Lab #1 Serial Device Driver and Protocol Communication

Daniel K. Krygsman

University of California, Santa Cruz

Baskin School of Engineering

ECE 121L: Lab Number 45444: Section 01D: Microcontroller System Design Lab

Professor Stephen Petersen

March 16, 2023

Abstract

The purpose of this lab is to establish serial communication between the pic32 development board and the lab interface on our computer. This is done through a uart peripheral system of the pic32 and a needed uart device driver. To understand the process, the lab consists of three parts,

1) Initializing and using the serial port,
2) Creating a full duplex interrupt driven rx/tx serial device driver, and
3) Utilizing the device driver to create a packet application.

The progression of this lab allows for a clear understanding of microcontroller serial communication, including the use of multiple circular buffers for short term storage of information and the use of urgent requests indicated by the UART subsystem.

Introduction

Coming into this class with no experience of embedded systems proved to be challenging, as interfacing software with microcontroller hardware varies with each family. In

this case, an understanding of interfacing the pic32 comes solely from the family reference manual, and from a thousand page document understanding how to find information is key. With some understanding, combined with the proper uart initialization and configuring the receive and transmit registers, allows us to accurately receive and retransmit single byte characters. With this we move to the second part of the lab and create circular buffer functions to allow short term storage for characters. The ISR interrupts our main program to enqueue to the rx buffer and dequeue the data from the TX buffer when needed. This allows us to send entire strings of characters without losing data. This is the main part of the lab as correct interrupt code configuration is the heart of all serial communication. The last step utilizes this device driver and incorporates it with a packet state machine, which builds packets as they incrementally come character by character. These data packets, depending on their payload ID perform various tasks such as setting leds, reading lit leds, or endianness conversion. With all parts working properly the microcontroller can predictably receive and interpret data as defined by the packet protocol.

Uart

The lab involves interfacing the uart of the pic32 development board. The uart has a hardware shift register and a buffer for both receive and transmit. Data is received at 16 times the baud rate to ensure reading. Unlike the software versions of these buffers, the hardware versions are only 9 bit wide. This is why they work only for the 8 bit, 1 byte character needed for check off 1. For everything after checkoff one, short term storage is needed to hold data for the time the rx register receives data to when the data is used in software. To solve this issue we create circular buffers in software that work in unison with interrupting code. The UART is initialized to raise interrupt flags whenever characters come in, this interrupting code pulls data out of the rx register and loads it to a buffer. With this we can receive strings of data without overrunning and corrupting data. Interrupts happen randomly during a program's normal running code, because of this we must consider interrupting code in the worst case situation. PutChar, the function responsible for enqueueing data to the TX software buffer and raising interrupts to take then print that data, must be written in a way where interrupting code does not lose interrupting flags if the hardware receive buffer interrupts at an undesirable time.

Circular Buffers

As discussed, circular buffers are created in software to hold data for the short time before they are able to be taken out of the rx side and used, or taken out of the tx side and printed. With two channels of data flow, functions are made using pointers, pointing to two instances of circular buffer structs. The functions include

`int Buff_Init(circular_buffer *w)`

- Sets all struct variables head, tail, and full flag to 0.

`int is_buff_empty(circular_buffer *w)`

- Checks if the desired circular buffer is empty based off of its head, tail, and full flag.

`int is_buff_full(circular_buffer *w)`

- Checks if the desired circular buffer is full based off of its head, tail, and full flag.

`unsigned char take_from_buff(circular_buffer *w)`

- Checks if the buffer is empty, and if not takes data from the head, increments head and returns data.

`int add_to_buff(circular_buffer *w, char ch)`

- Checks if the buffer is full, and if not assigns passed in data to the data array, increments tail, and checks if the buffer is full. If the buffer is full, the function raises the full flag.

