ECE341

**Lab7 - Asynchronous Serial Communications**

Report

**Seth Cram**

**10/26/2021**

**Introduction:**

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.

*Background information was sourced from lecture notes, Dr. J’s “RS232 Intro” slides, and the lab 7 handout.*

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.

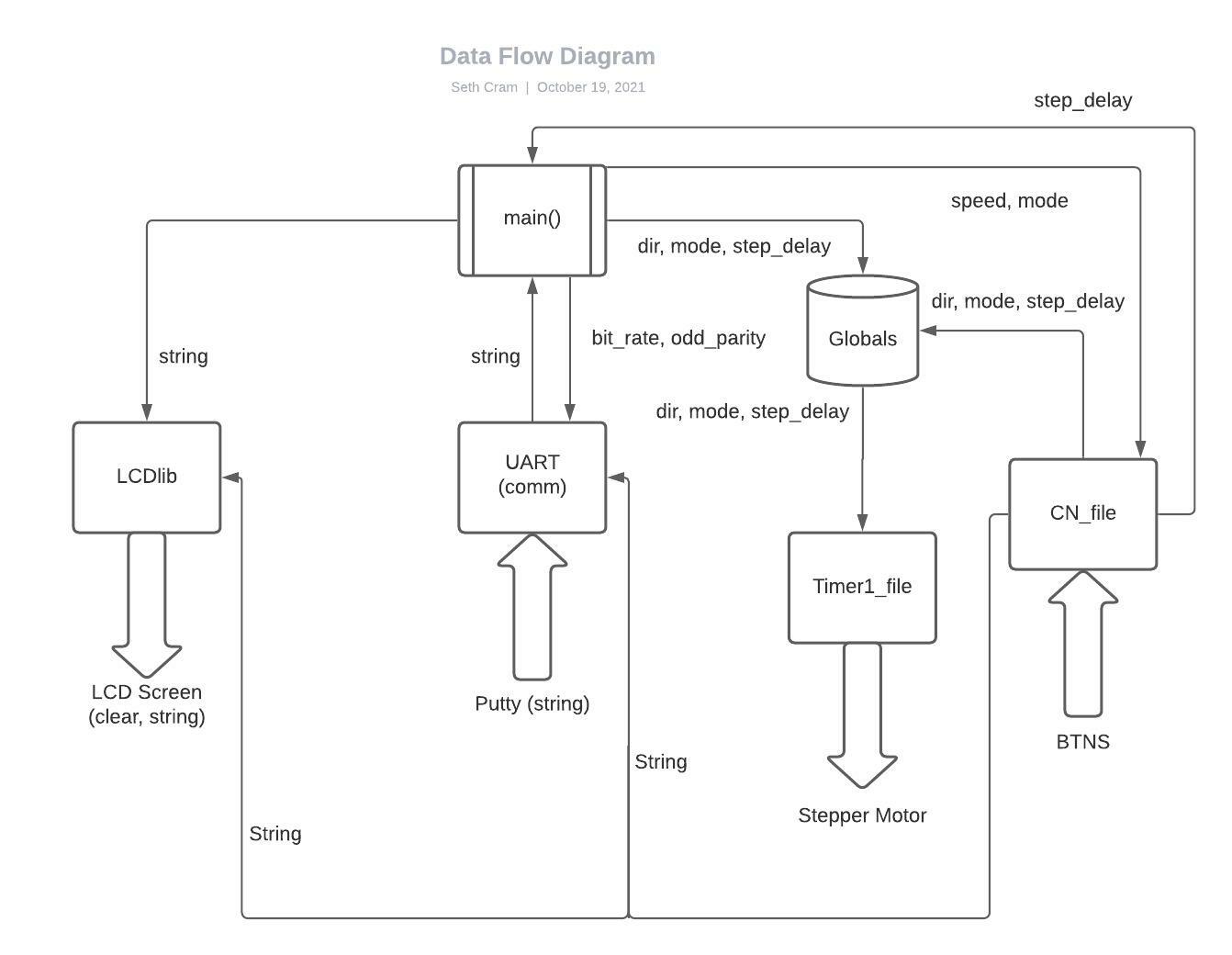
Then, I’ll take my lab 5 source and header file and separate the change notice and timer 1 ISRs into their own source files with their own header files. Then, I’ll create a “test.c” file that has the only main() function out of all my four other source files. I’ll include the four header files, the string 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.

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what functionality I’d need to incorporate and their overarching purpose in the grand scheme.

**

As seen above, we’ll need to add additional data passing capabilities to our Change Notice file. It now passes data to both LCDlib and the UART. Every other file communicates primarily with our file containing main().

