ECE 121
Lab 1: Protocol

CIRCULAR BUFFERS, SERIAL COMMUNICATION, AND PROTOCOL PACKETS

MEL HO
STUDENT ID #: 1285020
WINTER 2021

January 26, 2021

1 Introduction and Overview

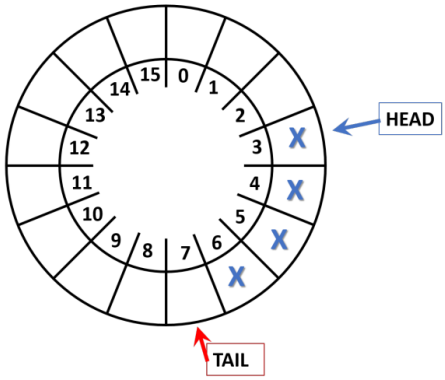

Figure 1: A picture diagram of a **circular buffer**. The **HEAD** index points to the first valid data and the **TAIL** index points to the next available empty slot of the buffer. As the name implies, a circular buffer wraps around, overwriting old data as it is being read. The X's represent valid data while the blanks are either empty space or filled with invalid data to be overwritten. Image taken from the Lab 1 manual.

In Lab1, we designed a serial communication protocol in MPLAB X that our UNO32 uses to communicate to the provided lab interface. This serial communication protocol was implemented using circular buffers, an 8-bit BSD checksum, the UNO32's UART registers, and a standard protocol packet definition given in the lab 1 manual. We tested our protocol library by connecting the microcontroller to the ECE 121 lab interface and sending packets to and from the UNO32 board; using the tests specified in lab 1, we were able to run **ID_LEDS_SET, ID_LEDS_GET, ID_DEBUG, ID_PING,** and **ID_PONG** successfully. While the complexity of this lab proved challenging, we gained a deeper understanding of how serial communication works in conjunction to a microcontroller. For the sake of a concise and cohesive narrative, this lab report combines certain sections of the lab manual when describing the build process. For specifics of code implementation, please refer to the appendix.

2 Serial Communication (UART)

To successfully implement our serial communication protocol for the UNO32, we built our protocol cumulatively. First, we explored how to build a circular buffer, un-

derstand the format of our protocol packet, and create a basic BSD 8-bit cumulative checksum for data integrity. We then applied this knowledge to send and receive messages using our the Universal Asynchronous Receiver/Transmitter (UART) serial port on the UNO32.

2.1 Circular Buffers (Serial FIFO Buffers)

Following the steps from the first prelab in Lab 1, pseudo-code for a basic circular buffer was written:

```
#define TRUE 1
#define FALSE 1

struct circBuffer
{
        uint8_t head;
        uint8_t tail;
        uint8_t data[BUFF_LEN];
}   circBuffer;

// init
void circBuffer_Init()
{
        circBuffer.head = 0;
        circBuffer.tail = 0;
}

// Enqueue
void enqueue(uint8_t inData)
{
        if bufferIsFull()
                return -1;
        circBuffer.data[circBuffer.tail] = inData;
        circBuffer.tail = (circBuffer.tail + 1) % BUFF_LENGTH;
}

// Dequeue
uint8_t dequeue()
{
        if bufferIsEmpty()
                return -1;
        data = circBuffer.data[circBuffer.head];
        circBuffer.head = (circBuffer.head + 1) % MAX_BUFFER_LENGTH;
        return data;
}

// Test if circular buffer is empty
```

```
int bufferIsEmpty()
{
        return (circBuffer.head == circBuffer.tail)
}
// Test if circular buffer is full
int bufferIsFull()
{
        return circBuffer.head == ((circBuffer.tail + 1) % BUFF_LEN)
}
// Return the number of items in the circular buffer
int bufferNum()
{
        int tail = circBuffer.tail
        if (circBuffer.tail < circBuffer.head)
                tail =(BUFF_LEN + circBuffer.tail);

        return (tail - circBuffer.head);
}
```

As shown in Fig. 1, a circular buffer is a type of array where data is stored and popped using a **TAIL** and **HEAD** pointer respectively. Circular buffers are ideal to use for our communication protocol because they not require moving all the data in the buffer every time a read occurs, making them far more efficient than the standard array. Now that we have the skeleton of our circular buffer that we will be using for our serial communication, we can now construct our checksum function and the scaffolding for our protocol.

2.2 Protocol Packet Anatomy

HEAD	LENGTH	(ID)PAYLOAD	TAIL	CHECKSUM	END
1	1	(1) < 128	1	1	\r \n (2 bytes)

Table 1: Protocol Packet. The numbered values indicate the number of bytes each section of the packet are. For our payload, the **MESSAGE_ID** is included in its size with the max value being 128 bytes.

Table 1 shows the the structure of the data packet we will be using for our serial communication protocol. Data packets allow for better flow control, filtering, and integrity check in our communication. Because data can easily be corrupted or contain errors during transmission, a packet structure ensures that our data is valid before being read. To check the integrity of our data, we implemented an 8-bit BSD checksum.

2.3 Checksum

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 $cksum$

Checksums are useful for ensuring that the data sent matches with the data received. They take in every character of the payload and cumulatively calculate a unique hex value for comparison. If the checksum calculated from the received payload fails to match the checksum value sent, the packet is considered invalid and is discarded by the protocol. Using the pseudo-code provided in Lab1, a modified 8-bit BSD cumulative checksum value was implemented into our protocol.c file; instead of feeding a full string into the checksum function, we cumulatively calculate the checksum every time a new character from the payload is processed. This implementation proved to be more efficient and atomic than processing the whole string at once. We use our calculate checksum function to encode and decode packets in our protocol later in the lab.

```
/**
 * @Function char Protocol_CalcIterativeChecksum(unsigned char charIn
   , unsigned char curChecksum)
 * @param charIn, new char to add to the checksum
 * @param curChecksum, current checksum, most likely the last return
   of this function, can use 0 to reset
 * @return the new checksum value
 * @brief Returns the BSD checksum of the char stream given the
   curChecksum and the new char
 * @author mecho */

  unsigned char Protocol_CalcIterativeChecksum(unsigned char charIn,
   unsigned char curChecksum)
{
    curChecksum = (curChecksum >> 1) + ((curChecksum & 1) << 7);
    curChecksum += charIn;
    curChecksum &= 0xff;
    return curChecksum;
}
```

Note: line 13 of our code block is used to bind our checksum values to 8-bits.

