

Title: Serial data communication
Author: Craig Duffy 9/9/14, 11/01/18, 29/10/18, 08/02/22, 22/02/22, 23/1/23
Module: Mobile and Embedded Devices, Secure Embedded Systems
Awards: BSc CSI, BSc Forensic Computing, Computer Security.
Prerequisites: Basic computer architecture and some C

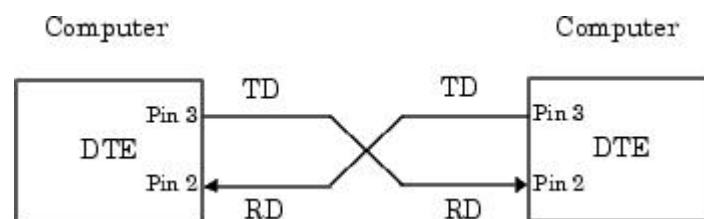
Having 1 bit input and output won't get us very far so we need to look a bit further afield to get something more useful. Firstly, we will look at basic serial communications.

Ye Olde RS-232

In computing terms basic serial, or RS-232 comms is pretty ancient – the original standard dates back to 1969! At one time it was the main source of connection between a computer, Data Terminal Equipment (DTE) in data communication jargon, and the outside world, Data Circuit-terminating Equipment (DCE) in the jargon. But now serial ports no longer come as standard. But this doesn't mean that RS-232 isn't worth studying. On embedded systems it is still the main way of connecting to a system, and on many devices, although it is often obscured, or even hidden, on the final products, it is heavily used in development. A large amount of 'technical' equipment, medical devices, industrial equipment, networking gear and so on all use serial ports as their main means of communication. Also, serial communication is still widely used e.g. USB and FireWire. So studying a serial protocol such as RS-232 will help you understand these technologies too.

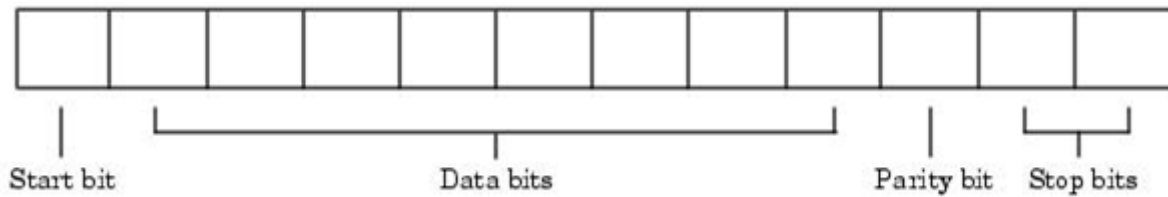
RS-232 does have its limitations, and this is part of the reason for its reduced usage. RS-232 data rates are relatively slow, the connectors are large and it is difficult to have multiple end points. However as a well understood, simple and reliable form of communication RS-232 still has a place in a computer engineers' tool kit.

The ARM Cortex M3 series come with a maximum of 5 serial devices. These devices are known as USART. USART stands for Universal Synchronous/Asynchronous Receiver Transmitter. This means that these ports can communicate with other devices with serial data in a synchronised mode (shared clock and data) or unsynchronised (shared data only). The Olimex board has 5 USARTs and these can be used to support such devices as serial ports/RS-232, infrared links IrDA, and modems.



The most basic serial communication link

The above diagram is for the UART configuration, meaning that the 2 computers only share data and don't share a common clock signal or have any other means for signalling their data transmissions. In order to work UARTs must send data in a standardised format thus allowing the receiver to disentangle the data and extract the required timing information from the data frame.

