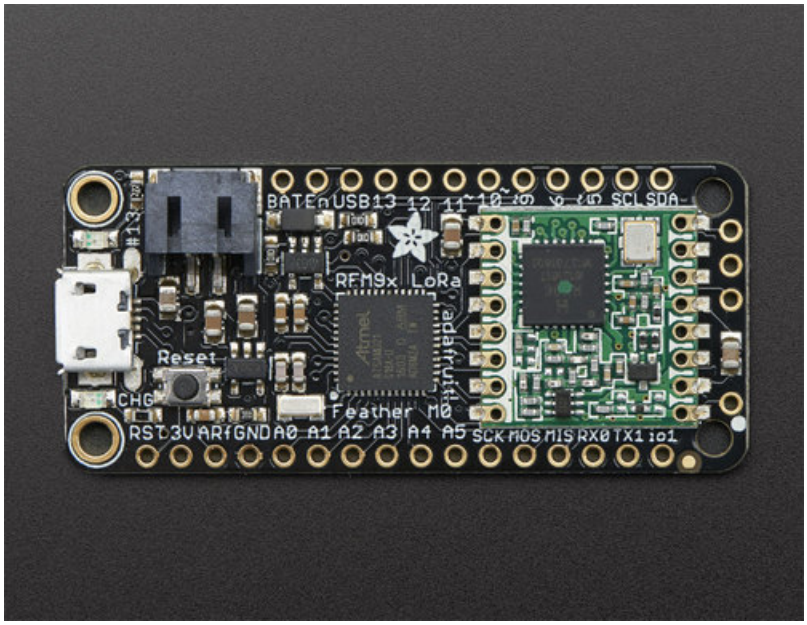
adafruit learning system

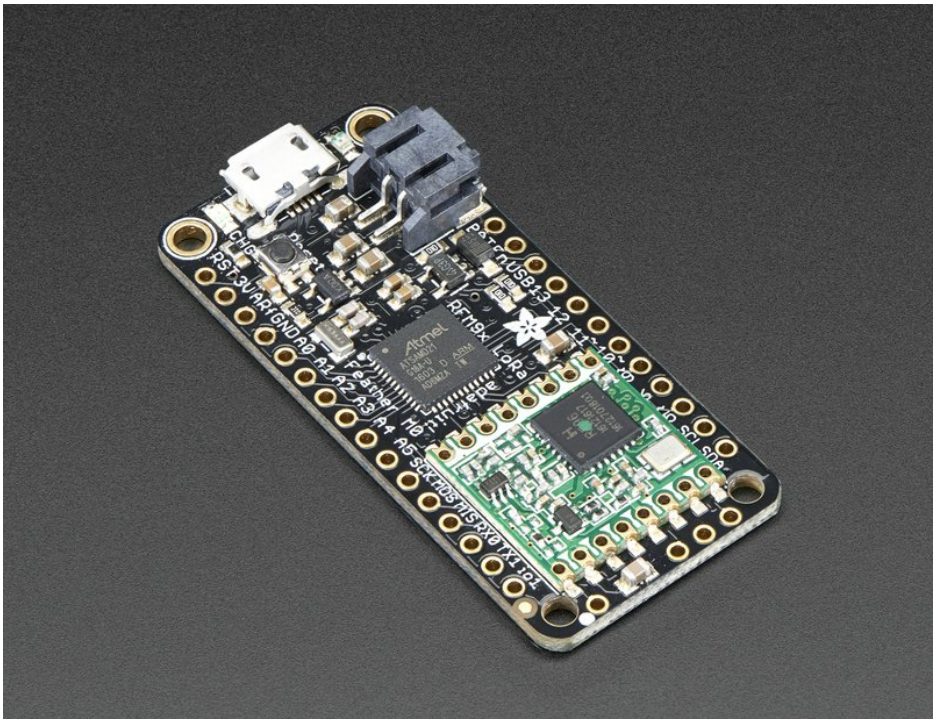
Adafruit Feather M0 Radio with LoRa Radio Module

Created by lady ada

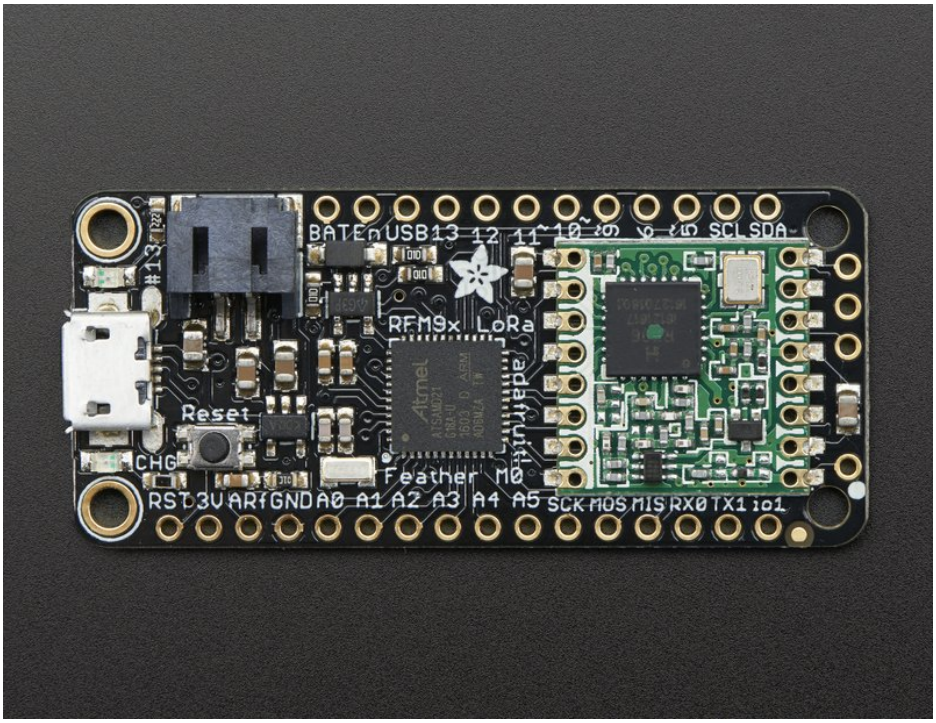


Last updated on 2020-06-09 09:00:45 PM EDT

Overview

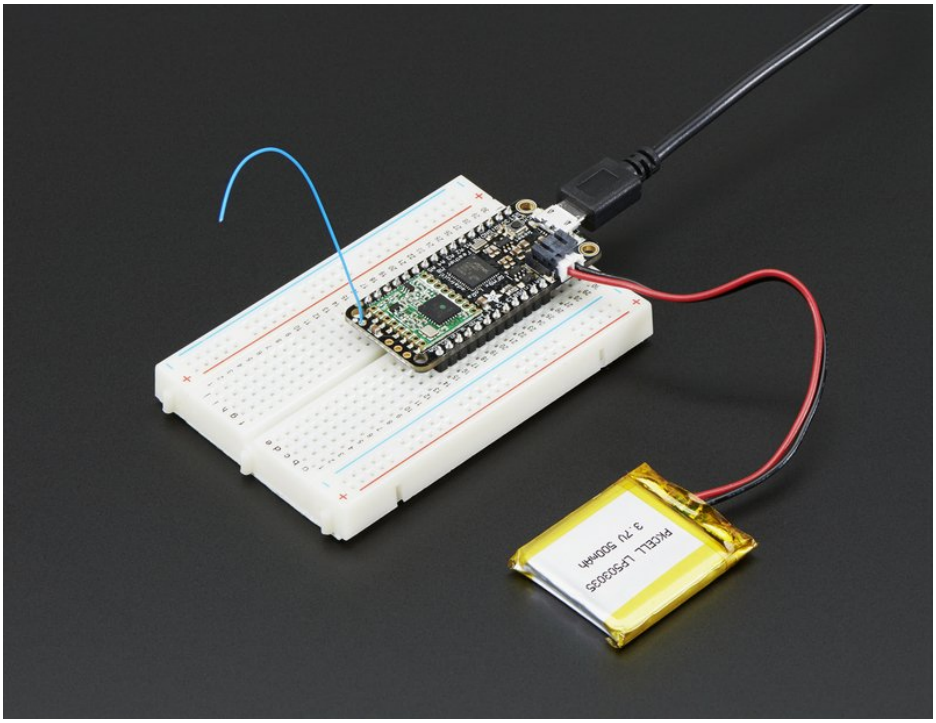


This is the **Adafruit Feather M0 RFM95 LoRa Radio (433 or 900 MHz)** - our take on an microcontroller with a 'Long Range (LoRa) (https://adafru.it/mFZ)" packet radio transceiver with built in USB and battery charging. Its an Adafruit Feather M0 with a Long Range radio module cooked in! Great for making wireless networks that are more flexible than Bluetooth LE and without the high power requirements of WiFi. We have other boards in the Feather family, check'em out here (https://adafru.it/l7B).



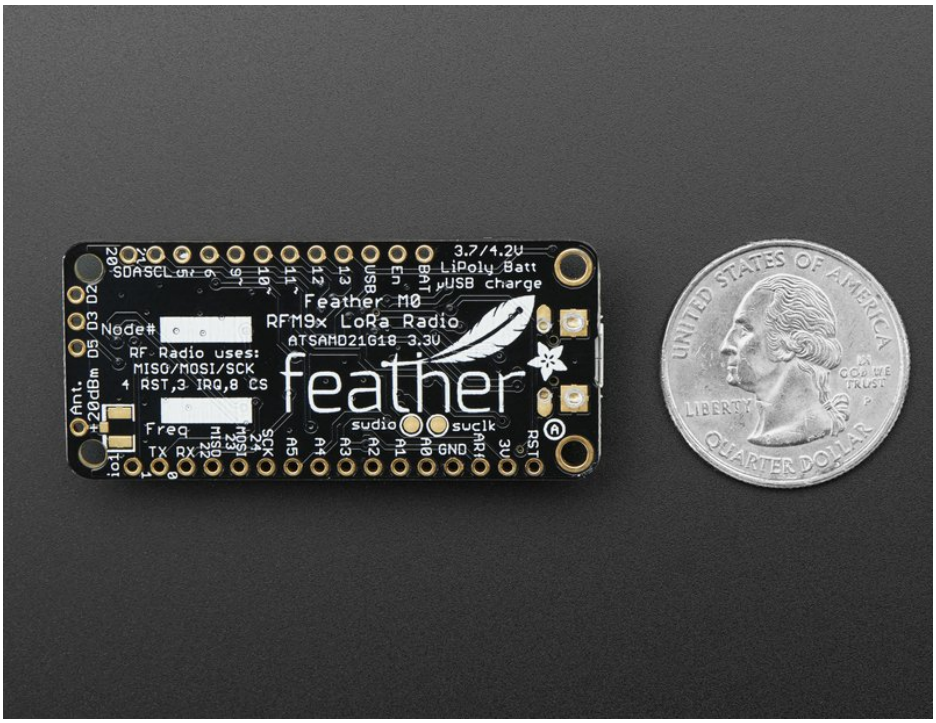
At the Feather M0's heart is an ATSAMD21G18 ARM Cortex M0 processor, clocked at 48 MHz and at 3.3V logic, the same one used in the new Arduino Zero (http://adafru.it/2843). This chip has a whopping 256K of FLASH (8x more than the Atmega328 or 32u4) and 32K of RAM (16x as much)! This chip comes with built in USB so it has USB-to-Serial program & debug capability built in with no need for an FTDI-like chip.

To make it easy to use for portable projects, we added a connector for any of our 3.7V Lithium polymer batteries and built in battery charging. You don't need a battery, it will run just fine straight from the micro USB connector. But, if you do have a battery, you can take it on the go, then plug in the USB to recharge. The Feather will automatically switch over to USB power when its available. We also tied the battery thru a divider to an analog pin, so you can measure and monitor the battery voltage to detect when you need a recharge.



Here's some handy specs! Like all Feather M0's you get:

- Measures 2.0" x 0.9" x 0.3" (51mm x 23mm x 8mm) without headers soldered in
- Light as a (large?) feather - 5.8 grams
- ATSAMD21G18 @ 48MHz with 3.3V logic/power
- No EEPROM
- 3.3V regulator with 500mA peak current output
- USB native support, comes with USB bootloader and serial port debugging
- You also get tons of pins - 20 GPIO pins
- Hardware Serial, hardware I2C, hardware SPI support
- 8 x PWM pins
- 10 x analog inputs
- 1 x analog output
- Built in 100mA lipoly charger with charging status indicator LED
- Pin #13 red LED for general purpose blinking
- Power/enable pin
- 4 mounting holes
- Reset button



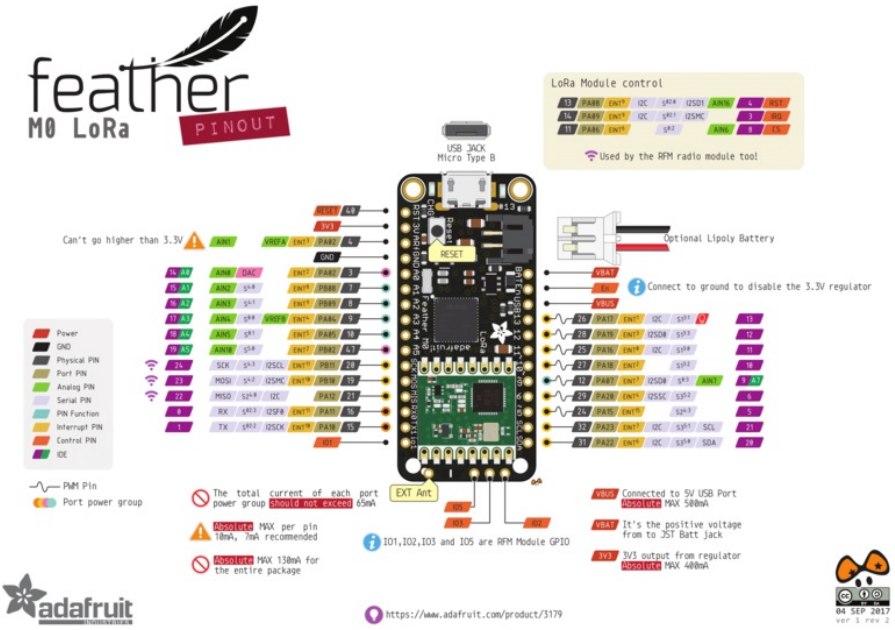
This **Feather M0 LoRa Radio** uses the extra space left over to add an RFM9x LoRa 868/915 MHz radio module. These radios are not good for transmitting audio or video, but they do work quite well for small data packet transmission when you need more range than 2.4 GHz (BT, BLE, WiFi, ZigBee).

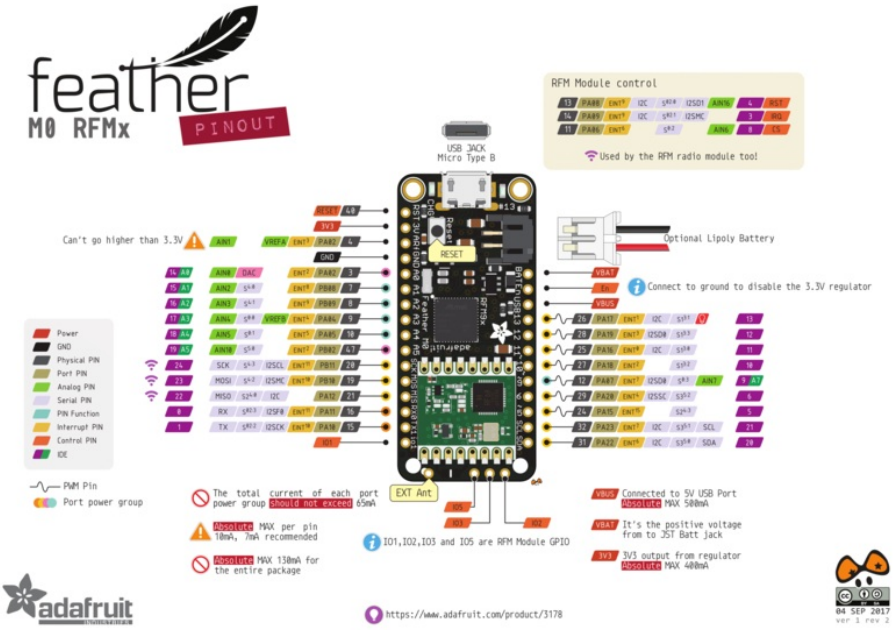
- SX1276 LoRa® based module with SPI interface
- Packet radio with ready-to-go Arduino libraries
- Uses the license-free ISM bands (ITU "Europe" @ 433MHz and ITU "Americas" @ 900MHz)
- +5 to +20 dBm up to 100 mW Power Output Capability (power output selectable in software)
- ~300uA during full sleep, ~120mA peak during +20dBm transmit, ~40mA during active radio listening.
- Simple wire antenna or spot for uFL connector

Our initial tests with default library settings: over 1.2mi/2Km line-of-sight with wire quarter-wave antennas. (With setting tweaking and directional antennas, 20Km is possible (https://adafru.it/mGa)).

Comes fully assembled and tested, with a USB bootloader that lets you quickly use it with the Arduino IDE. We also toss in some headers so you can solder it in and plug into a solderless breadboard. You will need to cut and solder on a small piece of wire (any solid or stranded core is fine) in order to create your antenna. **Lipoly battery and USB cable not included** but we do have lots of options in the shop if you'd like!

Pinouts





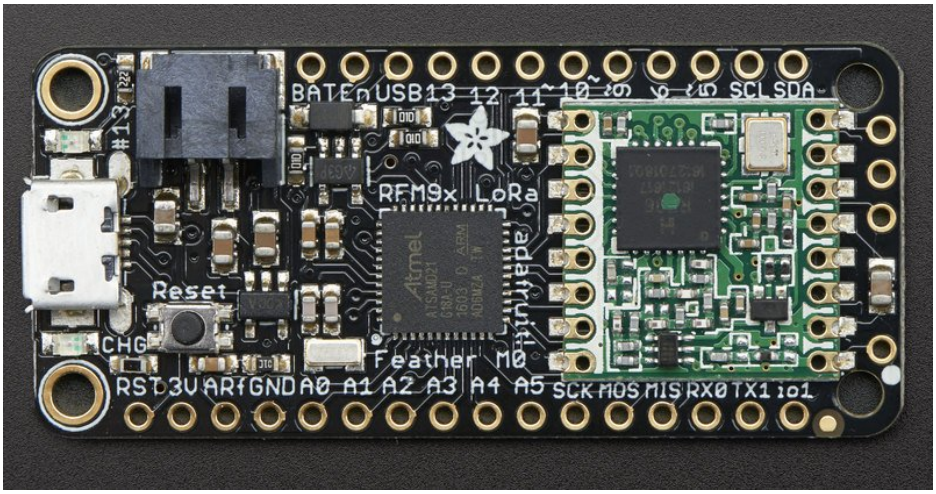
Note AREF in the diagram should be marked PA03 not PA02

The Feather M0 Radio is chock-full of microcontroller goodness. There's also a lot of pins and ports. We'll take you a tour of them now!

