

# **Introduction to Mechatronics (ME/AE 6705)**

## **Lab Assignment 4**

### **Serial Communication Using UART**

#### **Objective**

The main objective of this lab is to write a program that establishes 2-way serial communication between the MSPM0 and a PC/laptop via UART. The lab makes use of the Driverlib library, although you are welcome to use direct register access instead if desired.

#### **Deliverables and Grading**

To get credit for this lab assignment you must:

1. Submit a typed report as a PDF file to Canvas, answering the questions at the end of the lab (2 pages max, can be shorter). This is due at the beginning of class on the due date specified on Canvas. (20 points)
2. Demonstrate proper operation of your code to TAs or instructor during the office hours or demo hours. You must write your own code for this lab – no lab groups are allowed. (40 points)
3. Submit the commented final version of your code on Canvas. all your source files (.c) and header files (.h). This includes the source file with your main() function and any header files you added. Make sure to include the ti\_msp\_dl\_config.h and .c files. If you created the project using the method outlined in Lab 2, the ti\_msp\_dl\_config.h/c files should be located in the project folder under "Generated Source > SysConfig". Do not zip any of your files. This makes them impossible to view via the built-in Canvas grading system. Please submit the individual files. (Pass/Fail)

#### **Setup**

This lab requires Code Composer Studio, a SSH terminal (Putty for Windows and Linux, Terminal for MacOS) and the MSPM0 LaunchPad. The lab uses only the onboard electronics of the MCU and a USB cable. The MSPM0 LaunchPad contains 4 UART peripheral devices. We will use one of them to communicate with your laptop or desktop.

*Note: To create a new CCS project for the MSPM0 target device, follow the instructions in the document entitled "How to create a new project for the MSPM0.pdf" on Canvas under MSPM0 Documents.*

## Problem Statement

Write a program to transmit and receive data between the PC and MSPM0 over UART. The program should receive data from the SSH terminal console, and then echo it back. Specifically, the MSPM0 should receive data from the SSH terminal, storing all data in a character buffer. When a carriage return character is received (generated on the keyboard by the “enter” or return key), it should transmit the entire buffer back to the SSH terminal terminal (this is called “echoing”). It should then clear the buffer and wait for the next line of text to come in. This echoing process is repeated in an infinite loop.

You must use either a receive interrupt or transmit interrupt, or both. If you choose to use only one, you should use polling for the other functionality. Note – the program is easier to design if only a receive interrupt is used.

Your buffer should hold at least 200 characters. When data is echoed back to the terminal (upon hitting return or enter), your data should append a carriage return and newline character at the end to move to the next line in the SSH terminal.

## Background

### UART:

Communication between two devices can be serial or parallel. In serial communication, data is transferred one bit at a time, as opposed to multiple bits in parallel communication. Serial communication is typically used in applications where I/O pins are limited and clock synchronization requirements make parallel communication difficult.

Serial communication can be further subdivided into two categories: Synchronous communication and Asynchronous communication. Synchronous communication uses a data channel to transmit data and a clock channel to maintain synchronism. In asynchronous mode, only a data channel is used and data is transmitted at a certain baud rate. The clock channel is absent in asynchronous communication – rather, clock signals and baud rate are separately generated on each individual device. Synchronous communication is used in applications that require continuous communication between two devices whereas asynchronous communication is used when intermittent communication is acceptable or required.

UART is a hardware device that implements asynchronous communication. The communicating devices first agree on a baud rate. Data is transmitted in groups of 8 bits with parity (optional) and other optional bits as available on the communicating devices. The data packet also consists of one start bit and one or two stop bits. All of these settings are matched for both devices during their respective configurations.

Once configured, data can be transmitted by writing to the TXDATA register or by using the function `DL_UART_transmitData()` from the Driverlib library. Similarly, received data is accessed by reading RXDATA or using the function `DL_UART_receiveData()`.

Flags are raised when transmission or reception of data are complete. The transmission buffer empty flag TXFE is raised when all the data is transmitted and there is no more data in TXDATA. This flag is cleared by writing data to be transmitted to the TXDATA. Similarly, the receive complete flag RXINT is raised when a complete byte (8 bits) is received. This flag is cleared by reading the RXDATA register. When these

flags are raised, if enabled, corresponding interrupts are also triggered. The interrupt service routine can then be used to either transmit new data or to read the received data.

Note: All received data from the SSH terminal console will be encoded in ASCII format, since it will be typed at the keyboard. To determine the hex equivalent of an ASCII character, consult the ASCII encoding table found at: <http://www.asciitable.com/>

### Steps:

The program has to establish successful communication between microcontroller and computer. Initialize the UART module using the functions from the driverlib library as discussed in class. First select a baud rate to use (standard baud rates are 9600, 38400, or 57600). The configuration parameters (the integer and fractional divisors) can be computed from using the method shown in the slides from Lecture 11, assuming a certain oversampling rate. You can write the oversampling rate and the baud rate divisors to the appropriate registers using the Driverlib functions `DL_UART_setOversampling()` and `DL_UART_setBaudRateDivisor()`.

