Title:	Serial data communication Part 1
Author:	Craig Duffy 1/11/23
Module:	Embedded Systems Programming
Awards:	BSc Computer Science.
Prerequisites:	Basic computer architecture and some C

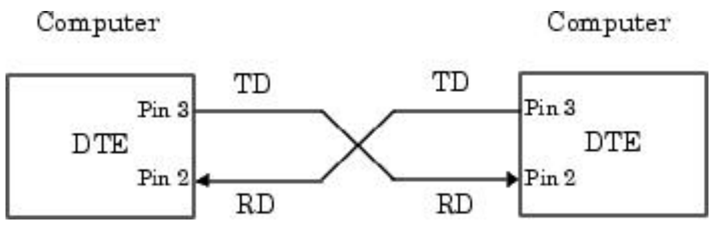
Having 1 bit input and output won't get us very far so we need to look a bit farther afield to get something more useful. Firstly, we will look at basic serial communications.

Ye Olde RS-232

In computing terms basic serial, or RS-232 comms is pretty ancient – the original standard dates back to 1960! 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.

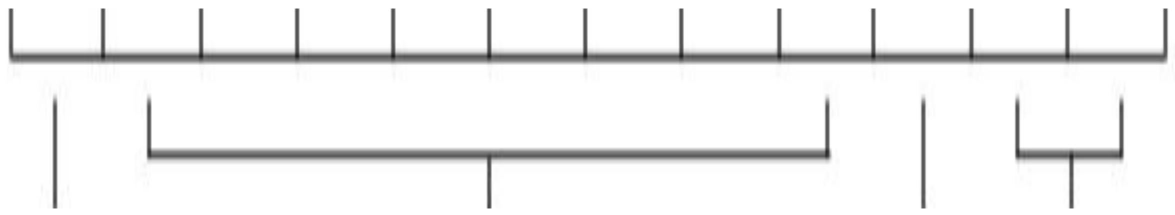
RS-232 does have its limitations, and this is part of the reason for its reduced usage. RS-232 data rates are relatively slow, the connectors are large, cable lengths are short and it is difficult to have multiple end points. However as a well understood, simple and reliable form of communication RS-232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

The ARM Cortex M4 series come with a maximum of 6 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 Discovery board has 5 USARTs and a low powered UART (LPUART) 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. It is normal for both machines to share a ground connection too – although that could be supplied via the (common) power source. 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.

