Introduction

This week in lab you will become familiar with configuring and using UART for serial communication. UART stands for Universal Asynchronous Receiver/Transmitter and is a standard component of microcontrollers. UART provides asynchronous serial data communications for compatible devices.

What does it mean for data transfer to happen asynchronously? Asynchronous means that the transmitter and receiver do not share a common clock. In other words, the sender transmits data using its own timing source, rather than sharing a clock with the receiver. Conversely, for synchronous data transfer, both the transmitter and receiver access the data according to the same clock. Therefore, a special line for the clock signal is required. For asynchronous communication, the transmitter and the receiver instead need to agree on a data transfer speed. Both transmitter and receiver set up their own internal circuits to make sure that the data transfer follows that agreement. In addition, synchronization bits are added to the data by the transmitter and used by the receiver to correctly read the information.

Data communications are serial, which means that information is sent serially as a sequence of bits. The bits are grouped together to form serial frames. A serial frame can contain 5 – 8 bits of data, and there are synchronization bits that distinguish the start and end of a frame. There is also the option of a parity bit to provide simple error detection over a communication medium that may be noisy.

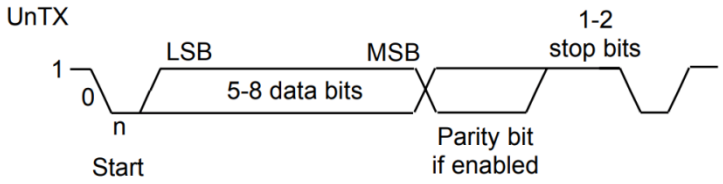
Serial communication sends one bit at a time on a single wire. The number of bits that can be transferred per second is indicated by baud rate. This tells us the speed of communication between the transmitter and receiver.

Below is a diagram that depicts how data are transferred. Information about baud rate generation can be found in section **14.3.2** of the **Tiva Datasheet**. Section **14.3.3** describes in more detail how data transmission works.

Communication requires at least one transmitter and receiver, or two endpoints (one transmitting and one receiving). The CyBot is one endpoint, and your lab computer will be the other endpoint. The lab PC uses a Putty terminal to facilitate communications from the PC end. The Putty software is able to talk directly to the serial port (e.g. COM1) of your PC. As you type characters in the Putty terminal, ASCII characters will be sent to the serial port and over the wire in the cable.

Both endpoints must use the same frame format and baud rate. Frame format indicates the number of bits in the frame and consists of a start bit, data bits, an optional parity bit, and 1 – 2 stop bits. The standard number of data bits in a frame can be 5, 6, 7, or 8. There is also a special 9-bit data mode that can be used. The parity bit can be odd, even, or none. Code examples for sending/receiving data are in **Figure 8.73** on **page 662** in the textbook (Note this code cannot be directly used, for reference only).

Figure 14-2. UART Character Frame



Reference files

The following reference files will be used in successful completion of this lab:

1. lcd.c, a program file containing various LCD functions
2. lcd.h, the header file for lcd.c
3. timer.c, program file containing various wait commands
4. timer.h, header file for timer.c
5. TM4C123GH6PM Datasheet

****Note**: In order to use interrupts, you must include driverlib/interrupt.h and stdbool.h. You may want to use the following header files.

```
#include <stdint.h>
#include <stdbool.h>
#include "timer.h"
#include "lcd.h"
#include <inc/tm4c123gh6pm.h>
#include "driverlib/interrupt.h"
```

In addition to the files that have already been included for you, you will need to write your own **uart.c** file and **uart.h** file and the associated functions for setting up and using UART. Separate functionalities should be in separate functions for good coding quality and reusability purposes. This means that in your uart.c file you should write separate functions for initializing/configuring UART and setting up interrupts, sending characters, and receiving characters. Remember to use good naming conventions for function names and variables. For example, you may want to name your UART initialization function **uart_init** as there may be other initialization functions you will write in later labs that will eventually have to be used together. Minimally, we recommend defining the following functions:

```
void uart_init(void);
void uart_sendChar(char data);
char uart_receive(void);
```

In addition, in Part 3, you will re-write your code to use interrupts, and you will then need to define an interrupt handler function (interrupt service routine):

```
void UART1_Handler(void);
```

Tip: In order to view functions that may be useful for configuring interrupts, once the interrupt library is included as a header file, you can use Ctrl + Click on the name of the include file to open it. This is also useful for viewing a file containing the register macros or jumping the code for a function you are calling.

Part 1: Receive and display text

Using **UART1 via PORTB**, write a function that will receive characters from your computer's putty terminal window. The program should display the entire series of characters on the LCD when 20 characters have been received or when you press ENTER ('/r'). When you press enter, you should not print out the character ('\r') on the LCD.

