WS2812 Controller

HARDWARE ARCHITECTURE SPECIFICATION

Developed By:

Jesse Farrell

Contact:

Jessefarrell92@gmail.com

Table of Contents

1. Introduction 4
2. Design Goals 4
3. Research 4
   3.1. Addressable RGB LED’s 4
   3.2. Crystal Oscillator 6
4. Component Selection 7
5. Schematic 8
6. PCB Layout 9
7. Widget Validation 10
   7.1. External Clock Validation 10
   7.2. Communication Protocol Validation 11
8. Power Dissipation 13

Appendix A – Code Tweaks 14

Table of Figures

Figure 1. Data Transmission Method 5
Figure 2. Data Encoding 5
Figure 3. Pierce Oscillator Using CMOS Inverter 6
Figure 4. Finalized Pierce Oscillator Circuit 7
Figure 5. Final Schematic 9
Figure 6. PCB Layers 9
Figure 7. PCB 3D Model 10
Figure 8. Crystal Oscillator - Stage 1 10
Figure 9. Crystal Oscillator - Stage 2 11
Figure 10. Crystal Oscillator - Stage 3 11
Figure 11. ATTINY13a Pin Toggle 12
Figure 12. WS2812B Encoded 0 12
Figure 13. Communication Protocol Testing 13
Figure 14. Bitstream Scope Capture 13
Figure 15. Power Requirements of 5 LEDs 14
Figure 17. Code Constants 14
Figure 18. Send_Bit Function 15

Revision History

Revision	Release Date	Comments
Rev 1.0	Jan. 18. 2022	First release with basic project overview Missing validation and software development
Rev 1.1	Feb. 13. 2022	Updated PCB layout Added final validation section Added programming comments to Appendix A Grammar fixing
Rev 1.2	Mar. 29. 2022	Final Grammar fixing

1. Introduction

The following Hardware Architecture Specification (HAS) outlines the design process for an addressable RGB controller. RGB LED's can add novel lighting affects to offices, labs, and of course gaming setups. This widget is designed to drive strips of WS2812 IC's; however, similar IC's can be driven with minor firmware tweaks. A colleague of mine introduced me to this IC and pointed out its interesting pulse width modulation (PWM) based single line communication protocol. *I say single line somewhat loosely here, we still need a GND reference*. Interested in this protocol, and the potential addition of RGB to my desk, I purchased a strip of WS2812's and began working through the following design process.

2. Design Goals

The goals for this project are minimal; be able to drive an ambiguous number of LEDs in one of several predefined patterns/colors. The LEDs don't need to be individually addressed; the entire strip may follow a single pattern if desired. Another personal goal is to experiment with crystal oscillators since I've somehow avoided these in my designs until now.

Cost and dimensional constraints are not considered in this design due to the simplicity of the widget.

The final widget should:

- Control any number of WS2812's (within reason <200 etc.)
- Include a crystal oscillator

3. Research

This section covers some basic research for the blocks of this widget. Research was conducted for the RGB IC, and crystal oscillator circuits.

3.1. Addressable RGB LED's

There are several popular RGB LEDs on the market such as the, WS2812 sometimes referred to as NeoPixel, WS2813, PI55TBTPRPGPB, and SK6812. All datasheets are nearly carbon copies of one another with minor timing differences. This needs to be kept in mind while working on the firmware for the project. Although this design will be based on the WS2812, it would be nice to accommodate other variants.

The WS2812 displays a color based on 3x 8bit values, one for each primary color. Resulting in 2^24 or 16777216 different color values. I'll refer to this 24-bit value as a packet. The IC's are cascaded and fed a stream of packets representing the color data for each LED along the chain. As the packets pass through the first IC, the first packet to arrive is received and "removed" from the bitstream and interpreted by that IC. All subsequent packets are passed along to the next IC. This is illustrated in Figure 1.

