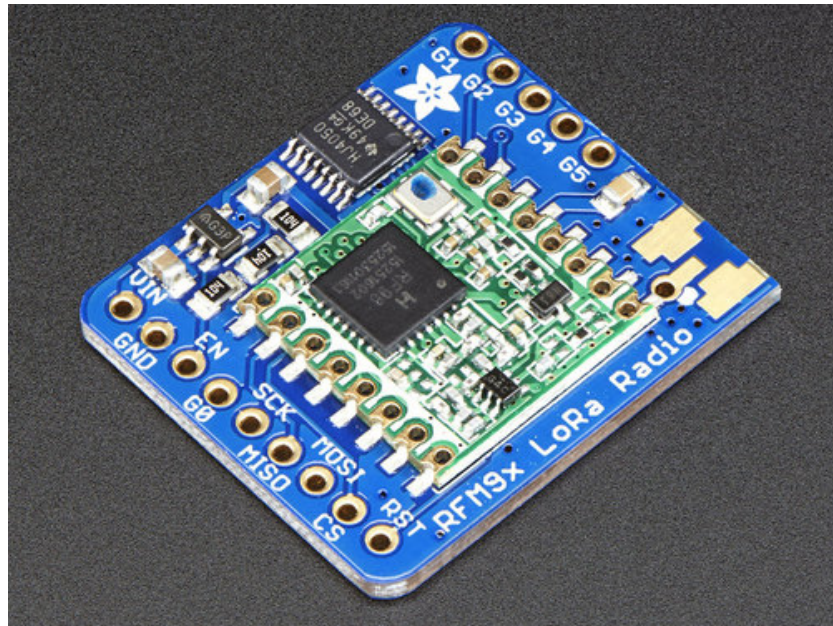


Adafruit RFM69HCW and RFM9X LoRa Packet Radio Breakouts

Created by lady ada



Last updated on 2017-08-01 04:24:49 AM UTC

Guide Contents

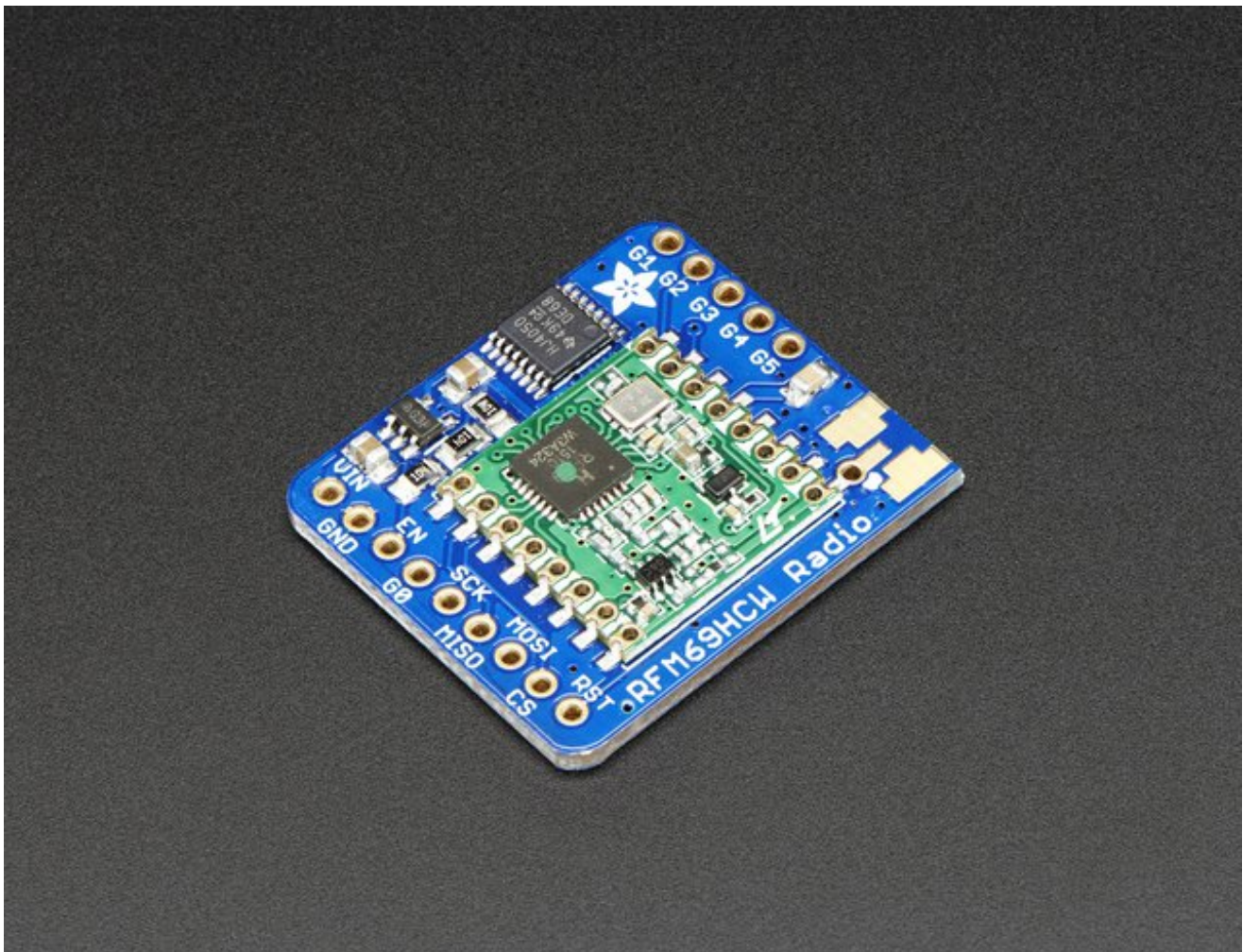
Guide Contents	2
Overview	4
Pinouts	9
Power Pins	9
SPI Logic pins:	10
Radio GPIO	11
Antenna Connection	12
Assembly	14
Prepare the header strip:	14
Add the breakout board:	15
And Solder!	16
Antenna Options	17
Wire Antenna	18
uFL Connector	20
uFL SMT Antenna Connector	20
SMA to uFL/u.FL/IPX/IPEX RF Adapter Cable	21
SMA Edge-Mount Connector	24
Wiring	29
Using the RFM69 Radio	31
"Raw" vs Packetized	32
Arduino Libraries	32
RadioHead Library example	32
Basic RX & TX example	33
Basic Transmitter example code	33
Basic receiver example code	34
Radio Freq. Config	37
Configuring Radio Pinout	37
Setup	38
Initializing Radio	38
Basic Transmission Code	39
Basic Receiver Code	40
Basic Receiver/Transmitter Demo w/OLED	41
Addressed RX and TX Demo	42

RFM9X Test	46
Arduino Library	46
RadioHead RFM9x Library example	47
Basic RX & TX example	47
Transmitter example code	47
Receiver example code	50
Radio Pinout	54
Frequency	54
Setup	55
Initializing Radio	55
Transmission Code	56
Receiver Code	56
Downloads	58
Datasheets & Files	58
Schematic	58
Fabrication Print	59
Radio Range F.A.Q.	61

Overview

"You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat."

Sending data over long distances is like magic, and now you can be a magician with this range of powerful and easy-to-use radio modules. Sure, sometimes you want to talk to a computer (a good time to use WiFi) or perhaps communicate with a Phone (choose Bluetooth Low Energy!) but what if you want to send data very far? Most WiFi, Bluetooth, Zigbee and other wireless chipsets use 2.4GHz, which is great for high speed transfers. If you aren't so concerned about streaming a video, you can use a lower [license-free ISM frequency bands](http://adafru.it/mOE) (<http://adafru.it/mOE>) such as 433MHz in ITU Europe or 900 MHz in ITU Americas. You can't send data as fast but you can send data a lot *farther*.



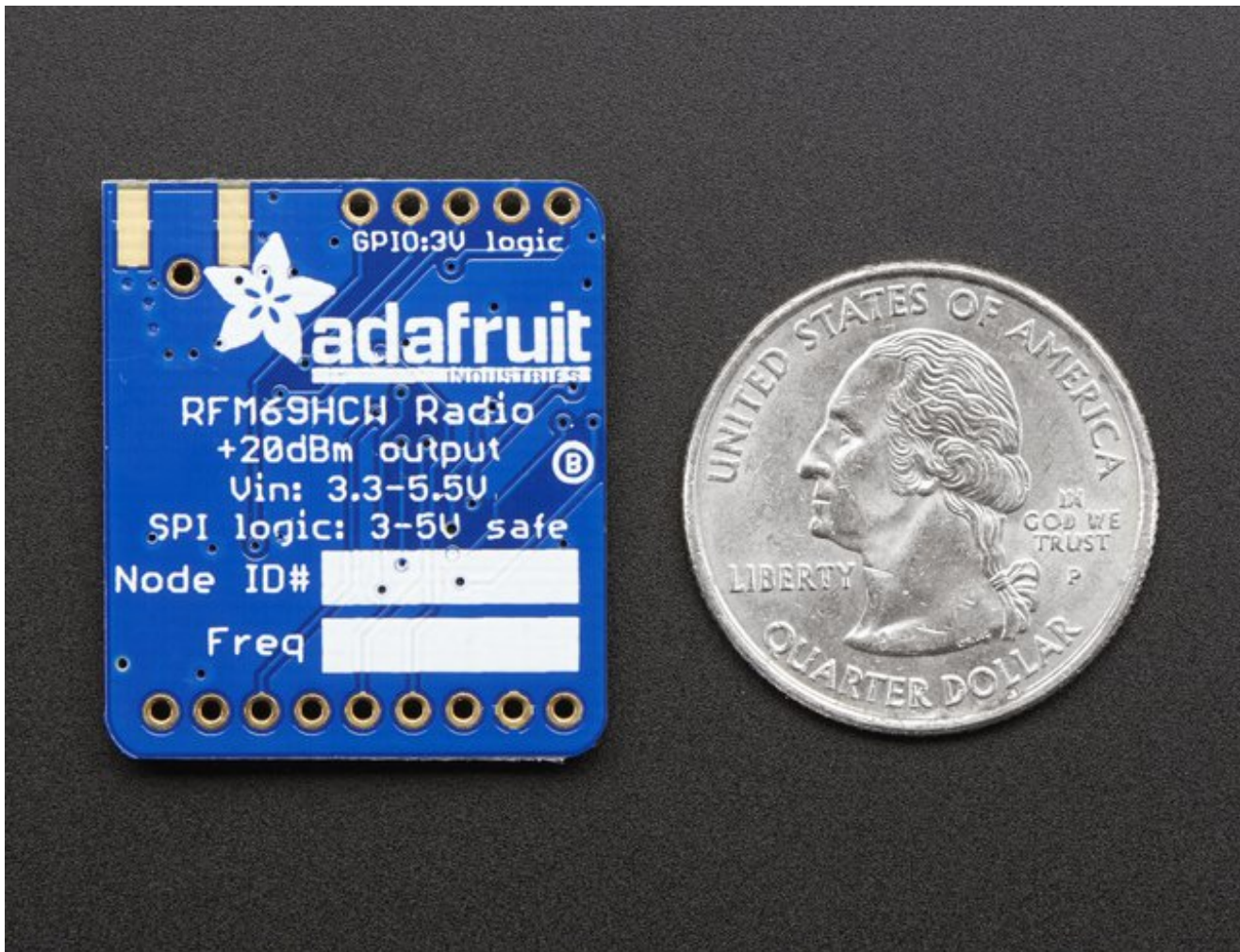
Also, these packet radios are simpler than WiFi or BLE, you don't have to associate, pair, scan, or worry about connections. All you do is send data whenever you like, and any other modules tuned to that same frequency (and, with the same encryption key) will receive. The receiver can then send a reply back. The modules do packetization, error correction and can also auto-retransmit so it's not like you have to worry about *everything* but less power is wasted on maintaining a link or pairing.

These modules are great for use with Arduinos or other microcontrollers, say if you want a sensor node network or transmit data over a campus or town. The trade off is you need two or more radios, with matching frequencies. WiFi and BT, on the other hand, are commonly included in computers and phones.

These radio modules come in **four variants** (two modulation types and two frequencies) The RFM69's are easiest to work with, and are well known and understood. The LoRa radios are exciting and more powerful but also more expensive.

