Lab #1: Protocol Communication

ECE-121 Introduction to Microcontrollers

University of California Santa Cruz

Additional Required Parts

None—you will be using the USB connection between your Uno32 kit and the *ECE121 Console* application on the lab PCs.

Introduction

In the first lab you checked the toolchain from end-to-end, and performed an embedded version of *Hello World!*. Note that you did not do the traditional hello world, due to the fact that you did not have `printf` available to you. There is a subtle point here, in an embedded system, printf sends a data stream through the serial port, and a terminal program on the PC side receives and displays the data.

In this class, however, you are not handed the serial port library already working (as is the case in your other classes that used the Uno32). Since we use no training wheels, the serial port effectively does not exist until you write the code that makes it operational. That will be part of your task in this lab. You will develop a communication protocol that will talk to the *ECE121 console* application on the PC. This protocol will be used in *all* subsequent labs, and will form the basis of most of your debugging capability in (this lab and) future labs.

Communications protocols are ubiquitous throughout the embedded (and non-embedded) world; they allow increased data integrity and robustness at the cost of consuming some additional bandwidth. As in all engineering, the tradeoff can be optimized for your specific application. The console application you will be using (written in Python) essentially gives a graphical interface to the protocol messages.[1] That is, a button on the console will send a specific message to your microcontroller (which will then respond accordingly). Likewise, messages sent from your micro will cause the console to display to update and change part of the GUI in response.

This lab will be the first real lab where you are expected to apply something you have learned in class to a real microcontroller; this will take effort and quite a bit of trial and error. This will be much more work than the previous lab. Be sure to start (very) early, and give yourself plenty of time to get things done. This lab manual and the teaching staff will guide you, but you will have to put the work in on your own.

[1]This is extremely common, and in fact many specialized programs are nothing other than a GUI wrapped around a specific protocol that changes in response to received messages and generates outbound messages in response to user inputs. For example, the QGroundControl application for managing UAVs in flight is really just a GUI wrapper on top of the MavLink protocol.

Prelab Part 1 (About Prelabs)

This lab also includes several *prelab* sections, which are called out. These are expected to be written up and turned in **before** you attempt the lab. They will be graded, and are an integral part of the lab. We strongly believe that some dedicated preparation (hence prelabs) will save you countless hours in the lab.

As with the previous lab, this lab will divide itself into several parts that will be solved sequentially. To borrow Blue Origin's motto: *Gradatim Ferociter*,[2] you will be developing and testing each required subpart before integrating them into a coherent whole.

In this lab, you will be undertaking the following big blocks of tasks: (i) Bring up the serial port, (ii) Write (and test) the protocol library, and (iii) Write an application using the protocol library over serial to demonstrate functionality.

Note that each of these tasks breaks down into many more smaller tasks (which should seem familiar to how you approach a coding problem in the first place). We will detail what you need to do in the appropriate sections of this manual.

Info: The strategy of incremental development is one which we will continually emphasize and reinforce through this and many other classes in your engineering education. Much work has been done analyzing project failures, and the most common problem is that systems integration took too long or did not work because the component parts had not been tested together. While at first glance, incremental development appears to be slow, it is–in fact–the fastest way to accomplish the project. This is especially true as complexity grows and more people are involved. Build (code) a little, test a little, repeat. Break things up into smaller, manageable pieces, and get those working and tested. We will be repeating this *ad nauseum*.

1 Serial and Parallel Communications

If you need to get information from one place to another, you need communication. The origins of the word communication comes from latin and means "to share." All communication methods (even ones you use with other humans) require the same basic (abstract) steps:

1. Decide what information needs to be shared
2. Message composition
3. Message encoding
4. Transmission on specified media
5. Reception (message may be corrupted by noise)
6. Message decoding
7. Interpretation of decoded message

This is a bit pedantic, and can generalize to almost every form of communication invented by humankind. For those interested in the topic, take a communications class or read the seminal paper by Claude Shannon.[3] For our purposes, we are concerned with digital communications between a microcontroller and something

[2]Meaning "ferociously incremental" in Latin

[3]See: http://math.harvard.edu/ ctm/home/text/others/shannon/entropy/entropy.pdf

else (peripheral, computer, etc.) that will ultimately rely on the digital I/O of specific pins on the micro for signaling purposes.

Digital communication has a rich history, encompassing early telegraphy using Morse code in 1809 and teletypes starting in 1906 all the way to modern standards such as USB in 1996. In fact, digital communications are very prevalent in our modern lives; all of the data in and out of your smart phone (or frankly any other device you use) is encoded in a digital stream.[4]

The first fundamental choice in communication is for both parties involved to agree on the medium. Without a medium, there is no message. In our case with microcontrollers, the medium will be wires and electrical signals on those wires. Obviously there are other communications systems that use RF or optical media, but they are beyond the scope of this class. Along with the choice of medium, both parties must agree what symbols mean, and what rules control access to the medium.

There are two fundamental approaches to digital communications: parallel and serial. At the simplest level, digital communications can be connecting the output of one digital I/O pin to the input of another (establishing common levels, timing, interpretation, etc). Recalling that all data within a microcontroller is represented using *bits*, the question becomes how to transmit the bits from one side to the other.

Parallel: Each bit gets its own digital I/O pin, and all bits are transmitted at once along with a synchronization signal. The advantage of this is that many bits get transferred at each time slot, and very high throughput can be achieved. One of the disadvantages of parallel communication is that it uses up quite a few pins on the channel. Parallel communications also have issues at high speeds and longer distances due to clock skew. See https://en.wikipedia.org/wiki/Parallel_communication for more information.

Serial: Each bit gets transmitted one bit at a time on the same signal line. That is, each bit is sent sequentially until the full message is transferred across. Obviously this uses far fewer pins than parallel communication does. With the bits being transmitted sequentially serial connections do not have issues with clock skew and can go longer distances. Practically all external communications links are serial today. See https://en.wikipedia.org/wiki/Serial_communication for more information.

Note that many modern high speed communications are serial devices but mimic parallel ones in having multiple channels of serial. In this class we will be exclusively using serial communications.[5]

1.1 Circular Buffers (Serial FIFO Buffers)

Given the mismatch between the speed of the communications on the medium, and the speed at which the microcontroller operates makes it such that you need buffers for both input and output streams. There is dedicated hardware (the UART, see Sec. 6) to handle the serial transactions, which operates on a byte at a time. If each processor/device was waiting on each byte, there would be no time to handle any other tasks. Instead, inbound bytes are added to the receive (RX) buffer, and outbound messages are placed into the transmit (TX) buffer which then feeds the hardware.

