*** Note, I did not write this, I just found it online and uploaded it to the project folder**

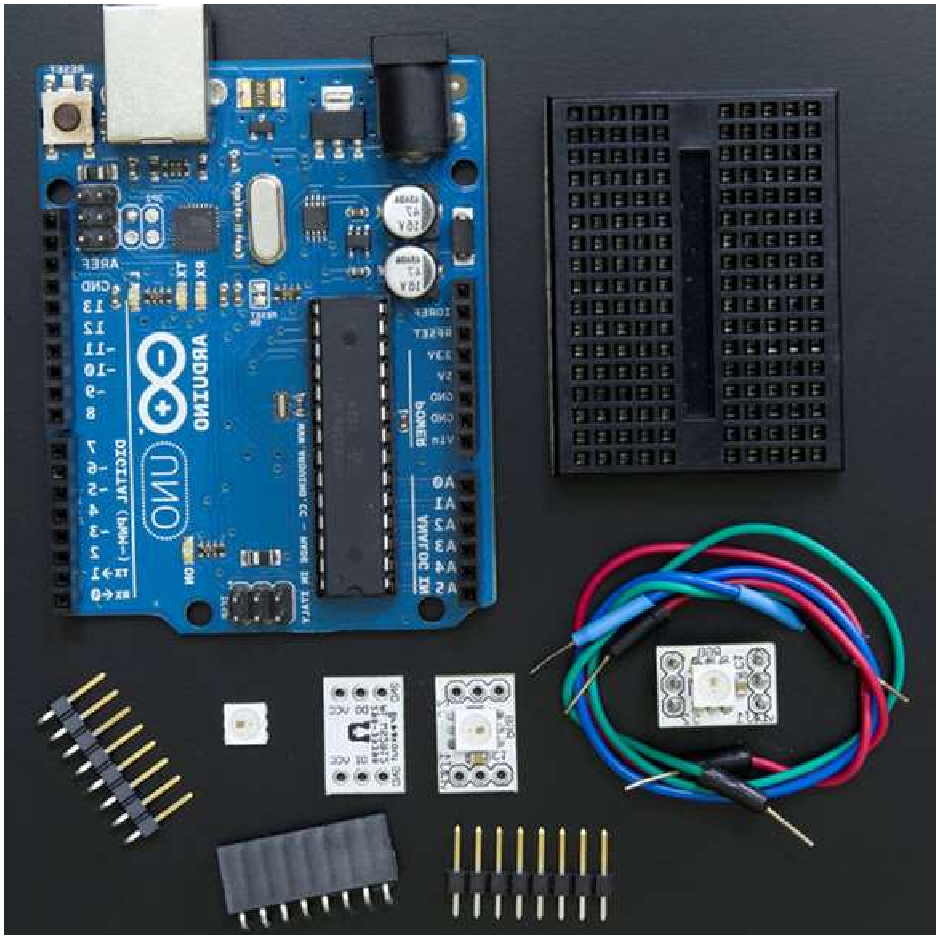
Disclaimer: *over the past year, a few different libraries have been written for controlling these ubiquitous RGB LEDs by fellow makers from Adafruit, PJRC, and the FastSPI project. The libraries work great, and we should try them all out. Recently, we were asked by a few people how the low-level code really worked. With the hope that others find the explanation useful, we put together this Instructable with a detailed answer.*
--

Overview
When we want a microcontroller to send/receive data to/from devices using some form of digital logic, we often do so by way of standard protocols such as SPI, I2C/TWI, UART, etc. However, there comes a time in every embedded hardware programmer's life where it is convenient or necespsary to roll-up her sleeves, and crank-out her own protocol. Such is the case for controlling the ubiquitous RGB LEDs from WorldSemi: the WS281X series.

It should be noted that there have been successful attempts to use the SPI protocol for controlling these LEDs. Nevertheless, given the nature of their communication protocol (described below) this is a perfect setting for implementing a custom solution using a programming technique known as bitbanging. This technique allows us to mimic different functions of specialized hardware using software. In this case, we'll use it to toggle a digital output pin on the ATMega328p microcontroller in a highly precise manner, so that the digital signal created allows us to turn on and off a 1-by-60 array of WS2812 RGB LEDs.

Difficulty level: Beginner+ (some familiarity with Arduino programming)
Time to completion: 15-30 Minutes

Step 1: List of Materials



Inside the WS2812 and WS2812B packages resides an embedded version of the WS2811 constant-current LED driver, as well as 3 individually controlled LEDs; one red, one green, and one blue. In a compact package, the WS2811 includes:
- An internal oscillator
- A signal reshaping and amplification circuit
- A data latch
- A 3-channel, programmable constant current output drive
- 2 digital ports (serial output/input)

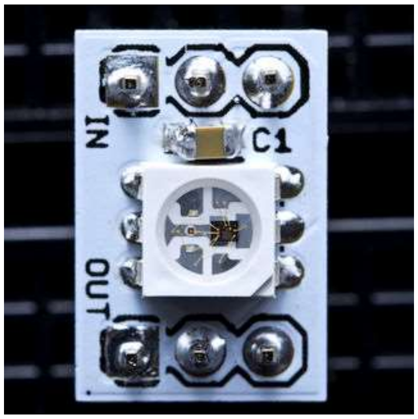
Despite all the intricacies under the hood, we only need to worry about one thing—besides providing power, of course—which is to send ones (1s) and zeros (0s) over to the serial input port according to what we'd like for the LEDs to do. And so, we first need to be clear as to what we want the LEDs to do before moving on to the how we want to do it.