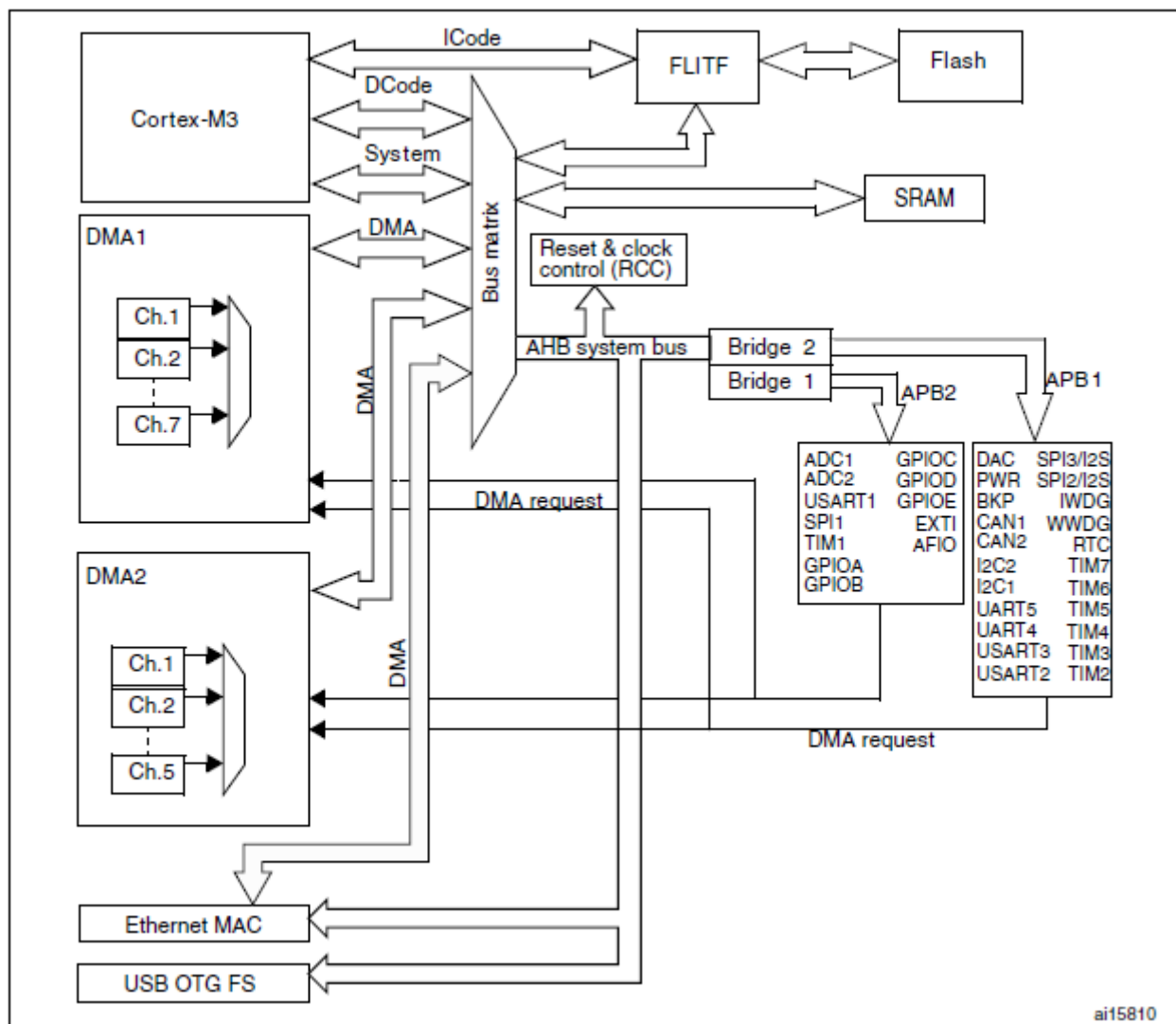


Serial data frame format.

The start bit allows the receiver to synchronise with the transmitter, however this requires the sender and receiver to agree on a transmission rate, normally called a baud rate. The data is the data being sent, so this could be ASCII characters or binary data for example. The parity bit is used for a simple low level error detection mechanism. The stops bit(s) are for further bit synchronisation, to allow the receiver to tell the difference between the end of one message and the start of the next.

USARTs on the STM32

The STM32 has been designed to support 5 USARTs although individual board designs may implement fewer devices. The diagram below shows the architecture of our device, the STM32F107.



The USART devices are, like all Cortex M3 peripherals, in a fixed memory location. The USART locations are given below in a table.

Boundary address	Peripheral	Bus
0x4001 3800 - 0x4001 3BFF	USART 1	APB2
0x4000 4400 - 0x4000 47FF	USART 2	APB1
0x4000 4800 - 0x4000 4BFF	USART 3	APB1
0x4000 4C00 - 0x4000 4FFF	USART 4	APB1
0x4000 5000 - 0x4000 53FF	USART 5	APB1

USART Devices, their addresses and Bus

Like other devices, such as the GPIO described in an earlier worksheet, the USART has a number of registers and all the devices have the same registers at similar offsets from the base addresses given above. These features make it easier for the software developer to create reasonably portable software fairly quickly. The registers are:

Name	Label	Offset	R/W	Reset
Status register	USART_SR	0x00	R	0x00C0
Data register	USART_DR	0x04	RW	Undefined
Baud rate register	USART_BRR	0x08	W	0x0000
Control register 1	USART_CR1	0x0C	W	0x0000
Control register 2	USART_CR2	0x10	W	0x0000
Control register 3	USART_CR3	0x14	W	0x0000
Guard time and prescaler register	USART_GTPR	0x18	RW	0x0000

USART registers and offsets.