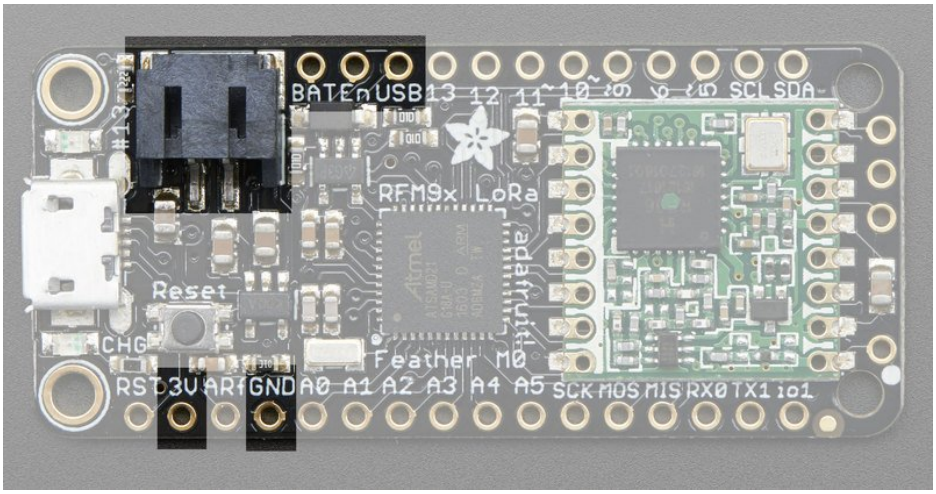
Note that the pinouts are identical for both the Feather M0 RFM69 and LoRa radios - you can look at the silkscreen of the Feather to see it says "RFM69" or "LoRa"

Pinouts are also the same for both 433MHz and 900Mhz. You can tell the difference by looking for a colored dot on

the chip or crystal of the radio, green/blue is 900MHz & red is 433MHz

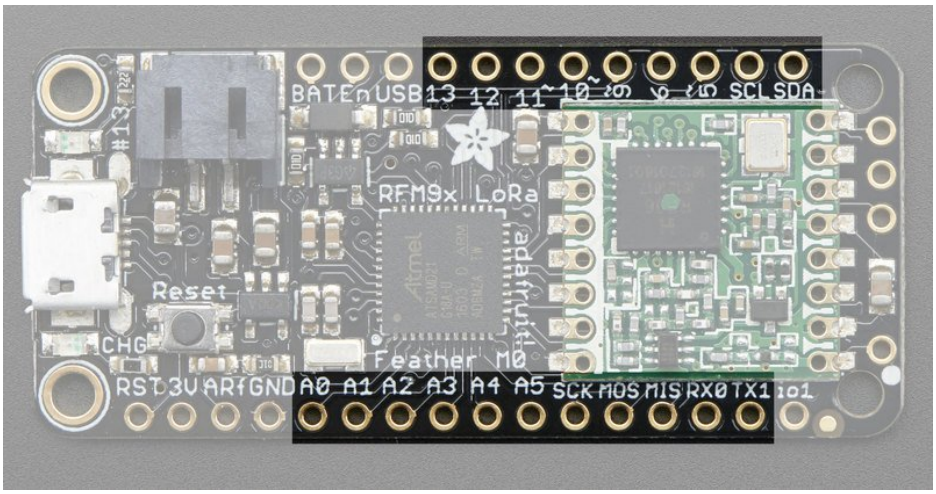


Power Pins



- **GND** - this is the common ground for all power and logic
- **BAT** - this is the positive voltage to/from the JST jack for the optional Lipoly battery
- **USB** - this is the positive voltage to/from the micro USB jack if connected
- **EN** - this is the 3.3V regulator's enable pin. It's pulled up, so connect to ground to disable the 3.3V regulator
- **3V** - this is the output from the 3.3V regulator, it can supply 500mA peak

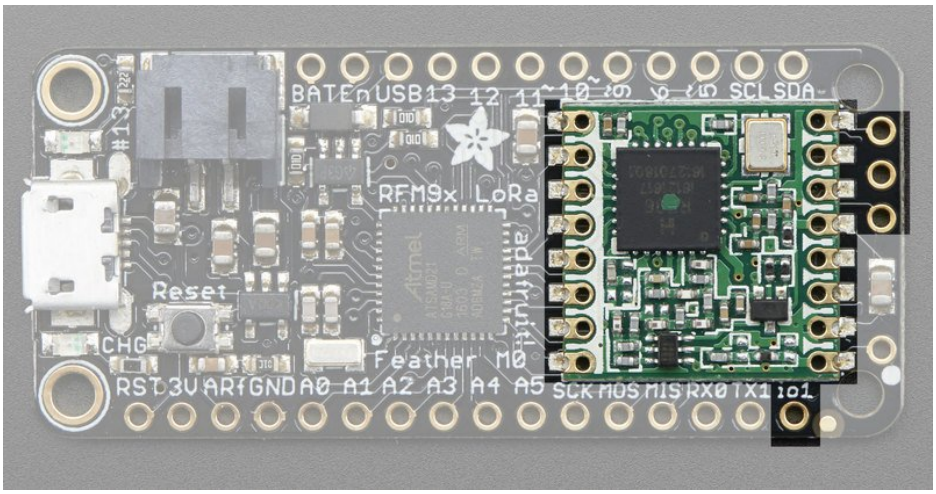
Logic pins



This is the general purpose I/O pin set for the microcontroller.
All logic is 3.3V
Nearly all pins can do PWM output
All pins can be interrupt inputs

- **#0 / RX** - GPIO #0, also receive (input) pin for **Serial1** (hardware UART), also can be analog input
- **#1 / TX** - GPIO #1, also transmit (output) pin for **Serial1**, also can be analog input
- **#20 / SDA** - GPIO #20, also the I2C (Wire) data pin. There's no pull up on this pin by default so when using with I2C, you may need a 2.2K-10K pullup.
- **#21 / SCL** - GPIO #21, also the I2C (Wire) clock pin. There's no pull up on this pin by default so when using with I2C, you may need a 2.2K-10K pullup.
- **#5** - GPIO #5
- **#6** - GPIO #6
- **#9** - GPIO #9, also analog input **A7**. This analog input is connected to a voltage divider for the lipoly battery so be aware that this pin naturally 'sits' at around 2VDC due to the resistor divider
- **#10** - GPIO #10
- **#11** - GPIO #11
- **#12** - GPIO #12
- **#13** - GPIO #13 and is connected to the **red LED** next to the USB jack
- **A0** - This pin is analog *input* **A0** but is also an analog *output* due to having a DAC (digital-to-analog converter). You can set the raw voltage to anything from 0 to 3.3V, unlike PWM outputs this is a true analog output
- **A1 thru A5** - These are each analog input as well as digital I/O pins.
- **SCK/MOSI/MISO** (GPIO **24/23/22**)- These are the hardware SPI pins, you can use them as everyday GPIO pins (but recommend keeping them free as they are best used for hardware SPI connections for high speed.

RFM/SemTech Radio Module



Since not all pins can be brought out to breakouts, due to the small size of the Feather, we use these to control the radio module

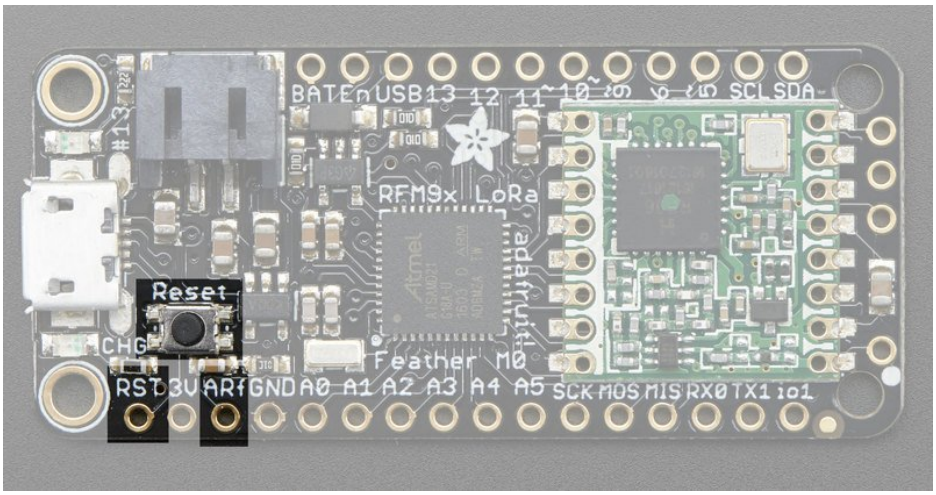
- **#8** - used as the radio **CS** (chip select) pin
- **#3** - used as the radio **GPIO0** / **IRQ** (interrupt request) pin.
- **#4** - used as the radio **Reset** pin

Since these are not brought out there should be no risk of using them by accident!

There are also breakouts for 3 of the RFM's GPIO pins (**IO1, IO2, IO3** and **IO5**). You probably wont need these for most uses of the Feather but they are available in case you need 'em!

The CS pin (#8) does not have a pullup built in so be sure to set this pin HIGH when not using the radio!

Other Pins!



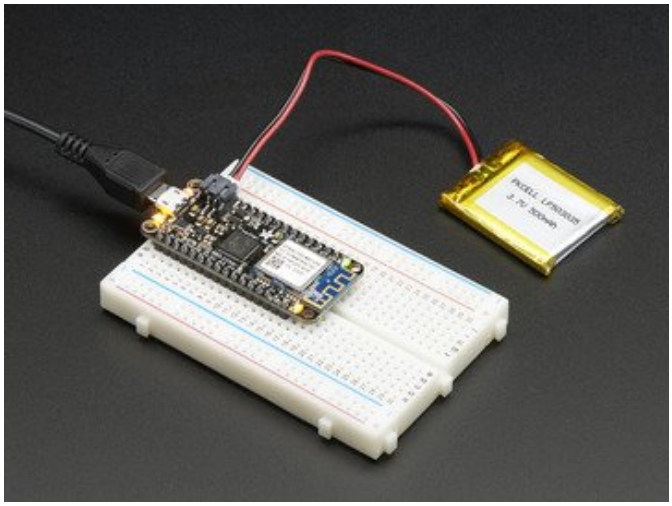
- **RST** - this is the Reset pin, tie to ground to manually reset the AVR, as well as launch the bootloader manually
- **ARef** - the analog reference pin. Normally the reference voltage is the same as the chip logic voltage (3.3V) but if you need an alternative analog reference, connect it to this pin and select the external AREF in your firmware. Can't go higher than 3.3V!

Assembly

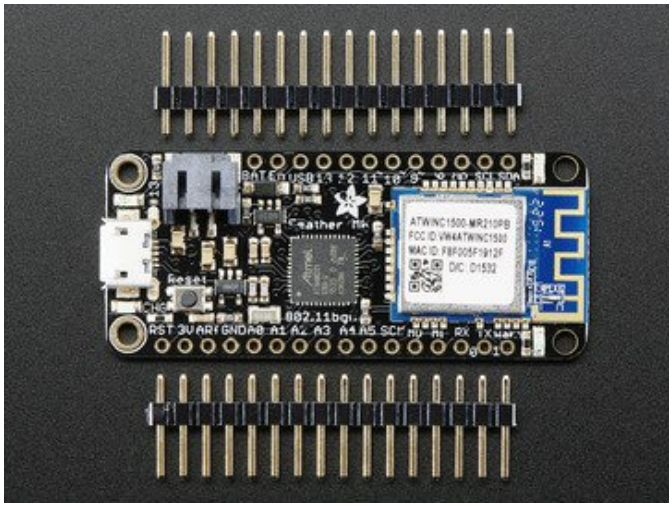
We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

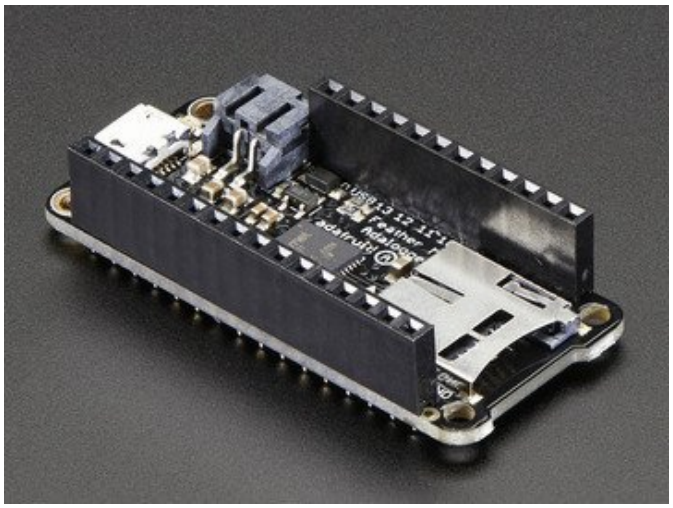
Header Options!

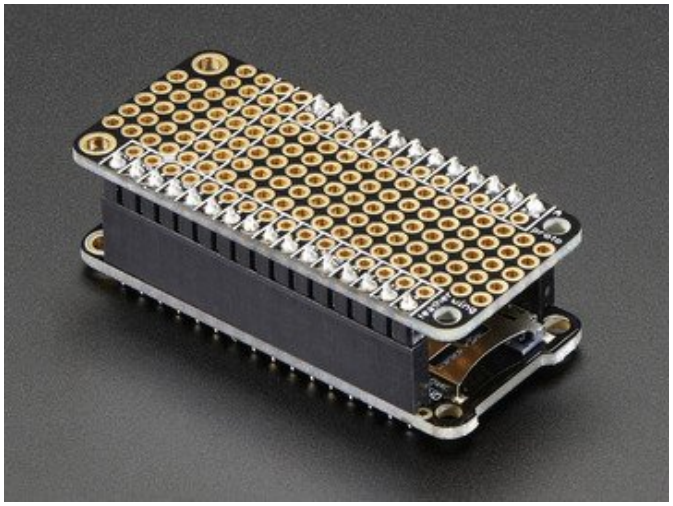
Before you go gung-ho on soldering, there's a few options to consider!



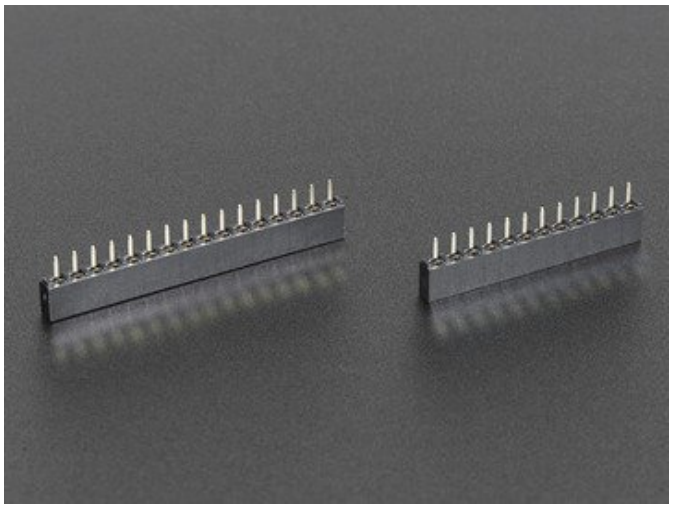
The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard



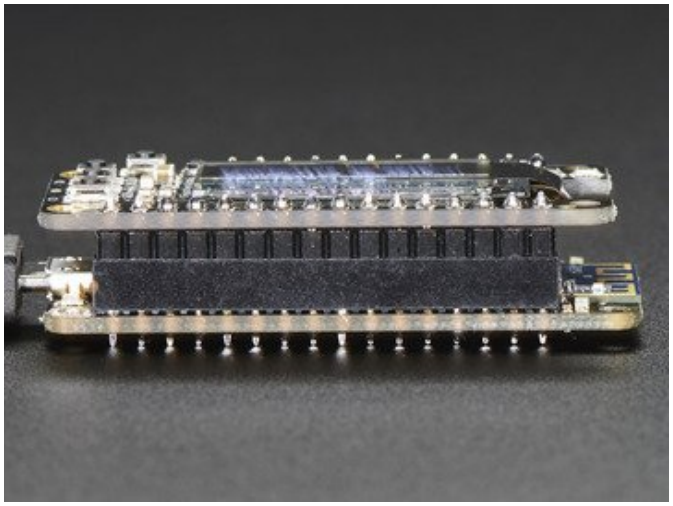


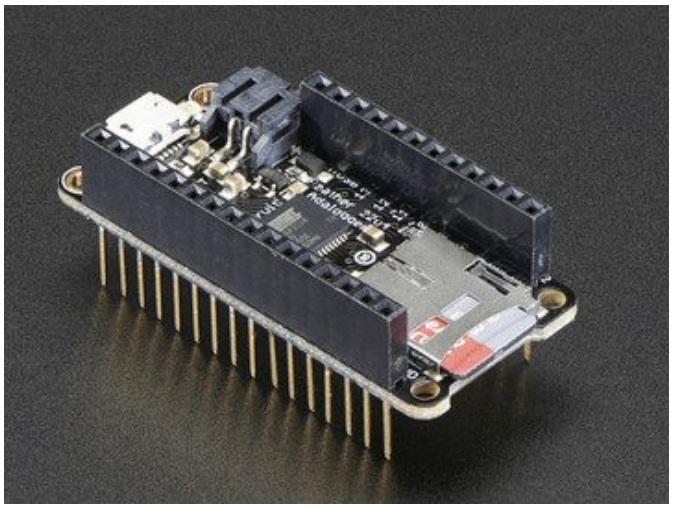


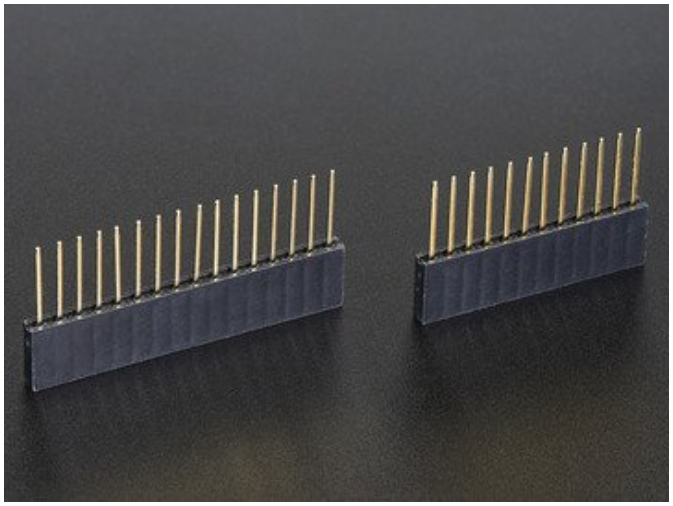
Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily



We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape

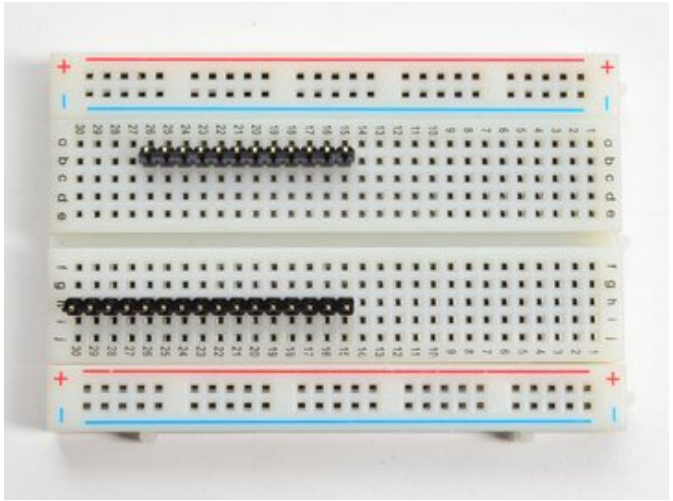






Finally, there's the "Stacking Header" option. This one is sort of the best-of-both-worlds. You get the ability to plug into a solderless breadboard *and* plug a featherwing on top. But its a little bulky

Soldering in Plain Headers

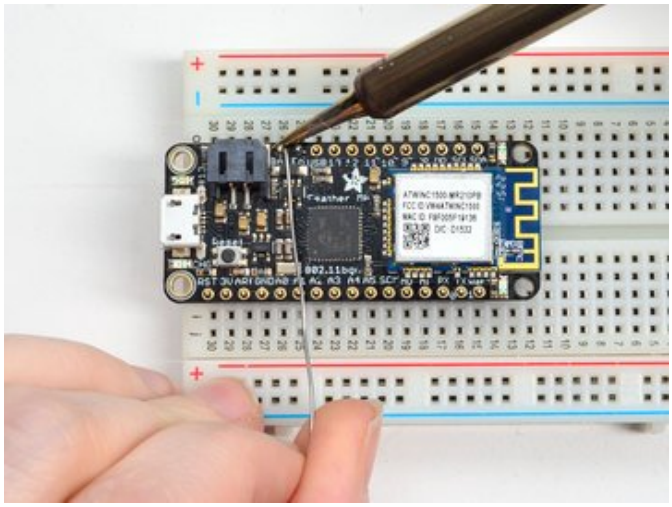


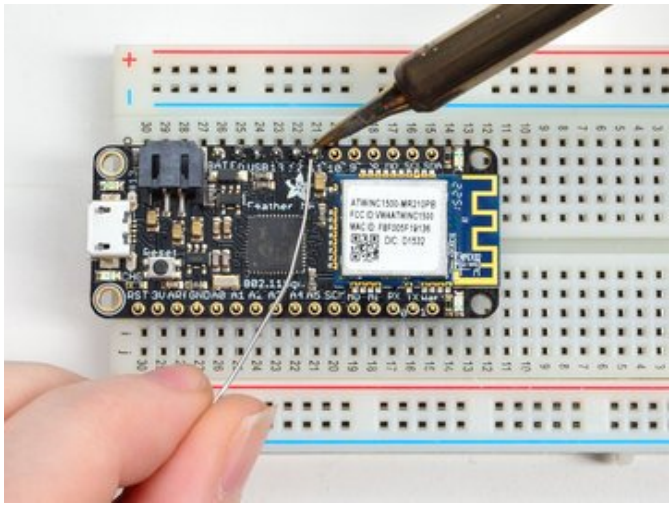
Prepare the header strip:

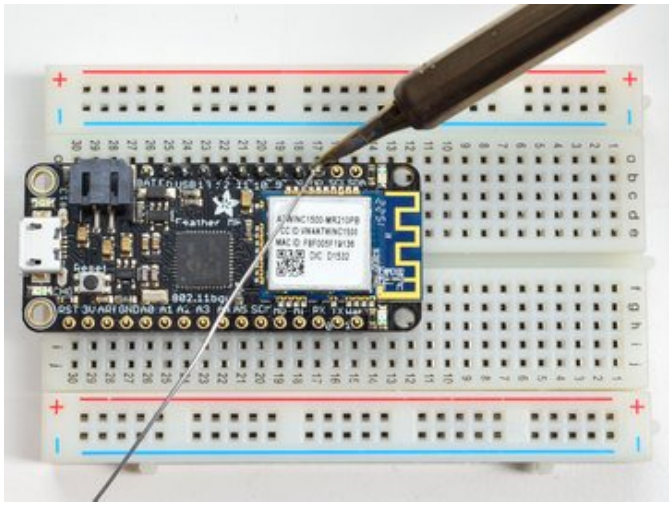
Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**

Add the breakout board:

Place the breakout board over the pins so that the short





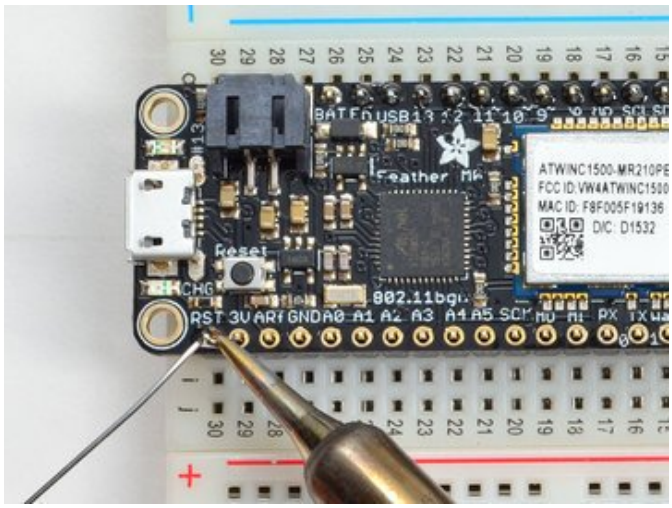


pins poke through the breakout pads

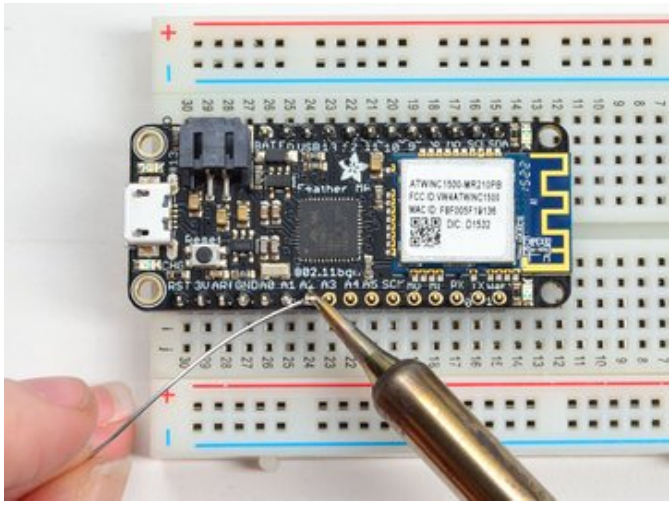
And Solder!