2.4 Receive and Echo Characters

2.4.1 UART Initialization

With the checksum function completed, we now focus our efforts into implementing a basic transmission (TX) and reception (RX) configuration on the UNO32 microcontroller. Referencing the PIC32 Family Reference Manual (FRM), we configured the serial port with the following settings:

```
1     #define FPB (BOARD_GetPBClock())
2     #define UART_BAUDRATE 115200
3
4     U1MODE = 0; //reset and clear UART1
5     U1STA = 0;
6
7     // calculates baud rate to 115200
8     U1BRG = (unsigned int) (FPB / (16 * UART_BAUDRATE)) - 1;
9
10    U1MODEbits.ON = 1; //enables UART1
11    U1MODEbits.RXINV = 0;
12
13    IPC6bits.U1IP = 2; // Interrupt priority is 2
14
15    U1STAbits.URXISEL = 0;
16    U1STAbits.UTXINV = 0;
17    U1STAbits.URXEN = 1; // enable RX
18    U1STAbits.UTXEN = 1; // enable TX
19
20    IEC0bits.U1RXIE = 1; // enable RX interrupt
21    IFS0bits.U1RXIF = 0; // clear RX interrupt
22    IEC0bits.U1TXIE = 1; // enable TX interrupt
23    IFS0bits.U1TXIF = 0; // clear TX interrupt
24 }
```

This gives enables and clears **UART1**, configures the baud rate to 115200, and sets the UART to 8 data bits, no parity, and 1 stop bit (**8-N-1**). We also enable transmission and reception and their corresponding interrupts. To calculate the baud rate setting, we used the equation given in the FRM section on UART.

Equation 21-1: UART Baud Rate with BRGH = 0

$$Baud\ Rate = \frac{F_{PB}}{16 \cdot (UxBRG + 1)}$$

$$UxBRG = \frac{F_{PB}}{16 \cdot Baud\ Rate} - 1$$

Note: F_{PB} denotes the PBCLK frequency.

Figure 2: UART 16-bit Baud Rate Generator(U1BRG) equation.

We solve for U1BRG with our F_{PB} = 40MHz (our PB clock rate). This means that the serial port is capable of transferring a maximum of 115200 bits per second. Because we plan on using our TX and RX registers to send and receive packets, their interrupts are also enable and cleared in our initialization. This allows events to be triggered by the TX and RX interrupt flags, making our code atomic. To demonstrate the differences between a blocking transmission and an interrupt-driven transmission on our UNO32, we first implemented a blocking example where we printed "Hello World!" to a serial monitor using **U1TXREG** and our **PutChar()** function inside protocol.c. Below is the PutChar() implementation for reference.

```
1   int PutChar(char ch)
2 {
3     if (transmitBuffer == NULL)
4         return ERROR;
5
6     if(!isFull(transmitBuffer))
7     {
8         isWritingToBuffer = TRUE;
9         write(transmitBuffer, ch);
10        isWritingToBuffer = FALSE;
11
12        if(U1STAbits.TRMT)
13            IFS0bits.U1TXIF = 1;
14    }
15
16    return SUCCESS;
17 }
18
```

We can see that there is a flag used to lock the program in until writing has been completed. Then, we refactored the same program to use the **UART1 interrupt** instead. Due to the simplicity of these two programs, they were omitted from the final version of our protocol file. The complete implementation of the UART1 interrupt is shown in Appendix A.

After familiarizing ourselves with how the UART1 interrupt and U1TXREG worked in conjunction to sending char data over serial, a circular buffer struct was created to hold the characters while transmitting and receiving.

2.4.2 Circular Buffer for TX and RX

Using the pseudo-code developed for a simple circular buffer, a circular buffer struct was created in our protocol file:

```
typedef struct circBuffer {
    unsigned char buffer[MAXPAYLOADLENGTH];
    int head;
    int tail;
    unsigned int length;
} circBuffer;
```

This circular buffer struct included read and write functions which get called in our UART1 interrupt. To prevent collisions from occurring while either reading or writing, a **mutex** was used to lock the program until a read/write process finishes. If a collision does occur, we reset the collision flag and "kick" the interrupt flag to break the microcontroller from its idle state. This is to ensure that the next characters can be grabbed by the interrupt.

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

    if (IFS0bits.U1RXIF)
    {
        IFS0bits.U1RXIF = 0;
        if(getBuffLength(receiverBuffer) != BUFFER_LENGTH)
        {
            if(!isReceivingFromBuffer) {

                write(receiverBuffer, U1RXREG);
            }
            else {
                receiveCollision = TRUE;
            }
        }
    }
    if (IFS0bits.U1TXIF)
    {
        IFS0bits.U1TXIF = 0;
        if(!isEmpty(transmitBuffer))
        {
            if (!isWritingToBuffer)
            {
                U1TXREG = read(transmitBuffer);
```

```
25                }
26                else {
27                    transmitCollision = TRUE;
28                }
29            }
30        }
31    }
```

As shown in the coding listing above, if either the RX or TX interrupt flags are triggered, the interrupt resets its interrupt flag and runs through each conditional. Depending on what caused the interrupt, the microcontroller will either be reading from the **U1RXREG** or writing to the **U1TXREG**. For an in-depth look at the UART code implemented, please consult Appendix A.

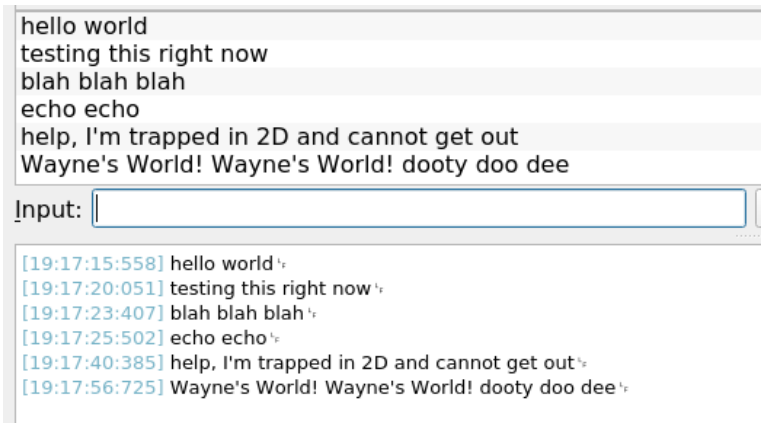

Figure 3: Echo program used to test our UART serial communication.

To test out our TX and RX buffer, we used a serial monitor to send messages to the microcontroller and have it echo these messages back to the monitor as seen in Fig. 7. With the basic raw serial communication example working with the UNO32, a full-fledged protocol library can now be built.

3 Protocol Library

The protocol library is extension of our previous code with the added complexity of our protocol packets and their message ids. It handles the encoding and decoding of messages for our UNO32 and also runs commands based on these messages. A new RX circular buffer and packet struct was created to accommodate for our protocol requirements for Lab 1. With the protocol library, we are expected to hold up to 5

packets in one RX buffer to parse and so a modular OOP approach was chosen to keep everything organized.

3.1 Protocal Packet Struct

Before building our new protocol functions specified in the **protocol.h** file, a packet struct was created to store all necessary data needed when decoding a transmitted packet.

```
  typedef struct packet{
    unsigned char buffer[MAXPAYLOADLENGTH];
    MessageIDS_t ID;
    unsigned int length;
} packet;

```

This struct contains a **char array** which holds the payload, a **MessageIDS_t** enum which stores the message ID of our packet, and the **length** of the payload received. Storing the message ID separately was a design choice that made other functions significantly easier to implement. While there have been functions implemented to access the payload struct directly in protocol.c, these functions were meant to troubleshoot the packet struct during development and are not used directly. A modified RX circular buffer was created to handle all changes done to the packet struct in our program.

3.2 New RX Circular Buffer

Due to the complexity of our protocol program, a new circular buffer struct was created to handle multiple payloads. The payload char array has been replaced by an array of packet structs. An error flag was also added to indicate whenever a protocol process failed to finish.

