

Controlling a strip of NeoPixel LEDs with LabVIEW, RoboRIO, and RIOduino via I2C

Peter Rifken, Mentor, Team 3958

Introduction

The Arduino and RoboRIO are both controllers running programs on their respective processors. Both are capable of reading sensors, making decisions, and outputting results to output ports. Useful bundles of code have been written for both platforms (some only on RoboRIO, others only on Arduino). The question is, how can we extract the most power from each? Can we make decisions in one, while outputting results in the other? Absolutely, and we do this by having both platforms agree to exchange information using a common language, or protocol, such as I2C or UART.

This guide is comprehensive because its goal is not only success with the tools utilized, but a deeper understanding of why the tools work and how to build on the knowledge to tackle more complex projects. Many lessons learned have been compiled to save you time in your own learning journeys.

Goal

- Complete control an individually addressable LED strip, driven by an Arduino, from the LabVIEW Environment
- Introduction to I2C, how to use it to establish communication between two different hardware and software platforms

What you'll need

- Arduino Microcontroller - [RIOduino](#) from Rev Robotics is used in this paper to leverage the MXP Expansion Port on the RoboRIO, though an [Arduino Uno](#) will work
- NeoPixels - Tests in this paper conducted with the [White 30 LED](#)
- FRC Control System - See [Layout](#), though you can get away with the [RoboRIO](#) and a [5V, 2A power supply](#) to drive the LED strip.
- Soldering Equipment
- 1000uF capacitor (Surge protection)
- 300-500 Ohm Resistor

Helpful Reading & Resources Used

- [Powering NeoPixels](#)
- [NeoPixel Arduino Library](#)
- [Master/Slave Technology - Wikipedia](#)
- [Master Writer / Slave Receiver](#) (How to send data from the RoboRIO to the Arduino via I2C)
- [Master Reader / Slave Sender](#) (How to send data from the Arduino to the RoboRIO via I2C)
- [Sparkfun – What is I2C?](#)
- [RIOduino User Manual](#)
- [FRC Software Setup \[2015\]](#)
- [Example: Touchscreen I2C Slave](#) ([Link](#) to PDF download)
- [For Loops & While Loops](#)

This guide focuses on helping the reader understand how to develop the communications interface between a Master controller (in this case, the RoboRIO robot “brain”), and Slave receiver (in this case, the Arduino). We are interested in establishing this communication to leverage the benefits of each development platform:

- **Arduino** – Controlling a NeoPixel from scratch is possible, though allegedly very difficult. Thus, we want to re-use the NeoPixel libraries (see: [what is a library?](#)) developed for the Arduino. What we don’t want is to constantly be developing in two environments.
- **RoboRIO**– Where most of your robot’s code will reside and where the programmer will spend most time developing (Rather, LabVIEW, Java, or a C++ IDE). Thus, it’s convenient to be able to create “hooks” to control the LED strip from those primary environments.

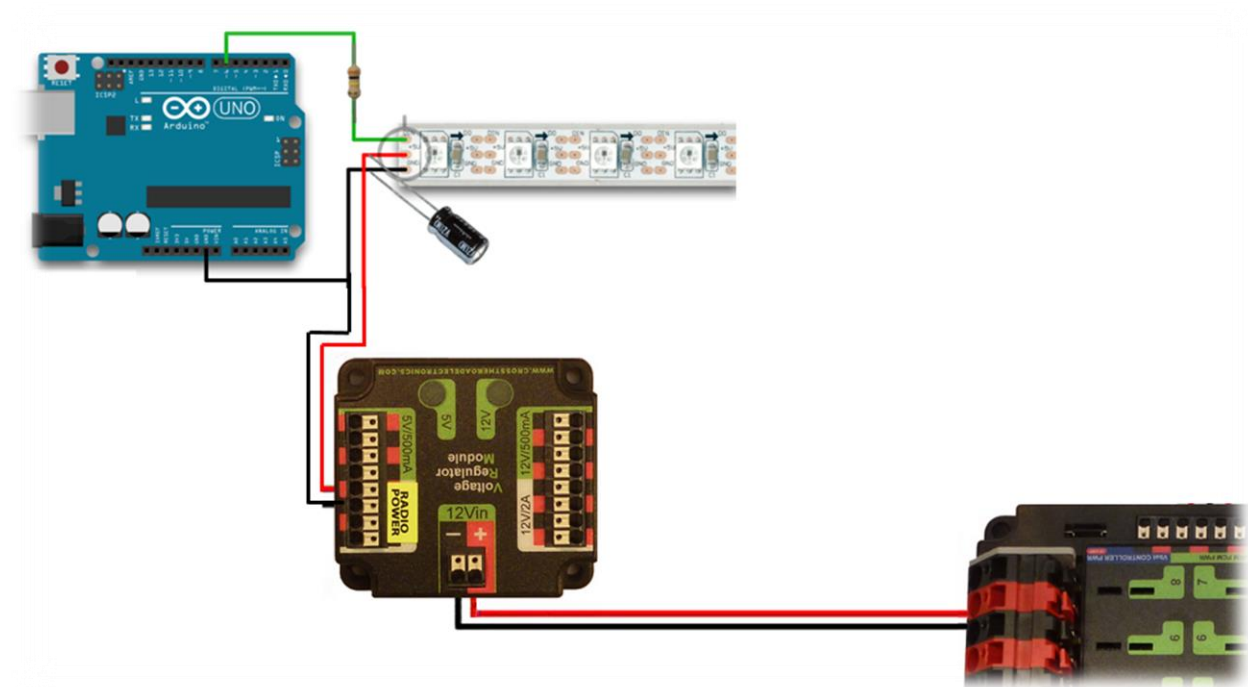
This guide implements the I2C protocol to accomplish these goals. At this point, I would review Sparkfun’s [I2C introductory guide](#). The I2C protocol is more flexible than, say, UART because it allows for a Master (e.g., RoboRIO, but can also be an Arduino) to interface with multiple slaves (e.g., many Arduinos, running different programs interfacing with a number of different things). The protocol also only requires two wires to implement!

