Milestone 1: Communicating with Will Byers

Glenn Dawson, Joshua Gould, and Jack Pedicone
Rowan University

October 17, 2017

1 Abstract

The milestone project was intended to simulate the Christmas lights communication from Stranger Things, where Will Byers communicates with his mother via a series of lights corresponding to letters of the alphabet. In order to achieve this, we worked with the MSP430F5529 development board to create a ”node” that accepted a package of hex bytes over UART communication, extracted duty cycle information for an RGB LED associated with the node, and transmitted the byte package to the next node, sans the RGB information for that node.

The UART protocol was provided in the project description; the implementation was left for us to develop independently. In order to implement the UART protocol, we utilized a series of switch statements nested inside of the USCI interrupt vector. This allowed us to determine which byte in the package was being inspected, and either assign that byte’s data to one of the RGB LED’s pulse-width modulation timers or pass the byte to the next node.

2 Introduction

In the Netflix series Stranger Things, a character named Will Byers is trapped in a parallel dimension, and can only communicate with this dimension by manipulating a string of Christmas lights corresponding to letters in the alphabet; he sends his messages letter-by-letter, as a Ouija, by blinking the appropriate light.

For this project, the Christmas lights have been reformulated into a series of RGB LEDs, each controlled by an MSP430 microprocessor controller. These microprocessors are themselves connected in series via UART communication channels, with each controller accepting data from the previous controller on its RX buffer and transmitting data to the next controller via its TX buffer. Will Byers himself has been abstracted into a ”master node”; that is, the series of MSP430 controllers is sent a package of hex bytes composed by the master node that contains RGB information for each node in the series.

2.1 UART Protocol

The package of hex bytes obeys a custom UART protocol designed specifically for this lab. In terms of global UART settings, the BAUD rate is set to 9600 bits per second. The messaging protocol itself asserts that each node will use three bytes to set their respective red, green, and blue LED duty cycles; therefore, for a series of 26 nodes (corresponding to 26 letters), each package will consist of

$$3\ bytes \times 26\ nodes = 78\ bytes$$

For debugging purposes, a trailing byte is added at the end of each package, consisting of a carriage return (ASCII 'CR', hex '0x0D'). In addition, a header byte is transmitted, containing the number of bytes in the package. Therefore, the total number of bytes in the package is

$$1\ header + 3\ bytes \times 26\ nodes + 1\ carriage\ return = 80\ bytes$$

It is important to note that note every node will be receiving the full 80 bytes; as the package is passed through each node, the node strips its own RGB data out of the first three non-header bytes, and then removes those bytes from the package that is passed to the next node. That is, where Byte 0 is the header node containing the number of bytes in the package, Bytes 1, 2, and 3 are used strictly by the receiving node, and bytes 4 through N-1 (where N is the number of bytes in the package; the final byte is byte N-1 since the byte indexing begins at 0) are not used by the receiving node and are transmitted without modification to the next node. An example of this is as follows:

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

After the current node takes Bytes 1-3 and uses them to determine its own duty cycles, it will pass the following package to the next node:

Byte Number	**Content**	**Meaning**
Byte 0	0x05	5 total bytes in the package
Byte 1	0x40	Red (next node): 25% duty cycle
Byte 2	0xFF	Green (next node): 100% duty cycle
Byte 3	0x00	Blue (next node): 0% duty cycle
Byte 4	0x0D	End of message (carriage return)

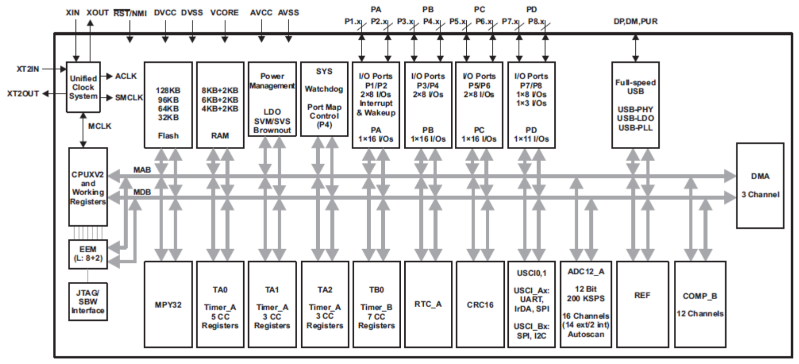
Note that the information contained in Byte 0 has changed; the number of bytes in the package has been reduced by 3. When the final node receives its package, it will strip the package of its RGB duty cycle data and then pass the extraneous package length and carriage return bytes to either a collector or nothing (if there is no receiver connected to its TX buffer).