These buffers are set up as FIFOs (First In First Out), and need to be limited to some fixed size. New inbound data is appended at the end of the buffer, and read from the front. A simple array could be used as this buffer, but it would require moving all the data in the buffer each time a read occurred. This is incredibly inefficient, and will thrash the memory using `memcopy`. Instead, the concept of front and back are treated relatively using circular buffers.

[4]See the wikipedia article for more information: https://en.wikipedia.org/wiki/Data_transmission.

[5]If time permits, we will cover parallel communications at the end of the quarter. It is doubtful you will be using it with a microcontroller unless you are interfacing with a legacy device.

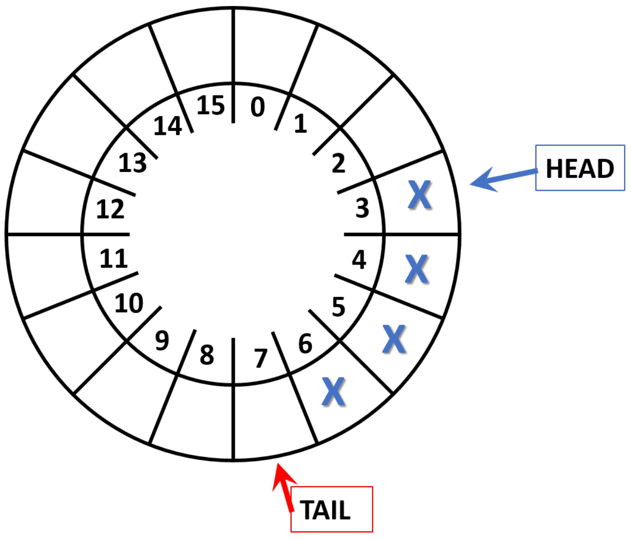

Figure 1: Circular Buffer: this is a graphical representation of a small circular buffer created from an array of length 16. The `HEAD` index points to the first valid data, and the `TAIL` index points to the next available empty slot in the buffer. The blue X's represent valid data. As data is added to the buffer, TAIL is incremented; As data is read from the buffer, HEAD is incremented. It is a *circular buffer* because the `HEAD` and `TAIL` indices will wrap around the boundary at the end.

Fig. 1 shows a graphical depiction of a small circular buffer, with its two associated pointers (`HEAD` and `TAIL`). Valid data is shown with blue `X`'s, and the rest of the buffer is empty. In the basic functionality, new data is added at tail (and then tail incremented, indicating the next available slot). Data is read from the head pointer (and then the pointer is incremented, consuming the data). When the tail and head pointers point at the same location, the buffer is empty; If the tail is immediately behind the head, then the buffer is full. It is an implantation (and application) choice as to how to handle buffer overruns (when trying to add data to a full buffer).

One method is to discard incoming data until there is room on the buffer (i.e.: the next read). The other is to overwrite the oldest data. Which one you choose will be entirely dependent on how you want to handle the overrun. Most commonly incoming data is discarded until there is more room on the buffer.

Conceptually, the head and tail pointers chase each other around the circle in the same direction as data is added and read from the buffer. The astute reader will pause and wonder how do you accomplish this using standard datatypes. The answer is to create a fixed length array, and use two integers as head and tail. Code Listing 1 shows an example in C with a data type of unsigned ints.

Revisiting Fig. 1, for this specific drawing the length of the buffer array is 16 (with valid indices being from 0 to 15—they are labeled on the interior of the drawing). `HEAD` = 3; thus the next data to be read is `Buff[HEAD]`. `TAIL` = 7; thus the next slot to write data into the buffer is `Buff[TAIL]`. There are four valid datum stored in the buffer. If you were to read from the buffer, `HEAD` would be incremented to 4, and the data stored in Buff[3] returned (copy the data out first, then increment `HEAD`); the buffer now contains 3 elements. If another datum came in to be added to the buffer, then Buff[7] would get the new data, and `TAIL` would be incremented to 8; and the buffer would now be back to 4 elements.

Warning: This code is for example *only*. Do NOT copy this into your own code; it will not work for what you need to do. Your own code will look *similar* to this one, but with different data types depending on what you are storing in your circular buffer.

Listing 1: Circular Buffer Example

```
static struct {
    unsigned int head;
    unsigned int tail;
    unsigned int data[MAX_BUFFER_LENGTH];
} circBuffer;

void AddData(unsigned int inData) {
    circBuffer.data[tail] = inData;
    circBuffer.tail = (circBuffer.tail + 1) % MAX_BUFFER_LENGTH;
}
```

The initialization of the circular buffer is to set both head and tail to 0 (the first element of the array). The code listing includes how to add data to the circular buffer and how to handle the rollover at the end of the array. Note the use of the % operator (modulus) to do the exact right thing of wrapping from the end back to the beginning.[6]

The test for an empty buffer is to check if head and tail are equal. Likewise a full buffer would be when the tail is immediately behind the head. Note here that we DO NOT use a separate variable to encode how much data is stored in the circular buffer; all the information is available from simple arithmetic using the HEAD and TAIL pointers.

Reading from the buffer is the symmetric operation to the example adding data, instead the head pointer would be incremented and again wrapped back to zero if it went off the end. Note that the buffer can be any length up to the full range of an unsigned int.

Spend some time coding a small example of a circular buffer of chars and adding and reading data from it. For example, put in the string "Hello, World!" and then read it back out, and add it back in again, etc. Add in some error checking (for full and empty); use your debugger to examine the memory and see how everything is getting stored. A good understanding of the circular buffer will go a very long way to making this lab go easier.

Prelab Part 2 (Circular Buffer)

Write pseudo-code of how to do the following operations on a circular buffer of length `BUFF_LEN` (assume that `BUFF_LEN` is `#defined` somewhere else in the code):

- Initialize the circular buffer
- Add an item to the circular buffer (enqueue)
- Pull an item off the circular buffer (dequeue)
- Test if the circular buffer is empty
- Test if the circular buffer is full
- Return the number of items in the circular buffer

[6]There are other ways to do this, but the % operator is easily the cleanest. For example, you can use the ++ pre-increment operator and test explicitly: `if(++TAIL >= MAX_BUFFER_LENGTH) { TAIL = 0; }`, but compared to the end of the code it seems clumsy.