In the coming listings, I tried to keep them short and to the point. I could’ve added a listing for every function in the communications library, but since we didn’t write them and instead utilized only two functions from it, I decided to specify the important pieces of it and its functioning. I also cut some code out of the test file to further emphasize new aspects of the lab, rather than include the entire file’s code.

**Listing 1. File Linking:**

An important aspect of this lab was getting multiple source code files to work together smoothly. This was accomplished through including another file’s header file if we wanted to use one of its functions. Another important addition was the use of the keyword “extern” to share common global variables across our source files. Using so many c files, the debugger wouldn’t know where to start if we had several, so we were required to only have a single main(). This restriction almost begged for a “test” file or a file of its own.

**Listing 2. Addition to decode\_buttons():**

Within my switch statement, I hardcoded stepper motor string based on the button combination to later output it to the UART and the LCD. The added string “outputStr” dynamically allocates the data needed to hold the string.

*char \*outputStr;*

*outputStr = "CW FULL 15";*

*outputStr = "CWW HALF 10";*

*outputStr = "CWW FULL 25";*

*outputStr = "CW HALF 15";*

*putsU1("\n"); //jump to next line*

*putsU1(outputStr); //output to UART*

*LCD\_puts(outputStr); //output to LCD on cleared screen*

I output a new line to the UART before displaying to avoid overwriting the contents of the current line in Putty. I could have concatenated this to the output string, but just did two operations for simplicity's sake. My LCD\_puts() function clears the screen before outputting a new string to the LCD screen. Local control using the buttons is now properly set up.

**Listing 3. ComputeStepDelay():**

Since the Putty terminal could pass speeds 1 through 30 in RPMs to the stepper motor, I needed a function to compute and return the necessary step delay based on the current mode and speed of the stepper motor.

*//separate function to compute step delay based on mode and speed:*

*int ComputeStepDelay( int mode, int speed ) //speed should be in RPMs*

*//FS = 1, HS = 2*

*{*

*int T\_delay;*

*T\_delay = 60000 / (speed \* 100 \* mode);*

*return T\_delay;*

*}*

The formula to compute step delay was found in the stepper motor lab handout. It assumes full step is 1, half step is 2, and speed is in RPMs. The three lines of code could be combined into one, but I wanted to keep them separate for readability. This function will be useful later on in our testing file that contains main().

**Listing 4. Test\_proj7.c:**

First, I included all header files with functions I planned on using. Next, I determined the maximum size possible for the overall stepper motor string, its direction, and step mode. I then added one for the null termination character, and set them to constant variables. I also added a couple extra characters just to be sure.

*const int LINE\_SIZE = 15; //should be 12*

*const int DIR\_SIZE = 5; //should be 4*

*const int MODE\_SIZE = 6; //should be 5*

*char line[LINE\_SIZE], currDir[DIR\_SIZE], currMode[MODE\_SIZE];*

I used these constants to initialize my string, direction, and mode character arrays.

Then, I called system\_init() which is the exact same system\_init() used to set up the Timer1 and Change Notice ISRs. Then, I opened the UART with odd parity and 19,200 bit rate. Next, I initialized the LCD then called the timer1 and change notice initialization functions. Both the LCD and UART initializations had to be above the change notice ISR because it used functions from both of them.

*system\_init(); //init syst from ISR lab*

*initialize\_uart1(baud, ODD\_PARITY);*

*initLCD();*

*t1\_intr\_init();*

*//setup change notice intrs for btn1 and btn2: (step\_delay init'd here)*

*cn\_intr\_init();*

In while(1), I then used getstrU1() from the communications library to read from the UART until a string was read in, using a while loop. I then echoed the string back to Putty using putsU1() with a new line character preceding it. The new line character was added for readability. I then disabled the change notice interrupt before outputting the string to the LCD. I then parsed the string into three separate variables using sscanf() from the standard I/O library.

*while(!getstrU1(line, sizeof(line))); //read UART string*

*putsU1("\n"); //jump next line*

*putsU1(line); //output to UART*

*mCNIntEnable(0); //when invoke LCD*

*LCD\_puts(line); //put read-in line on LCD: (clears before outputting)*

*sscanf(line, "%s %s %d", currDir, currMode, &currSpeed); //parse*

Finally, I used a combination of if, else-if, and strcasecmp() to set my global variables based on the three variables parsed from the stepper motor string. I used strcasecmp() from the strings library, instead of strcmp(), because strcasecmp() is case insensitive. This allowed for more error on the user’s part from the Putty terminal. I then re-enabled the change notice interrupt.