Some of the names are fairly self-explanatory – so the status register tells us the status of the last transmission or reception, and it is clear why that is effectively a read only register. The data register is in effect 2 registers – if it is written to it is the transmit register, the data written is sent out. If it is read from it is the receive register, reflecting the data that has been sent to the device. The baud rate register controls the rate at which the data is sent and received. These are in fixed gradations of bps (Bit Per Second) – 9600, 19200, etc – which have been agreed by various international, governmental and industrial bodies to allow equipment to be able to send and receive data sensibly. The control registers allow the programmer to specify many features of the devices. As the USART can be used for a number of purposes, as an infra-red (IrDa) device, or as a Smart Card, there are many settings, most of which we won't need. Also, the peripherals can work in several modes, as interrupt driven devices or using DMA (direct memory access) for example. At the moment we will be using the

device in its simplest form, as a simple polled UART so we will need to do a minimal amount of set up.

However, we will need to look at the status and control registers in some detail.

The Status Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NE	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

detected NE, and overrun ORE, IDLE allowing the USART to detect if the line changes from being IDLE to busy and allows it to generate an interrupt. RXNE – stands for Receiver Not Empty – this bit detects whether data has been received (1) or not (0). The TC bit is for frame transmissions, mean Transmission Complete. TXE which is for single byte transmission meaning Transmit data register Empty, tells us whether data has been transmitted (1) or not (0). The LND and CTS bits are to do with LIN (Local Interconnect Network) status and hardware flow control signals, both of which we won't have to deal with.

Control Register 1

The only control register we need concern ourselves with is the first one, as the other deal with lots of flow control, LIN, DMA and other features we won't be using.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK	
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The main bits we need to know about are the UE (13), TE (3) and RE (2). These are USART Enable, Transmit Enable and Receive Enable respectively. These bits allow the device to do its basic work. The M (12), PCE (10) and PS (9) bits are to do with word length (8 or 9 bit plus Stop bits), Parity Control Enable and Parity Select. The Parity bit is used as a very simple form of error detection. It is a bit which tells the receiver the number of either odd or even 1s in the transmission. So if parity is set to even then the receiver would expect the transmitter to set the parity bit to 0 every time there was a transmission of an even number of bits or to 1 when the number of 1s was odd. Obviously this can only detect single bit changes in received data a two-bit change would be undetected. Generally most error detection is done on larger blocks of data and at higher levels in protocols.

Programming the USART

In order to program the USART we can use the ST peripheral library code STM32F10x_StdPeriph_Lib_V3.5.0. These calls will program the above registers however we could do it manually ourselves. As with the earlier peripherals we need to do a number of set up tasks before we can do the actual work at hand. These tasks are a little bit more complicated than with the switches and LEDs but not massively more so.

Firstly we need to have a couple of variables to hold the data structures for the ST libraries and then we need to set up the clocks using the Reset and Clock Control (RCC).

```
USART_InitTypeDef USART_InitStructure;
GPIO_InitTypeDef GPIO_InitStructure;

/* Enable GPIO clock */
RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE );

/* Enable USART2 clock */
RCC_APB1PeriphClockCmd( RCC_APB1Periph_USART2, ENABLE );

/* Remap USART, as USART2 uses alternate pins */
GPIO_PinRemapConfig( GPIO_Remap_USART2, ENABLE );
```

The Clock setting code

From the previous architecture diagram, we can see that the USART clock is on peripheral bus 1 – APB1, and the APB2 bus is set up as we are going to use the GPIO pins on GPIOD in their alternate functions (AFIO), as these pins are being used for the receive (RX) and transmit (TX) lines.

```
/* Configure USART2 Tx pin */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; /*changed from 8 to 5 for USART 2 craig */
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init( GPIOD, &GPIO_InitStructure );

/* Configure USART2 Rx pin */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6; /* changed from 9 to 6 for USART 2 craig */
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init( GPIOD, &GPIO_InitStructure );

/* Configure USART 8N1 */
USART_InitStructure.USART_BaudRate = mainCOM_PORT_BAUD_RATE;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init( USART2, &USART_InitStructure );
```

