ECE 09.342: Milestone 1: Stranger Things UART PWM LED control

Matt Delengowski & Stephen Szymczak
Rowan University

October 16, 2017

1 Abstract

After learning the basics of how the MSP430 processors work, we were tasked with writing code and setting up an RGB LED node that is to be controlled via a stream of bytes utilizing UART as means of communication between the nodes and the master. The node is meant to be daisy-chained to other nodes and a master node (raspberry pi), and display like the Christmas lights in the show Stranger Things. This was successfully achieved by putting to use our knowledge of PWM, UART, C programming, and the MSP430 FR6989.

2 Introduction

Universal asynchronous receiver-transmitter (UART) is a hardware device for asynchronous serial communication. UART is (for this lab) part of an integrated circuit (IC) that allows serial communications over a computer or peripheral device serial port. For example, a micro-controller with UART capabilities can be made to interface with Blue-tooth. This lab utilizes UART and the MSP430FR6989 to make an ”Addressable” RGB LED.

There will be a master node and 26 (one for letter of the alphabet) slave nodes, the slave nodes will be connected in series within one another. For example the TX (transmit) port of node 2 will be connected to the RX (receive) port of node 3. The master node will send an 80 byte string that goes to the first node. The first node is to read the first byte of the string, and determine how many bytes are to be processed. The next three bytes are to be sent to capture/compare timer registers (CCR) to allow for PWM of an RGB LED. So for example, if the first node is to receive 80 bytes, the first byte says it is to receive 80 bytes. The second value being represented by the second byte goes to the CCR register that controls the duty cycle of the red LED. The third byte goes to CCR of the green LED, and the forth byte goes the CCR of the blue LED. After the forth byte is received, the remaining 76 bytes must be processed to be

transmitted. The amount of bytes that are to be transmitted should be 77, with the leading byte holding the hex value of 77 to indicate to the next node how many bytes it is to receive. So, if node[x] is in front of node[y] and node[x] is to receive n bytes, then node[y] is to receive (n-3) bytes.

3 Background

3.1 UART

The hardware that makes UART work is shift registers. UART works through the use of 2 pins, one for transmission (TX) and one for receiving (RX). Data is handled in bytes, each time a full byte (referred to as character in the FR6989 data sheet) is received, the UCRXIFG interrupt flag is set. However, how can a full byte be sent through 1 pin at once? It cannot. This is where shift registers come into play. The bits that make up each individual byte are shifted serially in a shift register. Once all 8 bits have been shifted in, the 8 bits are sent out at once in parallel. So, the RX pin utilizes what is known as a serial in parallel out shift register. Conversely, the TX pin transmits an entire byte at once into a shift register and the shift register then shifts out serially for the next device to receive serially. So, the TX pin utilizes a parallel in serial out shift register. The use of these shift registers is important, as it reduces costs, and board space. For each bit that is taken in parallel, an extra pin is required and this drives up costs and size.

Baud rate is the second important topic of UART. Baud rate is the amount of bits transfered per second. If the baud rate between two devices is not the same, error in transmission is to occur. This is because two communicating UARTS do not have a shared timing system. This lab requires a baud rate of 9600. Starting from there if we want a clock rate of 8Mhz then looking at Table 30-5. Recommended Settings for Typical Crystals and Baud Rates of slau367n then the modulation register (UCAxMCTLW) should be set such that UCBRSx equals 0x49, set UCBRF1, and have UCOS16 set high. This will minimize error in transmission and receiving.

Lastly, how the RX and TX interrupts work is vitally important. The data sheet states that the UART receive interrupt flag is set each time a character is received and loaded into the UCAxRXBUF (RX register) register. This implies that the port is constantly on the look to receive a character. Once the character has been received, the flag is set and the UCA_RX ISR is entered. Likewise, the data sheet states that the UART transmit interrupt flag is set by the transmitter to indicate that UCAxTXBUF (the TX register) is ready to accept another character. This was interpreted to mean that once a single byte is transmitted through the TX port, when another byte can be transmitted, the flag will be set and the UCATX ISR will be entered again automatically. Implying that the transmission must be started manually, probably somewhere in the UCARX ISR, once a receiving cycle has been completed.

Table 30-5. Recommended Settings for Typical Crystals and Baud Rates[1]