//if currDir is CW:

*if( strcasecmp( currDir, "CW" ) == 0 ) //IGNORE THESE ERRORS*

*{*

*dir = CW; //set global to it*

*}*

*//if its CCW*

*else if( strcasecmp( currDir, "CCW" ) == 0 )*

*.*

*.*

*.*

*//if currSpeed in the valid range:*

*if ( 1 <= currSpeed && currSpeed <= 30 )*

*{*

*//set global to calculated step\_delay:*

*step\_delay = ComputeStepDelay( mode, currSpeed );*

*}*

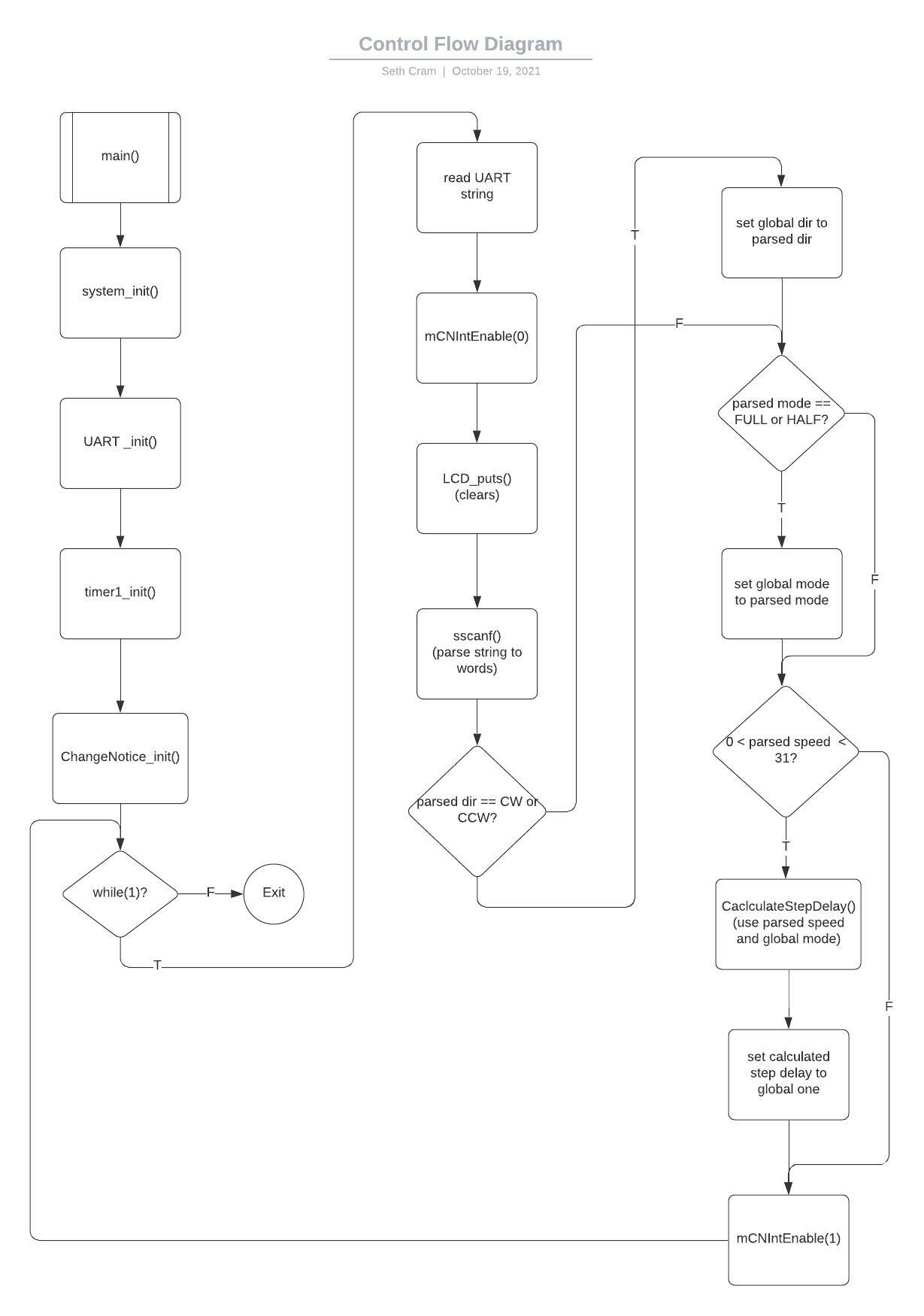
*mCNIntEnable(1); //when finish invoking LCD*

Setting the passed in UART string’s desired stepper motor speed to a corresponding value for the step\_delay global variable is where ComputeStepDelay() saw some use. Finally, the while(1) loop ended. I probably could’ve moved much of the code left in main() to functions. This would’ve increased readability and reusability, but for the sake of time I kept it simple.

