*Application Note*

*16 October 2017*

# *UART controlled RGB LEDs for MSP430*

Skylar Adams & Stephen Glass
Electrical & Computer Engineering, Rowan University

## 1. Abstract

Embedded systems can be implemented to interface with devices using a variety of protocols and send signals to discrete components. The MSP430 microcontroller can be used to interface with a serial terminal using UART protocol. The serial communication can transmit signals to the MSP430 embedded processor and perform hardware PWM to address a RGB LED. For this application note, a node-based system is used so multiple embedded processors can be linked together to control many LEDs simultaneously.

## 2. Introduction

An embedded system is computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts.

In a scene from Netflix's "Stranger Things", Will Byers is stuck in a parallel dimension and can only communicate to his mother by causing Christmas Lights to light up. The mother then strings up a set of lights like a Ouija Board so they can communicate. This scene can be recreated by lighting up RGB LEDs controlled by a MSP430 microprocessor.

For this application note, we will discuss an entire network of devices [nodes] connected with the purpose of addressing RGB LEDs.

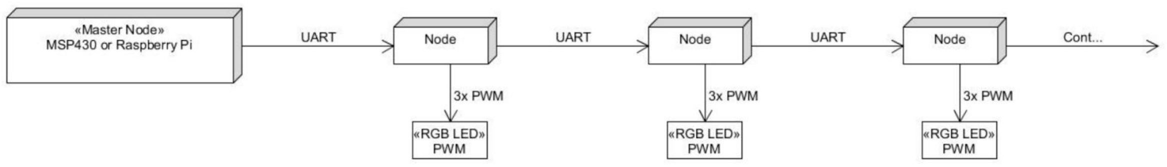


*Figure 1*

Each these nodes will be responsible for taking in a string of Hex values over the UART RX Line, using the three least significant bytes as their own RGB values. These RGB values will then be converted and transformed into duty cycles to replicate the proper color. The incoming string may be at most 80 bytes, the remaining string after removing the three least significant bytes needs to be transmitted over the UART TX line to the next node.

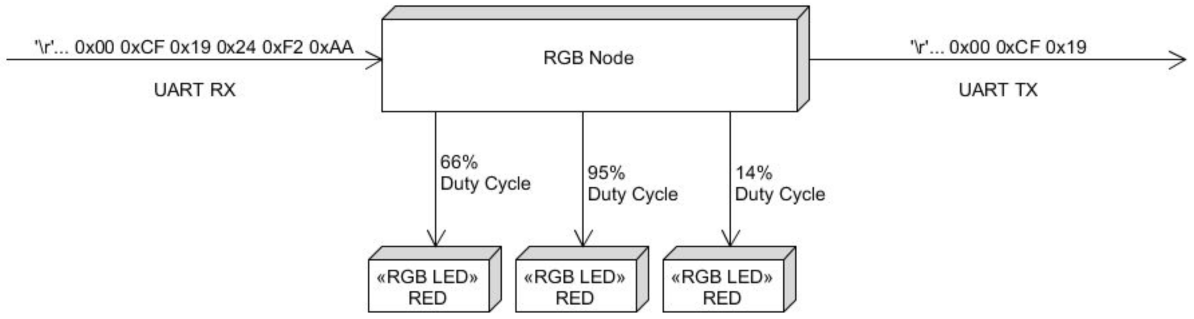
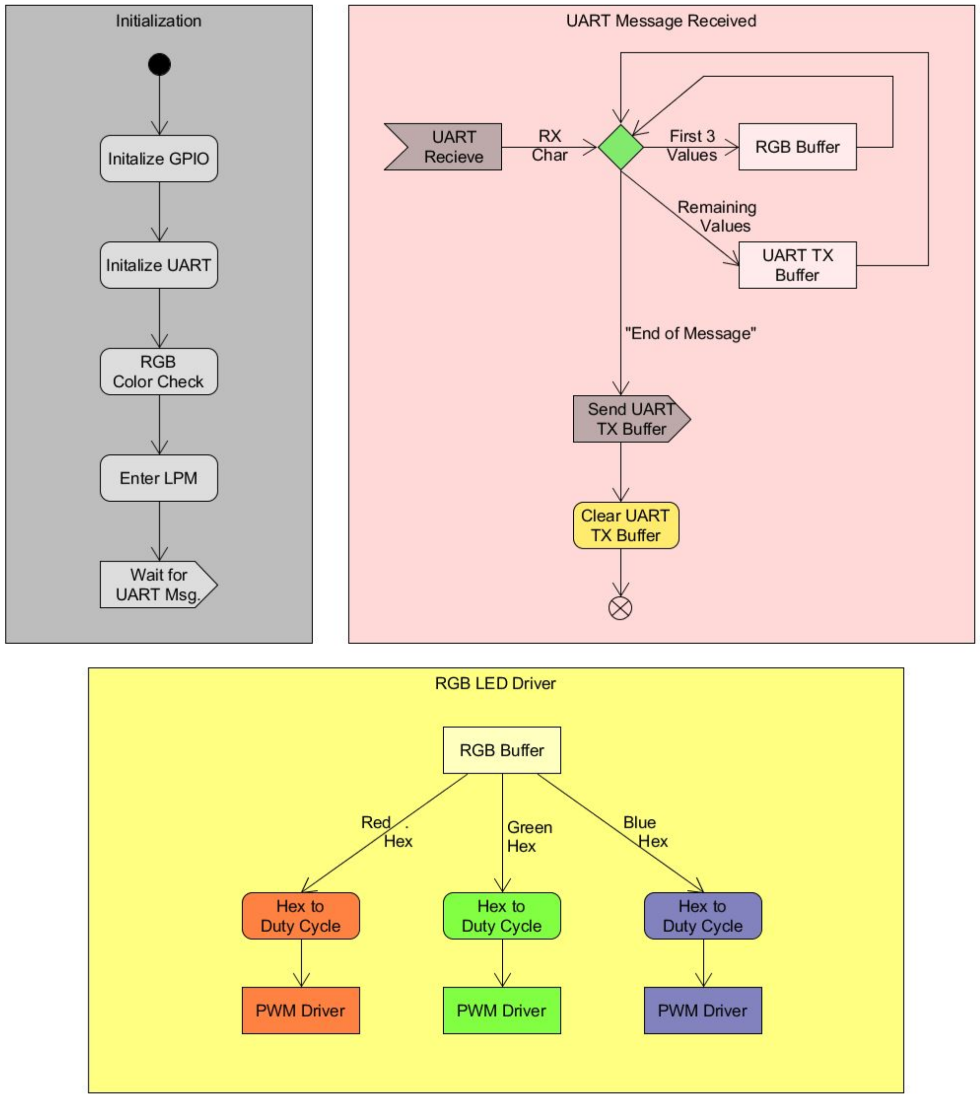


*Figure 2 & Figure 3*

Each node will be configured to initialize timers and use peripherals upon starting up and enter a low power mode while waiting for a message to be received over UART. When a message begins to be received, the first three bytes should be stored into a buffer which can be then be used to set the duty cycles for the RGB LED.