For debugging purposes, you should display each character on LCD line 1 as you receive it. You will also need to buffer these characters prior to displaying them all together after 20 characters have been received. As each character is received and placed into the buffer, a number should appear on the LCD that indicates how many characters are currently stored in the buffer, and you should print the character received. Once 20 characters have been received, clear the display prior to printing the entire series of characters on LCD line 1.

Don't worry about editing your series of characters. You don't have to manage your buffer to accommodate backspace or delete. However, if you are looking for an additional challenge, you may want to explore this feature.

To set up UART communication, you will need to visit **14.4 Initialization and Configuration for UART** in the **Tiva Datasheet**. This will walk you through step by step setting up the necessary registers. Additionally, **Figure 8.73** on **page 662** of the textbook provides a good example for initializing, transmitting, and receiving using the UART (**Note: code is for reference only, cannot be directly used).**

Putty Setup: See Lab 3 for a reminder for how to configure PuTTy

For the C code running on the CyBot:

- **Virtual Cybot:** use a baud rate = 115200, #Date bits = 8, #Stop bits =1, Parity = None
- **CyBot with RS232 cable:** use a baud rate = 9600, #Date bits = 8, #Stop bits =1, Parity = None
- **CyBot with WiFi:** use a baud rate = 115200, #Date bits = 8, #Stop bits =1, Parity = None

If using PuTTy with the Physical CyBot and a RS232 cable: Plug in the white serial cable to the CyBot board and open Putty. To communicate between the CyBot and your lab computer, a number of standards need to be used. The cable you attach to the CyBot's baseboard is a male DB-9 connector, and the signaling used over the cable complies with the RS-232 standard. There is an integrated circuit on the CyBot baseboard that provides RS-232 compliant signaling. Ensure the CyBot base board communication switch set to **RS232 (**not **WiFi)**. Putty configurations are shown below.

****Tip:** To prevent having to set Putty up again when opening a new session, in Session → Saved sessions, type a name for this session and select save. This will now appear in the box below. When you are on the same computer, your Putty sessions that you created will be saved. To load one, simply click on it and select load.

Checkpoint: Demonstrate UART communication to your mentor. Characters should be displayed on the LCD when 20 characters have been sent or 'enter' is pressed.

Part 2: Echo Received Character

In part 1, your focus was on receiving characters on the microcontroller. This part will focus on transmission of characters back to Putty. You will write a program to transmit each received character back to Putty.

Before beginning, turn off local echo on Putty by selecting **Terminal → Line Discipline Options → Force off local echo and local line editing**. When Enter is pressed only a **Carriage Return ('\r')** is sent by Putty. When transmitting back to Putty, you should send a **Line Feed ('\n')** following the Carriage Return. This will tell Putty to advance to the next line of the interface.

Checkpoint: Demonstrate character echo to your mentor. The bot should be transmitting each character it receives back to Putty to be displayed.

Part 3: UART Using Interrupts

Once you have demonstrated round trip communication (sending and receiving), you will implement this round-trip communication using interrupts for the CyBot receive side. You will need to set the interrupt mask register UART1_IM_R and the NVIC to enable interrupts. Additionally, you will need to create an interrupt handler for the UART to receive characters. There are several useful interrupts based on triggers that might be used, such as receive is complete, and FIFO is not empty. Make sure you enable and process the appropriate interrupt(s) by using the correct bits in the UART and NVIC configuration registers.

Checkpoint: Demonstrate round trip communication between the CyBot and Putty using interrupts. Your CyBot should transmit each character it receives back to Putty to be displayed. Additionally, communication should be done using a baud rate of 115,200.

Part 4: Update the 180 degree scan part of your Simple Mission code to integrate calibrated IR data, and manual Cybot driving

1) **Calibrated IR sensor data**:
   Update the scan portion of your "Simple Mission" lab to return both the PING distance, and a new column that gives the calibrated IR distance for a 180 degree scan whenever an 'm' is sent to the CyBot from PuTTy.

2) **Manually navigate to skinniest object using sensor scan data only:**
   Manually drive the CyBOT using the 'w','a','s','d' keys of the keyboard, and **as needed** use the 'm' key to have the CyBOT make an 180 degree scan as you drive to within 5 cm of the skinniest object (**without hitting it**). The user needs to be able to manually interpret the sensor data to manually drive to within 5 cm of the smallest object. In other words, the user is not allowed to look at the CyBOT. Only sensor data can be used for navigating the CyBOT. Use both a text log file of the collected data, and a graphical view of the sensor data.

Checkpoint: Demonstrate that you can cleanly display scan data to PuTTy, graphically display, and that you can manually drive to within 5 cm of the skinniest object based on the senor data that you display.