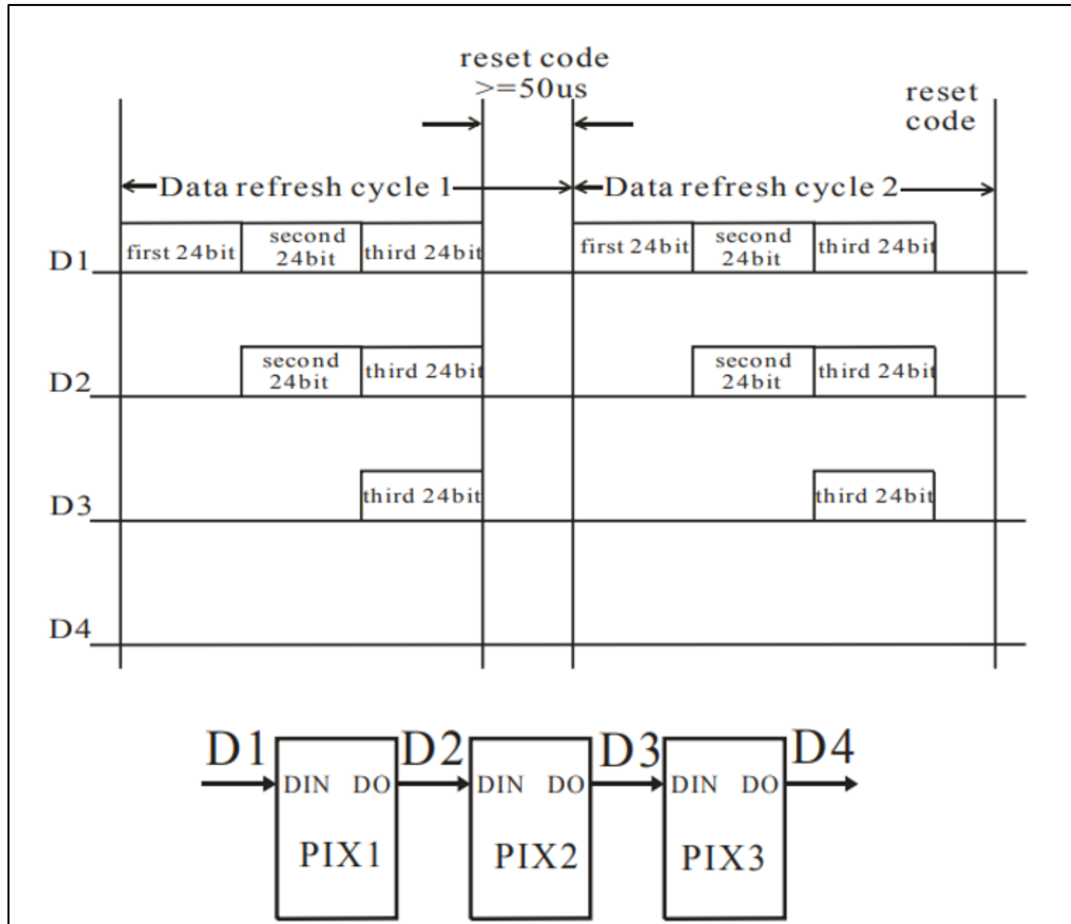

Figure 1. Data Transmission Method

The data is defined as a 1 or 0 based on the high and low times of each cycle. This is best shown in the graphic of Figure 2, using the timings presented in Table 1 as reference. Reset is used to end/start a new bitstream.

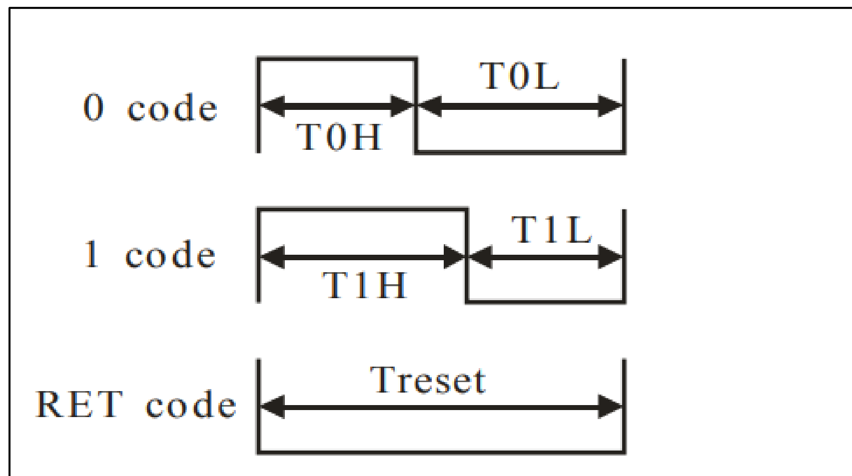

Figure 2. Data Encoding

Table 1. Data Encoding Times

T0H	0 code, high voltage	0.35us	±150ns
T1H	1 code, high voltage	0.7us	±150ns
T0L	0 code, low voltage	0.8us	±150ns
T1L	1 code, low voltage	0.6us	±150ns
RESET	low voltage	>50us	

3.2. Crystal Oscillator

The following section explains the design process for a crystal oscillator, and closely follows the outline provided by TI's application note.

A Pierce Oscillator will be used as the clock source, shown in Figure 3. Note that an additional resistor is sometimes included to isolate C2 from the output of the CMOS inverter. I will be using a 20MHz crystal (datasheet), and unbuffered CMOS inverter (datasheet).

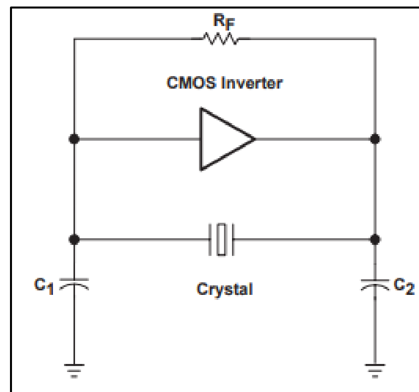

Figure 3. Pierce Oscillator Using CMOS Inverter

C_1 and C_2 should be chosen so that their series capacitance ($C = \frac{C_1 * C_2}{C_1 + C_2}$), is approximately equivalent to the load capacitance specified by the crystal manufacturer. For this 20MHz crystal the load capacitance is 20pF. Choosing $C_1 = C_2$, the recommended value is 40pF (eq.1). However, this value does not consider the capacitance of the PCB, or the input/output capacitance of the inverter. Depending on the board 30pF may be a more stable option (*or I suppose the correct term would be less stable since this is an oscillator*).