Though the guide is for those interested in learning more about I2C and Arduino, it does assume a working knowledge of programming & electronics common on Robotics teams.

Wiring

Let’s start by understanding the physical connections required for the Master and Slave to communicate to one another.

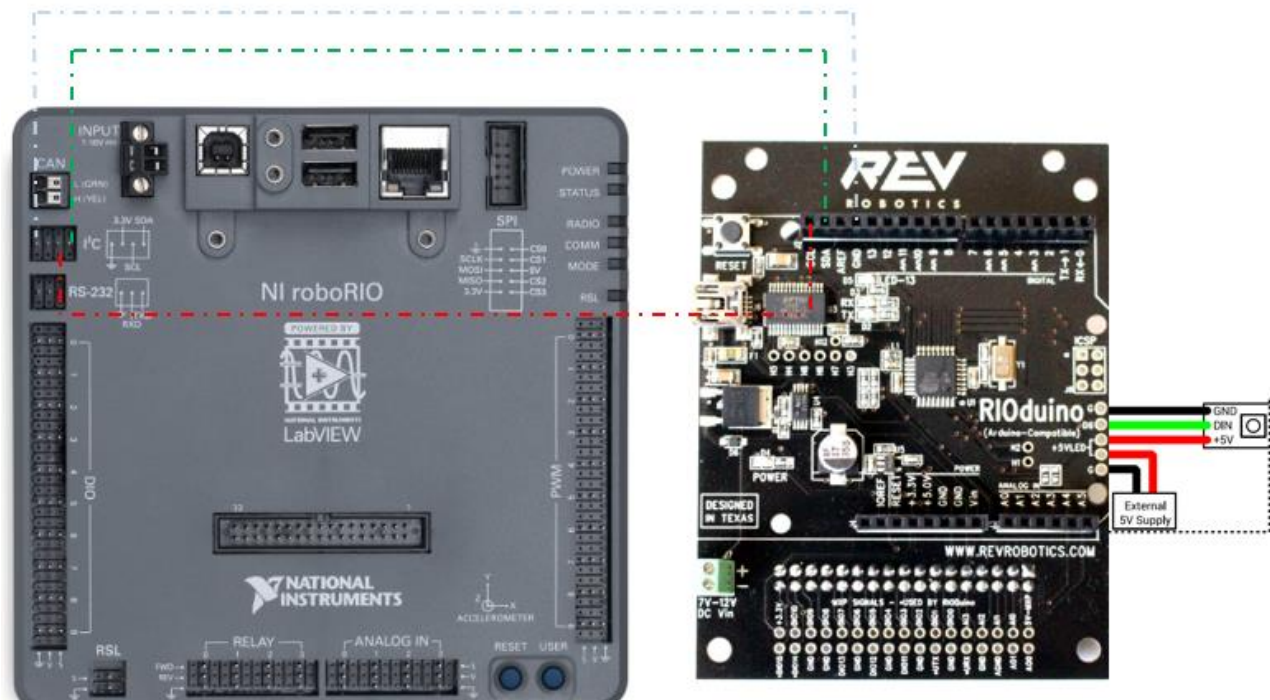
Here is the wiring diagram for power & control to the LED strip:



Don't forget the 1000uF Capacitor and 300-500 Ohm Resistor!

You'll need a Voltage Regulator Module ([am-2857](#)) dedicated to providing power to the LED strip, as the one used to power the access point shouldn't be shared. In all examples, the LED strip receives signal from PIN 6 on the microcontroller. In some versions of the microcontroller, such as with the RIOduino, you'll find through-holes you can solder connections to. This is what I did.

Here is the wiring diagram to connect the RoboRIO I2C port to the RIOduino I2C port:



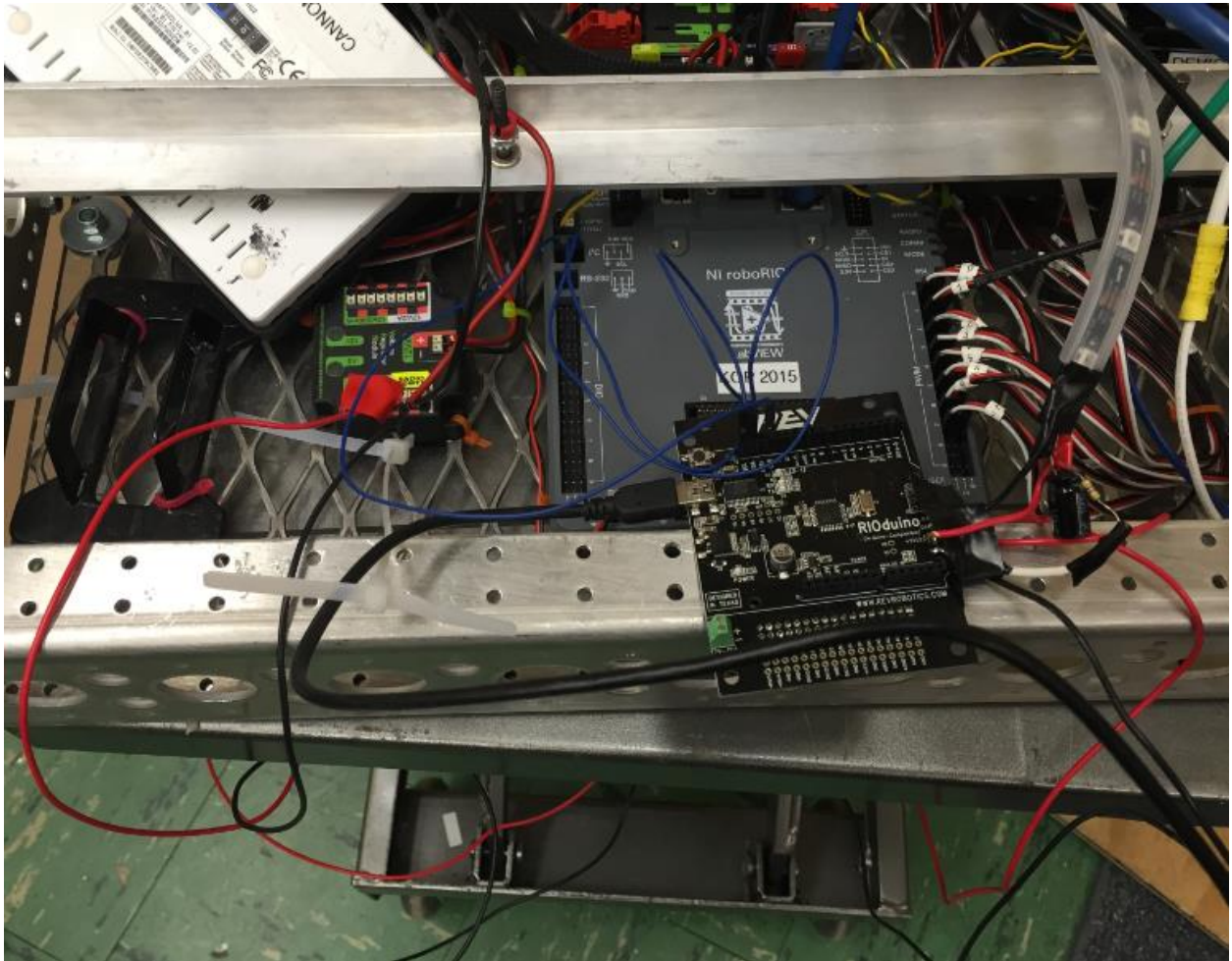
Again, Don't forget the 1000uF Capacitor and 300-500 Ohm Resistor (Not Shown Above)!

The external 5V supply is provided by the Voltage Regulator Module, and the Ground, Signal, and 5V lines (Black, Green, and Red respectively in the above diagram) connect to the corresponding pads on the LED strip. There might be small arrows on the strip indicating which side to attach the microcontroller to (it matters).

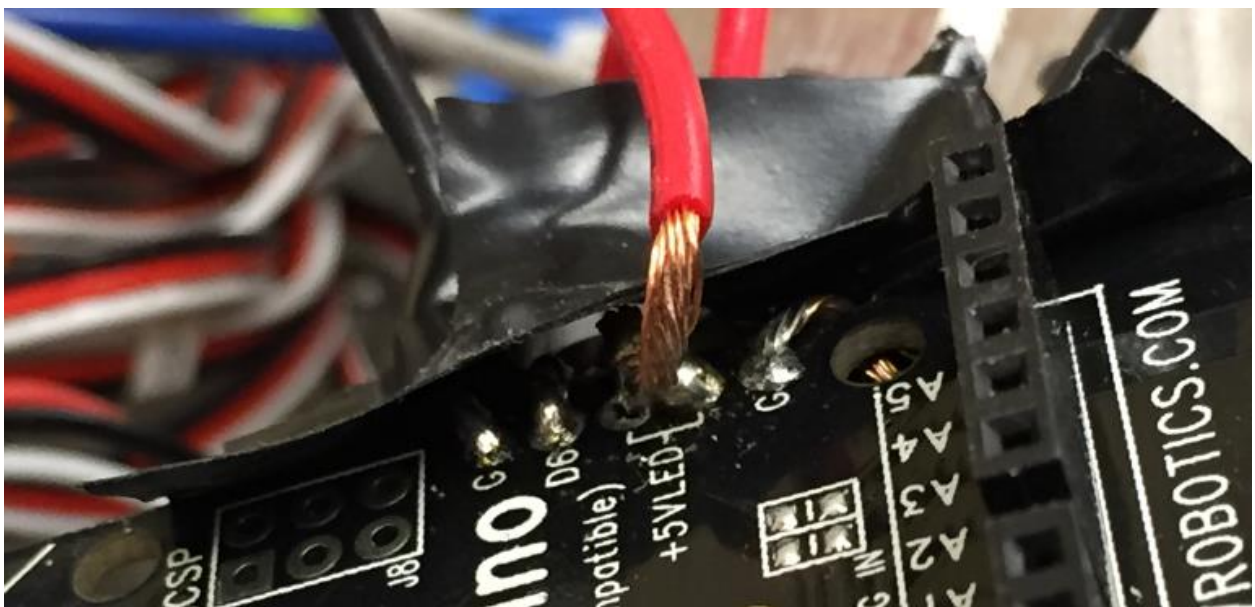
The roboRIO and RIOduino power connections are not shown here.

To wire the I2C ports of each device, you must connect SCL to SCL, SDA to SDA, and Ground to Ground. Once the wiring is down, you are ready to begin development.