3 Background

3.1 MSP430F5529 Architecture

The MSP430F5529 microcontrollers have support for USB 2.0, four 16-bit timers, a high-performance 12-bit analog-to-digital converter (ADC), two universal serial communication interfaces (USCI), a hardware multiplier, DMA, a real-time clock (RTC) module with alarm capabilities, and 63 I/O pins.

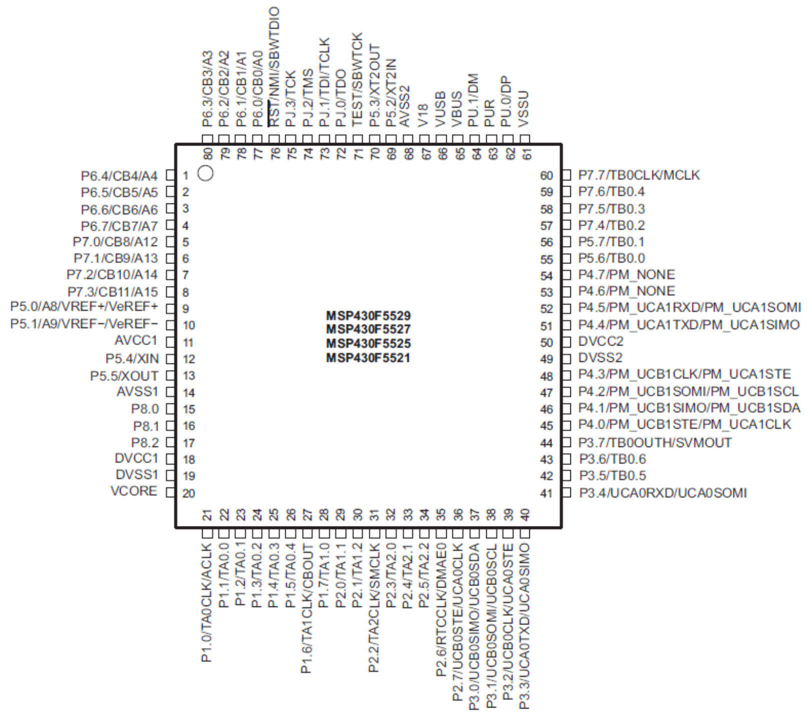
Figure 1: Block Diagram of the MSP430F5529 Microprocessor


XIN
XOUT
RST/NMI
DVCC
DVSS
VCORE
AVCC
AVSS
PA
PB
PC
PD
P1.x
P2.x
P3.x
P4.x
P5.x
P6.x
P7.x
P8.x
DP,DM,PUR
XT2IN
XT2OUT
Unified Clock System
ACLK
SMCLK
MCLK
128KB 96KB 64KB 32KB Flash
8KB+2KB 6KB+2KB 4KB+2KB RAM
Power Management LDO SVM/SVS Brownout
SYS Watchdog Port Map Control (P4)
I/O Ports P1/P2 2×8 I/Os Interrupt & Wakeup PA 1×16 I/Os
I/O Ports P3/P4 2×8 I/Os PB 1×16 I/Os
I/O Ports P5/P6 2×8 I/Os PC 1×16 I/Os
I/O Ports P7/P8 1×8 I/Os 1×3 I/Os PD 1×11 I/Os
Full-speed USB USB-PHY USB-LDO USB-PLL
CPUXV2 and Working Registers
MAB
MDB
DMA 3 Channel
EEM (L: 8+2)
JTAG/ SBW Interface
MPY32
TA0 Timer_A 5 CC Registers
TA1 Timer_A 3 CC Registers
TA2 Timer_A 3 CC Registers
TB0 Timer_B 7 CC Registers
RTC_A
CRC16
USCI0,1 USCI_Ax: UART, IrDA, SPI USCI_Bx: SPI, I2C
ADC12_A 12 Bit 200 KSPS 16 Channels (14 ext/2 int) Autoscan
REF
COMP_B 12 Channels

3.1.1 Pin Assignments

Pin assignments for the MSP430F5529 are provided in the MSP430F5529 data sheet. The pin assignments for the MSP430F5529 design are provided in Figure 2 below.