$$C = \frac{C_1 * C_2}{C_1 + C_2} = \frac{C_1}{2} => C_1 = C_2 = 2 * 20pF = 40pF \qquad (eq.1)$$

The purpose of R_F is to provide feedback for the inverter. It's typically 1-10MΩ. To calculate its recommended value, we need values for; the capacitance of the leads and electrodes (C_0), the load capacitance (C_L), the resistance at series resonance (R_ω), and lastly the open loop gain of the inverter (α). For this widget the calculation resulted in a recommended feedback resistor of 29kΩ (eq.2), this seems low based on other literature, so I will **ignore this result and use 2.2MΩ** based on the inverter datasheet. *(I think this error is due to the poor open loop gain of unbuffered inverters, I'm suspicious this equation wasn't intended to be used in such instances)*

$$\alpha = 20dBV = 10V/V$$

$$R_F = \left(\frac{1}{R*\omega^2*(C_0+C_L)^2}\right) * \alpha = \left(\frac{1}{30*(2\pi*20*10^6)^2*(7*10^{-12}+20*10^{-12})^2}\right) * 10 = \ 29k\Omega \qquad \text{(eq. 2)}$$

Another resistor (R_S) will be added to isolate C2 and the output of the inverter. Choosing ($R_S \approx X_{C2}$) will attenuate the signal by 50% at the resonant frequency, hence it should dramatically reduce the overshoot at the output. Rs will be 470Ω based on (eq.3).

$$Rs = R_2 = \frac{1}{2\pi*20*10^6*40*10^{-12}} = 398\Omega \qquad \text{(eq. 3)}$$

The final circuit is shown in Figure 4. Circuit validation is presented in section 7.1.

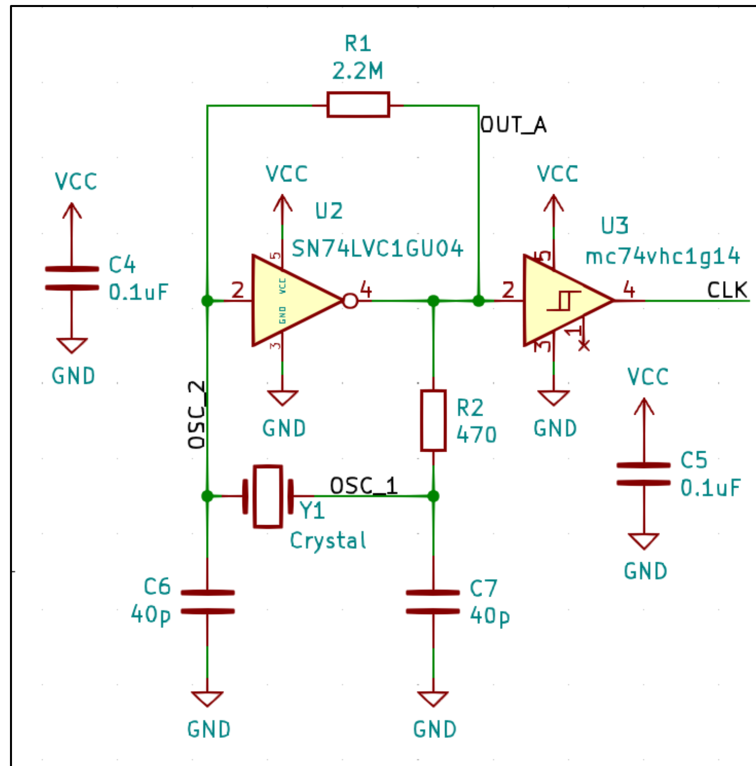

Figure 4. Finalized Pierce Oscillator Circuit

4. Component Selection

Component selection for this widget was primarily driven by my spare parts drawer. Where possible, and unless otherwise stated, components will use 0603 footprints. All bypass capacitors will be X5R or better and rated for at least 12V. A single 25V, 220uF electrolytic bulk capacitor is provided at the power input, along with a TVS diode for transient suppression. The PCB will be powered with a 5V wall adapter (*note that the maximum operating voltage of the MCU is 5.5V so it might be a good idea to do a sanity check of your wall adapter before use*); the power rating will be determined by the length of RGB strip connected to the widgets output. Since each IC can draw up to 60mA (max 20mA per emitter color), the maximum current draw of the widget at 5V is shown in (eq.4), where 'n' is the number of WS2812's being driven.

$$Approx. Current\ Rating\ @5V = 0.5_1 + n * 0.06A \qquad (eq. 4)$$

$$I = 0.5_1 + 300 * 0.06 = 18.5A \qquad (n = 300)$$

Note 1 – budgeted 500mA for the rest of the system and to provide overhead

An entire strip may contain up to ~300 LEDs. Such a system could theoretically require up to 18.5A if all elements were at their maximum brightness. Since this is an unrealistic requirement either (a) the maximum allowable LED strip must be reduced or (b) the color displayed must be well below its maximum brightness. This is investigated in section 8 and was resolved. (*TLDR: with 100 LEDs max current is about 2A, and use* $\boldsymbol{I = 0.035n + 0.02}$ $[\boldsymbol{A}]$ *for PSU selection*)

An Attiny13A will be at the heart of this widget. If the MCU is ran at 20MHz via an external clock, we should be able to manage the timing constraints of Table 1 via “bit-banging”. Alternatively, there is a configurable PWM generator that might come in handy for the single line protocol. The main constricting factor is the 1KB of flash, 64B of EEPROM and 64B of SRAM. There are existing libraries that would theoretically work with this MCU; however, these libraries are too large and would require substantial modification to meet this micro’s memory constraint.

