

ECEN 2020 - Applications of Embedded Systems

Albert Dayn, Christopher Morroni, and
Heinz Boehmer

Due: Friday October 10, 2016

Lab 3: UART

Introduction

In this lab we were introduced to serial communication through the UART standard and learned how to implement and store data in a circular buffer, allowing it to be handled in the order it was received. This included dynamically allocating memory to the heap, receiving from and transmitting to another serial device, and tying together the use of multiple types of interrupts to ensure that nothing interfered.

1. To set UART to use no parity or address bits, LSB first, 8-bit data, 115200 Baud rate, and 1 start and stop bit, we created the following functions, based on the calculations in the datasheet:

```
void configure_serial_port() {
    // Configure UART pins, set 2-UART pin as primary function
    P1SEL0 |= BIT2 | BIT3;           // 2 = RX 3 = TX
    P1SEL1 &= ~(BIT2 | BIT3);

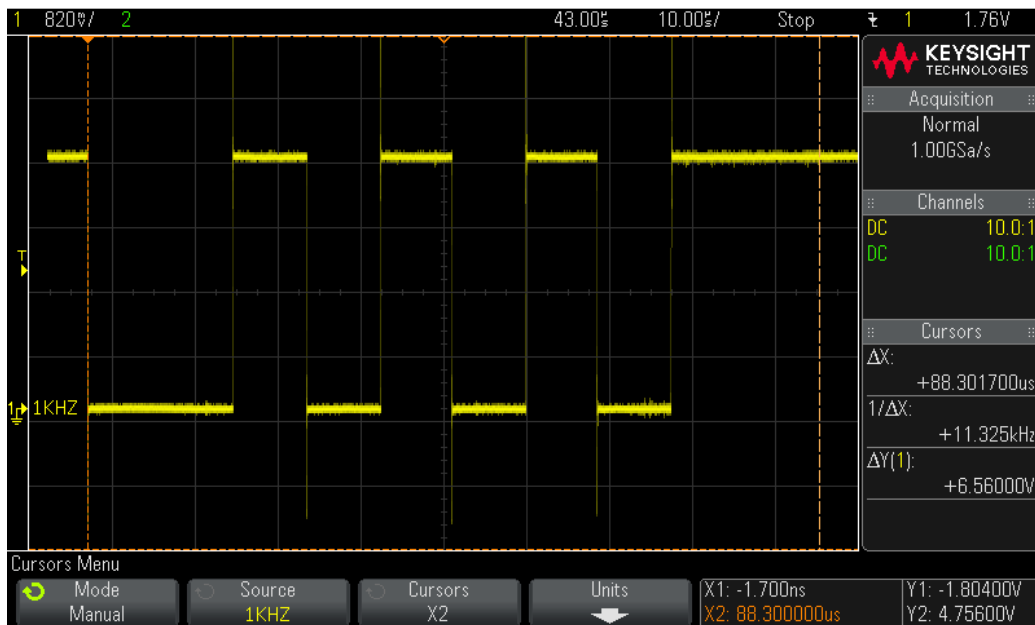
    // Configure UART
    UCA0CTLW0 |= UCSWRST;             // Put eUSCI in reset
    UCA0CTLW0 = EUSCI_A_CTLW0_SSEL__SMCLK | // Use SMCLK
                UCSWRST;
    UCA0BRW = 3;                     // Set clock scaler to 1
                                    // (6 MHz / 115200 / 16)
    UCA0MCTLW = 4 << EUSCI_A_MCTLW_BRF_OFS | // (3.26 - 3) * 16 = 4
                0x44 << EUSCI_A_MCTLW_BRS_OFS | // 0.63, Table 22-4 in family
                                    // data sheet
                EUSCI_A_MCTLW_OS16;         // Use modulator
    UCA0CTLW0 &= ~UCSWRST;           // Initialize eUSCI
    UCA0IE |= EUSCI_A_IE_RXIE;       // Enable USCI_A0 RX interrupt

    NVIC->ISER[0] = 1 << ((EUSCIA0_IRQn) & 31); // Enable eUSCIA0 interrupt in NVIC
}

void configure_clocks(void) {
    CS->KEY = 0x695A;                // Unlock CS module for register access
    CS->CTL0 = CS_CTL0_DCORSEL_1;    // Setup DCO Clock to 3 MHz (default)
    CS->CTL1 = CS_CTL1_SELA_2 |      // ACLK source set to REFOCLK
                CS_CTL1_SELS_3 |    // SMCLK set to DCOCLK
                CS_CTL1_SELM_3;     // MCLK set to DCOCLK
    CS->KEY = 0;                     // lock CS module
}
```

The DCO clock had to be set to 6 MHz to deal with a baud rate of 115200 along with data processing.

