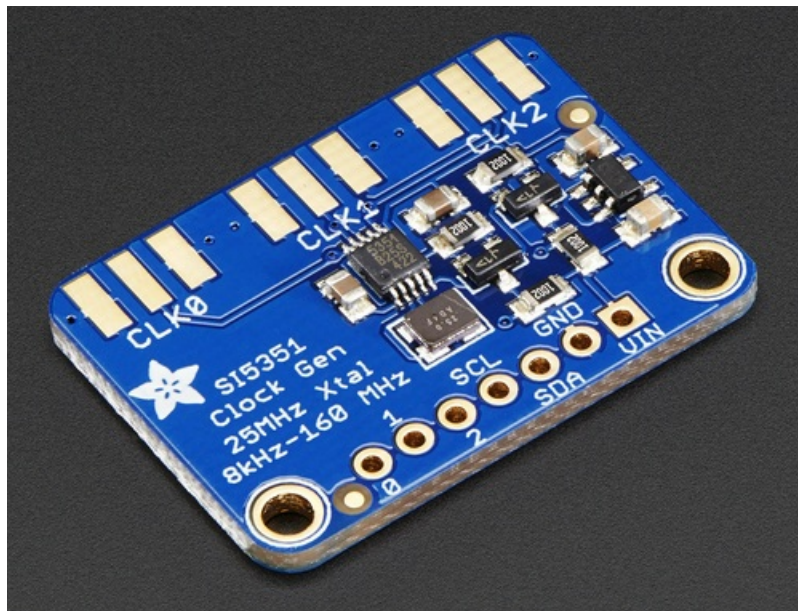




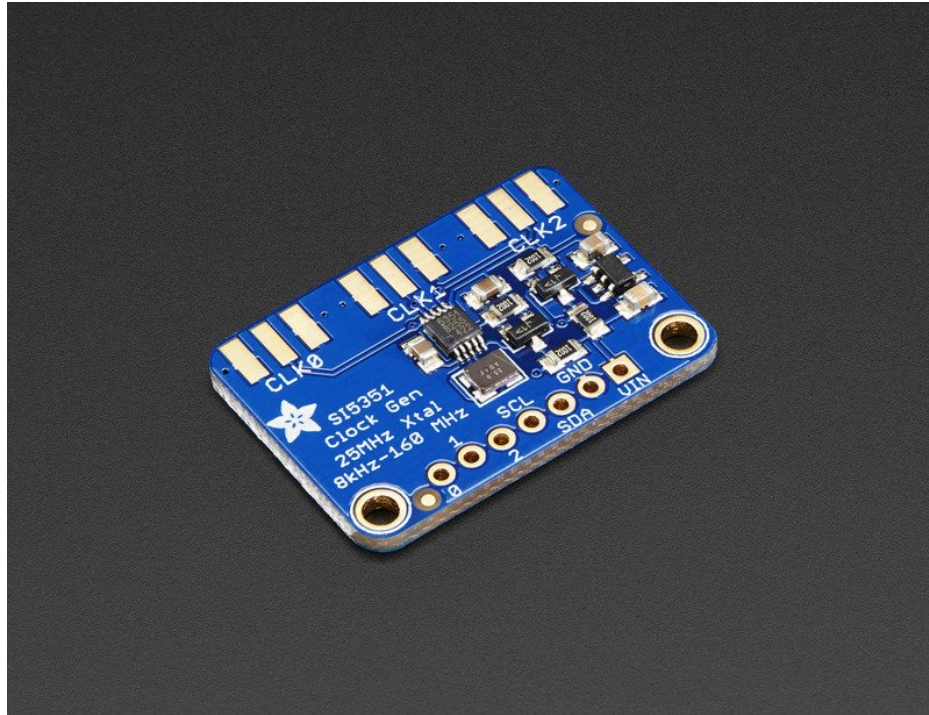
## Adafruit Si5351 Clock Generator Breakout

Created by lady ada

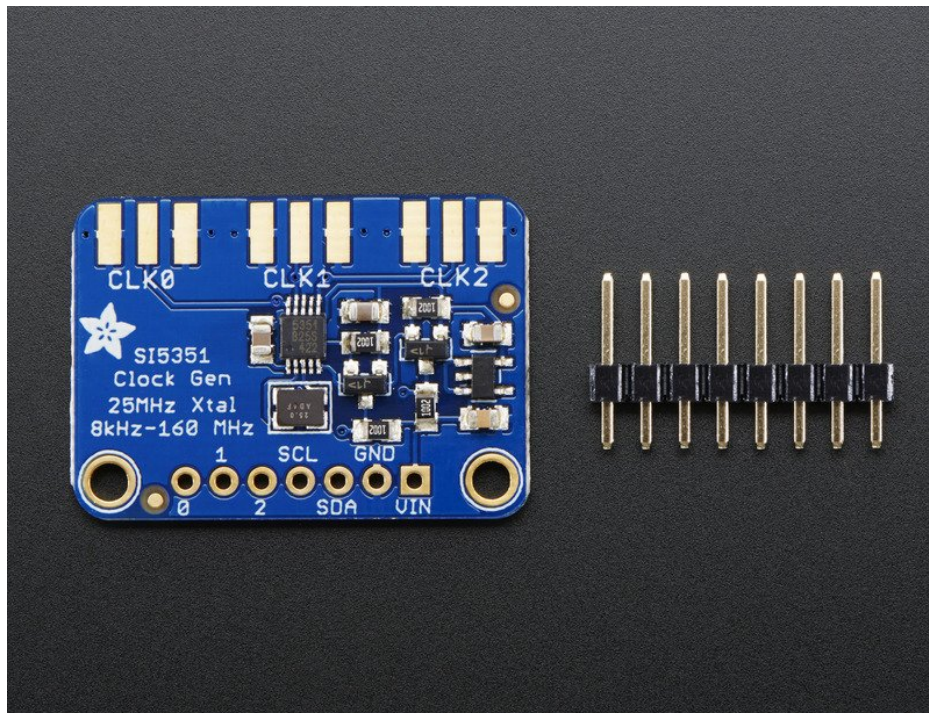


Last updated on 2019-11-18 03:05:03 AM UTC

## Overview

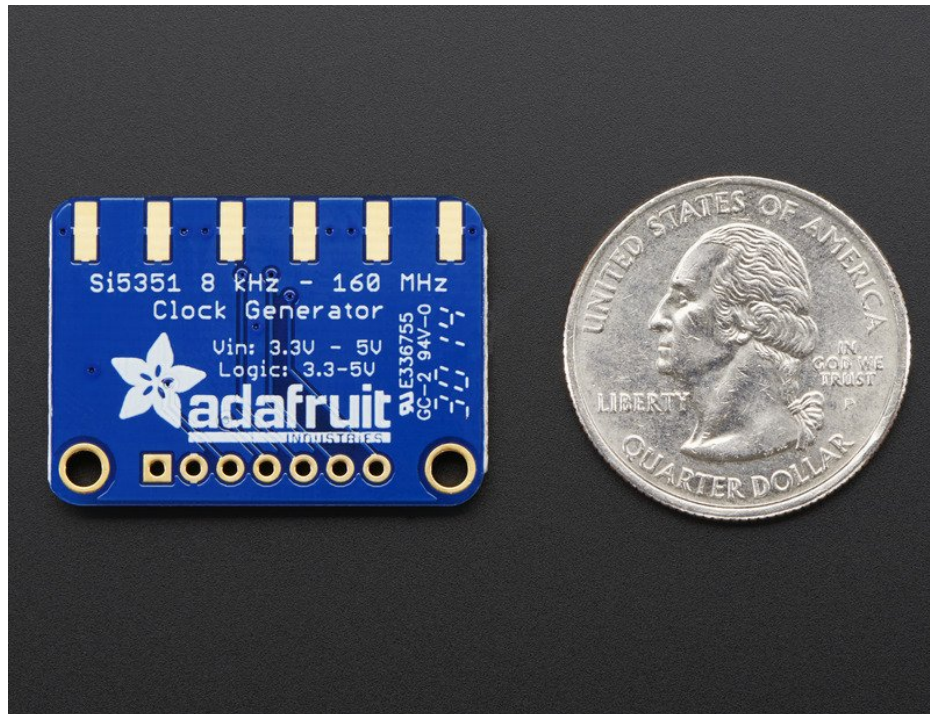


Never hunt around for another crystal again, with the Si5351 clock generator breakout from Adafruit! This chip has a precision 25MHz crystal reference and internal PLL and dividers so it can generate just about any frequency, from <8KHz up to 150+ MHz.