The remainder of the message should be placed into a UART TX buffer and when the incoming message is complete, it should be sent over the UART TX line. Once the message has been transmitted, the RGB LED will be set accordingly. The flow is outlined to the right.



## 3. Background

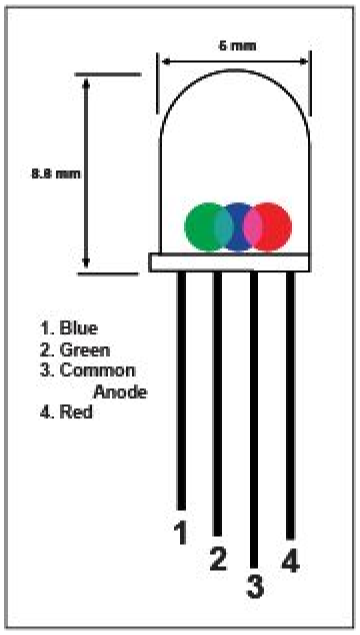
### 3.1 MSP430 Processor

The MSP430F5529 processor will be used for this application note. The F5529 has enough hardware addressable pins from the timer modules for hardware PWM applications. Additionally, the F5529 is one of the easier of the family of processors for PWM and UART applications. Because it is easy to develop for and because there is a large community of help for this processor, we chose to use this processor.

The MSP430 family of ultra-low power microcontrollers consists of several devices featuring peripheral sets targeted for a variety of applications. The architecture, combined with extensive low-power modes, is optimized to achieve extended battery life in portable measurement applications. The microcontroller features a powerful 16-bit RISC CPU, 16-bit registers, and constant generators that contribute to maximum code efficiency.

### 3.2 RGB LED

The embedded processor will ultimately be addressing the values of an RGB LED. An RGB LED contains 3-LEDs: Red, Green, and Blue. There are individual addressable pins for each color. The duty cycle of each color can be altered by using hardware PWM from the microprocessor. Adjusting the duty cycles for each color can provide any color combination.



The RGB LED we are using in this application node is *common anode*. The anode of the LED is connected to VCC (+5V) and a current limiting resistor is connected to each individual LED. Setting a *low* signal to the output pin will cause the LED to turn on. Setting a *high* signal to the output pin will cause the LED to turn off. This is called current sinking. *Figure 4*

### 3.3 UART Communication

Universal asynchronous receiver-transmitter is a computer hardware device for serial communication in which the data format and transmission speeds are configurable. UARTs are commonly included in microcontrollers such as the MSP430F5229.

UART is used to communicate between the microprocessor and a master node in this application node. The master node will send multiple bytes of data from a terminal to our microcontrollers. The microcontrollers will perform actions based on the data received.

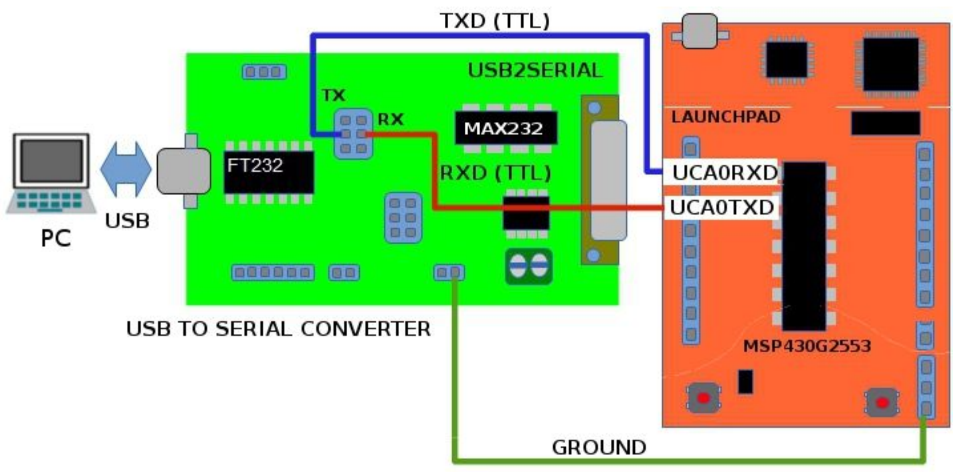


*Figure 5*

From the diagram (Figure 5) above the process of UART communication between an MSP430 processor and a PC is simplified. The Receive, Transmit, and Ground lines of the processor connect to the USB-to-Serial converter and the data is converted to be read by software in UART protocol. In section 4.4.3 we will discuss the register configurations to initialize UART for this application.

## 4. Discussion and Results

### 4.1 Data communication

Data will be received to each node through UART communication. For this application our UART transmission speed will be set to 9600 baud. The master terminal will be sending each node multiple bytes of data. At maximum we can expect 79 bytes total. Each node should take information from the master, perform action, then repackage the information to the next node in the chain. The following table explains the format of bytes received by a node:

| Byte Number | Contents | Example |
|---|---|---|
| Byte 0 | Number of bytes (N) including this byte | 0x50 (80 bytes) |
| Bytes 1 - (N - 2) | RGB colors for each node | 0xFF (red) 0x00 (green) 0x88 (blue) ... |
| Byte N - 1 | End of message character | 0x0D (carriage return) |

*Table 1*

**Example communication between two nodes**
The following would be received by a node

| Byte Number | Content | Meaning |
|---|---|---|
| Byte 0 | 0x08 | 8 total bytes in the package |
| Byte 1 | 0x7E | Red (current node): 50% duty |
| Byte 2 | 0x00 | Green (current node): 0% duty |
| Byte 3 | 0xFF | Blue (current node): 100% duty |
| Byte 4 | 0x40 | Red (next node): 25% duty |
| Byte 5 | 0xFF | Green (next node): 100% duty |
| Byte 6 | 0x00 | Blue (next node) 0% duty |
| Byte 7 | 0x0D | End of message check byte |

*Table 2*

After taking bytes 1-3 for use in the current node, the message would be repackaged into a new message and transmitted to the next node.

| Byte Number | Content | Meaning |
|---|---|---|
| Byte 0 | 0x05 | 5 total bytes |
| Byte 1 | 0x40 | Red (current node): 25% duty |
| Byte 2 | 0xFF | Green (current node): 100% duty |
| Byte 3 | 0x00 | Blue (current node): 0% duty |
| Byte 4 | 0x0D | End of message check byte |

*Table 3*

Each node will take in multiple bytes of data. The first byte indicates how many bytes there are total (including itself). The next three bytes are values to control the Red Green Blue LEDs respectively. The bytes after should be repackaged to be sent to the next node.