For the crystal oscillator circuit an unbuffered inverter, will drive the quartz-crystal oscillation, and the signal will be cleaned up by a buffered Schmitt trigger inverter. So long as the components are fast enough, any jelly-bean inverter should do the job. This widget will use an SN74LVC1GU04 as the unbuffered inverter and a MC74VHC1G14 for the buffered inverter.

5. Schematic

The final schematic is shown in Figure 5. Unused pins on the MCU were connected to an external dip switch to allow for user configuration. The SPI bus is routed to pads for side mounting 2.54mm male headers (see section 6), these headers can plug into an Arduino Uno for programming (pins 13->10). Note that **when programming the ATTINY all the dip switch inputs must be off**, otherwise some of the bus lines will be pulled to ground.

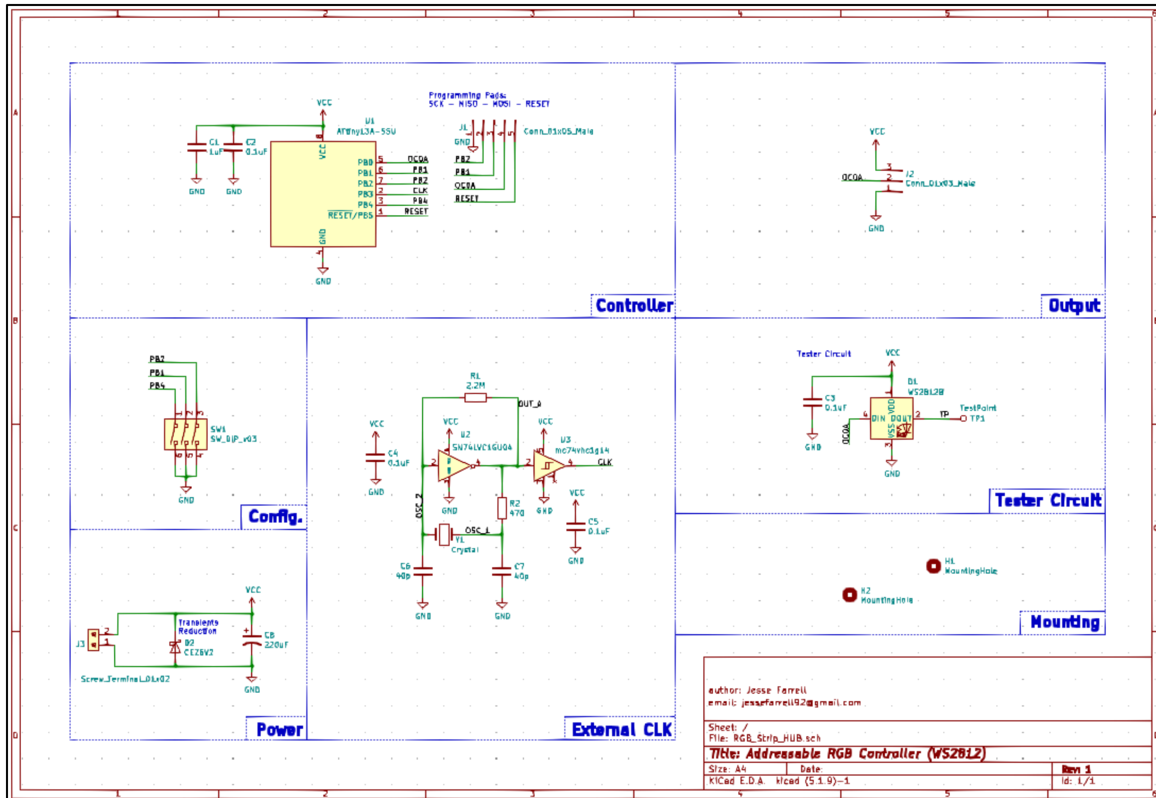

Figure 5. Final Schematic

6. PCB Layout

This widget uses a 2-layer board. All traces and components, aside from the tester circuit, and programming pads, were kept on the top layer. A ground plane was added to the bottom layer, and the remaining area of the top layer was poured for Vcc. The resulting PCB is shown in Figure 6.

