EEET2490 – Embedded System: OS and Interfacing

Lab 03 – UART Serial Port Programming

Objectives

Previously, our bare metal OS has been successfully booted up and got into the main() function. To continue the development of our OS, we need to work on peripherals and bring up their functionality. Specifically, in this lab, we will initialize and work with serial port (UART) of the Raspberry Pi board. With this peripheral, we can start transferring data in and out, for example, a simple “hello world” message is possible.

Hardware Platform

The tutorial series is written for Raspberry Pi 3/4 B+ board. The main difference is physical peripheral base address, which is 0x3F000000 in Raspberry 3, and 0xFE000000 in Raspberry 4.

Software Requirements

Make sure that you have successfully completed Tutorial 02 and have the following tools working properly:

- **Visual Studio Code** editor
- **GnuWin32** make tool
- **aarch64-none-elf** compiler
- **QEMU** emulation tool

Lab Technical Materials and References

Following are the main documents for datasheet and technical references:

- **BCM2837 ARM Peripherals - Pi 3**
- **BCM2711 ARM Peripherals - Pi 4**
- **ARM Programmer Guide** (for ARMv8-A)

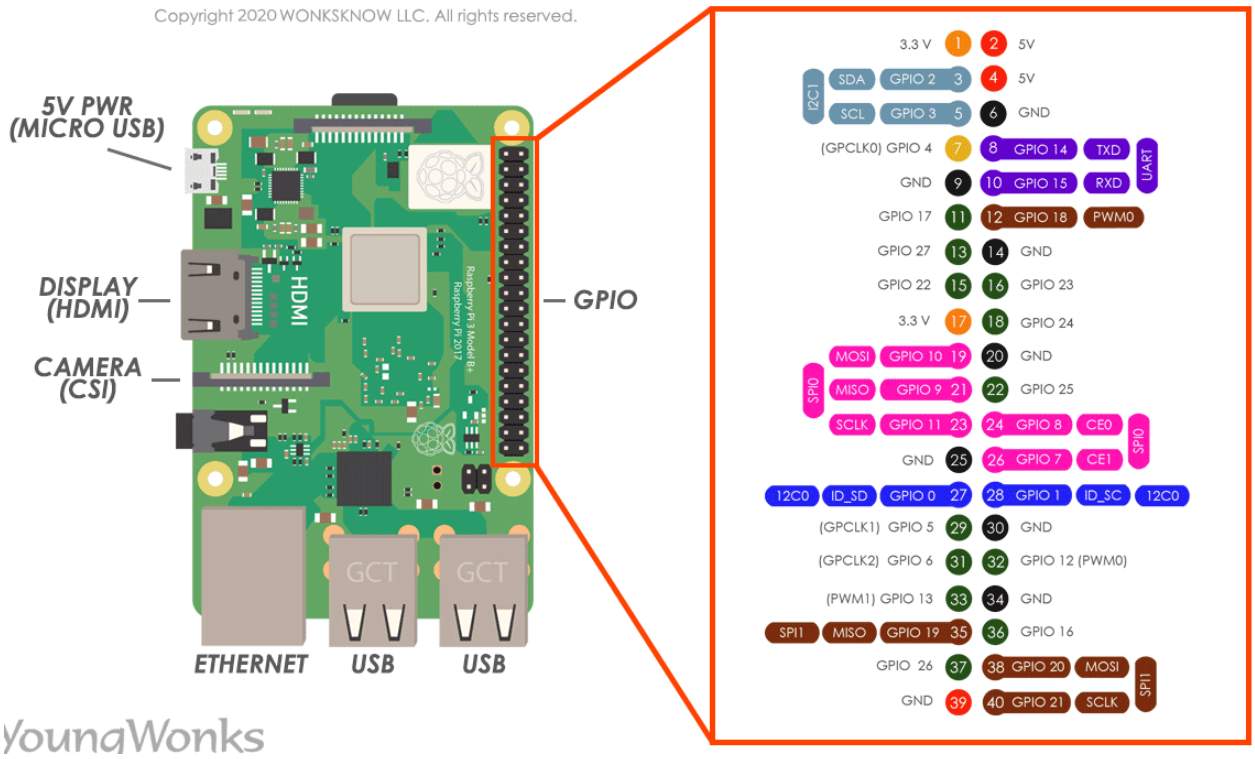
All documents are available on Canvas at **Board Documents** page.