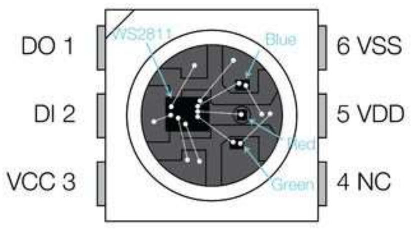
Materials

Required
This Instructable focuses on the firmware to communicate with the WS281X, so all we really need to follow along is a computer running the latest version of the Arduino programming interface (avaliable for OS X, GNU/Linux, and Widnows):
Arduino IDE V1.0.5

Optional
If we want to see the code controlling an actual WS2812 RGB LED, then we need the following parts:
1 x WS2812 RGB LED (pre-soldered onto a tiny breakout board)
1 x Solderless Breadboard
Solid Core Wire (assorted colors; 28 AWG)
1 x Arduino Uno R3
1 x Breakaway Pin Connector, 0.1" Pitch, 8-Pin Male

Step 2: Principles of Operation





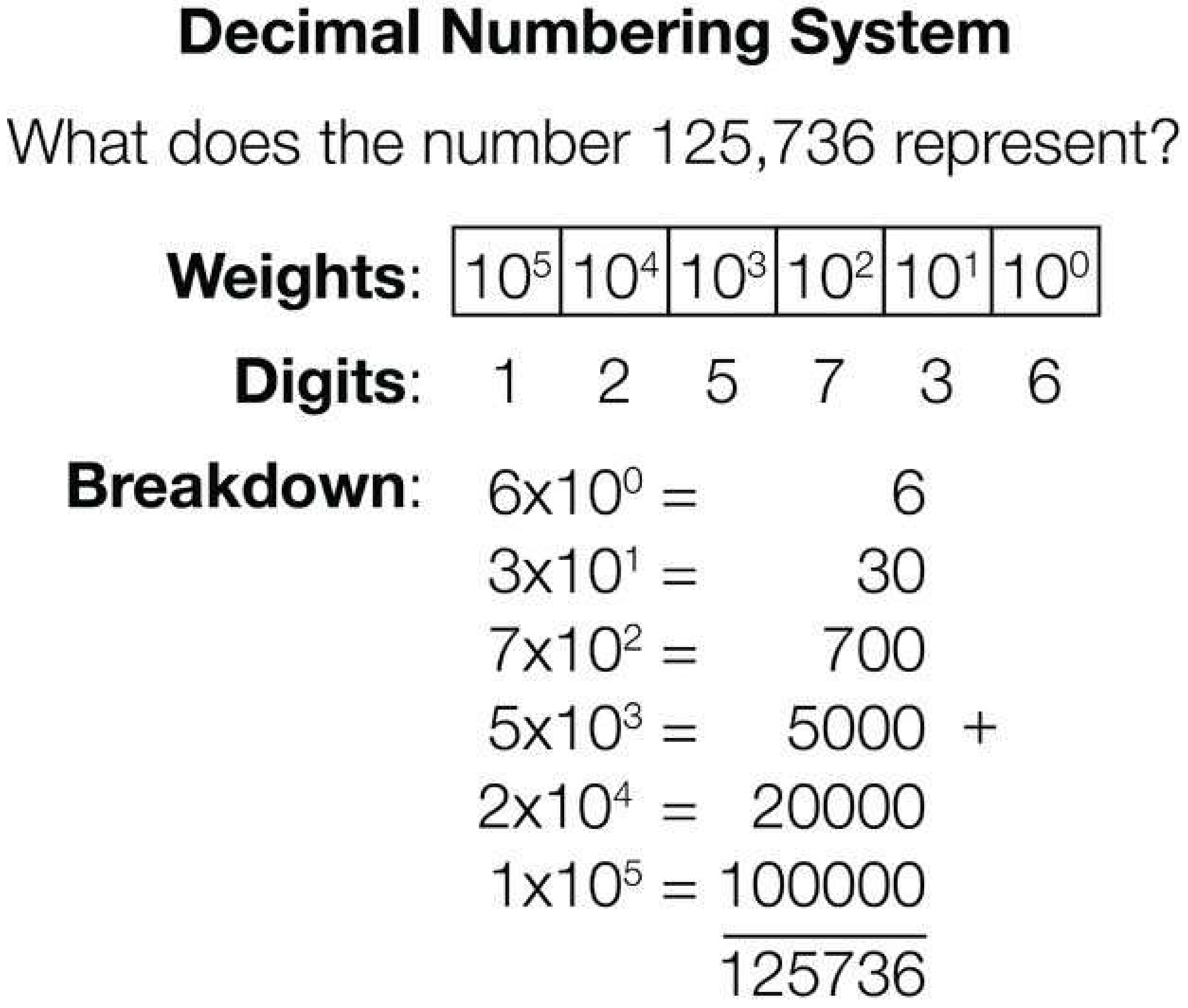
From a high-level, we want to control the intensity and color of the WS2812. Whereas intensity is an intuitive concept (fully on, fully off, and some range of intermediate values), color deserves a short explanation for those unfamiliar with RGB LEDs. As mentioned a few lines ago, the WS2812 contains 3 tiny LEDs that are very close to one another. When we turn them on simultaneously, our eyes perceive a combination of red, green, and blue light, which we interpret as different colors—this is also the principle behind the pixels on our computer screens. By changing the intensity of each LED relative to the other two, we can get a wide range of colors; for example, if we set the red, green, and blue LEDs to their maximum intensity our eyes perceive a whitish color.