HEAD	LENGTH	(ID)PAYLOAD	TAIL	CHECKSUM	END
1	1	(1) <128	1	1	\r\n

Table 1: Protocol Packet

2 Packet Communication on Top of Serial

So far, we have gone over the background of digital communications, and how to work with circular buffers. We will use these circular buffers to catch incoming and queue outgoing bytes to/from the hardware. That is, your communications software will have *at least* an incoming receive circular buffer to catch bytes that are inbound to the micro, and an outgoing transmit circular buffer (assuming you want to be able to queue up more than one byte). There are times when you will need more than these basic two, but that will be addressed later.

Many communication protocols use *packets* on top of the serial link. That is, data is grouped into a payload which is then combined with additional information such as ID, header, tail, error checking, etc. This is done for a variety of reasons: (i) routing messages to different parts of the network, (ii) flow control, (iii) guaranteeing some level of data checking, and (iv) enabling filtering on the packet before decoding.

When using data streams where the medium can have lots of error (e.g.: RF communication), the data must be robust to noise, loss of access, recovery from mangled data, and provide a method to ensure data integrity. This is typically done by superimposing a packet structure on top of the serial stream such that some of the data bandwidth is used to improve integrity. For example, having a predefined *head* that has a specific bit pattern allows the receiver to recover decoding the message if it looses the data stream (e.g.: from having too much noise on the medium). The receiver keeps discarding bytes until it sees a valid *head* and then begins processing. If a message is lost midstream the application is able to recover the next message.

In effect, packetizing allows you to group the data into manageable bundles that are predefined. This has the effect that the receiver can examine some subset of the packet before deciding what to do with it. These can be grouped into logical sets that make sense given a particular application. In this lab, we will be instantiating a specific protocol designed to ensure data integrity and to enable the use of the *ECE121 Console* application on the PC to interact with your microcontroller.

3 Protocol Packet Definition

Each and every protocol must begin with a specification for the structure of the packet. Typically these are field names and specified lengths (in bytes, though not always). This goes back to the Sec. 1 to the part about mutually agreeing on symbols. Packetization is also used to compress data into the smallest number of bits you need to transmit the relevant information. Some packet definitions break the packet at arbitrary boundaries.[7]

The protocol definition used in ECE121 is shown in Table. 1, and to make it a bit easier, we break the packet up into byte size boundaries. That is, every field is an integer number of bytes. Using the figure as a reference, the packet consists of several parts:

1. **HEAD** – 1 byte, defined in `protocol.h`
2. **LENGTH** – 1 byte, length of *payload* in bytes

[7]For example, the CAN message definition uses an 11 bit identifier, i.e. the first byte and the following 3 bits, the remaining bits of the second byte in CAN are the data length code and some other signaling bits.

3. **PAYLOAD** – variable length, max limit defined in `protocol.h` – (≤ 128)
4. **TAIL** – 1 byte, defined in `protocol.h`
5. **CHECKSUM** – 1 byte, checksum over the payload, see Sec. 4
6. **END** – 2 bytes, defined to be $\backslash r\backslash n$[8]

Note two important things: the first is that payload is simply a byte array of specified length, and second that the checksum is *only* for the payload (that is, it does not include the head or length or tail into its calculation.

The payload is structured such that the first byte of the payload will be the **Message ID**, and the subsequent bytes of the array will contain the information. These definitions are in the comments in `MessageIDs.h`, and are each unique to their IDs. That is, the protocol is itself agnostic about the contents of the payload, and cares only to the extent that it can encode or extract the correct number of bytes (as specified in the length field). We will examine a specific Message ID and construct a packet to demonstrate this in Sec. 5.

4 Checksum

One of the most important things about packetized communications is the ability to do a data integrity check and confirm that the received data is (to a high statistical certainty) identical to the transmitted data. This is needed because *every* medium that can be used for communications will always have noise and distortion such that messages will occasionally be garbled in transmission.

The way in which this is done is to have a special code included at the end of the packet that can be independently calculated by the receiver. If the one calculated by the receiver matches the one sent in the packet, then the message is considered to be valid.

There are many variants of these codes, from simple parity checks to cyclic redundancy checks (CRC) to error correcting codes that can recover the original data. See https://en.wikipedia.org/wiki/Checksum for more information.

In our protocol, we will use a relatively simple checksum to validate our messages. Messages that fail the checksum are discarded. The checksum we are using is an open source standard from the Berkeley Software Distribution (BSD), one of the many open Unix distributions.

The BSD checksum computes a checksum by adding up all of the bytes (unsigned chars) in the input data stream. Because addition is associative, another packet with the same bytes in a different order would still produce the same checksum, even though they were clearly not the same packet. In order to avoid this, the BSD checksum does a circular rotation to the right by 1 bit between each add.

The pseudo-code for the 8-bit BSD checksum is found in Algorithm 1.[9] The only tricky part is to implement the circular rotation (something available in MIPS assembly but not in C). This is done by right shifting the checksum by 1 and adding it with the checksum left shifted by 7. This has the effect of moving the LSB to the top position.

Warning: This algorithm will only work if $cksum$ is an unsigned char. If the memory associated with it is larger than 8 bits, than you will need some additional masking to limit the output to strictly 8 bits (i.e.: anything above bit 7 needs to be forced to 0).

[8]The end is chosen to be $\backslash r\backslash n$ so that if you are dumping the data stream to a terminal program, each packet will be on its own line, since $\backslash r\backslash n$ is the newline command.

[9]The normal BSD checksum is going to be 32 or 64 bits long, however that would be overkill for our protocol.

```
Algorithm 1: BSD Checksum
Input: char $*str$, input array
Result: unsigned char $cksum$, BSD checksum
Require: cksum$\leftarrow 0$;                         // initialize cksum to 0
for $i \leftarrow 0$ to $len(str)$ do
    $cksum \leftarrow (cksum \gg 1) + (cksum \ll 7)$;    // circular rotation
    $cksum += str[i]$;                                 // add byte
end
return $cksum$
```

A quick inspection of the BSD checksum algorithm will show that it can just as easily be set up as an iterative function that is fed one character at a time, along with the current checksum. Code up your iterative 8-bit BSD checksum on the micro and check it using various payloads to convince yourself that this is correct.[10]

In the *ECE121 Main Interface Console*, under the "Utilities" tab, there is a packet builder that allows you to generate the packets and calls out the payload and checksum. Note here that the checksum is *only* for the payload (which includes the message ID as its first byte). Refer to Sec. 3 for a full definition of the packet.