These functions work in unison to create multiple instances of data arrays that can be read from and written to as needed. The multiple instances of these buffers are defined as instances of the same structure.

```
typedef struct{
   int head;
   int tail;
   unsigned char data[BUFFER_SIZE];
   int full_flag;
}circular_buffer;

static circular_buffer tx_buffer;
static circular_buffer rx_buffer;
```

These functions and structures are needed for the microcontrollers device driver, but are useless without correctly written and initialized interrupt functions.

Interrupt Functions

I am referring to Uart_Init, PutChar, GetChar, and the ISR all as my interrupt functions as it helps me think and refer to them as they all are related to the ISR. I am aware that the ISR is the only function that actually interrupts the main program.These functions are as follows

`void Uart_Init(void);`

- Initializes interrupt priority, subpriority, select bits for RX and TX, and interrupt enable

`int PutChar(char ch);`

- Enqueues passed in data to tx circular buffer and raises interrupt flag depending on TX buffer status and RX shift register status

`unsigned char GetChar(void);`

- Dequeues and returns RX buffer data

`void __ISR(_UART_1_VECTOR)IntUart1Handler(void)`

- Enqueues RX buffer depending on status and lowers interrupt flag if the receive buffer raised the flag. Dequeues TX buffer and sends data to TX register depending on TX hardware and software buffer status. ISR also takes care of collisions encountered when RX interrupt is raised while TX enqueue is busy

The largest and most important part of configuring PutChar and GetChar with the Interrupt Service Routine is dealing with collisions. Collisions occur when the RX hardware buffer raises an interrupt during TX software enqueueing and if not done correctly lowers needed flags causing data to be stuck in software buffers never to be dequeued. Proper understanding of the RX and TX select bits along with code written to those select bits eliminates this problem and full strings can be written without a loss of data.

UART.c

These circular buffer functions and interrupt functions are the needed parts for checkoff 2 which is a properly functioning device driver. The program's main loop runs and constantly checks if data is received in the microcontrollers hardware shift register and buffer, as characters come into these hardware components the ISR is triggered to enqueue data to the RX software buffer. Large strings of data force the ISR to constantly be raised and enqueues characters until there is no more data in the RX hardware components. With this done and the ISR is no longer raised because of the RX track, the main program runs GetChar which dequeues the RX buffer and passes the enqueued data to PutChar. GetChar and PutChar run for each character, this

enqueues the data taken from the RX buffer to the TX buffer and sends it out the UART via the TX track of the UART. When done correctly GetChar PutChar and the ISR work together to send complete strings of characters and the device driver is done.

Protocol

The third and last part of the lab consists of using our working device driver to conduct packet protocol communication. Creating and sending data packets is a process used for data integrity in serial communications. Data gets added to the packet's payload, with the first character being the payload's ID. A fully formed packet is made up of a predefined head character, the length of the payload, the payload itself, a predefined tail, the iterative checksum, and two end characters. The third part of the lab is to use this packet notation and run an application that performs certain operations. We will receive a string message through the UART, and if the data is valid make a packet. Depending on what data our packet has, various operations must be done. This application needs functions which use all of our device driver functions. The functions needed are

`int Protocol_Init()`

- Initializes board, uart, RX and TX buffers, LED functions, and sets all flags to their proper initialization

`int Protocol_SendDebugMessage(char *Message)`

- Takes in a message and performs a array shift then prints our message as a packet via the send packet function

`uint8_t Protocol_QueuePacket ()`

- A for loop that dequeues data from the RX buffer and returns it to the build RX packet function

`int Protocol_SendPacket(unsigned char len, unsigned char ID, void *Payload)`

- Takes in the important aspects of a packet and uses PutChar to print the full packet, utilizes a for loop that displays all characters of the payload.

`uint8_t BuildRxPacket (rxpADT *y, unsigned char input)`

- The most intensive of the protocol functions which gets called each time a character is received from GetChar. Build RX Packet is a state machine that

cycles through states of building a proper packet. Once a packet is made the packet ready flag is raised to be utilized later.