```
typedef struct circReceiverBuffer {
    struct packet payloads[PACKETBUFFERSIZE];
    int head;
    int tail;
    int error;
    unsigned int length;
} circReceiverBuffer;

```

Parts of the original buffer functions were also modified and moved to the protocol functions given in protocol.h, like checking if the queue was empty or full. With a new

RX buffer and packet struct implemented, we then focused our attention to designing packet creation.

3.3 Protocol_SendMessage()

At the core, all packets are assembled inside the **SendMessage()** function of the protocol program. Based on the steps provided in the lab manual, a sequence of chars are sent to the TX register for transmission using **PutChar()**. Below is the code implementation:

```
  int Protocol_SendMessage(unsigned char len, unsigned char ID,
                           void *Payload)
{
    int i;
    char ch;
    char *payload = (char*)Payload;
    checksum = 0;

    // Enqueue the HEAD
    PutChar(HEAD);
    delay_ms(1);

    // Calculate correct length of payload argument + 1 (message ID)
    PutChar((unsigned char)((int)len + 1));
    delay_ms(1);

    // Initialize the checksum with the ID
    checksum = Protocol_CalcIterativeChecksum(ID, checksum);

    // Enqueue the ID
    PutChar(ID);
    delay_ms(1);


    // Enqueue the data one byte at a time(and update your checksum)
    for (i=0; i < (unsigned int)len; i++)
    {
        PutChar((payload[i]));
        delay_ms(1);
        ch = (payload[i]);
        checksum = Protocol_CalcIterativeChecksum(ch, checksum);
    }

    // Enqueue the TAIL
    PutChar(TAIL);
    delay_ms(1);

```

```
38      // Enqueue the CHECKSUM
39      PutChar(checksum);
40      delay_ms(1);
41
42      // Enqueue \r\n
43      PutChar('\r');
44      delay_ms(1);
45      PutChar('\n');
46      delay_ms(1);
47
48      return SUCCESS;
49
50  }
51
```

We can see the steps for sending a packet clearly based on this code block.

1. Enqueue the **HEAD**
2. Calculate the **LENGTH** and enqueue it.
3. Enqueue the **ID** and calculate its checksum
4. Iterate through the **PAYLOAD** array, enqueuing each char and calculating their cumulative checksum
5. Enqueue the **TAIL**
6. Enqueue the cumulative **CHECKSUM**
7. Enqueue the **END** (\r\n)

A milisecond delay was added to each enqueue. This was due to printing errors from the serial monitor where it appeared that parts of the function were being sent too quickly. For this delay function, the system clock was used for accuracy.

```
1  void delay_ms(unsigned int ms)
2  {
3      ms *= FPB / 40000;
4      _CP0_SET_COUNT(0);
5      while (ms > _CP0_GET_COUNT());
6
7  }
8
```

After a sending a couple messages using a test harnesses, the packets received had the correct hex values expected. This working function will now be the basis in packet transmissions. To assist our protocol development, a wrapper for debug messages was added to send messages to the Lab interface.

3.4 Protocol_SendDebugMessage()

Sending protocol message to the lab interface was a straightforward process and simply utilized the send message function already implemented in the previous section:

```
 int Protocol_SendDebugMessage(char *Message)
{
    if (Protocol_IsError())
        return ERROR;

    unsigned char message_length = strlen(Message);
    Protocol_SendMessage(message_length, ID_DEBUG, Message);
    return SUCCESS;
}

```

While we now have a working program to encode messages for transmission, we still need a function to decode and filter packets being received by the microcontroller.

3.5 Protocol RX State Machine

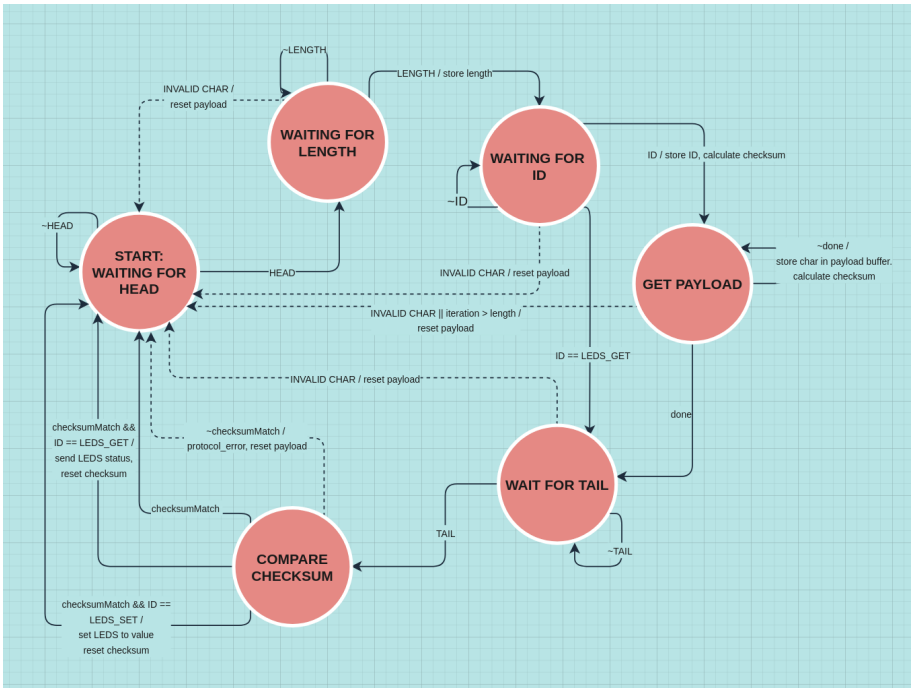

Figure 4: RX State Machine. The dotted lines indicate when a state returns to WAITING FOR HEAD due to a protocol error.

The process of the decoding packets inside the UNO32 is more complex and involved compared to encoding messages. A RX state machine was created to handle the multiple stages required to correctly reading a packet and ensuring its integrity. Due to the density of the code, please refer to the Appendix for specifics. Because the RX state machine is inside our interrupt, all code was made to be atomic and a switch statement was used to store the current state of the RX buffer. For simplicity, a State enum object was initialized as a static global variable and stored the state of the RX buffer externally. The iteration variable for our payload char array was also stored externally to prevent long, illegible code due to variable nesting in both the RX circular buffer and packet structs. After coding up the initial state machine, we tested our function by turning on LEDS on the I/O shield and sending over their state to the lab interface.

3.6 LEDS_SET and LEDS_GET

As stated in the lab manual, **ID_LEDS_SET()** and **ID_LEDS_GET** are handled internally by the state machine and are only stored inside the payload temporarily to check the packet's validity. These message IDs are handled inside the last state **COMPARE CHECKSUM** with conditionals and returns the state back to **WAITING FOR HEAD**.

