Microcontroller System Design

ECE121-Lab1

Professor Peterson

James Huang

Collaborated with Aaryan Redkar, Peter Le, Justin

03/8/24

1 Introduction

1.1 Hardware Background

In order for us to use the UART on the PIC32, we have to first configure the UART registers and set the baud rate first. We have to consult the datasheet for the register configurations and enabling UART.

Register 21-1: UxMODE: UARTx Mode Register

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
23:16	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
	—	—	—	—	—	—	—	—
15:8	R/W-0	U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
	ON(1)	—	SIDL	IREN	RTSMD(2)	—	UEN<1:0>(2)	
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>		STSEL

(Figure 1: UxMODE register table from the datasheet.)

This table shows the UxMODE register, this one holds the majority of the setup and enabling bits for UART. For example, PDSEL <1:0>, the parity and data selection bits control the amount of bits and what type of parity is wanted. For our case, we want 8 bit data with no parity bit, which is 0b00. If we were to set these two bits to 0b01, we would be getting 8 bit data, but with even parity. Besides the UxMODE register, we also use the UxSTA register, which is the status and control register for UART. This register has its own table with following bits that have different functions, this can all be obtained from the manual. This one contains a lot of useful bits like UTXEN, which enables TX transmission, UTXBF, which reads the status of the transmit buffer, and it determines if it is full or not. In addition, there are several other useful registers that we will be using, like the interrupt register.

Now every UART has to have a set baud rate in order to communicate correctly with another device, both devices have to be set to exactly the same baud rate. This determines how fast data is flowing from one device to another. The UxBRG register sets the baud rate.

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: Baud rate formula from the datasheet.)

This formula shows how to set the desired baud rate to our UxBRG, in our case we want 115200. The PBCLK, or the peripheral bus clock, can be found in the file BOARD.c. peripheral bus clock is like a clock for a specific part of the subsystem. As different parts of the system might run on different clocks. In our case, our PBCLK is defined as the system clock (80MHz) divided by two, which is 40MHz.

Lastly we will be covering Endianness, or the way bytes are ordered in a memory system. In a big endian system, the most significant byte is stored at the lowest memory address and the least significant byte is stored at the biggest memory address. In a little endian system, the logic is flipped. When data is sent between two devices with different endianness, the way we process data needs to be considered. We will be doing byte ordering and conversion in this lab, as the PIC32 is little endian and the data network is big endian.

1.2 Software Background

Besides setting up UART, the rest of Lab 1 is software development. A big part of this lab is the circular buffer, an abstract data type, or ADT. A typical ADT, like linked lists and graphs, are often defined using structs and pointers in C.

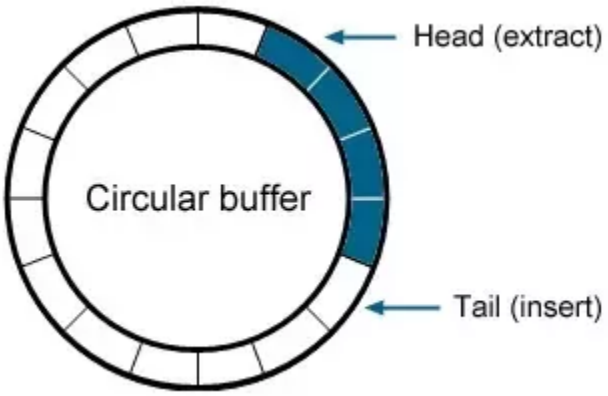
```
typedef struct BufferObj* Buffer;

typedef struct BufferObj{
    unsigned char buffer[BufferSize];
    int head;
    int tail;
    int full; //0 == not full 1 == full
    int err; //0 == no err 1 == err
}BufferObj;
```

(Figure 3: ADT structures.)

Structs and pointers go hand in hand in C code, in Figure 3 we have a Circular Buffer defined in the struct, with head and tail, and the buffer size. This circular buffer “datatype” is then named as

“Buffer”, which is a pointer to the struct.

Now a circular buffer works by inserting data into the tail and reading data from the head. How you define head and tail is up to you, you can choose to insert and read from either head or tail, but the two have to be well defined. In my case, I will be inserting data into the tail and reading from the head. As you are inserting and reading data, the head and tail will increment each time, as I add to the tail, the amount of data in the buffer increases, and as I read from the head, the amount of data decreases. It can be seen as putting data onto the buffer, and taking data off the buffer. Now there will be a point where the buffer is full, incoming data either cannot be written into the buffer or it will overwrite the old data. Constantly reading the data and taking it off the buffer ensures you don't have a full buffer.