BRCLK	Baud Rate	UCOS16	UCBRx	UCBRFx	UCBRSx[2]	TX Error[2] (%)		RX Error[2] (%)	
						neg	pos	neg	pos
32768	1200	1	1	11	0x25	-2.29	2.25	-2.56	5.35
32768	2400	0	13	-	0xB6	-3.12	3.91	-5.52	8.84
32768	4800	0	6	-	0xEE	-7.62	8.98	-21	10.25
32768	9600	0	3	-	0x92	-17.19	16.02	-23.24	37.3
1000000	9600	1	6	8	0x20	-0.48	0.64	-1.04	1.04
1000000	19200	1	3	4	0x2	-0.8	0.96	-1.84	1.84
1000000	38400	1	1	10	0x0	0	1.76	0	3.44
1000000	57600	0	17	-	0x4A	-2.72	2.56	-3.76	7.28
1000000	115200	0	8	-	0xD6	-7.36	5.6	-17.04	6.96
1048576	9600	1	6	13	0x22	-0.46	0.42	-0.48	1.23
1048576	19200	1	3	6	0xAD	-0.88	0.83	-2.36	1.18
1048576	38400	1	1	11	0x25	-2.29	2.25	-2.56	5.35
1048576	57600	0	18	-	0x11	-2	3.37	-5.31	5.55
1048576	115200	0	9	-	0x08	-5.37	4.49	-5.93	14.92
4000000	9600	1	26	0	0xB6	-0.08	0.16	-0.28	0.2
4000000	19200	1	13	0	0x84	-0.32	0.32	-0.64	0.48
4000000	38400	1	6	8	0x20	-0.48	0.64	-1.04	1.04
4000000	57600	1	4	5	0x55	-0.8	0.64	-1.12	1.76
4000000	115200	1	2	2	0xBB	-1.44	1.28	-3.92	1.68
4000000	230400	0	17	-	0x4A	-2.72	2.56	-3.76	7.28
4194304	9600	1	27	4	0xFB	-0.11	0.1	-0.33	0
4194304	19200	1	13	10	0x55	-0.21	0.21	-0.55	0.33
4194304	38400	1	6	13	0x22	-0.46	0.42	-0.48	1.23
4194304	57600	1	4	8	0xEE	-0.75	0.74	-2	0.87
4194304	115200	1	2	4	0x92	-1.62	1.37	-3.56	2.06
4194304	230400	0	18	-	0x11	-2	3.37	-5.31	5.55
8000000	9600	1	52	1	0x49	-0.08	0.04	-0.1	0.14
8000000	19200	1	26	0	0xB6	-0.08	0.16	-0.28	0.2
8000000	38400	1	13	0	0x84	-0.32	0.32	-0.64	0.48
8000000	57600	1	8	10	0xF7	-0.32	0.32	-1	0.36
8000000	115200	1	4	5	0x55	-0.8	0.64	-1.12	1.76
8000000	230400	1	2	2	0xBB	-1.44	1.28	-3.92	1.68
8000000	460800	0	17	-	0x4A	-2.72	2.56	-3.76	7.28
8388608	9600	1	54	9	0xEE	-0.06	0.06	-0.11	0.13
8388608	19200	1	27	4	0xFB	-0.11	0.1	-0.33	0
8388608	38400	1	13	10	0x55	-0.21	0.21	-0.55	0.33
8388608	57600	1	9	1	0xB5	-0.31	0.31	-0.53	0.78
8388608	115200	1	4	8	0xEE	-0.75	0.74	-2	0.87

(1) The listed UCBRSx settings are determined by a search algorithm for the lowest error. Other settings for UCBRSx might result in similar or same errors.
(2) Assumes a stable clock source for BRCLK with negligible jitter (for example, from a crystal oscillator). Any frequency variation or jitter of the clock source will make the errors worse.

Figure 1: Modulation Register Settings Given a Baud and Clock Rate

3.2 Pulse Width Modulation

Pulse Width Modulation (PWM) is the second half of the milestone. PWM is as the name implies, manipulation of the width of a pulse. This width of a pulse is known as duty cycle and can be described as the fraction of a signals period for which it is high. Concisely, the higher the duty cycle, the longer the signal is on in one period. This is mathematically defined as

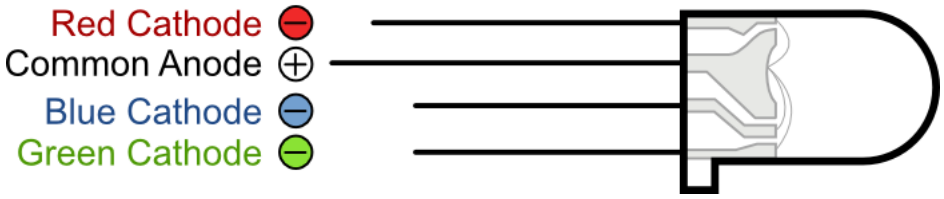
$$Duty\ Cycle = \frac{Time\ On}{Time\ Off} \tag{1}$$

The manipulation of the duty cycle will be implemented using CCR registers of the timer peripherals. Specifics will be discussed in the implementation section.

3.3 RGB LED

RGB LEDs are either common anode or common cathode. This can be seen by holding the LED up to a light and looking at the metal filaments inside. Ours is common

anode, therefore all the LEDs share a common power supply. This requires the RGB pins to connected to ground for them to turn on. So, to modify the duty cycle is to modify the amount of time that they are connected to ground.



It is good practice to place a resistor between each cathode and ground (in our case ground are the specific GPIO pins for PWM). This protects the LED from burning out and protects the microcontroller from sinking too much current through the GPIO. However there are trade offs to this, if the resistor used is too large then the LED will be very dim. If too small of a resistor is used then something could be damaged. To determine the resistor value requires to know how much current the microcontroller will sink through the GPIO pin, the turn on voltage for each LED, and the voltage being supplied to the common anode. In our case, 3.3 volts was supplied to the anode, and respectively the turn on voltages for the RGB LED are 1.8v, 2.8v, and 2.8v. The current sink was found to be 2mA in the datasheet, therefore using Ohm's law resistances are calculated to be

$$R_{red} = \frac{(3.3 - 1.8)v}{2mA} = 750\Omega \tag{2}$$

$$R_{green,blue} = \frac{(3.3 - 2.8)v}{2mA} = 250\Omega \tag{3}$$

Using these resistances the LED was deemed bright enough. If a brighter LED was required, a BC337 NPN amplifier transistor could be used for each cathode pin, connecting the base to each respective GPIO, each respective LED cathode to the source and each emitter to ground.

