Lab 3 - UART Communication

Matt Douglass, Priyanka Makin, Zach Passarelli, Savio Tran

October 13, 2016

ECEN 2020

The goal of this lab is to understand transmitting, receiving, and analyzing data using UART communication and the RealTerm interface. We first experiment with transmitting data to the RealTerm terminal. Then, we set up the MSP432 so that it can receive character input from the RealTerm console and relay it back. We also construct a circular buffer to store input from RealTerm and configure methods to read and analyze the data stored.

I/O and UART Configuration

When preparing the MSP432 for UART communication there are is a whole list of things we had to be sure we configured correctly. First, we had to configure the e_USCI A0 UART module to be in the primary mode configuration. Then, using the given equations,

$$N = f_{BRCLK}/Baudrate$$
$$UCA0BRW = Integer(N)$$

we determined the appropriate DCO frequency (3 MHz) for our chosen Baud rate of 115200. We would later discover, unfortunately, that we incorrectly configured it to 38400. After that we had to configure the UCA0CTLW0 register along the specifications given to us in step 1 of the lab (no parity, no address bit, LSB First, 8-bit data, 1 start/stop bit).

Then we had the task of testing our configurations by transmitting the ASCII data 0xAA onto the RealTerm terminal repeatedly. Figure 1 shows an oscilloscope reading of this data.

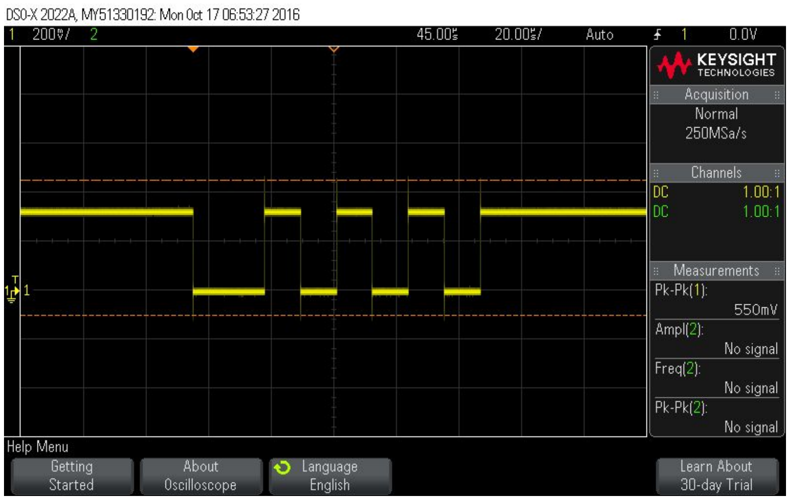

Figure 1 - 0xAA Transmission

We can actually check that the number 0xAA got transmitted correctly by analyzing this reading. 0xAA is equivalent to 0b10101010. In lecture, we also learned that start bits are generally high and stop bits are low. And, if we look back at our configuration it is important to notice that we specified that the least significant bit, or LSB, should be read first. So the values of the communication frame are 0010101011 which is depicted exactly in the scope reading of Figure

1. The reading also showed us that the transmission time of this piece of data was roughly 100 microseconds. Knowing that we transmitted 10 bits, we calculated a bit rate of 100 kilobits per second.

Sending Data

Next we took the transmission tests to the next level by attempting to transmit arrays of data to RealTerm. We constructed a function that took a pointer to an array of characters and it would iterate through the array and transmit data character by character to the RealTerm terminal. Due to the aforementioned Baud rate error, the resulting characters on the terminal did not match up to our inputs, however the quantity of data printing did make sense. For example, transmitting the string "Why not Zoidberg?", which consists of 17 characters and 1 null terminator value, would produce 18 values on RealTerm. Despite the Baud rate issue, this consistency gave us reason to believe the iteration functions worked.