LAB ACTIVITIES

1. Brief Info of the UART

The serial port of Raspberry Pi is implemented as UART function with RS232 standard interface.

By default, it is available on pin GPIO 14 and 15 (UART1/0) as shown in the pin out below:



There are two types of UART available on the Raspberry Pi 3/4 - PL011 and mini UART. The PL011 is a capable, broadly 16550-compatible UART, while the mini UART has a reduced feature set. *We will work with mini UART (UART1) first since it is easier to program.*

NOTE: All UARTs on the Raspberry Pi are 3.3V only - damage will occur if they are connected to 5V systems. An adaptor can be used to connect to 5V systems. Alternatively, low-cost USB to 3.3V serial adaptors are available from various third parties.

Name	Type	Board
UART0	PL011	RPI3/4
UART1	**mini UART**	RPI3/4
UART2	PL011	RPI4 only
UART3	PL011	RPI4 only
UART4	PL011	RPI4 only
UART5	PL011	RPI4 only

Beside the default pins GPIO14 and GPIO15, the UART modules can also be mapped on other GPIO pins by configuring the corresponding alternative functions for the GPIOs pins.

For example, as shown in the table below for RPI4, the UART4 (TXD4 and RXD4) can be mapped to GPIO8 and GPIO9 pins, and UART5 (TXD5 and RXD5) can be mapped to GPIO12 and GPIO13 pins.

GPIO	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO5	High	GPCLK1	SA0	DPI_D1	SPI4_MISO	RXD3	SCL3
GPIO6	High	GPCLK2	SOE_N / SE	DPI_D2	SPI4_MOSI	CTS3	SDA4
GPIO7	High	SPI0_CE1_N	SWE_N / SRW_N	DPI_D3	SPI4_SCLK	RTS3	SCL4
GPIO8	High	SPI0_CE0_N	SD0	DPI_D4	BSCSL / CE_N	TXD4	SDA4
GPIO9	Low	SPI0_MISO	SD1	DPI_D5	BSCSL / MISO	RXD4	SCL4
GPIO10	Low	SPI0_MOSI	SD2	DPI_D6	BSCSL SDA / MOSI	CTS4	SDA5
GPIO11	Low	SPI0_SCLK	SD3	DPI_D7	BSCSL SCL / SCLK	RTS4	SCL5
GPIO12	Low	PWM0_0	SD4	DPI_D8	SPI5_CE0_N	TXD5	SDA5
GPIO13	Low	PWM0_1	SD5	DPI_D9	SPI5_MISO	RXD5	SCL5
GPIO14	Low	TXD0	SD6	DPI_D10	SPI5_MOSI	CTS5	TXD1
GPIO15	Low	RXD0	SD7	DPI_D11	SPI5_SCLK	RTS5	RXD1
GPIO16	Low	<reserved>	SD8	DPI_D12	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	DPI_D13	RTS0	SPI1_CE1_N	RTS1

*Example View of **Alternative Function Assignments** on RPI4 board (page 77).*

*Note: You can refer to the **ARM Peripherals Guide (RPI3/RPI4)** of the board for detailed **Alternative Function Assignments** of all GPIO pins.*