4 Implementation

In order to implement our LED node, the requirements for functionality were broken down into two parts: PWM and UART & control.

Previously, in lab 4, software PWM and hardware PWM were both explored for our host of MSP430 processors. Knowing the way each board handled PWM, we chose to use the FR6989 for it's multitude of capture compare registers in Timer B that could be connected directly to GPIO pins in a Hardware PWM fashion. This is an important feature because three CCRs were needed to handle the PWM of each pin on the LED (red cathode pin, green cathode pin, and blue cathode pin).

UART communication was required to receive data from the previous node, process it, and transmit modified data to the next node. Since testing was done using RealTerm on a PC, and a micro-USB connected to the emulator side of the board, the emulator RX and TX pins were used. Additionally, USCI interrupts were used to handle all of the communication and preparation of the data stream for the next node.

4.1 PWM Implementation

Hardware PWM implementation was chosen due to the requirements of low power mode, implying polling was out of the question. This was aided by fact of it's code simplicity, merely setting the Selection Registers. We used the TB0.x option select for pins P2.4, P2.5, and P2.6 to send PWM signals to our RGB LED. These specific pins take TB0.3, TB0.4, and TB0.5 respectively:

```
1  void GPIOSetup()
2  {
3      // Configure GPIO
4    P3SEL0 |= BIT4 | BIT5;                  // USCI_A1 UART operation
5    P3SEL1 &= ~(BIT4 | BIT5);
6
7    P2SEL0 |= BIT4;                         // Direct TB0CCR3 (RED LED) to
       P2.4
8    P2SEL1 &= ~BIT4;
9
10   P2SEL0 |= BIT5;                         // Direct TB0CCR4 (GREEN LED)
       to P2.5
11   P2SEL1 &= ~BIT5;
12
13   P2SEL0 |= BIT6;                         // Direct TB0CCR5 (BLUE LED)
       to P2.6
14   P2SEL1 &= ~BIT6;
15
16   P2DIR |= (BIT4 + BIT5 + BIT6);          // Set P2.4,P2.5,P2.6 to
       Output
17 }
18
19 int main(void)
20 {
21 .
22 .
23 .
24     __bis_SR_register(LPM3_bits | GIE);       // Enter LPM3,
       interrupts enabled
25     __no_operation();                         // For debugger
26 }
```

CCR3,CCR4, and CCR5 were initialized to zero so that our LED would start off. Additionally, each of the CCRs' control registers had their OUTMOD_3 bits set high. OUTMOD_3 sets the signal of TB0.x high when CCRx is reached, and resets the

signal to low when CCR0 is reached:

```
1
2  void TimerB0Setup()
3  {
4      TB0CCR0 = 256 - 1;                          // Count Up To
5      TB0CTL |= TBSSEL_2 + MC_1 + TBCLR;          // SMCLK (1MHz) Up Mode
       Clear Timer
6      TB0CCR3 = 0;                                // RED LED DUTY CYCLE...
       Changed Set In UART RX ISR
7      TB0CCR4 = 0;                                // GREEN LED DUTY CYCLE
       ...Changed Set In UART RX ISR
8      TB0CCR5 = 0;                                // BLUE LED DUTY CYCLE
       ...Changed Set In UART RX ISR
9      TB0CCTL3 |= OUTMOD_3;                       // Set/Reset Mode
10     TB0CCTL4 |= OUTMOD_3;                       // Set/Reset Mode
11     TB0CCTL5 |= OUTMOD_3;                       // Set/Reset Mode
12 }
```

The use of OUTMOD_3 is vital due to the RGB LEDs being common anode. When the output of the CCR registers is low, the LEDs will be on. When the output is high the voltage is equal to that of the anode (3.3v) and there is no power. If the RBG LED were to be common cathode then OUTMOD_7 would have been required.

Additionally, CCR0 was set to 255 since this number is equivalent to 100% of two bytes being set high; as the max hex value that can be sent is 0xFF = 255. Timer B0 is initialized with SMCLK and Up Mode, and the timer is cleared.