The Si5351 clock generator is an I2C controller clock generator. It uses the onboard precision clock to drive multiple PLL's and clock dividers using I2C instructions. By setting up the PLL and dividers you can create precise and arbitrary

frequencies. There are three independent outputs, and each one can have a different frequency. Outputs are 3Vpp, either through a breadboard-friendly header or, for RF work, an optional SMA connector.

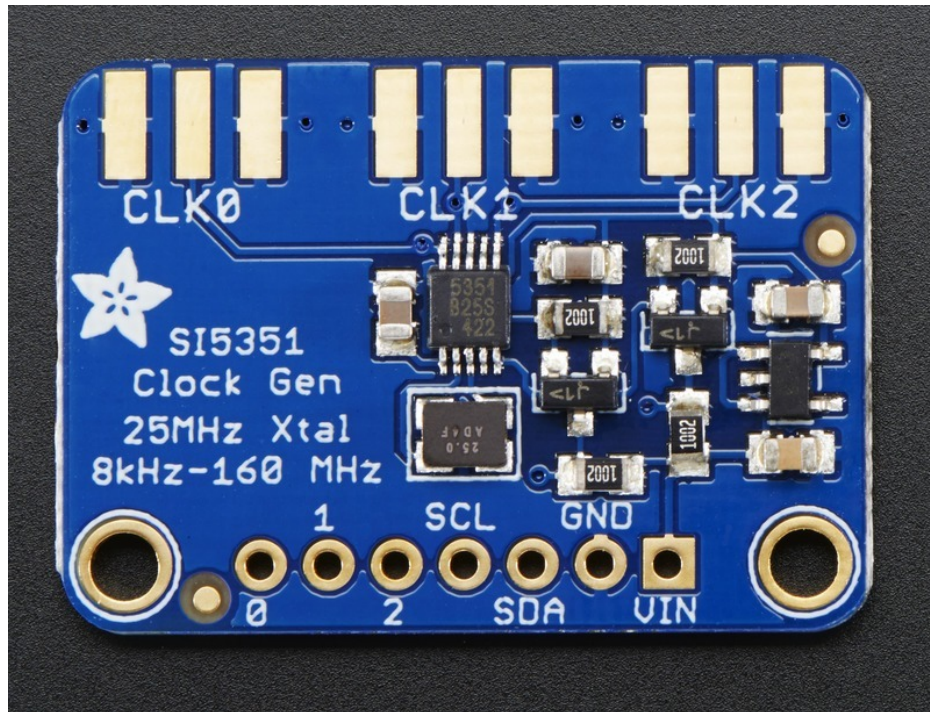


We put this handy little chip onto it's own breakout board PCB, with a 3.3V LDO regulator so it can be powered from 3-5VDC. We also put level shifting circuitry on the I2C lines so you can use this chip safely with 3V or 5V logic.

Best of all, we even have a great tutorial and library to get you started! Our code is designed for use with the Arduino microcontroller and IDE but is easily ported to your favorite platform with I2C support



## Pinouts



### Power Pins

The clock generator on the breakout requires 3V power. Since many customers have 5V microcontrollers like Arduino, we tossed a 3.3V regulator on the board. Its ultra-low dropout so you can power it from 3.3V-5V just fine.

- **Vin** - this is the power pin. Since the chip uses 3 VDC, we have included a voltage regulator on board that will take 3-5VDC and safely convert it down. To power the board, give it the same power as the logic level of your microcontroller - e.g. for a 5V micro like Arduino, use 5V
- **GND** - common ground for power and logic

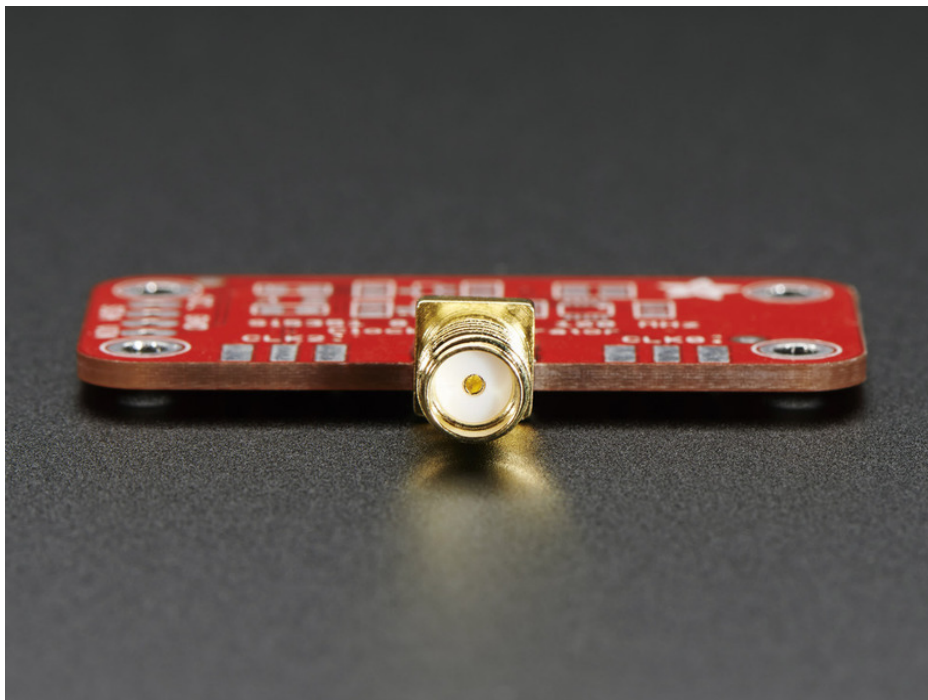
### I2C Pins

- **SCL** - I2C clock pin, connect to your microcontrollers I2C clock line. This pin is level shifted so you can use 3-5V logic, and there's a 10K pullup on this pin.
- **SDA** - I2C data pin, connect to your microcontrollers I2C data line. This pin is level shifted so you can use 3-5V logic, and there's a 10K pullup on this pin.

### Clock Out Pins

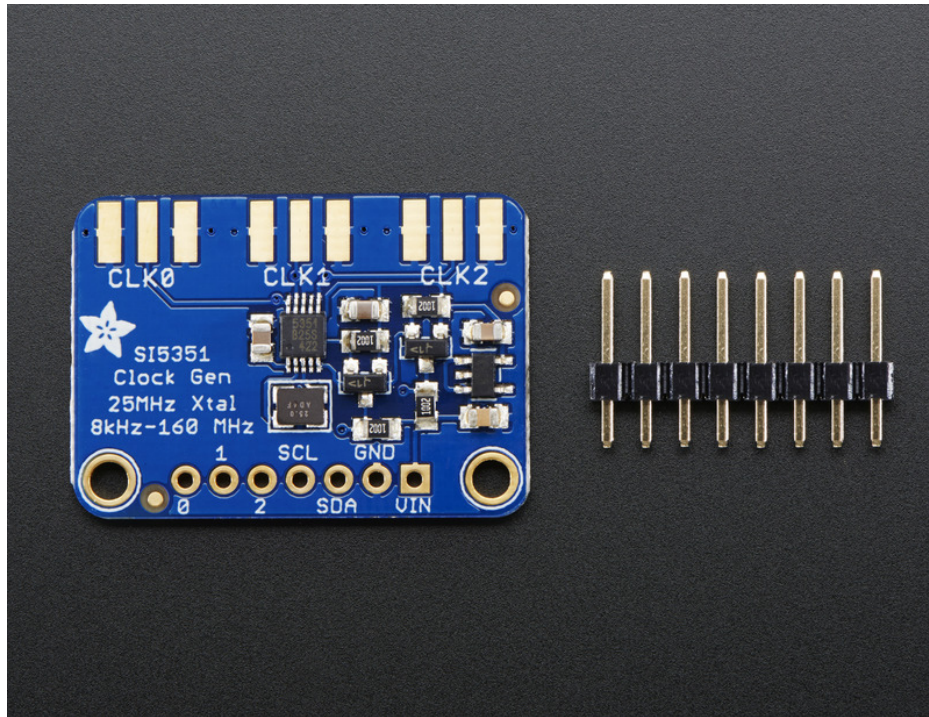
- **0, 1, and 2** - These are the 3 independent clock generated outputs. They are square waves, from 0-3V.