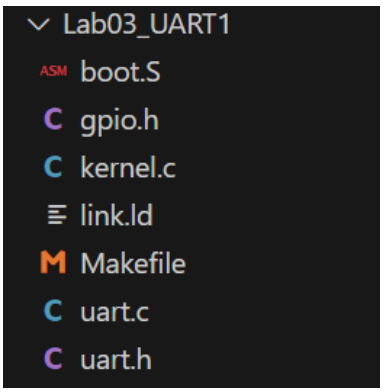
2. UART1 and “Hello World!” program

In this lab, we will start with UART1 (miniUART) because it is simpler. Follow the steps below:

a. Prepare your project folder

- Create a **new folder** within your workspace named “**Lab03_UART1**”.
- Reuse previous startup code, linker script and Makefile from Lab02 by copying “**boot.S**”, “**kernel.c**”, “**linker.ld**” and “**Makefile**” to the new folder.
- Create three new files named “**gpio.h**”, “**uart.h**” (header files for gpio and uart registers), and “**uart.c**” (driver for uart) within the folder.

  Your project folder within VS Code should look like below:



b. Header files

- Same as for Lab02, the header file "**gpio.h**" contains the registers' addresses for GPIO configuration. In fact, you can also reuse the "gpio.h" file in your Lab02. However, read the note below for more explanation of the file's content.

```
// ------------------------------------gpio.h ------------------------------------

/* Raspberry Pi's peripheral physical address (MMIO_BASE) is 0xFE000000 in RPI4,
0x3F000000 in RPI3
--> Select correct option to set the value properly
*/

#define RPI3 //enable when using RPI3 (QEMU emulation/ real board)

#ifdef RPI3 //RPI3
    #define MMIO_BASE       0x3F000000
#else //RPI4
    #define MMIO_BASE       0xFE000000
#endif

//Define GPIO Registers based on their addresses
#define GPFSEL0         (* (volatile unsigned int*)(MMIO_BASE+0x00200000))
#define GPFSEL1         (* (volatile unsigned int*)(MMIO_BASE+0x00200004))
#define GPFSEL2         (* (volatile unsigned int*)(MMIO_BASE+0x00200008))
#define GPFSEL3         (* (volatile unsigned int*)(MMIO_BASE+0x0020000C))
#define GPFSEL4         (* (volatile unsigned int*)(MMIO_BASE+0x00200010))
#define GPFSEL5         (* (volatile unsigned int*)(MMIO_BASE+0x00200014))
#define GPSET0          (* (volatile unsigned int*)(MMIO_BASE+0x0020001C))
#define GPSET1          (* (volatile unsigned int*)(MMIO_BASE+0x00200020))
#define GPCLR0          (* (volatile unsigned int*)(MMIO_BASE+0x00200028))
#define GPLEV0          (* (volatile unsigned int*)(MMIO_BASE+0x00200034))
#define GPLEV1          (* (volatile unsigned int*)(MMIO_BASE+0x00200038))
#define GPEDS0          (* (volatile unsigned int*)(MMIO_BASE+0x00200040))
#define GPEDS1          (* (volatile unsigned int*)(MMIO_BASE+0x00200044))
#define GPHEN0          (* (volatile unsigned int*)(MMIO_BASE+0x00200064))
#define GPHEN1          (* (volatile unsigned int*)(MMIO_BASE+0x00200068))
#define GPPUD           (* (volatile unsigned int*)(MMIO_BASE+0x00200094))
#define GPPUDCLK0       (* (volatile unsigned int*)(MMIO_BASE+0x00200098))
#define GPPUDCLK1       (* (volatile unsigned int*)(MMIO_BASE+0x0020009C))
```

Note:

All the GPIO register names and their addresses can be found in GPIO section of the ARM Peripherals Guide (page 89 for RPI3, and page 65 for RPI4).

- Pi4 boots into Low Peripheral mode by default. Thus, the registers' addresses are started at base **0xFE000000** (instead of legacy address 0x7E000000 in the document). Similarly, for RPI3 the registers' addresses are started at base address **0x3F000000** (instead of bus address 0x7E000000 in the document).
- The `volatile` keyword is to inform the compiler that those are hardware related variables/ addresses, whose values can be changed by hardware operation and thus, should not be optimized at compilation process.
- The `unsigned int*` is to specify that they are *pointers of unsigned 32-bit data type*, which are actually the GPIO registers' addresses.
- Finally, we dereference the address by * operator to access the register's value through its address.