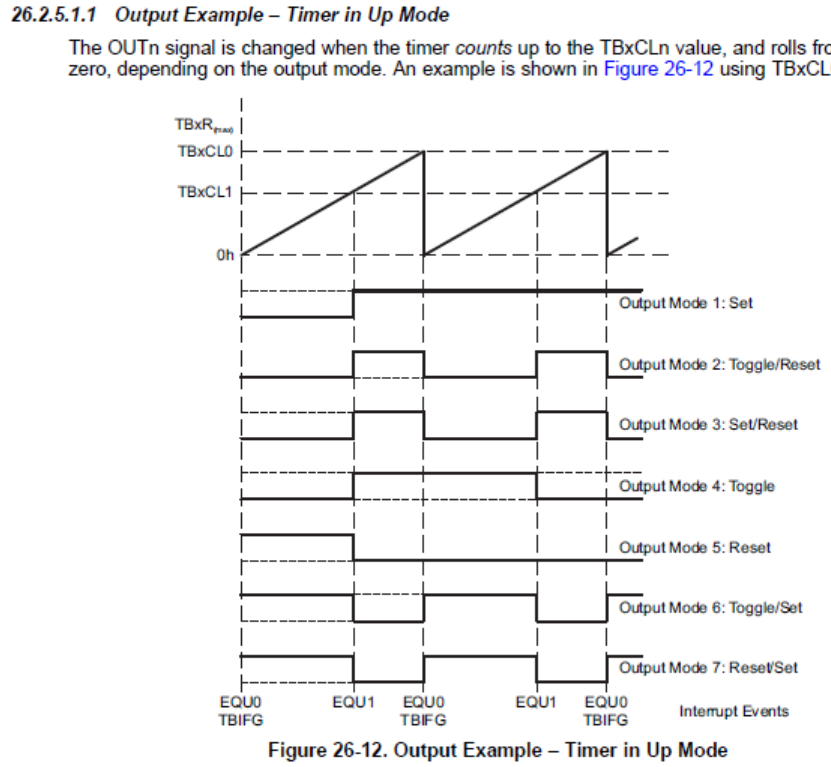


Figure 2: Graphic of effect of OUTMOD_x on CCRx registers

With the above code, our pins are receiving a duty cycle equivalent to

$$\frac{CCRx \times 100\%}{255} \tag{4}$$

Since CCR3, CCR4, and CCR5 directly control the PWM, they were given easier names to follow in the code. Additionally, two UART related LEDs, and registers were defined here as well:

```
#define red    TB0CCR3                  // For Controlling
    RED LED DUTY CYCLE
#define green TB0CCR4                   // For Controlling
    GREEN LED DUTY CYCLE
#define blue   TB0CCR5                  // For Controlling
    BLUE LED DUTY CYCLE
#define LED1ON P1OUT |= BIT0            // Indicates Start of
     Receive Cylce (RX)
#define LED1OFF P1OUT &= ~BIT0          // Indicates End of
    Receive Cycle (RX)
```

```
#define LED2ON P9OUT |= BIT7                    // Indicates Start of
     Transmission Cycle (TX)
#define LED2OFF P9OUT &= ~BIT7                  // Indicates End of
    Transmission Cycle (TX)
#define RECEIVED UCA1RXBUF                      // Byte Received
    Buffer
#define TRANSMIT UCA1TXBUF                      // Transmit Byte
    Buffer
#define EN_REC_INTERRUPT UCA1IE |= UCRXIE       // Enable UCA1 RX
    Interrupt
#define DIS_REC_INTERRUPT UCA1IE &= ~UCRXIE     // Disable UCA1 RX
    Interrupt
#define EN_TRAN_INTERRUPT UCA1IE |= UCTXIE      // Enable UCA1 TX
    Interrupt
#define DIS_TRAN_INTERRUPT UCA1IE &= ~UCTXIE    // Disable UCA1 TX
    Interrupt
```

These values: red, green, and blue are changed in the UART portion of the code in order to change the PWM of the RGB LED based on UART input. The rest of the code falls under UART and control.

4.2 UART and Control

Now that the PWM code is set up to interface with our RGB LED, it needs to be given proper input to change the color of the LED. This input comes from the stream of bytes that enter the processor through a UART RX pin. Since there is only the RX pin to input data to the processor, the UART communication must be set up in a way that it receives bytes at the same rate that the connected device sends them. This is done by determining the proper settings for the modulation register UCAxMCTLW, which was discussed in detail in the background section.

```
void UARTSetup()
{
    // Startup clock system with max DCO setting ~8MHz
  CSCTL0_H = CSKEY >> 8;                        // Unlock clock registers
  CSCTL1 = DCOFSEL_3 | DCORSEL;                 // Set DCO to 8MHz
  CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
  CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;         // Set all dividers
  CSCTL0_H = 0;                                 // Lock CS registers

  // Configure USCI_A1 for UART mode
  UCA1CTLW0 = UCSWRST;                          // Put eUSCI in reset
  UCA1CTLW0 |= UCSSEL__SMCLK;                   // CLK = SMCLK
  // Baud Rate calculation
  // 8000000/(16*9600) = 52.083
  // Fractional portion = 0.083
  // User's Guide Table 21-4: UCBRSx = 0x04
  // UCBRFx = int ( (52.083-52)*16) = 1
  UCA1BR0 = 52;                                 // 8000000/16/9600
```

```
19    UCA1BR1 = 0x00;
20    UCA1MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
21    UCA1CTLW0 &= ~UCSWRST;                    // Initialize eUSCI
22    UCA1IE |= UCRXIE;                         // Enable USCI_A1 RX
        interrupt
23
24    FRCTL0 = FWPW | NWAITS_0;                 // Removes Warning #10421-
        D, FWPW is FRAM password, NWAITS_0 adds 0 wait states to FRAM,
        changes nothing, simply removes warning
25  }
```

Given the interpretation of the UART transmit and receive interrupt operations, and design problem, the following algorithm was designed:

Start: Initialize Timer B, UART, GPIO, and go LPM

Wait: Receive character input through RX

Next: Store first byte and store in variable to determine total bytes N

Next: Receive byte 2 and store in CCR3

Next: Receive byte 3 and store in CCR4

Next: Receive byte 4 and store in CCR5

Next: Create array, store the value N - 3 in index 0 of array

Next: Receive byte 5 store in index 1

Next: Repeat for bytes 6 through N, storing each byte Z in index Z - 4. Array should have indexes zero through N - 4.