2. For the transmission of 0xAA, we measured 8.92 μ s per bit and 88.3 μ s for the entire frame. At our baud rate of 115200, each bit would ideally take 8.68 μ s, and the entire frame of 10 bits should take 86.8 μ s. Below is the oscilloscope plot of the transmission.



3. To iterate through an array of characters of a given length, we created the following function:

```
void uart_putchar_n(uint8_t *data, uint32_t length) {  
    // Increment up array, printing byte by byte until length is hit  
    while (length-->0) uart_putchar(*(data++));  
}
```

It iterates through length until it is set to negative, at which point the array has been printed.

4. The sentence "Why not Zoidberg?" took 1489.02 μ s to transmit, compared to the ideal 1388.89 μ s with an 115200 baud rate. Our transmission took an extra 100.13 μ s, which was most likely split into an extra 6.68 μ s between each character. Below is a screenshot of the sentence printing to the terminal.



5. An interrupt handler that would perform this function without the buffer would look like this:

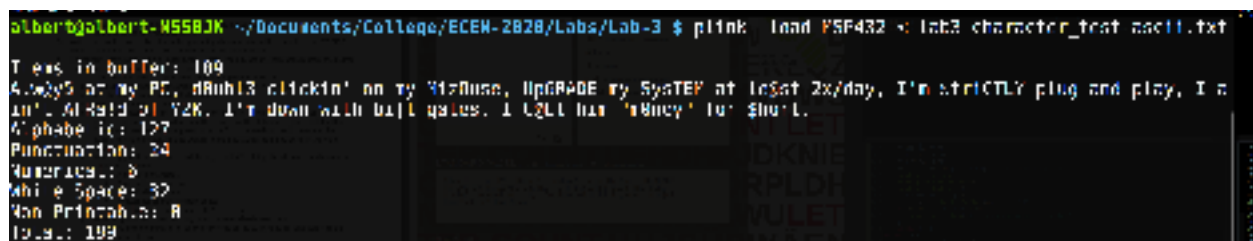
```
void eUSCIA0Handler(void) {
    uint8_t data = UCA0RXBUF;
    if (UCA0IFG & UCRXIFG) { // Triggered due to a receive.
        uart_putchar(data);
        UCA0IFG &= ~UCRXIFG;
    }
}
```

Since we implemented this with a circular buffer, our interrupt just pushes the received data onto the buffer and main constantly loops through it looking for chars to print. See the eUSCIA0Handler function for the interrupt handler implementation and main for the printing loop in main.c in the Appendix for the implementation that uses the buffer.

6. The buffer was implemented by creating a struct to hold the information about the array that represents the buffer in memory along with the head, tail, length, and num_items members. Functions use a bufError_T enum (defined in the header) to communicate if and what type of error occurred. Data is retrieved at the tail and added at the head, both of which wrap back around once they go past the top of the buffer in memory. See the code in the buffer_FIFO.h and buffer_FIFO.c sections in the appendix for the implementation of the circular buffer.

The RX interrupt is used to push data onto the buffer, check if the return key is hit or the buffer fills up, and analyze the data that comes in for statistics. By doing it in the interrupt, statistics can continue to accumulate even if the data does not entirely fit in the buffer. Statistics are handled by an inline function that checks for all classes of characters, and updates the proper entry in the stats struct. If either enter is hit or the buffer fills, it turns itself off, which acts as a signal to main() that the buffer must be printed and the interrupt re enabled afterwards. The left button clears the buffer and prints this operation to the terminal. The right button turns off the interrupt, prints the buffer and statistics, and reenables the interrupt. See main.c function port1Handler for the implementation of the button press interrupts and main for printing logic.