The clock out pins are also brought out to SMA edge-launch connectors on the other side of the PCB. [You can purchase and solder on some edge-launch SMA connectors \(https://adafru.it/dPr\)](https://adafru.it/dPr) if you want to pipe the signal into an RF cable.

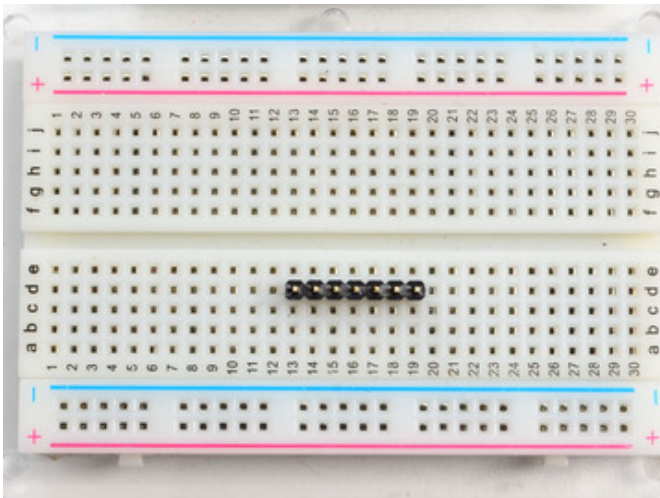




## Assembly

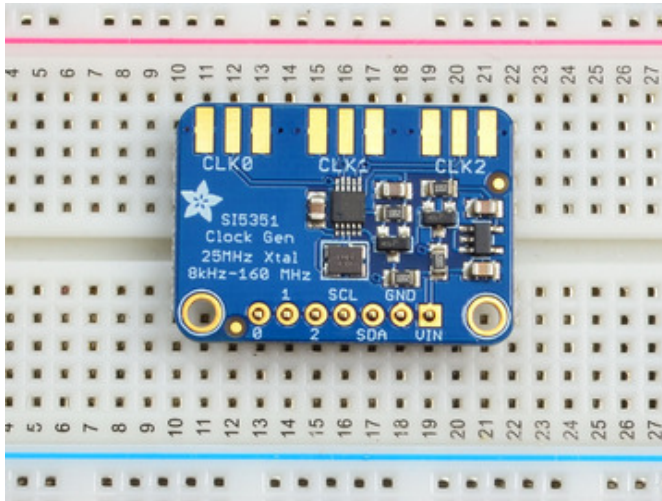


If you have the breadboard version of this sensor, you'll want to solder some header onto the sensor so it can be used in a breadboard.



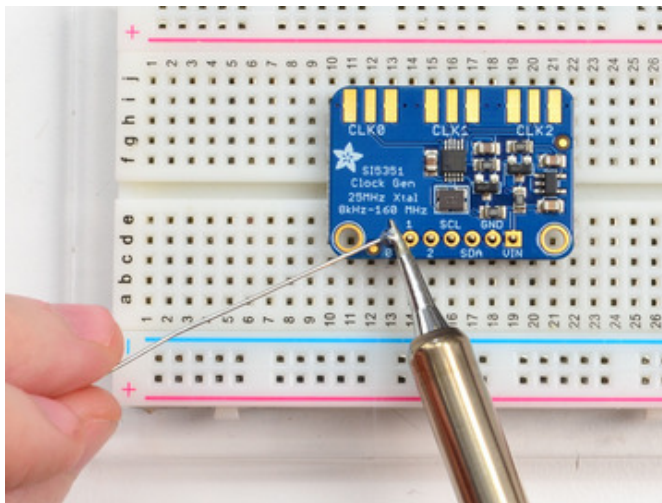
Prepare the header strip:

Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**



Add the breakout board:

Place the breakout board over the pins so that the short pins poke through the breakout pads

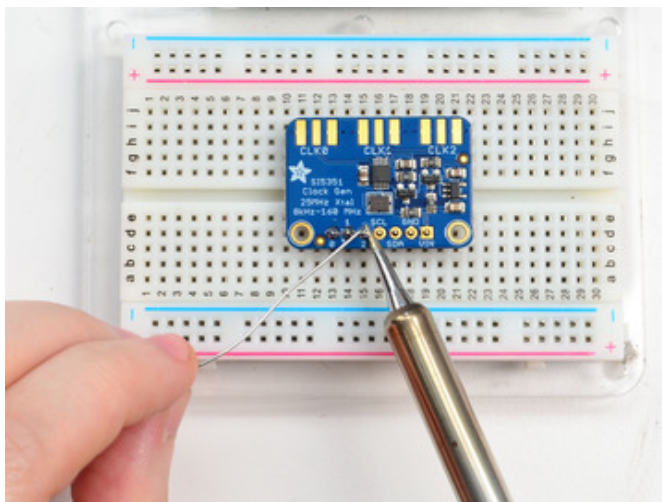


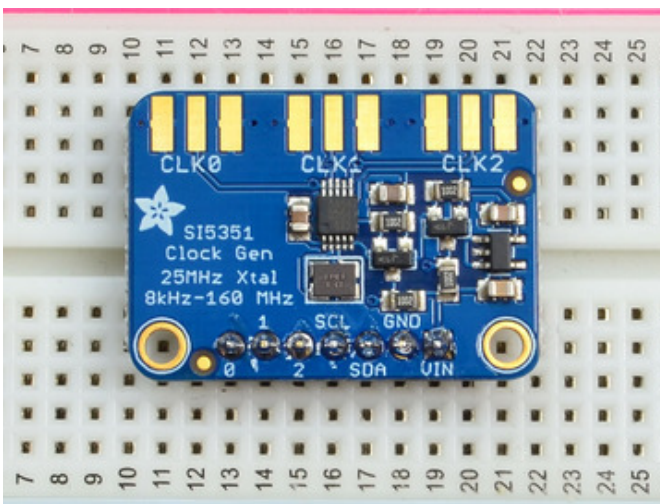
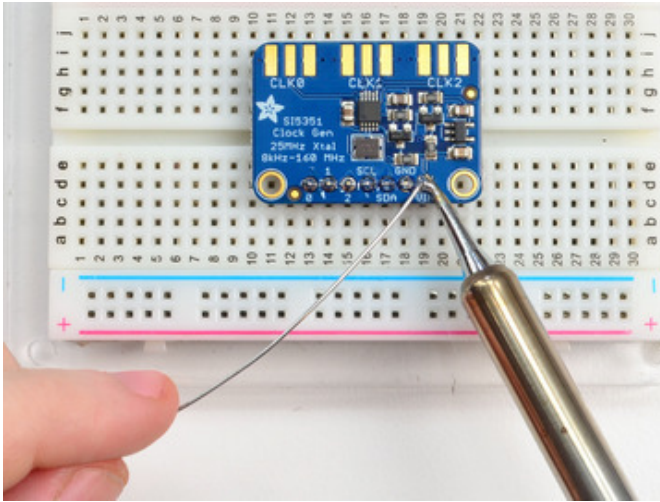
And Solder!