Receiving Data

We experimented on the MSP432's ability to transmit data, but we also wanted to explore how it can receive data. By configuring the RX interrupt functionality of the UART module, we were able to get the MSP432 to recognize and handle input entered into the RealTerm terminal. We could verify our inputs by having the interrupt handler store the input data and pass it into the transmission function, thus having our typed inputs appear directly in the terminal, after some transmission delay. Again, the Baud rate error would provide some inaccurate feedback.

Circular Buffer

The circular buffer utilizes the heap. First, we created a structure containing a pointer to the buffer, pointers for the head and tail, a variable containing the length of the buffer, and a variable to hold the number of items in the buffer. In order to initialize the buffer, memory is allocated using the malloc function, and the pointers are allocated to this memory. When adding items to the buffer, the head pointer is dereferenced and set equal to the value we want stored. Then, the moved to the next location depending on if it at the end of the buffer. The tail does the same, but rather than writing in data, it deletes old data before moving. However, these two functions also have checks to prevent the tail from ever passing up the head, which would result in data leakage. While items are added and subtracted, the variable holding the number of items in the buffer is incremented or decremented, respectively. This allows us to check if the buffer is full or if it is empty.

We utilized the circular buffer by adding inputs from RealTerm to the buffer. The code was designed to also handle cases when the RX interrupt received an “ENTER” character as well as when the circular buffer becomes full. We also would program the buttons on the MSP432 to trigger an interrupt in order to dump the buffer or clear the buffer as needed. Due to time constraints this section of code never came to fruition, though we have a good idea of how these concepts can be constructed.

Real World Applications

Using UART to transmit bits (information) from our written code to the RealTerm terminal and the MSP432 is an example of serial communication. Serial communication is the process of sending data one bit at a time over a channel or computer bus. Serial communication was originally designed to transfer data over a relatively long distance through some sort of data cable. Most computers are designed with serial ports for keyboard, mouse, and/or Ethernet connection. Also, serial communication is also implemented in the design of integrated circuits. When transmission speed is not an issue, circuits can be connected by a serial bus made of signal traces for effective communication.

Conclusion

Data transfer and communications are important. Information can be sent from peripherals to an embedded system for it to process. There is often more data coming in than a system can handle at once, so there must be a way to deal with incoming data without losing it. Circular buffers can be used to handle large amounts of data without allocating too much memory. In this lab, the buffer is utilized to hold the input characters until they are analyzed and then deleted. This is just an example of many applications of circular buffers. Circular buffers are useful for any data in a queue. So, if a network is slower than programs that are sending it data, the data can be stored in the buffer until it can be processed at a later time, thus allowing the program to process everything without losing data.

This lab was very difficult for us. While our code to transmit data was correct, as proven on the oscilloscope readings, RealTerm did not interpret our data correctly. We ultimately discovered that our Baud rate was incorrectly configured to a value other than 115200, which was the rate RealTerm was expecting to read. In future projects involving peripheral communication we must put extra effort into checking our Baud rate for correctness before proceeding.