Some quick examples (`payload` $\rightarrow$ `checksum`):

$$\begin{aligned} \texttt{0x817F} &\rightarrow \texttt{0x3F} \\ \texttt{0x807F35} &\rightarrow \texttt{0x14} \\ \texttt{0x8400257D96} &\rightarrow \texttt{0xE6} \end{aligned}$$

Note that the packet builder helpfully includes a C array that can be copied into your code for checking your algorithm. This checksum calculation is an integral part of the protocol library. Ensure that your function works and delivers the correct checksum.[11]

5 Payload Specifications and Message_ID

As stated in Sec. 3, the packet *payload* defined in our protocol consists of an unsigned char array (up to 128 bytes long) with the first byte being the Message ID and the remainder of the payload being defined by that particular message ID. That is, the *interpretation* of the remaining bytes of the payload is defined in the `MessageIDs.h` header file.

5.1 Example: ID_LEDS_SET

For example, we will assemble (by hand) an **ID_LEDS_SET** message. This message instructs the microcontroller to turn on any LED where the payload bit is 1 and off where the payload bit is 0. Note that while you can

[10]If you really want to understand the BSD checksum algorithm, take two small payloads and hand calculate it in binary. This will go a very long way to helping you debug your own implementation of the algorithm. It is painful to do it by hand, but well worth the effort.

[11]Do **NOT** test your code with only the examples above; make sure you generate packets using the packet builder and test against those. Don't be lazy, you want to be pretty exhaustive in your testing.

use the packet builder in the console, it is much more instructive to do it by hand a few times to understand how a protocol packet is assembled.

The message begins with **HEAD** and then has a **LENGTH** of 2, the payload consists of the **ID**, followed by a single byte, then the **TAIL**, **CHECKSUM**, and $\backslash r\backslash n$. Using the definitions in `Protocol.h` and `MessageIDs.h`, we have the message begin as: $\begin{bmatrix}\texttt{0xCC} & \texttt{0x02} & \texttt{0x81} & \texttt{0xFF} & \texttt{0xB9} & \ldots\end{bmatrix}$ where the `0xCC` is the message head, the `0x02` is the length code, the `0x81` is the message ID, the `0xFF` is the message payload (in this case, turn all LEDs on), and `0xB9` is the message tail.

Using the definition in Sec. 4 we calculate the BSD checksum to be `0xBF`.[12] Thus the fully assembled protocol package is: $\begin{bmatrix}\texttt{0xCC} & \texttt{0x02} & \texttt{0x81} & \texttt{0xFF} & \texttt{0xB9} & \texttt{0xBF} & \texttt{0x0D} & \texttt{0x0A}\end{bmatrix}$, where the last two bytes are the $\backslash r\backslash n$ in hex.

Check this for yourself, and try it out with a few different LED patterns to see if you can do this by hand. Check your results using the *ECE121 Console* packet builder.

5.2 Example: ID_PONG

Another rather simple message is **ID_PONG**, which is the microcontroller's answer to the inbound **ID_PING** message from the console. These two messages, PING and PONG, will serve as the application for protocol in the final part of this lab (see Sec. 8).

The **ID_PONG** answers the **ID_PING** by taking the signed int payload of **ID_PING** and dividing it by 2, and putting that into the **ID_PONG** message. Details of this are discussed in Sec. 8.2 and Sec. 8.3.

For now, assume the number calculated to be returned is 4443432 (which is `0x0043CD28` in Hex). Thus, putting the message together as in Sec. 5.1 we get that the message for the beginning looks very similar: $\begin{bmatrix}\texttt{0xCC} & \texttt{0x05} & \texttt{0x85} & \texttt{0x0043CD28} & \texttt{0xB9} & \ldots\end{bmatrix}$ where the `0xCC` is the message head, the `0x05` is the length code, the `0x85` is the message ID, the `0x0043CD28` is the message payload (in this case, pong with 4443432), and `0xB9` is the message tail.

Again, using the definition in Sec. 4 we calculate the BSD checksum to be `0xB7` (this will be harder to do by hand, but you can use the packet builder to confirm it). Thus the fully assembled packet is:

$$\begin{bmatrix}\texttt{0xCC} & \texttt{0x05} & \texttt{0x85} & \texttt{0x00} & \texttt{0x43} & \texttt{0xCD28} & \texttt{0xB9} & \texttt{0xBF} & \texttt{0x0D} & \texttt{0x0A}\end{bmatrix}$$

Where the last two bytes are the $\backslash r\backslash n$ in hex.

6 Serial Communications (UART)

Much of the previous sections has been a rather abstract background and highly specific details about the protocol, without any direct method for how to use it. In this lab (and for the rest of the class) all of our communication to the *ECE121 Console* application will be through the serial communications port. The hardware that controls the serial port is called the UART (Universal Asynchronous Receiver/Transmitter).

[12]Again, it is **very** useful to calculate this out for a few different messages by hand. It is not difficult (especially for a 2-byte payload) and will give you confidence in your iterative checksum algorithm.

The nuances of how the UART works, what signaling levels, how to indicate bus idle, when to sample, how to start a transaction, and how to end the message are all beyond the scope of this lab manual. You will go over many of these specifics in lecture, and be sure to review the material.

Before proceeding further in the lab, make sure you have reviewed the PIC32 Family Reference Manual Section 21, UART, as well as your notes from the class. While the UART subsystem is one of the more simple ones on the PIC32, it nevertheless requires a bit of care to get it working well. You will also need to understand interrupts and the interrupt system, see FRM Section 8, Interrupts as well.

In the following, we will lead you through a step-by-step incremental walk to getting the serial port working (and note that in later labs, we will not break it down for you so completely).

6.1 Serial Port Configuration

The first step to getting the serial port to work is to get the configuration of the serial port done correctly. The settings required are a baud rate of 115200, 8 data bits, no parity, 1 stop bit (8-N-1). For this lab, you will be using UART #1.

The main steps for configuring the UART 1 are:

1. Clear the control registers for UART 1
2. Calculate the Baud Rate Generator value to generate 115200
3. Set UART 1 for 8-N-1
4. Enable UART 1
5. Enable Transmission (TX)
6. Enable Reception (RX)

For this part of the lab, you are working with `Protocol.h/c` and should have created a simple project with those two files. You will be writing a `main()` function inside `Protocol.c`, which should be conditionally compiled based on a `#define` flag. This will eventually be your test harness (see Sec. 7.5) and must *always* reside within the library .c file. Use a `#ifdef IF_TESTING` ... `#endif` to force the conditional compilation when required.[13]

Info: The XC32 compiler has many features that make interfacing to the hardware much simpler. For example, you can create the correct mask for the `U1MODE` register to achieve the desired settings, or you can address each bit directly by name (matching the name in the FRM) by using `U1MODEbits.XXX`, where `XXX` is the name of whatever subfield in the FRM. If the bits don't pop immediately in the MPLAB-X IDE, make sure you `#include <xc.h>`.

