Seth Cram

ECE341

Lab7 Prelab

10/19/2021

**Prelab 7: Asynchronous Serial Communication**

Goal:

To use Universal Asynchronous Receiver-Transmitters (UARTs) to communicate with the Cerebot board through a secure shell terminal emulator, while on a computer. Also, to learn how to implement a serial, asynchronous communication protocol in conjunction with aspects from previous labs.

Background Information:

We’ll use a serial, asynchronous communication protocol for this lab, referred to as RS-232. RS-232 is robust and suitable for long distances, which will allow us to use remote control. The serial nature of our communication comes with some advantages and disadvantages. It’s normally slower than synchronous communication because the lack of a clock means the receiver has to estimate when to sample the receive line. Which results in the receiver oversampling the receive line to check for data. Asynchronous communication also requires overhead for message synchronization, so it operates at a slower data rate. The data bits have to be uniformly spaced. For our case of using the RS-232 protocol, it's only one device to another, which is referred to as point-to-point communication.

The BAUD rate is the inverse of the period, and referred to as bits / second. Meanwhile, the bit rate is the number of “data” bits / second. On a single data line, we’ll have 8 data bits, a start, parity, and stop bit, resulting in 11 total bits. This corresponds to a channel efficiency of around 73%. The start and stop bit signal the width of a transfer, and the parity bit checks the transmission’s accuracy. We’ll configure the parity bit for odd parity to ensure that an odd number of 1’s, excluding the parity bit, are transferred each time. This makes sense when considering that the start bit is always a zero, so we expect 9 bits that are 1’s. We have the option of using even, odd, or no parity bit. If left unconfigured, the parity bit will always be high. The stop bit is specifically to allow the receiver time to re-initialize before its next transfer. The sender and receiver have to be set to the same BAUD rate and parity bit for correct data transfers. When sending data, there is no acknowledgement back to the sender from the receiver.

We will be communicating our characters in ASCII (American Standard Code for Information Exchange), similar to how we did during the previous lab for the LCD. We’ll be using the Cerebot board with both local (buttons) and remote (computer) control. The computer and Cerebot board will interact in a peer-to-peer fashion. Which means that either one can initiate a data transfer. They are also connected with two lines, which are the sender to the receiver line, one in each direction. So, the communication is bi-directional.

Putty is a terminal emulator, and possible secure shell, which we’ll use to connect to a serial port. The order of operations of a write: First, we will enter a stepper motor string into the putty software, which will then display it to the computer screen, as well as pass it character-by-character to a UART. The RS-232 protocol will then transfer these characters from the computer’s UART along a USB cable to the Cerebot’s UART. The transferred data will then be sent to the Cerebot’s processor to be decoded into stepper motor input with the software we craft during this lab. Said software should also allow local communication using the 4 button combinations for the stepper motor, as well as display the currently output stepper motor code on the LCD.

We should be able to specify any speed, direction, and step mode for the stepper motor by passing a stepper motor string to the RS-232 protocol. The format of this string is very specific. Being able to pass any combination of these greatly increases our controllability from just the 4 button combinations used in previous labs. We will assemble and decode these strings using the standard I/O library’s “sprintf()” function to format our I/O. We will also use other external library functions, such as “sscanf()” and “strcmp()”, to read and parse a string by its spaces.

During this lab, we’re essentially using the ‘LCDlib.c’ and ‘LCDlib.h’ as low level code to provide basic functionality to access a hardware device (LCD), otherwise known as a driver specifically for the PIC32. Both Operating Systems and users utilize drivers to interact with hardware in most computing environments.

Verification is the combination of both controllability and observability. During our previous lab, we increased observability by moving from LEDs to the LCD Screen. The UARTs used during this lab will provide both controllability and observability. We could use a scripting language to communicate information with the UART, but for this lab we’ll manually type it into our putty. The UART requires three pins total, one for transmit, one for receive, and the third for ground. Compared to the 11 pins required to talk to the LCD, which was around 10% of our total pins, these three pins are trivial. In total, we have 6 UARTs in the PIC32, but we’ll only use one for this lab. The Cerebot board only supports hosting 2 UARTS.

This lab will be a combination of interrupts, ISRs, the LCD library, and UART communications. During this lab, we’ll also be introduced to the idea of managing a shared resource by mutually excluding access time between assessors. Three error flags are generateable by the processor on receiving serial communication: A parity error, framing error if the stop bit is low, and overrun error if data transfer isn’t complete when a new one comes in.

Plan:

First, I’ll configure Putty for 19,200 bit rate, whatever COM input the USB is connected to, turn off the echoing of entered lines, set parity to odd, set a single stop bit, and 8 data bits. I’ll be sure to save this session as “Cerebot” for future usage.

Then, I’ll take my lab 5 source and header file and separate the change notice and timer 1 ISRs into their own source files with their own header files. Then, I’ll create a “test.c” file that has the only main() function out of all my four other source files. I’ll include the four header files, the string and standard I/O libraries, and device specific files at the top of the test file. With an empty main() function, I’ll try to build the project to verify I linked and included the files correctly. I’ll be sure to define the global variables for each file they’re needed in. Next, I’ll use listings 1-3 from the lab 7 handout to test and understand precisely how the UART functions operate. But, to get these to work I’ll need to call UART1’s initialization function first. I’ll use odd parity and a bit rate of 19,200 since that’s the bit rate we told Putty to use.

Next, in our change notice .c file, we’ll need to add a function to calculate step delay times during run time based on the desired RPMs. Then, in our change notice decode\_buttons(), we’ll have to output whatever string we changed the stepper motor settings to. This’ll require the inclusion of both ‘comm.h’ and ‘LCDlib.h’ since we use their functions to output to the terminal and the LCD.

Back to our test file, we’ll need to use the same system\_init() utilized in setting up the ISRs. We’ll then set up the LCD, and finally initialize both interrupts. Within our while(1), we’ll wait for the line of text from the UART, then disable the change notice interrupt, clear the LCD, output our read-in string to the LCD, parse the string into three variables, set the stepper motor global variables based on the three variables, and finally re-enable the change notice interrupt. The change notice interrupt must be disabled while we’re outputting to the LCD and changing the global variables to mutually exclude these background operations from the change notice event accessing the shared resource of the LCD, UART, and global variables.