We have a good high-level view of what we'd like the LEDs to do, but we need to translate it into something that the WS2812 can understand. It turns out that this is not difficult to do, and is similar to how colors work on most digital displays (e.g., the screen on which you're reading this!). The intensity of all 3l LEDs inside the WS2812 can be independently set to a value ranging from 0 (fully off) to 255 (fully on). So to set the color to whitish as mentioned above, we need to tell the embedded WS2811 driver chip:

“Hey! Set the red LED to an intensity of 255, the blue to an intensity of 255, and the green to an intensity of 255.” (as illustrated by the video demo below)

But how exactly are we going send this message to the WS2811? We need to delve a little bit into digital logic (pun intended) to know exactly how to communicate these and any other allowable intensity values. After a couple of steps, we'll be able break down these values into their constituent 1s and 0s, and send them serially to the digital input port of the WS2811.

Step 3: From Decimal to Binary: Breaking Down Numbers into 1s and 0s



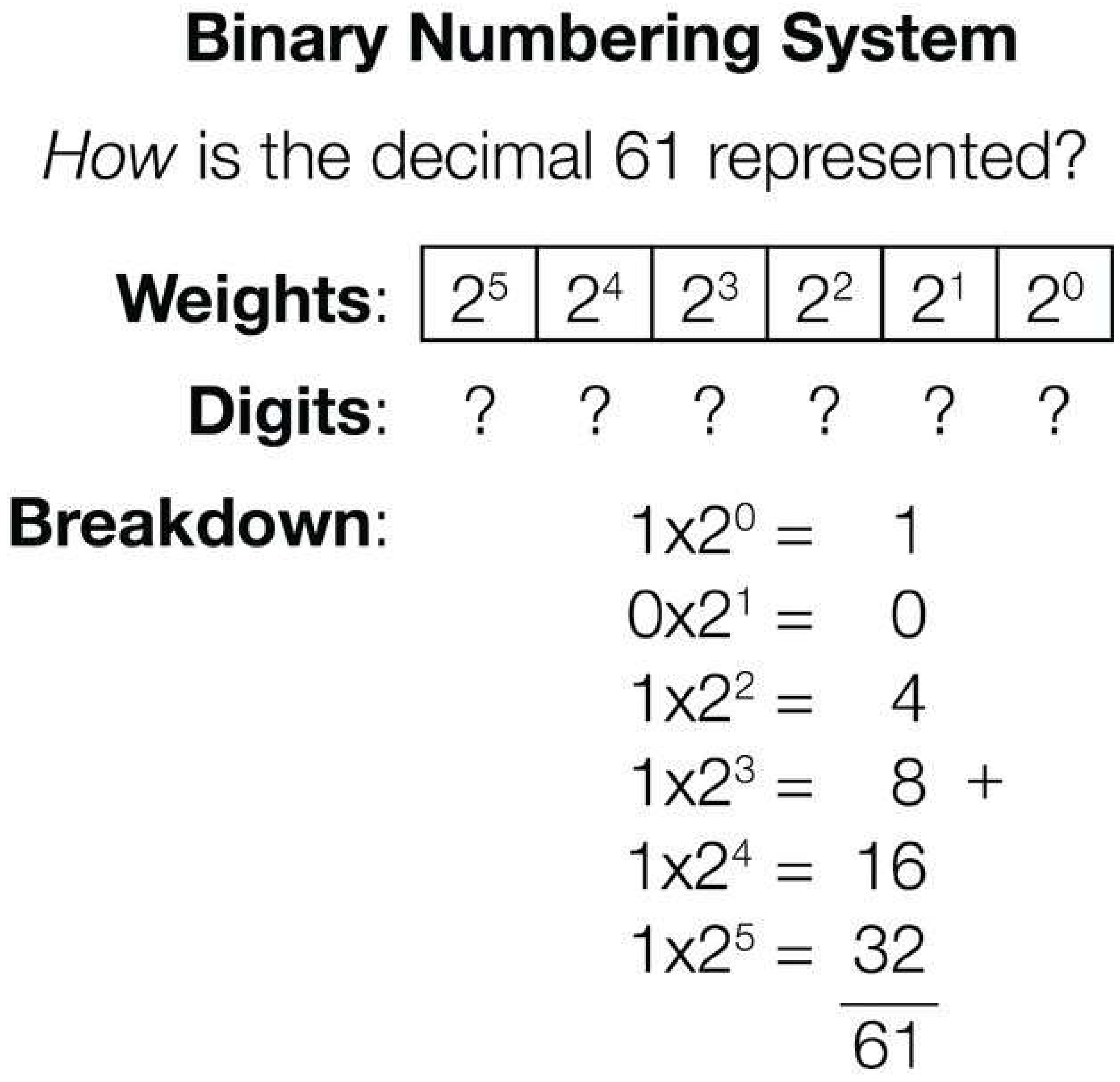
Decimal Numbering System

What does the number 125,736 represent?

Weights:	10^5	10^4	10^3	10^2	10^1	10^0
Digits:	1	2	5	7	3	6

Breakdown:

$$\begin{aligned}6\times10^0 &= 6\\ 3\times10^1 &= 30\\ 7\times10^2 &= 700\\ 5\times10^3 &= 5000 \quad +\\ 2\times10^4 &= 20000\\ 1\times10^5 &= \underline{100000}\\ & 125736\end{aligned}$$



Binary Numbering System

How is the decimal 61 represented?

Weights:	2^5	2^4	2^3	2^2	2^1	2^0
Digits:	?	?	?	?	?	?

Breakdown:

$$\begin{array}{rrl} 1\text{x}2^0 = & 1 & \\ 0\text{x}2^1 = & 0 & \\ 1\text{x}2^2 = & 4 & \\ 1\text{x}2^3 = & 8 & + \\ 1\text{x}2^4 = & 16 & \\ 1\text{x}2^5 = & 32 & \\ \hline & 61 & \end{array}$$

Breaking a number down into 1s and 0s really means using its binary representation. We need to remember that in a binary representation (e.g., 1101 in binary represents the number 13 in decimal), each position has a 'weight' that increases from right to left by a power of 2. Starting with the first position on the right, the 'weights' are: 2^0, 2^1, 2^2, 2^3... This is analogous to the decimal system where starting with the first digit on the right, the 'weights' increase by a power of 10: 10^0, 10^1, 10^2, 10^3... Different than the decimal system where we can have any number from 0...9 at each position, in the binary system we can either have a 1 or a 0.