All variants are:

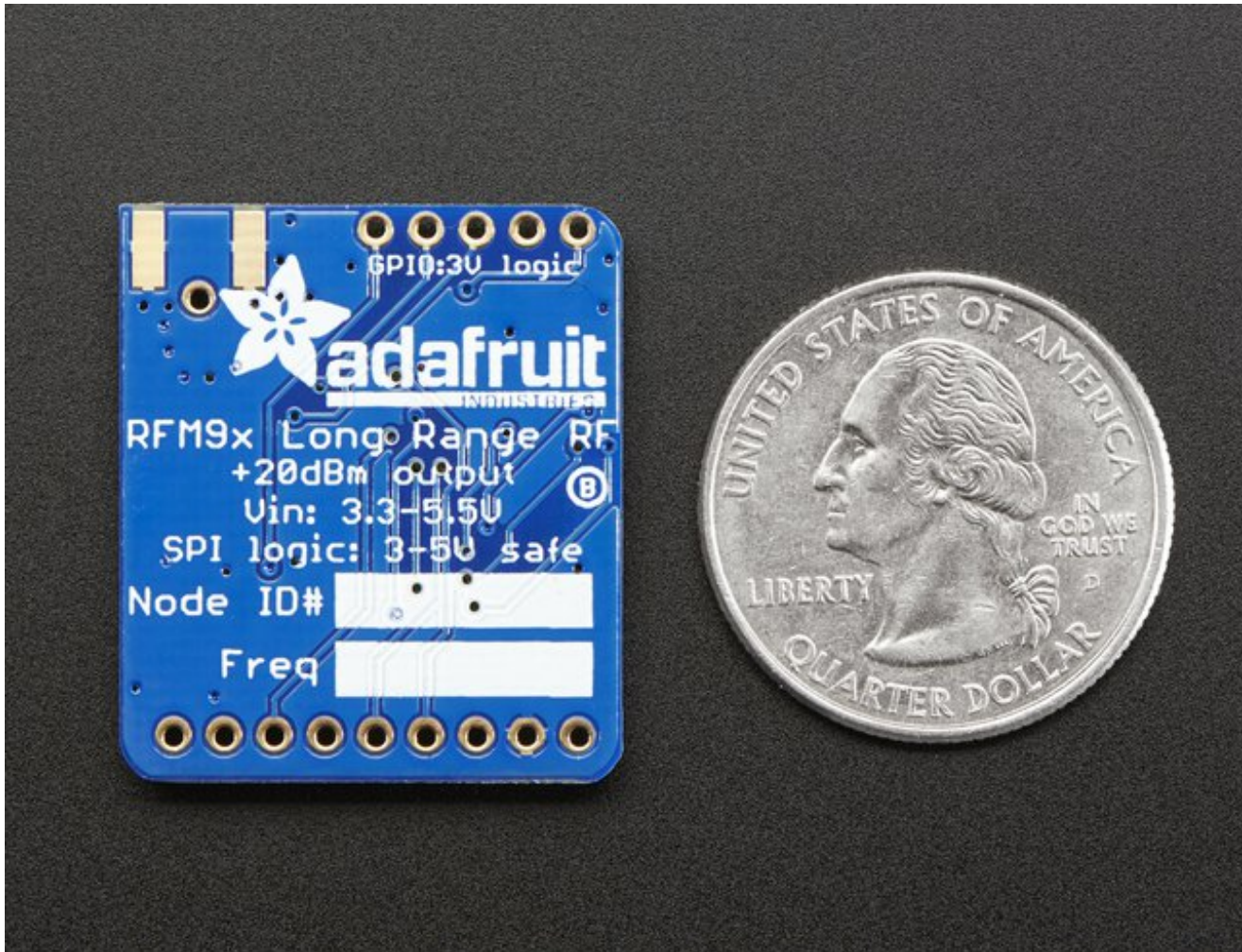
- Packet radio with ready-to-go Arduino libraries
- Uses the amateur or [license-free ISM bands](http://adafruit.com/moe) (<http://adafruit.com/moe>): 433MHz is ITU "Europe" license-free ISM or ITU "American" amateur with limitations. 900MHz is license free ISM for ITU "Americas"
- Use a simple wire antenna or spot for uFL or SMA radio connector



RFM69HCW in either 433 MHz or 868/915MHz

These are +20dBm FSK packet radios that have a lot of nice extras in them such as encryption and auto-retransmit. They can go about 200-500 meters line-of-sight using simple wire antennas, probably up to 5Km with well-tuned directional antennas, perfect line-of-sight, and settings tweakings

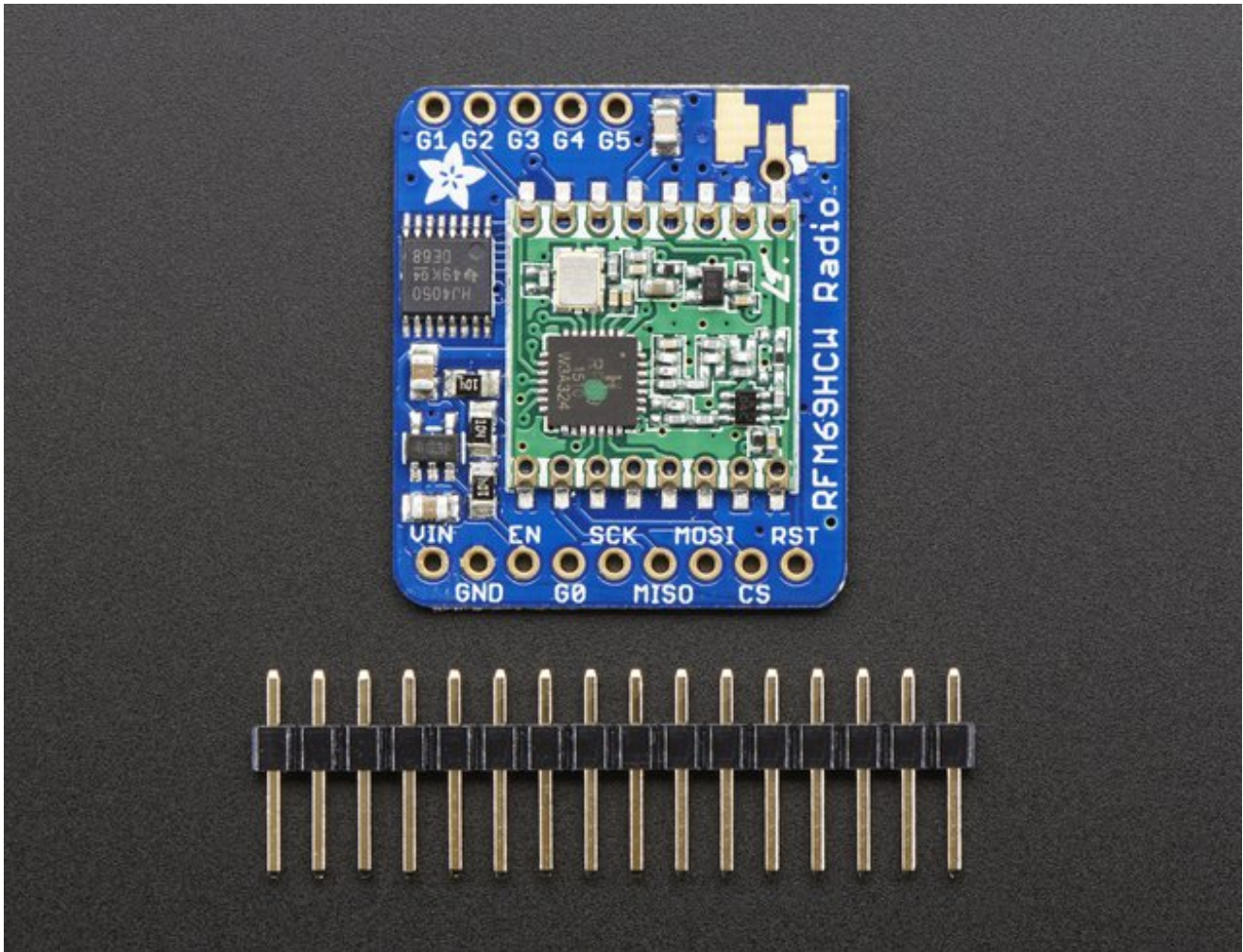
- SX1231 based module with SPI interface
- +13 to +20 dBm up to 100 mW Power Output Capability (power output selectable in software)
- 50mA (+13 dBm) to 150mA (+20dBm) current draw for transmissions, ~30mA during active radio listening.
- The RFM69 radios have a range of approx. 500 meters **line of 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.
- Create multipoint networks with individual node addresses
- Encrypted packet engine with AES-128



RFM9x LoRa in either 433 MHz or 868/915MHz

These are +20dBm LoRa packet radios that have a special radio modulation that is not compatible with the RFM69s *but* can go much much farther. They can easily go 2 Km line of sight using simple wire antennas, or up to 20Km with directional antennas and settings tweakings

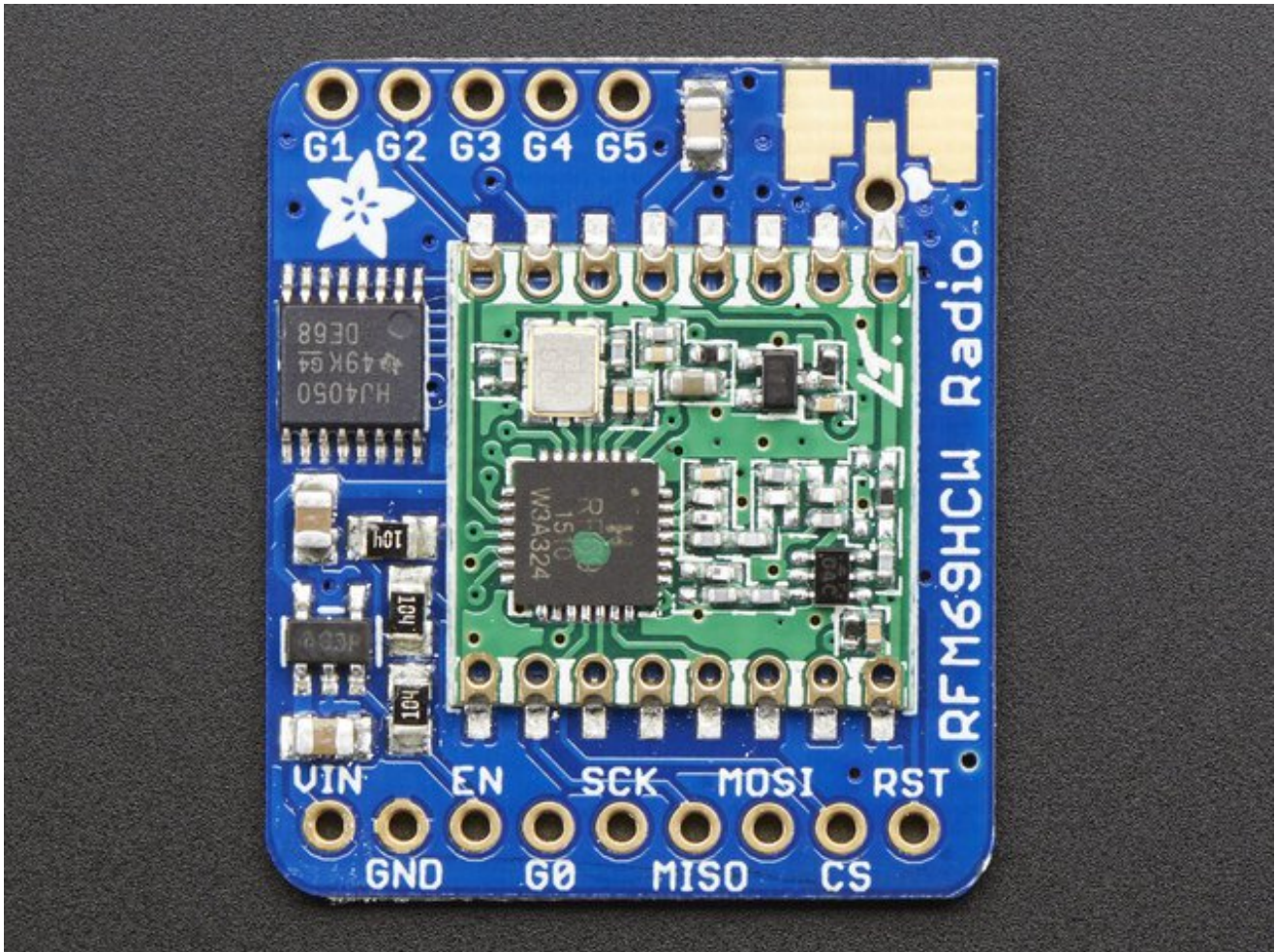
- SX1276 LoRa® based module with SPI interface
- +5 to +20 dBm up to 100 mW Power Output Capability (power output selectable in software)
- ~100mA peak during +20dBm transmit, ~30mA during active radio listening.
- The RFM9x radios have a range of approx. 2 km **line of sight** with tuned uni-directional antennas. Depending on obstructions, frequency, antenna and power output, you will get lower ranges - *especially* if you are not line of sight.



All radios are sold individually and can only talk to radios of the same part number. E.g. RFM69 900 MHz can only talk to RFM69 900 MHz, LoRa 433 MHz can only talk to LoRa 433, etc.