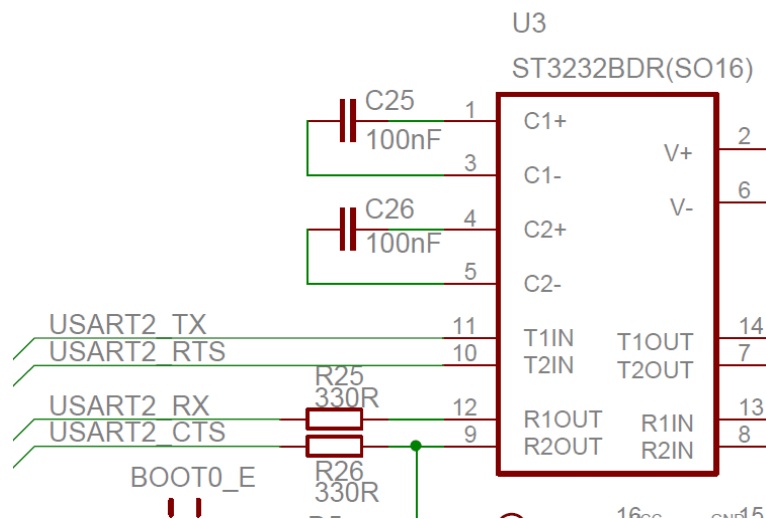
USART initialisation code

Finally, we need to call the function to actually commit the changes to the USART.

```
/* Enable USART */  
USART_Cmd( USART2, ENABLE );
```

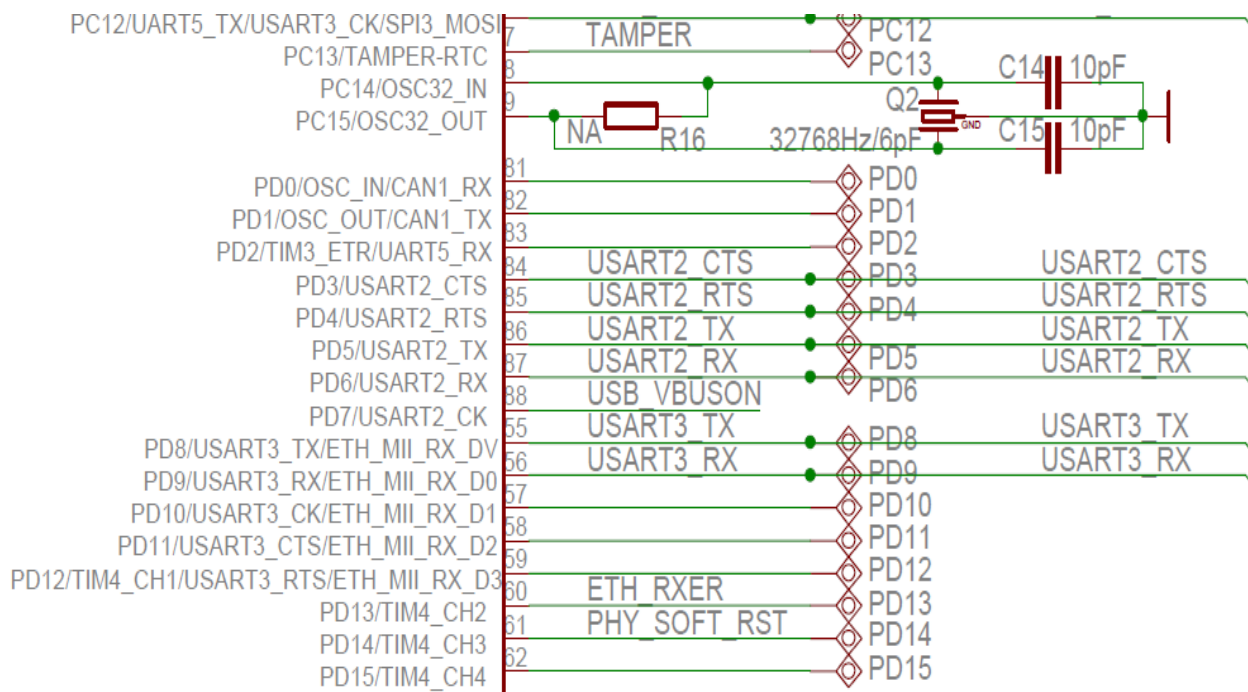
How do we know which pins to set up? Well the older version A used USART1, which was on another internal bus – APB2, and the USART used different pins. If we look at the schematic (STM32-P107-REV-B-SCH.pdf) from Olimex, the company that built the board we can see from that which pins are being used. First looking at the section that deals with the RSR-232.