Be sure to solder all pins for reliable electrical contact.

Solder the longer power/data strip first

(For tips on soldering, be sure to check out our [Guide to Excellent Soldering](https://adafruit.it/aTk) (<https://adafruit.it/aTk>)).





You're done! Check your solder joints visually and continue onto the next steps



## Arduino Code

### Wiring for Arduino

You can easily wire this breakout to any microcontroller, we'll be using an Arduino. For another kind of microcontroller, just make sure it has I2C capability, then port the code - its pretty simple stuff!

(<https://adafru.it/dPs>)



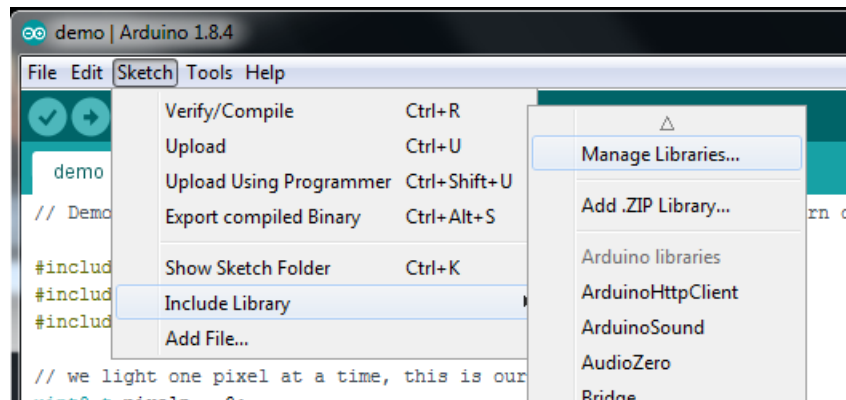
(<https://adafru.it/pBC>)

- Connect **Vin** to the power supply, 3-5V is fine. Use the same voltage that the microcontroller logic is based off of. For most Arduinos, that is 5V
- Connect **GND** to common power/data ground
- Connect the **SCL** pin to the I2C clock **SCL** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A5**, on a Mega it is also known as **digital 21** and on a Leonardo/Micro, **digital 3**
- Connect the **SDA** pin to the I2C data **SDA** pin on your Arduino. On an UNO & '328 based Arduino, this is also known as **A4**, on a Mega it is also known as **digital 20** and on a Leonardo/Micro, **digital 2**

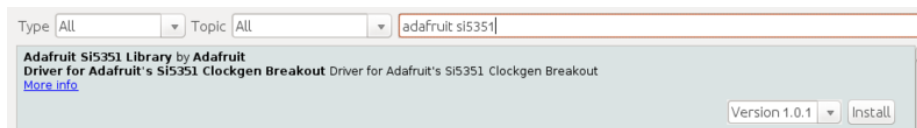
### Download Adafruit\_Si5351

To begin reading sensor data, you will need to download the **Adafruit Si5351** library from the Arduino library manager.

Open up the Arduino library manager:



Search for the **Adafruit Si5351** library and install it

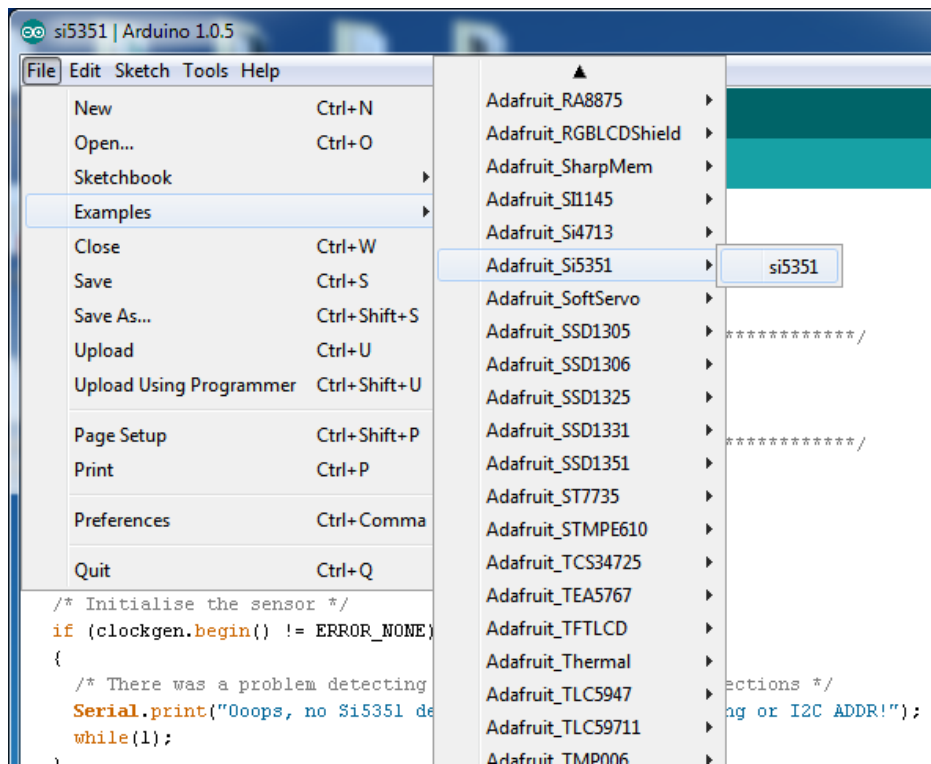


We also have a great tutorial on Arduino library installation at:

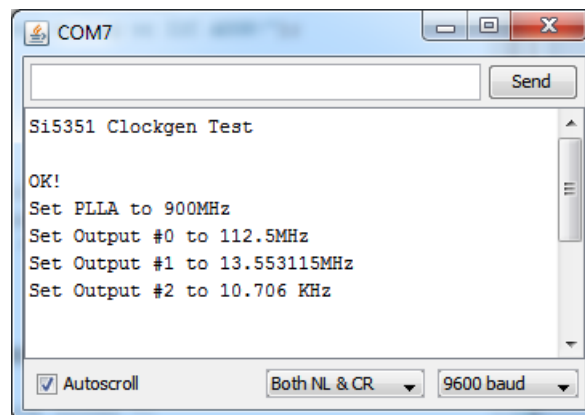
<http://learn.adafruit.com/adafruit-all-about-arduino-libraries-install-use> (<https://adafru.it/aYM>)

### Load Demo Sketch

Now you can open up **File->Examples->Adafruit\_Si5351->Si5351** and upload to your Arduino wired up to the sensor