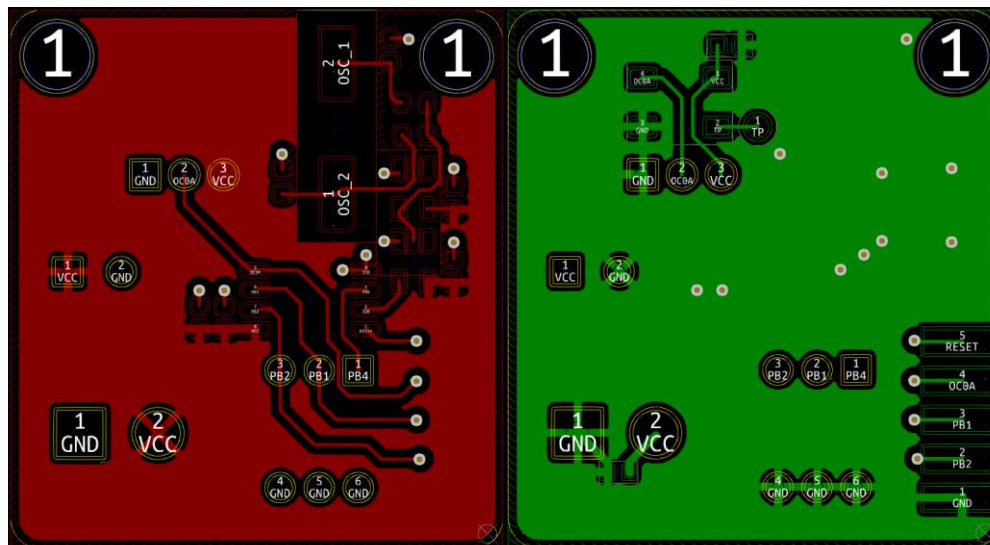

Figure 6. PCB Layers

The 3D model for the layout is presented in Figure 7.

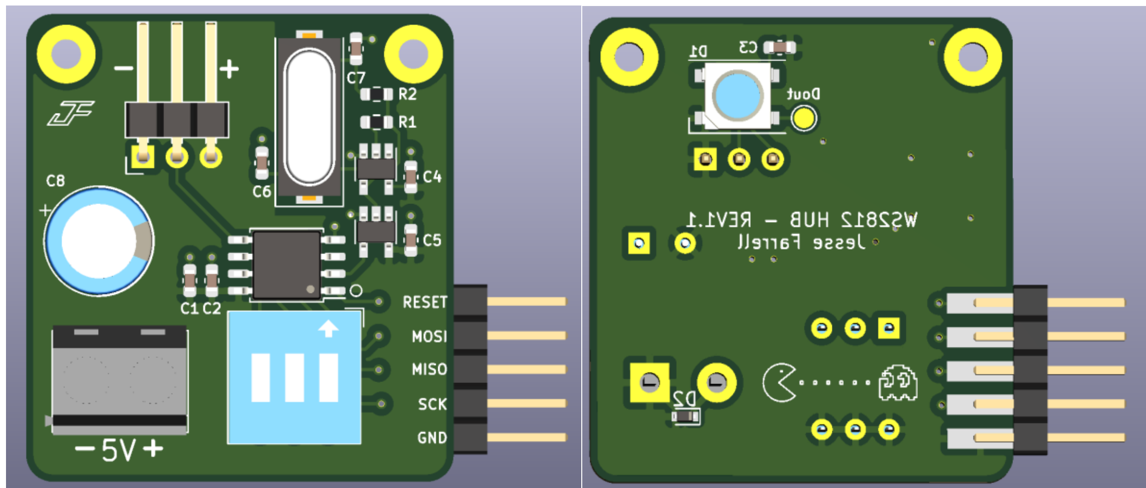

Figure 7. PCB 3D Model

7. Widget Validation

Both the crystal oscillator and the communication protocol need to be tested before firmware development can begin. The crystal oscillator is validated in section 7.1, and the protocol in section 7.2.

7.1. External Clock Validation

The clock source works as expected. At the oscillator is a stable 20MHz cosine with a peak-to-peak amplitude of 3.72V (Figure 8), and after the first unbuffered inverted, the signal is amplified to a square wave with 5ns rise and fall time (Figure 9). After the buffered inverter the signal is tightened up a bit further, and a small amount of overshoot is introduced (Figure 10).