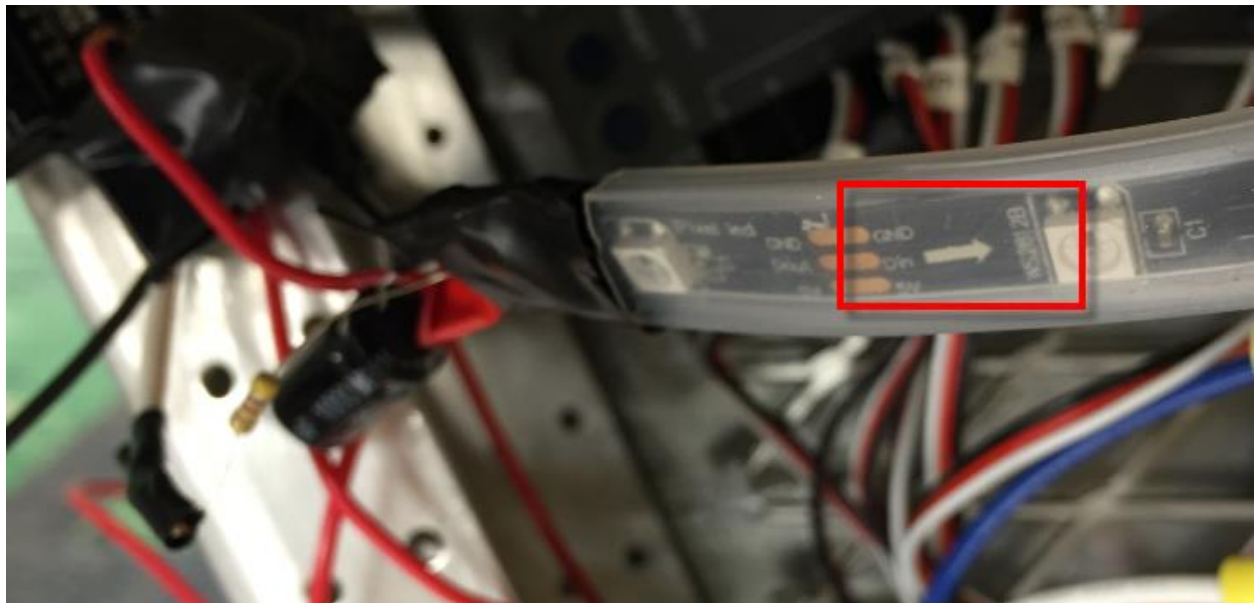
This is what reality looks like:



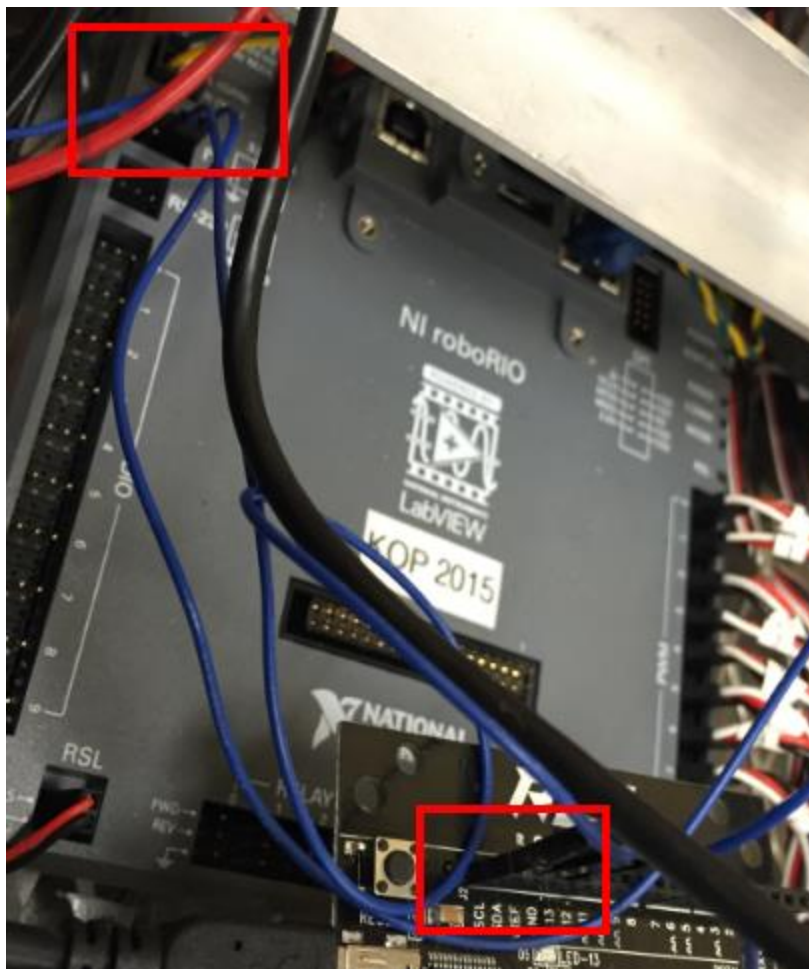
Closeup: Soldering wires to RIOduino



Closeup: Strip “Directionality”



Closeup: I2C Connections



Programming

Step 1: Verify you can run the example “strandtest” on the Arduino.

You’ll want to follow the steps outlined in Adafruit’s Website to install the Arduino IDE and NeoPixel libraries: [Learn>>Arduino Library](#). I encourage you to take a look at some of the other helpful tutorials on the website. Follow through the steps until you are able to deploy the example **strandtest** and see the LEDs working as expected. This should confirm the electronic wiring between the Arduino and strip LEDs has been done properly.