### 4.2 RGB LED

The pin for each color of the LED is connected with a current limiting resistor. The forward current for all colors (red, green, blue) is 20mA.

For the Red LED,
At forward current $I_F$ = 20mA, forward voltage $V_F$ = 2.0V (typically).
The current limiting resistor can be calculated by,

$$R_{lim} = \frac{V_i - V_f}{I_f} = \frac{5V - 2V}{20mA} = 150\Omega$$

For the Green LED,
At forward current $I_F$ = 20mA, forward voltage $V_F$ = 3.2V (typically).
The current limiting resistor can be calculated by,

$$R_{lim} = \frac{V_i - V_f}{I_f} = \frac{5V - 3.2V}{20mA} = 90\Omega$$

For the Blue LED,
At forward current $I_F$ = 20mA, forward voltage $V_F$ = 3.2V (typically).
The current limiting resistor can be calculated by,

$$R_{lim} = \frac{V_i - V_f}{I_f} = \frac{5V - 3.2V}{20mA} = 90\Omega$$

The RGB LED is a common anode component and should be configured in the following manner. Each LED should have its own current limiting resistor. The common anode should be connected to VCC (+5V).

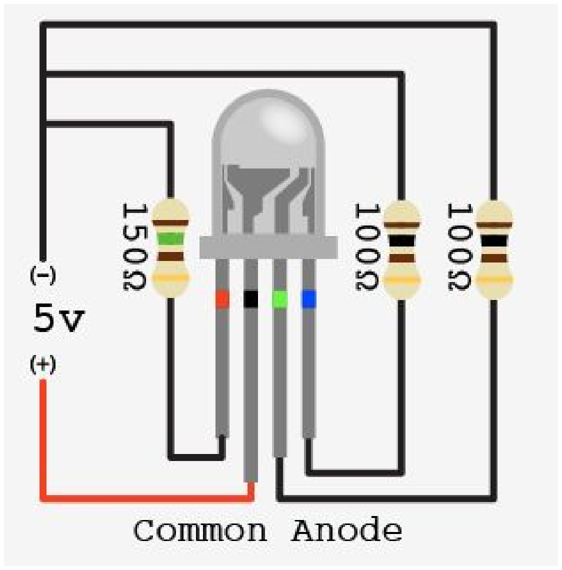


*Figure 6*

## 4.3 Hardware PWM

Hardware PWM (Pulse Width Modulation) will be used to control the duty cycle of the LEDs by utilizing the Timer A peripheral present on the MSP430F5529. PWM will allow different color combinations, rather than just solid red/green/blue. PWM controls the brightness by determining the duty cycle of the LEDs, where the higher the percentage, the brighter the LED. All of the LEDs start with a 0% duty cycle, meaning initially, they are all off, while a 100% duty cycle would mean the LED is on at full brightness. The PWM sets the duty cycle by pulsing the output of the microcontroller for a certain amount of time, hence the name pulse width modulation. In this case, the PWM is a hardware PWM because the pulsing is controlled by the clock peripheral of the microcontroller, which is a hardware element.

Hardware PWM on the MSP430 is done by configuring the timer modules to output to a timer output pin directly. In our application, we set configure the timer in *UP* mode and *OUTPUT MODE 7: RESET/SET*. When the timer is *UP* mode, the timer will increment until it reaches the value set in a capture compare register. This capture compare register determines the period of the signal. For this application, we set our capture compare registers to 255 as this works to ultimately create a 1kHz signal and helps map our RGB values to a duty cycle.

Every time the capture compare register value is reached it will perform the *RESET/SET* output, then reset the timer and start incrementing until the capture compare register value is reached again. The PWM process is visually demonstrated in the diagram below.

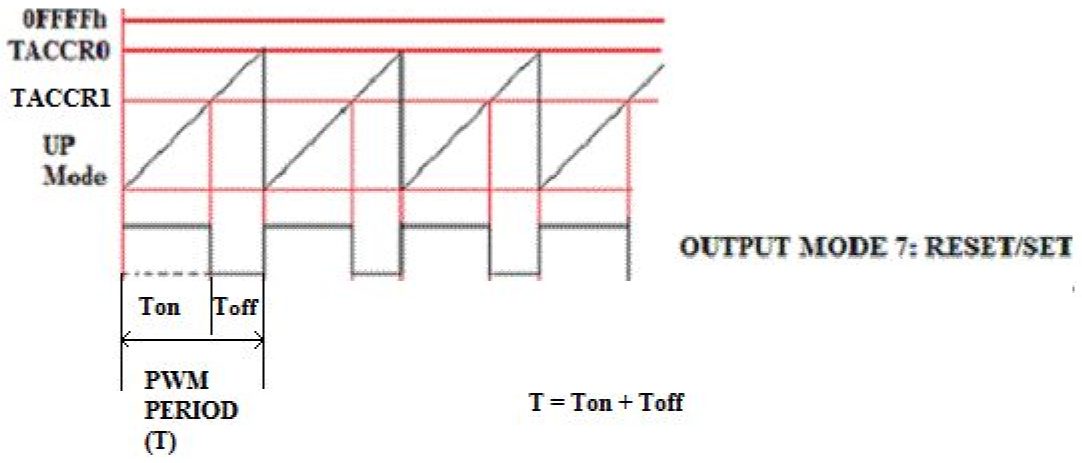


*Figure 7*

## 4.4 Code implementation

The code initializes the LEDs, Hardware PWM, and UART in the main function.

### 4.4.1 unitLeds(void)

This function sets the pins for output for the LEDs. The status LED is set as an output and the Red, Green, and Blue LEDs are set as outputs for the timer modules. Some unused pins are disabled to avoid unnecessary power draw.