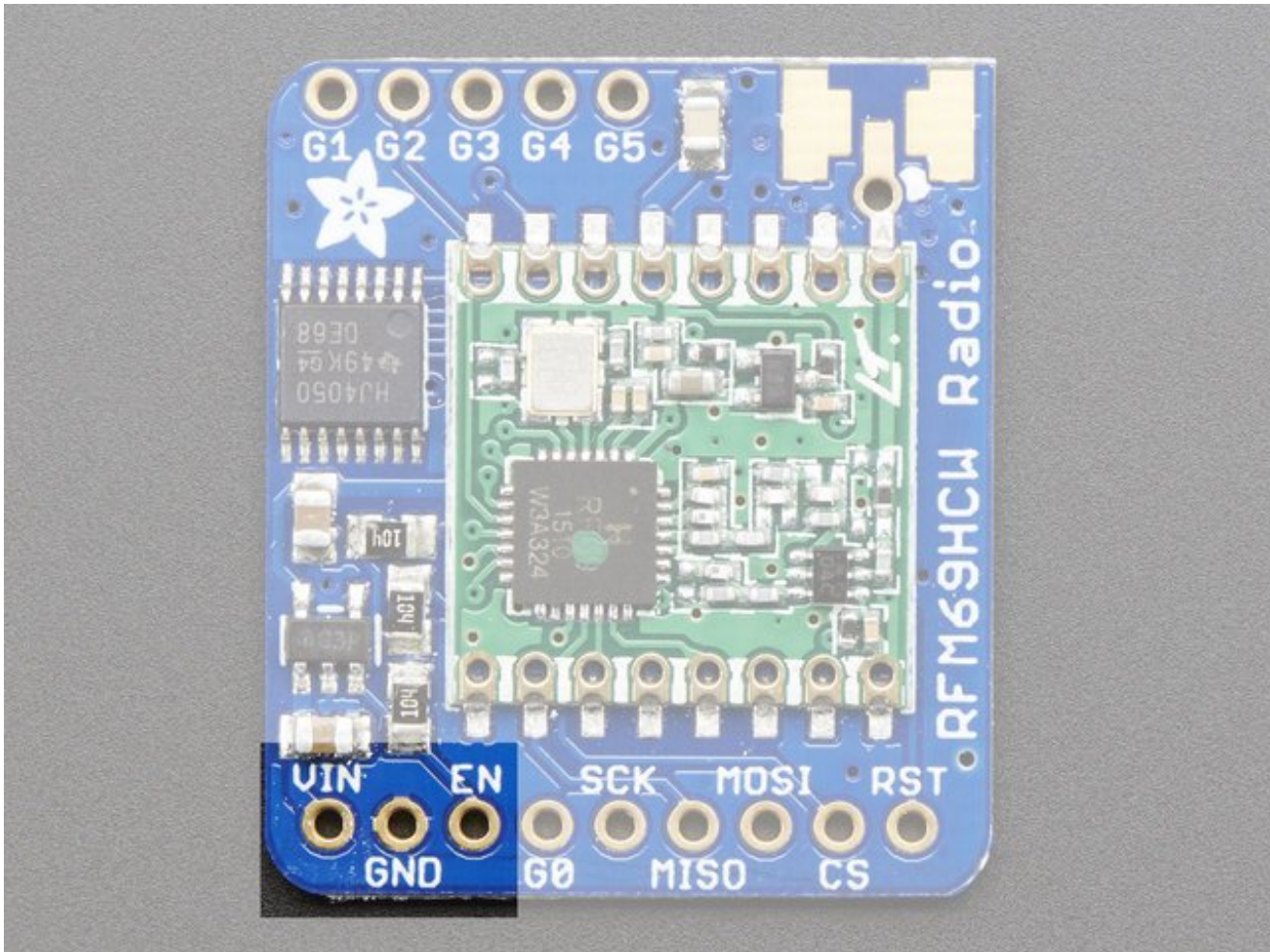
Each radio comes with some header, a 3.3V voltage regulator and levelshifter that can handle 3-5V DC power and logic so you can use it with 3V or 5V devices. Some soldering is required to attach the header. You will need to cut and solder on a small piece of wire (any solid or stranded core is fine) in order to create your antenna. Optionally you can pick up a uFL or SMA edge-mount connector and attach an external duck.

Pinouts



Both RFM69 and RFM9x LoRa breakouts have the exact same pinouts. The silkscreen will say RFM69HCW or LoRa depending on which variant you have. If there's a green or blue dot on top of the module, its 900 MHz. If there's a red dot, its 433 MHz

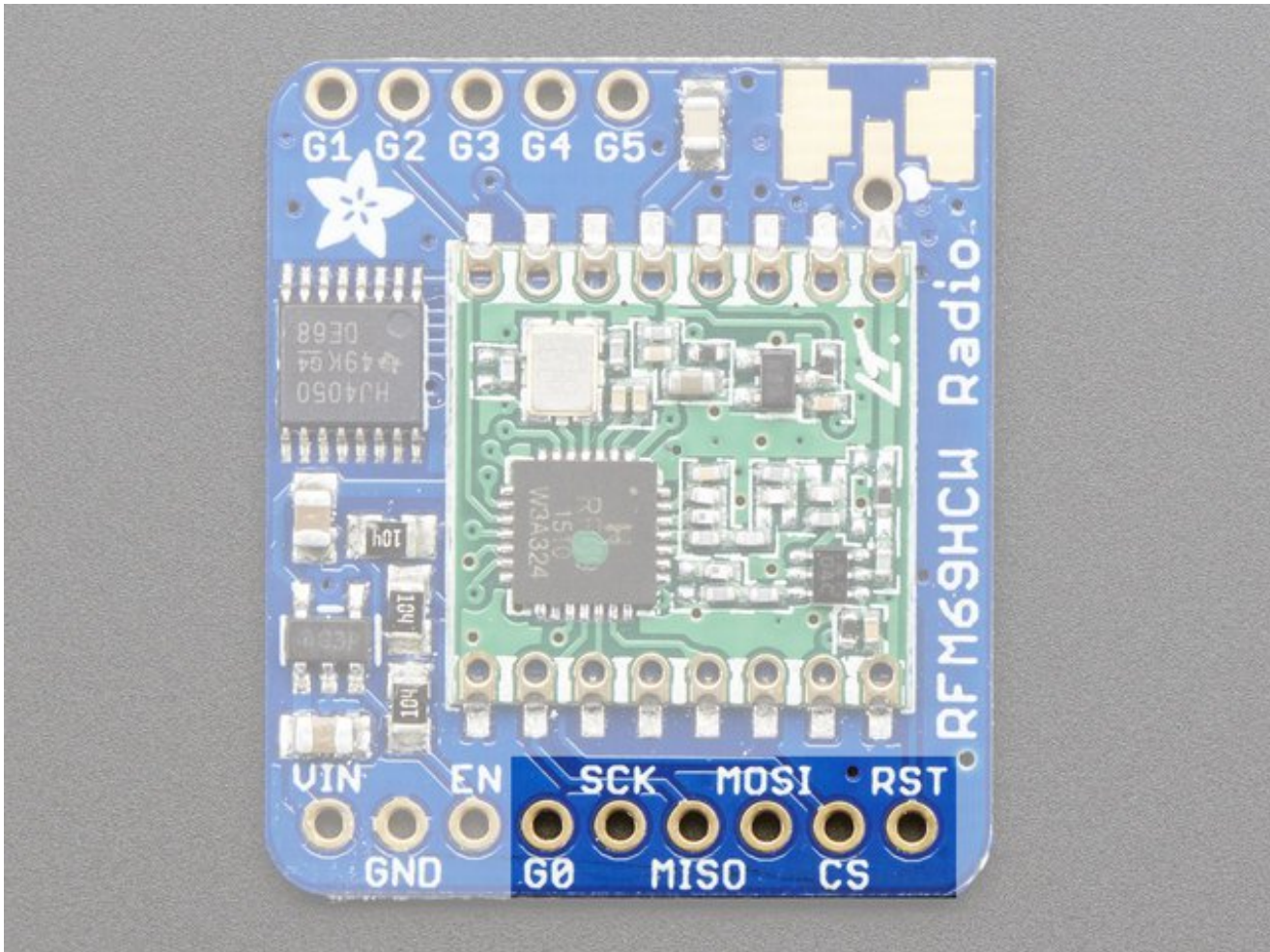
Power Pins



The left-most pins are used for power

- **Vin** - power in. This is regulated down to 3.3V so you can use 3.3-6VDC in. Make sure it can supply 150mA since the peak radio currents can be kinda high
- **GND** - ground for logic and power
- **EN** - connected to the enable pin of the regulator. Pulled high to **Vin** by default, pull low to completely cut power to the radio.

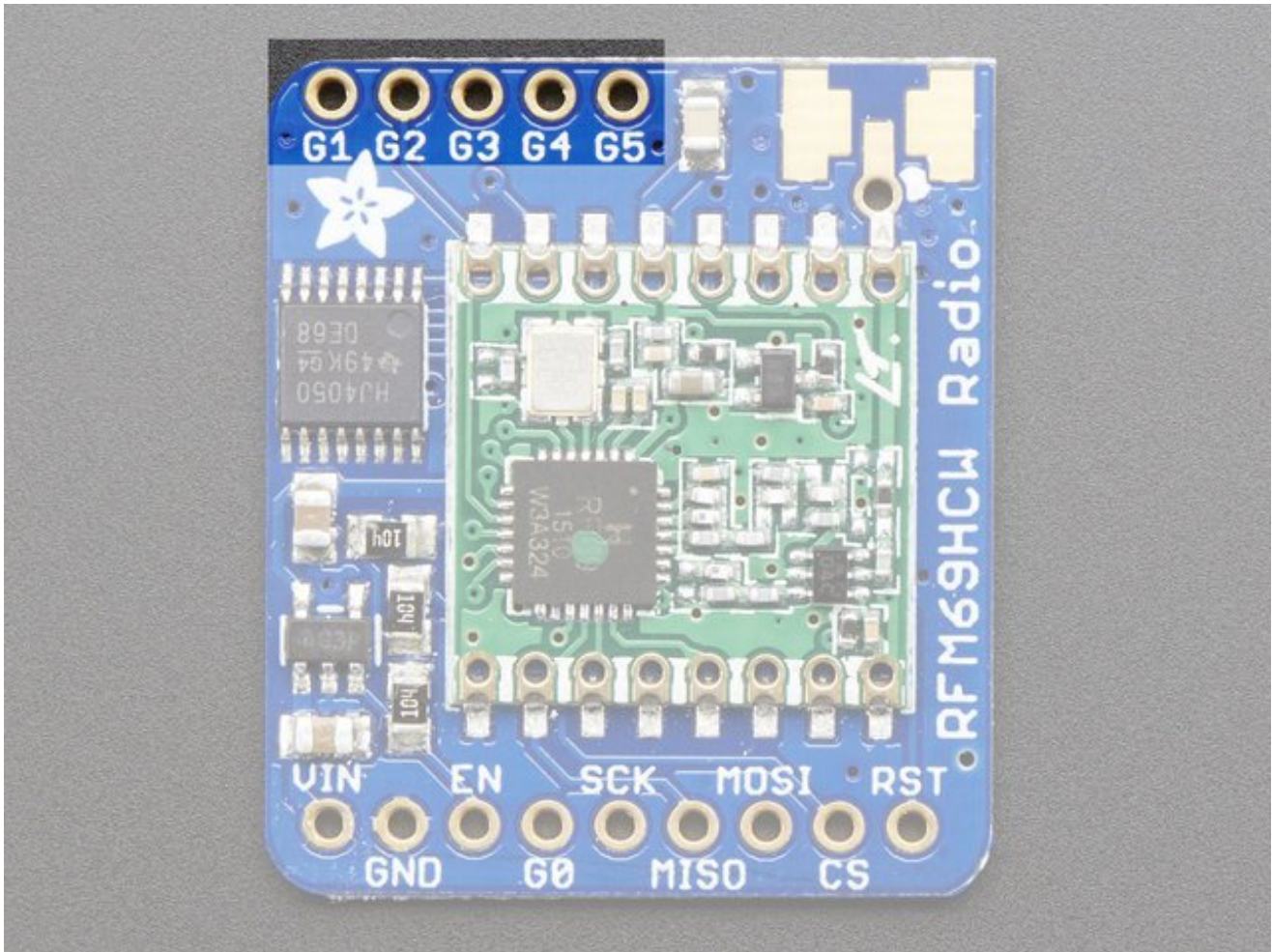
SPI Logic pins:



All pins going into the breakout have level shifting circuitry to make them 3-5V logic level safe. Use whatever logic level is on **Vin!**

- **SCK** - This is the **SPI Clock** pin, its an input to the chip
- **MISO** - this is the **Master In Slave Out** pin, for data sent from the radio to your processor, 3.3V logic level
- **MOSI** - this is the **Master Out Slave In** pin, for data sent from your processor to the radio
- **CS** - this is the **Chip Select** pin, drop it low to start an SPI transaction. Its an input to the chip
- **RST** - this is the **Reset** pin for the radio. It's pulled high by default. Pull down to ground to put it into reset
- **G0** - the radio's "GPIO 0" pin, also known as the **IRQ** pin, used for interrupt request notification from the radio to the microcontroller, 3.3V logic level