Next: Once finished receiving N bytes, disable RX interrupts, and enable TX interrupts, transmit first byte, index 0, of array through TX to next node.

Next: Transmit array index 1, repeat for indexes 2 through N - 4. Should be N - 3 transmits total.

Next: When byte N has been transmitted, index N - 4 of array, disable TX interrupts and enable RX interrupts.

Wait: Receive character input through RX

.
.
.

Our implementation that realizes this algorithm

```
#pragma vector=USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
    switch(__even_in_range(UCA1IV, USCI_UART_UCTXCPTIFG))
    {
        case USCI_NONE:
        {
            break;
        }
        case USCI_UART_UCRXIFG:
        {
            LED1ON;
      // Indicate We Are Receiving Bytes (Wont See Unless Using
    Breaks)
            if(byte_count == 0)
      // Store First Byte Received, Has Number of Bytes to be
    Received
            {
                num_of_bytes = RECEIVED;
                byte_count++;
            }
            else if((byte_count > 0) && (byte_count <= 3))
            {
                switch(byte_count)
                {
                    case 1:
                    {
                        red = RECEIVED;
      // Store RED LED Byte...should be 2nd byte, byte_count = 1
                        break;
                    }
                    case 2:
                    {
                        green = RECEIVED;
      // Store GREEN LED Byte...should be 3rd byte, byte_count = 2
                        break;
                    }
                    case 3:
                    {
                        blue = RECEIVED;
      // Store BLUE LED Byte...should be 4th byte, byte_count = 3
                        break;
                    }
                }
                byte_count++;
            }
            else if((byte_count <= num_of_bytes) && (byte_count > 3))
      // Bytes to be Transmitted
```

```
41              {
42                  if(byte_count == 4)
43                  {
44                     index_count = 0;
45                     messages[index_count] = num_of_bytes - 3;
       // Number of Bytes to eventually Transmit
46                  }
47                  index_count++;
48                  messages[index_count] = RECEIVED;
       // Storing RGB Bytes to be Transmitted
49                  byte_count++;
50                  if(byte_count == num_of_bytes)
51                  {
52                      index_count = 0;
           // Reset Array Index to Start Transmitting
53                      byte_count = 0;
           // Reset byte_count for Next Cycle of Receiving
54                      DIS_REC_INTERRUPT;
           // Disable USCI_A1 RX Interrupt
55                      EN_TRAN_INTERRUPT;
           // Enable USCI_A1 TX Interrupt
56                      TRANSMIT = messages[index_count];
           // Send First Transmission (Indicates Number of Bytes
    Being Sent)
57                      LED1OFF;
           // Indicate We have stopped receiving bytes
58                  }
59              }
60              __no_operation();
61              break;
62          }
63          case USCI_UART_UCTXIFG:
64          {
65              LED2ON;
       // Indicate Transmission Started (Wont See Unless Using Breaks
    )
66              if(index_count < (messages[0] - 1))
67              {
68                  index_count++;
69                  TRANSMIT = messages[index_count];
       // Start Transmitting Remaining Bytes
70              }
71              else if(index_count  == (messages[0] - 1))
72              {
73                  LED2OFF;
       // Indicate Transmission Ended
74                  index_count = 0;
       // Reset Index for Next Receive Cycle
75                  num_of_bytes = 0;
       // Reset For Next Receive Cycle
```

```
76                  EN_REC_INTERRUPT;
        // Enable USCI_A1 RX Interrupt
77                  DIS_TRAN_INTERRUPT;
        // Disable USCI_A1 TX Interrupt
78                  }
79                  break;
80              }
81              case USCI_UART_UCSTTIFG:
82              {
83                  break;
84              }
85              case USCI_UART_UCTXCPTIFG:
86              {
87                  break;
88              }
89      }
90  }
```

5 Testing

To test our implementation, we connected an RGB LED to a breadboard with appropriate resistor values and jumpers to the 6989 GPIO. To communicate with the board we utilized the comm port emulator of the ez-FET expansion board (top portion of 6989) and use the software RealTerm to send and receive hex strings.

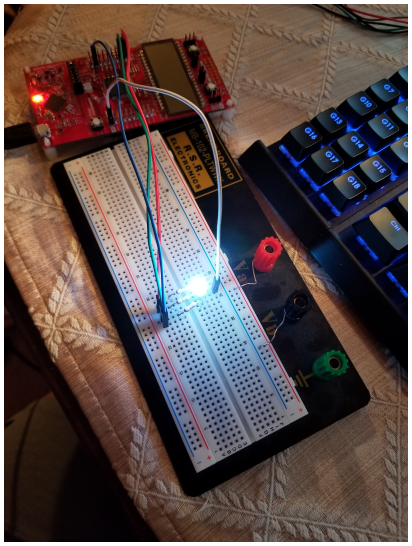


Figure 3: Breadboard Setup: In between each GPIO pin and it's respective cathode, there is a resistor in series and the anode is connected to the onboard 3.3V