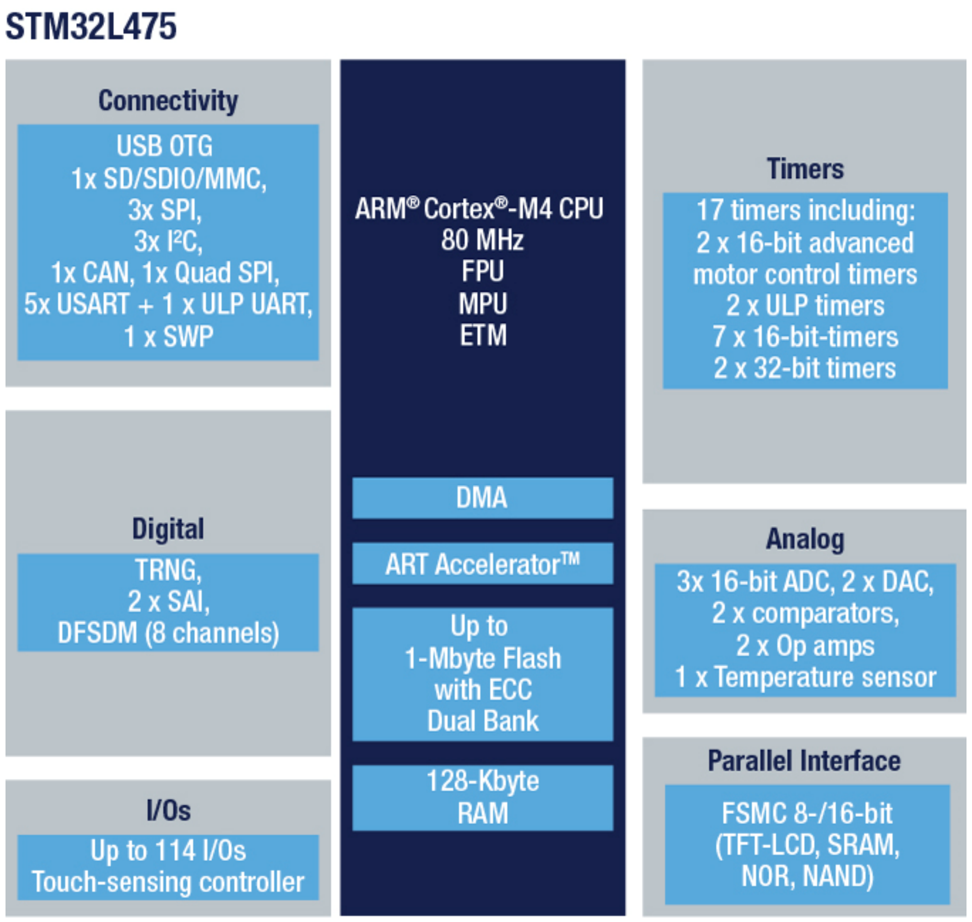

Serial data frame format.

The order of the data transmission is from left to right – so it starts with the start bit. 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.

In this worksheet we will be only looking at polled asynchronous data transfers so although the term USART, for a synchronous receiver/transmitter is referenced we wont be using the clock lines or ant other synchronous features.

USARTs on the STM32

The STM32 has been designed to support 5 USARTs, plus 1 low powered UART (LPUART) although individual designs may implement fewer devices. The diagram below shows the architecture of our device, the STM32L475.



STM32L4xx Block Diagram

The USART devices are, like all Cortex M4 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
0x4000 8000 – 0x4000 83FF	LPUART	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 address given above. These features make it easier for the software developer to create reasonably portable software fairly quickly. The registers are:

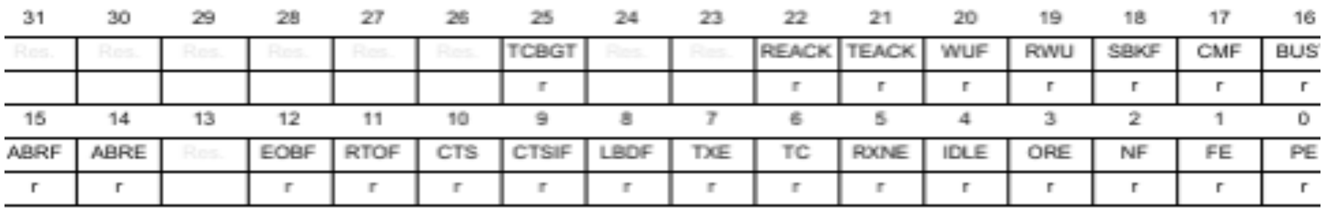
Name	Label	Offset	R/W	Reset
Control register 1	USART_CR1	0x00	W	0x0000
Control register 2	USART_CR2	0x04	W	0x0000
Control register 3	USART_CR3	0x08	W	0x0000
Baud rate register	USART_BRR	0x0C	W	0x0000
Guard time and prescaler register	USART_GTPR	0x10	R/W	0x0000
Receiver timeout register	USART_RTOR	0x14	W	0x0000
Request register	USART_RQR	0x18	W	0x0000
Interrupt and status register	USART_ISR	0x1C	R	0x0200 00C0
interrupt flag clear register	USART_ICR	0x20	W	0x0000
receive data register	USART_RDR	0x24	R	0x0000
transmit data register	USART_TDR	0x28	W	0x0000

USART registers and offsets.