RS232



The RS232 section of the schematic

We can see that the transmission lines are called USART2_TX and USART2_RX. If we then look at the CPU schematic we can see what pins these are connected too. The bottom corner of the schematic has only been shown as the whole thing is rather large.



STM32 Pin out

You will see that USART2_TX and USART2_RX are, respectively connected to PD5 and PD6, hence these are the pins we need to reassign.

Once this has been completed the USART is ready to receive and transmit. All we need is some code to do this. The transmit code is pretty simple,

```

int __io_putchar(int c) {
    /* Wait until ready to send */
    while (USART_GetFlagStatus(USART2, USART_FLAG_TXE) == RESET)
    {
    }

    USART_SendData(USART2, (u16) c);

    return c;
}

```

USART __io_putchar routine

The routine is very basic – it simply polls the status register until the transmit enable flag is reset, meaning that the USART is ready to transmit. This is of course quite dangerous as the code will loop forever if the USART never becomes ready, at best it will waste a lot of CPU time waiting for the USART. Then it just places the data into the transmit register, although the register is 32 bits only the bottom 8 are sent. The routine returns the value that it sent.

Note on using serial ports.

If you are using VMware then you may have problems using the serial ports in 2q53. If the VMware host is Windows then the driver may well not work correctly and you will get weird results. Unfortunately using a USB to serial adapter works no better on windows as windows hides the device from VMware! Therefore it is best to use the native Linux build, currently Ubuntu. This will handle the serial port correctly and also the USB to serial connector. However our Linux set up is slightly infected by contact with windows via Open Directory. Normally the administrator would set the user to the group dialout or tty to allow them permissions to access the serial devices in /dev – normally /dev/ttyS5 or /dev/ttyUSB0. For some reason our administrator can't do this. That means opening ports can be tricky. The default has been set to /dev/ttyS5 – the default for our serial devices. If this doesn't work then typing in

%minicom -os

Will put you in to the configuration menu – you can then select serial port set up and select the device you want – ttyUSB0 for example. Unfortunately, you have to do this every time you call minicom. Alternatively, you can use command line flags to set up interaction, for examples

%minicom -o -D /dev/ttyUSB0 -b 9600 -8

This will call minicom with the device /dev/ttyUSB0 running at 9600 baud with 8 data bits – the **-o** turns off any modem control signal because we don't want them. For full details of the flags look at the minicom man page. To find out which serial devices your kernel is working with call the kernel diagnostic messages (**dmesg**) and filter, using the **grep** command for tty devices;

%dmesg | grep tty

[0.000000] console [tty0] enabled

[1.009446] 00:06: ttyS0 at I/O 0x3f8 (irq = 4, base_baud = 115200) is a 16550A

[1.030892] 0000:00:16.3: ttyS4 at I/O 0xf140 (irq = 17, base_baud = 115200) is a 16550A

It is worth noting that some of the cheaper USB serial devices can be quite flaky at higher speeds, if in doubt start off slowly and speed up.