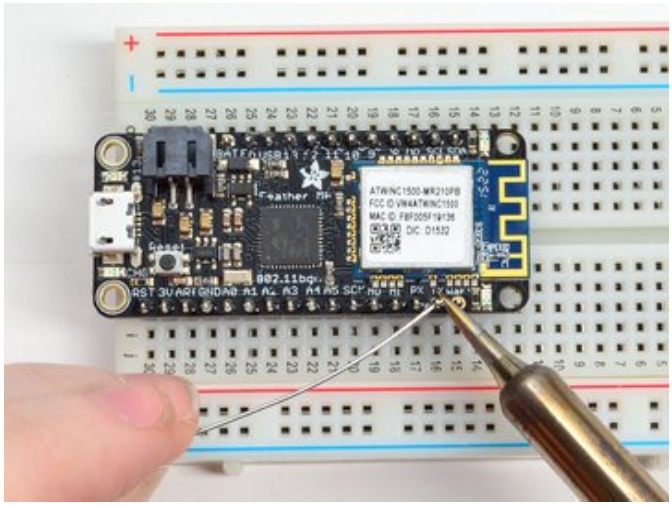
Be sure to solder all pins for reliable electrical contact.

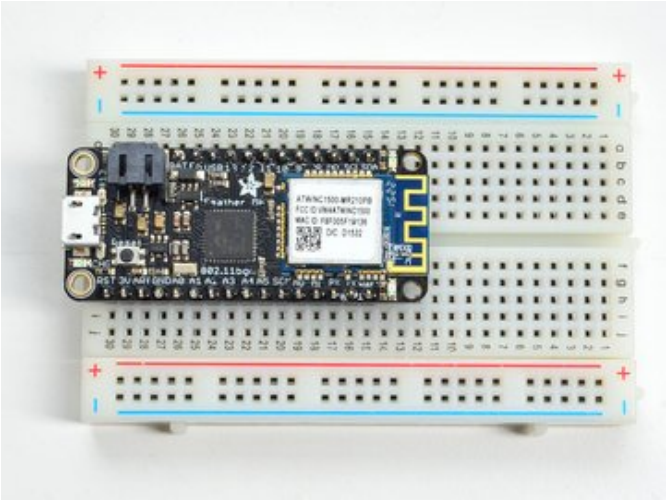
(For tips on soldering, be sure to check out our Guide to Excellent Soldering (https://adafru.it/aTk).



Solder the other strip as well.





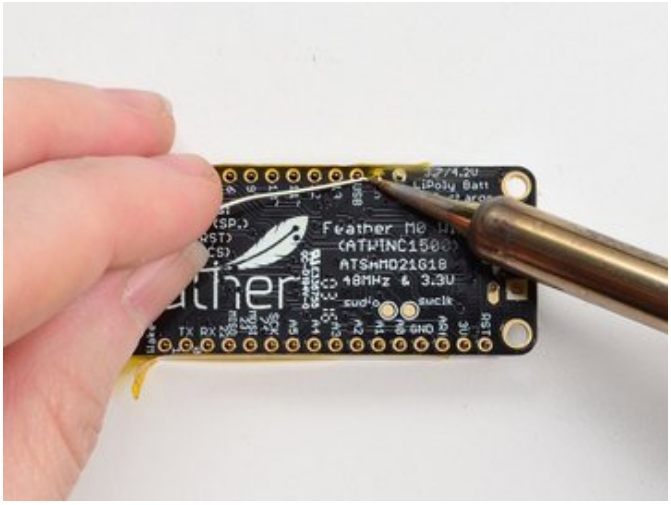


You're done! Check your solder joints visually and continue onto the next steps

Soldering on Female Header

Tape In Place

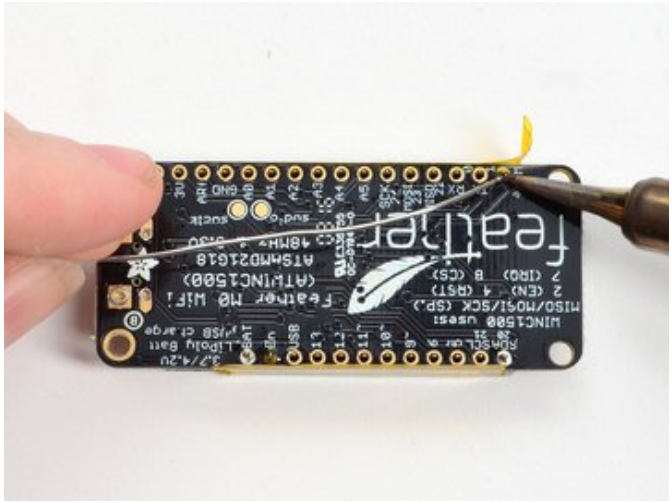
For sockets you'll want to tape them in place so when you flip over the board they don't fall out

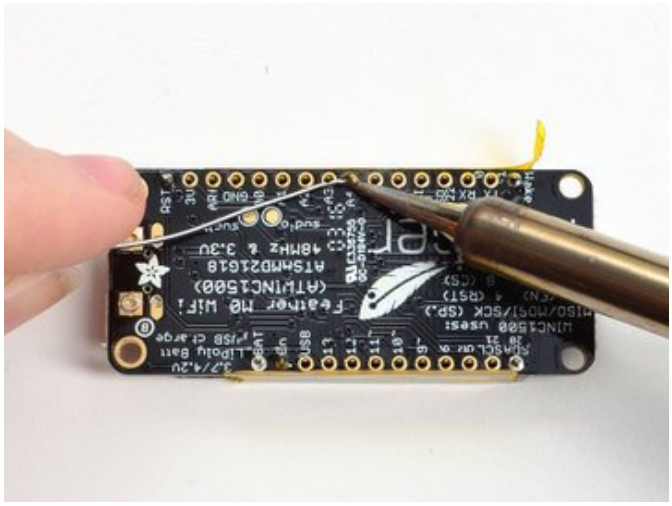


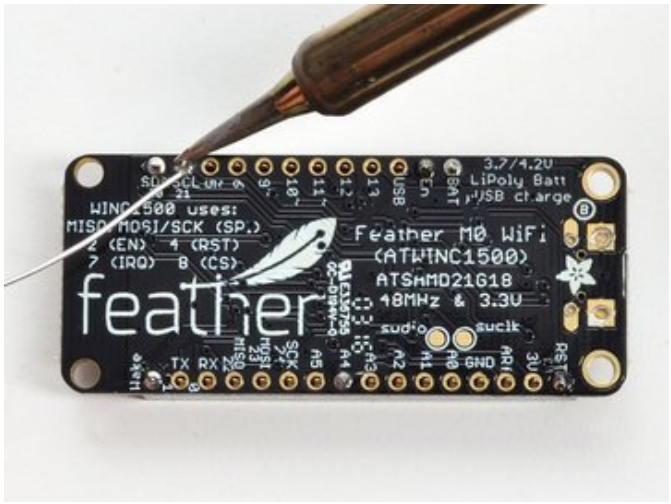
Flip & Tack Solder

After flipping over, solder one or two points on each strip, to 'tack' the header in place

feather
M0 WiFi


feather





And Solder!

Be sure to solder all pins for reliable electrical contact.

(For tips on soldering, be sure to check out our Guide to Excellent Soldering (https://adafru.it/aTk).

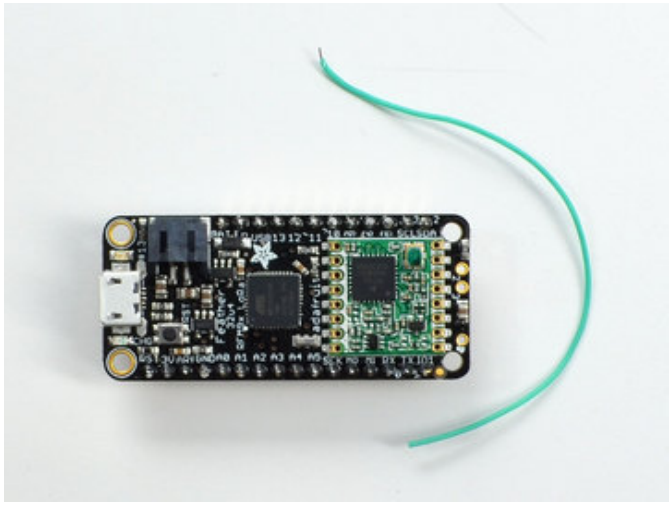
You're done! Check your solder joints visually and continue onto the next steps

Antenna Options

Your Feather Radio does not have a built-in antenna. Instead, you have two options for attaching an antenna. For most low cost radio nodes, a wire works great. If you need to put the Feather into an enclosure, soldering in uFL and using a uFL to SMA adapter will let you attach an external antenna

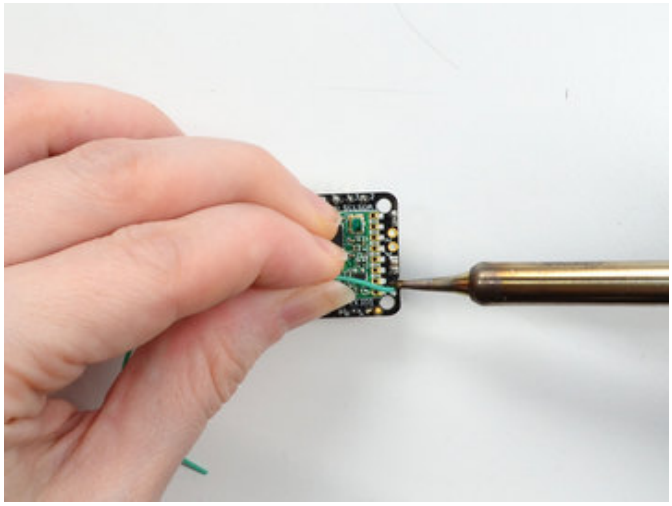
Wire Antenna

A wire antenna, aka "quarter wave whip antenna" is low cost and works very well! You just have to cut the wire down to the right length.

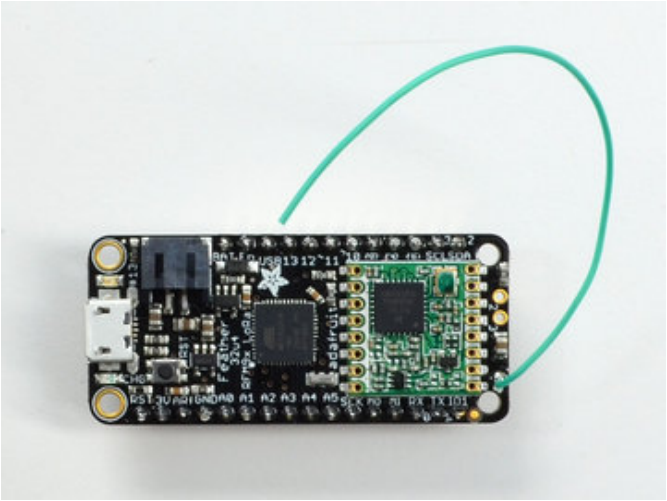


Cut a stranded or solid core wire to the proper length for the module/frequency

- **433 MHz** - 6.5 inches, or 16.5 cm
- **868 MHz** - 3.25 inches or 8.2 cm
- **915 MHz** - 3 inches or 7.8 cm



Strip a mm or two off the end of the wire, tin and solder into the **ANT** pad on the very right hand edge of the Feather



That's pretty much it, you're done!

uFL Antenna

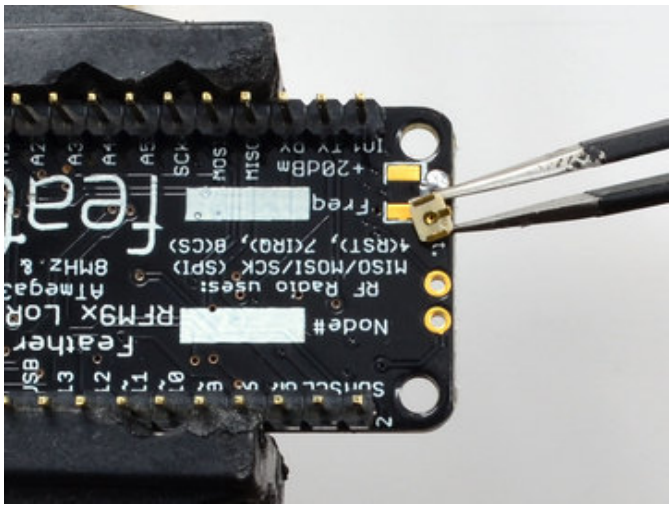
If you want an external antenna, you need to do a tiny bit more work but its not too difficult.

You'll need to get an SMT uFL connector, these are fairly standard (http://adafru.it/1661)

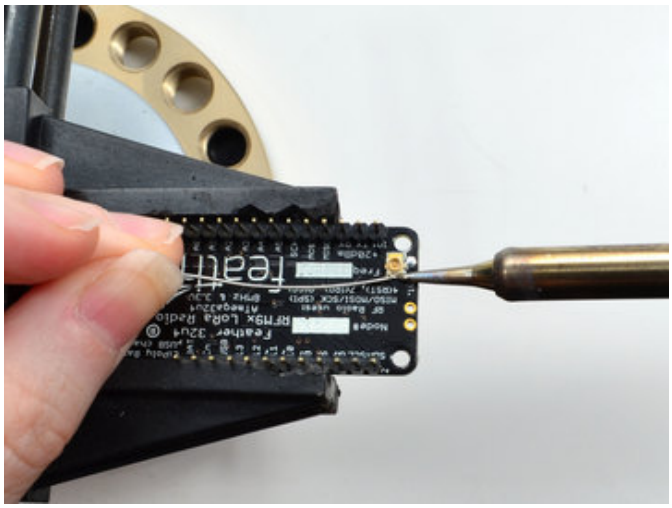
You'll also need a uFL to SMA adapter (http://adafru.it/851) (or whatever adapter you need for the antenna you'll be using, SMA is the most common

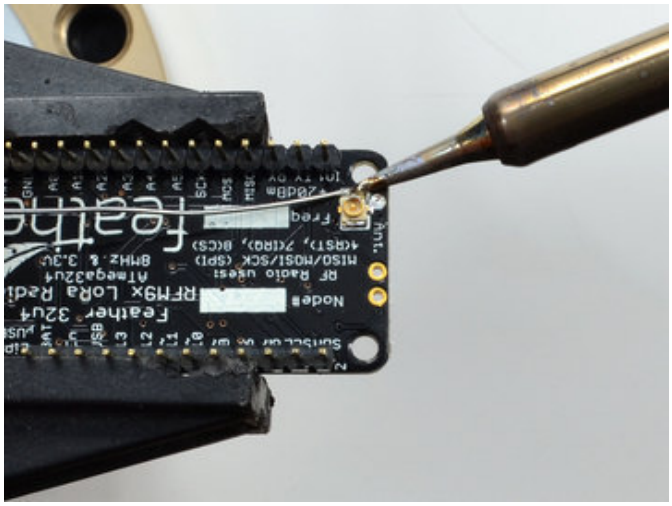
Of course, you will also need an antenna of some sort, that matches your radio frequency

uFL connectors are rated for 30 connection cycles, but be careful when connecting/disconnecting to not rip the pads off the PCB. Once a uFL/SMA adapter is connected, use strain relief!

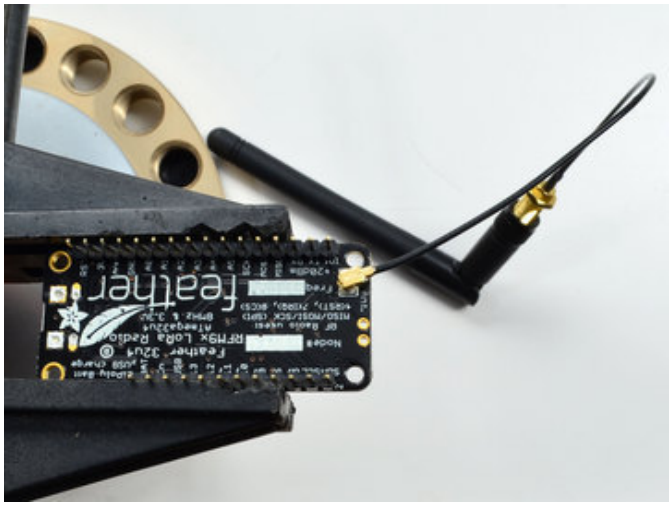


Check the bottom of the uFL connector, note that there's two large side pads (ground) and a little inlet pad. The other small pad is not used!

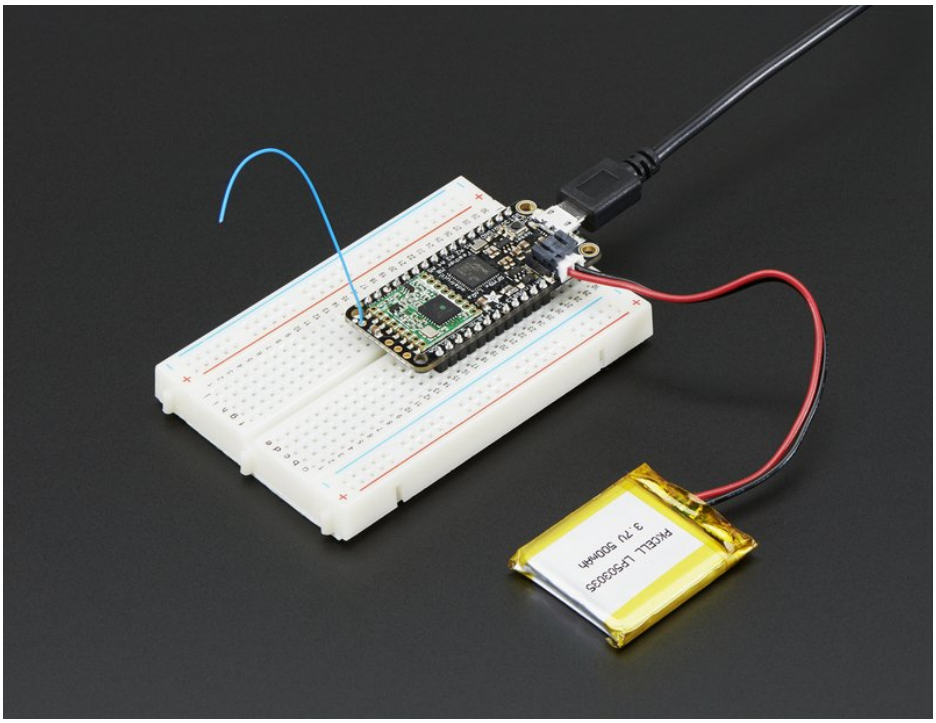




Solder all three pads to the bottom of the Feather



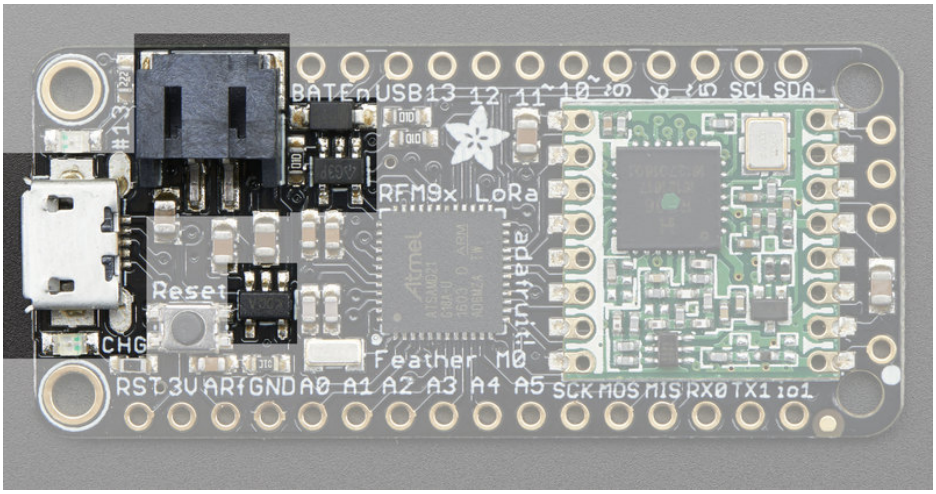
Once done attach your uFL adapter and antenna!



Battery + USB Power

We wanted to make the Feather easy to power both when connected to a computer as well as via battery. There's **two ways to power** a Feather. You can connect with a MicroUSB cable (just plug into the jack) and the Feather will regulate the 5V USB down to 3.3V. You can also connect a 4.2/3.7V Lithium Polymer (Lipo/Lipoly) or Lithium Ion (LiIon) battery to the JST jack. This will let the Feather run on a rechargable battery. **When the USB power is powered, it will automatically switch over to USB for power, as well as start charging the battery (if attached) at 100mA.** This happens 'hotswap' style so you can always keep the Lipoly connected as a 'backup' power that will only get used when USB power is lost.