Although the registers are 32 bits long often only some of those bits are used – for example the data transfers are only a maximum of 9 bits of data so the other bits are not used. These 'extra' bits of reserved as they maybe used on later Cortex models to implement new features. Some of the names of the registers 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 receive and transmit registers allow us to either send data (USART_TDR) or read the received data (USART_RDR). 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 a large number of settings, most of which we won't need. Also the peripherals can work in a number of modes, as interrupt driven devices or using DMA (direct memory access) for example. At the moment we will be using the device 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



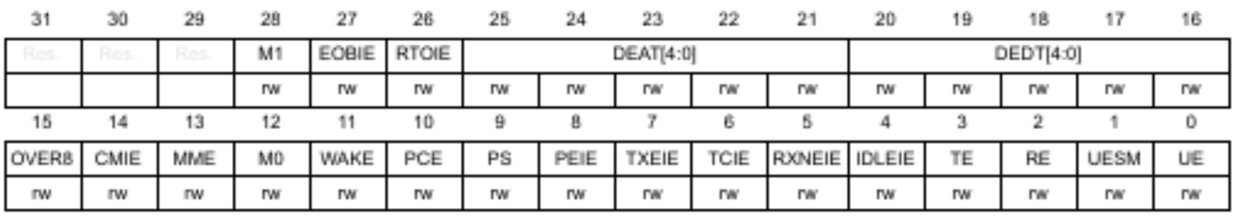
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	TCBGT	Res.	Res.	REACK	TEACK	WUF	RWU	SBKF	CMF	BUS
						r			r	r	r	r	r	r	r
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
ABRF	ABRE	Res.	EOBF	RTOF	CTS	CTSIF	LBDF	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
r	r		r	r	r	r	r	r	r	r	r	r	r	r	r

The status register format

The status register tells the programmer what happened during the last transmit or receive as well as tells us whether or not there is data incoming or the possibility to send data. The bottom 4 bits (0:3) are all error bits, for Parity PE (single byte transmission), frame FE (multi byte transmission), noise 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).

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
Res.	Res.	Res.	M1	EOBIE	RTOIE	DEAT[4:0]					DEDT[4:0]				
			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
OVER8	CMIE	MME	M0	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	UESM	UE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The control register 1 format

The main bits we need to know about are the UE (0), 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

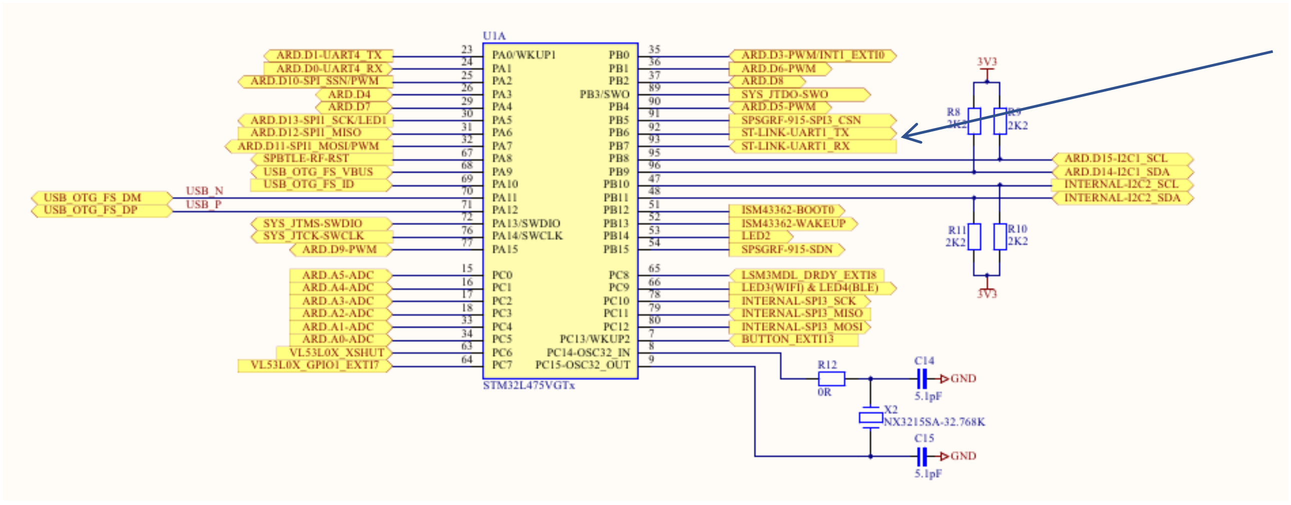
In order to program the USART we can use the ST peripheral library code STM32L4xx_HAL_Driver. These calls will program the above registers however we could do it manually ourselves. As with the earlier peripherals we need to do a number of set up tasks before we can do the actual work at hand. These tasks are a little bit more complicated than with the switches and LEDs but not massively more so.