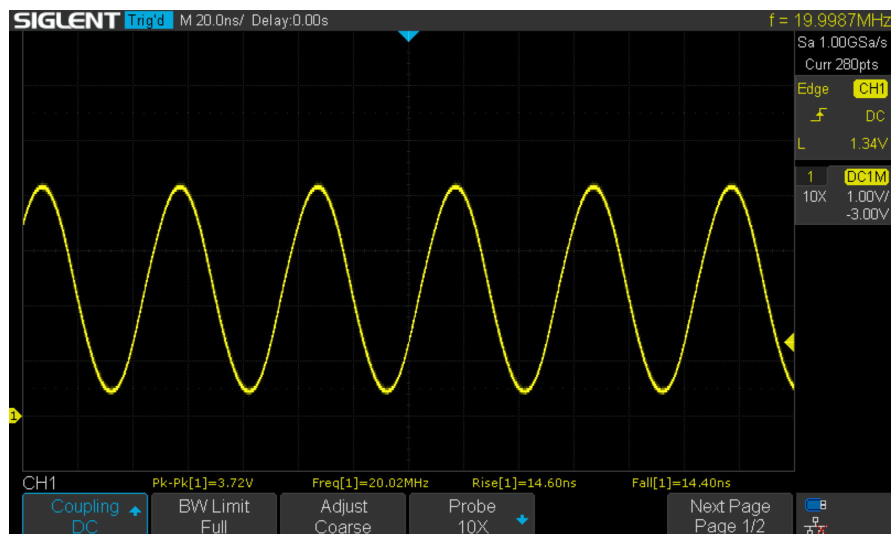

Figure 8. Crystal Oscillator - Stage 1

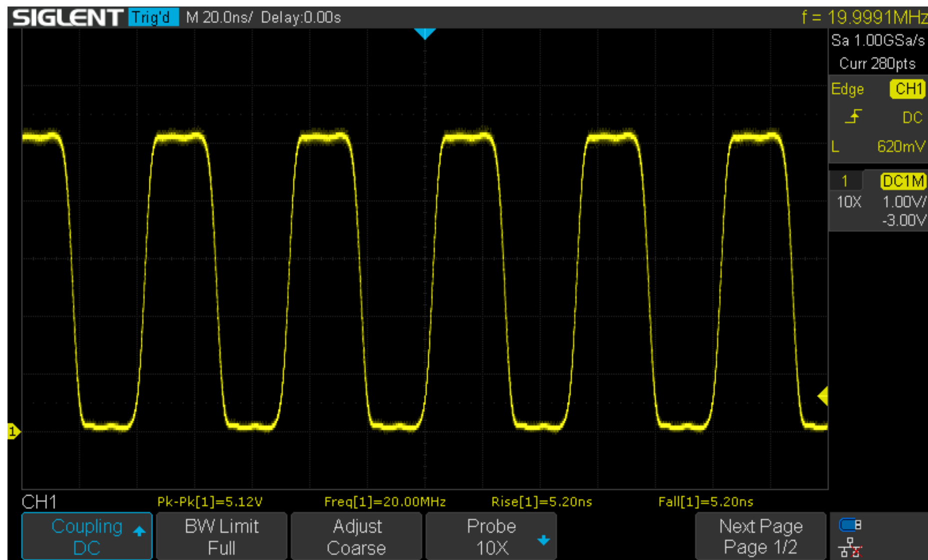

Figure 9. Crystal Oscillator - Stage 2

Figure 10. Crystal Oscillator - Stage 3

7.2. Communication Protocol Validation

After confirming the clock source was stable and configuring my IDE, I was able to program the ATTINY13a. To get started I wanted to understand what I was working with. If I just toggle the pin, I'm able to get ~50ns pulse widths. This makes sense since the clock is 20MHz = 50ns period.

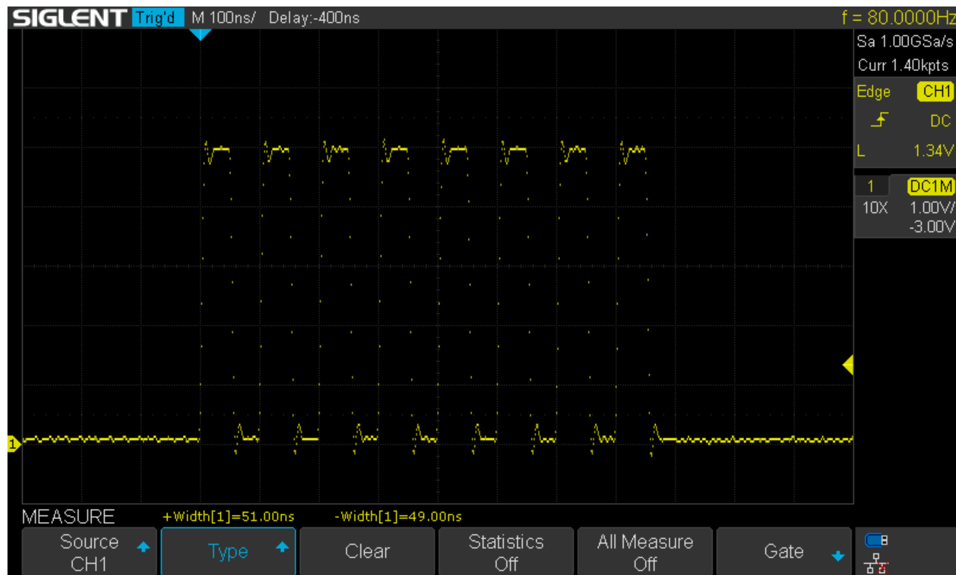

Figure 11. ATTINY13a Pin Toggle

To extend the pulse widths of each toggle I inserted _asm("nop") commands. This is an assembly command telling the MCU to perform no operation. The toggle command will need to be padded with 7/8, and 15/16 "nop" to achieve the timings in Table 1. A hardcoded 0 is presented in Figure 12.