Say we want to find the binary representation of the decimal number 23. We first notice that 23 is a combination of the number 3 set in the 10^0 position, and the number 2 set in the 10^1 position, which means that when we weigh each number according to its position (2*10^1+3*10^0) we get the number 23. If we tried to do the same in binary we would come up with the number 10111 because 1*2^4+0*2^3+1*2^2+1*2^1+1*2^0 = 23. Of course, finding the binary representation of a relatively small number such as 23 can be done without much calculation. But for larger numbers it becomes necessary to use the following algorithm:

- Increasing from 2^0, find the first power of 2 that's larger than the decimal number we have
- Starting with the power of 2 immediately below the one we found in the first step, divide the decimal number by the powers of 2 in decreasing order, all the way down to 2^0

- After each division step, we should get either a 1 or a 0 as the quotient, and some remainder value. The remainder eventually should go to 0 (this can occur prior to the last division step)
- The 1s and 0s obtained as the quotients give the binary representation of the decimal number

Well, if we haven't gone through the process before, it all sounds like gibberish. Nothing like going through an example to clear things up. Say we want to find the binary representation of the decimal number 117. Let's try to follow the algorithm above (I'll use some personal tweaks):

- We start with 2^0 which is smaller than 117, so we keep increasing. 2^1 is also smaller, keep going... Okay so we get to 2^5 which is still smaller than 117, but as soon as we hit 2^6 we notice that it is the "first power of 2 that's larger than the decimal number we have"
- So we know we need to start with the power of 2 immediately below 2^6, which is 2^5. [Personal tweaks] since we know that we're going to be dividing by all powers of 2 below 2^5, I write them all down beforehand so I don't forget. I also remind myself that the remainder of the division should end in 0 (although it could turn into 0 along the way.

