission. After the first character is transmitted, the rest of the transmission is done from the interrupt handler until all the characters queued up by the line discipline layer are transmitted.

It is implemented by the generic transmission function my\_uart\_char\_tx().The serial core provides a circular buffer mechanism for storing the characters that need to be transmitted. The serial core provides macros to operate on this buffer of which the following are used in this driver.

* **uart\_circ\_empty()** is used to find if the buffer is empty.
* **uart\_circ\_clear()** is used to empty the buffer.
* **uart\_circ\_chars\_pending()** is used to find the number of characters that are yet to be sent out.

Code prototype:

Reads the LSR value to check the tx\_empty flag and putting the value in the THR register

struct circ\_buf \*xmit = &port->info->xmit;

if (uart\_circ\_empty(xmit) || uart\_tx\_stopped(port)) {

xrm1172\_stop\_tx(port);

return;

}

while (1) {

unsigned char temp;

DEBUG("%c ", xmit->buf[xmit->tail]); //find the data here

UART\_PUT\_THR(port->line, xmit->buf[xmit->tail]);

do {

temp = UART\_GET\_LSR( port->line);

while(!(UART\_GET\_LSR(port->line) & THR\_EMPTY)) ;

} while(!(temp & THR\_EMPTY) );

xmit->tail = (xmit->tail + 1) & (UART\_XMIT\_SIZE - 1);

port->icount.tx++;

if (uart\_circ\_empty(xmit))

break;

}

if (uart\_circ\_chars\_pending(xmit) < WAKEUP\_CHARS)

uart\_write\_wakeup(port);

if (uart\_circ\_empty(xmit)) {

xrm1172\_stop\_tx(port);//in this function we can disable the THR interrupt in IER

}

**Data Reception**

Data reception happens in the context of an interrupt handler. The basis of the receive operation is the TTY flip buffer. This is a pair of buffers that is provided by the TTY layer. While one buffer is consumed by the line discipline for processing the characters received, the other buffer is available for writing. The TTY layer provides standard APIs for accessing the flip buffers. We are interested only in the functions for inserting the received character inside the available flip buffer and then flushing the received

characters to the line discipline from the flip buffer. These are done using the functions **tty\_insert\_flip\_char** and **tty\_flip\_buffer\_push**, respectively.

Code:

* Here,it reads the length of the data bytes by reading the RX FIFO level register.
* Until that length,it reads from the RHR register i.e

length = UART\_GET\_RXFIFO(chip->port.line);

until length--

ch = UART\_GET\_RHR(chip->port.line);

* And every time before pushing the data to the tty layer or application level,it checks with any frame/parity/oveerun error in LSR register.
* Another api’s

uart\_handle\_sysrq\_char(port, c.ch)) //doubt

uart\_insert\_char(port, status, OPTO\_US\_OVRE, c.ch, flg); //doubt

* tty\_flip\_buffer\_push(port->info->tty); //this will push the data to the tty layer



**Setting the baud rate in Linux kernel**

In set\_termios structure

int baud = uart\_get\_baud\_rate(port, termios, old, port->uartclk / 16 / 0xffff, port->uartclk / 16);

**port**: uart\_port structure describing the port in question.

**termios**: desired termios settings.

**old**: old termios (or NULL)

**min**: minimum acceptable baud rate

**max**: maximum acceptable baud rate

Decode the termios structure into a numeric baud rate,taking account of the magic 38400 baud rate (with spd flags), and mapping the %B0 rate to 9600 baud.If the new baud rate is invalid, try the old termios setting.If it's still invalid, we try 9600 baud.Update the termios structure to reflect the baud rate we're actually going to be using.

So if we are using 38400,the value of baud is 38400 using the above function.Then we will calculate the divisor for uart port using

unsigned int uart\_get\_divisor(struct uart\_port \*port, unsigned int baud)

**uart\_get\_divisor** - return uart clock divisor

**port**: uart\_port structure describing the port.

**baud**: desired baud rate

It Calculates the uart clock divisor for the port.