7. In order to receive and process all the characters in the file at 115200 baud, the clock speed of DC0 was increased to 6 MHz. This is the shot of the received data from the MSP432:



```
olbert@olbert-M55BJK ~/Documents/College/ECE428/Labs/Lab-3 $ plink -lmd MSP432 -r lab3 character_test test1.txt
Trans in buffer: 109
4.32% of my PC, dAnH13 clickin' on my V100use, UpGRADE my SySTeM at 10Sat 2x/day, I'm STRICTLY plug and play, I a
in' Al-Raid 91 Y2K. I'm down with bill gates, I ULL hit '98ney' for $hort.
Alphabe ic: 127
Punctuation: 24
Quotations: 0
While Spaces: 32
4th Printab.0: 8
10.0: 198
```

Conclusion

This lab served to introduce us to communication protocols, and, more specifically, to UART. We spent quite a while struggling with the setup of the protocol, especially with configuring the clocks. The datasheet's clock configuration chart helped us immensely, and once UART was configured, the rest of the lab was not as difficult. For the last question, the speed of the transmission led to a race condition in our program, where the buffer's `num_items` variable was not being properly incremented. We solved this issue by increasing the base frequency of the clock, so we had a few more clock cycles to add characters to the buffer. This lab was challenging, but the code provided in the lab overview seemed to lead us a bit astray. Once we went solely by the datasheet, setting up UART wasn't too hard, and using it was even easier.

Appendix

main.c:

```
#include "msp.h"
#include "buffer_FIFO.h"
#undef PB2
#undef PB4
#undef PB5
#undef PB6

CircBuf_T buffer; // The buffer that will hold received data
struct {
    uint32_t alpha;
    uint32_t punct;
    uint32_t numerical;
    uint32_t space;
    uint32_t noPrint;
} stats = {0};

void configure_clocks(void) {
    CS->KEY = 0x695A;           // Unlock CS module for register access
    CS->CTL0 = CS_CTL0_DCORSEL_2; // Setup DCO Clock to 6 MHz
    CS->CTL1 = CS_CTL1_SELA_2 |  // ACLK source set to REF0CLK
              CS_CTL1_SELS_3 |  // SMCLK set to DCOCLK
              CS_CTL1_SELM_3;   // MCLK set to DCOCLK
    CS->KEY = 0;                // lock CS module for register access
}

void configure_serial_port() {
    // Configure UART pins, set 2-UART pin as primary function
    P1SEL0 |= BIT2 | BIT3;           // 2 = RX 3 = TX
    P1SEL1 &= ~(BIT2 | BIT3);

    // Configure UART
    UCA0CTLW0 |= UCSWRST;           // Put eUSCI in reset
    UCA0CTLW0 = EUSCI_A_CTLW0_SSEL__SMCLK | // Use SMCLK
              UCSWRST;

    UCA0BRW = 3;                    // Set clock scaler to 3 (6 MHz / 115200 / 16)
    UCA0MCTLW = 4 << EUSCI_A_MCTLW_BRF_OFS | // (3.26 - 3) * 16 = 4
              0x44 << EUSCI_A_MCTLW_BRS_OFS | // 0.63, Table 22-4 in family data sheet
              EUSCI_A_MCTLW_OS16;           // Use modulator
    UCA0CTLW0 &= ~UCSWRST;           // Initialize eUSCI
    UCA0IE |= EUSCI_A_IE_RXIE;       // Enable USCI_A0 RX interrupt

    NVIC->ISER[0] = 1 << ((EUSCIA0_IRQn) & 31); // Enable eUSCIA0 interrupt in NVIC
}
```

```

void itoa(uint32_t num, char * string) {
    uint8_t i = 1;          // If number is zero, one char must still be transmitted: '0'
    uint8_t firstDigit;      // Used to diminish number of modulo operations
    uint32_t numCpy = num;   // Don't want to change num just yet

    // Since we are writing the number from right to left, the top of the stack must
    // contain the one's place. We do this by starting at the end. Therefore, we need to
    // know how long the output string will be:
    while (numCpy = (numCpy - (numCpy % 10)) / 10) i++; // i now holds index of '\0'
    string[i] = '\0';
    do {
        i--;
        firstDigit = num % 10;          // Get number in ones place
        string[i] = firstDigit + 0x30;  // 0x30 is ASCII offset to when digits start
        num -= firstDigit;              // Remove possibility of rounding issues
        num /= 10;                     // Shift numbers down one power of 10
    } while (i);
}

void uart_putchar(uint8_t tx_data) {
    P1OUT |= BIT0;                    // Turn on LED1 to indicate 'waiting to transmit'
    while (!(UCA0IFG & UCTXIFG));    // Block until transmitter is ready
    UCA0TXBUF = tx_data;              // Load data onto buffer
    P1OUT &= ~BIT0;                  // Turn off LED1 to indicate 'transmitting has started'
}

void uart_putchar_n(uint8_t * array, uint32_t length) {
    // Increment up array, printing byte by byte until length is hit
    while (length--) uart_putchar(*(array++));
}

void uart_putstring(char * string) {
    // Increment up string, printing char by char until '\0' is hit
    while (*string) uart_putchar(*(string++));
}

void uart_putenter(void) {
    uart_putchar('\n'); // Transmit a line break
    uart_putchar('\r');
}

void uart_putbuf(CircBuf_T *buf) {
    uint8_t item;
    char str[10];          // 2^32 is 10 chars long
    if (!buf) return;      // Woops! Null pointer!
    uart_putstring("Items in buffer: ");
    itoa(buf->num_items, str);
    uart_putstring(str);
    if (!bufferEmpty(buf)) { // Don't print newline if no items will be printed
        uart_putenter();
        while (removeItem(buf, &item) != BE_EMPTY) uart_putchar(item); // Print all items
    }
    uart_putenter();
}

