Modular RGB LED

Sean McGuire, Alex Frederick
Introduction To Embedded Fall 2017

October 17, 2017

1 Overview

This application note will cover the process of designing a system to control an RGB LED using pulse width modulation (PWM). This Application Note (AN) covers the process of selecting the optimal development board, designing the physical circuit with MOSFET's, and writing the code to adjust the color of the LED. When completed this module is able to behave independently, or as a unit with other modules to form a series of controllable LED's.

Contents

2 Background

The principles of pulse width modulation, or PWM, is to adjust the duty cycle, percentage of time a signal is held high. This change in duty cycle will adjust the average voltage across the output, in this case an LED on either red, green, or blue, thus changing the brightness. This is achieved by utilizing timers located on the MSP430F5529 to count upwards to a value of 255 and resetting to 0, repeating this process indefinitely. This could is done at a frequency of 255 kHz meaning the LEDs each flash at a rate of 1 kHz or having a period of 1 ms. In order to take advantage of varying brightnesses for the LEDs, the duty cycle must be determined somewhere on the count from 0 to 255 by assigning where the signal turns back to its low state. With LEDs brightness being proportional to the current through them, as the duty cycle of the driving signal is increased, the average current increases and therefore illuminates the LED at a higher intensity.

3 Development Board Selection

The first step in designing an embedded system is deciding which parts you will utilize.The initial thought process was to use the most basic board in order to make coding easier which would have been the MSP430G2553. However, upon review of the capabilities of the board it was determined to be unsuitable with a lack of Capture and compare registers to be utilized. This would have been insufficient to control the pulse width of the LEDs. In this case the MSP430F5529 was selected (figure 1). One of the reasons this board was select was because this board has UART capabilities on pins 3.3 and 3.4. Another reason this board has been selected is that it has the ability to be tested using software and hardware UART. This development board also has 4 timers, and timerA0 has 5 capture and compare registers making it suitable for this task.

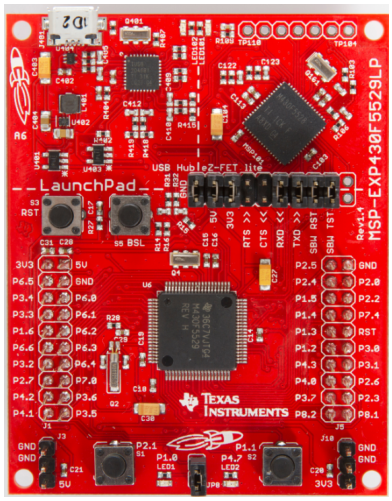

Figure 1: Selected Development Board (MSP430F5529)

4 Controlling the RGB LED

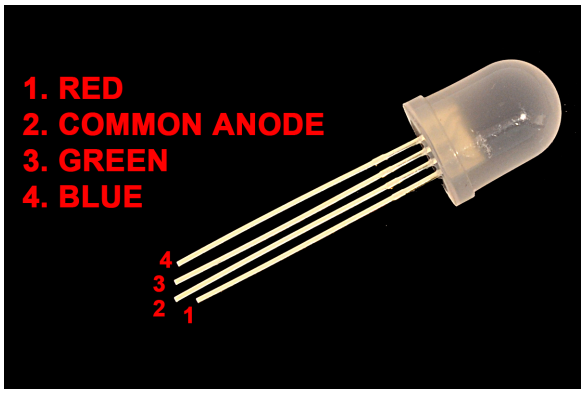

Figure 2: Diagram of LED

The RGB LED's that are utilized for this lab are common anode. This means that the anode is a single pin which is held at a constant voltage. To control each diode individually the cathodes must be manipulated. This is done by using N-type MOSFETS to act as switches on the cathode side of each leg of the LED. Another design choice which has been implemented is the addition of pull down resistors to the gates of the MOSFETS. This is done because without these resistors charge builds up on the gate which changes the color of the LED.

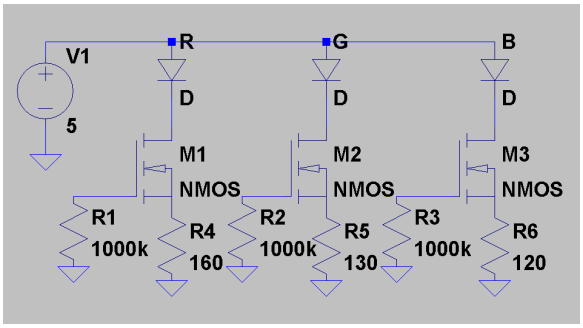

Figure 3: Schematic of LED

5 Software Implementation

5.1 UART

The UART communications between the PC and MSP430 was tested using both Putty and RealTerm. Using Putty we were able to test the example UART communication code which transmits every byte that is received creating an echoing effect. After this was completed and the code was finished we moved on to a full test of the system. Using RealTerm we were able to send multiple bytes at once to see what the MSP430 would transmit in response. While testing the UART end of the system it was found that the MSP430 responded with the number of bytes in the message, and the incoming message with the three bits which the module has analyzed removed from the sequence.

5.2 Timers

On this assignment the TimerA module of the MSP430F5529 was utilized. Inside of TimerA 4 CCR registers were utilized. CCR0 was used to control the reset of TimerA which is in up mode. The CCR0 register was set to 0x00FF so that the timer would count to 255 and reset which allows one to easily implement PWM with 255 steps. CCR1 was designated to control the pulse width modulation of the red LED, CCR2 was designated to control the green LED and CCR3 was designated to control the blue LED.