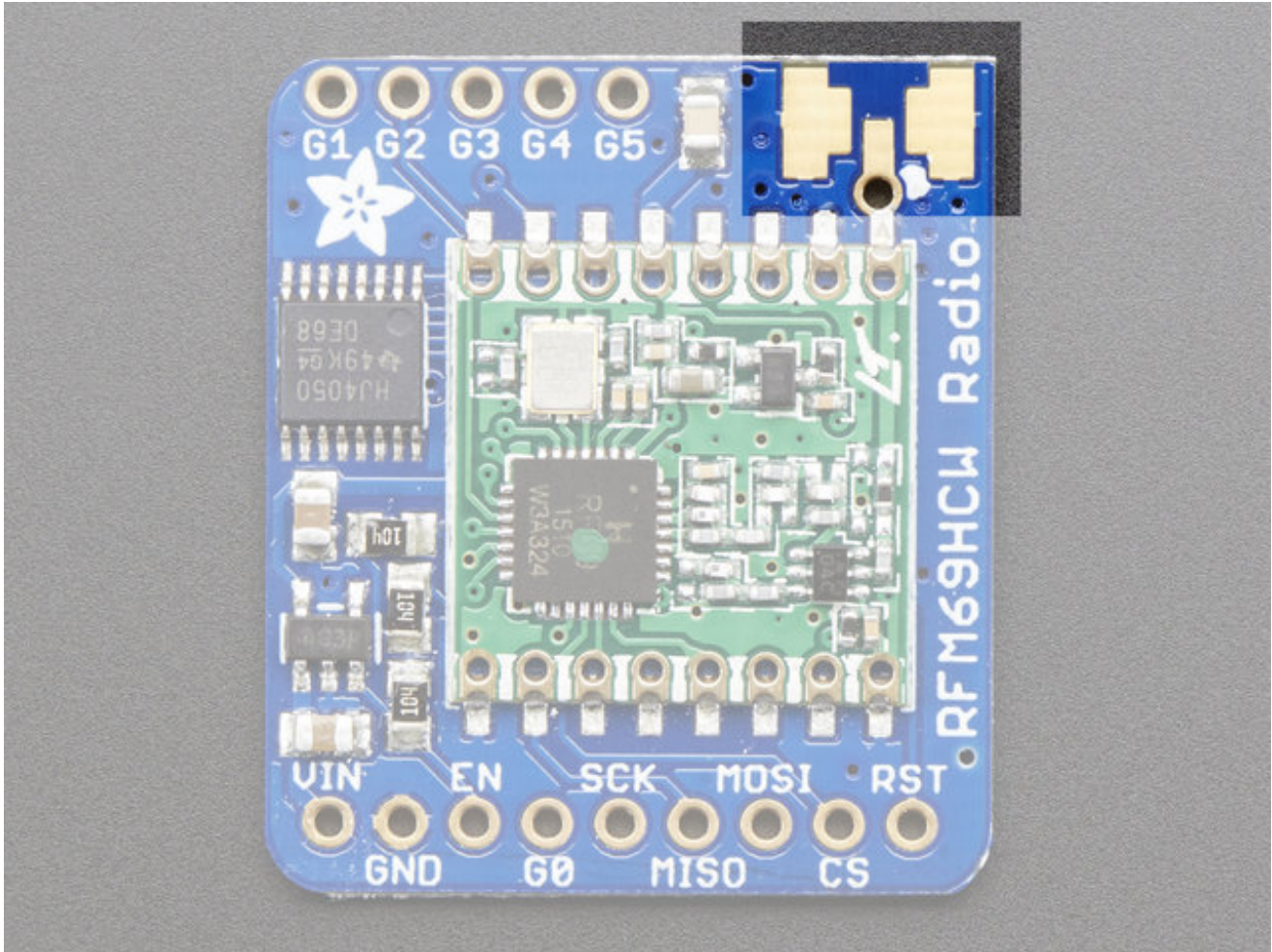
Radio GPIO



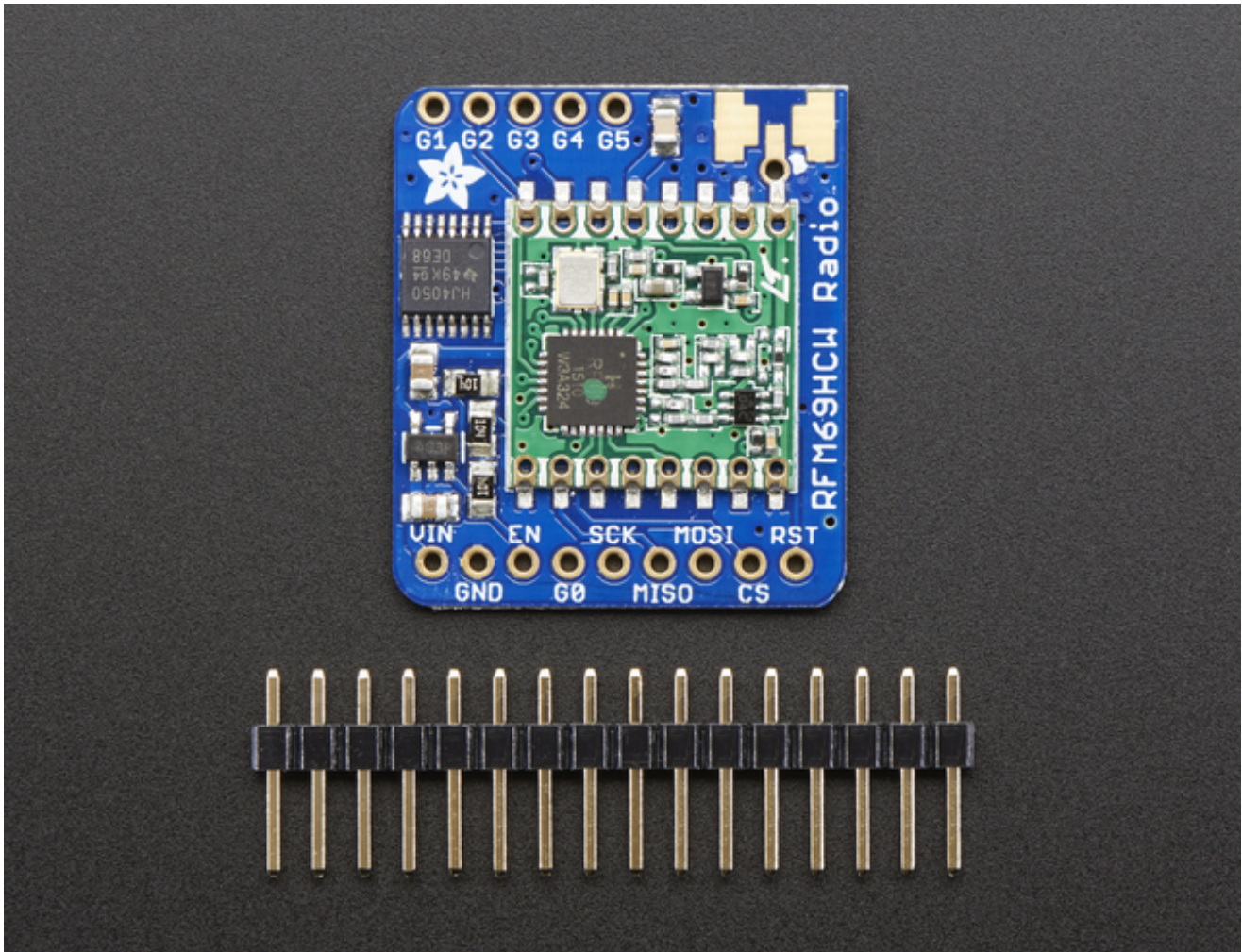
The radio's have another 5 GPIO pins that can be used for various notifications or radio functions. These aren't used for the majority of uses but are available in case you want them! All are 3.3V logic with no level shifting

Antenna Connection

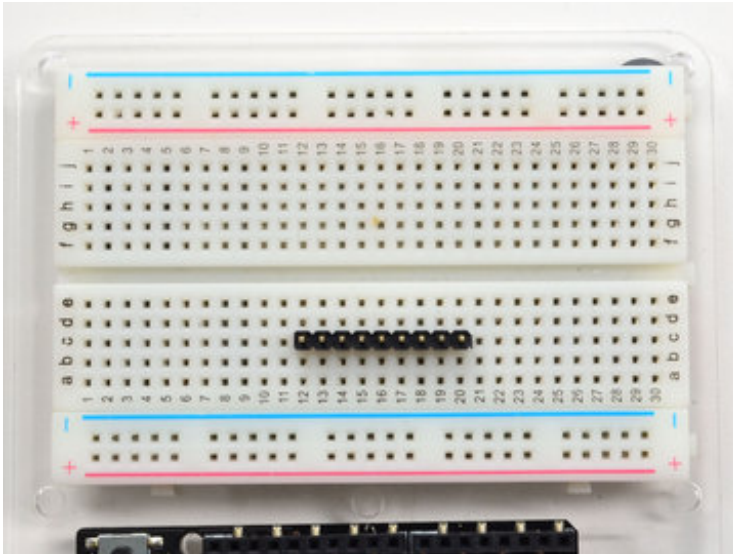
This three-way connection lets you select which kind of Antenna you'd like, from the lowest cost wire dipole to the fanciest SMA



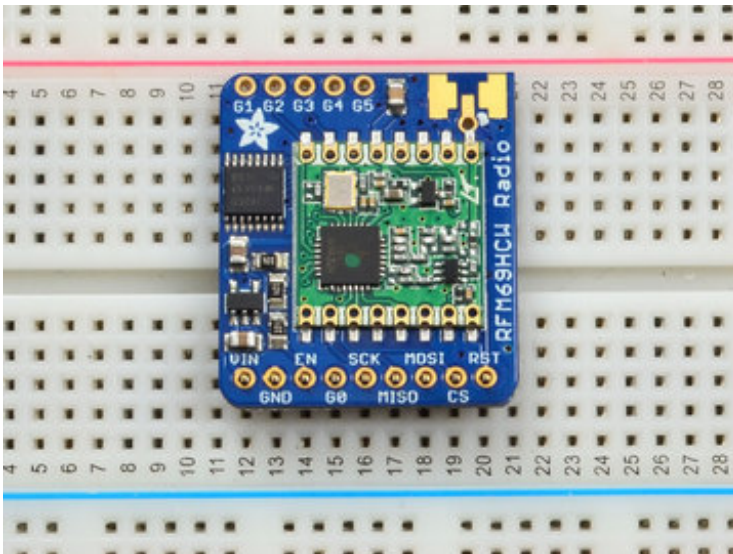
Assembly



Prepare the header strip:

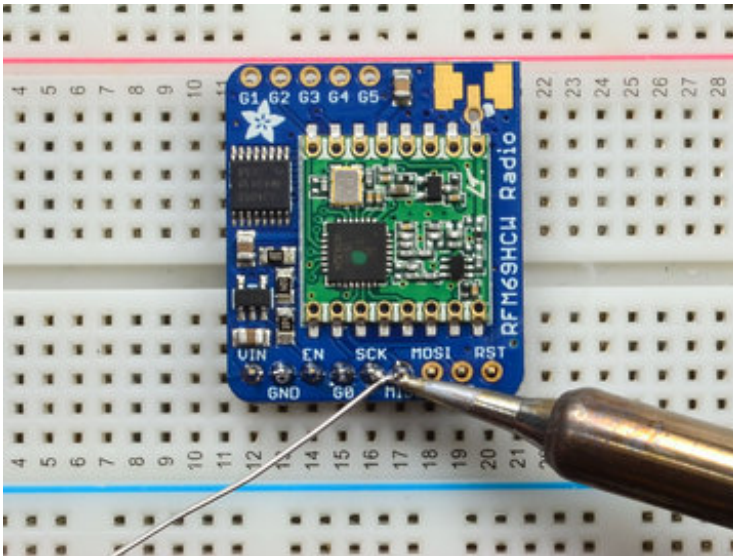
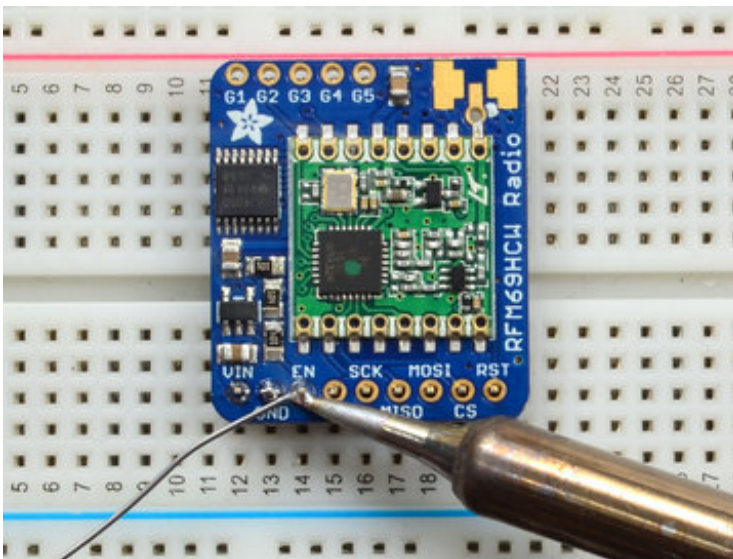
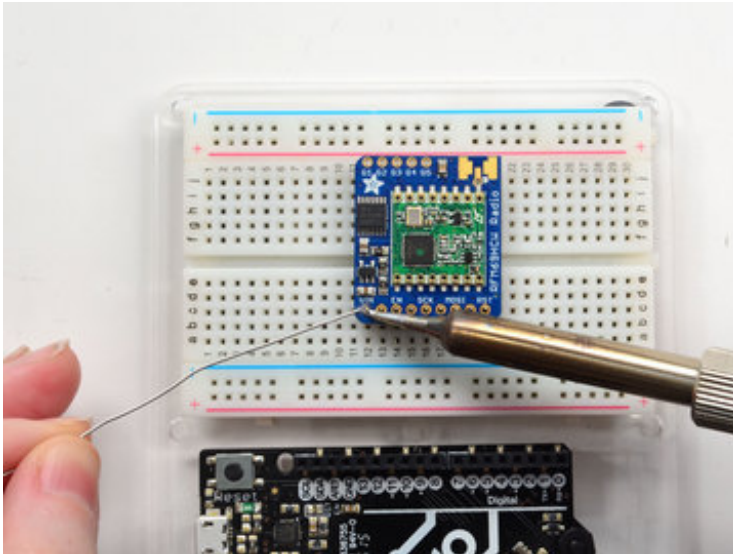


Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**



Add the breakout board:

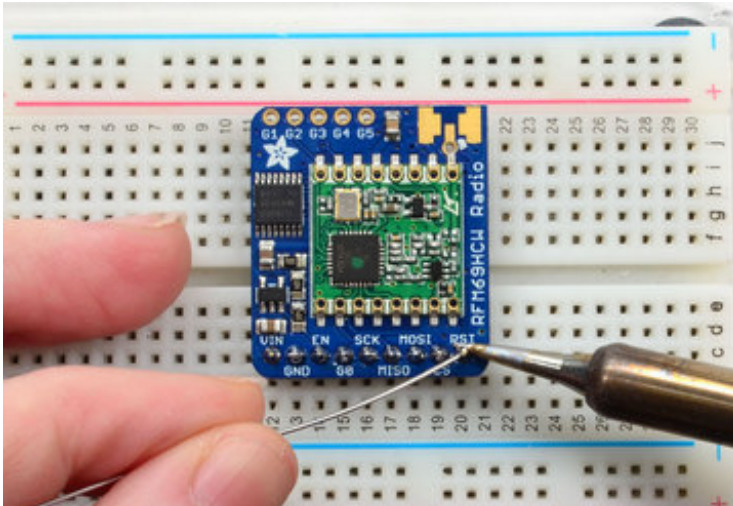
Place the breakout board over the pins so that the short pins poke through the breakout pads



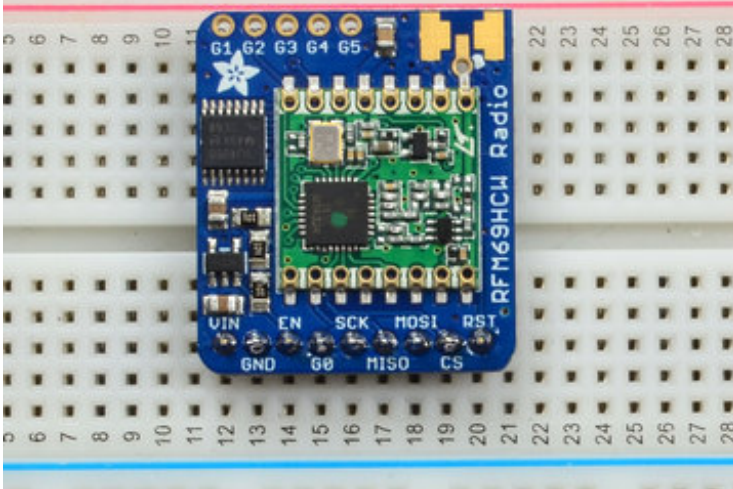
And Solder!