```
1             if (charIn == checksum)
2           {
3               checksum = 0;
4               if (receiverBuffer->payloads[receiverBuffer->tail].ID
    == ID_LEDS_SET)
5               {
6                   LEDS_SET(receiverBuffer->payloads[receiverBuffer->
    tail].buffer[0]);
7                   checksum = 0;
8                   iteration = 0;
9                   break;
10              }
11
12              if (receiverBuffer->payloads[receiverBuffer->tail].ID
    == ID_LEDS_GET)
13            {
14                sprintf(message, "%c", PORTE);
15                Protocol_SendMessage(1,ID_LEDS_STATE, message);
16                checksum = 0;
17                iteration = 0;
18                break;
19            }
20
```

From the code block above, we can see the specifics of this implementation. If the received checksum matches our calculated checksum, we then check to see if our packet ID is ID_LEDS_SET or ID_LEDS_GET. If our ID is ID_LEDS_SET, we then set our LEDS based on the value inside our payload array. If our ID is ID_LEDS_GET, we send a message to the lab interface which contains the state of our LEDs on the I/O shield. Testing this code, we get the following results:

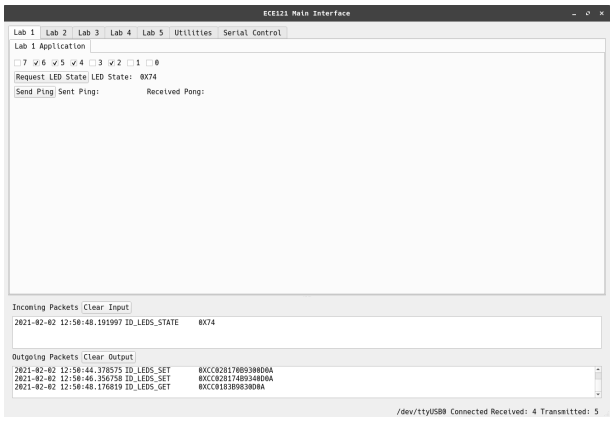

Figure 5: The LED set and get commands displayed on the lab interface.

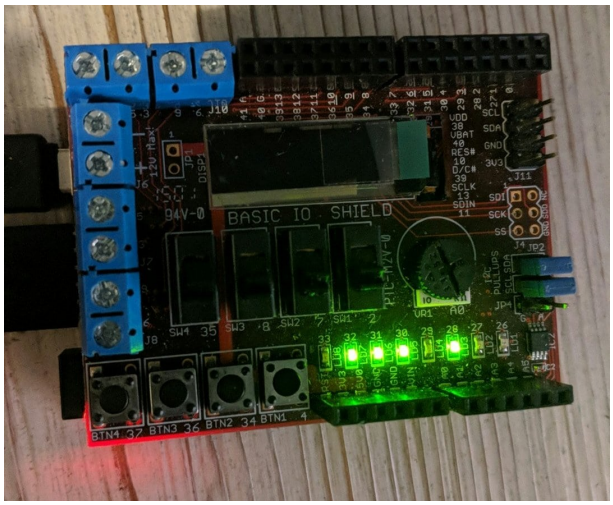

Figure 6: The corresponding LEDs lit on the I/O Shield

4 Ping Pong

In final section of our lab, we test our protocol using a ping pong program. Whenever a ping message is sent from the lab interface, a pong message will be returned by the microcontroller. This pong message contains the original payload sent with it's value shifted to the right. In order to correctly do this routine, we need to first convert the original payload to little endian for processing and then back to big endian for transmission. This is done by the symmetric function **Protocol_IntEndednessConversion()**.

4.1 Big Endian and Little Endian

Conversion between big and little endian is necessary for properly communicating between the serial monitor and microcontroller. This is to match the conventions used in most network streams and hardware architectures. In order to perform this symmetric swap, the following code was implemented for both an int and a short.

```
1  /**
2   * @Function char Protocol_ShortEndednessConversion(unsigned short
       inVariable)
3   * @param inVariable, short to convert endedness
4   * @return converted short
5   * @brief Converts endedness of a short. This is a bi-directional
       operation so only one function is needed
6   * @author mdunne */
7  unsigned short Protocol_ShortEndednessConversion(unsigned short
       inVariable)
8  {
9      return (inVariable >> 8) | (inVariable << 8);
10 }
11
12 /**
13  * @Function char Protocol_IntEndednessConversion(unsigned int
       inVariable)
14  * @param inVariable, int to convert endedness
15  * @return converted short
16  * @brief Converts endedness of a int. This is a bi-directional
       operation so only one function is needed
17  * @author mdunne */
18 unsigned int Protocol_IntEndednessConversion(unsigned int inVariable)
19 {
20     unsigned int b0,b1,b2,b3;
21
22     b0 = (inVariable & 0x000000ff) << 24;
23     b1 = (inVariable & 0x0000ff00) << 8;
24     b2 = (inVariable & 0x00ff0000) >> 8;
```

```
25      b3 = (inVariable & 0xff000000) >> 24;
26
27      return (b0 | b1 | b2 | b3);
28  }
29
```

We can see the symmetry in these conversion functions with the byte ordering swapping between the most significant byte and last significant byte in our unsigned int. Testing both of these functions in our harness, we get the following output for both the short and int converters.

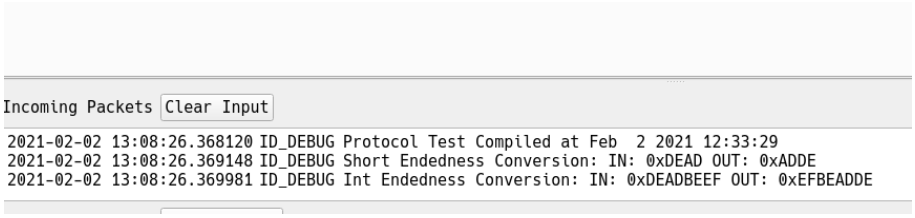

Figure 7: Testing our Endian conversion functions inside the lab interface.

4.2 Ping Pong

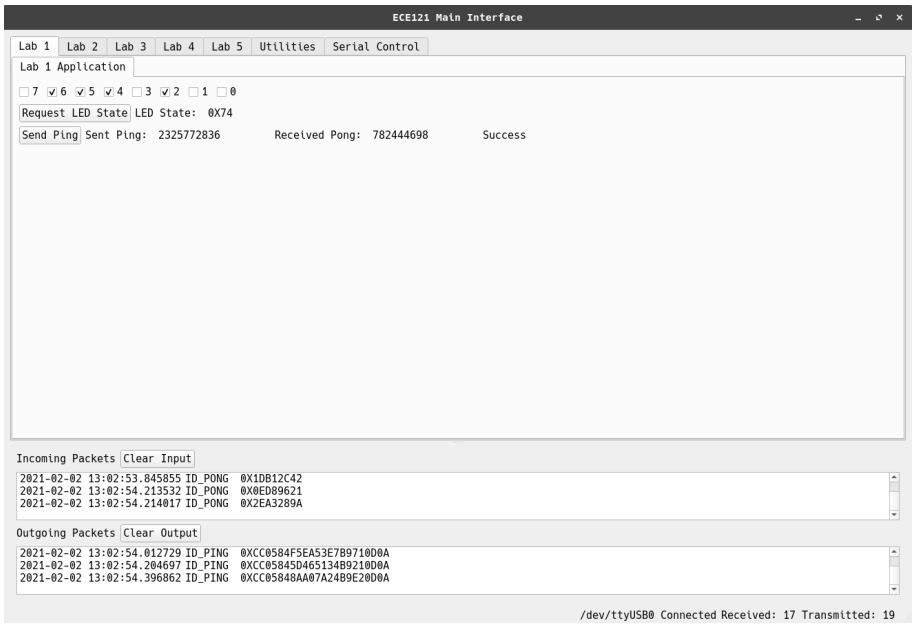

Figure 8: Our ping pong program in action on the lab interface.

Using the test harness provided on piazza, we tested combined all of our functions of protocol.c to respond to ping messages sent to the UNO32 from the lab interface.