**Listing 5. Communications Library:**

This source file and its corresponding header file were provided to us. The communications library was used in a similar fashion to the LCDlib. It had a large number of “helper” functions, with two majors ones for us to take advantage of. These two are seen above in our test function: putsU1() and getstrU1(). Also like the LCDlib, a putc and getc function assisted in sending/receiving new strings. These two string functions were so important because they allowed us to take advantage of the UART for serial communication to/from our Putty terminal.

Finally, my control flow diagram models the behavior of the above specified listings:



Not included in the data flow diagram are the change notice and timer1 ISRs that interrupt the normal sequencing shown above. Since those were displayed in lab 5, I saw no need to reiterate them.

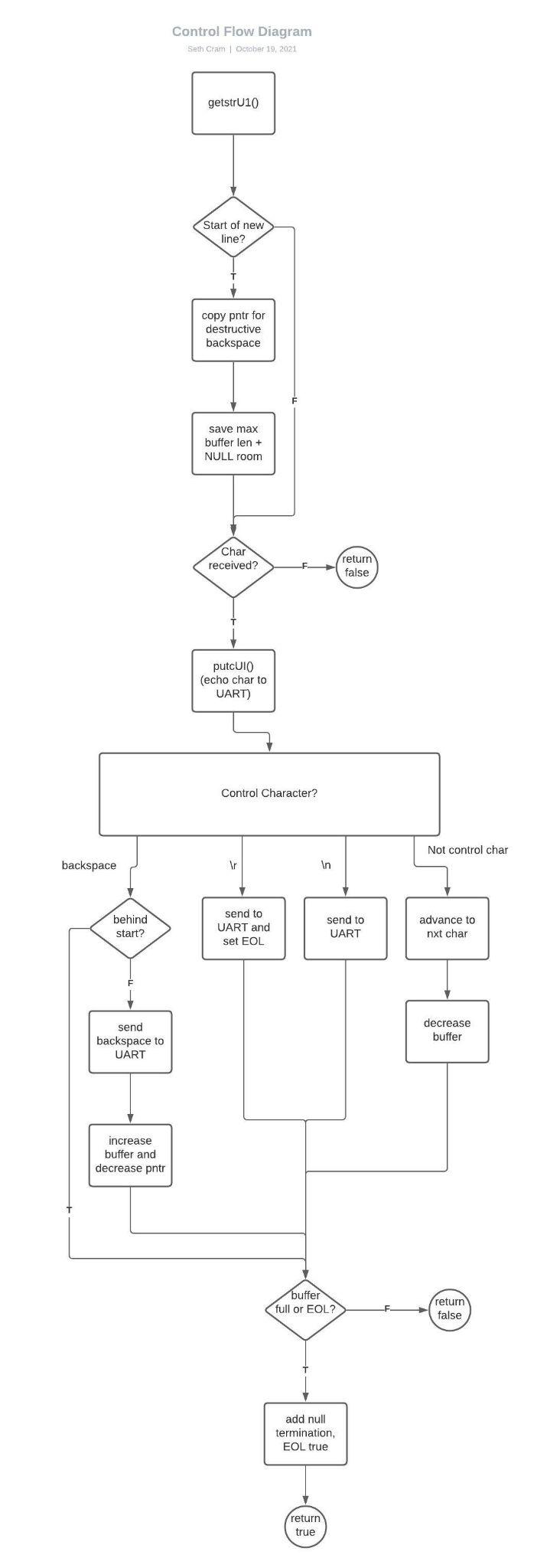
*– Why are the UART receive functions (getstrU1, getcU1) in the background? What would happen if they were in the foreground?*

Because they were developed specifically to be non-blocking. This is for the sake of the Timer1 and Change Notice interrupts. Since we don’t know how long it’ll take to get a string from the UART/Putty, we poll it. If this polling blocked the Timer1 and Change Notice interrupts, the buttons wouldn’t work and the stepper motor would stop turning. If these functions were in the foreground, then they would need to be a lower priority than both the Timer1 and Change Notice interrupt, which would require us to increase both their priorities. Otherwise, the buttons wouldn’t work and the stepper motor would stop turning.

*– What is the purpose of the \_mon\_putc function?*

It redirects printf() from outputting to standard output, to instead output to the UART serial port.

*– The CFD for the project must include the CFD sub-diagram for the function getstrU1. Essentially, read and understand the code that was supplied to you and use it to derive a CFD.*



**Testing and Validation:**

To showcase my work to the TA, I demonstrated local control from the buttons, and how that displayed on both the LCD screen and the putty terminal, as well as affected the stepper motor. Then, I showed him remote control capabilities by typing a stepper motor string into the putty terminal, having it echoed back, seeing it on the LCD screen, and once again watching as the stepper motor changed accordingly.