Be sure to solder all pins for reliable electrical contact.

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



•



•

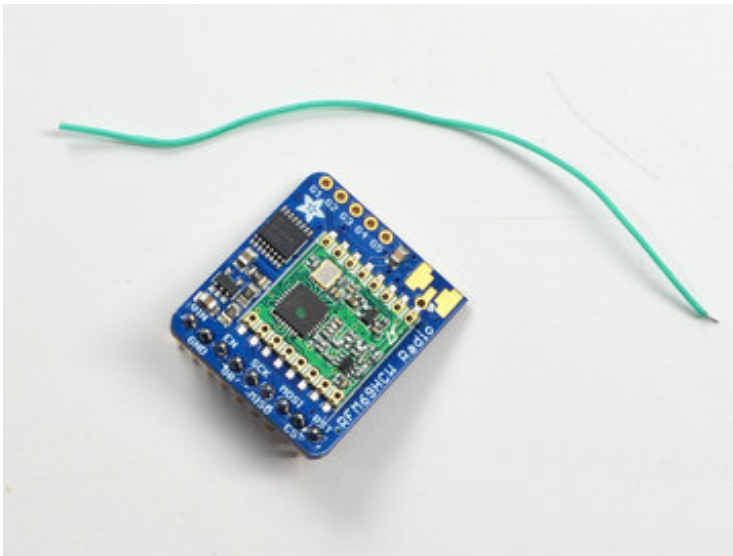
You're done! Check your solder joints visually and continue onto the next steps

Antenna Options

These radio breakouts do not have a built-in antenna. Instead, you have three options for attaching an antenna. For most low cost radio nodes, a wire works great. If you need to put the radio into an enclosure, soldering in uFL and using a uFL to SMA adapter will let you attach an external antenna. You can also solder an SMA edge-mount connector directly

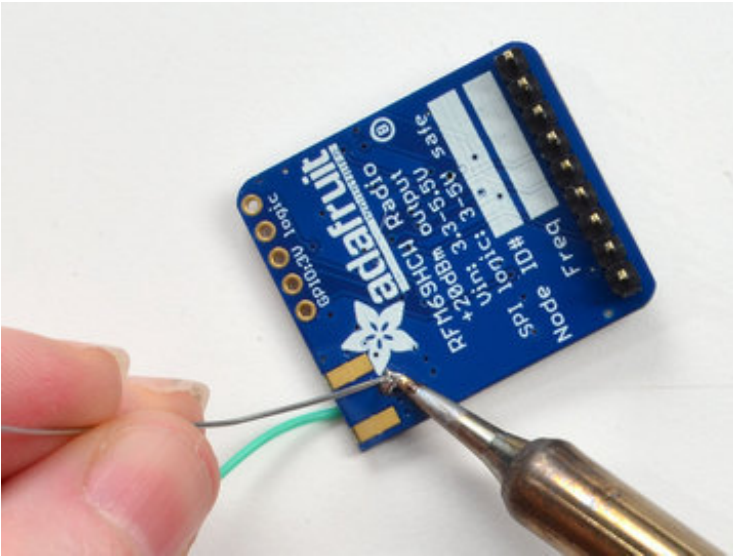
Wire Antenna

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.

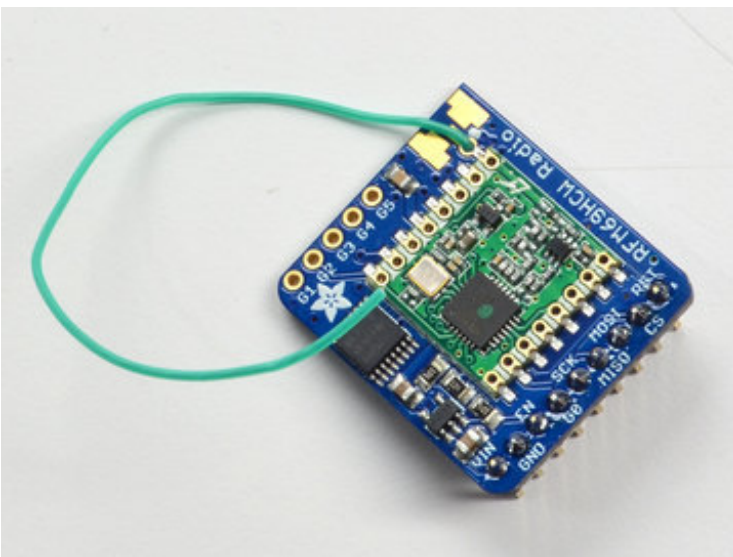


Cut a stranded or solid core wire the the proper length for the module/frequency

- **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.



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

uFL Connector

If you want an external antenna that is a few inches away from the radio, 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) (<http://adafru.it/1661>)



uFL SMT Antenna Connector

PRODUCT ID: 1661

uFL connectors are very small surface-mount parts used when an external RF antenna is desired but a big bulky SMA connector takes up too much space. We use this part on our GPS and WiFi...

<http://adafru.it/wCs>

\$0.75

IN STOCK

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



SMA to uFL/u.FL/IPX/IPEX RF Adapter Cable

PRODUCT ID: 851

This RF adapter cable is super handy for anyone doing RF work. Often times, small electronics save space by having a pick-and-placeable u.FL connector (also called uFL, IPEX, IPAX, IPX,...

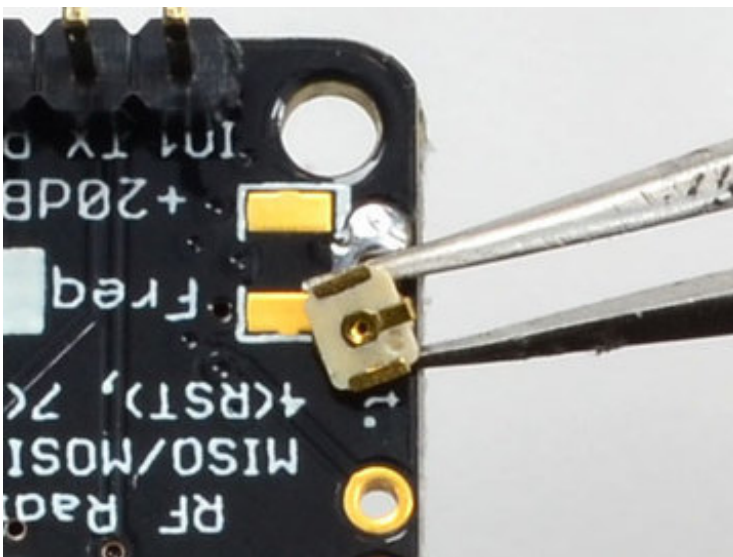
<http://adafru.it/wfx>

\$3.95

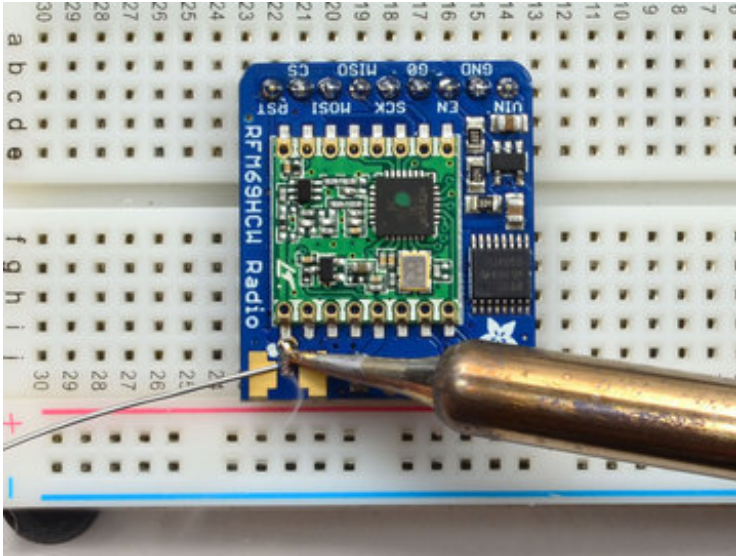
OUT OF STOCK

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

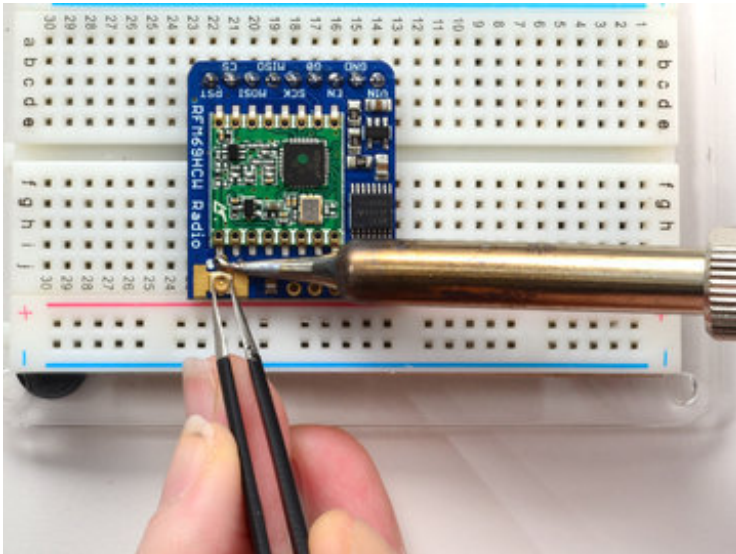
uFL connectors are rated for 30 connection cycles, but be careful when connecting/disconnecting to not rip the pads off the PCB. Once a uFL/SMA adapter is connected, use strain relief!



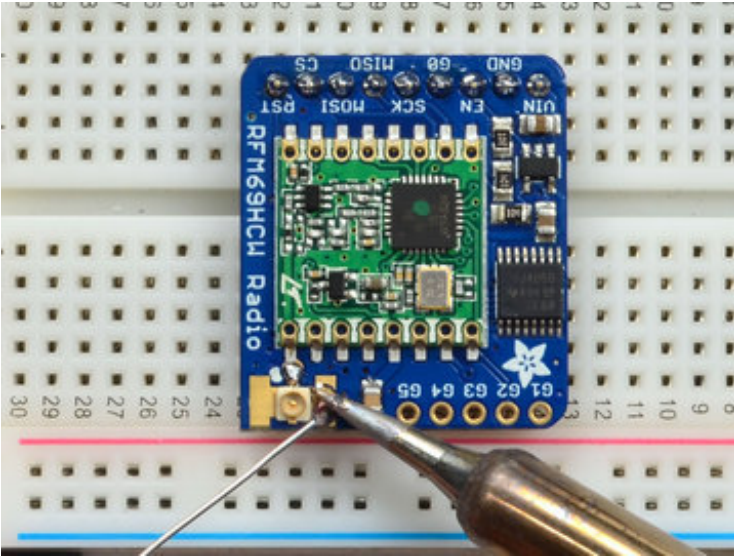
Check the bottom of the uFL connector, note that there's two large side pads (ground) and a little inlet pad. The other small pad is not used!