The JST connector polarity is matched to Adafruit LiPoly batteries. Using wrong polarity batteries can destroy your Feather

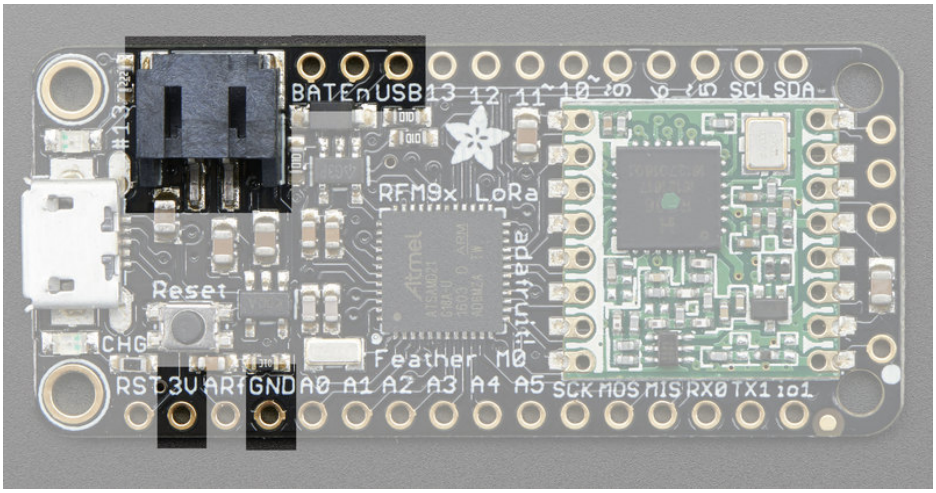


The above shows the Micro USB jack (left), Lipoly JST jack (top left), as well as the 3.3V regulator and changeover diode (just to the right of the JST jack) and the Lipoly charging circuitry (to the right of the Reset button). There's also a **CHG** LED, which will light up while the battery is charging. This LED might also flicker if the battery is not connected.

The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existant) battery. It's not harmful, and its totally normal!

Power supplies

You have a lot of power supply options here! We bring out the **BAT** pin, which is tied to the lipoly JST connector, as well as **USB** which is the +5V from USB if connected. We also have the **3V** pin which has the output from the 3.3V regulator. We use a 500mA peak regulator. While you can get 500mA from it, you can't do it continuously from 5V as it will overheat the regulator. It's fine for, say, powering an ESP8266 WiFi chip or XBee radio though, since the current draw is 'spikey' & sporadic.



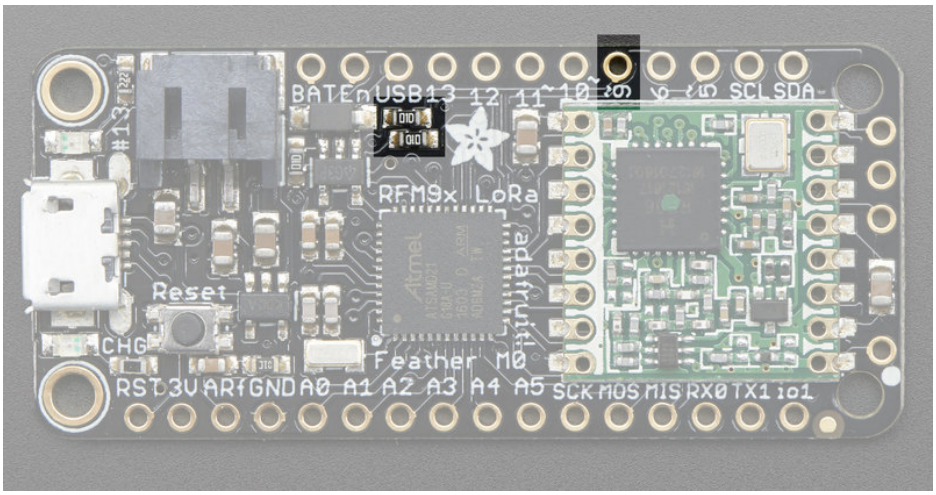
Measuring Battery

If you're running off of a battery, chances are you wanna know what the voltage is at! That way you can tell when the battery needs recharging. Lipoly batteries are 'maxed out' at 4.2V and stick around 3.7V for much of the battery life,

then slowly sink down to 3.2V or so before the protection circuitry cuts it off. By measuring the voltage you can quickly tell when you're heading below 3.7V

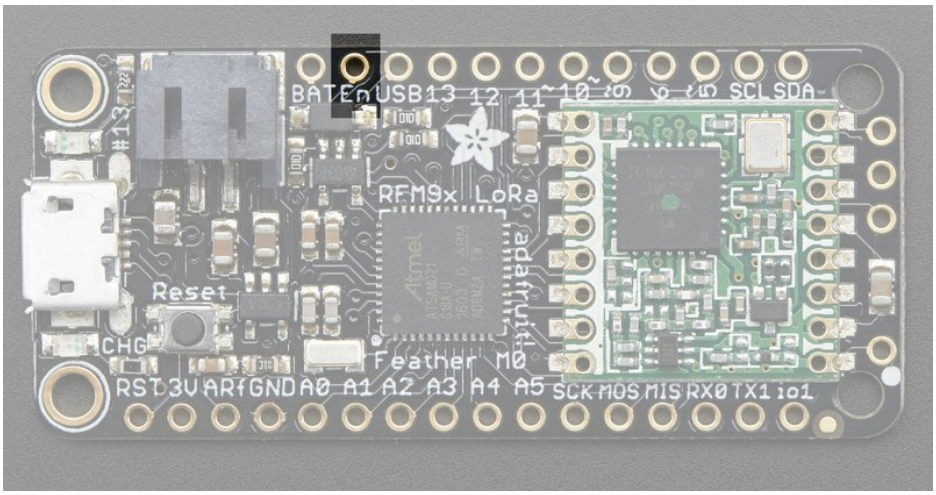
To make this easy we stuck a double-100K resistor divider on the **BAT** pin, and connected it to **D9** (a.k.a analog #7 **A7**). You can read this pin's voltage, then double it, to get the battery voltage.

```
#define VBATPIN A7

float measuredvbat = analogRead(VBATPIN);
measuredvbat *= 2;    // we divided by 2, so multiply back
measuredvbat *= 3.3;  // Multiply by 3.3V, our reference voltage
measuredvbat /= 1024; // convert to voltage
Serial.print("VBat: " ); Serial.println(measuredvbat);
```



ENable pin

If you'd like to turn off the 3.3V regulator, you can do that with the **EN**(able) pin. Simply tie this pin to **Ground** and it will disable the 3V regulator. The **BAT** and **USB** pins will still be powered



Alternative Power Options

The two primary ways for powering a feather are a 3.7/4.2V LiPo battery plugged into the JST port *or* a USB power

cable.

If you need other ways to power the Feather, here's what we recommend:

- For permanent installations, a 5V 1A USB wall adapter (https://adafru.it/duP) will let you plug in a USB cable for reliable power
- For mobile use, where you don't want a LiPoly, use a USB battery pack! (https://adafru.it/e2q)
- If you have a higher voltage power supply, use a 5V buck converter (https://adafru.it/DHs) and wire it to a USB cable's 5V and GND input (https://adafru.it/DHu)

Here's what you cannot do:

- **Do not use alkaline or NiMH batteries** and connect to the battery port - this will destroy the LiPoly charger and there's no way to disable the charger
- **Do not use 7.4V RC batteries on the battery port** - this will destroy the board

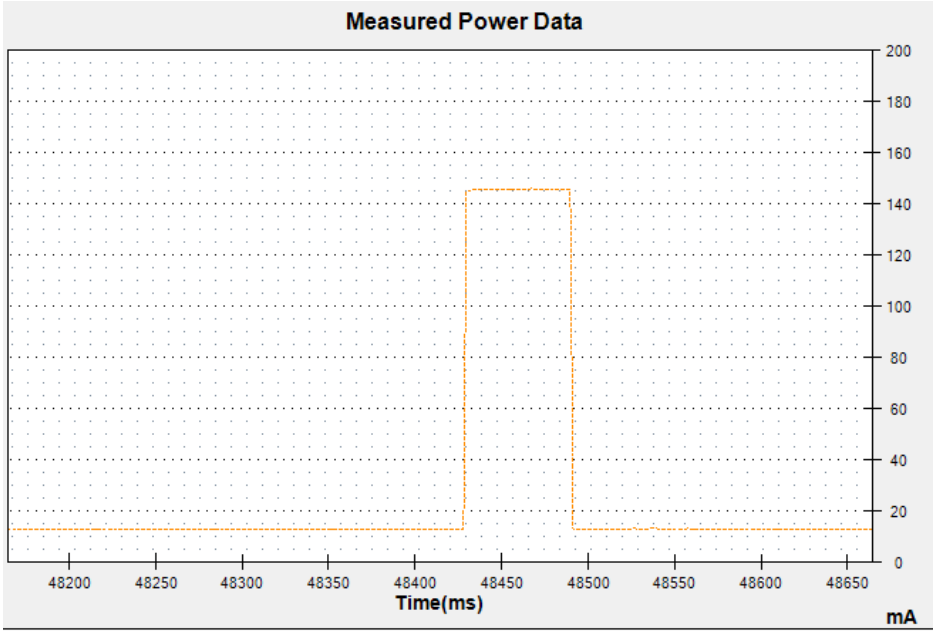
The Feather *is not designed for external power supplies* - this is a design decision to make the board compact and low cost. It is not recommended, but technically possible:

- **Connect an external 3.3V power supply to the 3V and GND pins.** Not recommended, this may cause unexpected behavior and the **EN** pin will no longer. Also this doesn't provide power on **BAT** or **USB** and some Feathers/Wings use those pins for high current usages. You may end up damaging your Feather.
- **Connect an external 5V power supply to the USB and GND pins.** Not recommended, this may cause unexpected behavior when plugging in the USB port because you will be back-powering the USB port, which *could* confuse or damage your computer.

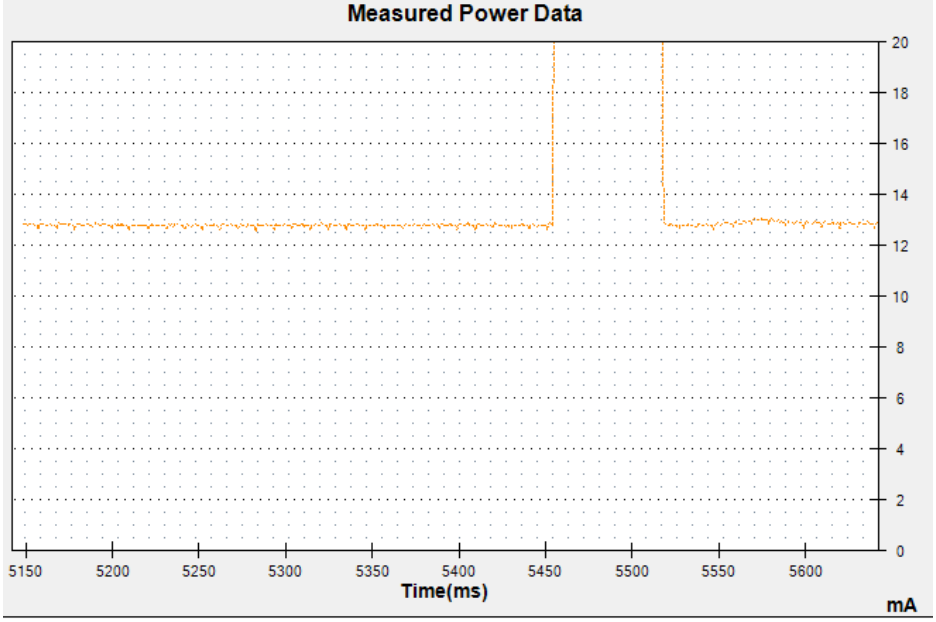
Radio Power Draw

You can select the power output you want via software, more power equals more range but of course, uses more of your battery.

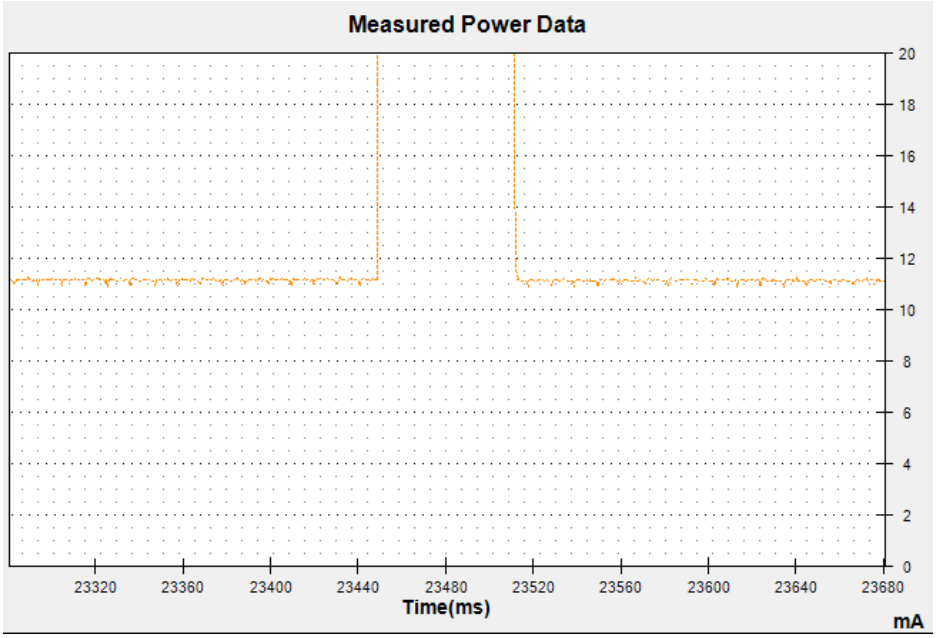
For example, here is the Feather with RFM9x 900MHz radio set up for +20dBm power, transmitting a data payload of 20 bytes. Transmits take about 130mA for 70ms



The ~13mA quiescent current is the current draw for listening (~2mA) plus ~11mA for the microcontroller. This can be reduce to amost nothing with proper sleep modes and not putting the module in active listen mode!



You can put the module into sleep mode by calling `radio.sleep();` which will save you about 2mA



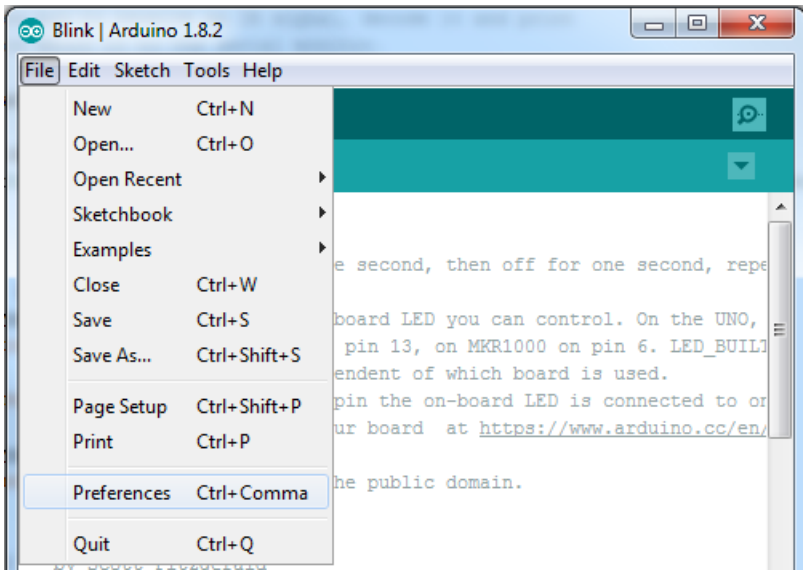
Arduino IDE Setup

The first thing you will need to do is to download the latest release of the Arduino IDE. You will need to be using **version 1.8** or higher for this guide

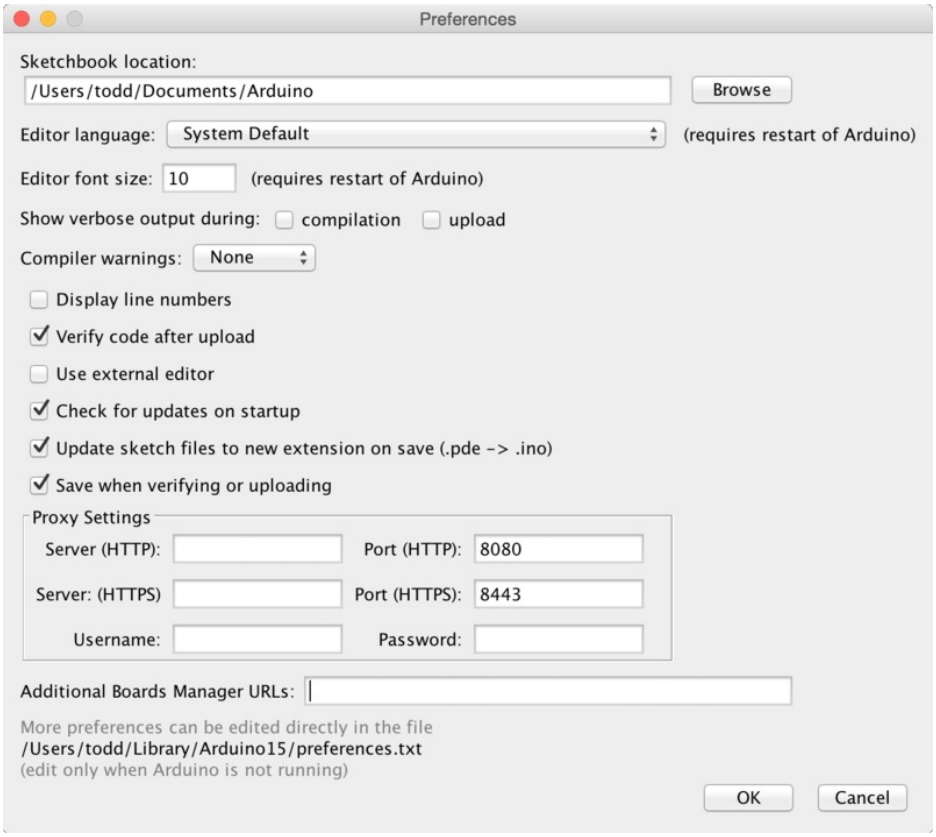
https://adafru.it/f1P

https://adafru.it/f1P

After you have downloaded and installed **the latest version of Arduino IDE**, you will need to start the IDE and navigate to the **Preferences** menu. You can access it from the **File** menu in *Windows* or *Linux*, or the **Arduino** menu on *OS X*.



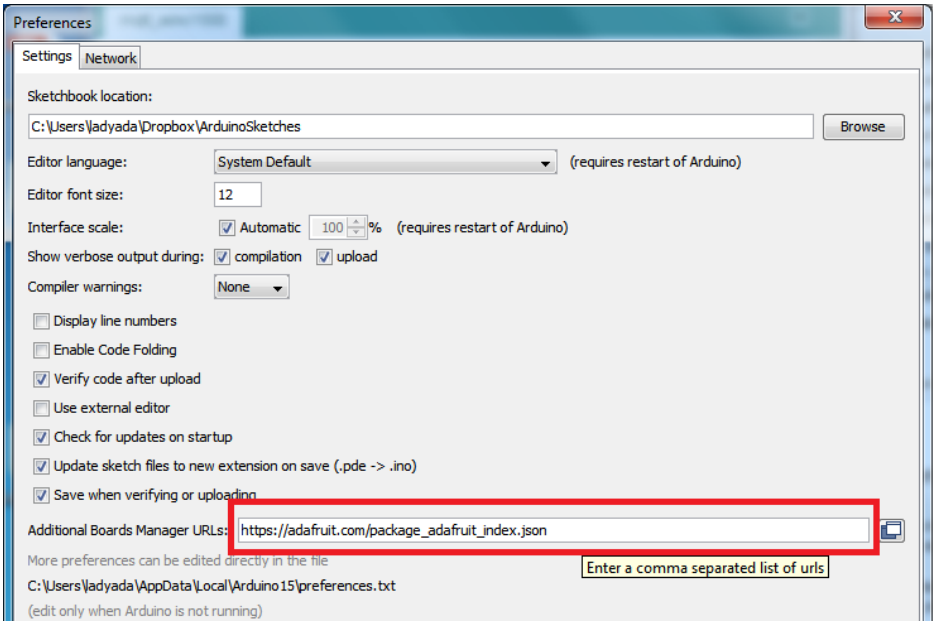
A dialog will pop up just like the one shown below.



We will be adding a URL to the new **Additional Boards Manager URLs** option. The list of URLs is comma separated, and *you will only have to add each URL once.* New Adafruit boards and updates to existing boards will automatically be picked up by the Board Manager each time it is opened. The URLs point to index files that the Board Manager uses to build the list of available & installed boards.

To find the most up to date list of URLs you can add, you can visit the list of third party board URLs on the Arduino IDE wiki (https://adafru.it/f7U). We will only need to add one URL to the IDE in this example, but ***you can add multiple URLS by separating them with commas***. Copy and paste the link below into the **Additional Boards Manager URLs** option in the Arduino IDE preferences.

```
https://adafruit.github.io/arduino-board-index/package_adafruit_index.json
```



Here's a short description of each of the Adafruit supplied packages that will be available in the Board Manager when you add the URL:

- **Adafruit AVR Boards** - Includes support for Flora, Gemma, Feather 32u4, Trinket, & Trinket Pro.
- **Adafruit SAMD Boards** - Includes support for Feather M0 and M4, Metro M0 and M4, ItsyBitsy M0 and M4, Circuit Playground Express, Gemma M0 and Trinket M0
- **Arduino Leonardo & Micro MIDI-USB** - This adds MIDI over USB support for the Flora, Feather 32u4, Micro and Leonardo using the arcore project (https://adafru.it/eSI).

If you have multiple boards you want to support, say ESP8266 and Adafruit, have both URLs in the text box separated by a comma (,)

Once done click **OK** to save the new preference settings. Next we will look at installing boards with the Board Manager.

Now continue to the next step to actually install the board support package!

Feather HELP!

Even though this FAQ is labeled for Feather, the questions apply to ItsyBitsy's as well!

My ItsyBitsy/Feather stopped working when I unplugged the USB!

A lot of our example sketches have a

```
while (!Serial);
```

line in setup(), to keep the board waiting until the USB is opened. This makes it a lot easier to debug a program because you get to see all the USB data output. If you want to run your Feather without USB connectivity, delete or comment out that line

My Feather never shows up as a COM or Serial port in the Arduino IDE

A vast number of Itsy/Feather 'failures' are due to charge-only USB cables

We get upwards of 5 complaints a day that turn out to be due to charge-only cables!