Figure 2: Pin Assignments for the MSP430F5229 Microprocessor



Pins used in this project are provided in the table below.

Pin	***Description***
P1.3	General-purpose digital I/O with port interrupt;
P1.4	General-purpose digital I/O with port interrupt
P1.5	General-purpose digital I/O with port interrupt
P3.3	Transmit data UART mode
P3.4	Receive data UART mode

Table 1: Pin assignments and purpose used in Lab. Provided by F5529 Datasheet

3.1.2 Timers

The timer clock can be sourced from ACLK, SMCLK, or externally from TAxCLK or INCLK. The clock source is selected with the TASSEL bits. The selected clock source may be passed directly to the timer, or have its value divided by using the ID bits or TAIDEX bits. The timer clock divider logic is reset when TACLR is set.

Timer clocks also perform in different operations. These operations can be UP, UP-DOWN, and CONTINUOUS. For our implementation, the clock was chosen to be

MC	***Mode***	***Description***
00	Stop	The timer is halted.
01	Up	The timer repeatedly counts from zero to the value of TAxCCR0
10	Continuous	The timer repeatedly counts from zero to 0FFFFh.
11	Up/Down	The timer repeatedly counts from zero up to the value of TAxCCR0 and back down to zero.

Table 2: Timer mode types

sourced by SMCLK in UP mode. The descriptions of these modes are provided above in Table 1. An example of timer select from our code is can be seen in the following snippet from our initializeGPIO function:

```
TA0CTL |= TASSEL_2 + MC_1 + TACLR;        // 1 Mhz clock in Up mode
TA0CCR0 = 1020;                          // 1 kHz frequency
```

3.1.3 Pulse-Width Modulation

The compare mode, selected when CAP = 0, is used to generate PWM output signals or interrupts at specific time intervals. PWM output can be produced by using a single TAxCCRx. For TAxCCR0, for example, either the rising or the falling edge of the PWM signal is controlled by TAxCCR0, or the period overflow.

The duty cycle of the PWM can be modulated by comparing the value within TAx-CCR0 to other registers. This can be done by assigning another capture/compare control, such as TA0CCTL1, to OUTMOD7 or reset/set. OUTMOD7 specifies that the edge of the PWM signal will rise whenever the timer reaches zero, which includes when the timer in upmode has overflowed and returned to its original position. The edge of the PWM signal will fall when the value of TAxCCR1 is reached, and this rising and falling edge represents the duty cycle. Therefore, the duty cycle can be modulated by adjusted the value of the other TAxCCRx to the desired value. An example can be seen below:

```
TA0CTL |= TASSEL_2 + MC_1 + TACLR;        // 1 MHz clock in Up mode
TA0CCR0 = 1000;                         // 1 kHz frequency
TA0CCTL1 = OUTMOD_7;                         // Reset/Set toggle
TA0CCR1 = 500;                          // (500/1000) = 0.5, a 50% duty cycle
TA0CCTL2 = OUTMOD_7;
TA0CCR2 = 250;                         // (250/1000) = 0.25, a 25% duty cycle
```

3.1.4 Interrupts

Interrupts can be toggled by the software of the MSP430. A register uses the function globally for all interrupts or groups of interrupts, and then additional registers for individual interrupts. One very common use for interrupts is to detect changes in GPIO inputs. Stemming from our button interrupt code which uses the P1IN register, enabling an interrupt on a GPIO would allow the hardware to signal the software when the input has changed values. In the MSP430F5529, interrupts are used to operate the UART protocol to receive and transmit. The code is provided in the appendix detailing the UART procedures.

3.2 Programming in C

Code for the MSP430 family of microprocessors is primarily written in C. C is a programming language that allows for direct access to low-level architectural components such as registers. It is utilized in microprocessors and microcontrollers because it is extremely efficient; because programming in C allows for direct access to the architecture, there is no overhead in terms of abstraction when the code is compiled into machine language (Assembly).

Programming Texas Instruments microprocessors is done using Code Composer Studio, an integrated development environment (IDE) that specifically supports TI's embedded software and microprocessor hardware and includes a library of header files containing useful macros. In our code, we utilized the MSP430F5529.h header file, which allowed us to address important aspects of the MSP430F5529, such as the watchdog timer, ports, timers, and interrupt vectors.