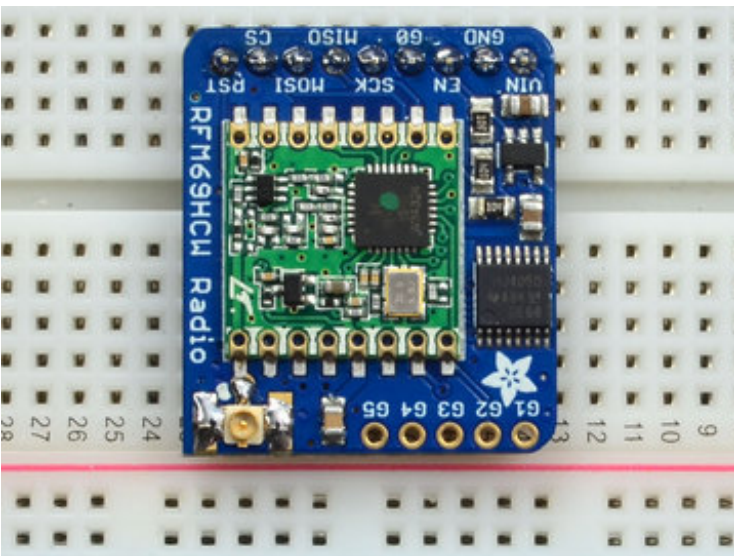
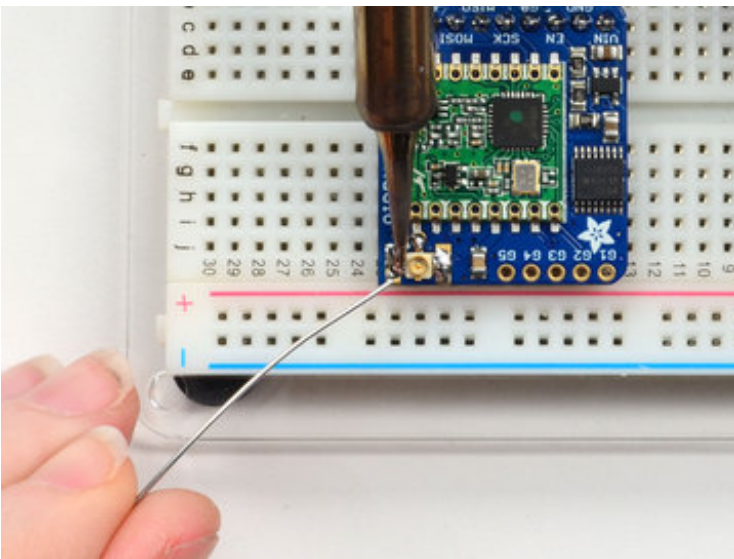
Put down a touch of solder on the signal pad



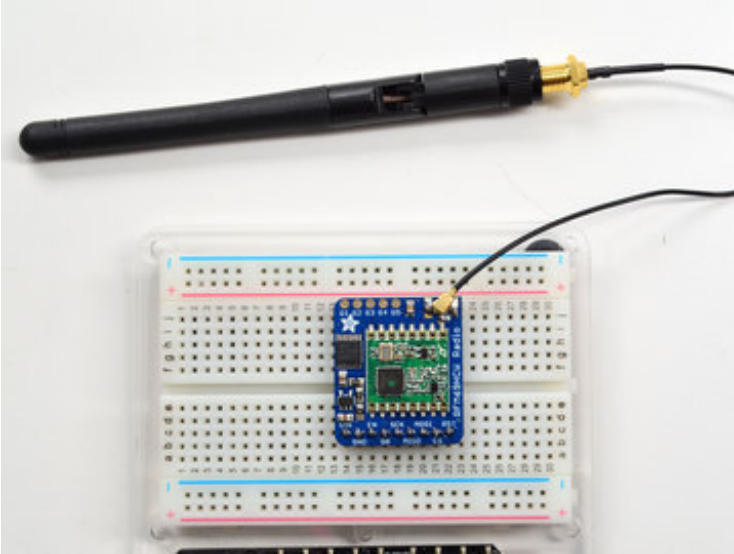
Solder in the first pad while holding the uFL steady



Solder in the two side pads, they are used for signal and mechanical connectivity so make sure there's plenty of solder



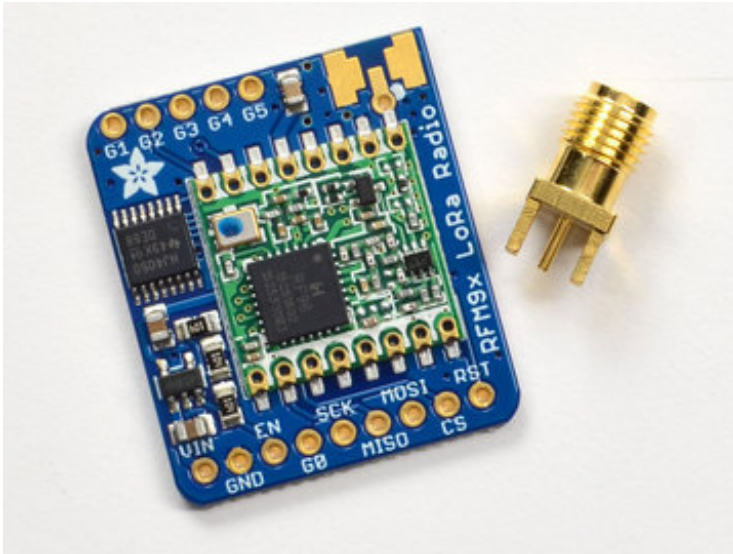
Once done, check your work visually



• Once done attach your uFL adapter and antenna!

SMA Edge-Mount Connector

OK so

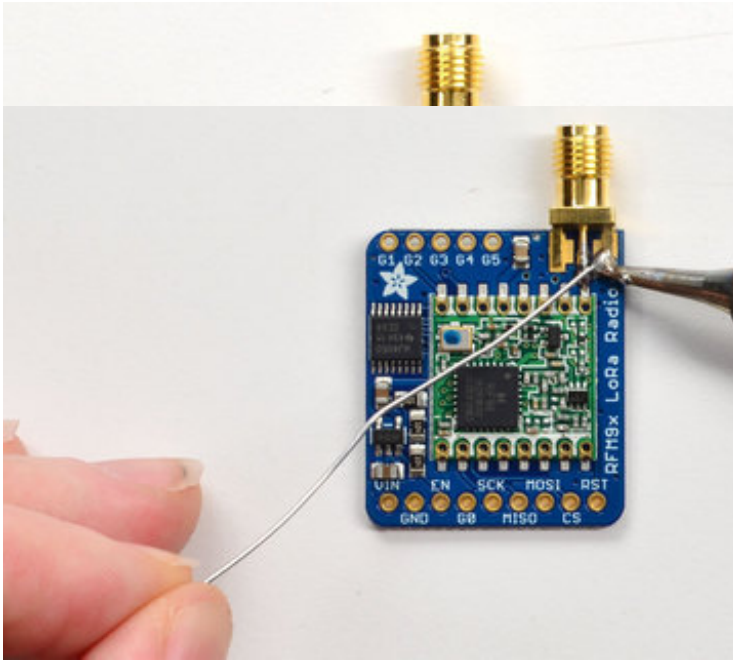


You'll need an SMA (or, if you need RP-SMA for some reason) Edge-Mount connector with 1.6mm spacing

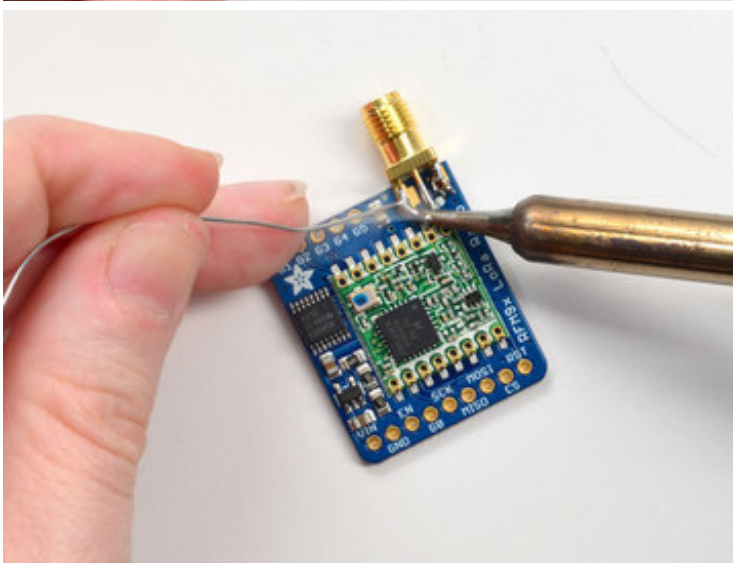


The SMA connector 'slides on' the top of the PCB

Once lined up, solder the center contact first

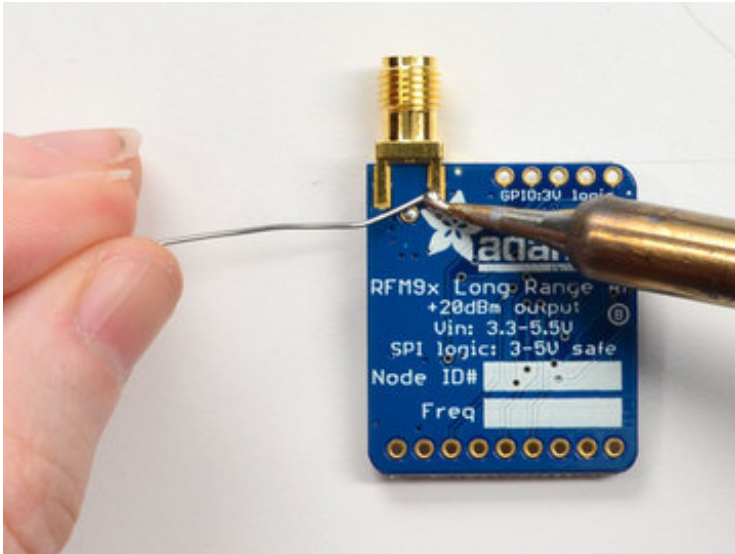


-
-

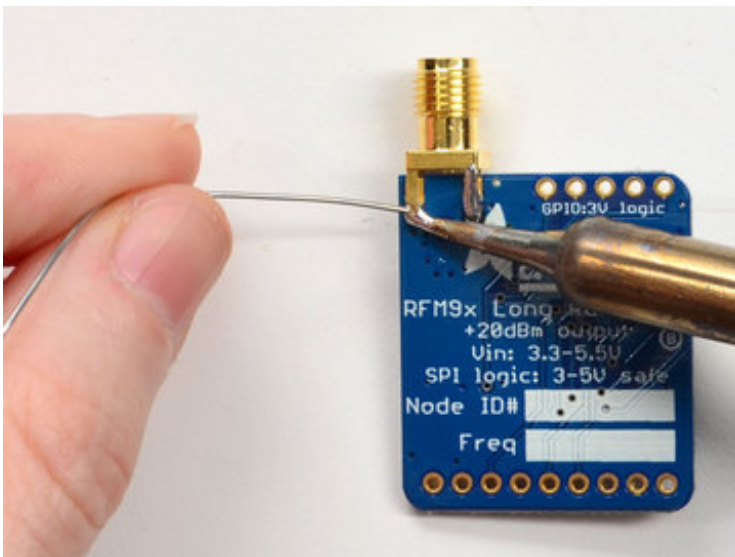


-

Solder in the two side ground pads. Note you will need a lot of heat for this, because the connector is an excellent heat sink and its got a huge ground plane



•



•

Flip over and also do the other side ground/mechanical contacts



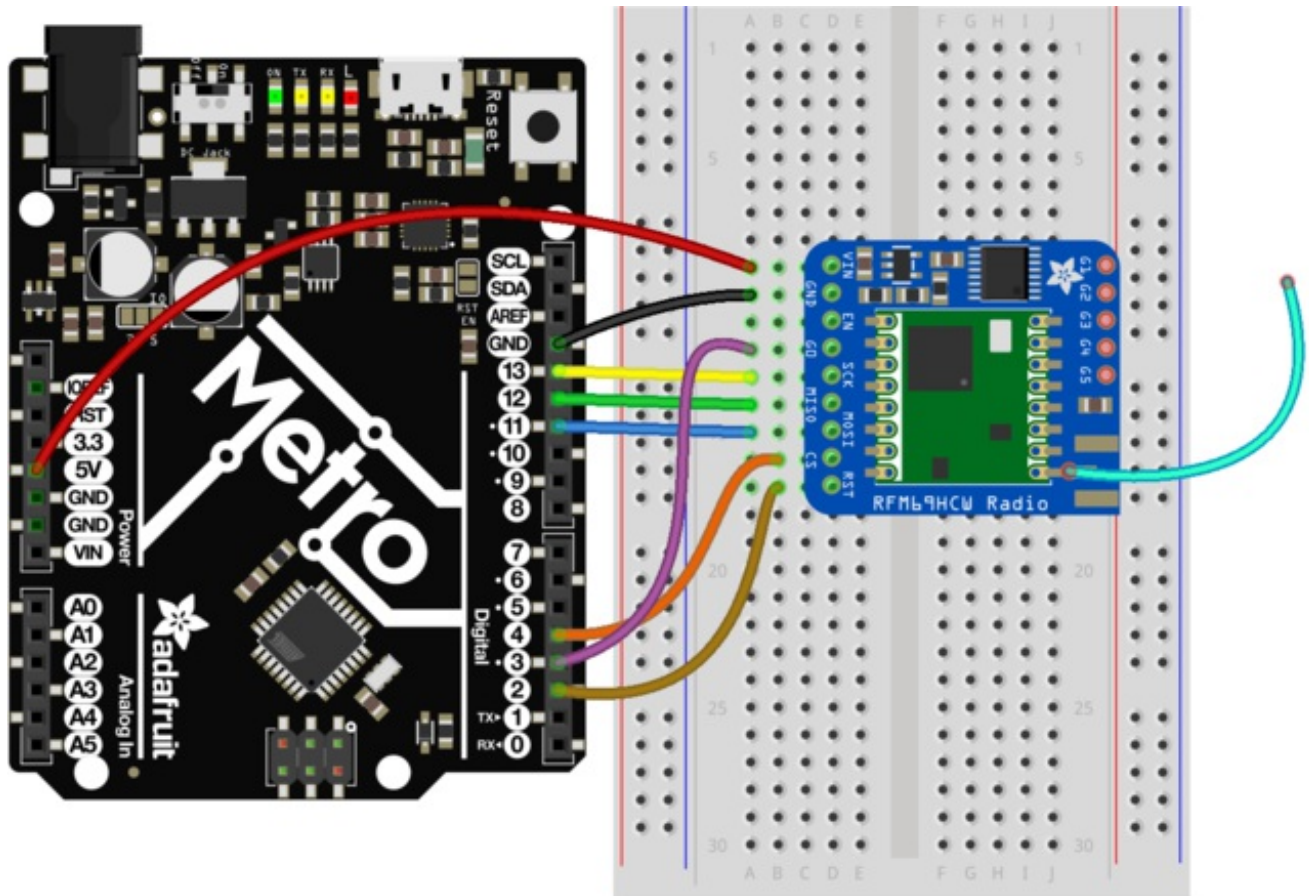
-

Attach on your antenna, you're done!