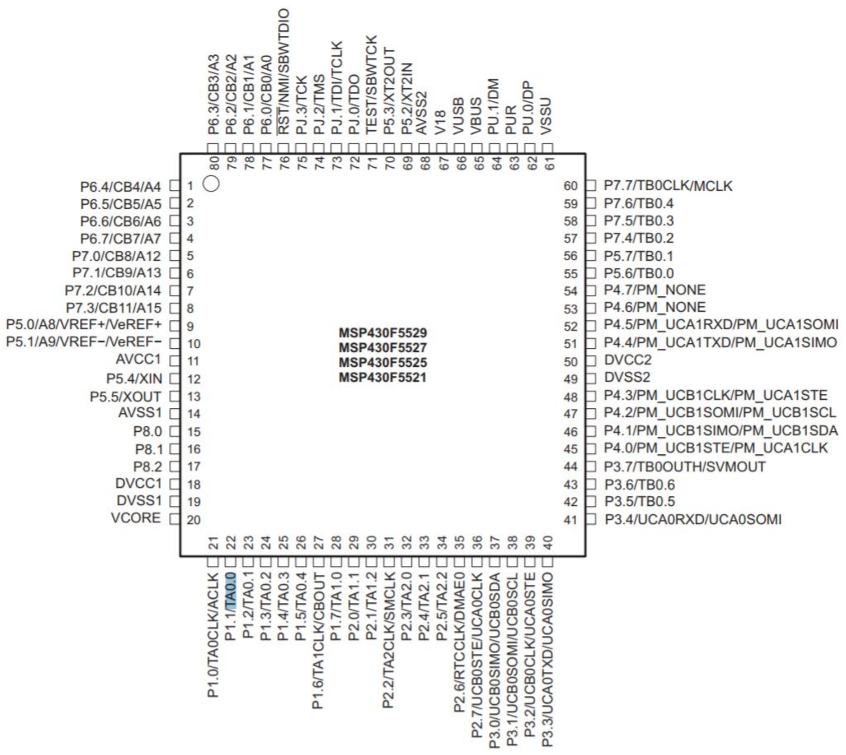


*Figure 8*

The pinout diagram for the MSP430F5529 shows the possible outputs for the Red/Green/Blue LEDs. The status LED is pre-defined as it is hardwired to pin P1.0 on the LaunchPad development board.

However, the output pins for the timers are variable and we must chose a pin which is able to be controlled by the timer peripheral and is also accessible from the LaunchPad development board.

- Pin P1.2 is accessible on the LaunchPad and also an output for TA0.1. This will be used for our Red LED.
- Pin P1.4 is accessible on the LaunchPad and also an output for TA0.3. This will be used for our Blue LED.
- Pin P2.0 is accessible on the LaunchPad and also an output for TA1.1. This will be used for our Green LEDs.

During testing of the application, we found unusual and unexpected behavior when using a third output pin for the Green LED under the Timer A0 module. Therefore, we switched the Green LED to use the Timer A1 module.

### 4.4.2 initPWM(void)

This function initializes timers for outputting to pins for hardware PWM. Timer A0 is set to the submode clock (1MHz) in *up* mode with a clock divider of 4 (ID_2). The capture compare register (period) is set to 255.

Therefore the clock is theoretically set to fire on a 1kHz frequency:
SMCLK = 1MHz
1MHz/4 = 250kHz

The clock interrupts with a period of 255 so,
250kHz / 255 = 980Hz.

Because 255 is an odd number it will be near impossible to reach an exact 1kHz frequency. Therefore, 980Hz is a desirable frequency when the goal is 1kHz.

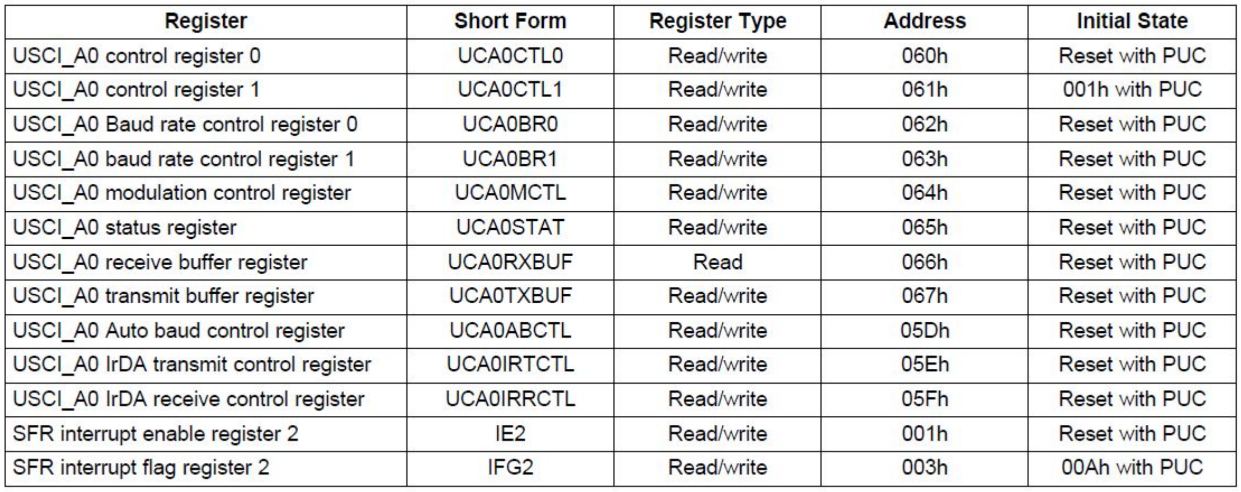
Capture compare registers TA0R1 and TA0R3 are set according to the duty cycle desired. When the register is set to 0 the duty cycle will be 100%. When the register is set to 255 the duty cycle will be 0%. Capture compare registers 1 and 3 are hooked up to the Red and Blue LEDs respectively using *OUTPUT MODE 7: RESET/SET*. This is explained more in section 4.3 Hardware PWM.

Timer A1 is set to the same clock configuration. Capture compare compare register TA1R1 is hooked up to the Green pin.

### 4.4.3 initUART(void)

This function initializes the UART communication and interrupt. Referring back to the MSP430F5529 pinout diagram in section 4.4.1 Figure 7 there are specific pins designated to the UART controller on the microprocessor.