Appendix

The following pages contain code for main.c and our circular buffer code CircBuff.c and CircBuff.h.

```c

#include "msp.h"
#include "CircBuff.h"
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define Proc2 //Single character transmit test
#define Proc3 //Multiple character transmit test
#define Proc4 //Why not Zoidberg?
#define Proc5 //Input test
#define Proc6 //Circular Buffer

void configure_clocks();
void configure_serial_port();
void uart_putchar(uint8_t tx_data);
void uart_putchar_n(char*data, uint32_t length);

void EUSCIA0_IRQHandler();

void main(void)
{

    WDTCTL = WDTPW | WDTHOLD; //Stop watchdog timer
    configure_clocks();
    configure_serial_port();

    __enable_interrupt(); //Global interrupt enable

#ifdef Proc2 //Transmit character 0xAA repeatedly
    while(1){
        uart_putchar(0xAA); //Pass in data 0xAA to putchar function
    }
#endif
#ifdef Proc3
    uint8_t proc3data[4] = {"g","o","o","d"}; //Array of test characters
    uint8_t*dataptr = (uint8_t*)proc3data; //Create pointer to test array
    //Calculate length of array
    uint32_t p3dlength = sizeof(proc3data)/sizeof(uint8_t);

    uart_putchar_n(dataptr,p3dlength); //Pass pointer and length into putchar_n

#endif
#ifdef Proc4
    char proc4data[] = "Why not Zoidberg?"; //Create test string
    char*dataptr = (char*)proc4data; //Create pointer to test string
    //Calculate length of test string
    uint32_t p4dlength = sizeof(proc4data)/sizeof(char);

    uart_putchar_n(dataptr,p4dlength); //Pass pointer and length into putchar_n
#endif
#ifdef Proc6
    CircBuff cbuf; //Create CircBuff structure
    CircBuff *cb; //Create CircBuff pointer
    cb = &cbuf; //Link pointer to structure
    InitializeBuffer(cb,256); //Initialize Buffer function, size 256 characters
#endif
}

void configure_clocks(void){
    CS->KEY = 0x695A; //Unlock CS module for register access
    CS->CTL0 = 0; //Reset tuning parameters
    CS->CTL0 = CS_CTL0_DCORSEL_1; //Setup DCO clock (3 MHz)
    //Select ACLK = REFO, SMCLK = MCLK = DCO
    CS->CTL1 = CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3;
    CS->KEY = 0; //Lock CS module for register access
}

void configure_serial_port(void) {
    //Configure UART pins, set 2-UART pin as primary function
    P1SEL1 &= ~(BIT2 | BIT3);
    P1SEL0 |= BIT2 | BIT3;

    //Configure UART
    UCA0CTLW0 |= UCSWRST; //Put eUSCI in reset
    //Select Frame parameters and clock source
    UCA0CTLW0 &= ~UCPEN; //Disable parity bit
    UCA0CTLW0 &= ~UCMSB; //Set LSB first
    UCA0CTLW0 &= ~UC7BIT; //Set 8-bit data
    UCA0CTLW0 &= ~UCSPB; //Set 1 stop bit
    UCA0CTLW0 &= ~UCMODE0; //Disable address bit
    UCA0CTLW0 &= ~UCMODE1;
    UCA0CTLW0 |= UCSSEL0 | UCSSEL1; //Set SMCLK = BRCLK

    UCA0BRW = 26; //Set Baud Rate (38400)
    UCA0CTLW0 &= ~UCSWRST; //Initialize eUSCI

    UCA0IE |= UCRXIE; //Enable USCI_A0 RX interrupts

    NVIC_EnableIRQ(EUSCIA0_IRQn); //Enable eUSCIA0 interrupt in NVIC
}

void uart_putchar(uint8_t tx_data){
    while(!(UCA0IFG & UCTXIFG)); //Wait for transmitter ready
    UCA0TXBUF = tx_data; //Load data onto buffer for transmission
}

void uart_putchar_n(char *data, uint32_t length){
    uint32_t i;
    for(i = 0; i< length;i++){ //Iterate through data
        //Transmit 1 character per iteration using putchar function
        uart_putchar(data[i]);
    }
}

//Int to ASCII conversion function (incomplete)
//char* itoa(int val, int base){
//
//  static char buffer[32] = {0};
//  uint8_t i = 30;
//  for(; val && i ; --i,)
//      buf[i] = "0123456789abcdef"[val % base];
//  return &buf[i+1];
//}
```

```c

void EUSCIA0_IRQHandler(void){
    uint8_t data;
    if (UCA0IFG & UCRXIFG) { //RX interrupt handler
#ifdef Proc5
        data = UCA0RXBUF; //Set variable to data on RX register
        uart_putchar(data); //Pass data into putchar for terminal feedback
#endif
#ifdef Proc6
        data = UCA0RXBUF; //Set variable to data on RX register
        //data conversion (incomplete)
        //(incomplete)
        //if(data == 0xD) //Conditional for "Enter" input
        // //Conditional for full buffer
        //AddItemToBuffer(cb,data);
#endif
    }
}