```
    unsigned int pingValue = 0xfff;
    checksum = 0;
    while (1) {
        if (Protocol_IsMessageAvailable()) {
            if (Protocol_ReadNextID() == ID_PING) {
                // send pong in response here
                Protocol_GetPayload(&pingValue);
                pingValue = Protocol_IntEndednessConversion(pingValue
    );
                pingValue >>= 1;
                pingValue = Protocol_IntEndednessConversion(pingValue
    );
                Protocol_SendMessage(4, ID_PONG, &pingValue);
            }
        }
    }
    while (1);

```

Based on this code snippet, we can see that if there is a new message available in our RX buffer, we check to see if the message ID is an **ID_PING** message. If so, we grab the payload of our received packet, convert our value to little endian, perform our bit shift, convert it back to big endian, and return a pong message. This section proved to be the most challenging part of this lab; in order to perform the bitwise operation correctly, the char array needed to be cast to an unsigned int. Because C cannot actually perform a pass by variable and the input of our **GetPayload** function uses a void pointer, a specific workaround was required to properly convert and modify the value inside pingValue:

```
int Protocol_GetPayload(void* Payload)
{
    if (Protocol_IsError())
    {
        return ERROR;
    }
    *((unsigned int*) Payload) = *((unsigned int*)receiverBuffer->
    payloads[receiverBuffer->head].buffer);
    receiverBuffer->head = (receiverBuffer->head + 1) %
    receiverBuffer->length;

    return SUCCESS;
}

```

After figuring out the correct syntax, the rest of the implementation was trivial and

we can see the results of our testing in Fig. 8.

5 Conclusion

Lab 1 demonstrated the complexity and challenging elements of designing a serial communication protocol for a microcontroller. One of the major hurdles in reproducing sections of this lab is the confusing organization of each part in the lab 1 manual. Often a section would densely describe a concept without context to how it is used in our project; in some cases, the concept would be introduced too late in the lab manual where time could have been saved in development if given earlier. An example of this is having the protocol test harness section near the bottom of the lab instead of the top or a pre-lab being at the end of a section when it should be in the forefront. A suggestion for future revisions is either explicitly saying in the introduction that this lab manual is not meant to be read chronologically or to reformat the sections where in-depth concepts are in an appendix with the logical steps in the main sections (similar to Lab 0). That being said, it is hard to create a lab manual that can balance the wealth of information needed to understand the technical process and the author made great attempt in doing so.

Another challenge when implementing this lab was how to effectively organize and work with the growing complexity of the project and troubleshooting inside the interrupt and state machine. Despite the efficiency and lowered complexity of having all the structs and functions housed inside protocol.c, code folding did little to minimize confusion when debugging; even with a incremental, modular approach and extensive commenting, the messiness of the code structure sunk time. In future implementations, breaking the code into separate libraries might be a necessary compromise to organize the code into something more human-readable. Due to the atomic nature of the interrupt, it was significantly more challenging to test conditionals inside; because printing message inside would break the program, testing needed to be done using LEDs inside the state machine. This was not explicitly hinted at inside the lab manual and was a hard lesson learned while troubleshooting. In debugging a debugging tool, workarounds and creative tests were needed to writing a function program. Overall, this program required an immense amount of time to complete and was very educational in the process.

Appendix

A. Source Code

```
/*
 * File:   Protocol.c
 * Author: Mel Ho
 *
 * Created on January 15, 2021, 1:12 AM
 */
#include <xc.h>
#include <BOARD.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/attribs.h>
#include "Protocol.h"
#include "MessageIDs.h"
#define TRUE 1
#define FALSE 0
#define EMPTY 0
#define NULL_VAL_ERROR 89
#define BUFF_EMPTY 100
#define BUFF_FULL 110
static char message[MAXPAYLOADLENGTH];
#define NOP 1
#define FPB (BOARD_GetPBClock())
#define BUFFER_LENGTH 135
#define ms_SCALE (FPB/40000);
enum PayloadStateType {
    waitingForHead,
    waitingForLength,
    waitingForMessageID,
    receivingPayload,
    waitingForTail,
    compareChecksum
};
/**********************************
 *       TEST HARNESSES          *
 **********************************/
//#define CIRCBUFF_TX_TEST 1
//#define  CHECKSUM_TEST 1
//#define CIRCBUFF_RX_ECHO_TEST 1
//#define SEND_MESSAGE 1
//#define TEST_ENDIANS 1
//#define TEST_LEDS 1
#define TEST_HARNESS
/**********************************
 *      PRIVATE DATATYPES        *
 **********************************/
/* Packet
 * Struct that contains the message ID,
 * Payload data, and length of the message
 */
typedef struct packet{
    unsigned char buffer[MAXPAYLOADLENGTH];
    MessageIDS_t ID;
    unsigned int length;
} packet;
// Pointer to packet struct
typedef struct packet* packetPointer;
/* circBuffer
 * Struct that handles a standard circular buffer
 * Used to send messages (TX buffer) or
 * receive raw data (receiver buffer)
 */
typedef struct circBuffer {
    unsigned char buffer[MAXPAYLOADLENGTH];
    int head;
    int tail;
    unsigned int length;
} circBuffer;
// Pointer to circBuffer struct
typedef struct circBuffer* cbPointer;
/* circReceiverBuffer
 * Specialized circular buffer for receiving packets
 * from the python console. Probably the ugliest, most
 * redundant code I've ever written for a use case.
```