```

```

void uart_putstats(void) {
    char str[10];
    uart_putstr("Alphabetic: ");
    itoa(stats.alpha, str);
    uart_putstr(str);
    uart_putenter();
    uart_putstr("Punctuation: ");
    itoa(stats.punct, str);
    uart_putstr(str);
    uart_putenter();
    uart_putstr("Numerical: ");
    itoa(stats.numerical, str);
    uart_putstr(str);
    uart_putenter();
    uart_putstr("White Space: ");
    itoa(stats.space, str);
    uart_putstr(str);
    uart_putenter();
    uart_putstr("Non Printable: ");
    itoa(stats.noPrint, str);
    uart_putstr(str);
    uart_putenter();
    uart_putstr("Total: ");
    itoa(stats.alpha + stats.numerical + stats.punct + stats.space + stats.noPrint, str);
    uart_putstr(str);
    uart_putenter();
}

inline void updateStats(uint8_t data) {
    // Check if data is printable
    if (data >= '!' && data <= '~') {
        // Alphabetic
        if (data >= 'A' && data <= 'Z' || data >= 'a' && data <= 'z') stats.alpha++;
        // Numeric
        else if (data >= '0' && data <= '9') stats.numerical++;
        // Punctuation
        else stats.punct++;
    }
    // Check if data is whitespace
    else if ((data >= 0x9 && data <= 0xD) || data == ' ') stats.space++;
    // Else, it is not printable
    else stats.noPrint++;
}

```



```

void eUSCIA0Handler(void) {
    uint8_t data;
    if (UCA0IFG & UCRXIFG) {          // Triggered due to a receive.
        P2OUT |= BIT1;
        data = UCA0RXBUF;
        // Need to print buffer if we receive 'enter' or the buffer is full
        if (data == '\r' || addItem(&buffer, data) == BE_FULL) {
#ifdef PB6
            // Turn off interrupt to indicate to main that printing must be done
            UCA0IE &= ~EUSCI_A_IE_RXIE; // Signals to main to print
#endif
        }
        updateStats(data);
        UCA0IFG &= ~UCRXIFG;
        P2OUT &= ~BIT1;
    }
}

void port1Handler(void) {
    UCA0IE &= ~EUSCI_A_IE_RXIE;      // Turn off RX interrupt
    if (P1IFG & BIT1) {              // left
        clearBuffer(&buffer);        // Clear buffer and stats
        stats.alpha = 0;
        stats.numerical = 0;
        stats.punct = 0;
        stats.space = 0;
        stats.noPrint = 0;
        uart_putenter();
        uart_putstring("Buffer cleared");
        uart_putenter();
    }
    if (P1IFG & BIT4) {              // right
        uart_putenter();
        uart_putbuf(&buffer);        // Print buffer and stats (clears buffer)
        uart_putstats();
        uart_putenter();
    }
    UCA0IE |= EUSCI_A_IE_RXIE;
    P1IFG = 0;
}

void configure_buttons(void) {
    P1DIR &= ~(BIT1 | BIT4);        // Buttons S1/S2 set to input
    P1REN |= BIT1 | BIT4;          // Enable pullup/down resistors
    P1OUT |= BIT1 | BIT4;          // Set to pullup mode
    P1IFG = 0;                     // Clear the interrupt Flag
    P1IES |= BIT1 | BIT4;          // Interrupt fires on high to low transition
    P1IE |= BIT1 | BIT4;           // Enable interrupt for buttons
    NVIC_EnableIRQ(PORT1_IRQn);    // Register port 1 interrupts with NVIC
}