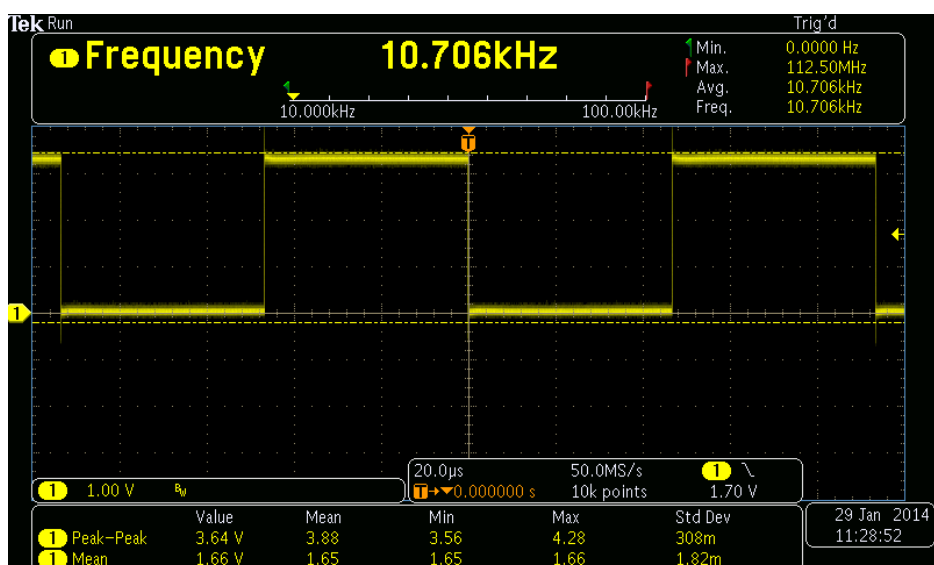
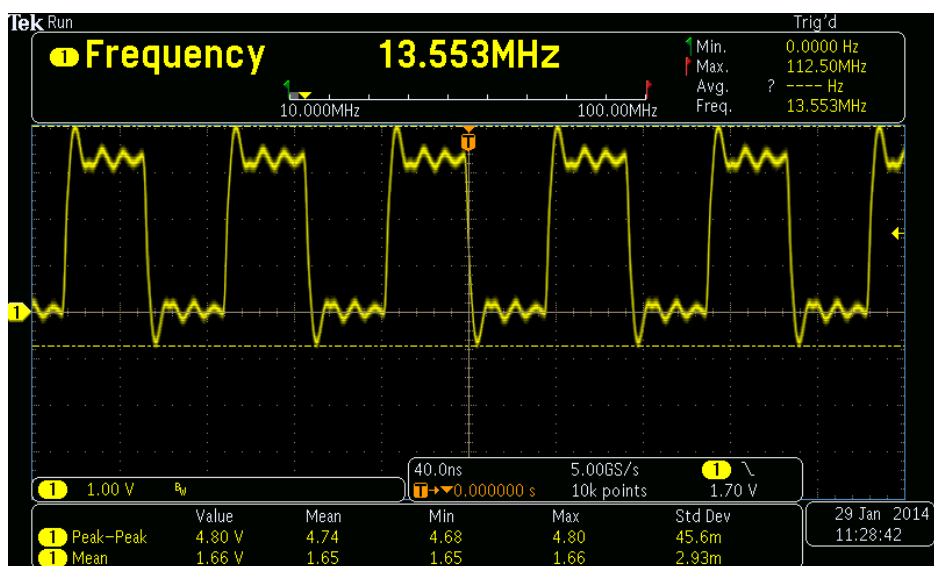
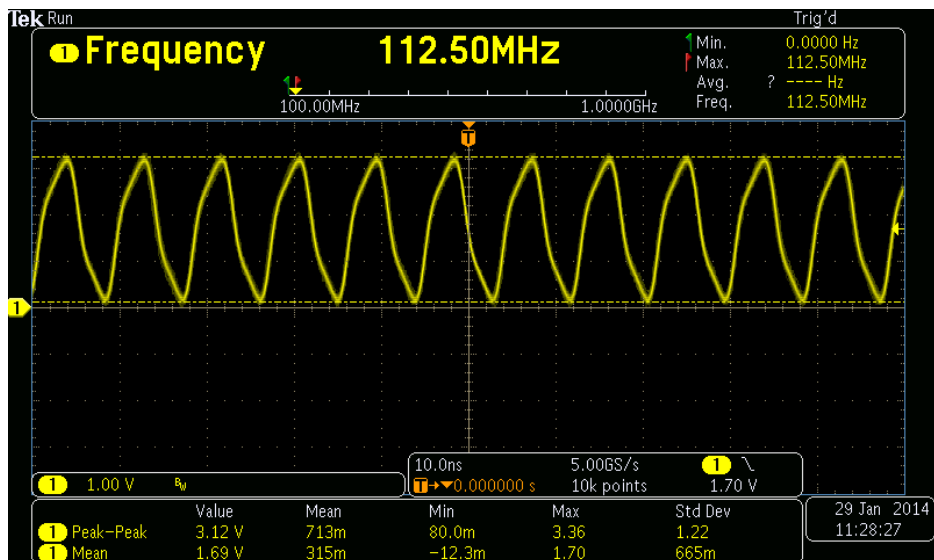
Then open up the Serial console at 9600 baud to check the output. You should see the following:



Now you can use your oscilloscope to probe the #0, #1 and #2 outputs



Depending on your oscilloscope make and model, it may not be possible for you to verify the 112.5MHz output frequency!



That's it! If you want to change the frequencies, adjust the example sketch and re-upload.



## Library Reference

---

The library we have is simple and easy to use

You can create the **Adafruit\_SI5351** object with:

```
Adafruit_SI5351 clockgen = Adafruit_SI5351();
```

I2C does not have pins, as they are fixed in hardware.

## Begin!

---

To initialize the chip, call **clockgen.begin()** which will check that it can be found. **begin()** returns true/false depending on these checks. We suggest you wrap **begin()** in a statement that will check if the chip was located:

```
if (clockgen.begin() != ERROR_NONE)
{
  /* There was a problem detecting the IC ... check your connections */
  Serial.print("Ooops, no Si5351 detected ... Check your wiring or I2C ADDR!");
  while(1);
}
```

## Set up the PLL

---

The chip uses two subsections to generate clock outputs. First it **multiplies** the 25MHz reference clock by some amount (setting up the PLL), then it **divides** that new clock by some other amount (setting up the clock divider)

By noodling with the multiplier and divider you can generate just about any clock frequency!

There are **two** PLL multipliers (A and B), so if you want to have three outputs, two outputs will have to share one PLL.

### Set up the PLL with 'integer mode'

The cleanest way to run the PLL is to do a straight up integer multiplication:

```
clockgen.setupPLLInt(SI5351_PLL_A or SI5351_PLL_B, m);
```

This sets **PLL\_A** or **PLL\_B** to be **25MHz \* m** and **m** (the integer multiplier) can range from **15** to **90**!

### Set up the PLL with 'fractional mode'

This mode allows a much more flexible PLL setting by using fractional multipliers for the PLL setup, however, the output may have a slight amount of jitter so if possible, try to use integer mode!

```
clockgen.setupPLLInt(SI5351_PLL_A or SI5351_PLL_B, m, n, d);
```

This sets **PLL\_A** or **PLL\_B** to be **25MHz \* (m + n/d)**

- **m** (the integer multiplier) can range from **15** to **90**
- **n** (the numerator) can range from **0** to **1,048,575**
- **d** (the denominator) can range from **1** to **1,048,575**

## Set up the clock divider

Once you have the PLLs set up, you can now divide that high frequency down to get the number you want for the output

Each output has its own divider. You can use the cleaner Integer-only divider:

```
clockgen.setupMultisynthInt(output, SI5351_PLL_x, SI5351_MULTISYNTH_DIV_x);
```

- For the **output** use 0, 1 or 2
- For the PLL input, use either **SI5351\_PLL\_A** or **SI5351\_PLL\_B**
- For the divider, you can divide by **SI5351\_MULTISYNTH\_DIV\_4**, **SI5351\_MULTISYNTH\_DIV\_6**, or **SI5351\_MULTISYNTH\_DIV\_8**

Again, integer output will give you the cleanest clock. If you need more flexibility, use the fractional generator/divider:

```
clockgen.setupMultisynth(output, SI5351_PLL_x, div, n, d);
```

- For the **output** use 0, 1 or 2
- For the PLL input, use either **SI5351\_PLL\_A** or **SI5351\_PLL\_B**
- The final frequency is equal to the PLL / (div + n/d)
- **div** can range from **4** to **900**
- **n** can range from **0** to **1,048,575**
- **d** can range from **1** to **1,048,575**

## Additional R Divider

If you need to divide even more, to get to the < 100 KHz frequencies, there's an additional R divider, that divides the output once more by a fixed number:

```
clockgen.setupRdiv(output, SI5351_R_DIV_x);
```

**output** is the clock output #

The R divider can be any of the following:

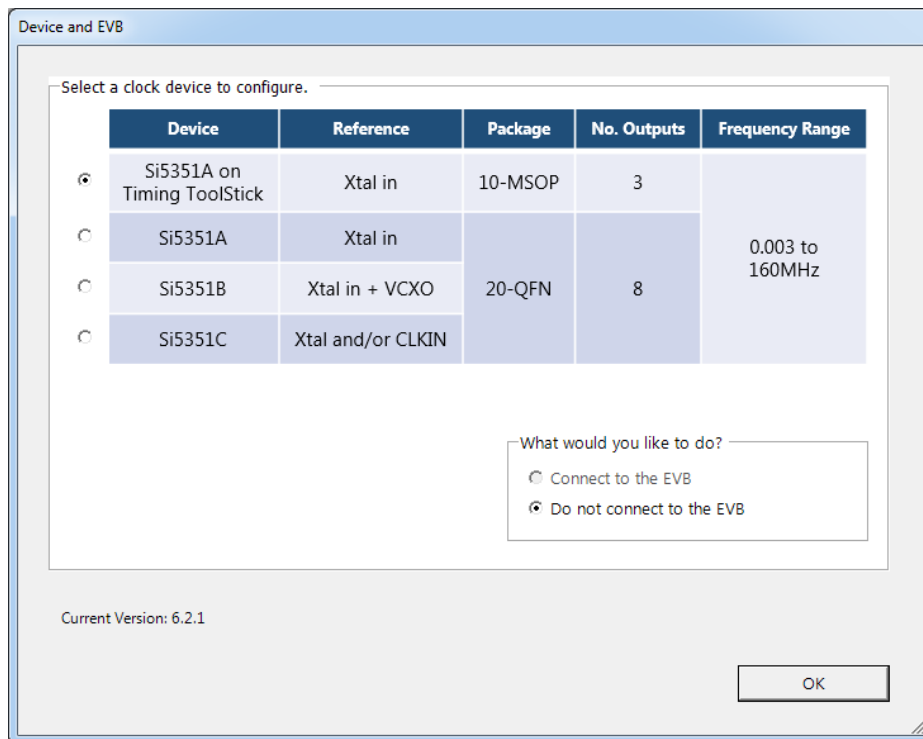
- SI5351\_R\_DIV\_1
- SI5351\_R\_DIV\_2
- SI5351\_R\_DIV\_4
- SI5351\_R\_DIV\_8
- SI5351\_R\_DIV\_16
- SI5351\_R\_DIV\_32
- SI5351\_R\_DIV\_64
- SI5351\_R\_DIV\_128

## Software

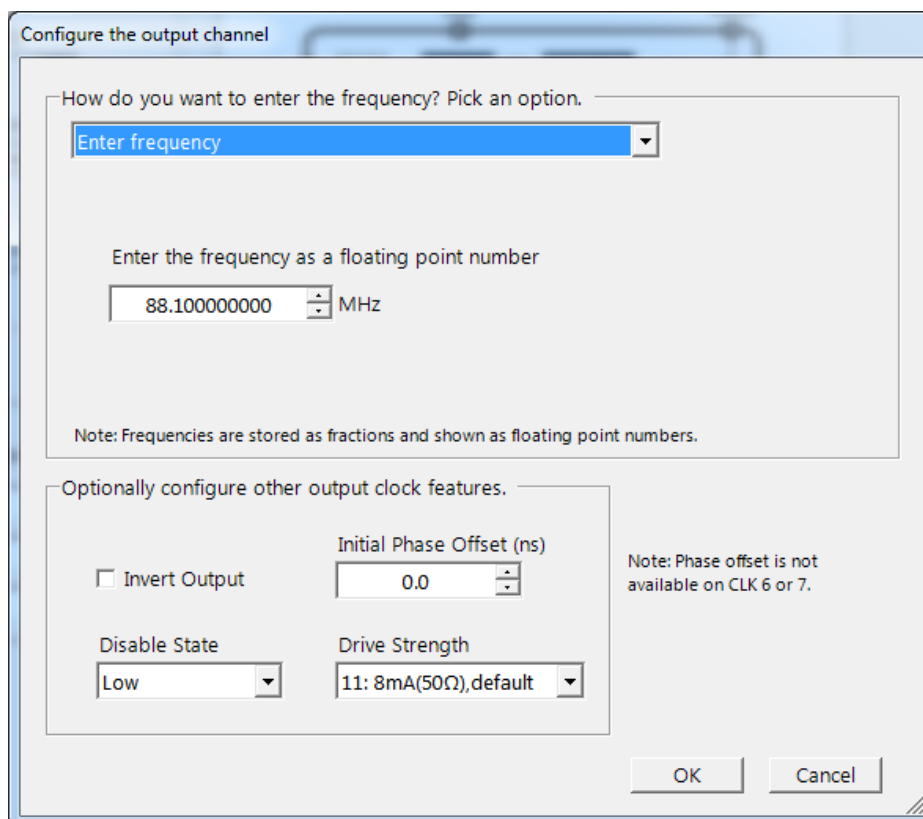
As you can see, the annoying part here is figuring out the best choice for PLL multiplier & divider! SiLabs has a desktop application called **ClockBuilder** (<https://adafru.it/dPj>) that can do some calculation of the PLL divider/multiplier for you.

It's windows only, but you only need to use it once for calculation.

Install and run, select the **Si5351A** with 3 outputs, and **Do not connect to the EVB**