First 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).

```
#define DISCOVERY_COM1                    USART1
#define DISCOVERY_COM1_CLK_ENABLE()          __HAL_RCC_USART1_CLK_ENABLE()
#define DISCOVERY_COM1_CLK_DISABLE()         __HAL_RCC_USART1_CLK_DISABLE()


#define DISCOVERY_COM1_TX_PIN               GPIO_PIN_6
#define DISCOVERY_COM1_TX_GPIO_PORT          GPIOB
#define DISCOVERY_COM1_TX_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define DISCOVERY_COM1_TX_GPIO_CLK_DISABLE()   __HAL_RCC_GPIOB_CLK_DISABLE()
#define DISCOVERY_COM1_TX_AF                GPIO_AF7_USART1

#define DISCOVERY_COM1_RX_PIN               GPIO_PIN_7
#define DISCOVERY_COM1_RX_GPIO_PORT          GPIOB
#define DISCOVERY_COM1_RX_GPIO_CLK_ENABLE()    __HAL_RCC_GPIOB_CLK_ENABLE()
#define DISCOVERY_COM1_RX_GPIO_CLK_DISABLE()   __HAL_RCC_GPIOB_CLK_DISABLE()
#define DISCOVERY_COM1_RX_AF                GPIO_AF7_USART1

UART_HandleTypeDef hDiscoUart;

……..
void BSP_COM_Init(UART_HandleTypeDef *huart)
{
  GPIO_InitTypeDef gpio_init_structure;

 GPIO_InitTypeDef gpio_init_structure;

  /* Enable GPIO RX/TX clocks */
  DISCOVERY_COM1_TX_GPIO_CLK_ENABLE();
  DISCOVERY_COM1_RX_GPIO_CLK_ENABLE();

  /* Enable USART clock */
  DISCOVERY_COM1_CLK_ENABLE();
```

The Clock setting code

From the previous architecture diagram we can see that the USART clock is on peripheral bus 2 – APB2, the RCC clock functions set up the USART clock - the __HAL_RCC_USART1_CLK_ENABLE is in fact a macro in Drivers/STM32L4xx_HAL_Driver/Inc/stm32l4xx_hal_rcc.h. The RX and TX lines are actually using pins 6 and 7 from GPIO port B, so those clocks need starting too – again these functions are really macros in the previously mentioned file. The actual pins and ports can all be found in the Discovery manual but they can also be found in the schematic which is at the end of that document.