Divider	Remainder	Quotient
2^6 (64)		
2^5 (32)		
2^4 (16)		
2^3 (8)		
2^2 (4)		
2^1 (2)		
2^0 (1)		
	0	

-With everything set, we start the division steps:

Divider	Remainder	Quotient
2^6 (64)	117	1
2^5 (32)	53	
2^4 (16)		
2^3 (8)		
2^2 (4)		
2^1 (2)		
2^0 (1)		
	0	

117 divided by 64 gives a quotient of 1 and a remainder of 53. Thus we know that in the 7th position of our binary representation of 117, there'll be a 1 (i.e., 1XXXXXX). To get the other positions we simply continue the division process:

Divider	Remainder	Quotient
2^6 (64)	117	1
2^5 (32)	53	1
2^4 (16)	21	1
2^3 (8)	5	0
2^2 (4)	5	1
2^1 (2)	1	0
2^0 (1)	1	1
	0	

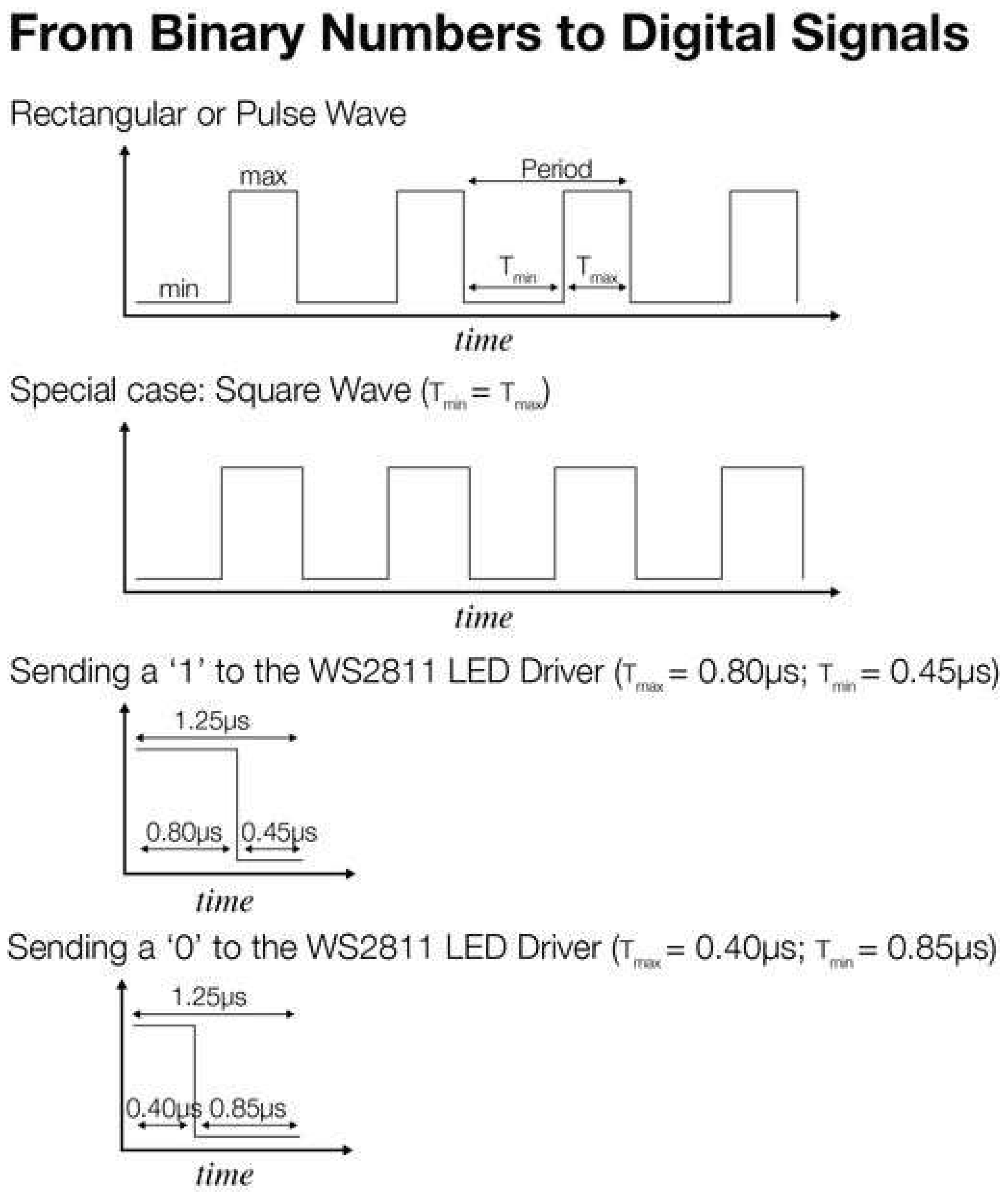
- And so, we get that the binary representation of the decimal number 117 is 1110101. We need to remember, that even if the remainder goes to 0 before the last division step, we need to continue the process all the way down to 2^0. Thus, in the case of the decimal number 48:

Divider	Remainder	Quotient
2^5 (32)	48	1

2^4 (16)	16	1
2^3 (8)	0	0
2^2 (4)	0	0
2^1 (2)	0	0
2^0 (1)	0	0
	0	

The binary representation is 110000 as opposed to 110, which we would get if we stopped diving when the remainder first reached 0. Knowing how to break down a number into its constituent 1s and 0s is very much necessary for being able to transmit data to the WS281X.