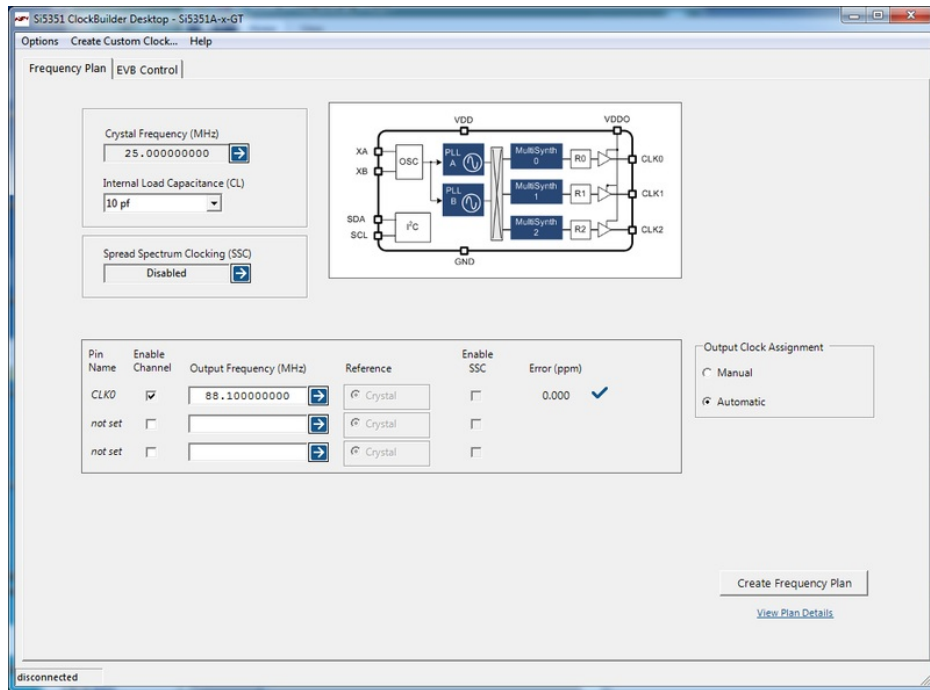


Enable the output you want, and set the frequency as floating point or fraction

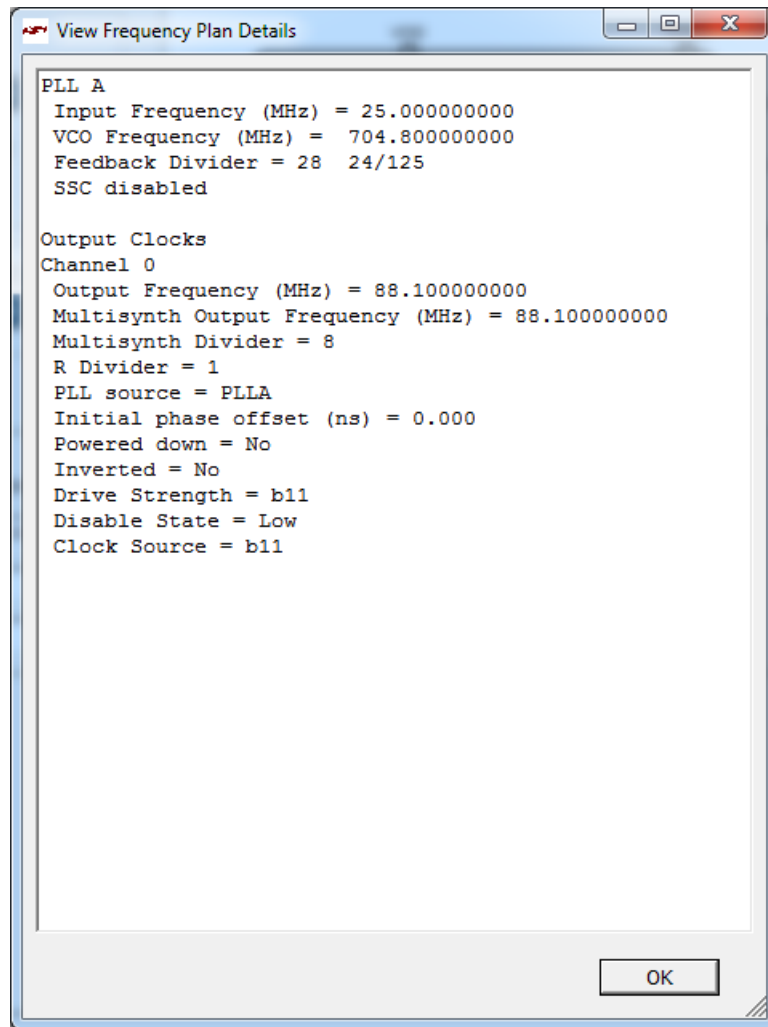


Set up the crystal to be 25 MHz (the default is 27 MHz)





Click on **Create Frequency Plan** to see the PLL and divider setups!



Earlier versions of this chip only take a divider of 900 or less, and our library doesn't let you select > 900 for the integer div. So if you get a higher value from the calculator, you may need to adjust it!

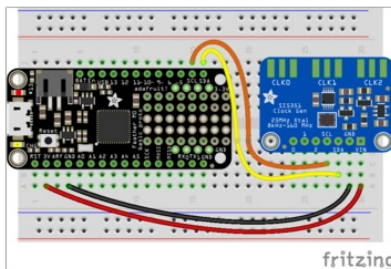
## CircuitPython Code

It's easy to use the SI5351 clock generator with Python or CircuitPython, and the [Adafruit CircuitPython SI5351](https://adafru.it/C5C) (<https://adafru.it/C5C>) module. This module allows you to easily write Python code that controls the clock output of the board.

You can use this sensor with any CircuitPython microcontroller board or with a computer that has GPIO and Python thanks to [Adafruit\\_Blinka](https://adafru.it/BSN), our [CircuitPython-for-Python compatibility library](https://adafru.it/BSN) (<https://adafru.it/BSN>).

## CircuitPython Microcontroller Wiring

First wire up a SI5351 to your board exactly as shown on the previous pages for Arduino using an I2C connection. Here's an example of wiring a Feather M0 to the sensor with I2C:

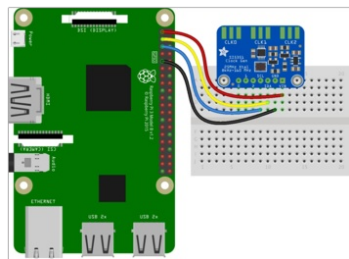


- Board 3V to sensor VIN
- Board GND to sensor GND
- Board SCL to sensor SCL
- Board SDA to sensor SDA

## Python Computer Wiring

Since there's *dozens* of Linux computers/boards you can use we will show wiring for Raspberry Pi. For other platforms, [please visit the guide for CircuitPython on Linux to see whether your platform is supported](https://adafru.it/BSN) (<https://adafru.it/BSN>).

Here's the Raspberry Pi wired with I2C:



- Pi 3V3 to sensor VIN
- Pi GND to sensor GND
- Pi SCL to sensor SCL
- Pi SDA to sensor SDA

## CircuitPython Installation of SI5351 Library

Next you'll need to install the [Adafruit CircuitPython SI5351](https://adafru.it/C5C) (<https://adafru.it/C5C>) library on your CircuitPython board.

First make sure you are running the [latest version of Adafruit CircuitPython](https://adafru.it/Amd) (<https://adafru.it/Amd>) for your board.

Next you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from [Adafruit's CircuitPython library bundle](https://adafru.it/zdx) (<https://adafru.it/zdx>). Our introduction guide has [a great page on how to install the library bundle](https://adafru.it/ABU) (<https://adafru.it/ABU>) for both express and non-express boards.



Remember for non-express boards like the, you'll need to manually install the necessary libraries from the bundle:

- `adafruit_si5351.mpy`
- `adafruit_bus_device`

Before continuing make sure your board's lib folder or root filesystem has the `adafruit_si5351.mpy`, and `adafruit_bus_device` files and folders copied over.

Next [connect to the board's serial REPL \(https://adafru.it/Awz\)](https://adafru.it/Awz) so you are at the CircuitPython >>> prompt.

## Python Installation of SI5351 Library

---

You'll need to install the Adafruit\_Blinka library that provides the CircuitPython support in Python. This may also require enabling I2C on your platform and verifying you are running Python 3. [Since each platform is a little different, and Linux changes often, please visit the CircuitPython on Linux guide to get your computer ready \(https://adafru.it/BSN\)!](#)

Once that's done, from your command line run the following command:

- `sudo pip3 install adafruit-circuitpython-si5351`

If your default Python is version 3 you may need to run 'pip' instead. Just make sure you aren't trying to use CircuitPython on Python 2.x, it isn't supported!

## CircuitPython & Python Usage

---

To demonstrate the usage of the sensor we'll initialize it and control the clocks from the board's Python REPL. Run the following code to import the necessary modules and initialize the I2C connection with the sensor:

```
import board
import busio
import adafruit_si5351
i2c = busio.I2C(board.SCL, board.SDA)
si5351 = adafruit_si5351.SI5351(i2c)
```

Now you're ready to configure the PLLs and clock outputs on the chip. The SI5351 is somewhat complex and allows quite many configurations to balance the accuracy and range of clock output values. You'll want to [review the datasheet \(https://adafru.it/C5E\)](#) to understand the capabilities and features of the chip.

## Configure Phase Lock Loops

---

First you'll need to configure one of the two PLLs (phase-locked loops, special hardware that can 'multiply' the 25 MHz crystal to much faster speeds) which will be used as the source of any of the clock outputs. The SI5351 object has two PLL objects you can manipulate:

- `pll_a` This is the PLL A hardware on the chip.
- `pll_b` This is the PLL B hardware on the chip.

You can configure each PLL's multiplier by calling the `configure_integer` or `configure_fractional` function. The integer configuration lets you set an integer multiplier for the 25 MHz base crystal frequency. This is the most accurate way to control frequency but you can only set integer multiples. For example to set PLL A to 500 MHz you'd set an integer

multiplier of 20 (25 MHz \* 20 = 500 MHz):

```
si5351.pll_a.configure_integer(20)
```

**Don't forget, if you read the datasheet you'll see the Si5351 only supports a PLL multiple of 15-90!**

If you're curious you can read the `frequency` property of the PLL to read back what frequency it is configured to use:

```
print('PLL A: {0} MHz'.format(si5351.pll_a.frequency/1000000))
```

```
>>> si5351.pll_a.configure_integer(20)
>>> print('PLL A: {0}mhz'.format(si5351.pll_a.frequency/1000000))
PLL A: 500.0mhz
>>>
```

The other way to configure a PLL is with a fractional multiplier. This allows you to set both the integer multiple and a fractional multiple (specified by a numerator and denominator). Fractional multipliers can let you specify an even greater range of frequencies, but they might not be accurate and could be susceptible to jitter over time.