```
 */
typedef struct circReceiverBuffer {
    struct packet payloads[PACKETBUFFERSIZE];
    int head;
    int tail;
    int error;
    unsigned int length;
} circReceiverBuffer;
// Pointer to circReceiverBuffer
typedef struct circReceiverBuffer* cbRPointer;
/*********************************************
 *  PACKET PRIVATE FUNCTIONS PROTOTYPES  *
 *********************************************/
packet newPacket(void);
void resetPacket(packetPointer pd);
int isInvalidCharacter(unsigned char charIn);
/************************************************
 * CIRC BUFFER PRIVATE FUNCTIONS PROTOTYPES  *
 ************************************************/
void newCircReceiverBuffer(cbRPointer circRBuff);
void newCircBuffer(cbPointer circBuff);
unsigned int getBuffLength(cbPointer circBuff);
int getHead(cbPointer circBuff);
int getTail(cbPointer circBuff);
unsigned char read(cbPointer circBuff);
void write(cbPointer circBuff, unsigned char data);
/**********************************
 *    NOP FUNCTION PROTOTYPES    *
 **********************************/
void delay_ms();
/**********************************
 *      PRIVATE VARIABLES        *
 **********************************/
enum PayloadStateType State = waitingForHead;
struct packet onePayload;
packetPointer payloadBuff;
struct circReceiverBuffer rxBuff;
cbRPointer receiverBuffer;
struct circBuffer txBuff;
cbPointer transmitBuffer;
static int iteration = 0;
static unsigned char checksum = 0;
static unsigned char protocol_error = FALSE;
static uint8_t isWritingToBuffer = FALSE;
static uint8_t transmitCollision = FALSE;
static uint8_t isReceivingFromBuffer = FALSE;
static uint8_t receiveCollision = FALSE;
/**********************************
 *          NOP DELAY            *
 **********************************/
void delay_ms(unsigned int ms)
{
    ms *= FPB / 40000;
    _CP0_SET_COUNT(0);
    while (ms > _CP0_GET_COUNT());
}
/**********************************
 *      PAYLOAD FUNCTIONS        *
 **********************************/
packet newPacket()
{
    int i;
    packet pd;
    for (i = 0; i < MAXPAYLOADLENGTH; i++)
       pd.buffer[i] = 0;
    pd.length = MAXPAYLOADLENGTH;
    pd.ID = ID_INVALID;
    return pd;
}
int isInvalidCharacter(unsigned char charIn)
{
    return charIn == HEAD || charIn == TAIL;
}
void newCircReceiverBuffer(cbRPointer circRBuff)
{
    circRBuff->head = 0;
    circRBuff->tail = 0;
    circRBuff->length = PACKETBUFFERSIZE;
}
void newCircBuffer(cbPointer circBuff)
{
    int i;
    for (i = 0; i < MAXPAYLOADLENGTH; i++)
```

```
        circBuff->buffer[i] = 0;
    circBuff->head = 0;
    circBuff->tail = 0;
    circBuff->length = MAXPAYLOADLENGTH;
}
unsigned int isEmpty(cbPointer circBuff)
{
    return (circBuff->head == circBuff->tail);
}
unsigned int isFull(cbPointer circBuff)
{
    return (circBuff->head == ((circBuff->tail + 1) % circBuff->length));
}
unsigned int getBuffLength(cbPointer circBuff)
{
    if (circBuff != NULL)
    {
        int tail = circBuff->tail;
        if (circBuff->tail < circBuff->head)
            tail = (circBuff->length + circBuff->tail);
        return (tail - circBuff->head);
    }
    else
        return NULL_VAL_ERROR;
}
unsigned int getReceiverBuffLength(cbRPointer circRBuff)
{
    if (circRBuff != NULL)
    {
        int tail = circRBuff->tail;
        if (circRBuff->tail < circRBuff->head)
            tail = (circRBuff->length + circRBuff->tail);
        return (tail - circRBuff->head);
    }
    else
        return NULL_VAL_ERROR;
}
unsigned char read(cbPointer circBuff)
{
    unsigned char payload;
    payload = circBuff->buffer[circBuff->head];
    circBuff->head = (circBuff->head + 1) % circBuff->length;
    return payload;
}
void write(cbPointer circBuff, unsigned char data)
{
    circBuff->buffer[circBuff->tail] = data;
    circBuff->tail = (circBuff->tail + 1) % circBuff->length;
}
int PutChar(char ch)
{
    if (transmitBuffer == NULL)
        return ERROR;
    if(!isFull(transmitBuffer))
    {
        isWritingToBuffer = TRUE;
        write(transmitBuffer, ch);
        isWritingToBuffer = FALSE;
        if(U1STAbits.TRMT)
            IFS0bits.U1TXIF = 1;
    }
    return SUCCESS;
}
/*********************************
 *      PROTOCOL FUNCTIONS       *
 *********************************/
int Protocol_Init()
{
    LEDS_INIT();
    transmitBuffer = (struct circBuffer*) &txBuff;
    newCircBuffer(transmitBuffer);
    receiverBuffer = (struct circReceiverBuffer*) &rxBuff;
    newCircReceiverBuffer(receiverBuffer);
    U1MODE = 0;
    U1STA = 0;
    U1BRG = (unsigned int) (FPB / (16 * UART_BAUDRATE)) - 1;
    U1MODEbits.ON = 1;
    U1MODEbits.RXINV = 0;
    IPC6bits.U1IP = 2;
    U1STAbits.URXISEL = 0;
    U1STAbits.UTXINV = 0;
    U1STAbits.URXEN = 1;
    U1STAbits.UTXEN = 1;
```

```
    IEC0bits.U1RXIE = 1;
    IFS0bits.U1RXIF = 0;
    IEC0bits.U1TXIE = 1;
    IFS0bits.U1TXIF = 0;
    return SUCCESS;
}
/**
 * @Function int Protocol_SendMessage(unsigned char len, void *Payload)
 * @param len, length of full <b>Payload</b> variable
 * @param Payload, pointer to data, will be copied in during the function
 * @return SUCCESS or ERROR
 * @brief
 * @author mdunne */
int Protocol_SendMessage(unsigned char len, unsigned char ID, void *Payload)
{
    int i;
    char ch;
    char *payload = (char*)Payload;
    checksum = 0;
//    PutChar(checksum);
    // Enqueue the HEAD
    PutChar(HEAD);
    delay_ms(1);
    // Calculate correct length of payload argument + 1 (for message ID)
    PutChar((unsigned char)((int)len + 1));
    delay_ms(1);
    // Initialize the checksum with the ID
    checksum = Protocol_CalcIterativeChecksum(ID, checksum);
//    PutChar(checksum);
    // Enqueue the ID
    PutChar(ID);
    delay_ms(1);
    // Enqueue the data one byte at a time(and update your checksum)
    for (i=0; i < (unsigned int)len; i++)
    {
        PutChar((payload[i]));
        delay_ms(1);
        ch = (payload[i]);
        checksum = Protocol_CalcIterativeChecksum(ch, checksum);
//        PutChar(checksum);
    }
    // Enqueue the TAIL
    PutChar(TAIL);
    delay_ms(1);
    // Enqueue the CHECKSUM
    PutChar(checksum);
    delay_ms(1);
    // Enqueue \r\n
    PutChar('\r');
    delay_ms(1);
    PutChar('\n');
    delay_ms(1);
//    checksum = 0;
    return SUCCESS;
}
/**
 * @Function int Protocol_SendDebugMessage(char *Message)
 * @param Message, Proper C string to send out
 * @return SUCCESS or ERROR
 * @brief Takes in a proper C-formatted string and sends it out using ID_DEBUG
 * @warning this takes an array, do <b>NOT</b> call sprintf as an argument.
 * @author mdunne */
int Protocol_SendDebugMessage(char *Message)
{
    if (Protocol_IsError())
        return ERROR;
    unsigned char message_length = strlen(Message);
    Protocol_SendMessage(message_length, ID_DEBUG, Message);
    return SUCCESS;
}
/**
 * @Function unsigned char Protocol_ReadNextID(void)
 * @param None
 * @return Reads ID of next Packet
 * @brief Returns ID_INVALID if no packets are available
 * @author mdunne */
unsigned char Protocol_ReadNextID(void)
{
    if (!Protocol_IsMessageAvailable())
        return (unsigned char) ID_INVALID;
    return (unsigned char)(receiverBuffer->payloads[receiverBuffer->head].ID);
}
/**
```