```

```

void main(void) {
    uint8_t data;
    bufError_T error;
    P1DIR |= BIT0;           // Green and red LED used to indicate activity
    P2DIR |= BIT1;
    P1OUT &= ~BIT0;
    P2OUT &= ~BIT1;
    WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
    configure_clocks();
    configure_serial_port();
    configure_buttons();
    __enable_interrupt();
    error = initializeBuffer(&buffer, 256);
    if (error == BE_LOW_MEM) {
        uart_putstr("Not enough memory for buffer!");
    }
    while(1) {
#ifdef PB2
        uart_putchar(0xAA);
#elif defined PB4
        uart_putstr("Why not Zoidberg?");
#elif defined PB5
        if (removeItem(&buffer, &data) != BE_EMPTY) uart_putchar(data);
#elif defined PB6
        // If the receive interrupt is off an enter was read or the buffer is full
        if (!(UCA0IE & EUSCI_A_IE_RXIE)) {
            uart_putenter();
            uart_putbuf(&buffer);
            uart_putstats();
            uart_putenter();
            UCA0IE |= EUSCI_A_IE_RXIE;
        }
#endif
    }
}

```

buffer_FIFO.h:

```
#include <stdlib.h>
#include <stdint.h>
#ifndef BUFFER_FIFO
#define BUFFER_FIFO

typedef enum {          // BE = Buffer Error
    BE_EMPTY = -4,      // A 'remove' was requested, but there are no more items
    BE_FULL = -3,       // An 'add' was requested, but there is no more space
    BE_NULL_PTR = -2,   // A null pointer was given
    BE_LOW_MEM = -1,    // The requested size of buffer is too large for the heap
    BE_NO_ERR = 0       // All went well
} bufError_T;

typedef struct CircBuf{
    uint8_t *head;
    uint8_t *tail;
    uint32_t num_items;
    uint32_t length;
    uint8_t *buffer;
} CircBuf_T;

bufError_T initializeBuffer(CircBuf_T *buf, uint32_t length);
void clearBuffer(CircBuf_T *buf);
void deleteBuffer(CircBuf_T *buf);
int8_t bufferFull(CircBuf_T *buf);
int8_t bufferEmpty(CircBuf_T *buf);
bufError_T addItem(CircBuf_T *buf, uint8_t item);
bufError_T removeItem(CircBuf_T *buf, uint8_t *val);

#endif
```

buffer_FIFO.h:

```
#include "buffer_FIFO.h"

bufError_T initializeBuffer(CircBuf_T *buf, uint32_t length){
    if (!buf) return BE_NULL_PTR;
    uint8_t *buffer = (uint8_t *) malloc(length);
    if (!buffer) return BE_LOW_MEM;
    buf->head = buffer;
    buf->tail = buffer;
    buf->num_items = 0;
    buf->length= length;
    buf->buffer = buffer;
    return BE_NO_ERR;
}

void clearBuffer(CircBuf_T *buf){
    buf->head = buf->buffer;
    buf->tail = buf->buffer;
    buf->num_items = 0;
}

void deleteBuffer(CircBuf_T *buf){
    free(buf->buffer);
}

int8_t bufferFull(CircBuf_T *buf){
    return buf->num_items >= buf->length;
}

int8_t bufferEmpty(CircBuf_T *buf){
    return !buf->num_items;
}

bufError_T addItem(CircBuf_T *buf, uint8_t item){
    if (!buf) return BE_NULL_PTR;
    if (bufferFull(buf)) return BE_FULL;
    *(buf->head) = item;
    buf->head = buf->buffer + ((buf->head - buf->buffer) + 1) % buf->length;
    buf->num_items++;
    return BE_NO_ERR;
}

bufError_T removeItem(CircBuf_T *buf, uint8_t *val){
    if (!buf || !val) return BE_NULL_PTR;
    if (bufferEmpty(buf)) return BE_EMPTY;
    *val = *(buf->tail);
    buf->tail = buf->buffer + ((buf->tail - buf->buffer) + 1) % buf->length;
    buf->num_items--;
    return BE_NO_ERR;
}
```