You will be using any simple terminal program (not the *ECE121 Main Interface* console) such as Putty to interface with the micro at this stage. Once you are using the protocol, then this will no longer be needed as you can use the *ECE121 Console*.

Configure Putty (or other serial program) to the same 115200, 8-N-1, and make sure it has the correct COM port open (opening the ECE121 *Console* will tell you the current serial ports, be sure to close it afterwards).

[13]Please do **NOT** use the actual text "IF_TESTING" as this is clearly a stand-in for something more appropriate. If we see you using it, we will dock your score. Don't be lazy.

Prelab Part 3 (Serial Port Configuration)

How are you going to configure the serial port (UART 1)? List all registers that you will use to configure the serial port to 115200 baud, 8-N-1 (assuming a peripheral bus clock of 40MHz), and the value you will set those registers to. You can either use the registers directly (e.g.: `U1MODE`), or their bit versions (e.g.: `U1MODEbits.ON`). You need to include enabling both a transmit interrupt and receive interrupts.

6.2 Blocking Transmission using PutChar

The first step to testing your serial communication is to send a single character to your Putty serial terminal on the PC. In order to do this, you need to define a function, `PutChar`, which will put a single character in the UART transmit buffer which enables the transmission. The function prototype is already in `Protocol.h`.

Your code will be inside your `main()` loop, and will poll your UART1 status to determine that the single character (pick any printable one at random) has been sent and it is ready for the next character. You should see the character appear on the Putty window.

Once you can reliably send a single character, modify your `main()` loop to send all of the characters in a string in succession ("Hello, World!" is a good example). That is, iterate through every character in your string, copy the character to the UART1 transmit buffer, wait for the character to be sent, and then send the next one. Note that this will be blocking code, since you are polling the status of the UART1 to determine if it can accept the next character. Again, you should see your string in the Putty window.

6.3 Interrupt Driven Transmission

Of course, blocking code means that your processor cannot do anything at all except send characters out the serial port. This is, of course, not how it is done. The UART subsystem is fully capable of handling serial transactions in an interrupt, thus freeing your processor to do other things.

You will need to go back to your configuration code and add an interrupt priority (`IPC6bits.U1IP = X`, X is priority level), and enable the transmit interrupt. You will also need to write the interrupt handler, which requires the function declaration to look like:

```
void __ISR(_UART1_VECTOR) IntUart1Handler(void) {
```

This line is given to you because it is quite complicated to figure out the `_UART1_VECTOR` is the correct one to use. There is only one interrupt from the UART, and that both the receive and transmit interrupt will trigger the same interrupt (you will have to check the `IFS0bits.XXX` flag to see which one interrupted).

Begin by having your interrupt handler light an LED to demonstrate to you that your code got in there. Then write your interrupt handler such that you can transmit the same character (e.g.: 'a') over and over each time the interrupt occurs. This is going to be ugly code, but good to learn how the interrupt works.

Modify your `main()` so that you can transmit a string (e.g.: "Hello, World!") as before, but this time using interrupt driven transmission. That is, your interrupt needs to grab the next valid character from the string until none are left. You will need a module level variable that keeps track of which character of the string you are on. It will be modified in the interrupt. Alternately, it could be a static variable inside the interrupt, but either way you need to keep track of where in the string you are. Note that you might have to force the interrupt to start things going.

Info: The UART is only going to initiate a transmission when it has data in the buffer (`U1TXREG`); at the end of the transmission the UART will then generate an interrupt (which should grab the next character). The problem here is that you need to **start** the transmission (by copying a character into `U1TXREG`).

The easiest and cleanest way to do this is to have the interrupt itself check if there is more data to transmit, and load the next chunk of data into `U1TXREG`. This means that you need to *force* the interrupt when you are ready to begin transmission (by setting `IFS0bits.U1TXIF`).

6.4 Circular Buffer for TX

It should be pretty clear by now that you need a transmit buffer to hold the characters in while the UART is busy sending data to the PC. The best way to do this is to instantiate a circular buffer to handle the data stream. This will be your transmit (TX) buffer, and the data type of the array should be unsigned chars. When you create the circular buffer, use a `#defined` constant for the array length. This will make it easy to change if you need to modify it later.

For outbound transmission, your circular buffer is simply a character array, since the UART can only transmit one byte at a time (what you place into the buffer will be dependent on the specifics of your application).

Your interrupt handler should get the next character from the buffer at the head pointer, and your `PutChar` function should be inserting characters at the tail pointer. Both functions should increment their respective pointers and check for collisions or empty buffer.

That is, `PutChar` should write to the current tail of the buffer, increment the tail buffer (and wrap it around the fixed length of the array). `PutChar` should check if the buffer is full before writing. If it is, simply discard the data. If the UART1 is currently idle (that is, there is no current byte being transmitted), `PutChar` should force an interrupt to get the transmission started.

Info: The UART system will often have to have the transmission "kicked" in order for it to start if it is not currently transmitting. As above in Sec. 6.3, you need to force the interrupt to grab the next character. In this specific case, it is done when you have added an new character to the circular buffer (using PutChar), and the UART is idle. You can check if the UART is idle using `U1STAbits.TRMT`; if it is idle, force the TX interrupt to occur by setting `IFS0bits.U1TXIF`.

The UART1 interrupt handler should first check if the buffer is empty, and if not, then send the character at the head pointer, increment the head pointer (and wrap it around the fixed length of the array).

Warning: When dealing with interrupts, remember that they can occur at any time even while other parts of your code are doing something unsafe to the data. You will need to create a flag that protects the buffer from collisions while `PutChar` is modifying it by adding to the circular buffer that the ISR can check. This will become relevant in Sec. 7.4, and will be covered in lecture.

Assuming that you have done all of this correctly, you should be able to send any arbitrary string using your `PutChar` function and the `sprintf` function in C. Test this out with many different strings; you should see the output on your Putty terminal.

You now have serial transmission (TX) working, and should be able to verify this with Putty. There is still no protocol running, but at least you can now overlay any protocol specific data on top of your working serial subsystem. The basic system of being able to transmit bytes using the ISR is working; the protocol will ride on top of this basic functionality in that the stream of characters send will have very specific definitions (see Sec. 3). Demonstrate that you can send arbitrary strings to the Putty terminal to the TAs for a checkoff.