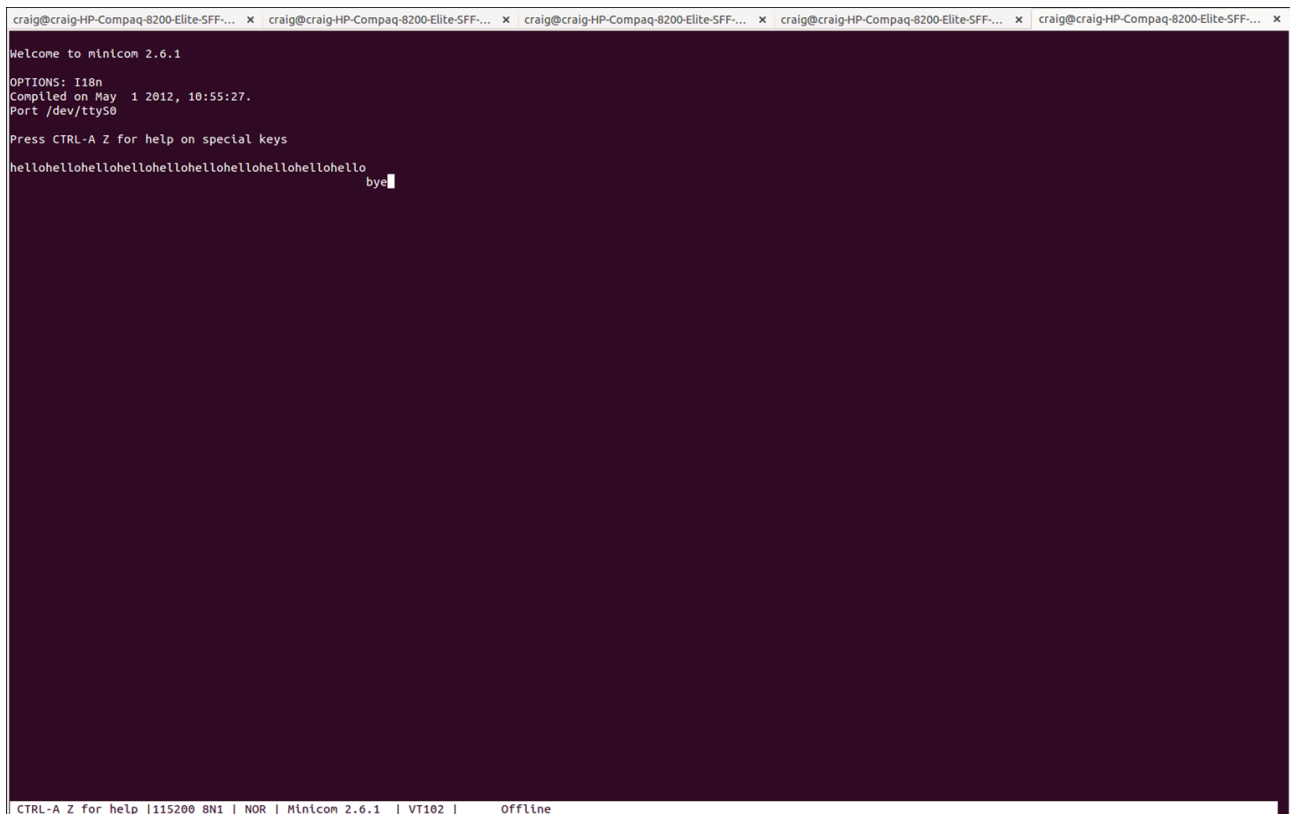
In order to test this routine we will need to connect a serial cable from the PC's serial port to the serial port on the Olimex board. This will require a male to female RS232 cable, of which there are some in the lab.



The male end connects to the 9 pin connector on the Olimex board and the female end to the 9 pin connector at the back of the PC. If you are using a USB to serial connector then see the appendix for details on how to use that. Once the cables are connected up we need a terminal program on the Linux host to receive the data. The program we will use is called **minicom**. If you type **minicom -o ttyS0** that should give you a connection to serial port 1 on your PC. You should see something like the following screen.

```
craig@craig-HP-Compaq-8200-Elite-SFF... x craig@craig-HP-Compaq-8200-Elite-SFF... x craig@craig-HP-Compaq-8200-Elite-SFF... x craig@craig-HP-Compaq-8200-Elite-SFF... x craig@craig-HP-Compaq-8200-Elite-SFF... x
Welcome to minicom 2.6.1
OPTIONS: I18n
Compiled on May 1 2012, 10:55:27.
Port /dev/ttyS0
Press CTRL-A Z for help on special keys
█

CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.6.1 | VT102 | Offline
```



The screenshot shows a terminal window with a dark purple background. The text displayed is as follows:

```
craig@craig-HP-Compaq-8200-Elite-SFF... x | craig@craig-HP-Compaq-8200-Elite-SFF... x | craig@craig-HP-Compaq-8200-Elite-SFF... x | craig@craig-HP-Compaq-8200-Elite-SFF... x | craig@craig-HP-Compaq-8200-Elite-SFF... x |
Welcome to minicom 2.6.1
OPTIONS: I18n
Compiled on May 1 2012, 10:55:27.
Port /dev/ttyS0
Press CTRL-A Z for help on special keys
hellohellohellohellohellohellohellohellohellohello
bye
```

The status bar at the bottom of the terminal window reads: CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.6.1 | VT102 | Offline

Program output.