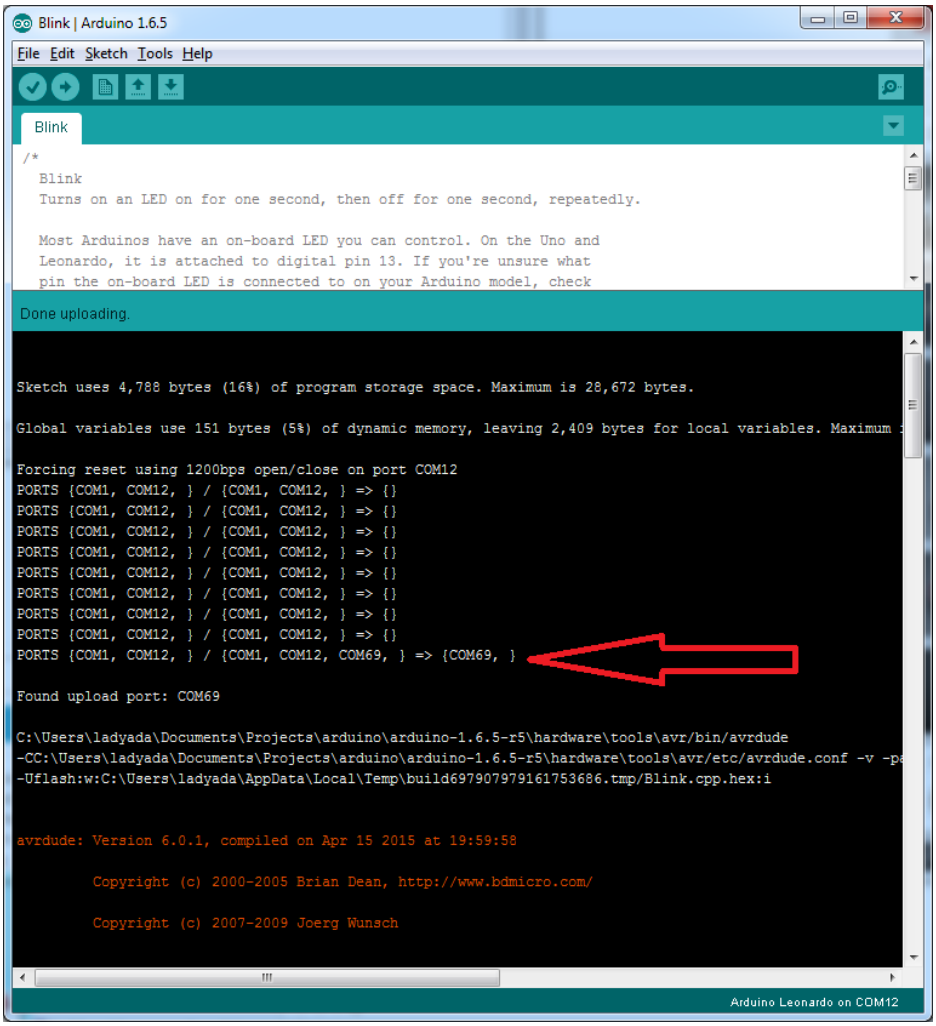
Use only a cable that you **know** is for data syncing

If you have any charge-only cables, cut them in half throw them out. We are serious! They tend to be low quality in general, and will only confuse you and others later, just get a good data+charge USB cable

Ack! I "did something" and now when I plug in the Itsy/Feather, it doesn't show up as a device anymore so I cant upload to it or fix it...

No problem! You can 'repair' a bad code upload easily. Note that this can happen if you set a watchdog timer or sleep mode that stops USB, or any sketch that 'crashes' your board

1. Turn on **verbose upload** in the Arduino IDE preferences
2. Plug in Itsy or Feather 32u4/M0, it won't show up as a COM/serial port that's ok
3. Open up the Blink example (Examples->Basics->Blink)
4. Select the correct board in the Tools menu, e.g. Feather 32u4, Feather M0, Itsy 32u4 or M0 *(physically check* your board to make sure you have the right one selected!)
5. Compile it (make sure that works)
6. Click Upload to attempt to upload the code
7. The IDE will print out a bunch of COM Ports as it tries to upload. **During this time, double-click the reset button, you'll see the red pulsing LED that tells you its now in bootloading mode**
8. The board will show up as the Bootloader COM/Serial port
9. The IDE should see the bootloader COM/Serial port and upload properly



I can't get the Itsy/Feather USB device to show up - I get "USB Device Malfunctioning" errors!

This seems to happen when people select the wrong board from the Arduino Boards menu.

If you have a Feather 32u4 (look on the board to read what it is you have) Make sure you select **Feather 32u4** for ATMega32u4 based boards! Do not use anything else, do not use the 32u4 breakout board line.

If you have a Feather M0 (look on the board to read what it is you have) Make sure you select **Feather M0** - do not

use 32u4 or Arduino Zero

If you have a ItsyBitsy M0 (look on the board to read what it is you have) Make sure you select **ItsyBitsy M0** - do not use 32u4 or Arduino Zero

I'm having problems with COM ports and my Itsy/Feather 32u4/M0

Theres **two** COM ports you can have with the 32u4/M0, one is the **user port** and one is the **bootloader port**. They are not the same COM port number!

When you upload a new user program it will come up with a user com port, particularly if you use Serial in your user

program.

If you crash your user program, or have a program that halts or otherwise fails, the user COM port can disappear.

When the user COM port disappears, Arduino will not be able to automatically start the bootloader and upload new software.

So you will need to help it by performing the click-during upload procedure to re-start the bootloader, and upload something that is known working like "Blink"

I don't understand why the COM port disappears, this does not happen on my Arduino UNO!

UNO-type Arduinos have a *seperate* serial port chip (aka "FTDI chip" or "Prolific PL2303" etc etc) which handles all serial port capability seperately than the main chip. This way if the main chip fails, you can always use the COM port.

M0 and 32u4-based Arduinos do not have a seperate chip, instead the main processor performs this task for you. It allows for a lower cost, higher power setup...but requires a little more effort since you will need to 'kick' into the bootloader manually once in a while

I'm trying to upload to my 32u4, getting "avrdude: butterfly_recv(): programmer is not responding" errors

This is likely because the bootloader is not kicking in and you are accidentally **trying to upload to the wrong COM port**

The best solution is what is detailed above: manually upload Blink or a similar working sketch by hand by manually launching the bootloader

I'm trying to upload to my Feather M0, and I get this error "Connecting to programmer: .avrdude: butterfly_recv(): programmer is not responding"

You probably don't have Feather M0 selected in the boards drop-down. Make sure you selected Feather M0.

- [ ] I'm trying to upload to my Feather and i get this error "avrdude: ser_recv(): programmer is not responding"

  You probably don't have Feather M0 / Feather 32u4 selected in the boards drop-down. Make sure you selected Feather M0 (or Feather 32u4).

I attached some wings to my Feather and now I can't read the battery voltage!

Make sure your Wing doesn't use pin #9 which is the analog sense for the lipo battery!

The yellow LED Is flickering on my Feather, but no battery is plugged in, why is that?

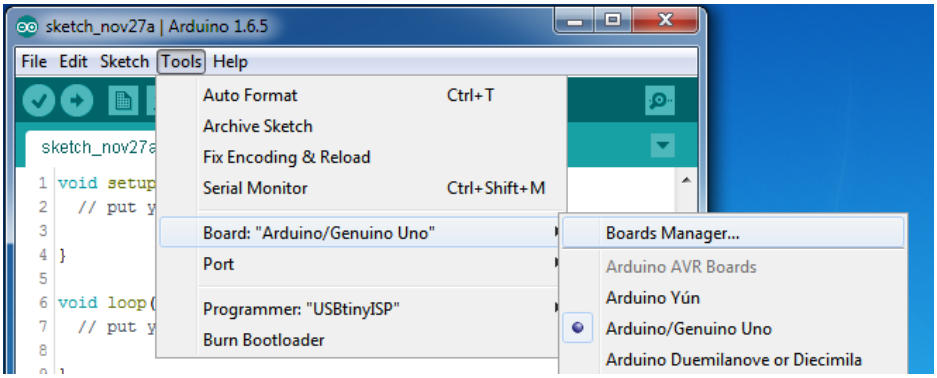
The charge LED is automatically driven by the Lipoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existant) battery.

It's not harmful, and its totally normal!

Using with Arduino IDE

The Feather/Metro/Gemma/Trinket M0 and M4 use an ATSAMD21 or ATSAMD51 chip, and you can pretty easily get it working with the Arduino IDE. Most libraries (including the popular ones like NeoPixels and display) will work with the M0 and M4, especially devices & sensors that use I2C or SPI.

Now that you have added the appropriate URLs to the Arduino IDE preferences in the previous page, you can open the **Boards Manager** by navigating to the **Tools->Board** menu.



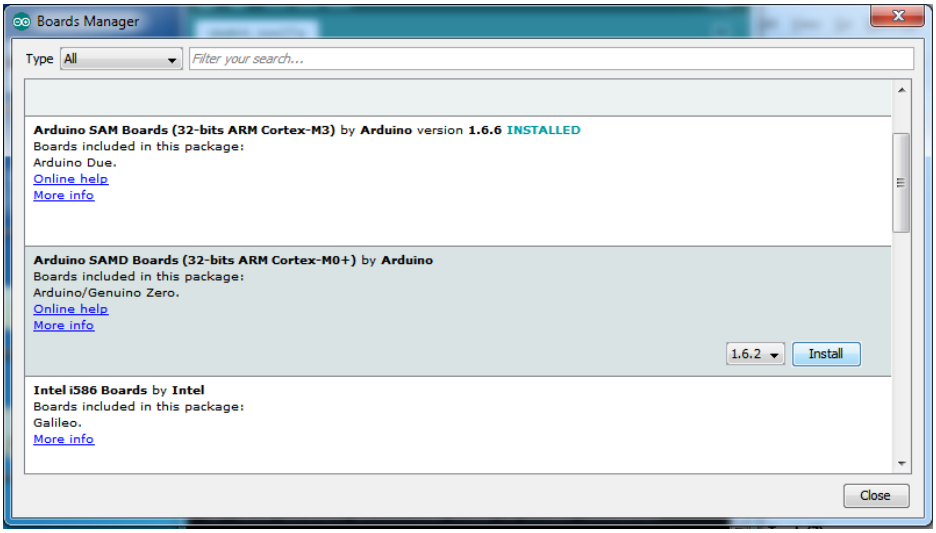
Once the Board Manager opens, click on the category drop down menu on the top left hand side of the window and select **All**. You will then be able to select and install the boards supplied by the URLs added to the preferences.

Remember you need SETUP the Arduino IDE to support our board packages - see the previous page on how to add adafruit's URL to the preferences

Install SAMD Support

First up, install the latest **Arduino SAMD Boards (**version **1.6.11** or later)

You can type **Arduino SAMD** in the top search bar, then when you see the entry, click **Install**

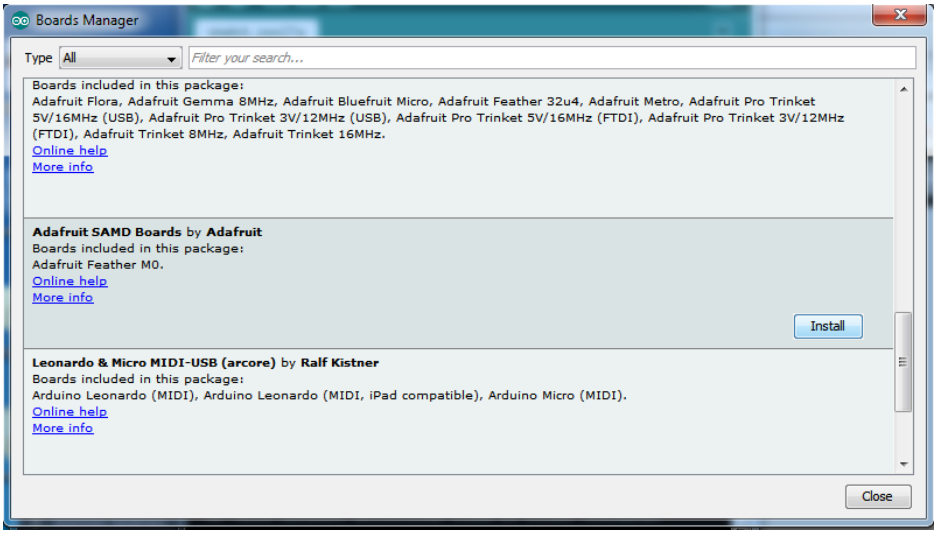


Install Adafruit SAMD

Next you can install the Adafruit SAMD package to add the board file definitions

Make sure you have **Type All** selected to the left of the *Filter your search...* box

You can type **Adafruit SAMD** in the top search bar, then when you see the entry, click **Install**

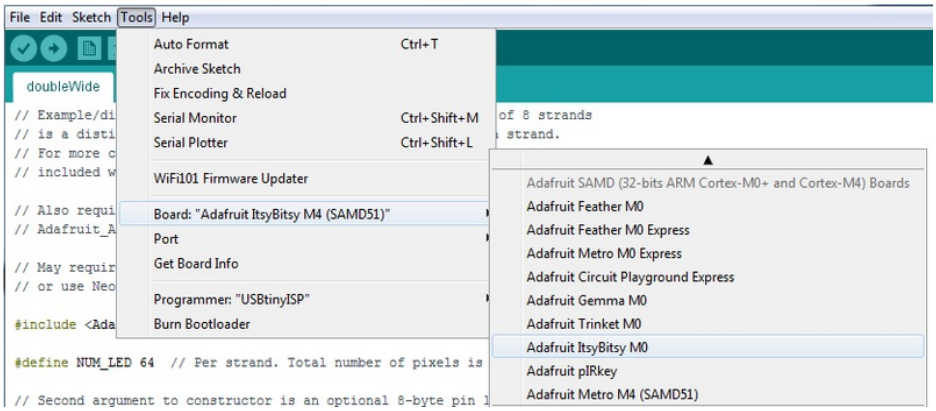


Even though in theory you don't need to - I recommend rebooting the IDE

Quit and reopen the Arduino IDE to ensure that all of the boards are properly installed. You should now be able to select and upload to the new boards listed in the **Tools->Board** menu.

Select the matching board, the current options are:

- **Feather M0** (for use with any Feather M0 other than the Express)
- **Feather M0 Express**
- **Metro M0 Express**
- **Circuit Playground Express**
- **Gemma M0**
- **Trinket M0**
- **ItsyBitsy M0**
- **Hallowing M0**
- **Crickit M0** (this is for direct programming of the Crickit, which is probably not what you want! For advanced hacking only)
- **Metro M4 Express**
- **ItsyBitsy M4 Express**
- **Feather M4 Express**
- **Trellis M4 Express**
- **Grand Central M4 Express**



Install Drivers (Windows 7 & 8 Only)

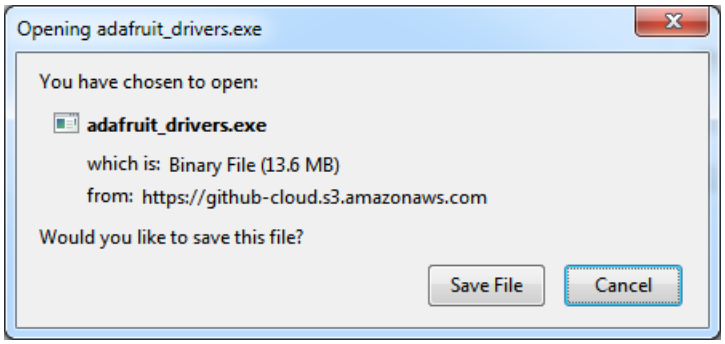
When you plug in the board, you'll need to possibly install a driver

Click below to download our Driver Installer

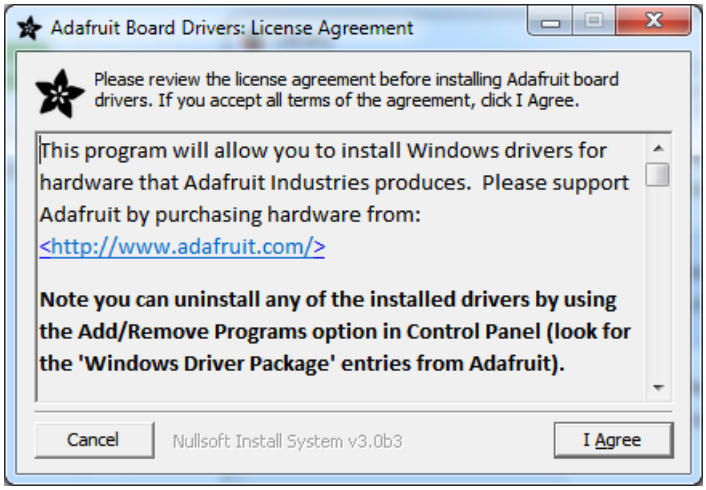
https://adafru.it/EC0

https://adafru.it/EC0

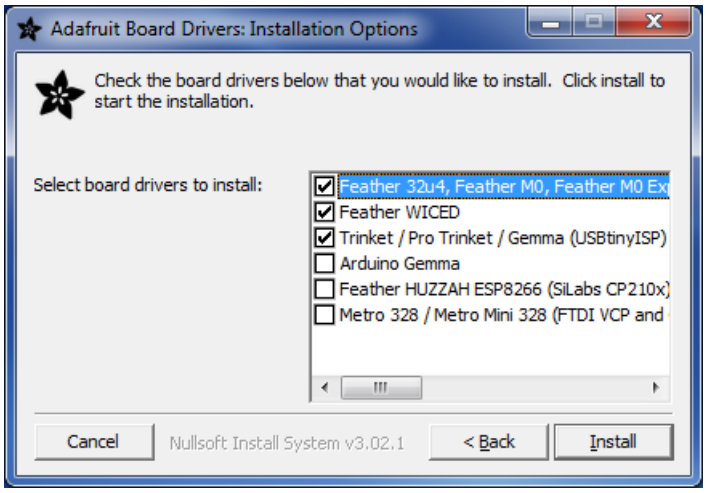
Download and run the installer



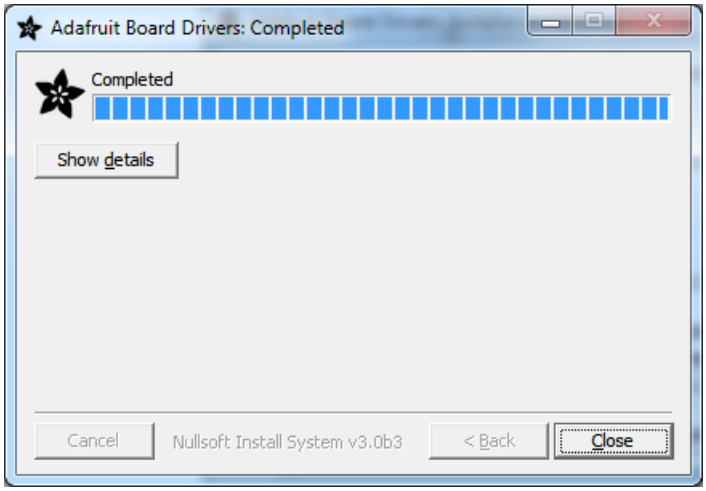
Run the installer! Since we bundle the SiLabs and FTDI drivers as well, you'll need to click through the license



Select which drivers you want to install, the defaults will set you up with just about every Adafruit board!



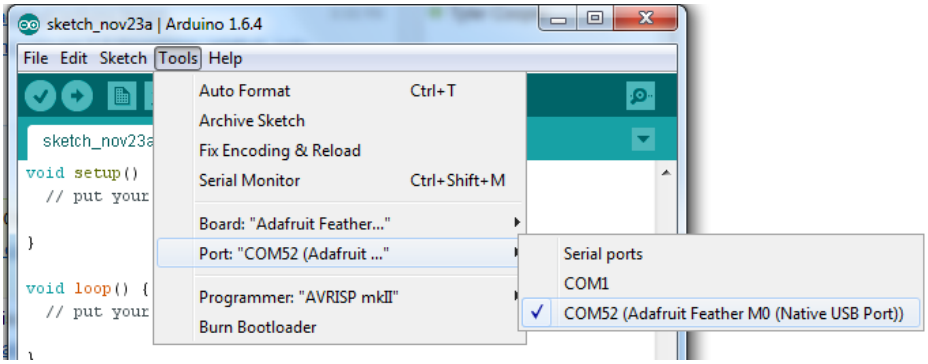
Click **Install** to do the installin'



Blink

Now you can upload your first blink sketch!

Plug in the M0 or M4 board, and wait for it to be recognized by the OS (just takes a few seconds). It will create a serial/COM port, you can now select it from the drop-down, it'll even be 'indicated' as Trinket/Gemma/Metro/Feather/ItsyBitsy/Trellis!



Now load up the Blink example

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

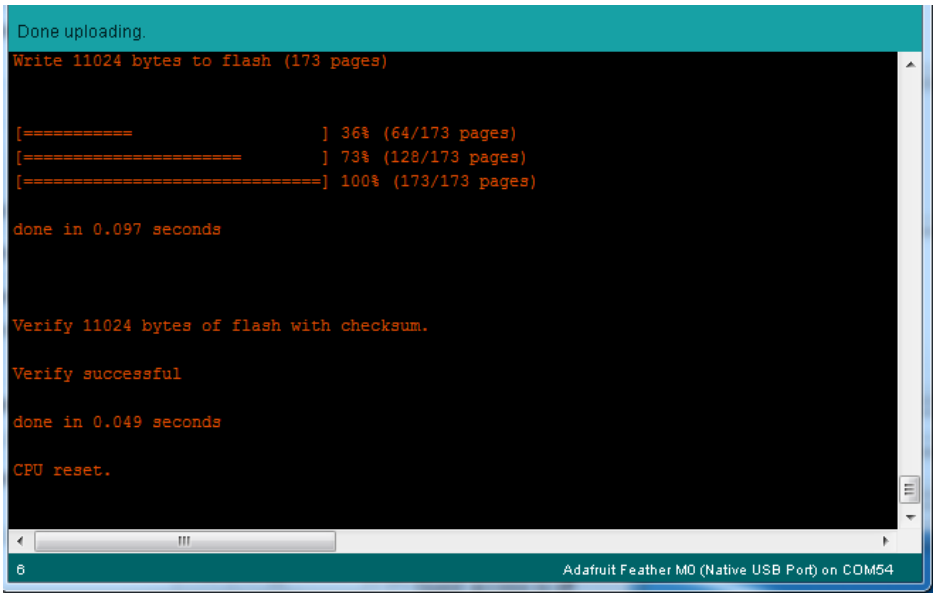
And click upload! That's it, you will be able to see the LED blink rate change as you adapt the **delay()** calls.