- Similarly, for the “uart.h” header file, we can write the registers’ addresses for configuration of UART1 as below:

```
// ------------------------------------uart.h ------------------------------------
/* Auxilary mini UART (UART1) registers */
#define AUX_ENABLE      (* (volatile unsigned int*)(MMIO_BASE+0x00215004))
#define AUX_MU_IO       (* (volatile unsigned int*)(MMIO_BASE+0x00215040))
#define AUX_MU_IER      (* (volatile unsigned int*)(MMIO_BASE+0x00215044))
#define AUX_MU_IIR      (* (volatile unsigned int*)(MMIO_BASE+0x00215048))
#define AUX_MU_LCR      (* (volatile unsigned int*)(MMIO_BASE+0x0021504C))
#define AUX_MU_MCR      (* (volatile unsigned int*)(MMIO_BASE+0x00215050))
#define AUX_MU_LSR      (* (volatile unsigned int*)(MMIO_BASE+0x00215054))
#define AUX_MU_MSR      (* (volatile unsigned int*)(MMIO_BASE+0x00215058))
#define AUX_MU_SCRATCH  (* (volatile unsigned int*)(MMIO_BASE+0x0021505C))
#define AUX_MU_CNTL     (* (volatile unsigned int*)(MMIO_BASE+0x00215060))
#define AUX_MU_STAT     (* (volatile unsigned int*)(MMIO_BASE+0x00215064))
#define AUX_MU_BAUD     (* (volatile unsigned int*)(MMIO_BASE+0x00215068))

/* Function prototypes */
void uart_init();
void uart_sendc(char c);
char uart_getc();
void uart_puts(char *s);
```

All the relevant registers for UART1 can be found in section **2. Auxiliaries: UART1, SPI1 & SPI2** in the ARM Peripherals Guide.

c. Write our uart1 driver (uart.c file):

```
// ------------------------------------uart.c ------------------------------------

#include "uart.h"
#include "gpio.h"

/**
 * Set baud rate and characteristics (115200 8N1) and map to GPIO
 */
void uart_init()
{
    unsigned int r;

    /* initialize UART */
    AUX_ENABLE |= 1;     //enable mini UART (UART1)
    AUX_MU_CNTL = 0;    //stop transmitter and receiver
    AUX_MU_LCR  = 3;     //8-bit mode (also enable bit 1 to be used for RBP3)
    AUX_MU_MCR  = 0;    //clear RTS (request to send)
    AUX_MU_IER  = 0;    //disable interrupts
    AUX_MU_IIR  = 0xc6;  //enable and clear FIFOs
    AUX_MU_BAUD = 270;   //configure 115200 baud [system_clk_freq/(baud_rate*8) - 1]

    /* map UART1 to GPIO pins 14 and 15 */
    r = GPFSEL1;
    r &= ~( (7 << 12)|(7 << 15) ); //clear bits 17-12 (FSEL15, FSEL14)
    r |=    (2 << 12)|(2 << 15);   //set value 2 (select ALT5: TXD1/RXD1)
    GPFSEL1 = r;

    /* enable GPIO 14, 15 */
    GPPUD = 0;            //No pull up/down control
    r = 150; while(r--) { asm volatile("nop"); } //waiting 150 cycles
    GPPUDCLK0 = (1 << 14)|(1 << 15); //enable clock for GPIO 14, 15
    r = 150; while(r--) { asm volatile("nop"); } //waiting 150 cycles
    GPPUDCLK0 = 0;        //flush GPIO setup

    AUX_MU_CNTL = 3;      //enable transmitter and receiver (Tx, Rx)
}
```

```
/**
 * Send a character
 */
void uart_sendc(char c) {
    // wait until transmitter is empty
    do {
         asm volatile("nop");
    } while ( !(AUX_MU_LSR & 0x20) );

    // write the character to the buffer
    AUX_MU_IO = c;
}

/**
 * Receive a character
 */
char uart_getc() {
    char c;

    // wait until data is ready (one symbol)
    do {
         asm volatile("nop");
    } while ( !(AUX_MU_LSR & 0x01) );

    // read it and return
    c = (unsigned char)(AUX_MU_IO);

    // convert carriage return to newline character
    return (c == '\r' ? '\n' : c);
}

/**
 * Display a string
 */
void uart_puts(char *s) {
    while (*s) {
        // convert newline to carriage return + newline
        if (*s == '\n')
            uart_sendc('\r');
        uart_sendc(*s++);
    }
}
```