6.5 Receive and Echo Characters

The protocol is bi-directional; packets go out, but also come in. Thus we need to have the receive (RX) part of the UART subsystem working as well. In order to get reception working, you will need to revisit your init code to ensure that the UART1 interrupts when the RX hardware buffer is not empty, and enable the UART1 RX subsystem.

The easy way to validate if you have the serial port RX working is to echo characters received from the Putty window back. That is, inbound RX should be copied to the transmit circular buffer and send back. Do this in your `main()` function, not inside the interrupt.

Warning: We are asking you to do something a bit weird here. Because you don't yet have a receive buffer (and you won't actually ever get one), you will need to copy the incoming data to a variable. You will then check that variable in `main()`, and if it is not `NULL`, then use `PutChar` to send it out and clear the variable. Do NOT simply resend the character inside the interrupt.

Write code that does this, and test it thoroughly (sending and echoing various characters). A problem here at this stage will haunt you later, so ensure that everything works very well at this point. Again, show this to the TAs for checkoff.

7 Protocol Library

At this point in the lab, you have a working serial port that can send out arbitrary strings using an interrupt-based handler, a circular buffer to contain the transmissions, and a receive function that does an echo of incoming characters. This is all well and good, and you should feel good about your accomplishments so far.

This is the last time in this class that you will use the serial port "raw." From here on out, you will be using the protocol for everything else in the class, including the rest of this lab. That means that extra care should be taken to get this code right, as errors here will make later labs difficult to implement.

Info: The choice of what you are going to put into your receive circular buffer has some potential performance and ease of coding ramifications that you need to consider. Every message has the same HEAD, TAIL, and END, which do not necessarily need to be stored. However, the length, payload (including ID) and the CHECKSUM will be different for each message. Note that the checksum also need not be stored, but does need to be validated before the message can be accepted. The elements of your circular buffer can be structs (minimally storing length and payload), or you can have multiple circular buffers (2D arrays) which are storing different parts of the message (e.g.: length and payload) but have the same `HEAD` and `TAIL` pointers; this is a design choice for you to make.

7.1 ECE121 Main Interface Console

From here on out, throughout this lab and all subsequent ones, you will be relying on the *ECE121 Main Interface Console* to interact with your microcontroller. The *ECE121 Console* is custom python code that has been written by the course instructor to enable this class to work. This means that it is (still) a work in progress (and we appreciate bug reports). It has several features we felt were useful for your debugging requirements (such at the Packet Builder), again if you feel something is missing, let us know and we will see if we can add it.[14]

The console has several tabs, and a status bar at the bottom. The "Serial Control" tab allows you to connect/disconnect from the micro, and will rescan the ports on the machine. This is less of a requirement on the lab computers, but will be very useful if you are running on your own laptop. The status appears on the lower right, and tells you which COM port you are running on, if you are connected or disconnected, and keeps a running sum of Received and Transmitted messages.

There are two lower panels, labelled "Incoming Packets" and "Outgoing Packets" respectively. They will print the raw packets as they come through regardless of which tab you are on, and can be cleared using the labeled buttons.

You should already be familiar with the "Utilities" tab, which has two sub-tabs, "Packet Builder" and "Data Logging." The packet builder was referenced in Sec. 4, and can be used to create any packet. The packet builder will break out the data, head, length, and checksum. Note that the packet builder does not check for compliance with the message ID definitions (e.g.: it will be perfectly happy to send an ID_LEDS_SET with a payload of 4 bytes after it). Data logging will be useful in later labs.

7.2 Protocol TX

The implementation of transmitting a given message using the protocol is fairly straightforward. Given that your `PutChar` function adds the character to the TX circular buffer as noted in Sec. 6.4, and you are given a message to transmit (ID, length, and a pointer to the payload), the steps are:

1. Enqueue the HEAD
2. Calculate correct length of payload argument + 1 (for message ID)
3. Enqueue the length
4. Initialize the checksum with the `ID`
5. Enqueue the `ID`

[14]Understand that we are way overloaded preparing a new class like this; we might not get to adding in the requested feature this year, or we might not agree that it is a good think, or it just might be too difficult to do.

6. Enqueue the data one byte at a time (and update your checksum)
7. Enqueue the TAIL
8. Enqueue the CHECKSUM
9. Enqueue $\backslash r \backslash n$

Enqueue is (formally) to put the character into the circular buffer (at the TAIL pointer); this is done with your `PutChar` function. The length argument is a bit tricky, as every payload includes a message ID which takes 1 byte; there is an ambiguity as to what length to use. Since the protocol is agnostic as to what is contained within the payload field, the length is the *entire* payload (including the message ID).

Since `PutChar` should be very very fast (it does not do much) then the entire transmission process should be very fast indeed. You should consider a semaphore or mutex to ensure that the TX circular buffer is not compromised by a call to transmit within an interrupt (see Sec. 7.4).

Code up your `Protocol_SendMessage()` function and test it using the *ECE121 Console*. Test your function thoroughly with many different messages to ensure that your function works as expected. You can use the packet builder to generate appropriate packets and to test them when generated by your micro. Note that the definition of length in the arguments for the `Protocol_SendMessage` *excludes* the message ID.

7.3 Protocol RX StateMachine

Looking at the protocol structure in Table 1, it should be clear to you that the receiving of an inbound message will require a state machine to process the byte stream into a message.

The state machine for receiving messages must run each time a new byte is received from the UART.[15] Thus, at each time the state machine is called, it has a new character to examine.

Warning: The function `Protocol_RunReceiveStateMachine` is already defined in the `Protocol.h` header). It needs to run inside the UART1 interrupt and get the next character received. Because it is running inside an interrupt, the RX state machine needs to be fast and atomic.

Obviously, the first thing to do is to wait for the HEAD byte. Until that happens, all incoming data should be ignored. Once the HEAD has been received, a new message is coming in and needs to be stored somewhere. The next character after HEAD will be the LENGTH byte. You will need to grab the length of the payload and store it in an appropriate place. The next i characters (as determined by the length you received) are the payload, and must be processed and stored. Note here that you want to calculate the checksum iteratively on every additional character received.[16]

Once you have received the correct number of bytes, the next one *must be* the TAIL; if you don't receive a tail, then the message is invalid and the state machine returns to waiting for head. If you get a correct tail, then the next step is to validate the message using your calculated checksum and compare it to the one received

[15]Unlike in your previous classes, we will not be providing you with a drawing of the state machine, rather, you will need to come up with it on your own.