-

Wiring



fritzing

fm69.fzz

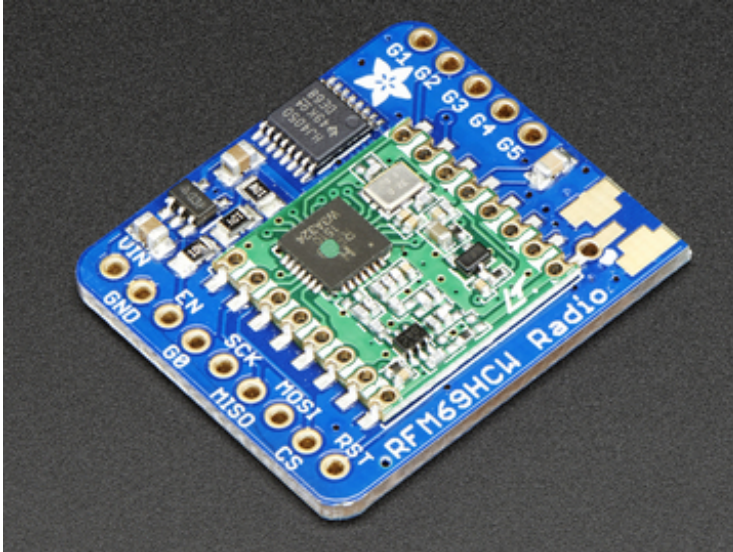
<http://adafru.it/vhb>

Wiring up the radio in SPI mode is pretty easy as there's not that many pins! The library requires hardware SPI and does not have software SPI support so you must use the hardware SPI port! Start by connecting the power pins

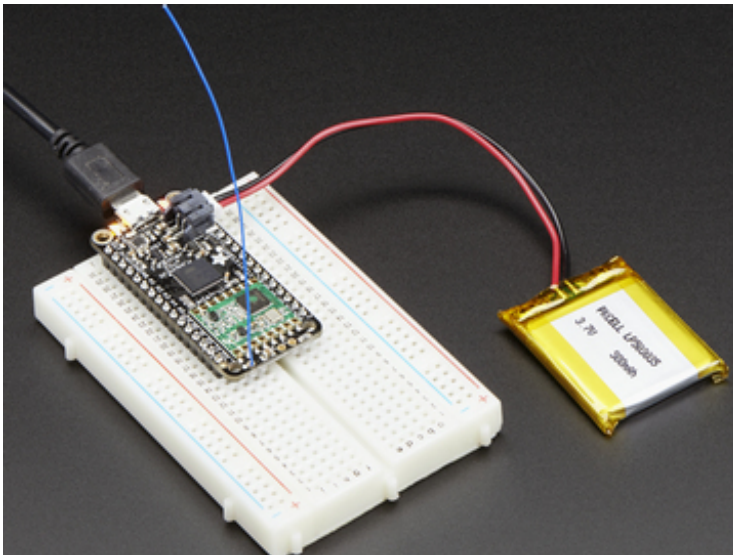
- **Vin** connects to the Arduino **5V** pin. If you're using a 3.3V Arduino, connect to **3.3V**
- **GND** connects to Arduino ground
- **CLK** connects to SPI clock. On Arduino Uno/Duemilanove/328-based, that's **Digital 13**. On Mega's, it's **Digital 52** and on Leonardo/Due it's **ICSP-3** ([See SPI Connections for more details \(http://adafru.it/d5h\)](http://adafru.it/d5h))
- **MISO** connects to SPI MISO. On Arduino Uno/Duemilanove/328-based, that's **Digital 12**. On Mega's, it's **Digital 50** and on Leonardo/Due it's **ICSP-1** ([See SPI Connections for more details \(http://adafru.it/d5h\)](http://adafru.it/d5h))

- **MOSI** connects to SPI MOSI. On Arduino Uno/Duemilanove/328-based, that's **Digital 11**. On Mega's, it's **Digital 51** and on Leonardo/Due it's **ICSP-4** ([See SPI Connections for more details \(http://adafru.it/d5h\)](http://adafru.it/d5h))
- **CS** connects to our SPI Chip Select pin. We'll be using **Digital 4** but you can later change this to any pin
- **RST** connects to our radio reset pin. We'll be using **Digital 2** but you can later change this pin too.
- **G0 (IRQ)** connects to an interrupt-capable pin. We'll be using **Digital 3** but you can later change this pin too. However, ***it must connect a hardware Interrupt pin*** Not all pins can do this! Check the board documentation for which pins are hardware interrupts, you'll also need the hardware interrupt number. For example, on UNO digital 3 is interrupt #1

Using the RFM69 Radio



•



•

This page is shared between the RFM69 breakout and the all-in-one Feather RFM69's. The example code and overall functionality is the same, only the pinouts used may differ! Just make sure the example code is using the pins you have wired up.

Before beginning make sure you have your Arduino or Feather working smoothly, it will make this part a lot easier. Once you have the basic 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 adjustable 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 RFM9x LoRa radio.

"Raw" vs Packetized

The SX1231 can be used in a 'raw rx/tx' mode where it just modulates incoming bits from pin #2 and sends them on the radio, however there's no error correction or addressing so we won't be covering that technique.

Instead, 99% of cases are best off using packetized mode. This means you can set up a recipient for your data, error correction so you can be sure the whole data set was transmitted correctly, automatic re-transmit retries and return-receipt when the packet was delivered. Basically, you get the transparency of a data pipe without the annoyances of radio transmission unreliability

Arduino Libraries

These radios have really great libraries already written, so rather than coming up with a new standard we suggest using existing libraries such as [LowPowerLab's RFM69 Library](http://adafru.it/mCz) (<http://adafru.it/mCz>) and [AirSpayce's Radiohead library](http://adafru.it/mCA) (<http://adafru.it/mCA>) which also supports a vast number of other radios

These are really great Arduino Libraries, so please support both companies in thanks for their efforts!

We recommend using the **Radiohead library** - it is very cross-platform friendly and used a lot in the community!

RadioHead Library example

To begin talking to the radio, you will need to [download our small fork of the Radiohead from our github repository](http://adafru.it/vgE) (<http://adafru.it/vgE>). You can do that by visiting the github repo and manually downloading or, easier, just click this button to download the zip

[Download RadioHead Library](http://adafru.it/vgF)
<http://adafru.it/vgF>

Rename the uncompressed folder **RadioHead** and check that the **RadioHead** folder contains files like **RH_RFM69.cpp** and **RH_RFM69.h** (and many others!)

Place the **RadioHead** library folder in your *arduinofolder/libraries/* folder. You may need to create the **libraries** subfolder if it's 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> (<http://adafru.it/aYM>)

Basic RX & TX example

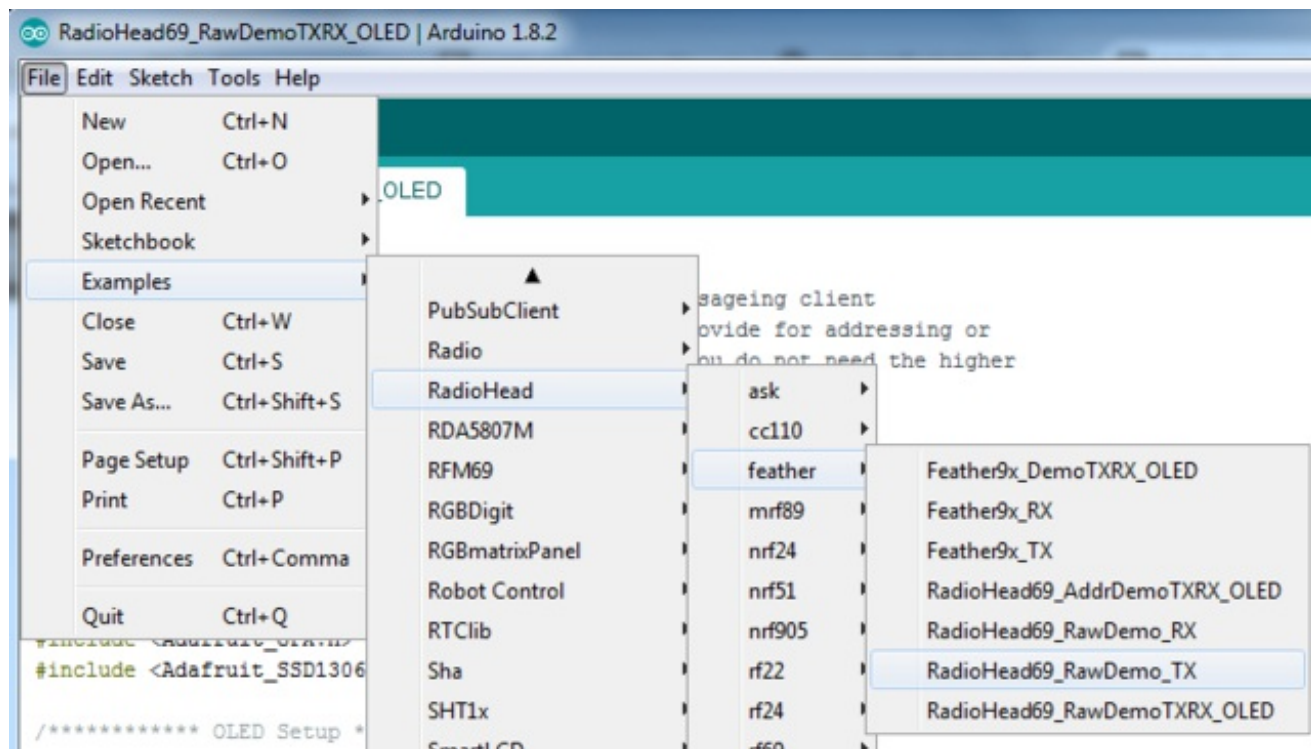
Lets get a basic demo going, where one radio transmits and the other receives. We'll start by setting up the transmitter

Basic Transmitter example code

This code will send a small packet of data once a second to another RFM69 radio, without any addressing.

Open up the example **RadioHead -> feather -> RadioHead69_RawDemo_TX**

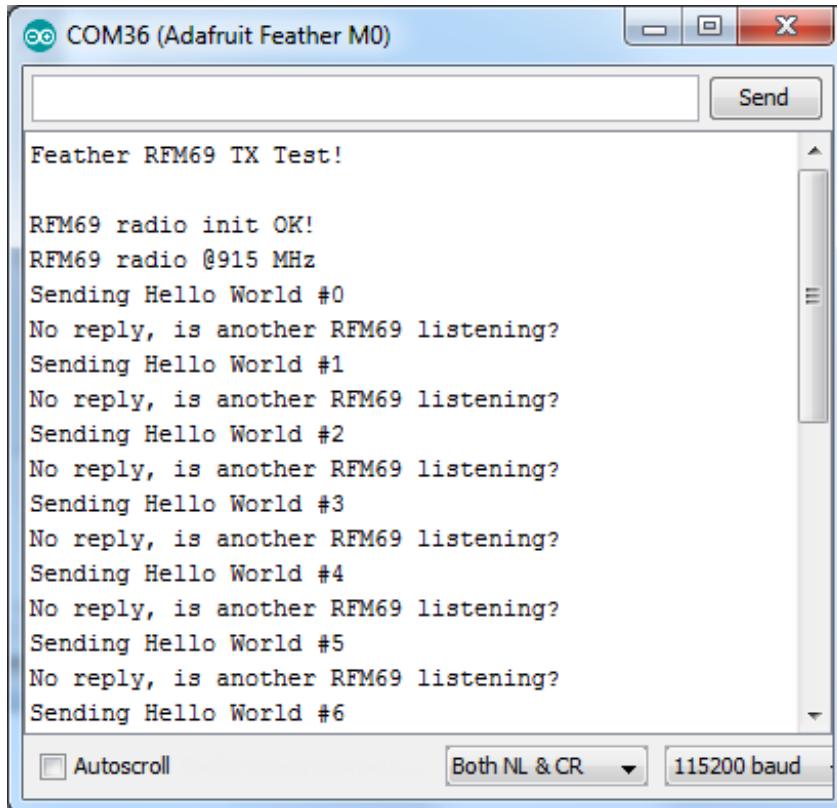
Load this code into your Transmitter Arduino or Feather!



Before uploading, check for the `#define FREQUENCY RF69_915MHZ` line and comment that out (and uncomment the line above) to match the frequency of the hardware you're using

These examples are optimized for the Feather 32u4/M0. If you're using different wiring, uncomment/comment/edit the sections defining the pins depending on which chipset and wiring you are using! The pins used will vary depending on your setup!