To elaborate, first of all, we wrote the `uart_init()` to initialize the UART1. Main steps include:

- Enable UART1 in **AUXENB** register.
- Configure **AUX_MU_CNTL** register to stop transmitter and receiver (TX and RX)
- Set operating mode (e.g., 8 bits, no interrupt, 115200 baud) through **AUX_MU_LCR, AUX_MU_MCR, AUX_MU_IER, AUX_MU_BAUD** registers.
- Clear data buffers by writing to **AUX_MU_IIR** register.
- Map UART1 (TXD1, RXD1) to appropriate GPIO pins, e.g. pins 14 and 15, through GPIO alternative function selection (FSEL fields in a **GPFSEL register**).
- Set no pull up/ pull down control, and enable clock for the selected GPIO pins through **GPPUD** and **GPPUDCLK** registers.
- Enable UART1's transmitter and receiver (TX and RX) in **AUX_MU_CNTL** register

Generally, the `uart_sendc, uart_getc` and `uart_puts` are functions to send/receive a char and a string. They will be the basic drivers that we can use in our main function to send/receive data.

Note: you can refer to page 11 of ARM Peripherals guide for **baud rate configuration of UART1** (system_clk_freq is 250MHz by default).

$$baudrate = \frac{system\_clock\_freq}{8*(baudrate\_reg+1)}$$

d. Write the main() function

In the kernel.c file, we can write a simple program as below, which utilize our developed uart driver to print out a "Hello World!" message through the UART serial port and send back twice all characters entered by the user.

```
// ------------------------------------main.c -------------------------------------
#include "uart.h"

void main(){
    // intitialize UART
    uart_init();

    // say hello
    uart_puts("Hello World!\n");

    // echo everything back
    while(1) {
       //read each char
       char c = uart_getc();

       //send back twice
       uart_sendc(c);
       uart_sendc(c);
    }
}
```

3. Build and test with QEMU emulation

First of all, the **Makefile** should almost the same as we did before in Lab 02. Note that it will automatically build all .c files, including the newly added "uart.c" file. The only difference is that you can modify the run target to access the Raspberry Pi's UART serial port through QEMU emulation.

```
#-------------------------------------Makefile-------------------------------------
CFILES = $(wildcard *.c)
OFILES = $(CFILES:.c=.o)
GCCFLAGS = -Wall -O2 -ffreestanding -nostdinc -nostdlib

all: clean kernel8.img

boot.o: boot.S
	aarch64-none-elf-gcc $(GCCFLAGS) -c boot.S -o boot.o

%.o: %.c
	aarch64-none-elf-gcc $(GCCFLAGS) -c $< -o $@

kernel8.img: boot.o $(OFILES)
	aarch64-none-elf-ld -nostdlib boot.o $(OFILES) -T link.ld -o kernel8.elf
	aarch64-none-elf-objcopy -O binary kernel8.elf kernel8.img

clean:
	del kernel8.elf *.o *.img

# Run emulation with QEMU
run:
	qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio
```

The command to run emulation with QEMU is

```
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio
```

Explanation:

-M raspi3 : the first argument tells QEMU to emulate Raspberry Pi 3 hardware.