**Serial Port Application program**

**//This is the header files should be used**

**#include <stdio.h>**

**#include <termios.h>**

**#include <stdlib.h>**

**#include <sys/fcntl.h>**

**#include <sys/time.h>**

**#include <sys/ioctl.h>**

**#include <unistd.h>**

**#include <string.h>**

**#include <linux/types.h>**

**#include <stdint.h>**

**#include <getopt.h>**

char \*DEVICE = "/dev/ttyXX"; //device to open

struct termios sTermOptions,oTermOptions; //termios structure

int fd; //file descriptor

main()

{

unsigned char buffer[] = "Serial Application program";

unsigned char buffer1[64];

int i = 0;

fd = open(DEVICE, O\_RDWR);

if (fd < 0){

printf("can't open device");

exit(-1);

}

tcgetattr(fd, &oTermOptions); //getting & saving the old termios settings of current port

bzero(&sTermOptions, sizeof(sTermOptions));

sTermOptions.c\_cflag = B38400 | CS8 | CREAD;//baud rate,data bits,etc

sTermOptions.c\_cflag &= ~PARENB; //parity

sTermOptions.c\_cflag &= ~CSTOPB; //stopbit

tcflush(fd, TCIOFLUSH);//flushing the I/O buffers of the port

tcsetattr(fd, TCSANOW, &sTermOptions);//setting the new attributes

while(1)

{

int cnt = write(fd,buffer,sizeof(buffer)); //writing data

if (cnt > 0 ){}

else

printf("\nFailed write\n");

int cnt1 = read(fd,buffer1,sizeof(buffer)); //Reading data

sleep(1);

if(cnt1 > 0)

{

buffer1[cnt1] = '\0'; //To print as a string

printf("\nRead-buf = %s\n",buffer1);

}

else

printf("\nNo Read\n");

}

Close(fd);

**Using select() system call in case of uart**

Select is being used when multiple ports comes to pictures in One application. One case where it has been used is while Updating the Software of ECG, SPO2, NIBP etc hardware through arm board. The design was there is a external pc software which is used to program the above hardwares directly.But in case of updating the software through arm board we used this select() system call using the same pc software.Just Receive the data in any of RS-232 channel and redirect it to the corresponding serial port where ECG,SPO2,NIBP etc are being connected.

Please find the below design diagram

RS-232 cable

WFSLOAD PC SOFTWARE

ECG/NIBP/MODULE

Programmer hardware

In case of using arm board means without using the programmer hardware above

RS-232 cable ttyS0

ECG/NIBP/MODULES

ARM BOARD

WFSLOAD PC SOFTWARE

ttyS1

So it concludes

* Read the data from WFSLOAD using ttyS1 and redirect i.e write to ttyS0
* Read the response from the Module i.e ECG/NIBP ttyS1 and write it back to WFSLOAD through ttyS1
* It continues like this until last and there will be time out which will stop the select working if no data is available.

**Code**

**Same header files as above application**

#define ECG\_PORTNAME "/dev/ttyS2" //ecg port

#define WFS\_PORTNAME "/dev/ttyS0" //debug port

int max = 2;

unsigned char cdatastore[2];

unsigned char ch;

int flg = 0;

unsigned char value\_hex = 0xaa;

int wfs\_fd,module\_fd,debug\_fd;

int count\_flg = 0;

int count =0;

struct termios wfs\_newtp, wfs\_oldtp,wfs\_curtp,wfs\_durtp;

struct termios module\_newtp, module\_oldtp,module\_curtp,module\_durtp;

struct timeval Timeout;

main()

{

fd\_set rfds;

FD\_ZERO(&rfds);

int retval;

int index ;

int y = 0,i = 0,j = 0;

wfs\_fd = open(WFS\_PORTNAME, O\_RDWR | O\_NOCTTY | O\_NDELAY);

if(wfs\_fd < 0)

