Milestone 1: Stranger Things Light Wall

Jacob Epifano & Lonnie Souder II
Rowan University

October 16, 2017

1 Abstract

This lab aims to take all the skills learned in labs one through four and apply it to one project. The goal is to control RGB LEDs through varying PWM signals that can be specified through a message recieved over UART which can be passed along to multiple microprocessors. In the end we should be able to have several microprocessors daisy chained together controlled by a single UART message.

2 Introduction and Background

To complete this lab, students will need to understand UART, GPIO, Timers, and PWM. A message will be sent over UART. the message will be structured as follows:

1. The first byte is the number of bytes in the message including this byte. (Note: This limits the message to 255 bytes.)
2. The next n bytes will correspond to a duty cycle to assign to an LED. The color order is Red, Green, Blue repeating.
3. The last bit is a carriage return (0x0D).

Each node in the chain should be able to read the bytes that pertain to it and pass the remaining message to the next node. This means that as the message goes down the line, it should get shorter. When the message reaches the last node, the output should *always* be the same: 0x02 0x0D.

3 Evaluation and Results

3.1 Microprocessor Selection

Initially we set out to use the MSP430G2553 as the processor for this project, but we soon realized, for our implementation, we needed 3 capture/compare registers on one timer. The MSP430G2553 only had 2 capture/compare modules on a timer. The processor that did meet this requirement was the MSP430F5229. We had issues getting the UART example code to work. The board would not echo back anything that was sent to it. We then decided to change to the MSP430FR5994, since this board met our primary requirements (UART peripheral, hardware PWM, 3 capture/compare modules on a single timer) and was close enough in price to the other 2 available chips that also met our requirements, we decided to move forward with the MSP430FR5994.

3.2 LED PWM

The code for controlling the duty cycles for the LED is extremely similar to Lab 4's hardware PWM. Timer B was set to SMCLK and set to Up mode in order to generate a 1 kHz signal. The Timer B module was selected on 3 pins to correspond with 3 capture/compare modules in Set/Reset mode. When the timer is equal to one of the three capture compare registers the bit is set high, and when the timer reaches CCR0 and overflows the GPIO pin is reset, thus generating a PWM signal for which the duty cycle can be modified by changing CCRx (where x is not 0).

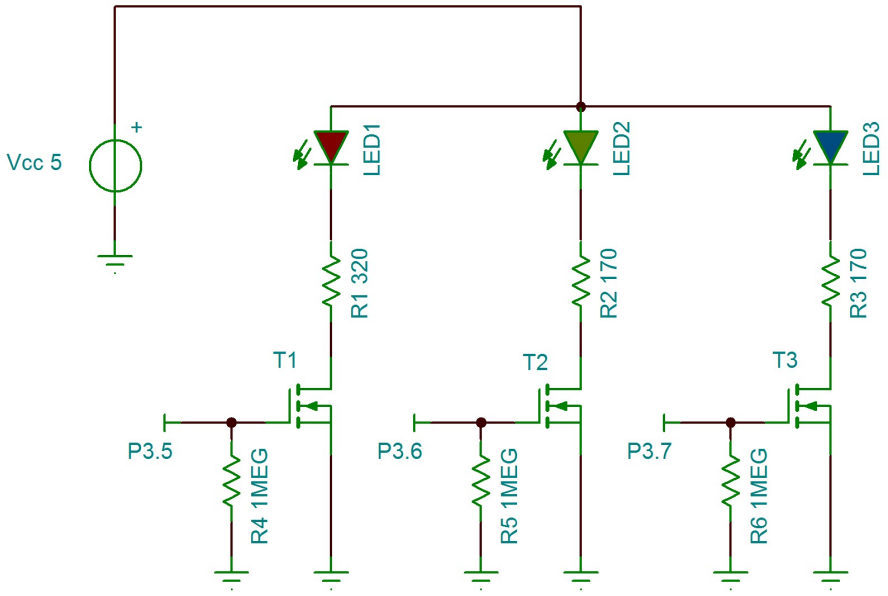
```
// Initialization of GPIO and timer registers for PWM on 3 pins

#define RED BIT5
#define GREEN BIT6
#define BLUE BIT7
#define BUTTON BIT5

PM5CTL0 &= ~LOCKLPM5;         // Enable GPIO
P3SEL0 |= RED + GREEN + BLUE;
P3DIR |= RED + GREEN + BLUE;
P3OUT &= ~(RED + GREEN + BLUE);

TB0CTL |=   TBSSEL_2 |ID_2 | MC_1;
TB0CCR0 = 255;              // Set clock period

TB0CCR4 = 1000;
TB0CCTL4 |= OUTMOD_3;       // Set/Reset mode

TB0CCR5 = 250;
TB0CCTL5 |= OUTMOD_3;       // Set/Reset mode
```

```
TB0CCR6 = 0;
TB0CCTL6 |= OUTMOD_3;        // Set/Reset mode
```

In the above code, 3 GPIO pins are set as outputs. They are initialized at 0. The Timer B source is set to SMCLK with a pre-divider of 2 in Up mode. CCR0 is set to 1000, meaning it will count to 1000 and then reset. CCR4 is for the red LED and is set to 1000. Its duty cycle will be 0%, because as soon as it gets set, it will reset. CCR5 is for the green LED and is set to 250. At 250 it will be set, and then reset at 1000, this means its duty cycle is 75%. CCR6 is the blue LED and is set at 0. This will make its duty cycle 100%.