- Pin P3.3 is UART Transmit
- Pin P3.4 is UART Receive



| Register | Short Form | Register Type | Address | Initial State |
|---|---|---|---|---|
| USCI_A0 control register 0 | UCA0CTL0 | Read/write | 060h | Reset with PUC |
| USCI_A0 control register 1 | UCA0CTL1 | Read/write | 061h | 001h with PUC |
| USCI_A0 Baud rate control register 0 | UCA0BR0 | Read/write | 062h | Reset with PUC |
| USCI_A0 baud rate control register 1 | UCA0BR1 | Read/write | 063h | Reset with PUC |
| USCI_A0 modulation control register | UCA0MCTL | Read/write | 064h | Reset with PUC |
| USCI_A0 status register | UCA0STAT | Read/write | 065h | Reset with PUC |
| USCI_A0 receive buffer register | UCA0RXBUF | Read | 066h | Reset with PUC |
| USCI_A0 transmit buffer register | UCA0TXBUF | Read/write | 067h | Reset with PUC |
| USCI_A0 Auto baud control register | UCA0ABCTL | Read/write | 05Dh | Reset with PUC |
| USCI_A0 IrDA transmit control register | UCA0IRTCTL | Read/write | 05Eh | Reset with PUC |
| USCI_A0 IrDA receive control register | UCA0IRRCTL | Read/write | 05Fh | Reset with PUC |
| SFR interrupt enable register 2 | IE2 | Read/write | 001h | Reset with PUC |
| SFR interrupt flag register 2 | IFG2 | Read/write | 003h | 00Ah with PUC |

*Table 4*

Additionally, the UART configuration specifies the baud rate [UCA0BR0/UCA0BR1] (9600) and the clock [UCA0CTL1] (SMCLK). The interrupt [UCA0IE/UCRXIE] is configured so when a byte of received from a terminal it will call a function to perform actions.

### 4.4.4 main(void)

This function stops the watchdog timer and initializes UART, LEDs, and Hardware PWM timers in that order. Lastly, the MSP4305529 is put into low power mode with the ability to output to GPIO.

**4.4.5 USCI_A0_ISR(void)**
This is the interrupt which is called when an incoming byte is being received from the UART controller. Whenever this interrupt is called, we turn on the Status LED to indicate that we are receiving data for debugging purposes.

If there is data in the UART Receive buffer we go through a switch statement (state machine) to figure out what byte is being received and what action to perform. If the first byte is being received (the length of the message), we need to read the length and transmit back the length minus three (because three of the bytes will be used to control our RGB LED).

If we are processing the second byte, we need to set our Red LED to the color specified in the message. Because we are using a common anode LED the message will be converted to an integer and subtracted 255 from the value to invert it. The inverted value is set to the capture compare register which corresponds to the Red LED output.

If we are processing the third or fourth byte, the same procedure as the Red LED is performed except with the Green and Blue output pins.

If the byte we are receiving is anything more than the fourth byte, our node does not care about it and we can simply transmit the data back to the next nodes in the chain. After all actions are performed, we check to see if the Receive data contains the ‘end of string’ value (0x0D). If it is not the end of string we increment the variable which keeps track of the number of byte received and advances the state machine. Else, it is the end of the message and we can reset our state machine and be ready to receive another message. A flowchart outlining this process can be seen below:

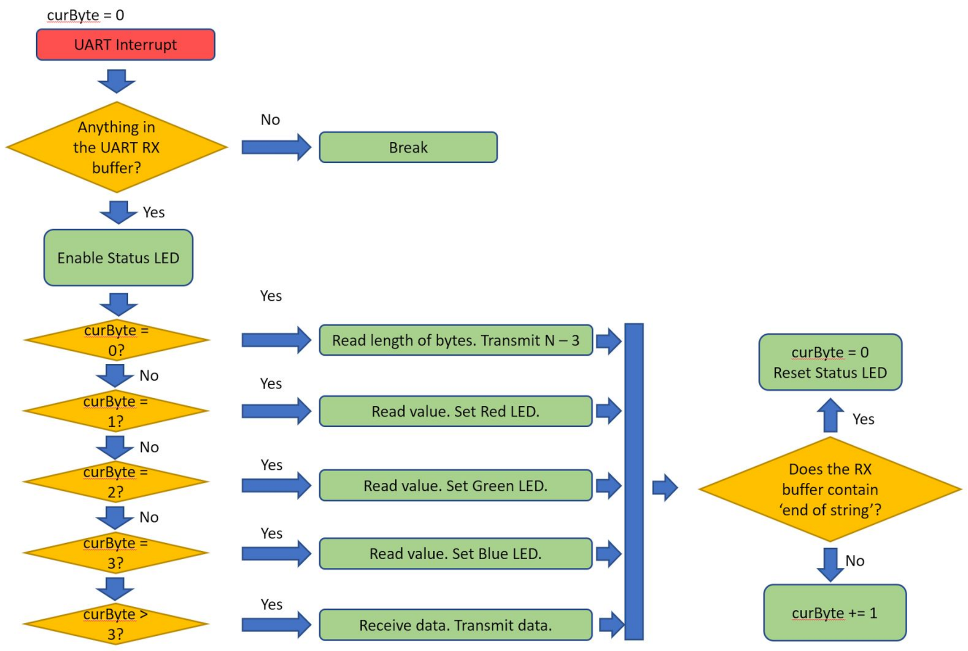


*Figure 9*

### 4.5 Usage of application

The MSP430F5529 is wired to a breadboard with the common anode RGB LED. The circuit is setup as shown in the diagram below:

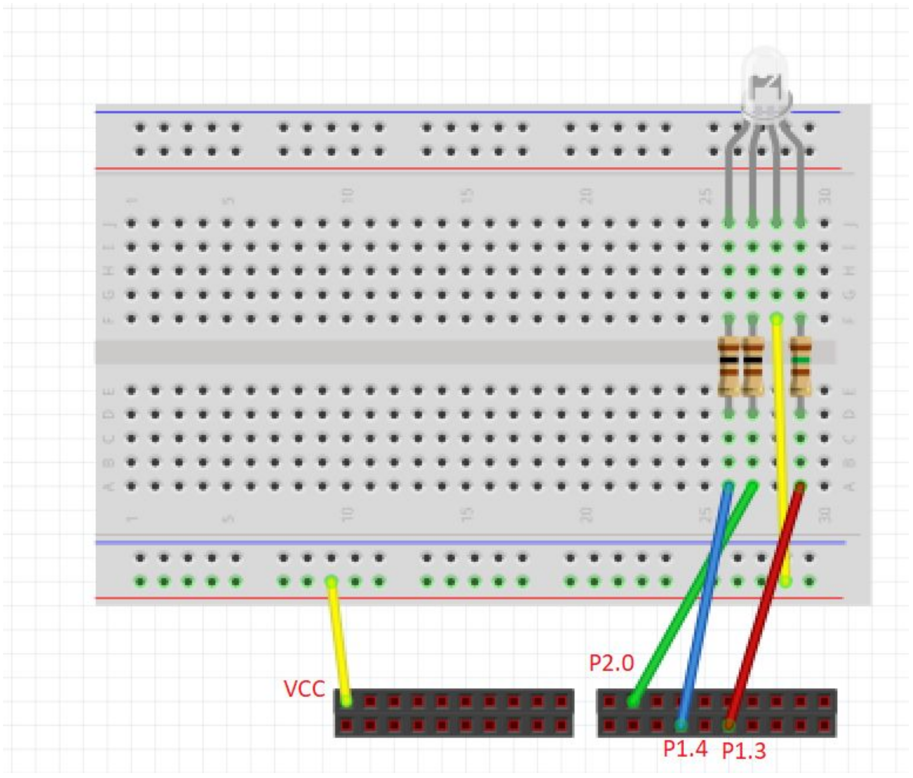


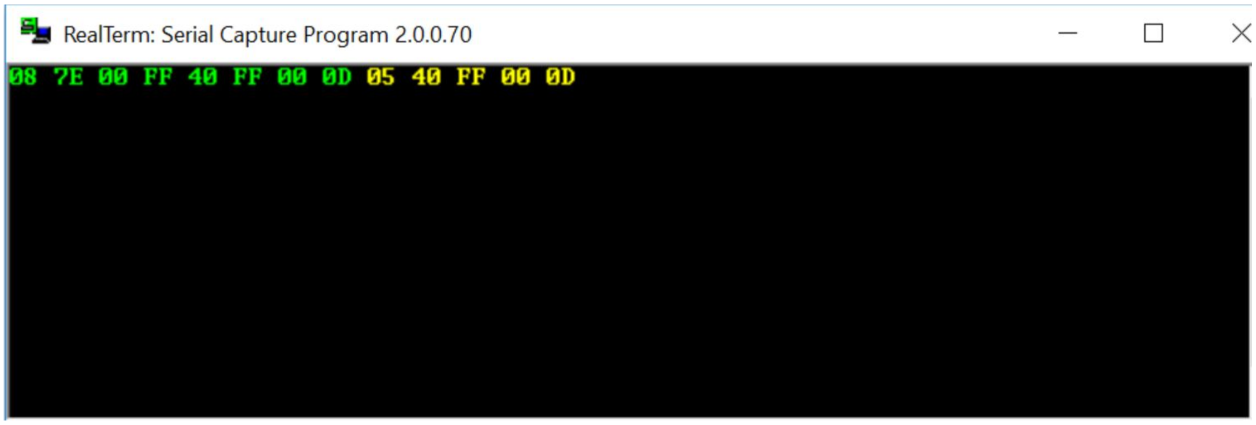
*Figure 10*

For UART communication the team used a Serial-to-USB converter for UART communication. The converter cable connects to ground, Receive [green] pin (P3.4), Transmit [white] pin (P3.3) on the MSP4305529. Using engineering software *Realterm* as a terminal for serial communication, we can simulate a message of hexadecimal values to the processor.

In *Realterm* ensure the following configurations are set:

- Display: Display as Hex[space]
- Display: Half-Duplex on
- Port: Baud 9600
- Port: Prolific USB-to-Serial Converter

For a sample, we will test the hexadecimal values explained in section 4.1 Table 2:
**0x08 0x7E 0x00 0xFF 0x40 0xFF 0x00 0x0D**

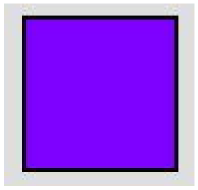


```
08 7E 00 FF 40 FF 00 0D 05 40 FF 00 0D
```

*Figure 11*

Because Half-Duplex is enabled, we will see what was transmitted in green and what was received from the MSP430F5529 in yellow.

Bytes 1 - 3 correspond to the RGB values which are to be set for our node. Byte 1, Byte 2, and Byte 3 are 0x7E, 0x00, and 0xFF respectively. This corresponds to RGB(126, 0, 255) (sample seen to the right). Therefore, our RGB LED lights up this color.



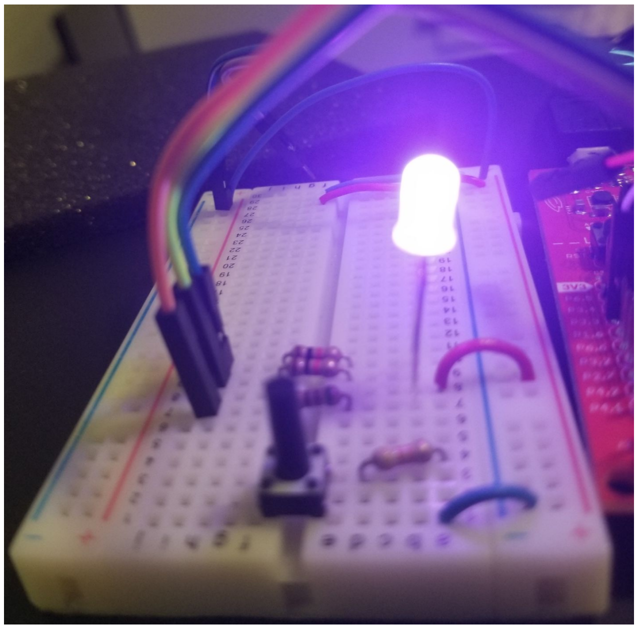
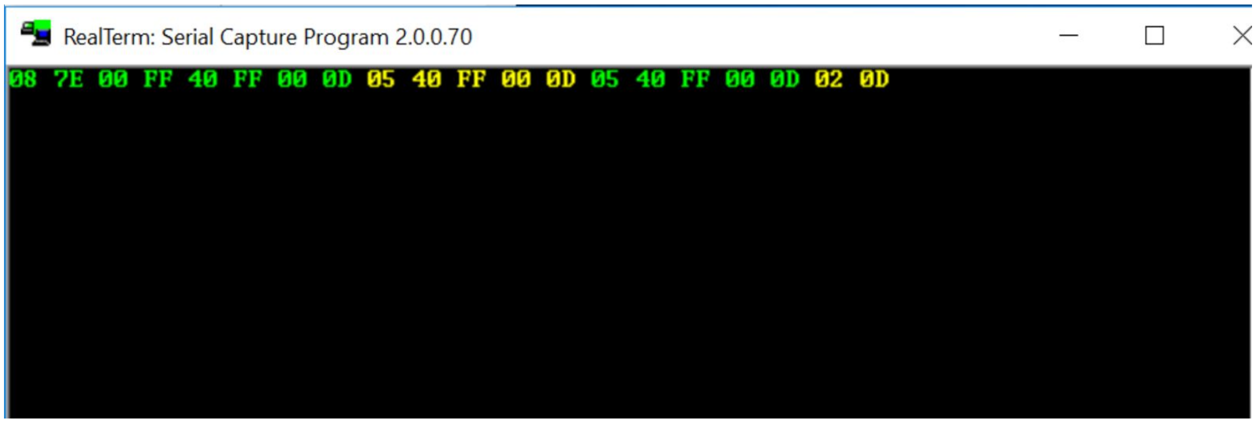