Use the default clock rate of 32 MHz, which is the default clock rate of the MSPM0. Your UART configuration should use normal UART mode, data word length of 8 bits, both transmit and receive directions, no parity bit, no hardware flow control, and one stop bit. These options should be used when setting the options in the UART configuration struct.

At the beginning of your main function you should configure the clock. Use the following commands to set up the clock:

As a global variable declare the following struct:

```
DL_UART_Main_ClockConfig gUART_0ClockConfig = {  
    .clockSel      = DL_UART_CLOCK_BUSCLK,  
    .divideRatio = DL_UART_CLOCK_DIVIDE_RATIO_1  
};
```

Then at the beginning of your main function place the following function call:

```
DL_UART_setClockConfig(UART0, &gUART_0ClockConfig);
```

This will configure your UART0 module to run at the same speed as the BUSCLK clock source, which is 32 MHz by default.

Follow the rest of the steps from the lecture to set up and initialize the UART. Also consult the Driverlib reference manual and MSPM0 technical reference manual for guidance as you set up your code.

## Recommended SSH terminals for different operating systems

### Putty Configuration (for Windows):

Download Putty (which is free). In the Connection Type field on the home screen, select "Serial". Place the baud rate in the Speed window. After plugging in the MSPM0, go to Device Manager and look for

“XDS110 Class Application/User UART”. The COM port listed (COM1, COM4, COM5, etc) is the COM port where the UART is found. Type this COM port in the Putty Serial line field.

Then go to the Serial window on the left. Match the data bit, stop bit, and parity configurations you used in your code. Turn Flow control to XON/XOFF. You can then save this configuration by going back to the Session window, typing a name in the Saved Sessions field, and clicking Save. To load the configuration, click on it and select Load.

To connect to the MSPM0, you must do the following:

The shorter approach:

- Start Putty and open your session as described above
- Start Code Composer Studio and program the MSPM0 using the debug button
- Once the code hits its initial breakpoint, terminate the Debug session in Code Composer Studio by hitting the red square “stop” button
- You should then be able to communicate with the MSP via the Putty console

If that doesn't work, follow the longer approach:

- After programming the MSPM0, close both Putty and Code Composer Studio
- Unplug the board
- Plug the MSPM0 back in
- Start Putty and Open your session without starting Code Composer Studio
- You should then be able to communicate with the MSP via the Putty console