If you're using **Trellis M4 Express**, you can go to the next page cause there's no pin 13 LED - so you won't see it blink. Still this is a good thing to test compile and upload!

If you are having issues, make sure you selected the matching Board in the menu that matches the hardware you have in your hand.

Successful Upload

If you have a successful upload, you'll get a bunch of red text that tells you that the device was found and it was programmed, verified & reset



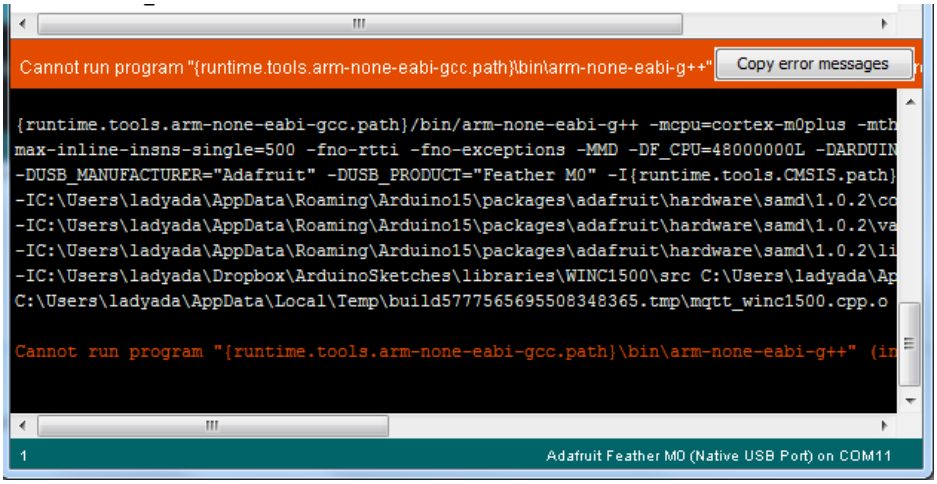
After uploading, you may see a message saying "Disk Not Ejected Properly" about the ...BOOT drive. You can ignore that message: it's an artifact of how the bootloader and uploading work.

Compilation Issues

If you get an alert that looks like

Cannot run program "{runtime.tools.arm-none-eabi-gcc.path}\bin\arm-non-eabi-g++"

Make sure you have installed the **Arduino SAMD** boards package, you need *both* Arduino & Adafruit SAMD board packages

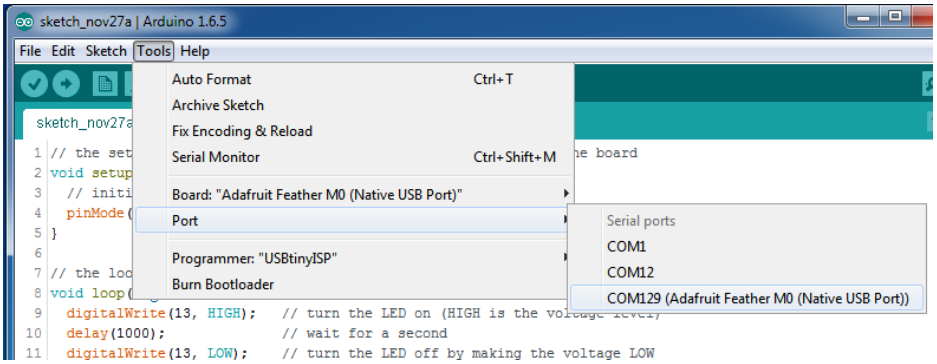


Manually bootloading

If you ever get in a 'weird' spot with the bootloader, or you have uploaded code that crashes and doesn't auto-reboot into the bootloader, click the **RST** button **twice** (like a double-click)to get back into the bootloader.

The red LED will pulse, so you know that its in bootloader mode.

Once it is in bootloader mode, you can select the newly created COM/Serial port and re-try uploading.



You may need to go back and reselect the 'normal' USB serial port next time you want to use the normal upload.

Ubuntu & Linux Issue Fix

Follow the steps for installing Adafruit's udev rules on this page. (https://adafru.it/iOE)

Adapting Sketches to M0 & M4

The ATSAMD21 and 51 are very nice little chips, but fairly new as Arduino-compatible cores go. **Most** sketches & libraries will work but here's a collection of things we noticed.

The notes below cover a range of Adafruit M0 and M4 boards, but not every rule will apply to every board (e.g. Trinket and Gemma M0 do not have ARef, so you can skip the Analog References note!).

Analog References

If you'd like to use the **ARef** pin for a non-3.3V analog reference, the code to use is `analogReference(AR_EXTERNAL)` (it's AR_EXTERNAL not EXTERNAL)

Pin Outputs & Pullups

The old-style way of turning on a pin as an input with a pullup is to use

```
pinMode(pin, INPUT)
digitalWrite(pin, HIGH)
```

This is because the pullup-selection register on 8-bit AVR chips is the same as the output-selection register.

For M0 & M4 boards, you can't do this anymore! Instead, use:

```
pinMode(pin, INPUT_PULLUP)
```

Code written this way still has the benefit of being *backwards compatible with AVR.* You don't need separate versions for the different board types.

Serial vs SerialUSB

99.9% of your existing Arduino sketches use **Serial.print** to debug and give output. For the Official Arduino SAMD/M0 core, this goes to the Serial5 port, which isn't exposed on the Feather. The USB port for the Official Arduino M0 core is called ***SerialUSB*** instead.

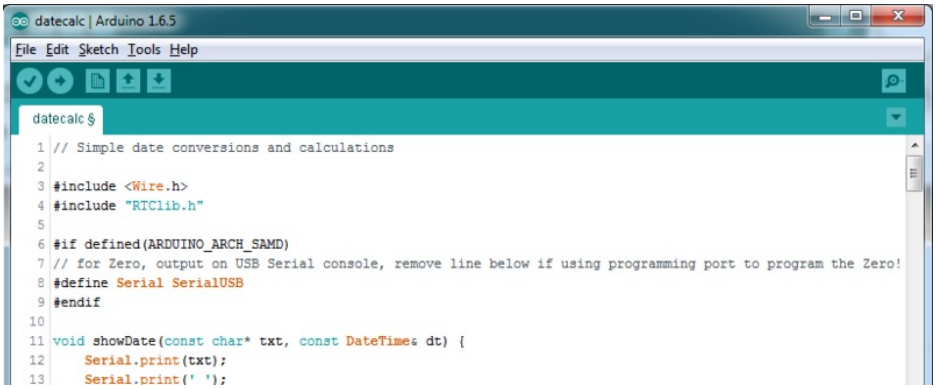
In the Adafruit M0/M4 Core, we fixed it so that **Serial goes to USB so it will automatically work just fine**.

However, on the off chance you are using the official Arduino SAMD core and *not* the Adafruit version (which really, we recommend you use our version because it's been tuned to our boards), and you want your Serial prints and reads to use the USB port, use *SerialUSB* instead of *Serial* in your sketch.

If you have existing sketches and code and you want them to work with the M0 without a huge find-replace, put

```
#if defined(ARDUINO_SAMD_ZERO) && defined(SERIAL_PORT_USBVIRTUAL)
 // Required for Serial on Zero based boards
 #define Serial SERIAL_PORT_USBVIRTUAL
#endif
```

right above the first function definition in your code. For example:



AnalogWrite / PWM on Feather/Metro M0

After looking through the SAMD21 datasheet, we've found that some of the options listed in the multiplexer table don't exist on the specific chip used in the Feather M0.

For all SAMD21 chips, there are two peripherals that can generate PWM signals: The Timer/Counter (TC) and Timer/Counter for Control Applications (TCC). Each SAMD21 has multiple copies of each, called 'instances'.

Each TC instance has one count register, one control register, and two output channels. Either channel can be enabled and disabled, and either channel can be inverted. The pins connected to a TC instance can output identical versions of the same PWM waveform, or complementary waveforms.

Each TCC instance has a single count register, but multiple compare registers and output channels. There are options for different kinds of waveform, interleaved switching, programmable dead time, and so on.

The biggest members of the SAMD21 family have five TC instances with two 'waveform output' (WO) channels, and three TCC instances with eight WO channels:

- TC[0-4],WO[0-1]
- TCC[0-2],WO[0-7]

And those are the ones shown in the datasheet's multiplexer tables.

The SAMD21G used in the Feather M0 only has three TC instances with two output channels, and three TCC instances with eight output channels:

- **TC[3-5],WO[0-1]**
- **TCC[0-2],WO[0-7]**

Tracing the signals to the pins broken out on the Feather M0, the following pins can't do PWM at all:

- **Analog pin A5**

The following pins can be configured for PWM without any signal conflicts as long as the SPI, I2C, and UART pins keep their protocol functions:

- **Digital pins 5, 6, 9, 10, 11, 12, and 13**
- **Analog pins A3 and A4**

If only the SPI pins keep their protocol functions, you can also do PWM on the following pins:

- **TX and SDA (Digital pins 1 and 20)**

analogWrite() PWM range

On AVR, if you set a pin's PWM with `analogWrite(pin, 255)` it will turn the pin fully HIGH. On the ARM cortex, it will set it to be 255/256 so there will be very slim but still-existing pulses-to-0V. If you need the pin to be fully on, add test code that checks if you are trying to `analogWrite(pin, 255)` and, instead, does a `digitalWrite(pin, HIGH)`

analogWrite() DAC on A0

If you are trying to use `analogWrite()` to control the DAC output on **A0**, make sure you do **not** have a line that sets the pin to output. ***Remove***: `pinMode(A0, OUTPUT)`.

Missing header files

There might be code that uses libraries that are not supported by the M0 core. For example if you have a line with

```
#include <util/delay.h>
```

you'll get an error that says

```
fatal error: util/delay.h: No such file or directory
 #include <util/delay.h>
                ^
compilation terminated.
Error compiling.
```

In which case you can simply locate where the line is (the error will give you the file name and line number) and 'wrap it' with #ifdef's so it looks like:

```
#if !defined(ARDUINO_ARCH_SAM) && !defined(ARDUINO_ARCH_SAMD) && !defined(ESP8266) &&
!defined(ARDUINO_ARCH_STM32F2)
 #include <util/delay.h>
#endif
```

The above will also make sure that header file isn't included for other architectures

If the #include is in the arduino sketch itself, you can try just removing the line.

Bootloader Launching

For most other AVRs, clicking **reset** while plugged into USB will launch the bootloader manually, the bootloader will time out after a few seconds. For the M0/M4, you'll need to ***double click*** the button. You will see a pulsing red LED to let you know you're in bootloader mode. Once in that mode, it wont time out! Click reset again if you want to go back to launching code.

Aligned Memory Access

This is a little less likely to happen to you but it happened to me! If you're used to 8-bit platforms, you can do this nice thing where you can typecast variables around. e.g.

```
uint8_t mybuffer[4];
```

`float f = (float)mybuffer;`

You can't be guaranteed that this will work on a 32-bit platform because **mybuffer** might not be aligned to a 2 or 4-byte boundary. The ARM Cortex-M0 can only directly access data on 16-bit boundaries (every 2 or 4 bytes). Trying to access an odd-boundary byte (on a 1 or 3 byte location) will cause a Hard Fault and stop the MCU. Thankfully, there's an easy work around ... just use memcpy!

```
uint8_t mybuffer[4];
float f;
memcpy(&f, mybuffer, 4)
```

Floating Point Conversion

Like the AVR Arduinos, the M0 library does not have full support for converting floating point numbers to ASCII strings. Functions like sprintf will not convert floating point. Fortunately, the standard AVR-LIBC library includes the dtostrf function which can handle the conversion for you.

Unfortunately, the M0 run-time library does not have dtostrf. You may see some references to using **#include <avr/dtostrf.h>** to get dtostrf in your code. And while it will compile, it does **not** work.

Instead, check out this thread to find a working dtostrf function you can include in your code:

http://forum.arduino.cc/index.php?topic=368720.0 (https://adafru.it/IFS)

How Much RAM Available?

The ATSAMD21G18 has 32K of RAM, but you still might need to track it for some reason. You can do so with this handy function:

```
extern "C" char *sbrk(int i);

int FreeRam () {
  char stack_dummy = 0;
  return &stack_dummy - sbrk(0);
}
```

Thx to http://forum.arduino.cc/index.php?topic=365830.msg2542879#msg2542879 (https://adafru.it/m6D) for the tip!

Storing data in FLASH

If you're used to AVR, you've probably used **PROGMEM** to let the compiler know you'd like to put a variable or string in flash memory to save on RAM. On the ARM, its a little easier, simply add **const** before the variable name:

const char str[] = "My very long string";

That string is now in FLASH. You can manipulate the string just like RAM data, the compiler will automatically read from FLASH so you dont need special progmem-knowledgeable functions.

You can verify where data is stored by printing out the address:
Serial.print("Address of str $"); Serial.println((int)&str, HEX);

If the address is $2000000 or larger, its in SRAM. If the address is between $0000 and $3FFFF Then it is in FLASH

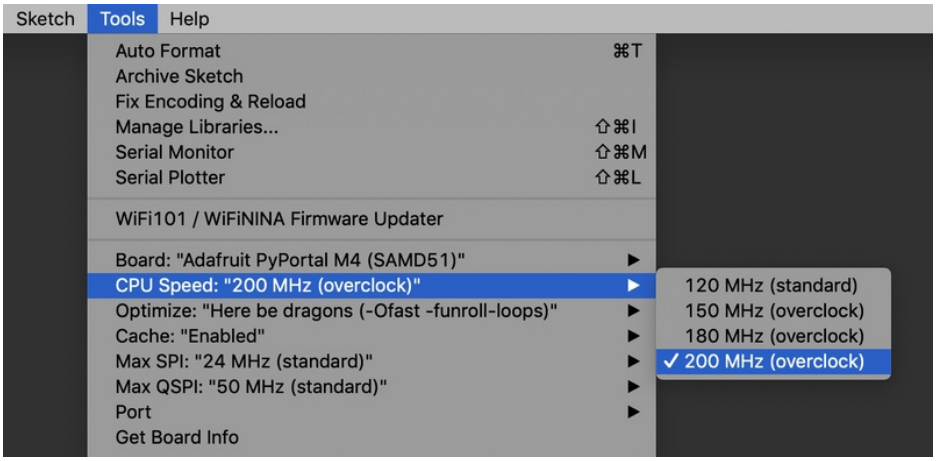
Pretty-Printing out registers

There's *a lot* of registers on the SAMD21, and you often are going through ASF or another framework to get to them. So having a way to see exactly what's going on is handy. This library from drewfish will help a ton!

https://github.com/drewfish/arduino-ZeroRegs (https://adafru.it/Bet)

M4 Performance Options

As of version 1.4.0 of the *Adafruit SAMD Boards* package in the Arduino Boards Manager, some options are available to wring extra performance out of M4-based devices. These are in the *Tools* menu.



All of these performance tweaks involve a degree of uncertainty. There's *no guarantee* of improved performance in any given project, and *some may even be detrimental,* failing to work in part or in whole. If you encounter trouble, **select the default performance settings** and re-upload.

Here's what you get and some issues you might encounter...

CPU Speed (overclocking)

This option lets you adjust the microcontroller core clock...the speed at which it processes instructions...beyond the official datasheet specifications.

Manufacturers often rate speeds conservatively because such devices are marketed for harsh industrial environments...if a system crashes, someone could lose a limb or worse. But most creative tasks are less critical and operate in more comfortable settings, and we can push things a bit if we want more speed.

There is a small but nonzero chance of code **locking up** or **failing to run** entirely. If this happens, try **dialing back the speed by one notch and re-upload**, see if it's more stable.

Much more likely, **some code or libraries may not play well** with the nonstandard CPU speed. For example, currently the NeoPixel library assumes a 120 MHz CPU speed and won't issue the correct data at other settings (this will be worked on). Other libraries may exhibit similar problems, usually anything that strictly depends on CPU timing...you might encounter problems with audio- or servo-related code depending how it's written. **If you encounter such code or libraries, set the CPU speed to the default 120 MHz and re-upload.**

Optimize

There's usually more than one way to solve a problem, some more resource-intensive than others. Since Arduino got its start on resource-limited AVR microcontrollers, the C++ compiler has always aimed for the **smallest compiled program size**. The "Optimize" menu gives some choices for the compiler to take different and often faster approaches, at the expense of slightly larger program size...with the huge flash memory capacity of M4 devices, that's rarely a problem now.

The "**Small**" setting will compile your code like it always has in the past, aiming for the smallest compiled program size.

The "**Fast**" setting invokes various speed optimizations. The resulting program should produce the same results, is slightly larger, and usually (but not always) noticably faster. It's worth a shot!

"**Here be dragons**" invokes some more intensive optimizations...code will be larger still, faster still, but there's a possibility these optimizations could cause unexpected behaviors. *Some code may not work the same as before.* Hence the name. Maybe you'll discover treasure here, or maybe you'll sail right off the edge of the world.

Most code and libraries will continue to function regardless of the optimizer settings. If you do encounter problems, **dial it back one notch and re-upload**.

Cache

This option allows a small collection of instructions and data to be accessed more quickly than from flash memory, boosting performance. It's enabled by default and should work fine with all code and libraries. But if you encounter some esoteric situation, the cache can be disabled, then recompile and upload.

Max SPI and Max QSPI

These should probably be left at their defaults. They're present mostly for our own experiments and can cause **serious headaches**.

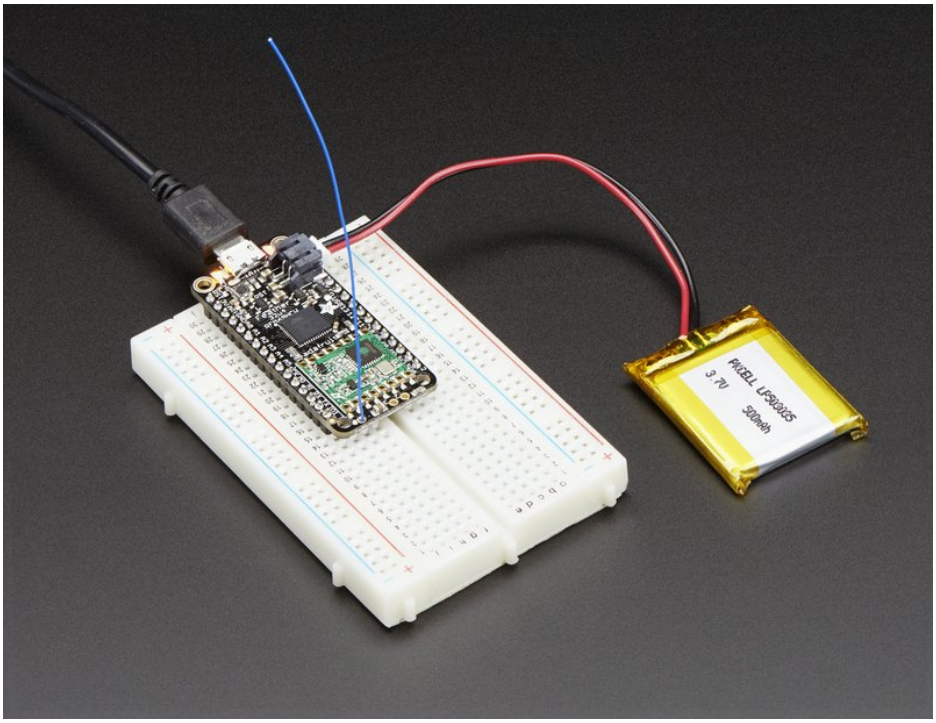
Max SPI determines the clock source for the M4's SPI peripherals. Under normal circumstances this allows transfers up to 24 MHz, and should usually be left at that setting. But...if you're using write-only SPI devices (such as TFT or OLED displays), this option lets you drive them faster (we've successfully used 60 MHz with some TFT screens). The caveat is, if using *any* read/write devices (such as an SD card), *this will not work at all...*SPI reads *absolutely* max out at the default 24 MHz setting, and anything else will fail. **Write = OK. Read = FAIL.** This is true *even if your code is using a lower bitrate setting...*just having the different clock source prevents SPI reads.

Max QSPI does similarly for the extra flash storage on M4 "Express" boards. *Very few* Arduino sketches access this storage at all, let alone in a bandwidth-constrained context, so this will benefit next to nobody. Additionally, due to the way clock dividers are selected, this will only provide some benefit when certain "CPU Speed" settings are active. Our PyPortal Animated GIF Display (https://adafru.it/EkO) runs marginally better with it, if using the QSPI flash.

Enabling the Buck Converter on some M4 Boards

If you want to reduce power draw, some of our boards have an inductor so you can use the 1.8V buck converter instead of the built in linear regulator. If the board does have an inductor (see the schematic) you can add the line `SUPC->VREG.bit.SEL = 1;` to your code to switch to it. Note it will make ADC/DAC reads a bit noisier so we don't use it by default. You'll save ~4mA (https://adafru.it/F0H).

Using the RFM9X Radio



Before beginning make sure you have your Feather working smoothly, it will make this part a lot easier. Once you have the basic Feather functionality going - you can upload code, blink an LED, use the serial output, etc. you can then upgrade to using the radio itself.

Note that the sub-GHz radio is not designed for streaming audio or video! It's best used for small packets of data. The data rate is adjustbale but its common to stick to around 19.2 Kbps (thats bits per second). Lower data rates will be more successful in their transmissions

You will, of course, need at least two paired radios to do any testing! The radios must be matched in frequency (e.g. 900 MHz & 900 MHz are ok, 900 MHz & 433 MHz are not). They also must use the same encoding schemes, you cannot have a 900 MHz RFM69 packet radio talk to a 900 MHz RFM96 LoRa radio.