Interestingly enough, I ran into an error that I wasn’t able to recreate. I ran my program, operated the buttons correctly, then entered my first putty string this session. That stepper motor string showed up on the LCD with one of its characters replaced by an error. I talked to the TA about it and he was just as baffled as I was. We tried a variety of methods to try and recreate the error but none of them worked.

Function Checklist:

| **Test:** | **Results:** |
| --- | --- |
| Strings entered into Putty aren’t echoed back when our program isn’t running |  |
| Our use of getStrU1() returns the string entered into Putty |  |
| The LCD clears and prints the string read in from Putty correctly (remote control) |  |
| Strings entered into Putty are echoed back when our program is running |  |
| The LCD clears and prints the correct string when a button/button combination is entered (local control) |  |
| Putty receives the button combination generated motor string |  |
| Entering every motor string into Putty that’s able to be generated through button combinations results in the same step delay  (we calculated the step delay correctly) |  |

*– Describe how each subsystem was incorporated into the project.*

Change Notice and Timer 1 were incorporated into the project with how we used their system\_init(), and separate initialization functions to set up local button control. Separately, the Buttons subsystem was decoded and used through its inclusion in the change notice ISR to support local control. The Stepper Motor subsystem saw use in the Timer1 ISR through implementing correct stepping, direction, and speed as a by-product of a software counter.

The LCD subsystem was incorporated through displaying stepper motor strings on it. This occurred for every change in the stepper motor. So, we output to the LCD in the change notice ISR when a new button combination is received, and within our while(1) loop when a string is received from Putty. We also had to initialize it before the Change Notice ISR initialization, since we used a function from the LCD library there.

The UART was used with the help of the communications library to send/receive stepper motor strings to/from Putty. Within the background, we polled for remote user input using it, and when received, sent it back out to the UART for confirmation. We also output to the UART within the change notice ISR when a new button combination occurs. Therefore, just like the LCD subsystem, we had to initialize it before the Change Notice ISR initialization.

*– When a subsystem was added, how did you test it? Describe the testing process for each subsystem.*

First, I tested the UART subsystem by outputting back to the UART as soon as I received the motor string. I also verified that output wasn’t displayed to the Putty terminal when my program wasn’t running.

Next, I tested the Buttons and Stepper Motor subsystems by commenting out the UART-related code, and verifying that they worked correctly in conjunction with one another in this new lab. I did so by testing the buttons resulting in correct stepper motor output. I then tested the code to output to the UART in decode\_buttons() within the Change Notice ISR through looking at the Putty terminal.

Then, I added the LCD subsystem. My first test was outputting to it in decode\_buttons() within the Change Notice ISR through looking at the Putty terminal. Next, I uncommented my background UART code and tested that the LCD would correctly receive the same string sent to the UART. Before this, I made sure to disable the change notice interrupt.

**Conclusion:**

In conclusion, we learned how to implement our second communications protocol, which is serial and asynchronous. Through the use of UARTs and Putty, we were able to remotely control our stepper motor. We utilized the LCD for observability, as well as the Putty terminal. This new method of communicating to the stepper motor remotely greatly increases the degree of control and observability we have over the stepper motor.

Some limitations to our design include: The syntax of the stepper motor strings input from putty has to be incredibly specific. As it is now, we would have to change decode\_buttons() back to how it was before this lab to use the Buttons subsystem independent of the other files. So, the subsystems aren’t completely independent and that limits their ease of reuse.

*– Address the advantages and disadvantages of serial and parallel communications. When might it be better to use one over the other? Support your answer with examples.*

Serial communication requires far less pins. So, an advantage for serial communication would be the number of pins it uses, which makes it simpler, but a disadvantage would be its data transfer rate. This makes it better for longer distance communications. So, using communication from building to building, you’d most likely want to use serial, but from board to board would have a better time using parallel for such a short distance. An advantage of parallel would be its speed of transferring, but a disadvantage is its complexity and the number of pins required. For example, a UART only uses 3 pins, while the PMP to LCD communication requires 11. The UART operates at 8 data bits per clock cycle, while the PMP to LCD is 8 data bits per clock cycle but they all arrive at once rather than sequentially.

*– Why is it important for a design to be modular?*

It increases readability and reusability. Readability is increased because we can look at descriptions of a file or function to determine its purpose. Reusability increases because now we can easily reuse a function or file in another project without any extra hassle of copy and pasting code segments and hoping they still work properly.