3.3 LED Circuit

Through trial and error we figured out our RGB LED was common anode. We then designed the following circuit to control the brightness of all three colors simultaneously.



The 5 V is the V_{cc} from the MSP430FR5994 chassis. It gives power to the LEDs. The transistors will act as a switch, letting current flow if the GPIO pins go HIGH. You will notice that R_1 is different from R_2 and R_3. This is because the typical voltage drop across a red LED is about 1.8 V while the typical voltage drop across a blue or green LED is roughly 3.3 V. The resistors are chosen to achieve a current of 10 mA with a supply voltage of 5 V. Another point of interest is the 1 MΩresistor going from the gate

of each MOSFET to ground. This is simply a pull down resistor put in place so that the MOSFET is never stuck in a state of HIGH. When the pin goes low, current can flow to ground through the pull down resistor so that the charge on the pin is actually dissipated.

3.4 UART Implementation

The UART implementation has two parts: the setup and the communication protocol. The setup basically sets all the necessary registers to enable UART for our application. As shown in the code below, the setup stage selects and sets up a clock source. Next the baud rate is configured. All of the numbers need for this can be taken directly from TI's datasheet in which there is a table which lists the numbers to set each register to in order to achieve a specific baud rate with a given clock frequency. The chart is complete with a list of timing errors. Lastly, the UART receive interrupt is enabled.

```
// Startup clock system with max DCO setting ~8MHz
CSCTL0_H = CSKEY_H;                     // Unlock CS 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_A0 for UART mode
UCA0CTLW0 = UCSWRST;                    // Put eUSCI in reset
UCA0CTLW0 |= UCSSEL__SMCLK;             // CLK = SMCLK
// The steps to set up the baud rate
// can be taken directly from the data sheet
// Here baud rate is set to 9600
UCA0BRW = 52;
UCA0MCTLW |= UCOS16 | UCBRF_1 | 0x4900;
UCA0CTLW0 &= ~UCSWRST;                  // Initialize eUSCI
UCA0IE |= UCRXIE;                       // Enable RX interrupt
```

More important, is the actual communication protocol. As explained previously, the node should expect a message with 3 parts: how many total bytes are in the message, duty cycle information, and an end of message byte. Out of all of this, 3 bytes are actually intended for out node. The rest will be passed along to the next node which should do the same thing with the message. The handling of the message is done entirely in the RX ISR. The basic course of action is as follows. Upon receiving a message, the processor begins counting how many bytes it has received. The first byte gets stored so that the processor knows exactly how long the message should be. We will be using 3 bytes from this message, so we subtract 3 from the first byte received and then pass it along to the next node. the next 3 bytes will be used to set the brightness of red, green, and blue. Because, we set up our timer to overflow at 255, the byte received should correspond directly to number we need to set the capture/compare registers to in order to achieve the desired duty cycle. The issue is,

if we just set CCRx to RXBUF, we would get the inverse of the duty cycle we wanted. For instance, if CCRx was set to 0x00, our duty cycle would be 100%, so we set our capture/compare registers to 255-RXBUF which gives us the proper duty cycle. For these 3 bytes, it is key that we **don't** pass these on to the next node, because they are used only to set the color of our LED. For every byte after our 3, we simply pass them along to the next node. When we finally get to the end of message (which we will know because we know how long the message should be), we reset the byte count in preparation for the next message. The logic portion of this is shown in the code below. Note that the code here is not the full implementation. It is merely a significant portion of the code relevant to this application note.

```
while(!(UCA0IFG & UCTXIFG));
    if (RxCount == 0 ) {
        UCA0TXBUF = UCA0RXBUF - 3;
        nBytes = UCA0RXBUF;
        RxCount ++;
    }
    else if (RxCount == 1) {        // Red info
        RxCount ++;
        TB0CCR4 = 255-UCA0RXBUF;
    }
    else if (RxCount == 2) {        // Green info
        RxCount ++;
        TB0CCR5 = 255-UCA0RXBUF;
    }
    else if (RxCount == 3) {        // Blue info
        RxCount ++;
        TB0CCR6 = 255-UCA0RXBUF;
    }
    else if (RxCount > 2) {        // Pass these along
        UCA0TXBUF = UCA0RXBUF;
        RxCount ++;
        if (RxCount == nBytes)        // EOM
            RxCount = 0;
    }
```

4 Conclusions

With the UART communication and PWM LED control combined, the ending result should be a system in which many nodes can be chained together in series to handle a single message which controls an entire array of RGB LEDs. While the application here is fairly simple, the concepts of UART used to pass messages from one node in a system to the next along with a general understanding of GPIO and PWM can be used for many more practical things.