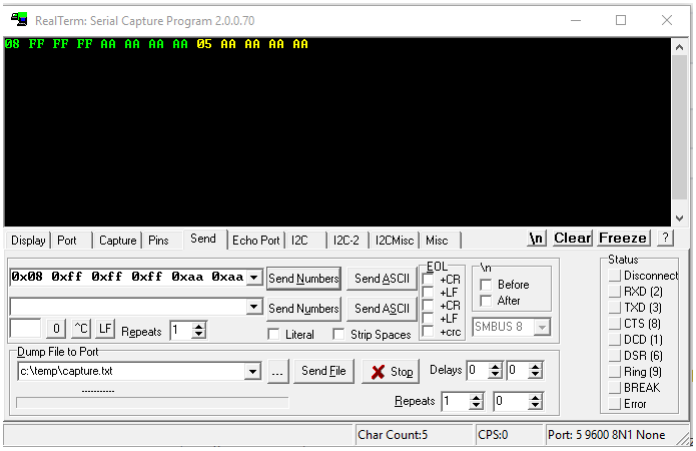


Figure 4: Communicating: Input String (Green) Output String (Yellow)

To test we sent the string (green) of hex values (0x08 0xff 0xff 0xff 0xaa 0xaa 0xaa 0xaa), so we sent 8 bytes to be received by the 6989. The first byte of the string indicates this, and the following three are the duty cycle settings for the red,green, and blue LEDs. A value of 0xff indicates 100 percent duty cycle, meaning the each LED will be on fully. Looking at the yellow string the first byte is 0x05, indicating the 6989 transmitted 5 bytes (as to be expected, N = 8, therefore 8 - 3 = 5 bytes needed to be transmitted). This test reveals a successful implementation. To create the network, all that must be done is merely connect a jumper from ez-FET TX pin to the RX pin on the following node, and connect a jumper from the ez-FET RX pin to the TX pin of the previous node (possible master node).

5.1 RealTerm Settings

The following settings were used for real term. To determine what comm port your device is (in windows), go to device manager and open up Ports section. The device will be listed as MSP Application UART1 (COMx) where x is the port number.