For example to configure PLL B to 512.5 MHz you can specify a multiplier of 20.5 by calling:

```
si5351.pll_b.configure_fractional(20, 1, 2)
```

Notice the `configure_fractional` function takes in again the integer multiple (20) but also adds a numerator (1) and denominator (2) value. The actual multiplier will be the combination of the integer multiplier plus the numerator/denominator. In this case 20.5 or 20 1/2 is the multiplier.

Again you can read the `frequency` property to confirm the value that was set.

```
print('PLL B: {0} MHz'.format(si5351.pll_b.frequency/1000000))
```

```
>>> si5351.pll_b.configure_fractional(20, 1, 2)
>>> print('PLL B: {0}mhz'.format(si5351.pll_b.frequency/1000000))
PLL B: 512.5mhz
>>>
```

## Configure Clock Outputs

Once a PLL is configured you can then configure any of the three clock outputs of the Si5351. Each clock output is fed in a PLL as input and will divide it down to a final frequency value. Any PLL can be plugged in to any clock output, but remember you only have two PLLs to configure and use.

Each of the clocks is exposed on the Si5351 object:

- `clock_0` The clock 0 output.
- `clock_1` The clock 1 output.
- `clock_2` The clock 2 output.

Just like configuring the PLLs you can configure the clock divider with a `configure_integer` and `configure_fractional` function. Again the same trade-off of precision vs. range applies for integer vs. fractional dividers applies. For example to set clock 0 to use PLL A as a source (running at 500 MHz as configured above) and a divider of 4 you

could call:

```
si5351.clock_0.configure_integer(si5351.pll_a, 4)
```

Notice you tell the `configure_integer` function both the PLL to use and the integer divider value. You can point at either of the `pll_a` or `pll_b` objects on the Si5351 to use a specific PLL as the source.

The final frequency set by the clock will then be configured as the PLL source frequency (500 MHz here) divided by the divider value (4), or 125 MHz. You can check by reading the `frequency` property of the clock:

```
print('Clock 0: {0:0.3f} MHz'.format(si5351.clock_0.frequency/1000000))
```

```
>>> si5351.clock_0.configure_integer(si5351.pll_a, 4)
>>> print('Clock 0: {0:0.3f}mhz'.format(si5351.clock_0.frequency/1000000))
Clock 0: 125.000mhz
>>>
```

Like the PLL configuration you can also set a fractional divider with the `configure_fractional` function on the clock. Let's set clock 1 to have PLL B as the source and a divider of 4.5:

```
si5351.clock_1.configure_fractional(si5351.pll_b, 4, 1, 2)
```

If you do the math you should expect a source PLL of 512.5 MHz divided by 4.5 would yield a clock output of 113.8889 MHz. Check the `frequency` property to be sure:

```
print('Clock 1: {0:0.3f} MHz'.format(si5351.clock_1.frequency/1000000))
```

```
>>> si5351.clock_1.configure_fractional(si5351.pll_b, 4, 1, 2)
>>> print('Clock 1: {0:0.3f}mhz'.format(si5351.clock_1.frequency/1000000))
Clock 1: 113.888mhz
>>>
```

## Enabling Outputs

There's one last thing to do before the clock outputs will actually work, you need to enable them. You can enable all the clock outputs by setting the `outputs_enabled` property to `True`:

```
si5351.outputs_enabled = True
```

Now the clock 0, 1, 2 outputs should output a square wave at the configured frequency based on their PLL and multipliers & dividers. You might need to use an oscilloscope that's very fast to measure some of these outputs as the Si5351 can generate quite high clockspeeds well above 100 MHz!

If you want to disable the outputs set `outputs_enabled` to `False`.

That's all there is to using the Si5351 clock generator with CircuitPython! Here's a complete example that will configure all 3 clocks in various ways. Save this as `code.py` on your board and read the comments to see how it configures each clock.

## Full Example Code



```

# Simple demo of the SI5351 clock generator.
# This is like the Arduino library example:
#   https://github.com/adafruit/Adafruit_SI5351_Library/blob/master/examples/si5351/si5351.ino
# Which will configure the chip with:
# - PLL A at 900mhz
# - PLL B at 616.66667mhz
# - Clock 0 at 112.5mhz, using PLL A as a source divided by 8
# - Clock 1 at 13.553115mhz, using PLL B as a source divided by 45.5
# - Clock 2 at 10.76khz, using PLL B as a source divided by 900 and further
#   divided with an R divider of 64.
import board
import busio

import adafruit_si5351

# Initialize I2C bus.
i2c = busio.I2C(board.SCL, board.SDA)

# Initialize SI5351.
si5351 = adafruit_si5351.SI5351(i2c)
# Alternatively you can specify the I2C address if it has been changed:
#si5351 = adafruit_si5351.SI5351(i2c, address=0x61)

# Now configure the PLLs and clock outputs.
# The PLLs can be configured with a multiplier and division of the on-board
# 25mhz reference crystal. For example configure PLL A to 900mhz by multiplying
# by 36. This uses an integer multiplier which is more accurate over time
# but allows less of a range of frequencies compared to a fractional
# multiplier shown next.
si5351.pll_a.configure_integer(36) # Multiply 25mhz by 36
print('PLL A frequency: {0}mhz'.format(si5351.pll_a.frequency/1000000))

# And next configure PLL B to 616.6667mhz by multiplying 25mhz by 24.667 using
# the fractional multiplier configuration. Notice you specify the integer
# multiplier and then a numerator and denominator as separate values, i.e.
# numerator 2 and denominator 3 means 2/3 or 0.667. This fractional
# configuration is susceptible to some jitter over time but can set a larger
# range of frequencies.
si5351.pll_b.configure_fractional(24, 2, 3) # Multiply 25mhz by 24.667 (24 2/3)
print('PLL B frequency: {0}mhz'.format(si5351.pll_b.frequency/1000000))

# Now configure the clock outputs. Each is driven by a PLL frequency as input
# and then further divides that down to a specific frequency.
# Configure clock 0 output to be driven by PLL A divided by 8, so an output
# of 112.5mhz (900mhz / 8). Again this uses the most precise integer division
# but can't set as wide a range of values.
si5351.clock_0.configure_integer(si5351.pll_a, 8)
print('Clock 0: {0}mhz'.format(si5351.clock_0.frequency/1000000))

# Next configure clock 1 to be driven by PLL B divided by 45.5 to get
# 13.5531mhz (616.6667mhz / 45.5). This uses fractional division and again
# notice the numerator and denominator are explicitly specified. This is less
# precise but allows a large range of frequencies.
si5351.clock_1.configure_fractional(si5351.pll_b, 45, 1, 2) # Divide by 45.5 (45 1/2)
print('Clock 1: {0}mhz'.format(si5351.clock_1.frequency/1000000))

# Finally configure clock 2 to be driven by PLL B divided once by 900 to get
# down to 685 15 khz and then further divided by a special R divider that

```

```

# down to 685.15 khz and then further divided by a special R divider that
# divides 685.15 khz by 64 to get a final output of 10.706khz.
si5351.clock_2.configure_integer(si5351.pll_b, 900)
# Set the R divider, this can be a value of:
# - R_DIV_1: divider of 1
# - R_DIV_2: divider of 2
# - R_DIV_4: divider of 4
# - R_DIV_8: divider of 8
# - R_DIV_16: divider of 16
# - R_DIV_32: divider of 32
# - R_DIV_64: divider of 64
# - R_DIV_128: divider of 128
si5351.clock_2.r_divider = adafruit_si5351.R_DIV_64
print('Clock 2: {0}khz'.format(si5351.clock_2.frequency/1000))

# After configuring PLLs and clocks, enable the outputs.
si5351.outputs_enabled = True
# You can disable them by setting false.

```

## Python Docs

[Python Docs \(https://adafru.it/C5F\)](https://adafru.it/C5F)