[16]You can calculate the checksum all at once when you receive the TAIL, however this can lead to a potentially long time tied up inside the interrupt if the message is long. It is better to calculate it iteratively as each character comes in; this ensures that the interrupt is atomic and short.

from the serial port. If the message is valid, place it on the "Incoming Packet" circular buffer; if the message is invalid, discard the message (and perhaps report a protocol error).[17]

At this point, the state machine should return to waiting for the HEAD, and it runs all over again. Note here that the protocol does not handle the payloads (with some exceptions, see Sec. 7.4). These are handled by the application layer, as the protocol is agnostic about what the contents of the payload mean.

Draw out your state machine, and implement it in C. Test in receiving from the *ECE121 Console* application. You can use the packet builder to generate a C array of a valid packet to test your state machine, and then eventually test it by having the console send the packet over the serial port to your UART1.

Info: Once your state machine has completed (and reset for the next incoming packet) you will have a validated packet. You will need to store both the LENGTH and the full PAYLOAD (including the message ID) into your Incoming Packet circular buffer. Since you need to store two items (an unsigned char length, and a character array payload), you can do this either with a struct, a 2D array, or two circular buffers that share HEAD and TAIL pointers.

Prelab Part 4 (Protocol RX State Machine)

1. Draw the full RX state machine that will run inside the UART receive interrupt. Ensure that you use descriptive names for your states, and use the "event/action" paradigm on your state machine. Note that the "event" that it is called with is always a the character received. See `Protocol.h` for details.
2. Sketch out (in pseudo-code or other method that is clear) how you will structure your Incoming Packet circular buffer, and how you would insert or extract `length` or `payload`.

7.4 LEDS_SET and LEDS_GET

There are two special message IDs that are consumed by the library and never get added to the Incoming Packet circular buffer, but rather are handled within the protocol (and in fact within the RX state machine): `ID_LEDS_SET` and `ID_LEDS_GET`.

If the payload first byte (ID) is `ID_LEDS_SET` and the message is valid, then the LEDs are turned on wherever there is a 1 in the bit pattern of the second payload byte (first byte after ID). Because the message is immediately consumed within the protocol, there is no need to add it to the Incoming Packet circular buffer.

If the payload ID is `ID_LEDS_GET` and the message is valid, the message is also consumed immediately and not added to the Incoming Packet circular buffer. The response, however, is slightly more complicated than that of `ID_LEDS_SET`; for the `ID_LEDS_GET`, the micro must immediately respond with an `ID_LEDS_STATE` message with the payload being the state of the LEDs (the latch for Port E). This can be accomplished by sending `ID_LEDS_STATE` directly within the interrupt. You will want a semaphore or mutex to ensure that no one else is writing to the outgoing circular buffer at the time.[18]

[17] Since the payload of the protocol packet has a variable length, you need to have an element of your circular buffer that is itself an array of characters. The best way to do this in an embedded system is to force each the arrays to be fixed length of a `#defined` maximum that is the size of the maximum length payload you will every receive.

[18] You can think of a semaphore (or mutex) as a way to keep only one thread accessing the resource at once. An analogy would be a "talking stick" where only the one holding the stick is allowed to talk. When writing to the TX circular buffer, the software writing needs the "talking stick" so that no one else (the interrupt) can write until it is done.

Warning: Be careful with immediate transmission (adding to the TX circular buffer) of the `ID_LEDS_STATE` message inside the interrupt service routine. If the code is in the middle of writing an outbound message, the two messages will step on each other and corrupt the output stream. It is best to use a semaphore of some sort to keep them from stepping on each other.

You can test the `ID_LEDS_SET` and `ID_LEDS_GET` using the *ECE121 Console* under the Lab 1 tab. Again, you can generate a message (as a C array) using the packet builder to test, and once confident, run it through the console and make sure everything responds correctly.

7.5 Protocol Test Harness

With the various parts of the Protocol.c implemented, you are now ready to complete the protocol library including its test harness. Having a decent test harness is very important to any code module, but it is especially important when dealing with embedded modular code.

A test harness, simply put, is a `main()` that exercises the functionality of the module and can be used to validate that it works. Good test harnesses are designed *before* the implementation code is written. If you think about it, you don't need to know how things are implemented to test them (presuming you have a well documented header file).

In the embedded world, you are *highly* recommended to keep your test harness within your module .c file. In this class, it will be a hard requirement. The way that you are able to have a `main()` within your module, but still be able to use that very same module elsewhere is using conditional compilation.

Conditional compilation is the construct `#ifdef PROTOCOL_TESTHARNESS <your test code> #endif` that is handled by the preprocessor. If `PROTOCOL_TESTHARNESS` is defined then the code is compiled, if it is not, then the preprocessor skips over to the line after `#endif`. The key to making the modules work both as stand-alone test harness and also to be included within another project and still work is to have the `#define PROTOCOL_TESTHARNESS` not be in the file itself, but rather set at the project level.

This can be handled using configurations within MPLAB-X. The point here is that the macro (#define) is defined in the project, not in the code. This allows you to include that very same code (with no alterations) into another different project, and main will be suppressed. To do this, in MPLAB-X navigate to customize → conifiguration → gcc-xc32 → preprocessor macros; define the macro `PROTOCOL_TESTHARNESS` there (and please don't use that name, it is just a stand-in).

Now, onto the actual assignment. Write your test harness for the protocol library. The test harness should, at a minimum, respond to `ID_LEDS_SET` and `ID_LEDS_GET` in the appropriate way. Note that using these two messages will test most of your protocol library. Test your test harness using the ECE121 Console, and make sure it responds correctly.

Demonstrate to your TAs that your protocol library works and that the LEDs respond. Write up in your lab report what you decided to test and how (and why).

Congratulations: you have managed to get the protocol library working on your embedded microcontroller. This is no small feat, and you will be using the protocol extensively throughout the rest of the quarter.

8 Protocol Application

In Sec. 7 you completed the protocol library test harness, and verified that it was working. The *application* will use that protocol library to implement some further specific messages.

8.1 ID_DEBUG Message

The first message that is not natively inborn to the protocol module is the `ID_DEBUG` message which is the protocol version of printf. An `ID_DEBUG` has a payload that is an ASCII string that will be printed on the console. Note that it is not terminated by `NULL` since you are already sending the length in the protocol.