`unsigned char Protocol_CalcIterativeChecksum(unsigned char charIn, unsigned char curChecksum)`

- A function used inside the send packet function which calculates the checksum for each value of the payload to then be displayed.

As well as these functions, copies of the buffer functions used in part two must be made to allow for the parameter of a different structure. This structure is identical to the circular buffer structure except the data array must be of type struct. In this circular buffer we are enqueuing whole structures instead of characters. This is very intuitive because the function stays the same except for the data array's type.

```
int pkt_Buff_Init(packet_buffer *x)
int is_pkt_buff_empty(packet_buffer *x)
int add_to_pkt_buff(packet_buffer *x, rxpADT y)
rxpADT take_from_pkt_buff(packet_buffer *x)
int is_pkt_buff_full(packet_buffer *x)
```

The structures used in these functions are important because they show the levels of which the data is held. The largest structure is the packet buffer which is the modified circular buffer, and inside that is the rxpADT buffer which hold all the important packet variables.

```
typedef struct rxpT {
   uint8_t ID;
   uint8_t len;
   uint8_t checkSum;
   unsigned char payLoad[MAXPAYLOADLENGTH];
 }rxpADT;
```

```
typedef struct{
   int head;
   int tail;
   rxpADT data[PACKETBUFFERSIZE];
   int full_flag;
}packet_buffer;
```

For the same reason we added character to the character circular buffers in part one, entire packets are added to the packet buffer to sit before the interrupting code takes them out. After a packet has been successfully made it gets enqueued to the packet buffer then when used it gets taken out and used. The use of the packet buffer is intuitive because of the use and understanding of the character buffers. For the sake of relevant information, I will only be discussing the build rx packet function in depth because the others consist of correctly implementing certain operations or configuring these functions with the device driver.

Build RX Packet

Build RX Packet is a state machine that gets called for each character. This function is inside my queue packet function which sits in the while loop of my main program. The states of this state machine are 1)HEAD 2)LEN 3)ID 4)PAYLOAD 5)TAIL 6)CKSUM 7)\R 8)\N these are all of the values of a packet. This is important because a length two payload will be called 8 takes, once for each character. Before the function is called the state must be initialized to HEAD, once it receives the correct head character, 0xCC, it sets the state to LEN. The function will be called again once another character has been received, in this state it checks if the length is of valid size, and if so gets assigned to the packet's length variable, then sets the state to ID. The ID is loaded into the ID variable of the packet, if the length received is 1 then we change to the tail state, if it's bigger than 1 it will go to the PAYLOAD state. The PAYLOAD state loads the value to the payload[1] variable of the struct then leaves the function. Each consecutive function call will fill up incrementing payload variables until the length is reached, it will then move to the TAIL state. Tail state does not load any value, it simply checks to see if the next character passed in of hex value 0xB9, this is a check to make sure the incoming packet is correct, if it is then it will proceed to the next state, CKSUM. In CKSUM the iterative checksum which is

calculated after each value of the payload is checked with the passed in value to verify if they are correct, if so it proceeds. \R and \N iterate through, and in \N we can now raise the packet ready flag and do packet actions as we have verified an entire packet is successfully made. If at any of these steps an incoming character is not valid then the state is set to HEAD and keeps running until a valid one is made. In \N, if ID received is the predefined LED SET ID then I initialize the led's to whatever the passed in payload is, if the ID is the predefined LED GET ID then I change it to the LED STATE ID change the length to 2 and set payload[1] of the packet to the current lit leds. IF the received packet has the predefined PING ID then I change it to the PONG ID, endian conversion is needed for a PING message but is done outside of my build rx packet. The function now enqueues the packet to the packet buffer via the add to packet buffer function. With this the build rx packet function has successfully built a packet one character at a time and enqueued for future use.