Exercises.

- 1) Change the baud rate of the program to 9600 – see what happens with minicom. Then change minicom to accept 9600 baud rate.

How did we write the `__io_putchar()` routine and how do we write the `__io_getchar()` routine?

Well the first thing was to make it consistent with the standard C library – libc. To find out the parameters and return values we could look in the manual for the call. We can do this by typing *man putchar* at the Linux command line – the same can be achieved by typing *man putchar* into Google. This will tell us the format and structure of the putchar library call – it tells us

```
int putchar(int c);
```

.....

putchar(c); is equivalent to **putc(c,stdout).**

So putchar returns an int value and takes an int parameter – we will find out what stdout means later in worksheet 6. If we ask what the getchar routine is the we get

```
int getchar(void);
```

.....

getchar() is equivalent to **getc(stdin).**

Again we will find out about stdin in worksheet 6. But we can see that getchar doesn't take a parameter and returns an integer.

Once we get to the internals of `__io_putchar()` then we see that this is using the ST Micro library code – assuming that the port has been initialised, then we can poll the port with the code

```
While (USART_GetFlagStatus(USART2,USART_FLAG_TXE) == RESET)
```

The while loop just loop, possibly infinitely, waiting for the transmit flag to indicate that some data is ready to be read. If it does appear then we can read it with

```
USART_SendData(USART2, (u16) c);
```

The details of these functions can be found in the library files – so if you look in the file `stm32f10x_usart.c` you will find the `USART_SendData` function and just after it there is the `USART_ReceiveData` function. The code and comments will tell you how to call the functions and what values to send and receive. A good way of browsing the code is through gitlab.

Exercise

- 2) Write an inbyte routine. Write a main function that accepts some characters from the keyboard and then writes them out to the screen.

Credit exercise.

- 3) Write a serial port receive routine which implements full error handling – for overrun, noise framing and parity errors. You can either print out error messages or flash warning LEDs. Demonstrate by sending deliberately corrupted data.

Appendix

main.c serial port code

```
#include "com_port.h"

#include <stm32f10x.h>
#include <stm32f10x_rcc.h>
#include <stm32f10x_gpio.h>
#include <stm32f10x_usart.h>

int __io_putchar(int c) {
    /* Wait until ready to send */
    while (USART_GetFlagStatus(USART2, USART_FLAG_TXE) == RESET)
    {
    }

    USART_SendData(USART2, (u16) c);

    return c;
}

void COMPortInit ( void ) {
    USART_InitTypeDef USART_InitStructure;
    GPIO_InitTypeDef GPIO_InitStructure;

    /* Enable GPIO clock */
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIOID | RCC_APB2Periph_AFIO, ENABLE );

    /* Enable USART2 clock */
    RCC_APB1PeriphClockCmd( RCC_APB1Periph_USART2, ENABLE );

    /* Remap USART, as USART2 uses alternate pins */
    GPIO_PinRemapConfig( GPIO_Remap_USART2, ENABLE );

    /* Configure USART2 Tx pin */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5; /*changed 8 to 5 for USART2 craig */
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_Init( GPIOID, &GPIO_InitStructure );

    /* Configure USART2 Rx pin */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6; /*changed 9 to 6 for USART2 craig */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init( GPIOID, &GPIO_InitStructure );

    /* Configure USART 8N1 */
    USART_InitStructure.USART_BaudRate = mainCOM_PORT_BAUD_RATE;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
```

```
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init( USART2, &USART_InitStructure );

/* Enable USART */
USART_Cmd( USART2, ENABLE );
}

int main(void)
{
    int i;
    COMPortInit();
    for ( i=0; i != 10; i++)
    {
        __io_putchar('h');
        __io_putchar('e');
        __io_putchar('l');
        __io_putchar('l');
        __io_putchar('o');
    }

    __io_putchar('\n');
    __io_putchar('b');
    __io_putchar('y');
    __io_putchar('e');
}

#ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number
 *        where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */

void assert_failed(uint8_t* file, uint32_t line)
{
    /* User can add his own implementation to report the file name
    and line number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

    /* Infinite loop */
    while (1)
    {
    }
}

#endif

/**
 * @}
 */

/**
 * @}
 */

/***** (C) COPYRIGHT 2011 STMicroelectronics *****END OF FILE*****/
```