*Figure 12*

Bytes 4 - 7 are to be received by the next node so we need to package those values to be transmitted. The total length of the next message will be 5 (8 minus 3). The next message will look as follows:

**0x05 0x40 0xFF 0x00 0x0D**

Let's assume that we are the next node in the chain. We will try to input the next message in *Realterm*.



RealTerm: Serial Capture Program 2.0.0.70

```
08 7E 00 FF 40 FF 00 0D 05 40 FF 00 0D 05 40 FF 00 0D 02 0D
```

*Figure 13*

Because Half-Duplex is enabled, we will see what was transmitted in green and what was received from the MSP430F5529 in yellow.

Bytes 1 - 3 correspond to the RGB values which are to be set for our node. Byte 1, Byte 2, and Byte 3 are 0x40, 0xFF, and 0x00 respectively. This corresponds to RGB(64, 255, 0) (sample seen to the right). Therefore, our RGB LED lights up this color.

🟩

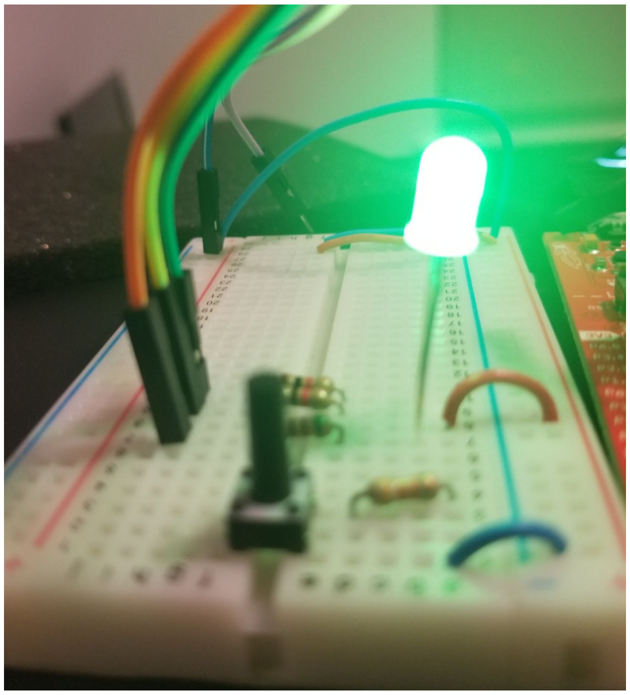


*Figure 14*

There are no RGB values left in the message to send to the next node. The total length of the next message will be 2 (5 minus 3). The next message will look as follows:

**0x02 0x0D**

The next node should realize that Byte 1 is a ‘end of string’ carriage return and it should not perform any actions.

These two demonstrations prove that our application is receiving data, performing actions based on the information, and successfully transmitting data to be received by additional nodes.

**4.6 Board choice and issues**
The MSP430F5529 was ultimately chosen as the platform to build this application on. The board was chosen for the following reasons:

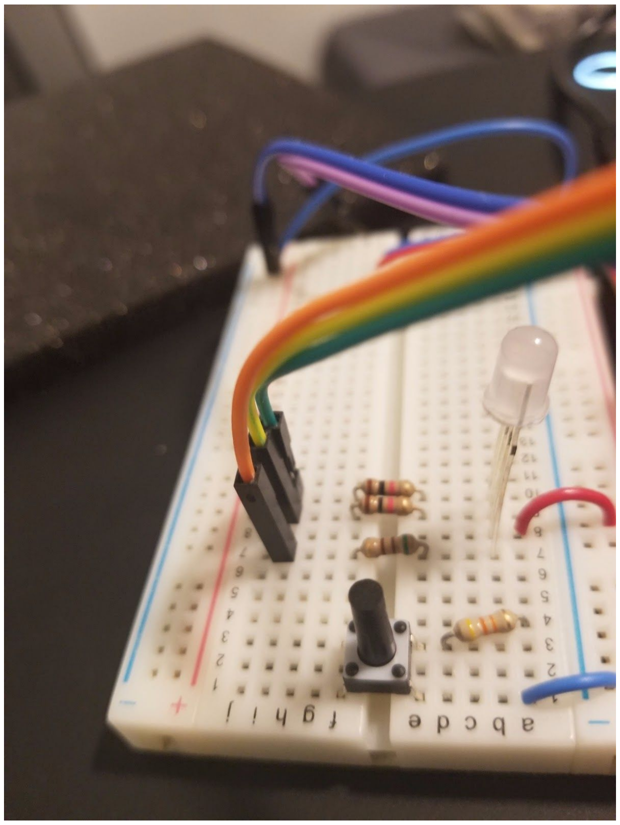
- Accessible pins from the timer outputs hardware perperhial
- Sufficient amount of capture compare registers for timer PWM
- Large of amount of community resources and help topics