6 Explaining the code

The code used to create this effect consists of four main sections, the initialization of the UART communication, the initialization of the pins to be used for PWM, setting up the timer capture and compare registers, and the interrupt code for received bits.

The first block of text after initializing variables and stopping the watchdog timer sets up UART communications by first setting pins P3.3 and P3.4 to be used for transmission and receiving. Then the sub-main clock is selected as the timing source and the baud rate, or speed of communication, is set to 9600 with the UCA0BR0/BR1 bits being set to 6 and 0 respectively. The UART control register is then set for proper modulation. Finally, the state machine controlling the communication is initialized and the interrupt for received communications is set.

Then the pins used to control the LEDs, P1.2, P1.3, and P1.4, are set to be used as outputs and are selected to use the timer controlled outputs from TA0.1/2/3. This allows for us to use timer A0 to both count upwards to 255 and set each LED to a different value for its duty cycle. Using out mode seven on all three Capture/Compare registers means that the pins are all set to high when the count reaches 255 and they are set to low when their respective value is reached.

Finally the interrupt code works in such a way that it works through a state machine to perform various functions to the bits received. The first byte taken into the processor is supposed to list the number of bytes in the string. This number is saved as the BitRX variable, then that value less three is sent to the transmission buffer to be sent to the next node. Then the count increases changing states so that the next value is stored in CCR1 to control the duty cycle of the red LED. The count increases again and the next two bits are stored in CCR2 and CCR3 for the green and blue led duty cycle values. The final stage transmits each bit received until the count is of bytes in equals the value of the first byte meaning the end of the message. From there the count is reset to zero, bringing the system back to its ready and waiting position where the next message can be received.

7 Appendix

```
#include <msp430.h>

 // Initial Greeting you should see upon properly connecting your Launchpad
int i = 0;
volatile int count = 0;
volatile int BitRX = 0;
int main(void)
{

    WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT


    P3SEL = BIT3+BIT4;                        // P3.4,5 = USCI_A0 TXD/RXD
    UCA0CTL1 |= UCSWRST;                      // **Put state machine in reset**
    UCA0CTL1 |= UCSSEL_2;                     // SMCLK
    UCA0BR0 = 6;                              // 1MHz 9600 (see User's Guide)
    UCA0BR1 = 0;                              // 1MHz 9600
    UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;   // Modln UCBRSx=0, UCBRFx=0,
                                              // over sampling
    UCA0CTL1 &= ~UCSWRST;                     // **Initialize USCI state machine**
    UCA0IE |= UCRXIE;                         // Enable USCI_A0 RX interrupt

    //Initialize LEDs for pwm
    P1DIR |= 0x0E;  //LEDs set as output
    P1SEL |= 0x0E;  //LEDs set to TA0.1/2/3

    //Set up CCRs for clock count and pwm control
    TA0CCR0  = 0x00FF; //After timerA counts to 255 reset
    TA0CCTL1 = OUTMOD_7; //Put Capture Control 1 in set and reset mode
    TA0CCR1 = 0; //Initialize Capture Control Register 1 to 0
    TA0CCTL2 = OUTMOD_7; //Put Capture Control 2 in set and reset mode
    TA0CCR2  = 0; //Initialize Capture Control Register 2 to 0
    TA0CCTL3 = OUTMOD_7; //Put Capture Control 3 in set and reset mode
    TA0CCR3 = 0; //Initialize Capture Control Register 3 to 0
    TA0CTL = TASSEL_2 + MC_1 + ID_3; //TimerA uses SMCLK, with divider of 8,
     in count up mode
    __bis_SR_register(LPM0_bits + GIE);       // Enter LPM0, interrupts enabled
    __no_operation();                         // For debugger


}

// Echo back RXed character, confirm TX buffer is ready first
```

```
// Interupt vector for UART communication
#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(USCI_A0_VECTOR))) USCI_A0_ISR (void)
#else
#error Compiler not supported!
#endif
{
    switch(__even_in_range(UCA0IV,USCI_UCTXIFG)){

        case USCI_NONE: break;

        case USCI_UCRXIFG:
//Switch through the count which shows which byte is being recieved
            switch (count){

                case 0:
//While the transmit buffer is still in use wait
                    while(!(UCA0IFG & UCTXIFG));
//Set the temporary varriable BitRx to the recived buffer
                    BitRX = UCA0RXBUF;
//Subract 3 from the recived byte and transmit it (this is the
                     number of byte to expect for the next module)
                    UCA0TXBUF = UCA0RXBUF - 0x0003;
                    __no_operation();
                    break;

                case 1:
//Sets CCR1 to the 2nd byte recieved
                    TA0CCR1 = (UCA0RXBUF);
                    break;

                case 2 :
//Sets CCR2 to the 3rd byte recieved
                    TA0CCR2 = (UCA0RXBUF);
                    break;

                case 3 :
//Sets CCR3 to the 4th byte recived
                    TA0CCR3 = (UCA0RXBUF);
                     break;

                default:
//If past the first 4 bytes then transmit the byte recieved once transmit buffer is cl
```

```
                        while(!(UCA0IFG & UCTXIFG));
                        UCA0TXBUF = UCA0RXBUF;
                    }
            //If not at the end of the message then increment count
                    if(count != BitRX - 1){
                        count += 1;
                     } else if (count == BitRX - 1){
             //At the end of the message reset count in anticipation for the next message
                         count = 0;

                     }
                        break;
                     case USCI_UCTXIFG : break;
                     default: break;
                        }
                    }
```