(Figure 4: Circular Buffer)

Interrupt Service Routine or ISR is a program that pauses the processor from whatever it is doing to handle an interrupt. An interrupt is an event from either hardware or software that tells the processor it needs immediate action. Actions like button press can be a type of interrupt, where the hardware sends an interrupt signal. For us, we want to use the UART interrupt flags, which can be enabled and set up in the interrupt register for the PIC32. Interrupts can be set to desired priorities, which controls what interrupts to handle first over other ones, however we will not be using priorities in this lab.

In part three of the lab, we will be implementing a state machine to help us parse incoming characters. A state machine is useful for a system that requires clear conditions with certain outputs. The key concepts include states, which represent a certain condition that exist in the system, and only one state can be valid at a time. Then there are transitions between states, these are rules set on how states change and what needs to occur to make a change. Then there are the events that drive the change or drive a certain output. Lastly, in each state there will be specific actions that can occur either during a state or on state transitions. There are different types of state machines, but we will be focusing on finite state machines, which is quite common. Often a switch case structure is used for state machines in C code.

2 Lab0

2.1 Lab Part 1 Introduction

In the first part of the lab, we want to enable UART correctly, and start using it at a basic level. By using Coolterm, we want to be able to type something on our computers and have it sent out to the microcontroller, and have it echo back to our computers. It will look like we are typing on Coolterm, but each character is going through the microcontroller's UART system.

2.2 Lab Part 1 Initial Approach

First we have to consult the datasheet for the PIC32, as seen earlier in the introduction, we can see control registers and its corresponding control bits. We want to find the control register and the bits that allow us to enable and set up our UART to desired configuration. We will ignore the ISR and the circular buffer at this stage of development. By looking at the datasheet we are able to find crucial information like the baud rate register, the tx/rx port, the status register, the control/mode register.

2.3 Lab Part 1 Implementation

First we will want to clear the UART settings to default, this will require setting ALL of the bits in the control register to 0. Then we will set the desired baud rate (115200) to the baud rate register, with the correct formula seen in the introduction. Next we need to set up the correct data stream configuration, which will be 8-bit data with no parity bit. Lastly, we want to enable the actual pin on the rx and tx, making sure we can receive and transmit data.

```
void Uart_Init(unsigned long baudRate){
    BOARD_Init();
    U1MODE = 0; //clear UART
    U1BRG = BOARD_GetPBClock() / (16 * baudRate) - 1;
    U1MODEbits.PDSEL = 0b00; //8N1
    U1MODEbits.STSEL = 0b0; //1 stop bit
    U1MODEbits.ON = 1; //enable UART

    U1STAbits.UTXEN = 1; //enable TX pin
    U1STAbits.URXEN = 1; //enable RX pin
```

(Figure 5: UART setup and configuration.)

Lastly, we want to write our main function to see our UART is working properly. We will be using a while 1 loop, this makes it will forever look for incoming characters from our computers. Using the status bit, we can check if a character has arrived at the rx pin, if so, then we want to echo this character back by sending it out with the transmit, so we will set the rx pin to the tx pin.

2.4 Lab Part 2 Introduction

After successfully setting up our UART, we want to add circular buffers to our rx and tx.

Everytime a character arrives at the rx pin, it will be put into a rx buffer, which will store the data. Similarly, there will be a tx buffer, and everytime we want to transmit, data is taken off the buffer and sent to the tx pin. In addition, we will be using an ISR to control when to read and write to these buffers. Here, we want to focus on how we can use UART to print stuff to Coolterm, but rerouting the default "printf" function. We should be able to call printf in our code and see it displayed on Coolterm.

2.5 Lab Part 2 Initial Approach

My initial approach to this part is to create a circular buffer ADT that will be managing my data. In addition, we need functions to manipulate these buffers, we need to be able to write and read from the buffers, which requires specific algorithms. We first need to create functions that will help us correctly read and write data from buffers, and keep track of the head and tail increments. The ISR will be checking for flags raised by these functions, which will tell us a certain action has to occur. Lastly, we will have more registers to configure, because we are now working with interrupts and flags, which have their own specific registers.

2.6 Lab Part 2 Implementation

The interrupts and flags have to be initialized, the priority of interrupts must be set correctly.

```
    U1STAbits.URXISEL = 0b00; //config the interrupt register for rx
    U1STAbits.UTXISEL = 0b00; //config the interrupt register for tx
    IEC0bits.U1RXIE = 1;
    IEC0bits.U1TXIE = 1; //enable flags
    IPC6bits.U1IP = 0b100; //set priority for interrupts, since only one device doesnt rly matter
}
```

(Figure 6: set up for interrupts, flags and priority.)