-kernel kernel8.img : the second argument specifies the kernel filename to be used.

-serial null -serial stdio : the last argument redirects the emulated UART1 to the standard input/output of the terminal running QEMU, so that everything sent to the serial port will be displayed on the terminal, and every key typed in the terminal will be received by the serial port. As UART1 is *not* redirected by default, thus we use two **-serial** arguments (*first argument is for UART0*, and it is null since we don't run it now).

*Now you can simply type "**make**" to build the project, and type "**make run**" to execute the emulation.*

*If all steps are done correctly, you should see a "**Hello World!**" message displayed on the console (sent by our OS on Raspberry board through UART serial port). In addition, all characters typed by the user will be sent back to the console twice as we program in our main() function.*

```
PS D:\WORKSPACE_S2_2023\eOSWorkspace\Lab03_UART1> make
del kernel8.elf *.o *.img
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c boot.S -o boot.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c kernel.c -o kernel.o
aarch64-none-elf-gcc -Wall -O2 -ffreestanding -nostdinc -nostdlib -c uart.c -o uart.o
aarch64-none-elf-ld -nostdlib boot.o kernel.o uart.o -T link.ld -o kernel8.elf
aarch64-none-elf-objcopy -O binary kernel8.elf kernel8.img
PS D:\WORKSPACE_S2_2023\eOSWorkspace\Lab03_UART1> make run
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial null -serial stdio

(qemu-system-aarch64.exe:13764): GdkPixbuf-WARNING **: Cannot open pixbuf loader module file 'C:\Program
ingTools\qemu20181127\lib\gdk-pixbuf-2.0\2.10.0\loaders.cache': No such file or directory

This likely means that your installation is broken.
Try running the command
  gdk-pixbuf-query-loaders > C:\ProgramingTools\qemu20181127\lib\gdk-pixbuf-2.0\2.10.0\loaders.cache
to make things work again for the time being.
Hello World!
aabbcc
```

4. Test our OS on real Raspberry Pi board

- If you are using RPI4, go to the header file **gpio.h** and comment out the line `#define RPI3` (keep other content unchanged). Then type "make" in the terminal to compile the project again for RPI4.

```
// ------------------------------------gpio.h ------------------------------------

/* Raspberry Pi's peripheral physical address (MMIO_BASE) is 0xFE000000 in RPI4,
0x3F000000 in RPI3
--> Select correct option to set the value properly
*/

#define RPI3 //enable when using RPI3 (QEMU emulation/ real board)

#ifdef RPI3 //RPI3
    #define MMIO_BASE       0x3F000000
#else //RPI4
    #define MMIO_BASE       0xFE000000
#endif

.....
```

- Copy our **kernel8.img** file to the microSD card and remove all other *.img files inside. Of course, the microSD card needs to be formatted properly before.

- In order to use the mini UART, we also need to configure the Raspberry Pi to use a fixed VPU core clock frequency. This is because the mini UART clock is linked to the VPU core clock, so that when the core clock frequency changes, the UART baud rate will also change.

  To do so, the **enable_uart** and **core_freq** settings can be added to **config.txt** file inside the microSD card to change the behaviour of the mini UART as below:

```
File: config.txt
Add two additional settings
enable_uart=1
core_freq=250
```

- In addition, we will need an **USB to COM adapter cable** so that we can connect the serial port of raspberry pi board our computer through its USB port. There are two types of USB to COM adapters that you can use:

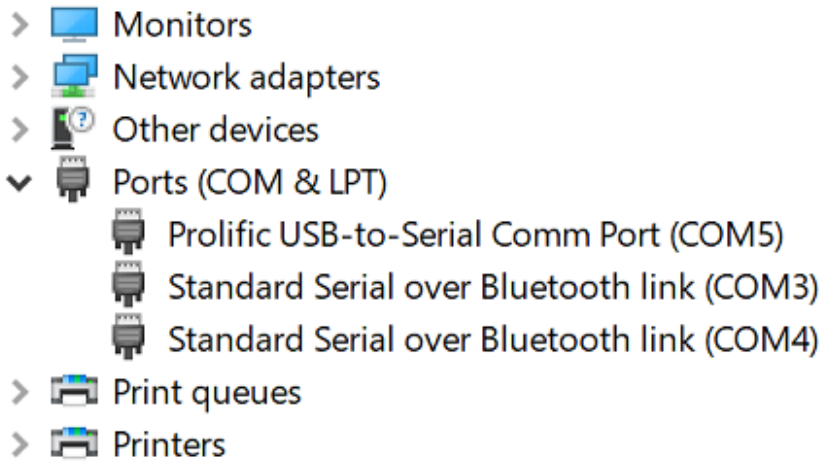
  - **USB TO COM PL2303 V2** (blue cable)

    If your computer does not recognize that device automatically, you may need to install its driver from the seller's page or from the official website: http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41

    For Windows 10, there could be an error with the driver. If so, you can reinstall it with this guide.

  - **USB SERIAL CONVERTER FT232** (black cable)

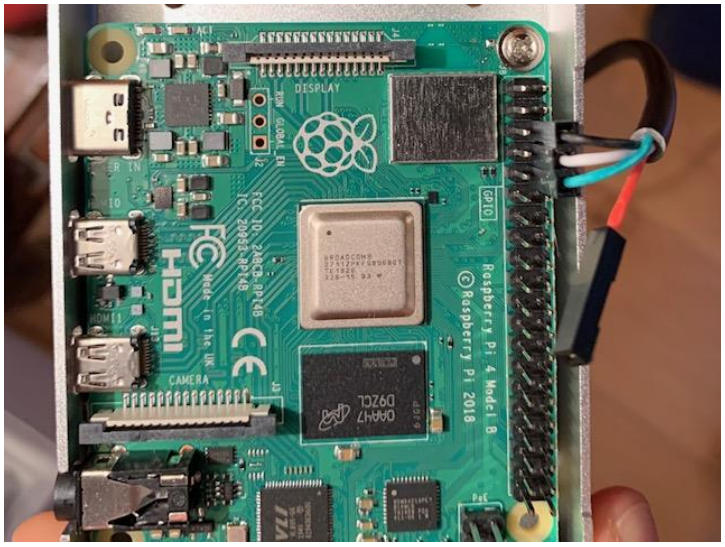
  After driver installation, you will see the port in Windows "**Device Manager**" (search and run from search bar). For example, it is currently running as **COM5** port in the figure below:



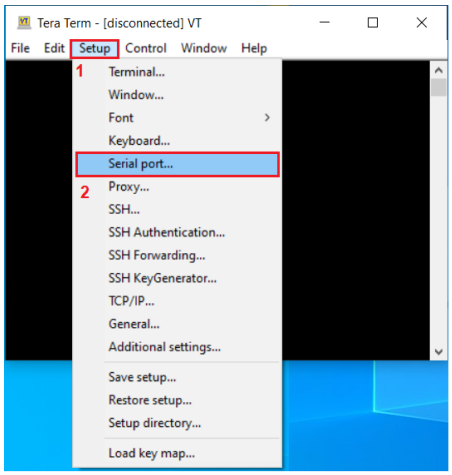
- **Connecting the cable**

  In the adapter cable, BLACK, WHITE and GREEN are respectively Ground, RXD and TXD pins of the cable. It is the same for PL2303 (blue) or FT232 (black) cables.