Not lighting up? Errors here can be due to:

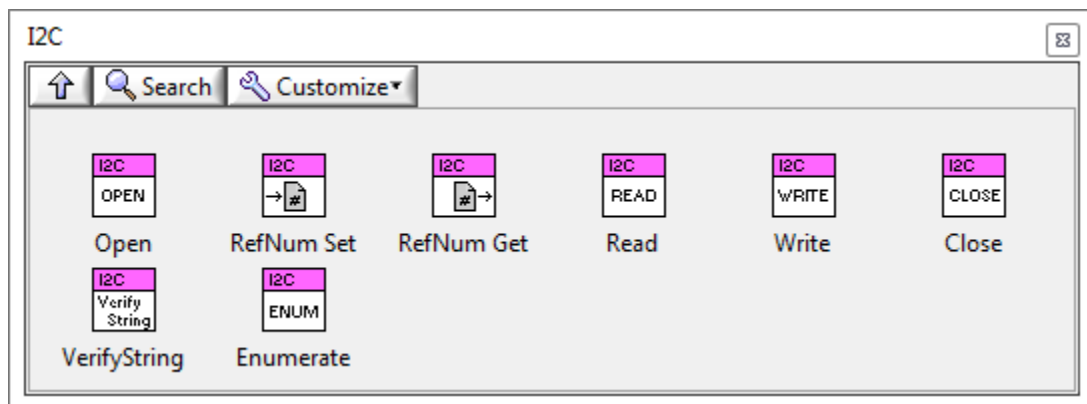
1. Wiring to the wrong side of the LED strip
2. Not wiring the proper pads (e.g., connecting the “5V” wire to the “DIN” pad)
3. Poor soldering (accidentally shorting connections or connections breaking off)
4. Damaged strip (possible if not protected with resistor & capacitor)

Where possible, secure your electrical job with heat shrink or electrical tape.

[How-to: Heat Shrink](#)

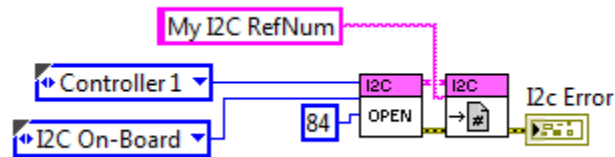
Step 2: Deploy a simple block of I2C Master Code to the RoboRIO

The palette we’ll be using to read and write I2C commands resides in **Functions >> WPI Robotics Library >> Communications >> I2C**



For help installing the LabVIEW FRC Update, consult the [FRC software setup guide](#), also linked at the beginning of the guide. I’m going to walk you through the components of a simple project that demonstrates sending a data from the RoboRIO to the Arduino.

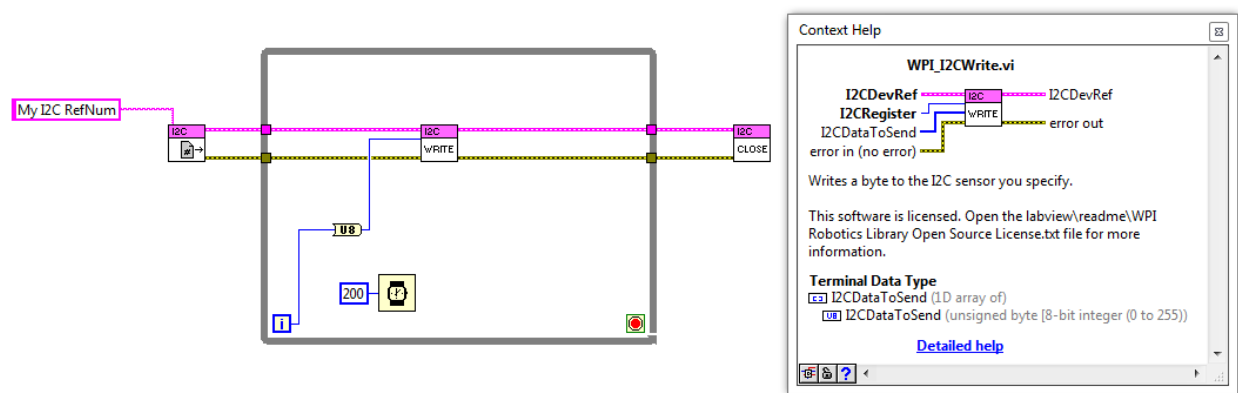
Opening an I2C connection to a Slave Address in Begin.VI



In this step, we open and save a reference to an I2C slave at address **84**. The connection is established via the on-board I2C port, though should also be possible via the MXP expansion port (I have not had success with this, you would select I2C MXP as the device bus). The address 84 is arbitrarily chosen, though it is important to set up the Arduino to join as a slave on address 84 as well. This will be explained later.