Now we need to construct our circular buffer ADT, with the structure given in the introduction, we are able to create a struct, which I will be using pointers to manipulate the ADT. This ADT however needs to be properly initialized, sufficient memory must be allocated for the ADT to work. In this case, I used a static memory allocation for my ADT, and in addition, the default or starting values of my buffer must be set, which I have created a function for.

```
struct BufferObj rxBufferStatic;
struct BufferObj txBufferStatic; //allocate memory for the BufferObj struct
Buffer rxBuffer = &rxBufferStatic;
Buffer txBuffer = &txBufferStatic; //set the pointer to the address of memory

void Buffer_InitStatic(Buffer cb){
    //instead of memory allocation in the init function, it takes the already allocated memory and set values to it.
    cb->head = 0;
    cb->tail = 0;
    cb->full = 0;
    cb->err = 0; //set all 0s, init
}
```

(Figure 7: Circular buffer memory allocation and initialization.)

We need to be able to write and read data from these buffers, so I made two functions which take care of inserting or fetching data, incrementing head or tail, checking buffer status and raising interrupt flags. In both functions, we have some sort of status check before action is taken, for example, we cannot read a buffer that is empty, or write to a buffer that is full. After this check, we are able to actually insert or fetch data, and as each action, we need to move the head or tail. After the head and tail has been moved, we want to check if the buffer is empty or full, which will update our buffer status. Lastly, we want to raise the interrupt flag to let the ISR know we have completed a write or read, and another action must be taken.

```
unsigned char GetChar(Buffer cb, unsigned char *data){
    //checks if head and tail caught up, and it is not full (meaning empty buffer nothing to read)
    if ((cb->head == cb->tail) && (!cb->full)){
        cb->err = 1;
        return '?'; //if seg fault check here
    }
    //otherwise read the data from the head and set it to data
    *data = cb->buffer[cb->head];
    cb->head = (cb->head + 1) % BufferSize; //increment head after it has been read
    if (cb->full){
        cb->full = 0; //if the buffer was full, after reading it it should have become not full
    }
    IFS0bits.U1RXIF = 1;
    return *data;
}
```

(Figure 8: The read function for a circular buffer.)

With working circular buffers, we want to complete the ISR so we know what to do when an interrupt is set from interactions with buffers. In our ISR, we are continuously checking for flags raised, which are set in IFS0 registers. In our handler, as soon as a flag is raised, we want to lower the flag, acknowledging the flag has been recognized. Then we need to control the data flow from our rx and tx pins and into our buffers, for example, if a data has been inserted into the tx buffer, and then we want to send it out the tx port so we are able to read it. Therefore, in our write data function, we set the tx flag high, and the ISR will be alerted and respond by taking the data off the tx buffer and sending it out the tx pin. The opposite logic can be applied for the rx buffer and the read function, as we take data off the rx buffer, more space is available. By setting the rx flag high, it alerts the ISR to write more data from the rx pin into the rx buffer.

```
void __ISR(_UART1_VECTOR) IntUart1Handler(void) {
    //your interrupt handler code goes here
    //check if rx flag has been raised
    if (IFS0bits.U1RXIF == 1){
        //check if rx has data
        IFS0bits.U1RXIF = 0; //clear the rx flag, located at RXIF
        while (U1STAbits.URXDA){
            //only write to rxBuffer it is not full
            if (rxBuffer->err != 1){
                PutChar(rxBuffer, U1RXREG);
            }else{
                rxBuffer->err = 0;
            }
        }
    }

    //check if tx flag has been raised
    if (IFS0bits.U1TXIF == 1){
        //check if tx buffer has at least one more space
        IFS0bits.U1TXIF = 0; // clears the tx flag, located at TXIF
        if (!U1STAbits.UTXBF){
            unsigned char var; //intermediate variable due to data not passed in correctly
            GetChar(txBuffer, &var);//GetChar reads data from txBuffer and set it to "var"
            if (txBuffer->err != 1){
                U1TXREG = var;
            }else{
                txBuffer->err = 0; //clear txBuffer err flag
            }
        }
    }

}
```

(Figure 9: ISR function.)

2.7 Lab Part 3 Introduction

In the last part, we want to create a protocol system utilizing the UART and buffers we have created in previous parts. We want to be able to receive and send packets, as well as responding to packet information. With the use of the Python script, the main lab interface, we want to control lighting up the LEDs and sending out packets with current LED status, indicating which ones are lit. Lastly, we will be reading an incoming packet, with an integer payload, divide the payload by 2 and then send it back out.