```
 * @Function int Protocol_GetPayload(void* payload)
 * @param payload, Memory location to put payload
 * @return SUCCESS or ERROR
 * @brief
 * @author mdunne */
int Protocol_GetPayload(void* Payload)
{
    if (Protocol_IsError())
    {
        return ERROR;
    }
    *((unsigned int*) Payload) = *((unsigned int*)receiverBuffer->payloads[receiverBuffer->head].buffer);
    receiverBuffer->head = (receiverBuffer->head + 1) % receiverBuffer->length;
    return SUCCESS;
}
/**
 * @Function char Protocol_IsMessageAvailable(void)
 * @param None
 * @return TRUE if Queue is not Empty
 * @brief
 * @author mdunne */
char Protocol_IsMessageAvailable(void)
{
    return ((receiverBuffer->head != receiverBuffer->tail));
}
/**
 * @Function char Protocol_IsQueueFull(void)
 * @param None
 * @return TRUE is QUEUE is Full
 * @brief
 * @author mdunne */
char Protocol_IsQueueFull(void)
{
    return (receiverBuffer->head == ((receiverBuffer->tail + 1) % receiverBuffer->length));
}
/**
 * @Function char Protocol_IsError(void)
 * @param None
 * @return TRUE if error
 * @brief Returns if error has occurred in processing, clears on read
 * @author mdunne */
char Protocol_IsError(void)
{
    if (protocol_error == TRUE)
    {
        protocol_error = FALSE;
        return TRUE;
    }
    return FALSE;
}
/**
 * @Function char Protocol_ShortEndednessConversion(unsigned short inVariable)
 * @param inVariable, short to convert endedness
 * @return converted short
 * @brief Converts endedness of a short. This is a bi-directional operation so only one function is needed
 * @author mdunne */
unsigned short Protocol_ShortEndednessConversion(unsigned short inVariable)
{
    return (inVariable >> 8) | (inVariable << 8);
}
/**
 * @Function char Protocol_IntEndednessConversion(unsigned int inVariable)
 * @param inVariable, int to convert endedness
 * @return converted short
 * @brief Converts endedness of a int. This is a bi-directional operation so only one function is needed
 * @author mdunne */
unsigned int Protocol_IntEndednessConversion(unsigned int inVariable)
{
    unsigned int b0,b1,b2,b3;
    b0 = (inVariable & 0x000000ff) << 24;
    b1 = (inVariable & 0x0000ff00) << 8;
    b2 = (inVariable & 0x00ff0000) >> 8;
    b3 = (inVariable & 0xff000000) >> 24;
    return (b0 | b1 | b2 | b3);
}
/*******************************************************************************
 * PRIVATE FUNCTIONS
 * generally these functions would not be exposed but due to the learning nature of the class they
 * are to give you a theory of how to organize the code internal to the module
 ******************************************************************************/
/**
 * @Function char Protocol_CalcIterativeChecksum(unsigned char charIn, unsigned char curChecksum)
 * @param charIn, new char to add to the checksum
```

```
 * @param curChecksum, current checksum, most likely the last return of this function, can use 0 to reset
 * @return the new checksum value
 * @brief Returns the BSD checksum of the char stream given the curChecksum and the new char
 * @author mdunne */
unsigned char Protocol_CalcIterativeChecksum(unsigned char charIn, unsigned char curChecksum)
{
    curChecksum = (curChecksum >> 1) + ((curChecksum & 1) << 7);
    curChecksum += charIn;
    curChecksum &= 0xff;
    return curChecksum;
}
/**
 * @Function void Protocol_runReceiveStateMachine(unsigned char charIn)
 * @param charIn, next character to process
 * @return None
 * @brief Runs the protocol state machine for receiving characters, it should be called from
 * within the interrupt and process the current character
 * @author mdunne */
void Protocol_RunReceiveStateMachine(unsigned char charIn)
{
  switch(State)
  {
      case waitingForHead:
          iteration = 0;
          if (charIn == HEAD && !Protocol_IsQueueFull())
          {
              State = waitingForLength;
          }
          break;
      case waitingForLength:
          if (charIn > MAXPAYLOADLENGTH || charIn <= 0)
          {
              protocol_error = TRUE;
              State = waitingForHead;
              return;
          }
          receiverBuffer->payloads[receiverBuffer->tail].length = (unsigned int) charIn;
          State = waitingForMessageID;
          break;
      case waitingForMessageID:
          if(isInvalidCharacter(charIn))
          {
              protocol_error = TRUE;
              State = waitingForHead;
              return;
          }
          receiverBuffer->payloads[receiverBuffer->tail].ID = (int)charIn;
          checksum = Protocol_CalcIterativeChecksum(charIn, checksum);
          if (receiverBuffer->payloads[receiverBuffer->tail].ID == ID_LEDS_GET)
          {
              State = waitingForTail;
              return;
          }
          State = receivingPayload;
          break;
      case receivingPayload:
          if (iteration == receiverBuffer->payloads[receiverBuffer->tail].length)
          {
              protocol_error = TRUE;
              State = waitingForHead;
              return;
          }
          if (isInvalidCharacter(charIn))
          {
              protocol_error = TRUE;
              State = waitingForHead;
              return;
          }
          receiverBuffer->payloads[receiverBuffer->tail].buffer[iteration] = charIn;
          checksum = Protocol_CalcIterativeChecksum(charIn,checksum);
          if (iteration ==  receiverBuffer->payloads[receiverBuffer->tail].length - 2)
          {
              State = waitingForTail;
              return;
          }
          iteration++;
          break;
      case waitingForTail:
          if (charIn != TAIL)
          {
              protocol_error = TRUE;
              State = waitingForHead;
              return;
```