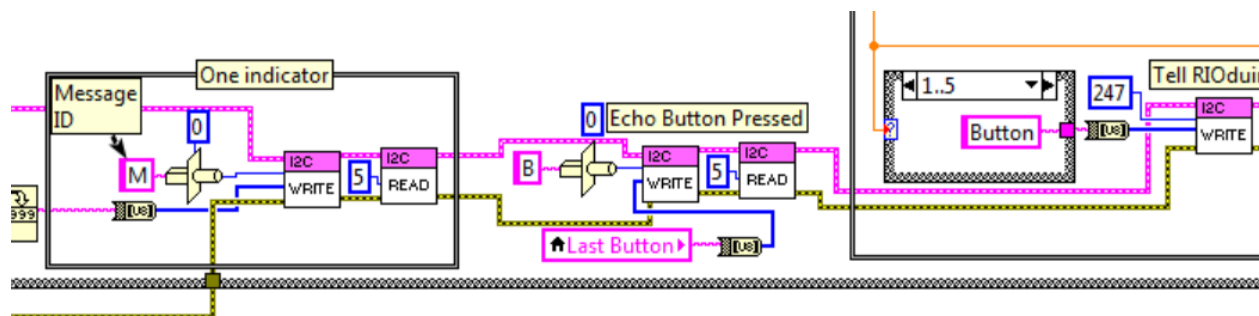
Send Data to I2C Slave in Periodic Tasks.VI

Open *I2C Hello World – LabVIEW.lvproj* and navigate to the block diagram of *Periodic Tasks.VI*. Deploy the code via *Robot Main.VI* to initiate data transmission over the I2C bus to Address 84 (Arduino).



In this block of code, we open a reference to the I2C channel created in `Begin.VI`, and continuously write an unsigned 8-bit integer value to the input, **I2CRegister**. You may wonder what the difference between **I2CRegister** and `I2CDataToSend`. Some I2C based sensors require the master to specify a register to send data to.

[This block of code](#) linked by the team that developed an I2C touch screen depicts this to some extent:



Each “write” block bundles a message ID with the message itself. In the case of the Arduino slave, we define how we want to send and parse data. In the hello world example above, we’re sending one byte of data (the loop iteration value) every 200 ms. Our goal is to read and display this information with the Arduino to the Serial Port Monitor.

Receive Data from I2C Master

Open *I2Cslavereceiver* in the Arduino IDE. This program is copied from Arduino’s website: [Master Write/Slave Receiver](#). It would be a good time to review this resource. The important step is to ensure the Arduino joins the I2C bus as a slave with address 84 (or however you’ve configured your I2C master in Begin.VI).

Take look at the setup (initialization) block of code:

```
void setup()
{
  Wire.begin(84);           // join i2c bus with address #4
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600);       // start serial for output
}
```

The setup phase above only occurs once and for the purpose of initializing Serial & I2C communication. The Serial communication is important because we’ll use the serial port to display the data received from the RoboRIO.

Let’s look at the remainder of the code:

```
void loop()
{
  delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany)
{
  while(1 < Wire.available()) // loop through all but the last
  {
    char c = Wire.read(); // receive byte as a character
    Serial.print(c);       // print the character
  }
  int x = Wire.read();      // receive byte as an integer
  Serial.println(x);        // print the integer
}
```

The looping is done in **Block 1**. Every 100 milliseconds, the loop executes and does essentially nothing. If the Arduino were to receive no messages from the RoboRIO, then the Arduino would serve no functional purpose. However, in **Block 2** we’ve defined a function `receiveEvent`. It *only* executes when a message is detected on the I2C bus from the I2C Master.

When `receiveEvent` executes, a loop begins that reads a byte of data from the bus, stores, and prints the data to the Serial Port Monitor. Every time the loop iterates, the program reads and displays a byte of data. To view the Serial Monitor, hold **Ctrl + Shift + M**, or **Tools >> Serial Monitor**.



You should begin to see an output:

1
2
3
4
...

This represents the bytes of data being generated by the loop in LabVIEW from the iteration terminal, sent across the I2C bus to a slave of address 84. The Arduino, with its portal initialized to address 84, recognizes the messages are meant for its consumption and receives & prints the data. Success here indicates we can move on and begin to control the LED output.

Step 2: Change individual LEDs to a color of choice


Deploy ReducedI2C to the Arduino

This piece of code is slightly more complicated as uses global variables and a few Adafruit library-specific functions such as `strip.setPixelColor` and `strip.setBrightness`. It would be a good time to review the [Arduino Library](#) page of the Adafruit Uberguide. The general operation of this code is:


Block 1: Initialize global variables to specify a) which LED you want to control, b) LED strip brightness, c) The R, G, and B values that correspond to the color of interest.

Block 2: Initialize I2C Communication, Serial Communication, and the LED strip

```
// IMPORTANT: To reduce NeoPixel burnout risk, add 1000 uF capacitor across  
// pixel power leads, add 300 - 500 Ohm resistor on first pixel's data input  
// and minimize distance between Arduino and first pixel. Avoid connecting  
// on a live circuit...if you must, connect GND first.
```



```
int ledindex = 0;  
int brightness = 0;  
int R = 0;  
int G = 0;  
int B = 0;
```



```
void setup() {  
  strip.begin();  
  strip.show(); // Initialize all pixels to 'off'  
  Wire.begin(84); // join i2c bus with address #84  
  Wire.onReceive(receiveEvent); // register event  
  Serial.begin(9600); // start serial for output  
  Serial.print("init"); //initialize complete  
  Wire.onRequest(requestEvent);  
}
```

Complete program continue on next page...

Block 3: Start the looping phase. During every loop iteration, use `strip.setPixelColor` to adjust an LED at position `ledindex` to the color corresponding to the R, G, and B combination. Use `strip.setBrightness` to adjust brightness and `strip.show` to update the strip by deploying the settings.