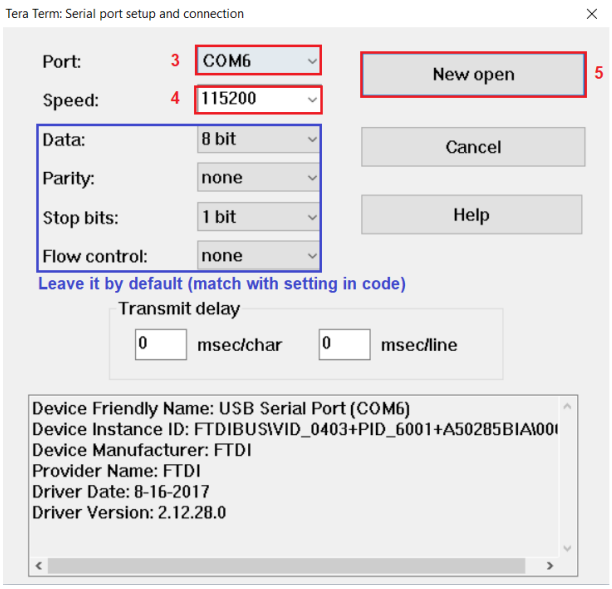
  To connect two UART devices, we need to connect RX pin of a device to TX pin of the other and vice versa. Therefore, you should connect the **BLACK wire to Ground** (Pin 6), the **WHITE wire to TXD** (GPIO 14) and the **GREEN wire to RXD** (GPIO 15) of the Raspberry as below. Refer to pinout diagram provided at the beginning of the lab guide for details.



- **Use a terminal tool to communicate with the serial port**
  - For Windows, install TeraTerm tool at this link: TeraTerm. Then use TeraTerm for serial connection as below:



Guide: follow step 1->5 listed above (*COM port at step 3 is usually the one with larger number if there are more than one*)



*Select **Setup** > **Serial port.***

*Select the COM port which is connected to the cable in **Port** option.*

Select the values for UART connection as configured in our code:
Speed: 115200 (baud rate).
Data: 8 bit
Parity: none
Stop bits: 1
Flow control: None

Finally, select **New Open** to start the connection.

- For MacOS, you can do it as follow:
  - Open an OS X terminal session
  - Find the right TTY device by typing: `ls /dev/cu.*`

    With the USB-Serial adapter plugged in, you'll get a list, including something like this:

    ```
    $ ls /dev/cu.*
    /dev/cu.Bluetooth-Modem         /dev/cu.iPhone-WirelessiAP
    /dev/cu.Bluetooth-PDA-Sync      /dev/cu.usbserial
    ```

  - Then type: `screen /dev/cu.usbserial 115200` to connect to the serial port
  - Reference for more details: https://pbxbook.com/other/mac-tty.html

- Now, power on your raspberry pi board. You should see the same "**Hello Word!**" message and it sends back all typed characters as we see on emulation.

5. Lab Exercises:

a. Exercise 1

Modify your main() function so that it will convert all received characters from UPPERCASE to lowercase and vice versa and then send back to the console.

b. Exercise 2

Create another project, and refer to the ARM Peripherals document to write the driver for UART0.

You can use the provided files here, however, refer to the ARM Peripherals guide to complete the code (filling /*YOUR CODE HERE*/ parts).

Note: To test uart0, we use the command below:

```
qemu-system-aarch64 -M raspi3 -kernel kernel8.img -serial stdio
```

(the first `-serial` argument is for uart0, and the second one if available is for uart1).

c. Exercise 3

Inside one project, put both uart.c, uart.h, uart0.h and uart0.c. Modify the makefile so that you can control to build uart1 or uart0 through a single identifier definition.

Hint: In Makefile, you can have conditional statement as in example below:

```
#RunEmulation = 1
ifdef RunEmulation
all: clean kernel8.img run
else
all: clean kernel8.img
endif
```

References

This lab guide is based on and adapted from the documents below:

Writing a "bare metal" operating system for Raspberry Pi 4, Adam Greenwood-Byrne, tech CEO @RealVNC, https://isometimes.github.io/rpi4-osdev/

Learning operating system development using Linux kernel and Raspberry Pi, Sergey Matyukevich, https://github.com/s-matyukevich/raspberry-pi-os

Bare Metal Programming on Raspberry Pi 3, Zoltan Baldaszti, https://github.com/bztsrc/raspi3-tutorial