Step 4: From Binary Numbers to Digital Logic


From Binary Numbers to Digital Signals
Rectangular or Pulse Wave
max
Period
min
T_{min}
T_{max}
time
Special case: Square Wave ($T_{min} = T_{max}$)
time
Sending a '1' to the WS2811 LED Driver (T_{max} = 0.80μs; T_{min} = 0.45μs)
1.25μs
0.80μs
0.45μs
time
Sending a '0' to the WS2811 LED Driver (T_{max} = 0.40μs; T_{min} = 0.85μs)
1.25μs
0.40μs
0.85μs
time

Okay, so now that we're familiar with the binary representation of decimal numbers we can communicate the intensity values we want to the WS2811 LED driver IC. Since the values go from 0 to 255 for each LED, we will need 8 positions (called bits in digital logic) to cover the entire range—255 is 11111111 in binary. And, we'll need 24 bits to transmit the values for all 3 LEDs inside each WS2812. But how exactly can we tell the WS2811 that we want a 0 or a 1. Well, it turns out that we need to manipulate the timing of a square wave signal to do this.

Disclaimer: *There is a small variation of the timing described below depending whether you're using an actual WS2811 IC, or the embedded version inside the WS2812/WS2812B. The numbers used below correspond to the latter case (WS2812/WS2812B). If you're using the WS2811 IC then consult the* datasheet *for the slightly different numbers (other than that, everything else described below is the same).*

Principle of operation
The WS2811 expects two things:
1) A pulse (i.e., rectangular) wave signal with a frequency around 800KHz—other frequencies work as well, but we'll stick to 800KHz in this tutorial—that sets the intensity values in an internal shift register. Let's note however, that the WS2811 behaves differently than a standard shift register in that the data are shifted in a First-In Last-Out fashion.
2) After the data are shifted into place, the WS2811 expects a low signal lasting at least 50μs in order to latch the data to their respective outputs.

Shifting the data
Those unfamiliar with the term 'pulse wave' might have heard of its special case: the square wave. These type of non-sinusoidal signals consist of an alternating amplitude between a fixed maximum and a fixed minimum at a constant frequency. When the alternation occurs symmetrically, that is, when the time during which the signal has a maximum value is identical to the time during which the signal has a minimum value, then we have the special case of a square wave. At around 800KHz, each period of the pulse wave is around 1.25μs long (1/1.25μs = 800KHz). For communicating with the WS2811 we need to adjust the time during which the signal is either high or low in order to signal a 0 or a 1. There's a mistake in the datasheet from WorldSemi, so the real values should be (credit to the folks over at Adafruit for catching this):

Transmitting a 1:
Time for the signal to remain high (T1H): 0.8μs
Time for the signal to remain low (T1L): 0.45μs

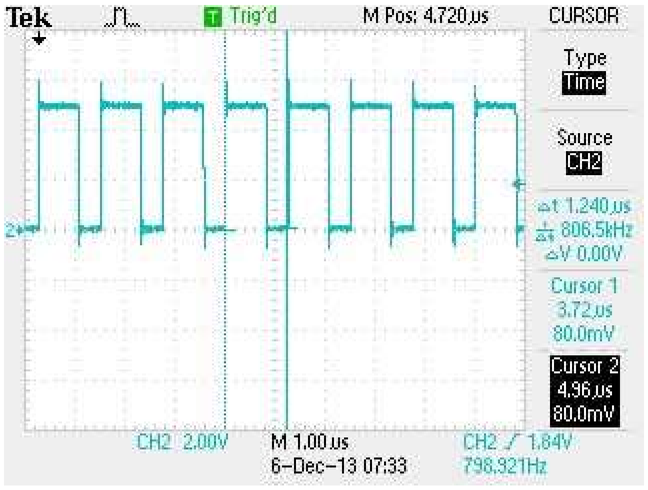
Transmitting a 0:
Time for the signal to remain high (T0H): 0.4μs
Time for the signal to remain low (T0L): 0.85μs

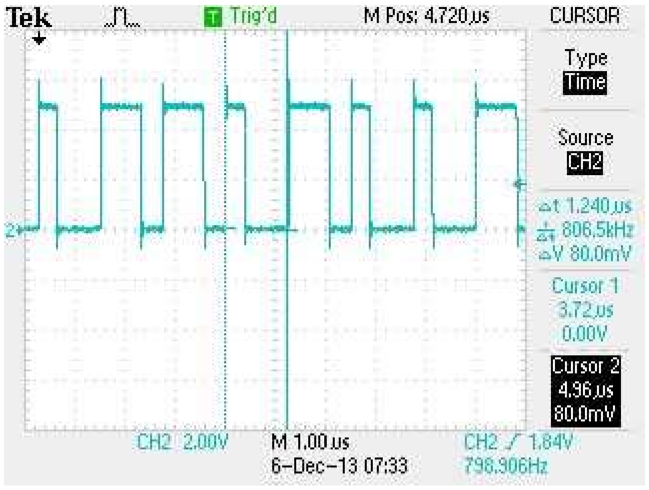
Latching the data
After sending all the bits corresponding to the intensity values of all the LEDs that we want to control, then we need simply hold the value of the pulse wave at its minimum value for at least 50μs.