```

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#ifndef CIRCBUFF_H_
#define CIRCBUFF_H_
    typedef struct CircBuff_t{
        volatile uint8_t*head;
        volatile uint8_t*tail;
        volatile uint32_t num_items;
        volatile uint32_t length;
        uint8_t * anchor;
    } CircBuff;



#endif /* CIRCBUFF_H_ */

```

```c

#include "CircBuff.h"

void InitializeBuffer(CircBuff*buf, uint32_t size){
        buf->length = size; //Set length value of buffer
        //Set address of buffer start in the heap using malloc
        buf->anchor = (uint8_t*)malloc(buf->length);
        if(buf->anchor == NULL){ //Conditional for failed allocation
            printf("Allocation FAILED");
        }
        buf->head = buf->anchor; //Set head to buffer start
        buf->tail = buf->anchor; //Set tail to buffer start
        buf->num_items = 0; //Set count of items in buffer to 0
    }

void ClearBuffer(CircBuff*buf){
    uint32_t i;
    buf->head = buf->anchor; //Set head to buffer start
    for(i=0;i<buf->length;i++){ //Iterate through entire buffer
        *(buf->head) = 0;  //Clear value held at address
        buf->head += 0x02; //Increment to next address
    }
}

void DeleteBuffer(CircBuff*buf){
    free((void*)buf->anchor); //Free memory allocation of buffer start
}

int8_t BufferFull(CircBuff*buf){ //Returns non-zero value if full
    if(buf->num_items == buf->length){
        return -1;
    }
    return 0;

}

int8_t BufferEmpty(CircBuff*buf){ //Returns non-zero value if empty
    if(buf->head == buf->tail){
        if(buf->num_items == 0){
            return -1;
        }
        return 0;
    }
    return 0;
}

void AddItemToBuffer(CircBuff*buf, uint8_t data){ //Adds data to buffer
    if(buf->head == buf->tail){
        //Conditional for first case after buffer initialization
        if(buf->num_items == 0){
            *(buf->head) = data; //Record data at head location
            buf->head += 0x02; //Increment head location
            buf->num_items++; //Increment counter
        }
        //Conditional for head overlapping tail
        else{
            printf("Error: Overwriting tail");
        }
    }
    //Conditional for head at the 'end' of buffer capacity
    else if(buf->head == (buf->anchor +(0x02 * buf->length))){
        *(buf->head) = data; //Record data
        buf->head = buf->anchor; //Wrap head location to start of buffer
        buf->num_items++; //Increment counter
    }
    //All other cases
    else{
        *(buf->head) = data; //Record data
        buf->head += 0x02; //Increment head location
        buf->num_items++; //Increment counter
    }
}

uint8_t RemoveItemFromBuffer(CircBuff*buf){ //Gather data from buffer
    uint8_t data;
    if(buf->head == buf->tail){
        //Conditional for empty buffer, returns NULL data
        if(buf->num_items == 0){
            return NULL;
        }
        //Conditional for removing data when head and tail match
        else{
            data = *(buf->tail); //Read data held at tail
            *(buf->tail) = 0; //Clear data held at tail
            buf->num_items--; //Decrement counter
            return data; //Return data for use
        }
    }
    //Conditional for tail at end of buffer capacity
    else if(buf->tail == (buf->anchor +(0x02 * buf->length))){
            data = *(buf->tail); //Read data
            *(buf->tail) = 0; //Clear data held at tail
            buf->tail = buf->anchor; //Wrap tail location to buffer start
            buf->num_items--; //Decrement counter
            return data; //Return data for use
    }
    //All other cases
    else{
        data = *(buf->tail); //Read data
        *(buf->tail) = 0; //Clear data held at tail
        buf->tail += 0x02; //Increment tail location
        buf->num_items--; //Decrement counter
        return data; //Return data for use
    }
}

```