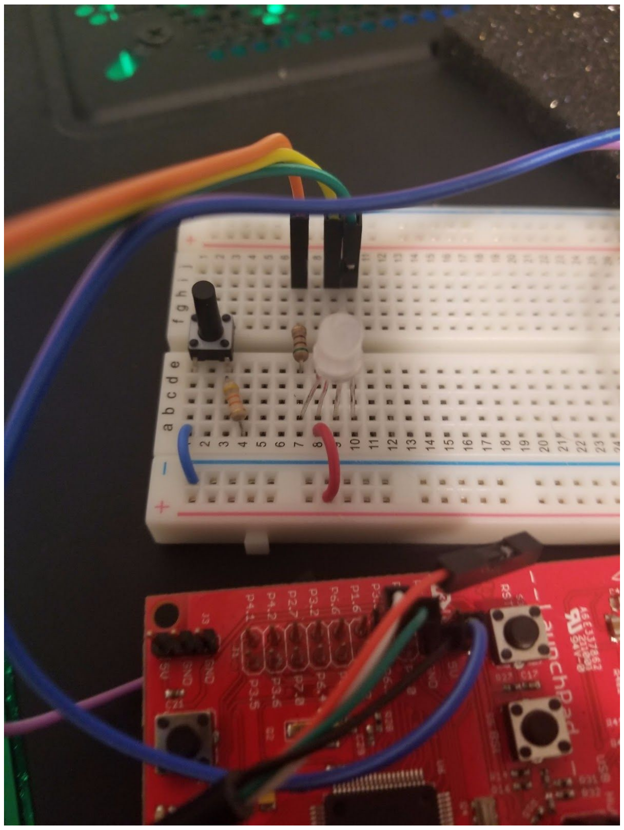
Albeit a processor such as the MSP430G2553 is cheaper, the application would have to been done using Software PWM due to lack of accessible output pins and UART pin conflicts. Additionally, we found that the MSP430G2553 repeatedly shut down during testing due to excessive current draw. In a real-world scenario, we believe that the ease of software development may outweigh the processor costs.

The decision to use Timer A1 instead of Timer A0 stems from unusual behavior we experienced during testing. When the Green LED was connected to any output pin from Timer A0 the duty cycle would be constant 100%, no matter what we specify. However, when we change the Green LED to be an output from Timer A1 there was no such problem. Due to this behavior, we decided to make a design decision for the Green LED to be an output of Timer A1.

During testing of the application we also encountered issues with the default Application UART over the USB port. The UART communication would not successfully interface with any terminal program on the PC. However, using a USB-to-Serial converter solved this issue and we were then able to connect using a terminal. The design decision was made to interface our processor using a USB-to-Serial converter cable.

## 5. Appendix





```
/**
 * main.c
 Receive a string of hexadecimal values via UART.
 Control a RGB LED using hardware PWM.
 MSP430F5529
 Stephen Glass
 */

#include <msp430.h>


/* Macros and Prototypes */

#define LED_RED     BIT2
#define LED_GREEN   BIT0
#define LED_BLUE    BIT4
#define LED_STATUS  BIT7

unsigned int curByte = 0;
```

```
void initUart(void);
void initLeds(void);
void initPWM(void);

/* Initialize UART */

void initUart(void)
{
   P3SEL    |=  BIT3;      // UART TX
   P3SEL    |=  BIT4;      // UART RX
   UCA0CTL1 |=  UCSWRST;   // Resets state machine
   UCA0CTL1 |=  UCSSEL_2;  // SMCLK
   UCA0BR0   =  6;         // 9600 Baud Rate
   UCA0BR1   =  0;         // 9600 Baud Rate
   UCA0MCTL |=  UCBRS_0;   // Modulation
   UCA0MCTL |=  UCBRF_13;  // Modulation
   UCA0MCTL |=  UCOS16;    // Modulation
   UCA0CTL1 &= ~UCSWRST;   // Initializes the state machine
   UCA0IE   |=  UCRXIE;    // Enables USCI_A0 RX Interrupt
}



/* Initialize LEDs */

void initLeds(void)
{
   P4DIR |= (LED_STATUS);    // Status LED
   P4OUT &= ~(LED_STATUS);   // Status LED default off
   P1DIR |= (LED_RED);       // P1.2 output
   P1SEL |= (LED_RED);       // P1.2 to TA0.1
   P2DIR |= (LED_GREEN);     // P2.0 output
   P2SEL |= (LED_GREEN);     // P2.0 to TA1.1
   P1DIR |= (LED_BLUE);      // P1.4 output
   P1SEL |= (LED_BLUE);      // P1.4 to TA0.3

   P1SEL &= ~(BIT1 + BIT5); // Disable unused GPIO
   P1DIR &= ~(BIT1 + BIT5);
}



/* Initialize PWM */
```

```
void initPWM(void)
{
    TA0CTL = (MC__UP + TASSEL__SMCLK + ID_2); // Configure TA0: Upmode using
1MHz clock / 4 = 250k
    TA0CCR0 = 255; // 250k / 255 = ~1kHz, set compare to 255
    TA0CCTL1 = OUTMOD_7; // Set to the output pin (Hardware PWM)
    TA0CCR1  = 255; // Set duty cycle to 0% as defau;t
    TA0CCTL3 = OUTMOD_7;
    TA0CCR3  = 255;

    TA1CTL = (MC__UP + TASSEL__SMCLK + ID_2); // Configure TA1
    TA1CCR0 = 255;
    TA1CCTL1 = OUTMOD_7;
    TA1CCR1 = 255;
}

/* Main */

void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;

    initUart();
    initLeds();
    initPWM();

    __bis_SR_register(LPM0_bits + GIE);
}

/* UART */

#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
    P4OUT |= LED_STATUS;
    switch(__even_in_range(UCA0IV,USCI_UCTXIFG))
    {
        case USCI_NONE: break;
        case USCI_UCRXIFG:
        {
            switch(curByte) // Switch between which byte is being processed
            {
```

```
            case 0: // Number of bytes (N) including this byte
            {
               while(!(UCA0IFG & UCTXIFG)); // Wait until buffer is ready
               UCA0TXBUF = (int)UCA0RXBUF - 3; // Number of bytes after processing will
be current - 3
               __no_operation();
               break;
            }
            case 1:
            {
               TA0CCR1 = 255 - (int)UCA0RXBUF; // Set the correct duty cycle to the RED
LED
               break;
            }
            case 2:
            {
               TA1CCR1 = 255 - (int)UCA0RXBUF; // Set the correct duty cycle to the GREEN
LED
               break;
            }
            case 3:
            {
               TA0CCR3 = 255 - (int)UCA0RXBUF; // Set the correct duty cycle to the BLUE
LED
               break;
            }
            default:
            {
               while(!(UCA0IFG & UCTXIFG)); // Wait until the buffer is ready
               UCA0TXBUF = UCA0RXBUF; // Everything after RGB is extra that we need to
send to next node
               break;
            }
         }
         if(UCA0RXBUF != 0x0D) curByte++; // Don't increment and do stuff if there's
nothing to parse
         else // If we reached the end of the string
         {
            curByte = 0; // Reset and start receiving again
            P4OUT &= ~(LED_STATUS); // Reset the status LED
         }
         break;
      }
```

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