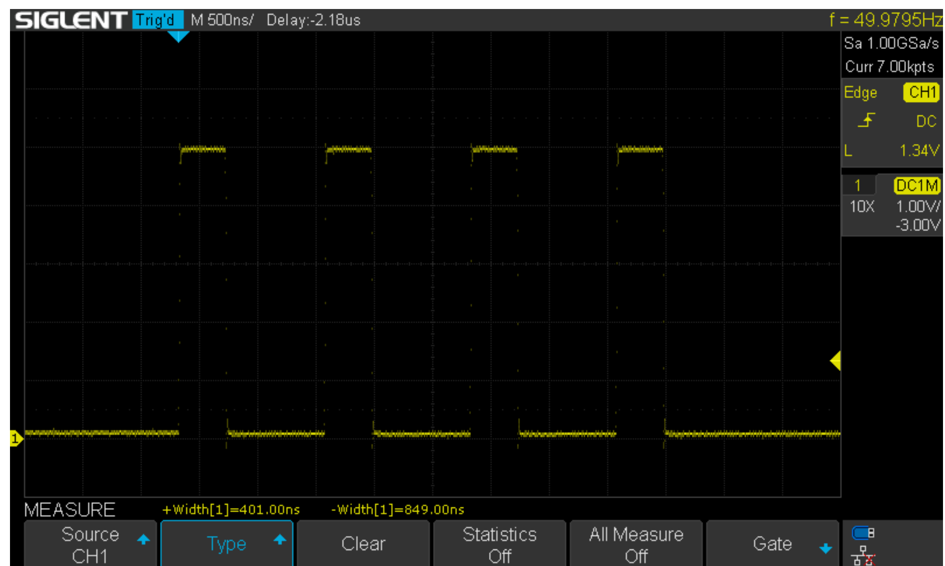

Figure 12. WS2812B Encoded 0

The hardcoded 1 and 0 were then moved into functions and called 24 times to test a single WS2812 LED. Note that the timings had to be tweaked to account for the overhead of entering and exiting the function. The result of this test can be seen in Figure 13.

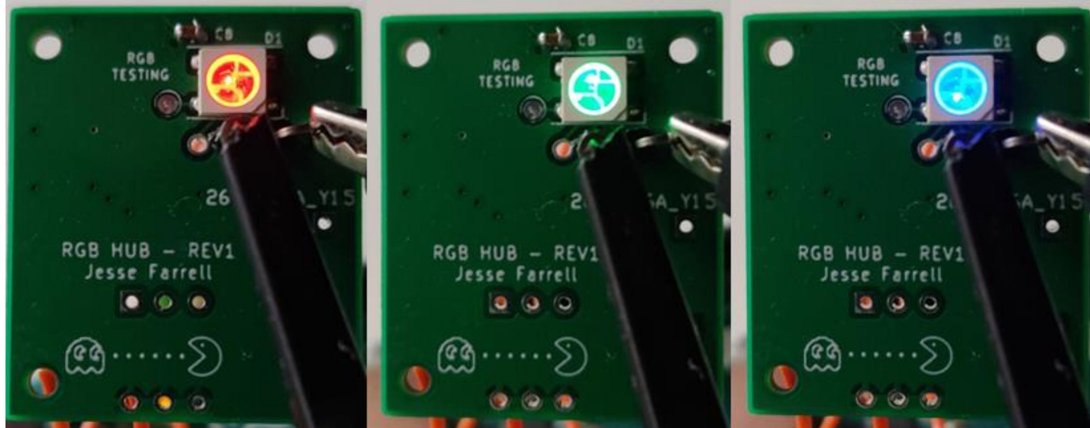

Figure 13. Communication Protocol Testing

Testing the reset command, it looks like the IC follows its datasheet closely. It won't reset below a 50us hold time. In the capture shown in Figure 14 the signal was held low for approximately 50us, but the LED did not update (*cursors were likely a bit offset here*). Increasing the delay to 55us and the IC worked as expected. The datasheets were somewhat unclear what needs to be done at the end of the bitstream. The extended on-time in this capture, was later deemed to be unnecessary.

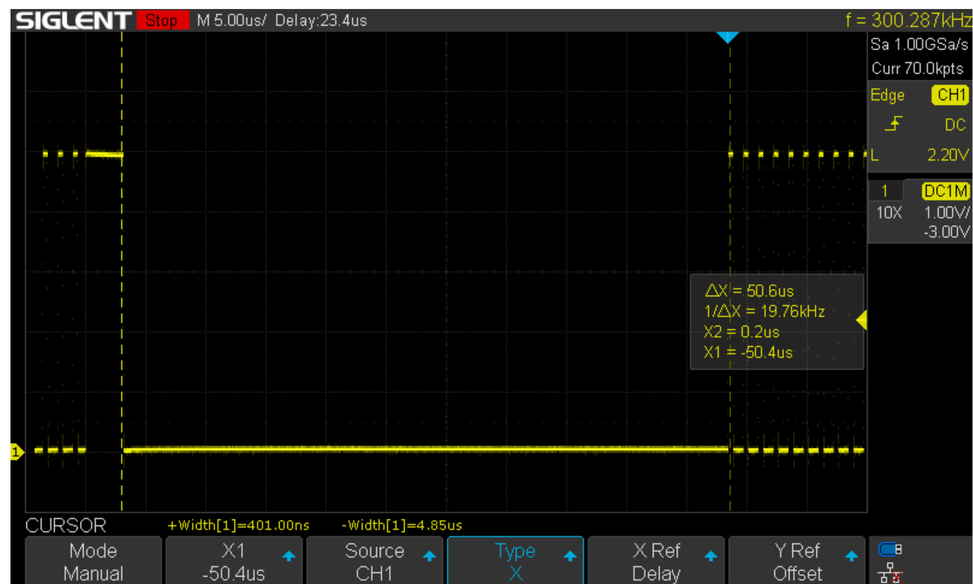

Figure 14. Bitstream Scope Capture

8. Power Dissipation

After developing the firmware, the following power consumption measurements were recorded.