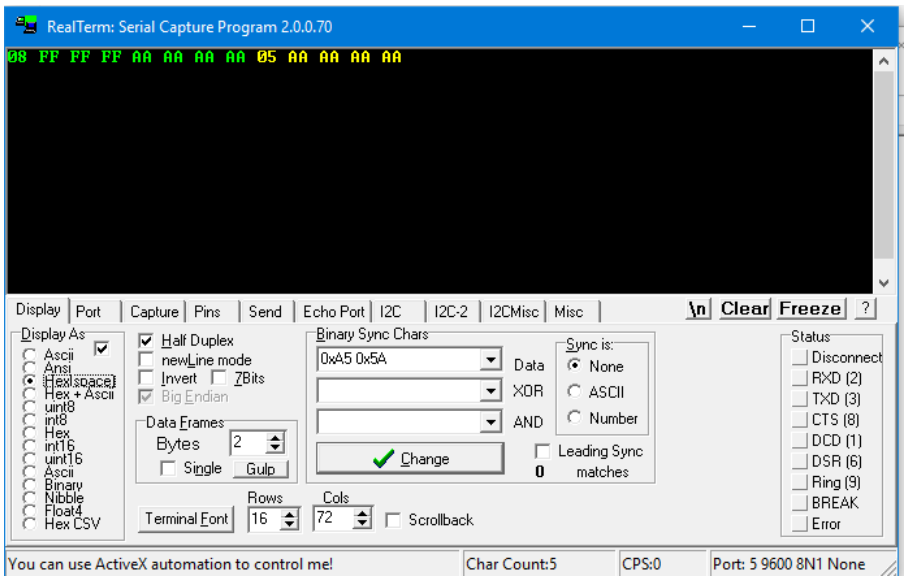


Figure 5: Display Settings For RealTerm

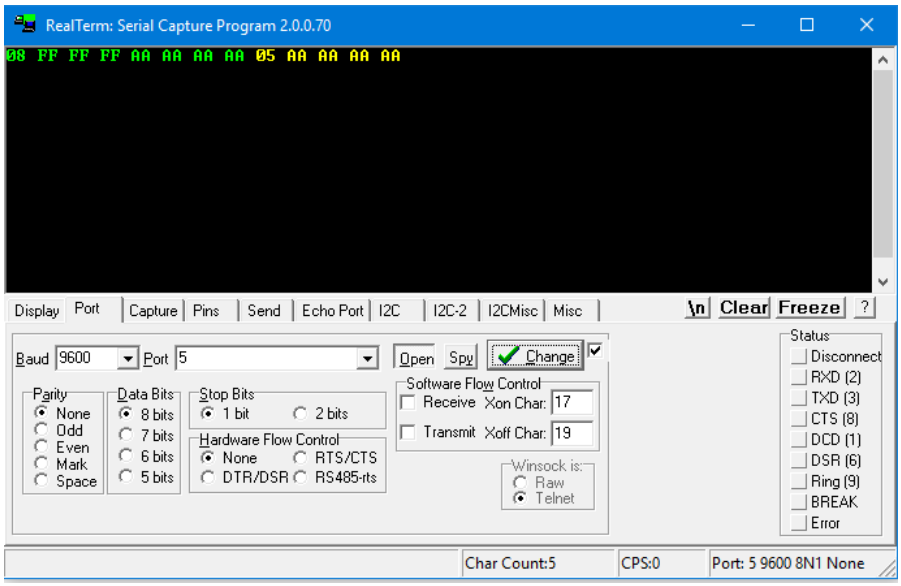


Figure 6: Port Settings For RealTerm

6 Full Code

```
1  //MILESTONE 1
2  //Authors: The Polish Brotherhood: Delengowski & Szymczak
3  //Device: MSP430FR6989
4
5
6  #include <msp430.h>
7
8  volatile unsigned int num_of_bytes;          // Takes on the Value of
       Received Byte Zero (when byte_count = 0),
9                                               // Stores Total Number
       of Bytes to be Received
10 volatile unsigned int byte_count = 0;        // For Counting Number
       of Bytes Received through RXBUF
11 volatile unsigned int index_count = 0;       // For Assigning
       Received Bytes to messages[i] Array
12 volatile unsigned int messages[80];          // For Storing Received
       Bytes from RXBUF...at most we will receive 80 bytes,
13                                              // Later Used for
       Transmitting Messages to TXBUF
14
15 #define red    TB0CCR3                          // For Controlling
       RED LED DUTY CYCLE
16 #define green TB0CCR4                          // For Controlling
       GREEN LED DUTY CYCLE
17 #define blue   TB0CCR5                         // For Controlling
       BLUE LED DUTY CYCLE
18 #define LED1ON P1OUT |= BIT0                   // Indicates Start of
        Receive Cylce
19 #define LED1OFF P1OUT &= ~BIT0                 // Indicates End of
       Receive Cycle
20 #define LED2ON P9OUT |= BIT7                   // Indicates Start of
        Transmission Cycle
21 #define LED2OFF P9OUT &= ~BIT7                 // Indicates End of
       Transmission Cycle
22 #define RECEIVED UCA1RXBUF                     // Byte Received
       Buffer
23 #define TRANSMIT UCA1TXBUF                     // Transmit Byte
       Buffer
24 #define EN_REC_INTERRUPT UCA1IE |= UCRXIE      // Enable UCA1 RX
       Interrupt
25 #define DIS_REC_INTERRUPT UCA1IE &= ~UCRXIE    // Disable UCA1 RX
       Interrupt
26 #define EN_TRAN_INTERRUPT UCA1IE |= UCTXIE     // Enable UCA1 TX
       Interrupt
27 #define DIS_TRAN_INTERRUPT UCA1IE &= ~UCTXIE   // Disable UCA1 TX
       Interrupt
28
29 void TimerB0Setup();
30 void UARTSetup();
31 void GPIOSetup();
```

```
32 void LEDSetup();
33
34 int main(void)
35 {
36     WDTCTL = WDTPW | WDTHOLD;                    // Stop Watchdog
37     PM5CTL0 &= ~LOCKLPM5;                        // Disable the GPIO
    power-on default high-impedance mode to activate,
38                                                  // previously
    configured port settings
39
40     TimerB0Setup();
41     UARTSetup();
42     GPIOSetup();
43     LEDSetup();
44
45     __bis_SR_register(LPM3_bits | GIE);          // Enter LPM3,
    interrupts enabled
46     __no_operation();                            // For debugger
47
48 }
49
50 #pragma vector=USCI_A1_VECTOR
51 __interrupt void USCI_A1_ISR(void)
52 {
53     switch(__even_in_range(UCA1IV, USCI_UART_UCTXCPTIFG))
54     {
55         case USCI_NONE:
56         {
57             break;
58         }
59         case USCI_UART_UCRXIFG:
60         {
61             LED1ON;
       // Indicate We Are Receiving Bytes (Wont See Unless Using
    Breaks)
62             if(byte_count == 0)
       // Store First Byte Received, Has Number of Bytes to be
    Received
63             {
64                 num_of_bytes = RECEIVED;
65                 byte_count++;
66             }
67             else if((byte_count > 0) && (byte_count <= 3))
68             {
69                 switch(byte_count)
70                 {
71                     case 1:
72                     {
73                         red = RECEIVED;
       // Store RED LED Byte...should be 2nd byte, byte_count = 1
```

```
74                      break;
75                  }
76                  case 2:
77                  {
78                      green = RECEIVED;
    // Store GREEN LED Byte...should be 3rd byte, byte_count = 2
79                      break;
80                  }
81                  case 3:
82                  {
83                      blue = RECEIVED;
    // Store BLUE LED Byte...should be 4th byte, byte_count = 3
84                      break;
85                  }
86              }
87              byte_count++;
88          }
89          else if((byte_count <= num_of_bytes) && (byte_count > 3))
    // Bytes to be Transmitted
90          {
91              if(byte_count == 4)
92              {
93                 index_count = 0;
94                 messages[index_count] = num_of_bytes - 3;
    // Number of Bytes to eventually Transmit
95              }
96              index_count++;
97              messages[index_count] = RECEIVED;
    // Storing RGB Bytes to be Transmitted
98              byte_count++;
99              if(byte_count == num_of_bytes)