Transmitting a 'latch command':
Time for the signal to remain low (TL): >=50μs

This type of signal has the special properties of being self-clocked, and non-zero return (NZR). So, what remains is to see how we can set our ATMega328p to produce a precisely timed signal so that we can transmitting to the array of WS2812 RGB LEDs. [Spoiler alert!] We'll be using the bitbanging technique.

Step 5: Bitbanging a Pulse Wave on an ATMega328p Microcontroller





One of the advantages of using a microcontroller as opposed to, say a computer's CPU, is that we have a very tight control over the timing of the instructions we program into it. In fact, to show how precise that control can be, we'll be using assembly instructions instead of the typical high-level functions such as digitalWrite. The use of assembly allows us to know exactly how many clock cycles are taken up during the execution of each instruction.

Since the Arduino Uno R3 development board maintains a 16MHz external clock signal on the onboard ATMega328p, the microcontroller executes a 1 clock-cycle instruction in exactly 62.5ns (1/16MHz = 62.5ns). Since we can find out how many clock cycles each instruction takes, we can precisely control how many instructions we need to generate our signal.

As we saw previously, in order to transmit a 1 to the WS281X chip we need to transmit a signal that stays at a maximum (HIGH) value for 0.8µs, and then stays at a minimum (LOW) value for 0.45µs. Thus, we want to write a list of instructions that:

- Set digital pin to HIGH
- Wait 0.8µs
- Sets digital pin to LOW
- Waits 0.45µs

In assembly language, this can be achieved by the following code:

```
 asm volatile(
  // Instruction       Clock   Description   Phase     Bit Transmitted
  "sbi  %0, %1\n\t"  // 2     PIN HIGH      (T =  2)
  "rjmp .+0\n\t"     // 2     nop nop       (T =  4)
  "rjmp .+0\n\t"     // 2     nop nop       (T =  6)
  "rjmp .+0\n\t"     // 2     nop nop       (T =  8)
  "rjmp .+0\n\t"     // 2     nop nop       (T = 10)
  "rjmp .+0\n\t"     // 2     nop nop       (T = 12)
  "nop\n\t"          // 1     nop           (T = 13)
  "cbi   %0, %1\n\t" // 2     PIN LOW       (T = 15)
  "rjmp .+0\n\t"     // 2     nop nop       (T = 17)
  "rjmp .+0\n\t"     // 2     nop nop       (T = 19)
  "nop\n\t"          // 1     nop           (T = 20)     1
  ::
  // Input operands
```

```
   "I" (_SFR_IO_ADDR(PORT)), //%0
   "I" (PORT_PIN)          //%1
 );
```

Instruction
The first column includes the assembly instruction followed by a linefeed and tab characters, which make the final assembler listing generated by the compiler more readable.

Clock
The second column shows the number of clock cycles each instruction takes. For this set of simple instructions there is only one possible value, we'll see later how some instructions (e.g., conditional) may have 1, 2, or 3 possible values. Remember that each clock cycle on the 16MHz Arduino Uno takes 62.5ns.

Description
The third column shows a very brief description of what each operation does.

Phase
Using the term a bit loosely, we use it to indicate the cumulative sum of clock cycles taken by the instructions that have been executed thus far.

In order to send a single 255 value—11111111 in binary—to the WS281X we need to repeat this set of instructions 8 times. In addition, if we insert a 50µs (or greater) pause between transmissions of the 8-bit sequence, the WS281X latches the transmitted data to its output register. Once the data are latched, the first LED (green) of the WS281X should turn on to a maximum brightness level. The Arduino sketch inside bitbang_255.zip demonstrates this operation.

To send a 0 we need to change the code that produces a 1 by decreasing the time during which the signal has a HIGH (maximum) value, and increasing the time during which the signal is at a LOW (minimum). In addition, we should note that the values to each LED should always be specified using 8 bits. For instance, if we wanted to send a value of 105—1101001 in binary—we would need to send the 8 bits 01101001 including the leading 0. The code that produces a 0 looks like:

```
 asm volatile(
   // Instruction      Clock   Description   Phase     Bit Transmitted
   "sbi  %0, %1\n\t"  // 2      PIN HIGH      (T =  2)
   "rjmp .+0\n\t"      // 2      nop nop       (T =  4)
   "rjmp .+0\n\t"      // 2      nop nop       (T =  6)
   "cbi   %0, %1\n\t" // 2      PIN LOW       (T =  8)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 10)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 12)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 14)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 16)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 18)
   "rjmp .+0\n\t"      // 2      nop nop       (T = 20)     0
   ::
   // Input operands
   "I" (_SFR_IO_ADDR(PORT)), //%0
   "I" (PORT_PIN)          //%1
 );
```

We can use the Arduino sketch inside bitbang_105.zip to generate the signal whose image can be seen on the oscilloscope screen captures that are attached to this step.