```
            }
            State = compareChecksum;
            break;
        case compareChecksum:
            State = waitingForHead;
            if (charIn == checksum)
            {
                checksum = 0;
                if (receiverBuffer->payloads[receiverBuffer->tail].ID == ID_LEDS_SET)
                {
                    LEDS_SET(receiverBuffer->payloads[receiverBuffer->tail].buffer[0]);
                    checksum = 0;
                    iteration = 0;
                    break;
                }
                if (receiverBuffer->payloads[receiverBuffer->tail].ID == ID_LEDS_GET)
              {
                  sprintf(message, "%c", PORTE);
                  Protocol_SendMessage(1,ID_LEDS_STATE, message);
                  checksum = 0;
                  iteration = 0;
                  break;
              }
              receiverBuffer->tail = (receiverBuffer->tail + 1) % receiverBuffer->length;
              iteration = 0;
              break;
            }
            protocol_error = TRUE;
            checksum = 0;
            iteration = 0;
            break;
        default:
            break;
    }
}
/**********************************
 *  ISR UART1 INTERRUPT HANDLER  *
 **********************************/
void __ISR ( _UART1_VECTOR ) IntUart1Handler(void) {
    if (IFS0bits.U1RXIF)
    {
        IFS0bits.U1RXIF = 0;
        if (!isReceivingFromBuffer)
        {
          Protocol_RunReceiveStateMachine(U1RXREG);
        }
        else
            receiveCollision = TRUE;
    }
    if (IFS0bits.U1TXIF)
    {
        IFS0bits.U1TXIF = 0;
        if(!isEmpty(transmitBuffer))
        {
            if (!isWritingToBuffer)
            {
                U1TXREG = read(transmitBuffer);
            }
            else {
                transmitCollision = TRUE;
            }
        }
    }
}
#ifdef CHECKSUM_TEST
#include "xc.h"
#include "BOARD.h"
#include "Protocol.h"
#include "stdio.h"
#include "string.h"
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    unsigned char checksum = (unsigned char) 0x0000;
    unsigned char payload1 = (unsigned char) 0x80;
    unsigned char payload2 = (unsigned char) 0x7f;
    unsigned char payload3 = (unsigned char) 0x35;
    checksum = Protocol_CalcIterativeChecksum(payload1, checksum);
    checksum = Protocol_CalcIterativeChecksum(payload2, checksum);
    checksum = Protocol_CalcIterativeChecksum(payload3, checksum);
    PutChar(checksum);
    return 0;
```

```
}
#endif
#ifdef CIRCBUFF_TX_TEST
#include "xc.h"
#include "BOARD.h"
#include <sys/attribs.h>
#include <GenericTypeDefs.h>
#include <string.h>
#include "stdio.h"
#define BUFFER_SIZE 50
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    char buffer[BUFFER_SIZE];
    int i;
    sprintf(buffer, "Testing a bunch of different strings. It all began when...");
    for (i = 0; i < strlen(buffer); i++)
        PutChar(buffer[i]);
    return 0;
}
#endif
#ifdef CIRCBUFF_RX_ECHO_TEST
#include "xc.h"
#include "BOARD.h"
#include <sys/attribs.h>
#include <GenericTypeDefs.h>
#include <string.h>
#include "stdio.h"
#define BUFFER_SIZE 50
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    while(TRUE)
        if(getBuffLength(transmitBuffer) <= BUFFER_LENGTH)
        if (getBuffLength(oldReceiverBuffer) != 0)
            PutChar(GetChar());
    return 0;
}
#endif
#ifdef SEND_MESSAGE
#include "xc.h"
#include "BOARD.h"
#include "Protocol.h"
#include "stdio.h"
#include "string.h"
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    payload->payloadArray[0] = (unsigned char) 74;
    payload->payloadArray[1] = (unsigned char) 65;
    payload->payloadArray[2] = (unsigned char) 73;
    unsigned char length = (unsigned char)strlen(payload->payloadArray);
    unsigned char ID = (unsigned char) 0x81;
    Protocol_SendMessage(length, ID, payload->payloadArray);
    return 0;
}
#endif
#ifdef TEST_ENDIANS
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    short shortTestValue = 0xDEAD;
    short shortResultValue;
    int intTestValue = 0xDEADBEEF;
    int intResultValue;
    sprintf(testMessage, "State Machine Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(testMessage);
    sprintf(testMessage, "hello, world!");
    Protocol_SendDebugMessage(testMessage);
//
    shortResultValue = Protocol_ShortEndednessConversion(shortTestValue);
    sprintf(testMessage, "Short Endedness Conversion: IN: 0x%X OUT: 0x%X", shortTestValue&0xFFFF, shortResultValue&0xFFFF);
    Protocol_SendDebugMessage(testMessage);
    intResultValue = Protocol_IntEndednessConversion(intTestValue);
    sprintf(testMessage, "Int Endedness Conversion: IN: 0x%X OUT: 0x%X", intTestValue, intResultValue);
    Protocol_SendDebugMessage(testMessage);
    return 0;
}
#endif
```

```
#ifdef TEST_LEDS
int main(void)
{
    BOARD_Init();
    Protocol_Init();
    short shortTestValue = 0xDEAD;
    short shortResultValue;
    int intTestValue = 0xDEADBEEF;
    int intResultValue;
//    sprintf(testMessage, "State Machine Test Compiled at %s %s", __DATE__, __TIME__);
//    Protocol_SendDebugMessage(testMessage);
//    sprintf(testMessage, "hello, world!");
//    Protocol_SendDebugMessage(testMessage);
////
//    shortResultValue = Protocol_ShortEndednessConversion(shortTestValue);
//    sprintf(testMessage, "Short Endedness Conversion: IN: 0x%X OUT: 0x%X", shortTestValue&0xFFFF, shortResultValue&0xFFFF);
//    Protocol_SendDebugMessage(testMessage);
//
//    intResultValue = Protocol_IntEndednessConversion(intTestValue);
//    sprintf(testMessage, "Int Endedness Conversion: IN: 0x%X OUT: 0x%X", intTestValue, intResultValue);
//    Protocol_SendDebugMessage(testMessage);
    while(1);
//    LEDS_SET(0x4F);
    return 0;
}
#endif
#ifdef TEST_HARNESS
    int main(void)
{
    BOARD_Init();
    Protocol_Init();
    char testMessage[MAXPAYLOADLENGTH];
    sprintf(testMessage, "Protocol Test Compiled at %s %s", __DATE__, __TIME__);
    Protocol_SendDebugMessage(testMessage);
    short shortTestValue = 0xDEAD;
    short shortResultValue;
    int intTestValue = 0xDEADBEEF;
    int intResultValue;
    shortResultValue = Protocol_ShortEndednessConversion(shortTestValue);
    sprintf(testMessage, "Short Endedness Conversion: IN: 0x%X OUT: 0x%X", shortTestValue&0xFFFF, shortResultValue&0xFFFF);
    Protocol_SendDebugMessage(testMessage);
    intResultValue = Protocol_IntEndednessConversion(intTestValue);
    sprintf(testMessage, "Int Endedness Conversion: IN: 0x%X OUT: 0x%X", intTestValue, intResultValue);
    Protocol_SendDebugMessage(testMessage);
    unsigned int pingValue = 0xfff;
    checksum = 0;
    while (1) {
        if (Protocol_IsMessageAvailable()) {
            if (Protocol_ReadNextID() == ID_PING) {
                // send pong in response here
                Protocol_GetPayload(&pingValue);
                pingValue = Protocol_IntEndednessConversion(pingValue);
                pingValue >>= 1;
                pingValue = Protocol_IntEndednessConversion(pingValue);
                Protocol_SendMessage(4, ID_PONG, &pingValue);
            }
        }
    }
    while (1);
    return 0;
}
#endif
```