100             {
101                 index_count = 0;
        // Reset Array Index to Start Transmitting
102                 byte_count = 0;
        // Reset byte_count for Next Cycle of Receiving
103                 DIS_REC_INTERRUPT;
        // Disable USCI_A1 RX Interrupt
104                 EN_TRAN_INTERRUPT;
        // Enable USCI_A1 TX Interrupt
105                 TRANSMIT = messages[index_count];
        // Send First Transmission (Indicates Number of Bytes
  Being Sent)
106                 LED1OFF;
        // Indicate We have stopped receiving bytes
107             }
108         }
109         __no_operation();
110         break;
111     }
```

```
112         case USCI_UART_UCTXIFG:
113         {
114             LED2ON;
        // Indicate Transmission Started (Wont See Unless Using Breaks
    )
115             if(index_count < (messages[0] - 1))
116             {
117                 index_count++;
118                 TRANSMIT = messages[index_count];
        // Start Transmitting Remaining Bytes
119             }
120             else if(index_count  == (messages[0] - 1))
121             {
122                 LED2OFF;
        // Indicate Transmission Ended
123                 index_count = 0;
        // Reset Index for Next Receive Cycle
124                 num_of_bytes = 0;
        // Reset For Next Receive Cycle
125                 EN_REC_INTERRUPT;
        // Enable USCI_A1 RX Interrupt
126                 DIS_TRAN_INTERRUPT;
        // Disable USCI_A1 TX Interrupt
127             }
128             break;
129         }
130         case USCI_UART_UCSTTIFG:
131         {
132             break;
133         }
134         case USCI_UART_UCTXCPTIFG:
135         {
136             break;
137         }
138     }
139 }
140
141 void TimerB0Setup()
142 {
143     TB0CCR0 = 256 - 1;                          // Count Up To
144     TB0CTL |= TBSSEL_2 + MC_1 + TBCLR;          // SMCLK (1MHz) Up Mode
    Clear Timer
145     TB0CCR3 = 0;                                // RED LED DUTY CYCLE...
    Changed Set In UART RX ISR
146     TB0CCR4 = 0;                                // GREEN LED DUTY CYCLE
    ...Changed Set In UART RX ISR
147     TB0CCR5 = 0;                                // BLUE LED DUTY CYCLE
    ...Changed Set In UART RX ISR
148     TB0CCTL3 |= OUTMOD_3;                       // Set/Reset Mode
149     TB0CCTL4 |= OUTMOD_3;                       // Set/Reset Mode
```

```
150     TB0CCTL5 |= OUTMOD_3;                     // Set/Reset Mode
151 }
152
153 void UARTSetup()
154 {
155     // Startup clock system with max DCO setting ~8MHz
156   CSCTL0_H = CSKEY >> 8;                      // Unlock clock registers
157   CSCTL1 = DCOFSEL_3 | DCORSEL;               // Set DCO to 8MHz
158   CSCTL2 = SELA__VLOCLK | SELS__DCOCLK | SELM__DCOCLK;
159   CSCTL3 = DIVA__1 | DIVS__1 | DIVM__1;       // Set all dividers
160   CSCTL0_H = 0;                               // Lock CS registers
161
162   // Configure USCI_A1 for UART mode
163   UCA1CTLW0 = UCSWRST;                        // Put eUSCI in reset
164   UCA1CTLW0 |= UCSSEL__SMCLK;                 // CLK = SMCLK
165   // Baud Rate calculation
166   // 8000000/(16*9600) = 52.083
167   // Fractional portion = 0.083
168   // User's Guide Table 21-4: UCBRSx = 0x04
169   // UCBRFx = int ( (52.083-52)*16) = 1
170   UCA1BR0 = 52;                               // 8000000/16/9600
171   UCA1BR1 = 0x00;
172   UCA1MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
173   UCA1CTLW0 &= ~UCSWRST;                      // Initialize eUSCI
174   UCA1IE |= UCRXIE;                           // Enable USCI_A1 RX
        interrupt
175
176   FRCTL0 = FWPW | NWAITS_0;                  // Removes Warning #10421-
        D, FWPW is FRAM password, NWAITS_0 adds 0 wait states to FRAM,
        changes nothing, simply removes warning
177 }
178
179 void GPIOSetup()
180 {
181     // Configure GPIO
182   P3SEL0 |= BIT4 | BIT5;                      // USCI_A1 UART operation
183   P3SEL1 &= ~(BIT4 | BIT5);
184
185   P2SEL0 |= BIT4;                             // Direct TB0CCR3 (RED
        LED) to P2.4
186   P2SEL1 &= ~BIT4;
187
188   P2SEL0 |= BIT5;                             // Direct TB0CCR4 (GREEN
        LED) to P2.5
189   P2SEL1 &= ~BIT5;
190
191   P2SEL0 |= BIT6;                             // Direct TB0CCR5 (BLUE
        LED) to P2.6
192   P2SEL1 &= ~BIT6;
193
```

```
194   P2DIR |= (BIT4 + BIT5 + BIT6);                  // Set P2.4,P2.5,P2.6 to
          Output
195 }
196
197 void LEDSetup()
198 {
199     P1DIR |= BIT0;                                // Set P1.0 LED to Output
200     P9DIR |= BIT7;                                // Set P9.7 LED to Output
201
202     P1OUT &= ~BIT0;                               // Disable P1.0 LED
203     P9OUT &= ~BIT7;                               // Disable P9.7 LED
204 }
```