You should give yourself a local character array as a destination for `sprintf` which is how you can construct a string to send using the ID_DEBUG message. The packet would then be assembled using the correct ID, the length (as extracted using the `strlen` function), and your string, etc.[19]

You application code should *always* begin with an ID_DEBUG message that includes at a minimum a descriptive title of the code, and the date and time of compilation (these can be accessed using the __DATE__ and __TIME__ macros in the C preprocessor). This small little habit can save you countless hours in the future when you realize immediately that you are accidentally running code that was compiled a week ago.

8.2 Little Endian and Big Endian

All packets with single byte payloads, or ones with a string of bytes (e.g.: the **ID_DEBUG**) will work fine going between any two machines. This is due to the fact that the smallest addressable memory unit is a byte, and thus the order of bits is consistent across all machines.

As soon as you go to a variable that is larger than a byte (e.g.: a short or int), then the byte ordering becomes important. There are only two ways to order the bytes in memory: Little-Endian or Big-Endian. Endian-ness is a historical artifact dating back to far less capable hardware which leveraged Endian-ness for speed.

From the wikipedia article at: https://en.wikipedia.org/wiki/Endianness:

> A Big-Endian ordering places the most significant byte first and the least significant byte last, while a Little-Endian ordering does the opposite. For example, consider the unsigned hexadecimal number `0x1234`, which requires at least two bytes to represent. In a big-Endian ordering they would be `[0x12, 0x34]`, while in a little-Endian ordering, the bytes would be arranged `[0x34, 0x12]`.

For whatever interesting reasons, most network streams have converged on Big-Endian, and most hardware architectures on Little-Endian. To match the prevailing conventions, we have set the protocol to be Big-Endian.[20]

[19] As an example, if the text in the ID_DEBUG message were “hello, world!” then entire message would look like (paying close attention to the length field): `0xCC0E8068656C6C6F2C20776F726C6421B9950D0A`.

[20] It also makes the packet easy to parse in a normal display. When the value is `0x0FFF`, that is what you see, as opposed to the Little-Endian version of `0xFF0F`.

Warning: Be careful here, the Uno32 is little Endian. Anytime you are working with a short or int on the protocol, you will need to swap the byte ordering before interpreting the number or sending it to the console through the protocol.

Note that for ints (4 bytes) the swapping order is complete. Thus `0xDEADBEEF` looks like `0xEFBEADDE` for Little-Endian; the good news is that the swapping is symmetric so converting from one to the other is the same. With some clever masking and shifting, you can handle the ints using the same function that handles shorts.

Write your two functions `Protocol_ShortEndednessConversion` and `Protocol_IntEndednessConversion` and test them using your debugger (or `ID_DEBUG` and the console) to convince yourself that you can effectively swap the byte order.

8.3 PING and PONG

A common function that is required from any networked system is to determine who is connected and if they are still responding. This is often done using a ping and pong message. Typically the central server sends a "ping" message, and the nodes respond with a "pong" message.

The basic idea is that the micro receives an ID_PING message with an int in the payload; the micro divides that value by 2 ($\gg 1$), and then sends it back to the console as an ID_PONG. Because the protocol stream is Big-Endian and the micro is Little-Endian, to get the correct answer you must swap the bytes of the int before doing the shift (or divide) and then swap it again when you assemble the payload. If you do not do this, the result will be wrong.

For example, if you receive an ID_PING message that is `0xCC05840000CC58B9860D0A`, the payload is `0x840000CC58`, or stripped of the message ID, `0x0000CC58` (the int is 52312). In order to correctly do the shift, the int field must be first converted to Little-Endian (`0x58CC0000`) and then shifted by 1 ($\div 2$) to get the correct value of 26156. This must then be swapped again to generate the payload for the corresponding ID_PONG, which would have an int of `0x0000662C` (the full packet would be `0xCC05850000662CB9370D0A`).

Do this by hand (at least once) and see if you get the expected results. See what happens when you forget to swap the byte ordering. Make sure you understand what is happening so you can code this up correctly.

The application layer implements this byte swapping. The *ECE121 Console*, under the Lab 1 tab, has a button to send ping. As discussed in Sec. 5.2, the response to a ping is to divide the payload by 2, and then send it back as the payload for the pong message (correcting for endian-ness). The divide by 2 should be implemented as a right shift by 1.

Warning: The payload for ping is an int in Big-Endian byte ordering. You will need to swap the byte order before shifting, do the shift, and then swap the byte ordering again to send it out.

In summary, your application needs to:

1. Send an ID_DEBUG message that includes compilation date and time
2. Respond to a PING message by replying with an appropriate PONG
3. Respond to an LEDS_SET message by turning on the specified LEDs
4. Respond to an LEDS_GET message by immediately sending a LEDS_STATE message

PING		PONG	
Full Message	(Number)	Full Hex Message	(Number)
`0xCC0584[ABAFAADA]B9180D0A`	$288,041,674$	`0xCC0585[08959565]B96E0D0A`	$144,020,837$
`0xCC0584[649DE730]B9DF0D0A`	$1,688,069,936$	`0xCC0585[324EF39]B9430D0A`	$844,034,968$
`0xCC0584[71E32C52]B9970D0A`	$1,910,713,426$	`0xCC0585[38F19629]B90F0D0A`	$955,356,713$
`0xCC0584[0449E275]B9810D0A`	$71,950,965$	`0xCC0585[0224F13A]B9540D0A`	$35,975,482$

Table 2: Examples of Ping and Ping Messages

Some examples of correct ping and pong queries are shown in Table 2. The hex of the number is called out from the center of the packet using square brackets. Note that *all* `ID_PING` packets begin with `0xCC0584` and that all `ID_PONG` packets begin with `0xCC0585`.

Remember that the LEDS_SET and LEDS_GET are native to the protocol library and are always active, thus the application should respond appropriately to these messages. Demonstrate your application code running to the TAs. Write up your lab report and remember to submit your code through the CANVAS assignment.

In the lab report, ensure that a student who has already taken the class and is fairly well versed in embedded software would be able to recreate your solution. Include details and algorithms, pseudo-code, flow-charts, and any "bumps and road-hazards" that a future student might watch out for. Please be clear and concise, use complete sentences, and maintain a dispassionate passive voice in your writing.[21]

Congratulations: you have completed the first "real" lab of this class. If you were able to do this easily, you are in good shape for the rest of the class. If you have struggled, you should review your C coding, make sure you get extra help from the TAs, and otherwise endeavor to improve your performance.

[21]English teachers hate the passive voice, but it is used as the primary narration in scientific writing as you want the reader to know what was done, but not be concerned with who did it. In this sense, proper scientific writing is *egoless*. If passive voice is too burdensome for a particular sentence, use the "we performed" or other such construct instead.