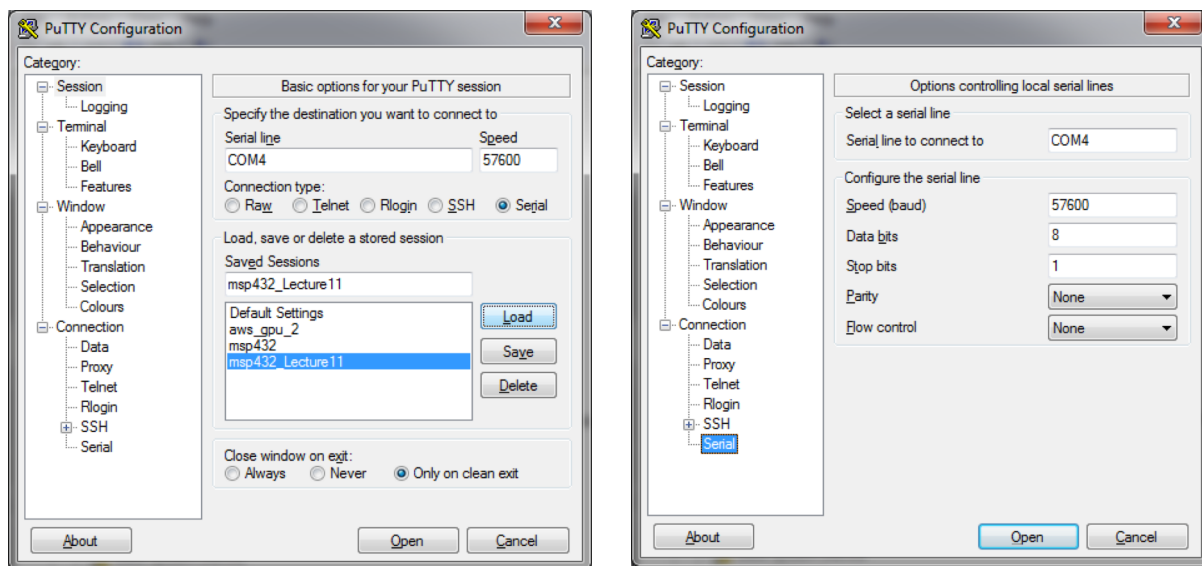


Figure 1: Putty Configuration in Windows

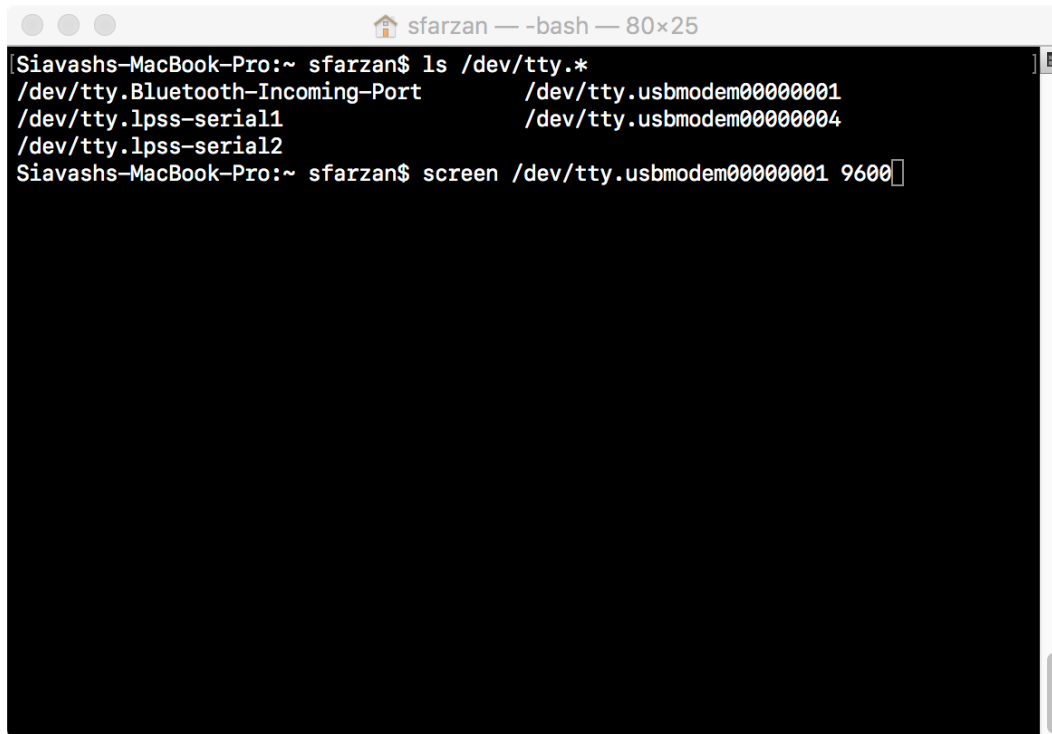
### Terminal Configuration (for MacOS):

Terminal app is a terminal emulator included in the macOS operating system by default.

To start Terminal, go to your Mac's Applications folder, click on the Utilities folder, and then click on Terminal.

- Connect the MSPM0 Dev board to a USB port of your Mac

- Type **ls /dev/tty.\*** to list the serial devices on your Mac and find the one you want to connect with (it starts with /dev/tty.usbmodem).
- Type **screen /dev/tty.usbmodem00000001 9600** (replace /dev/tty.usbmodem00000001 with the right one listed on your Mac and also replace **9600** with the baud rate you have selected) to start the connection to the serial port of the MSPM0 dev board.



```
Siavashs-MacBook-Pro:~ sfarzan$ ls /dev/tty.*
/dev/tty.Bluetooth-Incoming-Port    /dev/tty.usbmodem00000001
/dev/tty.lpss-serial1               /dev/tty.usbmodem00000004
/dev/tty.lpss-serial2
Siavashs-MacBook-Pro:~ sfarzan$ screen /dev/tty.usbmodem00000001 9600
```

Figure 2: Terminal Configuration

- You should then be able to communicate with the MSPM0 LaunchPad via the Terminal console.
- To stop the communication, close the session by: **control+a**, then **k**, then **y**.

## Requirements

1. Successfully demonstrate your program and all the required functionalities to TAs or instructor.
2. Submit the commented final version of the code on Canvas.
3. Answer the below questions and submit the typed report to Canvas.

## Questions

1. Suppose you are using a baud rate of 38400. How far off can the clock rate of your two devices be before you start reading data incorrectly? Do not use the 5% approximation, but rather compute the answer exactly. How does it compare to the 5% approximation?
2. Suppose your clock signals differ by more than the amount in Question 1. What error flag would you expect to see, in what register? Why would this error flag get triggered in this situation?

3. How could you solve the above problem (i.e., by changing the baud rate)? What is the inherent performance penalty associated with this solution?