4 Node Code

The MSP430F5229 microprocessor is controlled by the code found in Appendix A.

4.1 Heading

The heading begins with by #including the MSP430F5529.h header file, giving access to macros and vector names. Next, three new macros are defined using #define for the purposes of generalization, each corresponding to a specific output pin for a specific LED output.

The main() function is modularized by compartmentalizing the GPIO and UART initialization processes; these functions are prototyped in the heading, and defined below the main().

Finally, the global variables are initialized to control the UART protocol inside the USCI interrupt vector.

4.2 Initialize GPIO

The GPIO initialization function initializes the LED pins and the timers used in this implementation.

First, Timer 0 is set to use the onboard SMCLK (1,048,576 cycles per second) in Up mode; the TACLR command simply resets the Timer 0 registers so that the system will begin "fresh" at each startup. TA0CCR0, the value that Timer A0 will count up to, is set to 1020; this number was chosen because it is equal to four times the maximum value that can be held by an 8-bit hex byte (0xFF, or 255_d), as well as conveniently making the length of one period roughly $1\,\mathrm{ms}$.

The Port 1 pin assignments were chosen arbitrarily to be P1.4 for the red LED, P1.5 for the green LED, and P1.6 for the blue LED. On the MSP430F5529 board, these pins had no specific function by default, meaning they were free to use for our arbitrary pin assignments. These pins are asserted as outputs by setting the pin directions high on these bits, and by setting the pin function selects high on these bits, we assert them to be peripherals connected to the timer outputs.

The TA0CCTLx assignments that follow simply assert the behavior of the previously-defined outputs; output mode 7 is defined for the MSP430 family to be reset/set, that is, when the value stored in TA0CCRx is reached, the output pin is asserted low, and when the value stored in TA0CCR0 is reached, the output pin is asserted high. Because of this behavior, the duty cycle can be controlled by simply modulating the value of TA0CCRx, in this case by taking the input hex value and multiplying it by 4.

Finally, at initialization the TA0CCRx values are set to 0, meaning that the duty cycles of the red, green, and blue LEDs are initialized to 0%.

4.3 Initialize UART

The UART initialization code is taken directly from the sample code provided in Lab 1. It does magic.

4.4 Main

The main function of the node code simply disables the watchdog timer, calls the initialization functions for GPIO and UART, and includes a line that sets the microprocessor to low-power mode and enables global interrupts. The latter is extremely important, as without interrupts enabled, the controller will never be able to enter the interrupt vector code, which is where the data processing code is located.

4.5 USCI Interrupt Vector

This interrupt vector is where the incoming data, received as a UART transmission on the RX buffer, is processed. The original code is authored by Nick Gorab, though changes and optimizations have been made. The interrupt vector will activate whenever the USCI interrupt flag is raised; this includes whenever the RX buffer receives data, or whenever the TX buffer receives data. To deal with the various instances where this flag may have been raised, a switch statement is used. Each case is detailed below.

4.5.1 USCI NONE

This case handles the instances where the input to the interrupt vector is null. As there is nothing to process, no action need be taken, so this case immediately breaks out of the switch statement.

4.5.2 USCI UCRXIFG

This is the most important instance, where the interrupt flag was raised due to incoming data on the RX buffer. Once this flag is raised, we know that there is some kind of input data being provided; in order to determine how to process this data, a second switch statement is used, this time considering the global variable “byte” defined in the heading.

If the byte received is the first byte, then the controller first stores this information in the second global variable, “num_bytes” (used below). Then, it subtracts 3 from this data (representing the three bytes stripped by the node) and passes the difference to the TX buffer to be sent to the next node.

If the byte received is the second, third, or fourth byte, then the controller multiplies the incoming data by 4 (to scale the data to the TA0CCR0 maximum) and passes the product to the relevant TA0CCRx register, indicating the duty cycle of the appropriate LED.

For all other bytes, they are passed unchanged to the TX buffer to be sent to the next node.

The conditional statement following the internal switch statement considers the currently-indexed byte and compares it against the total number of bytes in the package. If the current byte is not the final byte, then the controller increments the byte counter to indicate that a new byte will be coming; if the current byte is the final byte, then the controller resets the byte counter to prepare for the next package.