2.8 Lab Part 3 Initial Approach

First, incoming packets have to be recognized and digested by our code. An ADT structure will be used to save information of a packet, each packet contains various parts and a variable payload size. A state machine will be used to parse and categorize different parts of our packets, for example the ID or the payload. This will tell us when we have received a complete and valid packet, saved to our packet ADT. Then save it in a packet circular buffer. Just like the circular buffer created before, except this time instead of storing characters we are storing the packet ADTs. Finally, we have the application layer, which needs to be able to tell what type of incoming packet is received by the ID. Then it needs to take it off the circular buffer and decide what actions to complete based on the type of packet.

2.9 Lab Part 3 Implementation

The state machine will be implemented with a switch case, and using enums, we will define the different states. Along with other global variables to keep track of our status inside the

state machine.

```
typedef enum {
    WAIT_FOR_HEAD,
    READ_LENGTH,
    READ_ID,
    READ_PAYLOAD,
    READ_TAIL,
    READ_CHECKSUM,
    PROCESS_PACKET
} State; //define my states
static uint8_t payloadLen = 0; //payload length counter
static unsigned char checksum = 0; //checksum counter
static State currState = WAIT_FOR_HEAD; //initialize first state
```

(Figure 10: State machine setup.)

By constantly calling the read function from part 2 of the lab, characters will be pulled in constantly and parsed by our state machine. With each incoming character, if it is recognized then it enters a state of building the packet. The structure of our packet is shown below.

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

Table 1: Protocol Packet Structure

(Figure 11: Packet structure.)

Inside the state machine, each part of the packet is represented by a state, as incoming characters will be recognized and the state machine will begin looking for the next section of the packet. For example, if the head byte is recognized, the state machine will transition to looking for the length, and so on and so forth. Throughout this process, the checksum will be calculated by a function as the state machine parses. This function will continuously calculate the Berkeley Software Distribution checksum, which at the end will be compared to the checksum received. In addition, as the state machine parses, it will add the information to our packet ADT, forming a complete packet in the process. At the end, it will take the completed packet ADT and add it onto the packet circular buffer.

```
uint8_t BuildRxPacket (){
    unsigned char byte;
    while (GetChar(rxBuffer, &byte) != '?'){ //GetChar read data in
        switch(currState){
            case WAIT_FOR_HEAD: //checks for head to start reading
                if (byte == HEAD){
                    currState = READ_LENGTH;
                }
                break;
            case READ_LENGTH: //checks for length byte
                myPacket->len = byte; //set relevant data to our packet
                currState = READ_ID; //move next state
                break;
            case READ_ID: //checks for id byte
                checksum = Protocol_CalcIterativeChecksum(byte, checksum); //add to checksum counter
                myPacket->id = byte; //set packet id
                payloadLen++; //keep track length of the payload
                if (payloadLen == myPacket->len){
                    currState = READ_TAIL; //check the case if len = 1, packet contains only ID in payload, no actual payload to read. goes straig
                }else{
                    currState = READ_PAYLOAD;// otherwise there will be actual data in payload to be read
                }
                break;
```

(Figure 12: Part of our state machine.)

The packet is now on the circular buffer, using the same algorithm and logic as our old buffer. Now we can move to the application stage, where we will have a function that looks at the ID of the next packet in line in our buffer. This function will determine what type of packet is next in line, it doesn't take it off the buffer, only reads it.

```
unsigned char Protocol_ReadNextPacketID(void){
    myPacket = myPacketBuffer->buffer[myPacketBuffer->head]; //read the next packet from buffer, but doesnt take it off the buffer
    return myPacket->id; //we want to read the id of the next packet
}
```

(Figure 13: Function to read the next packet ID.)

After determining the type of packet, we have three possible actions to do. First, lighting up LEDs, by simply using the LED macros defined in the .h file, we take the packet off the buffer with our read function and set the payload to our LED.

```
if (Protocol_ReadNextPacketID() == ID_LEDS_SET){ //check if the packet type set leds
    Protocol_GetInPacket(&id, &len, payload); //read packet off buffer, payload is updated
    LEDS_SET(myPacket->payload[0]); //get the first byte that hold the hex to indicate which leds to set
}
```

(Figure 14: Setting LEDs with our "read" function.)

Second, the status of the LED needs to be sent out in a packet. By using the LED macros, we can get the status, which can be sent out using a send packet function.

```
else if(Protocol_ReadNextPacketID() == ID_LEDS_GET){ //check if packet is ID_LEDS_GET, have to send led status packet back
    uint8_t ledState = LEDS_GET(); //get the current status of leds
    Protocol_SendPacket(0x02, ID_LEDS_STATE, &ledState); //send a packet out containing led status
}
```

(Figure 15: LED status implementation.)