The schematic showing the pin assignments.

```
/* Initialize all configured peripherals */
   hDiscoUart.Instance = DISCOVERY_COM1;
   hDiscoUart.Init.BaudRate = 115200;
   hDiscoUart.Init.WordLength = UART_WORDLENGTH_8B;
   hDiscoUart.Init.StopBits = UART_STOPBITS_1;
   hDiscoUart.Init.Parity = UART_PARITY_NONE;
   hDiscoUart.Init.Mode = UART_MODE_TX_RX;
   hDiscoUart.Init.HwFlowCtl = UART_HWCONTROL_NONE;
   hDiscoUart.Init.OverSampling = UART_OVERSAMPLING_16;
   hDiscoUart.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
   hDiscoUart.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;

   BSP_COM_Init(&hDiscoUart);


.........
/* Configure USART Tx as alternate function */
  gpio_init_structure.Pin = DISCOVERY_COM1_TX_PIN;
  gpio_init_structure.Mode = GPIO_MODE_AF_PP;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_HIGH;
  gpio_init_structure.Pull = GPIO_NOPULL;
  gpio_init_structure.Alternate = DISCOVERY_COM1_TX_AF;
  HAL_GPIO_Init(DISCOVERY_COM1_TX_GPIO_PORT, &gpio_init_structure);

  /* Configure USART Rx as alternate function */
  gpio_init_structure.Pin = DISCOVERY_COM1_RX_PIN;
  gpio_init_structure.Mode = GPIO_MODE_AF_PP;
  gpio_init_structure.Alternate = DISCOVERY_COM1_RX_AF;
  HAL_GPIO_Init(DISCOVERY_COM1_RX_GPIO_PORT, &gpio_init_structure);
```

USART initialisation code

The settings for the actual serial port are sent to the HAL_GPIO_Init function, after having been put into a structure, called UART_HandleTypeDef, which holds all of the settings.   This structure is defined in Drivers/STM32L4xx_HAL_Driver/Inc/stm32l4xx_hal_uart.h. Once all of the settings have been passed to the device and pushed into its control register it can be enabled.

```
/* USART configuration */
  huart->Instance = DISCOVERY_COM1;
  HAL_UART_Init(huart);
```

The routine to output the characters is fairly simple

```
int putchar(int ch)
{
 /* e.g. write a character to the serial port and Loop until the end of transmission */
 while (HAL_OK != HAL_UART_Transmit(&hDiscoUart, (uint8_t *) &ch, 1, 30000))
 {
  ;
 }
 return ch;
}
```

The HAL_UART_Transmit function needs to know which UART – this is passed in the hDiscoUart structure, it needs the character, in the variable ch, it needs to know how many characters – 1 in this case and finally there is a timeout value to stop the function from holding up main program. The HAL_UART_Transmit code is in Drivers/STM32L4xx_HAL_Driver/Src/stm32l4xx_hal_uart.c. The putchar() routine returns the same value as it sends, which is standard for UNIX putchar routines – see the manual page for putchar *man putchar* from the command line.

To run the program you will need to connect to the board with *openocd* and *telnet* as usual but also this time you will need to connect to the terminal using *minicom* – the settings for baud and so on are the same as in worksheet 1 so you can use the command *minicom -o IoTBoard* to connect.

Exercises

0. Run the program and test it out – try changing the text.
1. Change some of the UART settings – baud rate, parity and so on. See what happens when you run it with *minicom -o IoTBoard*. Then reset the minicom settings so that you can see the output.
2. Write the getchar() routine. This is almost identical to the putchar() routine except it uses HAL_UART_Receive instead of HAL_UART_Transmit. The standard Unix/Linux Getchar returns an int – the character read but it doesn't take a parameter. The HAL_UART_Receive library function is slightly different in that it takes a parameter, the address of the variable to be read into. It also returns a status value this can let you know if there has been an error – a timeout for example but you are hoping for a HAL_OK which means it worked and data has been returned. In order to maintain consistency with standard C libraries you will need to write a function which takes no parameters and returns the character read – see *man* getchar for details. To show it works in the main program change the loop in main to allow you to input a character and output it a number of times – you can use the LEDs to help you debug.

Note on using serial ports.

If you are using VMware then you may have problems using the serial ports in some rooms. 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/tyyUSBS0. 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 – ttyUSBS0 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

```
%mincom -o -D /dev/ttyUSBS0 -b 9600 -8
```

This will call minicom with the device /dev/ttyUSBS0 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 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 of slowly and speed up.