Block 4: When the RoboRIO sends a packet of data, `receiveEvent` fires and stores that data into the global variables defined. The first byte is stored in `ledindex`, the second to R, the third to G, the fourth to B, the fifth to `brightness`. It is up to us to ensure the RoboRIO is sending **5 bytes** of data to populate each of the global variables.

```
void loop()
{

strip.setPixelColor(ledindex, R, G, B);
strip.setBrightness(brightness);
strip.show();

}
```

3

```
void receiveEvent(int howMany)
{

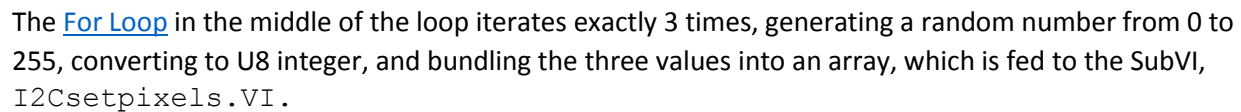
    //char c = Wire.read(); // receive byte as a character
    //Serial.print(c);      // print the character

    ledindex = Wire.read(); // receive byte as an integer
    R = Wire.read();
    G = Wire.read();
    B = Wire.read(); //Serial.println(x);          // print the integer
    brightness = Wire.read();
    Serial.println("Received");

}
```

4

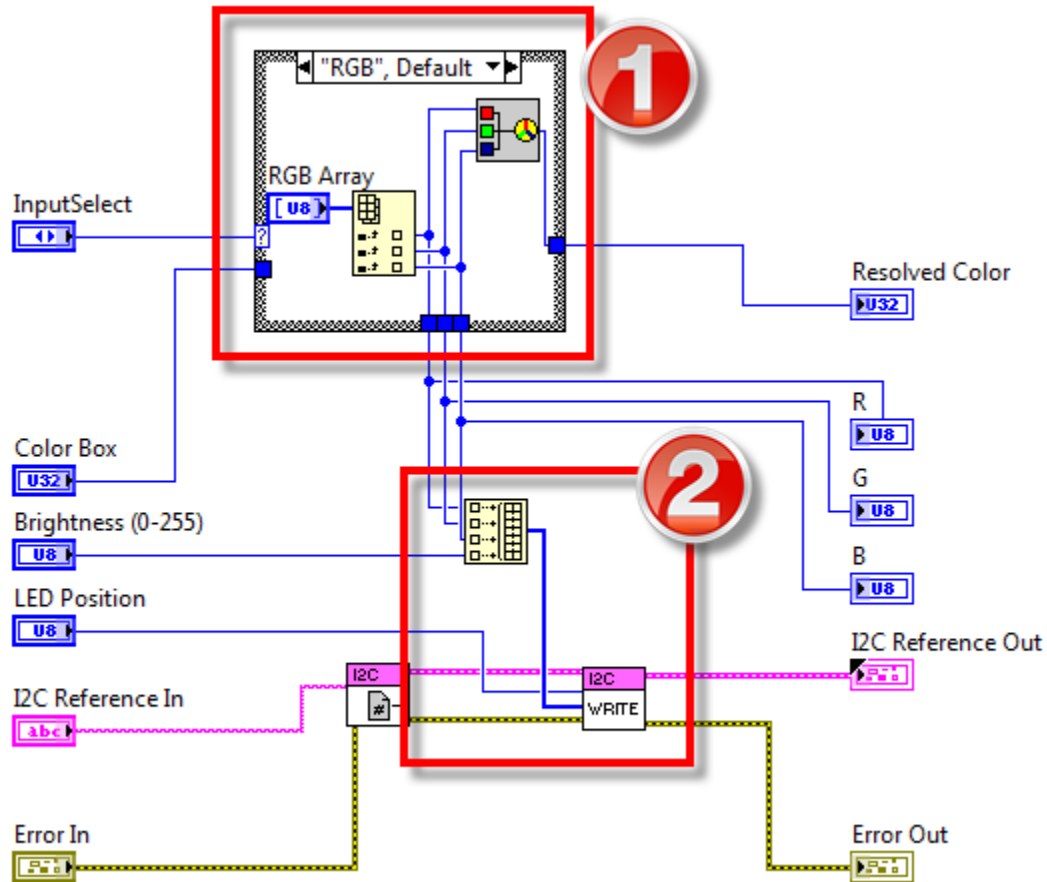
Navigate to **Periodic Tasks.VI** and let's study the upgrades to the code. This is a program that cycles through each of the 30 LEDs and uses a random number generator to set the color of each LED to a random color.



Byte 1: Loop Iteration (cycles from 1 to 30, as defined by the **N** terminal of the For Loop). For larger LED strips, the input to the terminal would need to be increased.

Byte 5: LED brightness, taken from a control on the Front Panel, will be fed to the global variable brightness on the Arduino.

Let's see what's happening inside of the SubVI:

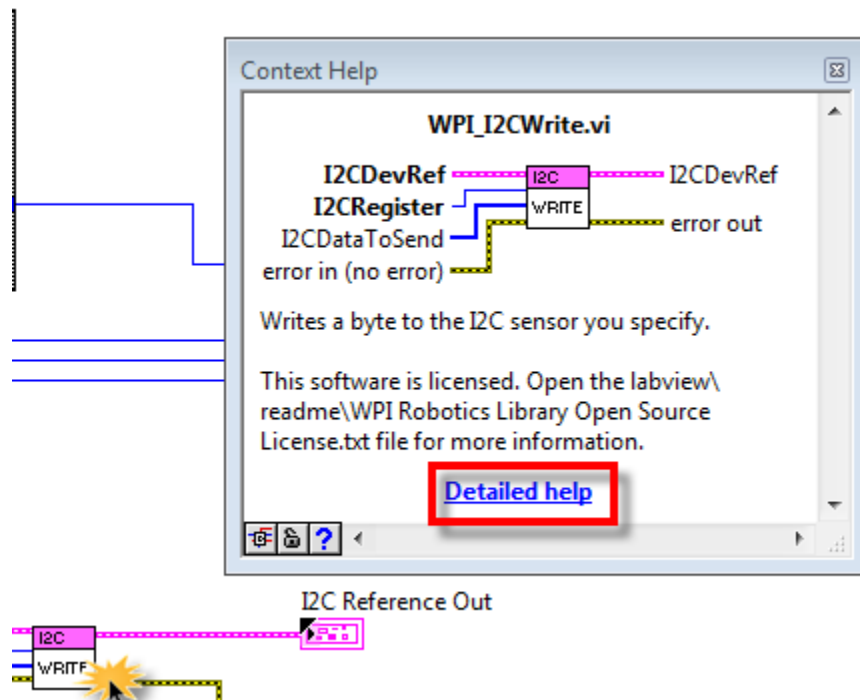


I realize reflecting on this code that there are some optimizations to be made. For instance, it doesn't make much sense to unbundle the RGB Array in **Block 1**, only to bundle it back together in **Block 2** with the brightness control. We could have just as easily taken care of the bundling outside of the SubVI. The main purpose of the SubVI is to clean up the block diagram and handle the case structure in **Block 1**. You can choose to set LED color with either a Color Box or explicit R, G, and B values.

Take a moment to notice how data is being sent to I2C Write; order matters!

I suggest learning more about the I2C Palette using the Detailed Help:

To do this press **Ctrl + H** (Context Help) and move your cursor over the I2C Write function, select **Detailed Help**



Scroll down and read up on the **I2CRegister** and **I2CDataToSend** input terminals:

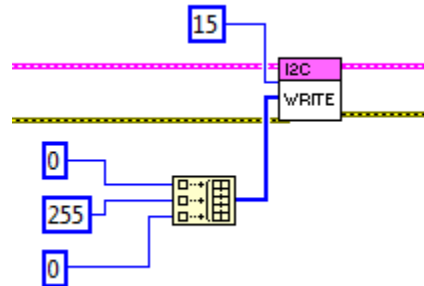
[U8] **I2CRegister** specifies the register of the I²C sensor to which you want to write.

[U8] **I2CDataToSend** specifies the array of bytes you want to write to the I²C sensor.

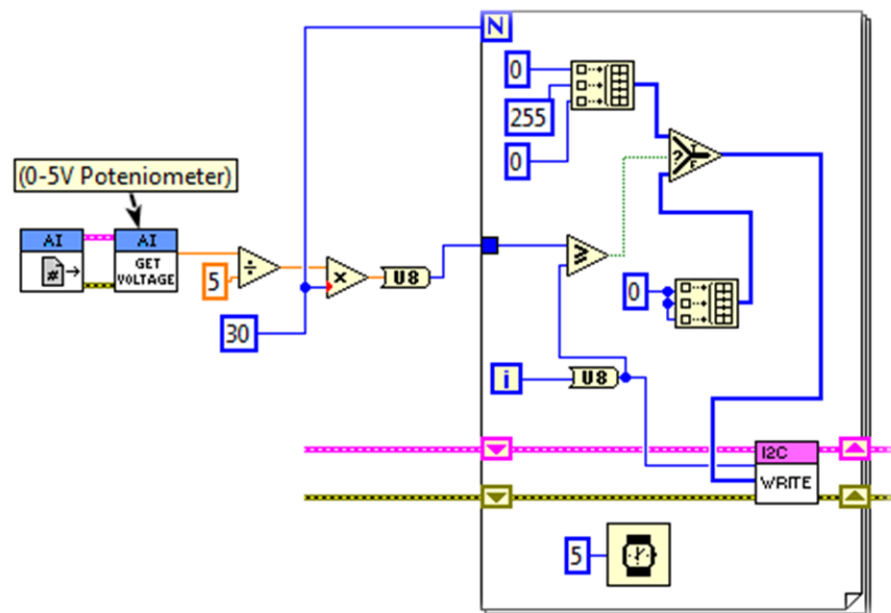
You can send more or less bytes to the Arduino by adjusting the array in **I2CDataToSend**. Just make sure you are handling the bytes properly on the other end. For your learning, you'll want to do some testing to understand what happens when you send more bytes than you handle, or try to handle more bytes than you are sending. (Hint: to send more bytes, [increase the dimension](#) of the array)

At this point, you can begin to get creative by generating custom lighting patterns in LabVIEW. Assuming the same code on the Arduino, here are a few simple example scenarios in LabVIEW:

Turn LED at Index 15 Green



Turn LEDs on one by one as potentiometer value increases from 0 to 5 volts.



In this program (running in a larger loop), the output voltage of the 5V potentiometer is scaled to reflect an integer value between 0 and 30 (the number of LEDs in this test strip). In each iteration of the For Loop, this scaled value is compared with the iteration number. If the scaled value is *larger* than the iteration number, then the upper array is sent (corresponding to Green). Otherwise, the lower array is sent (corresponding to Off, or 0 0 0).

Thus, while the program is running, a potentiometer output of 2.5V (or half turn), should cause the first half the LED strip to turn Green. As the potentiometer continues turning from 2.5V to 5V, additional LEDs should turn green, until the potentiometer is fully turned and all the LEDs have illuminated.

Concluding Words

In this guide, we covered a range of topics, with lessons that can be extended to development platforms that best suit your project needs. It is my hope that the explanations were clear and helpful in advancing your understanding and appreciation of electronics & programming. I encountered many gaps in documentation on my learning journey and did my best to consolidate here. Please feel free to submit feedback to prifken@bchigh.edu where additional clarification is required or you see an error.

Good luck!