Driving 5, max brightness, LED's at 5V, required 950mW (190mA), and with all LEDs off its about 100mW (20mA). This corresponds to approximately 175mW (35mA) per LED. Note that these are just ballpark values since the PSU used has limited accuracy and precision. See Figure 15.

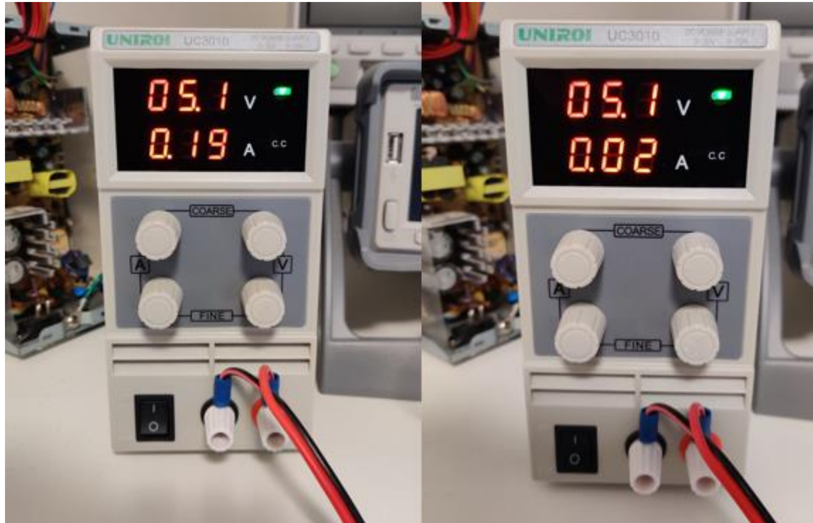

Figure 15. Power Requirements of 5 LEDs

After getting a general idea of the power requirements a string of 100 WS2812s was tested resulting in 15.4W, so about 153mW per LED (adjusted for quiescent). For power supply recommendations I will go with the larger value, 175mW per LED + 100mW of overhead. The final power supply recommendation is presented below.

$\boldsymbol{I = 0.035n + 0.02\ [A]}$ //**5V** supply current rating for 'n' LEDs

Appendix A – Code Tweaks

The code for this widget should be attached in a nearby document folder. There are several defined values that can be tweaked to alter the widgets operation (Figure 16).

- **NUM_LED:** The maximum number of LED's on the string. This is the number of packets that the MCU will attempt to send down the line.
- **MAX_BRIGHTNESS:** Used by the RGB function to define the brightness ceiling.
- **MIN_BRIGHTNESS:** Used by the RGB function to define the brightness floor.
- **RGB_SEED_1:** Changing this value will alter the RGB strobing pattern.
- **CUSTOM_n:** These are default color codes that the user can configure.

```
#define LED_PIN_ON 0xff          //PB0 (0x01)
#define LED_PIN_OFF 0xfe         //need this defined to speed up timings
#define PACKT_SIZE 24            //size of color packet 3x 8bit
#define NUM_LED 300              //Max number LEDs, you can tweak this to slow down color changes
#define MAX_BRIGHTNESS 150       //max value 255!!!
#define MIN_BRIGHTNESS 10        //min value 1!!!
#define RGB_SEED_1 0xf02802      //RGB seed 1 (greens/reds)
#define CUSTOM_0 0xFF0070        //Custom color 0 (Purple/pink)
#define CUSTOM_1 0x00CCDF        //Custom color 1 (cyan)
#define CUSTOM_2 0x00d00f        //Custom color 2 (green)
#define CUSTOM_3 0xFF0000        //Custom color 3 (red)
#define CUSTOM_4 0x150F4F        //Custom color 4 (blue)
#define CUSTOM_5 0x550F6F        //Custom color 5 (dim-purple)
```

Figure 16. Code Constants

If you want to alter this code for different RGB IC's, you'll need to alter the number of _asm("nop") commands for each pulse. Each instruction adds a single clock cycle (so 50ns) to the execution time. Note that the low time on bit 1 is done externally, so is somewhat capped to 450ns.

```
void send_bit(bool value) {
  if (!value) {             //send a zero
    PORTB = LED_PIN_ON;     //want 400ns +width
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );

    PORTB = LED_PIN_OFF;    //want 850ns -width
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );

  }
  else {                    //Send a One
    PORTB = LED_PIN_ON;     //0.80us +width
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );
    __asm( "nop" );

//  PORTB = 0x00;
//  To speed up process I had to do this externally
  }
}
```

Figure 17. Send_Bit Function

Lastly the reset command is primarily configured by a delay_clk(300) at line 79. If your LED defines a different reset timing you may need to tweak this value. Currently my code pulls the bus low for about 68us.