Arduino Library

These radios have really excellent code already written, so rather than coming up with a new standard we suggest using existing libraries such as AirSpayce's Radiohead library (https://adafru.it/mCA) which also supports a vast number of other radios

This is a really great Arduino Library, so please support them in thanks for their efforts!

RadioHead RFM9x Library example

To begin talking to the radio, you will need to download the RadioHead library (https://adafru.it/mCA). You can do that by visiting the github repo and manually downloading or, easier, just click this button to download the zip corresponding to version 1.62

Note that while all the code in the examples below are based on this version you can visit the RadioHead

documentation page to get the most recent version which may have bug-fixes or more functionality (https://adafru.it/mCA)

https://adafru.it/q6f

https://adafru.it/q6f

Uncompress the zip and find the folder named **RadioHead** and check that the **RadioHead** folder contains **RH_RF95.cpp** and **RH_RF95.h** (as well as a few dozen other files for radios that are supported)

Place the **RadioHead** library folder your **arduinosketchfolder/libraries/** folder.
You may need to create the **libraries** subfolder if its your first library. Restart the IDE.

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)

Basic RX & TX example

Lets get a basic demo going, where one Feather transmits and the other receives. We'll start by setting up the transmitter

Transmitter example code

This code will send a small packet of data once a second to node address #1

Load this code into your Transmitter Arduino/Feather!

Before uploading, check for the #define RF95_FREQ 915.0 line and change that to 433.0 if you are using the 433MHz version of the LoRa radio!

Uncomment/comment the sections defining the pins for Feather 32u4, Feather M0, etc depending on which chipset and wiring you are using! The pins used will vary depending on your setup!

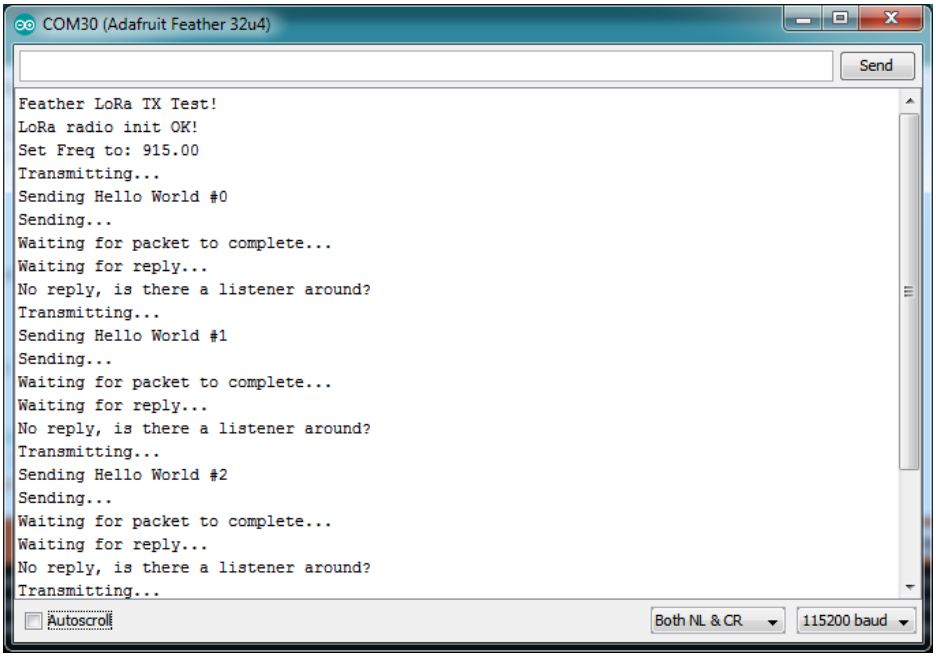
```
// Feather9x_TX
// -*- mode: C++ -*-
// Example sketch showing how to create a simple messaging client (transmitter)
// with the RH_RF95 class. RH_RF95 class does not provide for addressing or
// reliability, so you should only use RH_RF95 if you do not need the higher
// level messaging abilities.
// It is designed to work with the other example Feather9x_RX

#include <SPI.h>
#include <RH_RF95.h>

/* for feather32u4
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 7
*/

/* for feather m0
```

```
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 3
*/

/* for shield
#define RFM95_CS 10
#define RFM95_RST 9
#define RFM95_INT 7
*/

/* Feather 32u4 w/wing
#define RFM95_RST     11   // "A"
#define RFM95_CS      10   // "B"
#define RFM95_INT     2    // "SDA" (only SDA/SCL/RX/TX have IRQ!)
*/

/* Feather m0 w/wing
#define RFM95_RST     11   // "A"
#define RFM95_CS      10   // "B"
#define RFM95_INT     6    // "D"
*/

#if defined(ESP8266)
  /* for ESP w/featherwing */
  #define RFM95_CS  2    // "E"
  #define RFM95_RST 16   // "D"
  #define RFM95_INT 15   // "B"

#elif defined(ESP32)
  /* ESP32 feather w/wing */
  #define RFM95_RST     27   // "A"
  #define RFM95_CS      33   // "B"
  #define RFM95_INT     12   //  next to A

#elif defined(NRF52)
  /* nRF52832 feather w/wing */
  #define RFM95_RST     7   // "A"
  #define RFM95_CS      11   // "B"
  #define RFM95_INT     31   // "C"

#elif defined(TEENSYDUINO)
  /* Teensy 3.x w/wing */
  #define RFM95_RST     9   // "A"
  #define RFM95_CS      10   // "B"
  #define RFM95_INT     4    // "C"
#endif

// Change to 434.0 or other frequency, must match RX's freq!
#define RF95_FREQ 915.0

// Singleton instance of the radio driver
RH_RF95 rf95(RFM95_CS, RFM95_INT);

void setup()
{
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  Serial.begin(115200);
```

```
  while (!Serial) {
    delay(1);
  }

  delay(100);

  Serial.println("Feather LoRa TX Test!");

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  while (!rf95.init()) {
    Serial.println("LoRa radio init failed");
    Serial.println("Uncomment '#define SERIAL_DEBUG' in RH_RF95.cpp for detailed debug info");
    while (1);
  }
  Serial.println("LoRa radio init OK!");

  // Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250, +13dbM
  if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
    while (1);
  }
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);

  // Defaults after init are 434.0MHz, 13dBm, Bw = 125 kHz, Cr = 4/5, Sf = 128chips/symbol, CRC on

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  rf95.setTxPower(23, false);
}

int16_t packetnum = 0;  // packet counter, we increment per xmission

void loop()
{
  delay(1000); // Wait 1 second between transmits, could also 'sleep' here!
  Serial.println("Transmitting..."); // Send a message to rf95_server

  char radiopacket[20] = "Hello World #      ";
  itoa(packetnum++, radiopacket+13, 10);
  Serial.print("Sending "); Serial.println(radiopacket);
  radiopacket[19] = 0;

  Serial.println("Sending...");
  delay(10);
  rf95.send((uint8_t *)radiopacket, 20);

  Serial.println("Waiting for packet to complete...");
  delay(10);
  rf95.waitPacketSent();
  // Now wait for a reply
  uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);
```

```
  Serial.println("Waiting for reply...");
  if (rf95.waitAvailableTimeout(1000))
  {
    // Should be a reply message for us now
    if (rf95.recv(buf, &len))
   {
      Serial.print("Got reply: ");
      Serial.println((char*)buf);
      Serial.print("RSSI: ");
      Serial.println(rf95.lastRssi(), DEC);
    }
    else
    {
      Serial.println("Receive failed");
    }
  }
  else
  {
    Serial.println("No reply, is there a listener around?");
  }

}
```

Once uploaded you should see the following on the serial console



Now open up another instance of the Arduino IDE - this is so you can see the serial console output from the TX Feather while you set up the RX Feather.

Receiver example code

This code will receive and acknowledge a small packet of data.

Load this code into your **Receiver** Arduino/Feather!

Make sure the #define RF95_FREQ 915.0 matches your transmitter Feather!

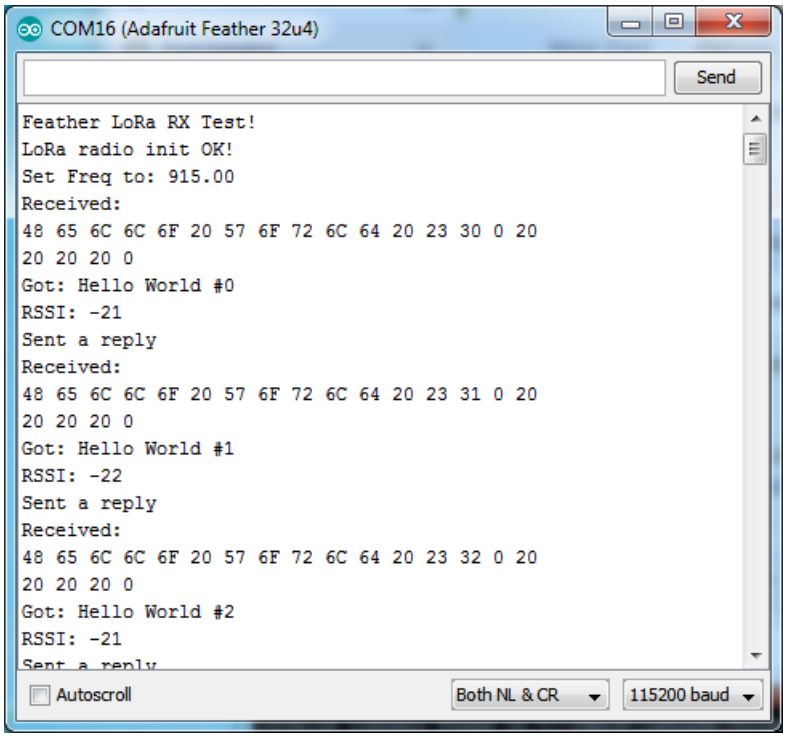
Uncomment/comment the sections defining the pins for Feather 32u4, Feather M0, etc depending on which chipset and wiring you are using! The pins used will vary depending on your setup!

```
// Feather9x_RX
// -*- mode: C++ -*-
// Example sketch showing how to create a simple messaging client (receiver)
// with the RH_RF95 class. RH_RF95 class does not provide for addressing or
// reliability, so you should only use RH_RF95 if you do not need the higher
// level messaging abilities.
// It is designed to work with the other example Feather9x_TX

#include <SPI.h>
#include <RH_RF95.h>

/* for Feather32u4 RFM9x
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 7
*/

/* for feather m0 RFM9x
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 3
*/

/* for shield
#define RFM95_CS 10
#define RFM95_RST 9
#define RFM95_INT 7
*/

/* Feather 32u4 w/wing
#define RFM95_RST     11   // "A"
#define RFM95_CS      10   // "B"
#define RFM95_INT     2    // "SDA" (only SDA/SCL/RX/TX have IRQ!)
*/

/* Feather m0 w/wing
#define RFM95_RST     11   // "A"
#define RFM95_CS      10   // "B"
#define RFM95_INT     6    // "D"
*/

#if defined(ESP8266)
  /* for ESP w/featherwing */
  #define RFM95_CS  2    // "E"
  #define RFM95_RST 16   // "D"
  #define RFM95_INT 15   // "B"

#elif defined(ESP32)
  /* ESP32 feather w/wing */
  #define RFM95_RST     27   // "A"
```

```
  #define RFM95_CS      33   // "B"
  #define RFM95_INT     12   //  next to A

#elif defined(NRF52)
  /* nRF52832 feather w/wing */
  #define RFM95_RST     7   // "A"
  #define RFM95_CS      11   // "B"
  #define RFM95_INT     31   // "C"

#elif defined(TEENSYDUINO)
  /* Teensy 3.x w/wing */
  #define RFM95_RST     9   // "A"
  #define RFM95_CS      10   // "B"
  #define RFM95_INT     4    // "C"
#endif


// Change to 434.0 or other frequency, must match RX's freq!
#define RF95_FREQ 915.0

// Singleton instance of the radio driver
RH_RF95 rf95(RFM95_CS, RFM95_INT);

// Blinky on receipt
#define LED 13

void setup()
{
  pinMode(LED, OUTPUT);
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  Serial.begin(115200);
  while (!Serial) {
    delay(1);
  }
  delay(100);

  Serial.println("Feather LoRa RX Test!");

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);

  while (!rf95.init()) {
    Serial.println("LoRa radio init failed");
    Serial.println("Uncomment '#define SERIAL_DEBUG' in RH_RF95.cpp for detailed debug info");
    while (1);
  }
  Serial.println("LoRa radio init OK!");

  // Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250, +13dbM
  if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
    while (1);
  }
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);
```

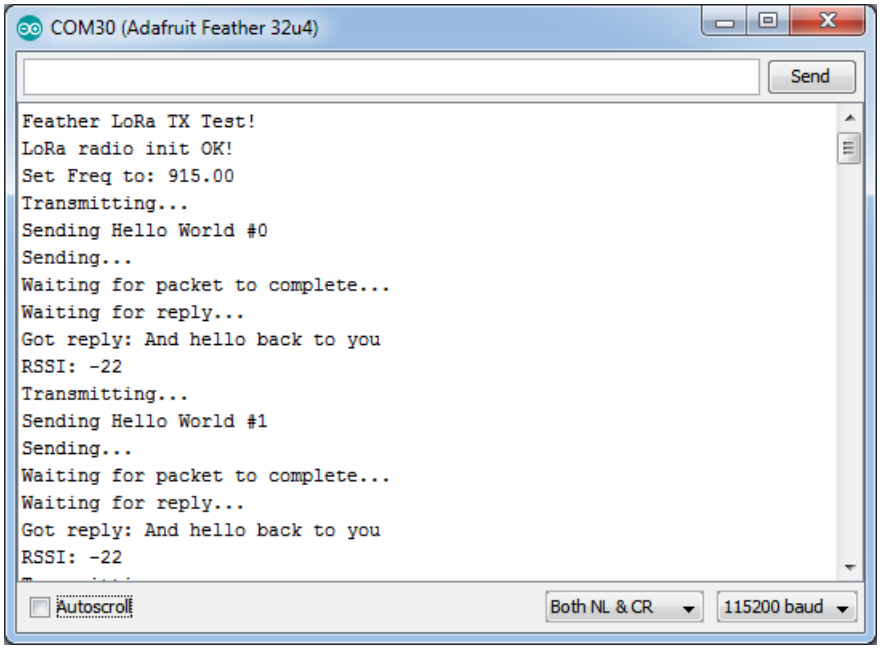
```
  // Defaults after init are 434.0MHz, 13dBm, Bw = 125 kHz, Cr = 4/5, Sf = 128chips/symbol, CRC on

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  rf95.setTxPower(23, false);
}

void loop()
{
  if (rf95.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);

    if (rf95.recv(buf, &len))
    {
      digitalWrite(LED, HIGH);
      RH_RF95::printBuffer("Received: ", buf, len);
      Serial.print("Got: ");
      Serial.println((char*)buf);
       Serial.print("RSSI: ");
      Serial.println(rf95.lastRssi(), DEC);

      // Send a reply
      uint8_t data[] = "And hello back to you";
      rf95.send(data, sizeof(data));
      rf95.waitPacketSent();
      Serial.println("Sent a reply");
      digitalWrite(LED, LOW);
    }
    else
    {
      Serial.println("Receive failed");
    }
  }
}
```

Now open up the Serial console on the receiver, while also checking in on the transmitter's serial console. You should see the receiver is...well, receiving packets



You can see that the library example prints out the hex-bytes received `48 65 6C 6C 6F 20 57 6F 72 6C 64 20 23 30 0 20 20 20 20 0`, as well as the ASCII 'string' `Hello World`. Then it will send a reply.

And, on the transmitter side, it is now printing that it got a reply after each transmisssion `And hello back to you` because it got a reply from the receiver



That's pretty much the basics of it! Lets take a look at the examples so you know how to adapt to your own radio setup

Feather Radio Pinout

This is the pinout setup for all **Feather 32u4** RFM9X's:

```
/* for feather32u4 */
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 7
```

This is the pinout for all **Feather M0** RFM9X's:

```
/* for feather m0 */
#define RFM95_CS 8
#define RFM95_RST 4
#define RFM95_INT 3
```

Frequency

You can dial in the frequency you want the radio to communicate on, such as 915.0, 434.0 or 868.0 or any number really. Different countries/ITU Zones have different ISM bands so make sure you're using those or if you are licensed, those frequencies you may use

```
// Change to 434.0 or other frequency, must match RX's freq!
#define RF95_FREQ 915.0
```

You can then instantiate the radio object with our custom pin numbers.

```
// Singleton instance of the radio driver
RH_RF95 rf95(RFM95_CS, RFM95_INT);
```

Setup

We begin by setting up the serial console and hard-resetting the Radio

```
void setup()
{
  pinMode(LED, OUTPUT);
  pinMode(RFM95_RST, OUTPUT);
  digitalWrite(RFM95_RST, HIGH);

  while (!Serial); // wait until serial console is open, remove if not tethered to computer
  Serial.begin(9600);
  delay(100);
  Serial.println("Feather LoRa RX Test!");

  // manual reset
  digitalWrite(RFM95_RST, LOW);
  delay(10);
  digitalWrite(RFM95_RST, HIGH);
  delay(10);
```

Remove the **while (!Serial);** line if you are not tethering to a computer, as it will cause the Feather to halt until a USB

connection is made!

Initializing Radio

The library gets initialized with a call to **init()**. Once initialized, you can set the frequency. You can also configure the output power level, the number ranges from 5 to 23. Start with the highest power level (23) and then scale down as necessary

```
  while (!rf95.init()) {
    Serial.println("LoRa radio init failed");
    while (1);
  }
  Serial.println("LoRa radio init OK!");

  // Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250, +13dbM
  if (!rf95.setFrequency(RF95_FREQ)) {
    Serial.println("setFrequency failed");
    while (1);
  }
  Serial.print("Set Freq to: "); Serial.println(RF95_FREQ);

  // Defaults after init are 434.0MHz, 13dBm, Bw = 125 kHz, Cr = 4/5, Sf = 128chips/symbol, CRC on

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  rf95.setTxPower(23, false);
```

Transmission Code

If you are using the transmitter, this code will wait 1 second, then transmit a packet with "Hello World #" and an incrementing packet number

```
void loop()
{
  delay(1000); // Wait 1 second between transmits, could also 'sleep' here!
  Serial.println("Transmitting..."); // Send a message to rf95_server

  char radiopacket[20] = "Hello World #      ";
  itoa(packetnum++, radiopacket+13, 10);
  Serial.print("Sending "); Serial.println(radiopacket);
  radiopacket[19] = 0;

  Serial.println("Sending..."); delay(10);
  rf95.send((uint8_t *)radiopacket, 20);

  Serial.println("Waiting for packet to complete..."); delay(10);
  rf95.waitPacketSent();
```

Its pretty simple, the delay does the waiting, you can replace that with low power sleep code. Then it generates the packet and appends a number that increases every tx. Then it simply calls **send** to transmit the data, and passes in the array of data and the length of the data.

Note that this does not any addressing or subnetworking - if you want to make sure the packet goes to a particular

radio, you may have to add an identifier/address byte on your own!

Then you call **waitPacketSent()** to wait until the radio is done transmitting. You will not get an automatic acknowledgement, from the other radio unless it knows to send back a packet. Think of it like the 'UDP' of radio - the data is sent, but its not certain it was received! Also, there will not be any automatic retries.

Receiver Code

The Receiver has the same exact setup code, but the loop is different

```
void loop()
{
  if (rf95.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);

    if (rf95.recv(buf, &len))
    {
      digitalWrite(LED, HIGH);
      RH_RF95::printBuffer("Received: ", buf, len);
      Serial.print("Got: ");
      Serial.println((char*)buf);
       Serial.print("RSSI: ");
      Serial.println(rf95.lastRssi(), DEC);
```

Instead of transmitting, it is constantly checking if there's any data packets that have been received.**available()** will return true if a packet with proper error-correction was received. If so, the receiver prints it out in hex and also as a 'character string'

It also prints out the RSSI which is the receiver signal strength indicator. This number will range from about -15 to about -100. The larger the number (-15 being the highest you'll likely see) the stronger the signal.

Once done it will automatically reply, which is a way for the radios to know that there was an acknowledgement

```
// Send a reply
      uint8_t data[] = "And hello back to you";
      delay(200);
      rf95.send(data, sizeof(data));
      rf95.waitPacketSent();
      Serial.println("Sent a reply");
```

It simply sends back a string and waits till the reply is completely sent

CircuitPython for RFM9x LoRa

It's easy to use the RFM9x LoRa radio with CircuitPython and the Adafruit CircuitPython RFM9x (https://adafru.it/BjD) module. This module allows you to easily write Python code that sends and receives packets of data with the radio. Be careful to note this library is for the RFM95/96/97/98 LoRa radio only and **will not** work with the simper RFM69 packet radio.