4.5.3 USCI UCTXIFG

This case handles the instance where the TX buffer interrupt flag has been raised. As there is nothing to process, no action need be taken, so this case immediately breaks out of the switch statement.

Appendix A C Code

milestone_1.c

```
/************************************************************
*  Stranger Things RGB LED Node                   *
*  Authors: Glenn Dawson, Joshua Gould, and Jack Pedicone *
*  2017-10-16                               *
***********************************************************/

#include <msp430f5529.h>
#define RED BIT2;
#define GRN BIT3;
#define BLU BIT4;

void initializeGPIO();
void initializeUART();

volatile int byte = 0;
volatile int num_bytes;

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer

  initializeGPIO();    // Initialize timers and LEDs
  initializeUART();    // Initialize UART controls

  // Set low-power mode and enable global interrupts
  __bis_SR_register(LPM0_bits + GIE);
}

#pragma vector=USCI_A0_VECTOR
/* * Accepts data on RX buffer, determines which byte in
   the message is being examined
   * Uses "bytes" and "rgb" as state machine control
     variables
   * Assumes byte 0 is total number of bytes in message
      * Sends value of leading byte - 3 to next node
   * Extracts PWM values from bytes 1, 2, 3
   * Assumes final byte is '/r'
       * Passes all unused bytes, including final byte,
         to next node
   * Credit: Nick Gorab and Glenn Dawson           */

```

```
__interrupt void USCI_A0_ISR(void)
{
  switch (__even_in_range(UCA0IV, USCI_UCTXIFG)) {
    case USCI_NONE:
          break;
    case USCI_UCRXIFG:
          /* Switch statement determines how to handle
             byte input */
      switch (byte) {
                case 0:
          /* Wait until UTXBUF is empty
                       (UTXIFG1 = 1) */
          while (!(UCA0IFG & UCTXIFG));
          // Extract the total number of bytes
            num_bytes = UCA0RXBUF;
          /* Subtract the number of bytes needed
            (3) from the total */
            UCA0TXBUF = UCA0RXBUF - 3;
                  // Halt until something changes
            __no_operation();
          break;
        case 1:
          /* Red LED receives the first byte from
                       the string */
          TA0CCR1 = (UCA0RXBUF * 4);
          break;
        case 2:
          /* Green LED receives the second byte
                       from the string */
          TA0CCR2 = (UCA0RXBUF * 4);
                  break;
        case 3:
          /* Blue LED receives the third byte
                       from the string */
          TA0CCR3 = (UCA0RXBUF * 4);
                   break;
        default:
          // Pass all other bytes to TXBUF
                   while (!(UCA0IFG & UCTXIFG));
          UCA0TXBUF = UCA0RXBUF;
            }

            /* Conditional keeps track of byte handling in
               switch statement */

            // If the current byte is not the final byte...
        if (byte < num_bytes - 1) {
          // ...iterate byte.
          byte++;
        }

        // If the current byte is the final byte...
              else if (byte == num_bytes - 1) {
          /* ...reset byte to get ready for the next
                    package */
          byte = 0;
              }
              break;
      case USCI_UCTXIFG:
            break;
      default:
            break;
    }
}

void initializeGPIO(void)    // Initialize LEDs and timers
{
  // 1 Mhz clock in Up mode
  TA0CTL |= TASSEL_2 + MC_1 + TACLR;
  // 1 kHz frequency
    TA0CCR0 = 1020;

  P1DIR |= RED; //p1.4 to red
  P1SEL |= RED;
  P1DIR |= GRN; //p1.5 to green
  P1SEL |= GRN;
  P1DIR |= BLU; //p1.6 to blue
  P1SEL |= BLU;

  // Initialize LED_RED to OFF
  TA0CCTL1 = OUTMOD_7;       // OUTMODE reset/set
  TA0CCR1 = 0;           //CCR1 Default Zero

  // Initialize LED_GREEN to OFF
  TA0CCTL2 = OUTMOD_7;       // OUTMODE reset/set
  TA0CCR2 = 0;           //CCR2 Default Zero

    //Initialize LED_BLUE to OFF
  TA0CCTL3 = OUTMOD_7;       // OUTMODE reset/set
  TA0CCR3 = 0;           //CCR3 Default Zero
}

```

```
void initializeUART(void)       // from Lab 1 example code
{
  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
}
```

milestone_1.c