Now, for the WS281X to display the whitish color we want, we need to send not one but three 255 values—in which case our signal consists of 24 ones—before waiting the 50µs for the data to latch. We could do this by copy-pasting the eleven assembly instructions above 23 times (you can give it a try modifying the bitbang_255.ino sketch). But the code would be impractical for sending values to more than one WS281X chips. A better solution would be to write a loop that would iterate through the 8-bit values until all three of them have been sent.

The sketch inside bitbang_whitish.zip includes a clear description of the steps taken to achieve the desired outcome. The main section, written in assembly following the logic described above, looks as follows:

```
 asm volatile(
  //  Instruction       Clock   Description                        Phase
  "nextbit:\n\t"           // -    label                           (T =  0)
   "sbi  %0, %1\n\t"     // 2    signal HIGH                      (T =  2)
   "sbrc %4, 7\n\t"      // 1-2  if MSB set                       (T =  ?)
    "mov  %6, %3\n\t"  // 0-1   tmp'll set signal high           (T =  4)
   "dec  %5\n\t"         // 1    decrease bitcount                (T =  5)
   "nop\n\t"             // 1    nop (idle 1 clock cycle)         (T =  6)
   "st   %a2, %6\n\t"    // 2    set PORT to tmp                  (T =  8)
   "mov  %6, %7\n\t"   // 1    reset tmp to low (default)     (T =  9)
   "breq nextbyte\n\t"  // 1-2  if bitcount ==0 -> nextbyte  (T =  ?)
   "rol  %4\n\t"         // 1    shift MSB leftwards              (T = 11)
   "rjmp .+0\n\t"         // 2    nop nop                         (T = 13)
   "cbi   %0, %1\n\t"    // 2    signal LOW                       (T = 15)
   "rjmp .+0\n\t"         // 2    nop nop                         (T = 17)
   "nop\n\t"             // 1    nop                              (T = 18)
   "rjmp nextbit\n\t"     // 2    bitcount !=0 -> nextbit         (T = 20)
  "nextbyte:\n\t"         // -    label                              -
   "ldi  %5, 8\n\t"      // 1    reset bitcount                   (T = 11)
   "ld   %4, %a8+\n\t" // 2    val = *p++                         (T = 13)
   "cbi   %0, %1\n\t"   // 2    signal LOW                        (T = 15)
   "rjmp .+0\n\t"        // 2    nop nop                          (T = 17)
   "nop\n\t"            // 1    nop                               (T = 18)
   "dec %9\n\t"         // 1    decrease bytecount                (T = 19)
   "brne nextbit\n\t"    // 2    if bytecount !=0 -> nextbit     (T = 20)
   ::
 );
```

The best way to understand the operation of this section is to consider different case scenarios, and follow the assembly code line by line. For instance, we know that in order to send a value of 255, we need to send 8-bits with a timing corresponding to a 1. In other words, the Digital Pin connected to the WS281X should remain HIGH for 13 cycles (0.8125µs), and LOW for 7 (0.4375µs). Does the code above achieve this? Let's see what happens when we first start transmitting:

```
asm volatile(
  "nextbit:\n\t"          // This is only a label for directing the jumps below.
   "sbi  %0, %1\n\t"    // The signal is set to HIGH, instruction uses 2 cycles.
```

```
    "sbrc %4, 7\n\t"       // True. Sending 255 implies current MSB is 'set' (=1).
     "mov  %6, %3\n\t"  // This is executed. “tmp” is set to HIGH.
    "dec  %5\n\t"          // Bit is being transmitted, decrease bit counter.
    "nop\n\t"                  // Need to idle for getting to the 13 clock cycles.
    "st   %a2, %6\n\t"    // Write the “tmp” value to the PORT (pin still HIGH).
    "mov  %6, %7\n\t"   // Set “tmp” to low for the next pass through the loop.
    "breq nextbyte\n\t"  // False. Bit counter isn't 0, use 1 cycle and continue.
    "rol  %4\n\t"            // Shift the byte value MSB leftwards.
    "rjmp .+0\n\t"          // Idle for 2 clock cycles. Phase reached T = 13.
    "cbi   %0, %1\n\t"    // Set signal to LOW.
    "rjmp .+0\n\t"          // Idle for 2 clock cycles.
    "nop\n\t"                  // Idle for 1 clock cycle.
    "rjmp nextbit\n\t"     // Bit counter wasn't 0 so jump to next bit. T = 20.
  );
```

So the instructions that actually get executed generate a signal on the data pin that is 13 cycles HIGH (0.8125µs) and 7 LOW (0.4375µs), thus sending a bit with a value of 1 to the WS281X. If we continue to study what the code does when the rest of the bits are sent, and what it does when values other than 255 are used, we'll get a deeper understanding of this particular implementation of bitbanging.

I personally hope that you find this tutorial useful for getting started with bitbanging your own communication protocols whenever it's necessary!