This function takes the relevant parts of a packet as arguments, and constructs a complete packet and sends it out to the tx buffer we have created earlier. Then from the tx buffer it will be sent out the tx pin and into our computers.

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

    PutChar(txBuffer, HEAD);
    PutChar(txBuffer, len);
    PutChar(txBuffer, ID);
    checksum = Protocol_CalcIterativeChecksum(ID, checksum); //construct a complete packet
    //parse through the packet one character at a time and calc checksum as we parse
    for (int i = 0; i < len - 1; i++){
        unsigned char byte = payload[i];
        PutChar(txBuffer, byte);
        checksum = Protocol_CalcIterativeChecksum(byte, checksum);
    }
    PutChar(txBuffer, TAIL);
    PutChar(txBuffer, checksum);
    PutChar(txBuffer, '\r');
    PutChar(txBuffer, '\n');
}
```

(Figure 16: Send packet function.)

Third, we have our integer division packet type, called ping and pong. We get the integer from reading the packet off the buffer, obtaining the payload. Next, we have to convert the endianness of this payload. By manually converting the payload, an array of unsigned chars, to an unsigned int to do division, the byte order will be put into the correct ordering. Assigning each unsigned char to a specific part of the 32 bit of the unsigned int using bit shifting and masking. That creates an integer that can perform division on, and afterward it is converted from unsigned int back into an array of unsigned chars. Again, manually assigning each of the 4 bytes from the integer to a correct order of array. Lastly, it is sent out using our send packet function and received by our computer.

```
else if (Protocol_ReadNextPacketID() == ID_PING){ //check is packet type is PING
    Protocol_GetInPacket(&id, &len, payload); //read packet off the buffer
    /*converting unsigned char array to an unsigned int manually, this way of explicitly constructing the byte in a specific
    order makes it so it can work on any devices regardless of its endianness*/
    unsigned int var = (unsigned int)payload[0] << 24 | // shift the first byte left by 24 bits
            (unsigned int)payload[1] << 16 | // shift the second byte left by 16 bits
            (unsigned int)payload[2] << 8  | // shift the third byte left by 8 bits
            (unsigned int)payload[3]; //add the fourth byte to the end

    var = var / 2;

    unsigned char data[4];
    //converting the unsigned int back to unsigned char, again explicitly managing the byte order.
    data[0] = (var >> 24) & 0xFF; // extracts the first (most significant) byte
    data[1] = (var >> 16) & 0xFF; // extracts the second byte
    data[2] = (var >> 8) & 0xFF;  // extracts the third byte
    data[3] = var & 0xFF;  // extracts the fourth (least significant) byte
    Protocol_SendPacket(0x5, ID_PONG, &data); //construct packet and send it out
```

(Figure 17: ping and pong implementation.)

3 Conclusion

3.1 Summary

This lab was incredibly interesting and it taught me an insane amount of knowledge starting from hardware all the way to the software application. It taught me how to read a datasheet for control registers and configuration of hardware on microcontrollers. In order for any hardware to function properly, it has to be configured correctly. Making an ISR function from scratch gave me a deep understanding of how interrupts work. This type of programming was never really taught before, as before the ISR functions were given. The ADTs reinforced my learning from CSE101, which was all about creating and manipulating ADTs like linked lists, matrices and graphs. It also reinforced how pointers and memory allocation work in C code. Lastly, and most importantly, the use of UART and serial communication with packet protocols gave me an insight into some real world communication processes. How two devices can talk to each other using UART and how packets are used to send data in a network.

3.2 Challenges

The implementation of low level hardware code all the way up to application software takes a lot of knowledge and time. I have never worked with embedded systems before, and this was the first time reading datasheets and configuring control registers. That was a big learning curve for me, as these files were provided for us before. I have never looked in depth into a microcontroller datasheet before, how the hardware actually works under the hood had to be figured out. The ISR was also one of the biggest challenges in my opinion, as it is the first time I am writing this type of function. Its structure and logic was unfamiliar to me, so it took me a good amount of time to properly implement it. The ADTs were not too bad, but some memory allocation caused a couple segmentation faults in my code early on. Finally, debugging this type of hardware code is incredibly difficult and time consuming. Using the JTAG debugger was very different from what we are used to in other coding classes. However, once you get the hold of the debugging tools, it is very powerful and can give A LOT of insight into where exactly in your code is broken.