{

perror(WFS\_PORTNAME);

return 0;

}

tcgetattr(wfs\_fd,&wfs\_oldtp);

wfs\_newtp.c\_cflag = B38400 | CS8 | CREAD; //Opening Port at 38400

wfs\_newtp.c\_cflag &= ~PARENB;

wfs\_newtp.c\_cflag &= ~CSTOPB;

tcflush(wfs\_fd, TCIOFLUSH);

tcsetattr(wfs\_fd,TCSANOW,&wfs\_newtp);

tcgetattr(wfs\_fd,&wfs\_newtp);

wfs\_curtp.c\_cflag = B9600 | CS8 | CREAD; //Changing Baud Rate for Initial Hanshaking

tcflush(wfs\_fd,TCIOFLUSH);

tcsetattr(wfs\_fd,TCSANOW,&wfs\_curtp);

module\_fd = open(ECG\_PORTNAME, O\_RDWR | O\_NOCTTY | O\_NDELAY ); //ECG Port

if(module\_fd < 0)

{

perror(ECG\_PORTNAME);

return 0;

}

tcgetattr(module\_fd,&module\_oldtp);

module\_newtp.c\_cflag = B38400 | CS8 | CREAD; //Opening ECG Port at 38400

module\_newtp.c\_cflag &= ~PARENB;

module\_newtp.c\_cflag &= ~CSTOPB;

tcflush(module\_fd, TCIOFLUSH);

tcsetattr(module\_fd,TCSANOW,&module\_newtp);

tcgetattr(module\_fd,&module\_newtp);

module\_curtp.c\_cflag = B9600 | CS8 | CREAD; //Changing Baud rate for Handshaking

tcflush(module\_fd,TCIOFLUSH);

tcsetattr(module\_fd,TCSANOW,&module\_curtp);

while(1)

{

FD\_ZERO(&rfds);

FD\_SET(wfs\_fd,&rfds);

FD\_SET(module\_fd,&rfds);

Timeout.tv\_usec = 0; /\* milliseconds \*/

Timeout.tv\_sec = 10; /\* seconds \*/

retval = select(module\_fd+1,&rfds,NULL,NULL,&Timeout);

if(retval == 0){

system("reboot"); //if time out occurs

}

if(retval == -1){

exit(-1); //if select fails

}

else if(FD\_ISSET(wfs\_fd,&rfds)) //if select passes

{

read(wfs\_fd,cdatastore,1);

write(module\_fd,cdatastore,1);

}

else if(FD\_ISSET(module\_fd,&rfds))

{

read(module\_fd,cdatastore,1);

if(flg != 1)

{

if(ch == 0xAA )

{

if(cdatastore[0] == 0xFF || cdatastore[0] == 0xFE)

{

write(wfs\_fd,cdatastore,1);

ch = 0x00;

tcgetattr(wfs\_fd,&wfs\_curtp);

wfs\_durtp.c\_cflag = B38400 | CS8 | CLOCAL | CREAD; //Data rate is 38400-ECG

tcflush(wfs\_fd,TCIOFLUSH);

tcsetattr(wfs\_fd,TCSANOW,&wfs\_durtp);

tcgetattr(module\_fd,&module\_curtp);

module\_durtp.c\_cflag = B38400 | CS8 | CLOCAL | CREAD; //Data Rate is 38400-ECG

tcflush(wfs\_fd,TCIOFLUSH);

tcsetattr(module\_fd,TCSANOW,&module\_durtp);

flg = 1;

count\_flg = 1;

}

else

{

ch = 0x00;

}

}

}

if(cdatastore[0] == 0xAA)

{

ch = cdatastore[0];

write(wfs\_fd,cdatastore,1);

}

if(cdatastore[0] != 0xAA)

{

write(wfs\_fd,cdatastore,1);

}

}

}//while

return;

}