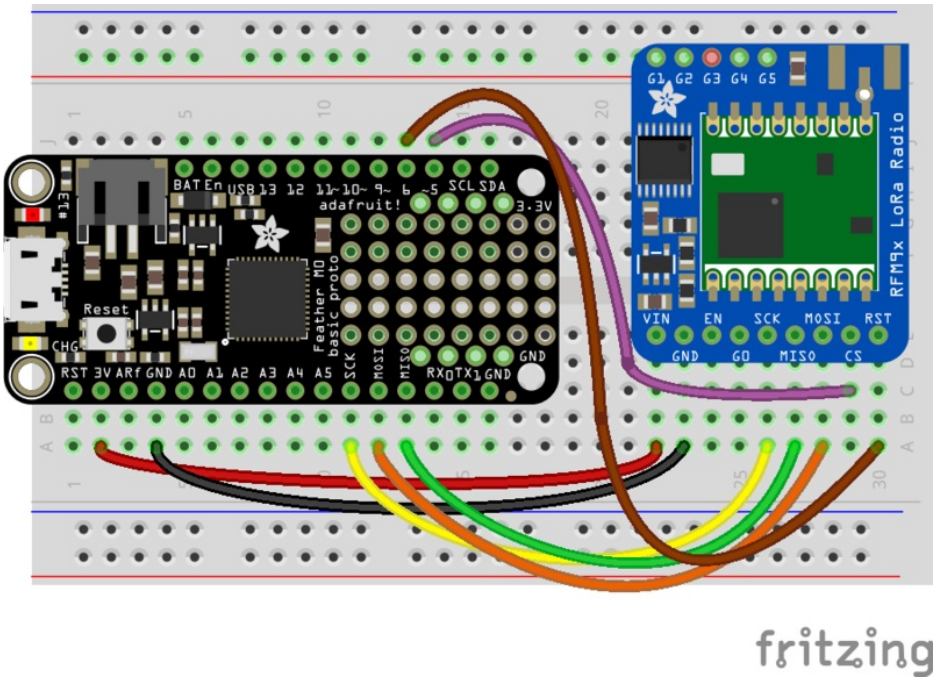
Design Considerations

One thing to be aware of before you use the RFM9x series of radios with CircuitPython are some of the limitations and design considerations for its module. Keep these in mind as you think about projects using the RFM9x and CircuitPython:

- You can only send and receive packets up to 252 bytes in length at a time. The size of the radio's internal buffer dictates this limit so if you want to send longer messages you'll need to break them into a series of smaller send calls in your application code.
- Receiving packets is a 'best effort' in pure Python code. Unlike the Arduino versions of the RFM9x library there is no interrupt support which means when a packet is received it must be immediately processed by the Python code or it could be lost. For your application it will work best to only receive small, single packet messages at a time. Don't try to receive kilobytes of data or else you'll lose packets. This module is really intended for simple single packet messages like 'ON', 'OFF', etc.
- Sending and receiving packets will 'block' your Python code until the packet is fully processed. This means you can't do a lot of other things while sending and waiting for packets to be received. Design your application so the radio usage is the primary scenario and very little other tasks need to happen in the background.
- The module is written to be compatible with the RadioHead RFM95 Arduino library. This means by default the module will setup the radio with the same modulation and configuration for transmitting and receiving at the maximum distance with LoRa support. In addition the CircuitPython module uses the same packet preamble (8 bytes) and header (4 bytes) as RadioHead. If you want to use different modulations or settings you'll need to configure the radio yourself after carefully consulting the datasheet.
- The CircuitPython module supports advanced RadioHead features like the node addressing and "Reliable Datagram". "Reliable DataGram" mode in CircuitPython has some additional parameters to control timing that are not available with the RadioHead library. It may be difficult to get reliable transmission to work between the RadioHead library and CircuitPython.
- Encryption and sync words are also not supported by the LoRa radio module. You must perform these operations yourself in your application code if they're desired.

Wiring With Breakout

First wire up a RFM9x breakout to your board as shown on the previous pages for Arduino. Note that the G0/interrupt line is not used by the CircuitPython module and can remain unconnected. Here's an example of wiring a Feather M0 to the radio with a SPI connection:



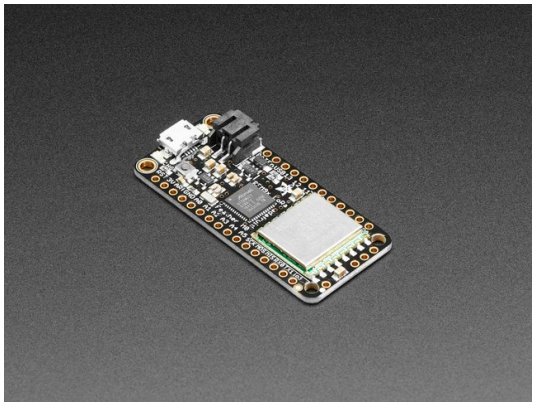
- **Board 3V** to **radio VIN**
- **Board GND** to **radio GND**
- **Board SCK** to **radio SCK**
- **Board MOSI** to **radio MOSI**
- **Board MISO** to **radio MISO**
- **Board D5** to **radio CS** (or any other digital I/O pin)
- **Board D6** to **radio RST** (or any other digital I/O pin)

The Feather M0 LoRa does NOT come with UF2 bootloader or CircuitPython pre-installed, you must do 2 things before you can use it with CircuitPython - update the bootloader manually, and install CircuitPython

Usage with All-In-One Feather M0

Alternatively you can use the Feather M0 RFM9x board but be sure you've loaded the adafruit-circuitpython-feather_m0_rfm9x-*.bin (https://adafru.it/tBa) version of CircuitPython on your board! **This is very important as the RFM9x build has special pins added to the board module which are used to access the radio's control lines!**

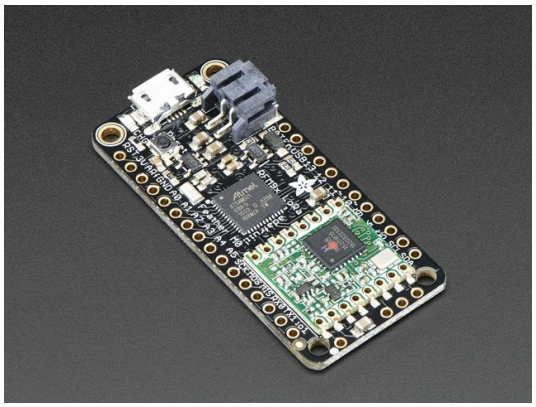
For details on how to load a binary circuitpython build, check out our Non-UF2-Install guide (https://adafru.it/Bed)



Adafruit Feather M0 with RFM95 LoRa Radio - 900MHz

OUT OF STOCK

Out Of Stock



Adafruit Feather M0 RFM96 LoRa Radio - 433MHz

$34.95

IN STOCK

Add To Cart

Module Install

Next you'll need to install the Adafruit CircuitPython RFM9x (https://adafru.it/BjD) module on your CircuitPython board. Before you do that make sure you are running the latest version of Adafruit CircuitPython (https://adafru.it/Amd) for your board too (again be sure to the load the Feather M0 RFM9x version if you're using that board and want to use its built-in radio module).

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). Our introduction guide has a great page on how to install the library bundle (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_rfm9x.mpy**
- **adafruit_bus_device**

You can also download the **adafruit_rfm9x.mpy** from its releases page on Github (https://adafru.it/Bl1).

Before continuing make sure your board's lib folder or root filesystem has the **adafruit_rfm9x.mpy,** and **adafruit_bus_device** files and folders copied over.

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

Usage

To demonstrate the usage of the radio we'll initialize it and send and receive data from the board's Python REPL.

Run the following code to import the necessary modules and initialize the SPI connection with the radio:

```
import board
import busio
import digitalio
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
```

Now define a few of the pins connected to the RFM9x, specifically the CS and RST pins:

```
cs = digitalio.DigitalInOut(board.D5)
reset = digitalio.DigitalInOut(board.D6)
```

However if you're using the Feather M0 RFM69 board with a built-in RFM9x radio (and you've loaded the special version of CircuitPython just for this board as mentioned above), you instead want to use these pins for the CS and RST lines:

```
cs = digitalio.DigitalInOut(board.RFM9X_CS)
reset = digitalio.DigitalInOut(board.RFM9X_RST)
```

You're ready to import the RFM9x module and create an instance of the RFM9x class inside it. Before you create the radio module instance you'll need to check if you're using a 433mhz or 915mhz radio module as the initializer requires the frequency to be specified--confirm which frequency your module uses and run **one** of the following lines.

For a 915mhz radio use:

```
import adafruit_rfm9x
rfm9x = adafruit_rfm9x.RFM9x(spi, cs, reset, 915.0)
```

Or for a 433mhz radio use:

```
import adafruit_rfm9x
rfm9x = adafruit_rfm9x.RFM9x(spi, cs, reset, 433.0)
```

Notice the initializer takes the following required parameters:

- **spi** - The SPI bus connected to the board.
- **cs** - The DigitalInOut instance connected to the CS line of the radio.
- **reset** - The DigitalInOut instance connected to the RST or reset line of the radio.
- **frequency** - The frequency in megahertz of the radio module. Remember this frequency depends on which type of radio you're using and the frequency you desire to use!

In addition there are some optional parameters you might specify:

- **baudrate** - The baud rate to use for the SPI connection to the radio. By default this is 10mhz which is as fast as the radio can handle, but in some cases it might be too fast if you're wiring up a breakout to a breadboard (breadboards can be notorious for not working well with high speed signals). If you run into odd errors like being unable to find the RFM9x radio try lowering the baudrate by specifying a **baudrate=1000000** keyword (which sets the speed to a lower 1mhz value).

Once the RFM9x class is created and initialized you're ready to start sending and receiving data.

Remember by default the module will be configured to interface with the "RadioHead" RFM9x setup so you can also send and receive packets with an Arduino running the RFM95 TX/RX examples!

To send a message simply call the **send** function and provide a string or byte string of data:

```
rfm9x.send('Hello world!')
```

```
Received nothing! Listening again...
Received (raw bytes): bytearray(b'Hello World!')
Received (ASCII): Hello World!
Received nothing! Listening again...
```

Remember you can only send a message up to 252 bytes in length at a time! Attempting to send a message longer than 252 bytes will fail with an exception error. If you need to send a longer message it will have to be broken up into multiple send calls and reconstructed on the receiving side.

If you have another RFM9x on the same frequency waiting to receive messages (like another CircuitPython module running receive code below) you should see it receive the message.

You can even have an Arduino running the RadioHead library's RFM95 client example see the message that was sent:

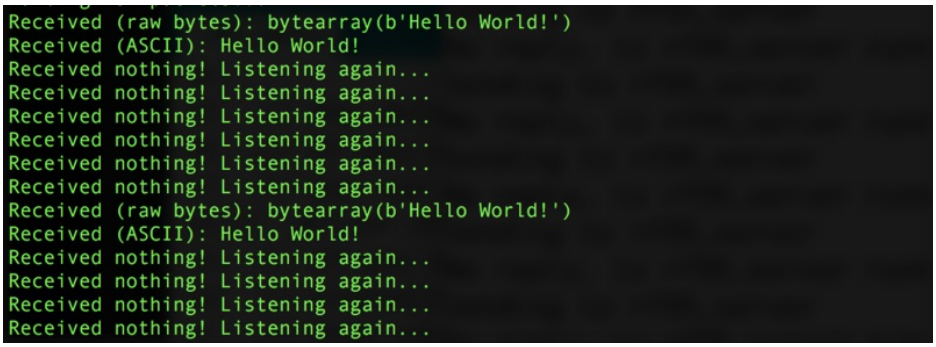
To receive a message simply call the `receive` function. This function will wait for half a second for any packet to be received. If a packet is found it will be returned as a byte string (remember packets are at most 252 bytes long), or if no packet was found a result of `None` is returned.

```
rfm9x.receive()
```

You can increase the amount of time the module waits for a packet to be received by specifying the time in seconds as a parameter to the receive call:

```
rfm9x.receive(timeout_s=5.0)  # Wait 5 seconds instead of 0.5 seconds.
```

Notice this waits longer at the REPL for a packet to be received before returning. If you have another RFM9x setup try having it send a message while the other is waiting to receive it. You should see a byte string returned. You can also have an Arduino running the RadioHead library's RFM95 client example send messages that are received by your code:



One thing to note in Python byte strings aren't exactly like text strings and you might not be able to do all the text processing (like find, replace, etc.) as you expect. However you can convert a byte string into text by assuming a specific text encoding like ASCII. For example to receive a packet and convert the contents to an ASCII text string you can run code like:

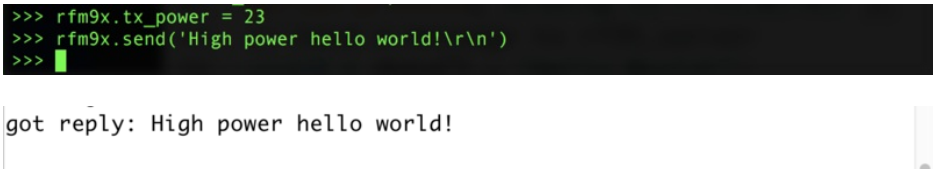
```
packet = rfm9x.receive()  # Wait for a packet to be received (up to 0.5 seconds)
if packet is not None:
    packet_text = str(packet, 'ascii')
    print('Received: {0}'.format(packet_text))
```

Notice this code first receives a packet, then checks if one was actually found (the packet is `not None` check--if no packet is received a value of `None` is returned), and then converts the packet data to a string assuming an ASCII text encoding.

Beyond RX & TX

Beyond basic sending and receiving there are a few properties of the RFM69 class you might want to interact with:

- **tx_power** - This is a power level (in dB) to use when transmitting with the radio. By default this is set to a moderate 13 dB value, however you can increase this depending on the type of radio you're using. For high power radios (the modules sold by Adafruit) they support a range of TX power from 5 to 23 dB. Try increasing this to the maximum 23 dB level (however check your local laws for permission to transmit with such power!) to get the most distance and range.
- **rssi** - The received signal strength indicator is a property you can read to see the strength of the radio signal being received. This is updated when packets are received and returns a value in decibels (typically negative, so the *smaller* the number and closer to 0, the higher the strength / better the signal).



```
>>> rfm9x.receive(timeout_s=5.0)
bytearray(b'Hello World!')
>>> print('RSSI: {0} dB'.format(rfm9x.rssi))
RSSI: -73 dB
>>>
```

That's all there is to the basic RFM9x radio usage! *Remember the CircuitPython module is designed for sending and receiving small up to 252 byte control messages and not large or high bandwidth amounts of data.*

Here's a complete example of sending a message and waiting to receive and print any received messages. Save this as **main.py** on your board and open the serial REPL to see it print data and any received messages. If you have two boards and radios setup to run this code at the same time they'll send each other a message on start up!

```
# Simple demo of sending and recieving data with the RFM95 LoRa radio.
# Author: Tony DiCola
import board
import busio
import digitalio

import adafruit_rfm9x


# Define radio parameters.
RADIO_FREQ_MHZ = 915.0  # Frequency of the radio in Mhz. Must match your
# module! Can be a value like 915.0, 433.0, etc.

# Define pins connected to the chip, use these if wiring up the breakout according to the guide:
CS = digitalio.DigitalInOut(board.D5)
RESET = digitalio.DigitalInOut(board.D6)
# Or uncomment and instead use these if using a Feather M0 RFM9x board and the appropriate
# CircuitPython build:
# CS = digitalio.DigitalInOut(board.RFM9X_CS)
# RESET = digitalio.DigitalInOut(board.RFM9X_RST)

# Define the onboard LED
LED = digitalio.DigitalInOut(board.D13)
LED.direction = digitalio.Direction.OUTPUT

# Initialize SPI bus.
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)

# Initialze RFM radio
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, RADIO_FREQ_MHZ)

# Note that the radio is configured in LoRa mode so you can't control sync
# word, encryption, frequency deviation, or other settings!

# You can however adjust the transmit power (in dB).  The default is 13 dB but
# high power radios like the RFM95 can go up to 23 dB:
rfm9x.tx_power = 23

# Send a packet.  Note you can only send a packet up to 252 bytes in length.
# This is a limitation of the radio packet size, so if you need to send larger
# amounts of data you will need to break it into smaller send calls.  Each send
# call will wait for the previous one to finish before continuing.
rfm9x.send(bytes("Hello world!\r\n", "utf-8"))
print("Sent Hello World message!")

# Wait to receive packets.  Note that this library can't receive data at a fast
```

```
# Wait to receive packets.  Note that this library can't receive data at a fast
# rate, in fact it can only receive and process one 252 byte packet at a time.
# This means you should only use this for low bandwidth scenarios, like sending
# and receiving a single message at a time.
print("Waiting for packets...")

while True:
    packet = rfm9x.receive()
    # Optionally change the receive timeout from its default of 0.5 seconds:
    # packet = rfm9x.receive(timeout=5.0)
    # If no packet was received during the timeout then None is returned.
    if packet is None:
        # Packet has not been received
        LED.value = False
        print("Received nothing! Listening again...")
    else:
        # Received a packet!
        LED.value = True
        # Print out the raw bytes of the packet:
        print("Received (raw bytes): {0}".format(packet))
        # And decode to ASCII text and print it too.  Note that you always
        # receive raw bytes and need to convert to a text format like ASCII
        # if you intend to do string processing on your data.  Make sure the
        # sending side is sending ASCII data before you try to decode!
        packet_text = str(packet, "ascii")
        print("Received (ASCII): {0}".format(packet_text))
        # Also read the RSSI (signal strength) of the last received message and
        # print it.
        rssi = rfm9x.last_rssi
        print("Received signal strength: {0} dB".format(rssi))
```

Radio Range F.A.Q.

Which gives better range, LoRa or RFM69?

All other things being equal (antenna, power output, location) you will get better range with LoRa than with RFM69 modules. We've found 50% to 100% range improvement is common.

What ranges can I expect for RFM69 radios?

The RFM69 radios have a range of approx. 500 meters **line of sight** with tuned uni-directional antennas. Depending on obstructions, frequency, antenna and power output, you will get lower ranges - *especially* if you are not line of sight.

What ranges can I expect for RFM9X LoRa radios?

The RFM9x radios have a range of up to 2 km **line of sight** with tuned uni-directional antennas. Depending on obstructions, frequency, antenna and power output, you will get lower ranges - *especially* if you are not line of sight.

I don't seem to be getting the range advetised! Is my module broken?

Your module is probably *not* broken. Radio range is dependant on *a lot of things* and all must be attended to to make sure you get the best performance!

1. Tuned antenna for your frequency - getting a well tuned antenna is incredibly important. Your antenna must be tuned for the exact frequency you are using
2. Matching frequency - make sure all modules are on the same exact frequency
3. Matching settings - all radios must have the same settings so they can communicate
4. Directional vs non-directional antennas - for the best range, *directional* antennas like Yagi will direct your energy in one path instead of all around
5. Good power supply - a nice steady power supply will keep your transmissions clean and strong
6. Max power settings on the radios - they can be set for higher/lower power! Don't forget to set them to max.
7. Line of sight - No obstructions, walls, trees, towers, buildings, mountains, etc can be in the way of your radio path. Likewise, outdoors is way better than indoors because its very hard to bounce radio paths around a building
8. Radio transmission speed - trying to transmit more data faster will be hard. Go for small packets, with lots of retransmissions. Lowering the baud rate on the radio (see the libraries for how to do this) will give you better reliability

How do I pick/design the right antenna?

Various antennas will cost diferent amounts and give you different directional gain. In general, spending a lot on a large fixed antenna can give you better power transfer if the antenna is well tuned. For most simple uses, a wire works pretty well

The ARRL antena book is recommended if you want to learn how to do the modeling and analysis

But nothing beats actual tests in your environment!

What frequency is my module?

Look for a little colored paint dot on top of the module.

- **GREEN** or **BLUE** = **900 MHz**
- **RED** = **433 MHz**

Every now and then the paint dot shows up without a color or with the ink dot burnt. This is just a manufacturing variance and there is nothing wrong with the board. You should get the frequency you ordered though. So if you plan on mixing these up, you may want to add a new mark of your own.

My radio has a burnt blob on it, is it damaged?

Nope! The radios have an ink dot on them, which sometimes gets toasty when we put the board through the oven, or rework it, so it may have a burnt appearance. The chip is fine!

Downloads

Datasheets & Files

For the LoRa version:

- SX127x Datasheet (https://adafru.it/oBm)- The RFM9X LoRa radio chip itself
- RFM9X (https://adafru.it/FTK) - The radio module, which contains the SX1272 chipset

For the RFM69 version:

- SX1231 Transceiver Datasheet (https://adafru.it/mCv)
- RFM69HCW Datasheet: V1.1 (https://adafru.it/mCu)
- FCC Test Report (https://adafru.it/oC0)
- RoHS Test Report (https://adafru.it/oC1)
- RoHS Test Report (https://adafru.it/oC2)
- REACH Test Report (https://adafru.it/oC3)

For both:

- PCB files on GitHub (https://adafru.it/obv)
- Fritzing Library (https://adafru.it/aP3)

https://adafru.it/z4a

https://adafru.it/z4a

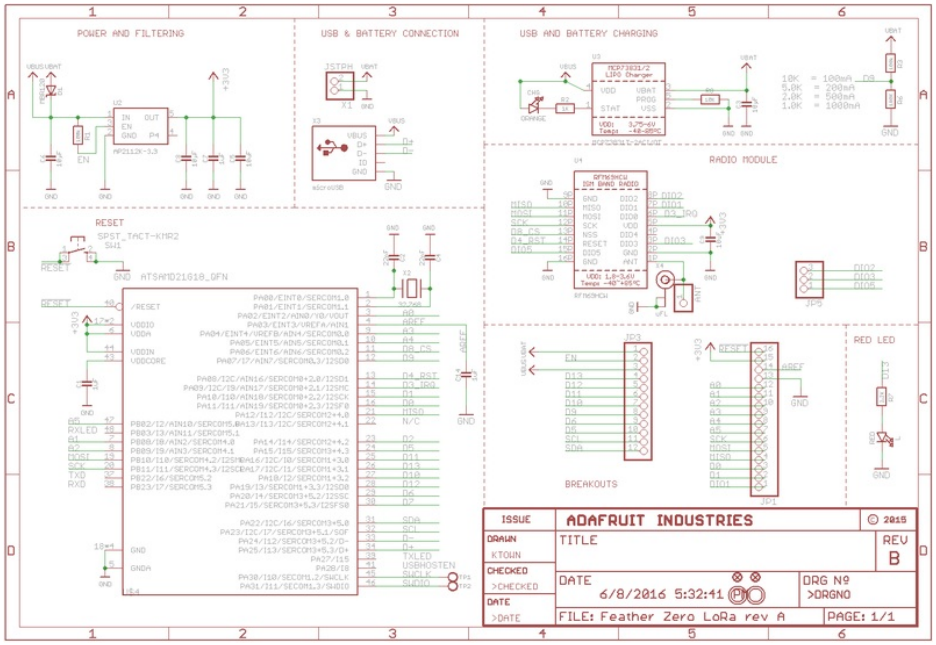
https://adafru.it/z4b

https://adafru.it/z4b

Note AREF in the diagrams should be marked PA03 not PA02

Schematic

Click to embiggen. Same schematic for both LoRa and RFM69 (the modules have the same pinout)



Fabrication Print

Dimensions in Inches. Same PCB design for both LoRa and RFM69 (the modules have the same pinout)