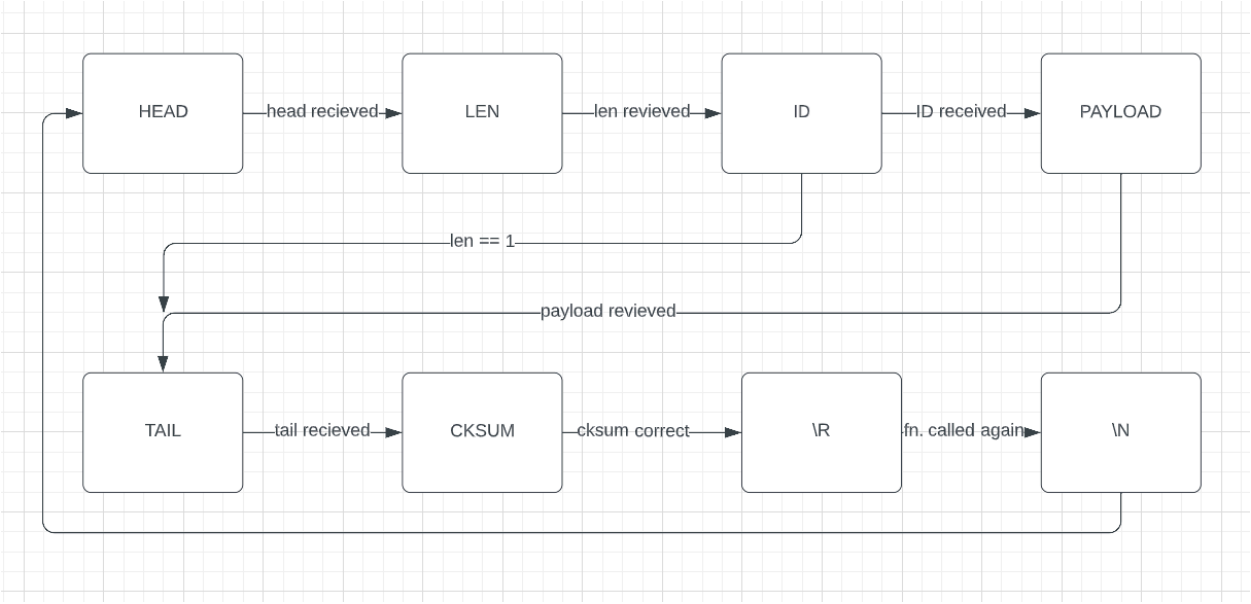

Figure 1: Build RX Packet state diagram

Application

The application of my program only runs if there is a packet ready. If the packet ready flag from the build rx state machine is raised then we enter the section of code which is the application. The flag is then lowered and the packet is dequeued from the packet buffer. Now I

check between three ID options. If the packet dequeued has the SET ID or ID STATE which was changed from GET, print the packet using the send packet function. If the packet has the PONG ID which was also changed from PING in the state machine then i convert the payloads endianness by converting the payload chars into a unsigned int with the little endian order, dividing that int by 2 and reassigning to the char payload using bitwise shift and logical AND to get the desired bits. After this the converted packet is printed out. This completes the objective of the lab.

Conclusion

From interfacing the Uno32's UART peripherals, incorporating it with software to create a functioning device driver, and being able to use that device driver to create a packet protocol communication application, we are able to understand how to create a serial communication embedded system. Learning how to interpret literature from the manufacturer proved to be the hardest part of the lab, but with persistence, an understanding of the information is possible. Interfacing the interrupts also proved to be very challenging as specific cases arise with various interrupt selection. My biggest mistake was not fully trying to digest what the family reference manual is saying with control registers, and doing so could have saved me hours of time and headaches. The way one choses when an interrupt is raised determines the approach of their code. I went through many iterations of my PutChar and ISR, with different interrupt select options before I could finally find one where collisions make sense and can be correctly taken care of. I am a firm believer that without a proper understanding, no progress can be made.