Once uploaded you should see the following on the serial console



```
COM36 (Adafruit Feather M0)
Feather RFM69 TX Test!
RFM69 radio init OK!
RFM69 radio @915 MHz
Sending Hello World #0
No reply, is another RFM69 listening?
Sending Hello World #1
No reply, is another RFM69 listening?
Sending Hello World #2
No reply, is another RFM69 listening?
Sending Hello World #3
No reply, is another RFM69 listening?
Sending Hello World #4
No reply, is another RFM69 listening?
Sending Hello World #5
No reply, is another RFM69 listening?
Sending Hello World #6
```

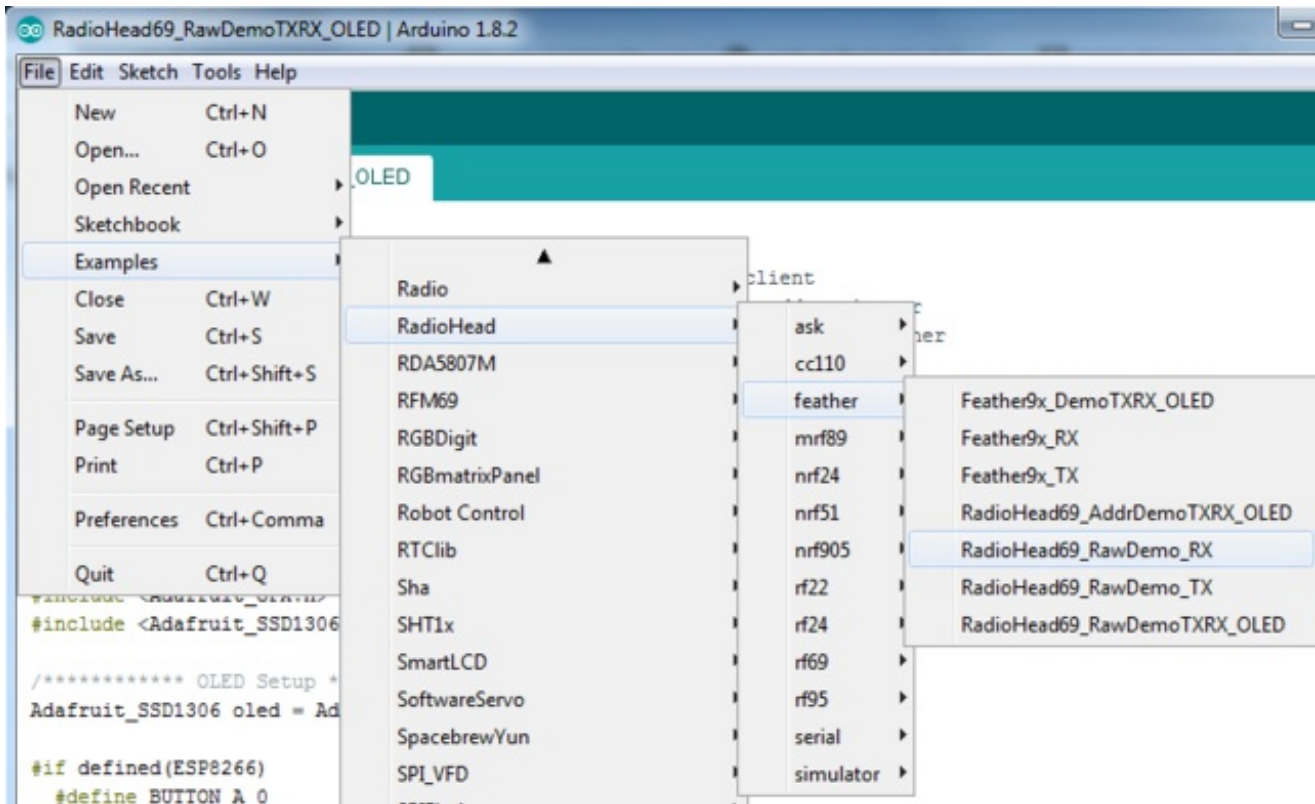
Now open up another instance of the Arduino IDE - this is so you can see the serial console output from the TX device while you set up the RX device.

Basic receiver example code

This code will receive and reply with a small packet of data.

Open up the example **RadioHead -> feather -> RadioHead69_RawDemo_RX**

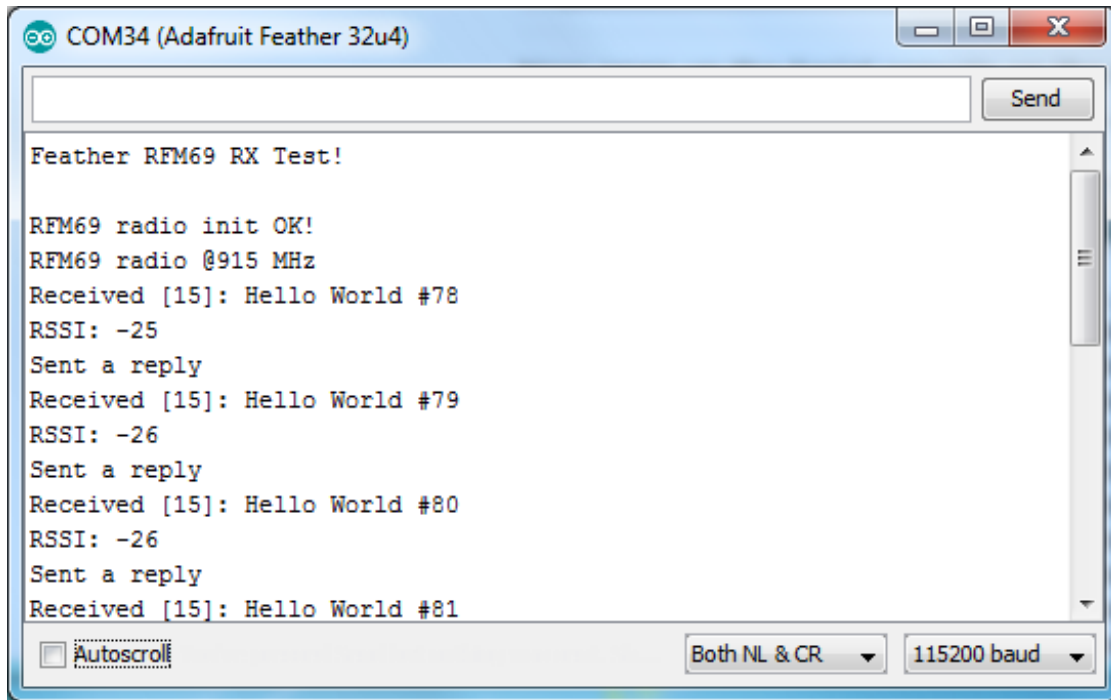
Load this code into your **Receiver** Arduino/Feather!



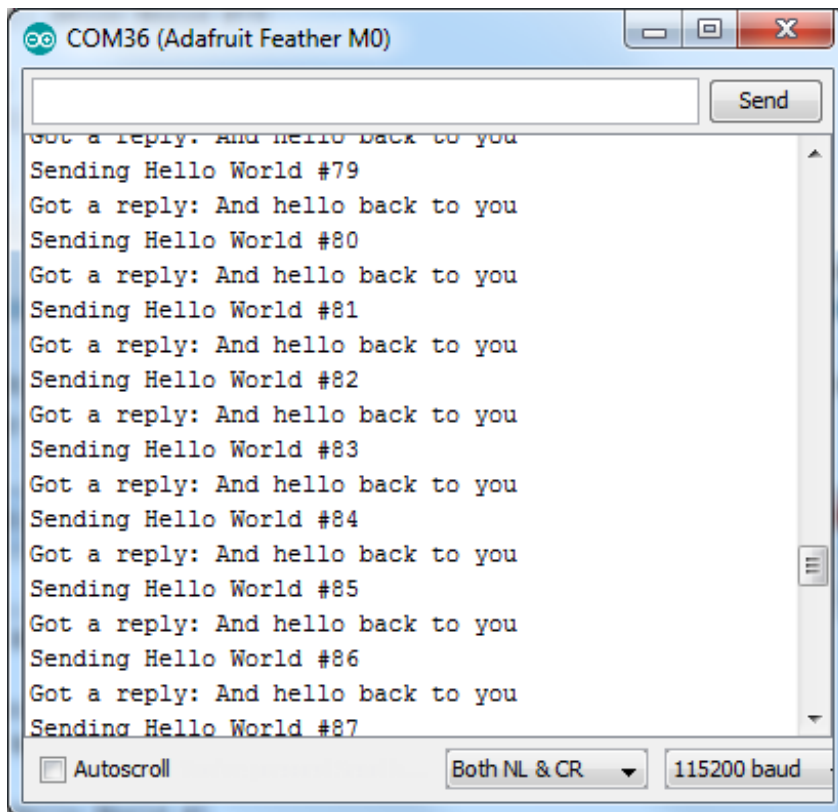
Before uploading, check for the `#define FREQUENCY RF69_915MHZ` line and comment that out (and uncomment the line above) to match the frequency of the hardware you're using

These examples are optimized for the Feather 32u4/M0. If you're using different wiring, uncomment/comment/edit the sections defining the pins depending on which chipset and wiring you are using! The pins used will vary depending on your setup!

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



And, on the transmitter side, it is now printing **Got Reply** after each transmission 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 network

Radio Freq. Config

Each radio has a frequency that is configurable in software. You can actually tune outside the recommended frequency, but the range won't be good. 900 MHz can be tuned from about 850-950MHz with good performance. 433 MHz radios can be tuned from 400-460 MHz or so.

```
// Change to 434.0 or other frequency, must match RX's freq!  
#define RF69_FREQ 915.0
```

For all radios they will need to be on the same frequency. If you have a 433MHz radio you will want to stick to 433. If you have a 900 Mhz radio, go with 868 or 915MHz, just make sure all radios are on the same frequency

Configuring Radio Pinout

At the top of the sketch you can also set the pinout. The radios will use hardware SPI, but you can select any pins for **RFM69_CS** (an output), **RFM_IRQ** (an input) and **RFM_RST** (an output). RFM_RST is manually used to reset the radio at the beginning of the sketch. **RFM_IRQ** must be an interrupt-capable pin. Check your board to determine which pins you can use!

Also, an LED is defined.

For example, here is the Feather 32u4 pinout

```
#if defined (__AVR_ATmega32U4__) // Feather 32u4 w/Radio  
#define RFM69_CS 8  
#define RFM69_INT 7  
#define RFM69_RST 4  
#define LED 13  
#endif
```

If you're using a Feather M0, the pinout is slightly different:

```
#if defined(ARDUINO_SAMD_FEATHER_M0) // Feather M0 w/Radio  
#define RFM69_CS 8  
#define RFM69_INT 3  
#define RFM69_RST 4  
#define LED 13  
#endif
```

If you're using an Arduino UNO or compatible, we recommend:

```
#if defined (__AVR_ATmega328P__) // UNO or Feather 328P w/wing  
#define RFM69_INT 3 //
```

```
#define RFM69_CS    4 //
#define RFM69_RST  2 // "A"
#define LED        13
#endif
```

If you're using a FeatherWing or different setup, you'll have to set up the `#define` statements to match your wiring

You can then instantiate the radio object with our custom pin numbers. Note that the IRQ is defined by the IRQ pin not number (sometimes they differ).

```
// Singleton instance of the radio driver
RH_RF69 rf69(RFM69_CS, RFM69_INT);
```

Setup

We begin by setting up the serial console and hard-resetting the RFM69

```
void setup()
{
  Serial.begin(115200);
  //while (!Serial) { delay(1); } // wait until serial console is open, remove if not tethered to computer

  pinMode(LED, OUTPUT);
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, LOW);

  Serial.println("Feather RFM69 RX Test!");
  Serial.println();

  // manual reset
  digitalWrite(RFM69_RST, HIGH);
  delay(10);
  digitalWrite(RFM69_RST, LOW);
  delay(10);
```

If you are using a board with 'native USB' make sure the **while (!Serial)** line is commented out if you are not tethering to a computer, as it will cause the microcontroller to halt until a USB connection is made!

Initializing Radio

Once initialized, you can set up the frequency, transmission power, radio type and encryption key.

For the **frequency**, we set it already at the top of the sketch

For **transmission power** you can select from 14 to 20 dBi. Lower numbers use less power, but have less range. The second argument to the function is whether it is an HCW type radio, with extra amplifier. This should *a/ways* be set to **true**!

Finally, if you are **encrypting** data transmission, set up the encryption key

```
if (!rf69.init()) {
  Serial.println("RFM69 radio init failed");
  while (1);
}
Serial.println("RFM69 radio init OK!");

// Defaults after init are 434.0MHz, modulation GFSK_Rb250Fd250, +13dbM (for low power module)
// No encryption
if (!rf69.setFrequency(RF69_FREQ)) {
  Serial.println("setFrequency failed");
}

// If you are using a high power RF69 eg RFM69HW, you *must* set a Tx power with the
// ishighpowermodule flag set like this:
rf69.setTxPower(20, true); // range from 14-20 for power, 2nd arg must be true for 69HCW

// The encryption key has to be the same as the one in the server
uint8_t key[] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
                 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
rf69.setEncryptionKey(key);
```

Basic 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, then check for a reply

```
void loop() {
  delay(1000); // Wait 1 second between transmits, could also 'sleep' here!

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

  // Send a message!
  rf69.send((uint8_t *)radiopacket, strlen(radiopacket));
  rf69.waitPacketSent();

  // Now wait for a reply
  uint8_t buf[RH_RF69_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);

  if (rf69.waitAvailableTimeout(500)) {
    // Should be a reply message for us now
```

```

if (rf69.recv(buf, &len)) {
  Serial.print("Got a reply: ");
  Serial.println((char*)buf);
  Blink(LED, 50, 3); //blink LED 3 times, 50ms between blinks
} else {
  Serial.println("Receive failed");
}
} else {
  Serial.println("No reply, is another RFM69 listening?");
}
}

```

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() waitPacketSent() to wait until is is done transmitting.

It will then wait up to 500 milliseconds for a reply from the receiver with waitAvailableTimeout(500). If there is a reply, it will print it out. If not, it will complain nothing was received. Either way the transmitter will continue the loop and sleep for a second until the next TX.

Basic Receiver Code

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

```

void loop() {
if (rf69.available()) {
  // Should be a message for us now
  uint8_t buf[RH_RF69_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);
  if (rf69.recv(buf, &len)) {
    if (!len) return;
    buf[len] = 0;
    Serial.print("Received [");
    Serial.print(len);
    Serial.print("]: ");
    Serial.println((char*)buf);
    Serial.print("RSSI: ");
    Serial.println(rf69.lastRssi(), DEC);

    if (strstr((char *)buf, "Hello World")) {
      // Send a reply!
      uint8_t data[] = "And hello back to you";
      rf69.send(data, sizeof(data));
      rf69.waitPacketSent();
      Serial.println("Sent a reply");
      Blink(LED, 40, 3); //blink LED 3 times, 40ms between blinks
    }
  } else {

```



```

Serial.println("Receive failed");
}
}
}

```

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 the proper encryption has been received. If so, the receiver prints it out.

It also prints out the RSSI which is the receiver signal strength indicator. This number will range from about -15 to -80. The larger the number (-15 being the highest you'll likely see) the stronger the signal.

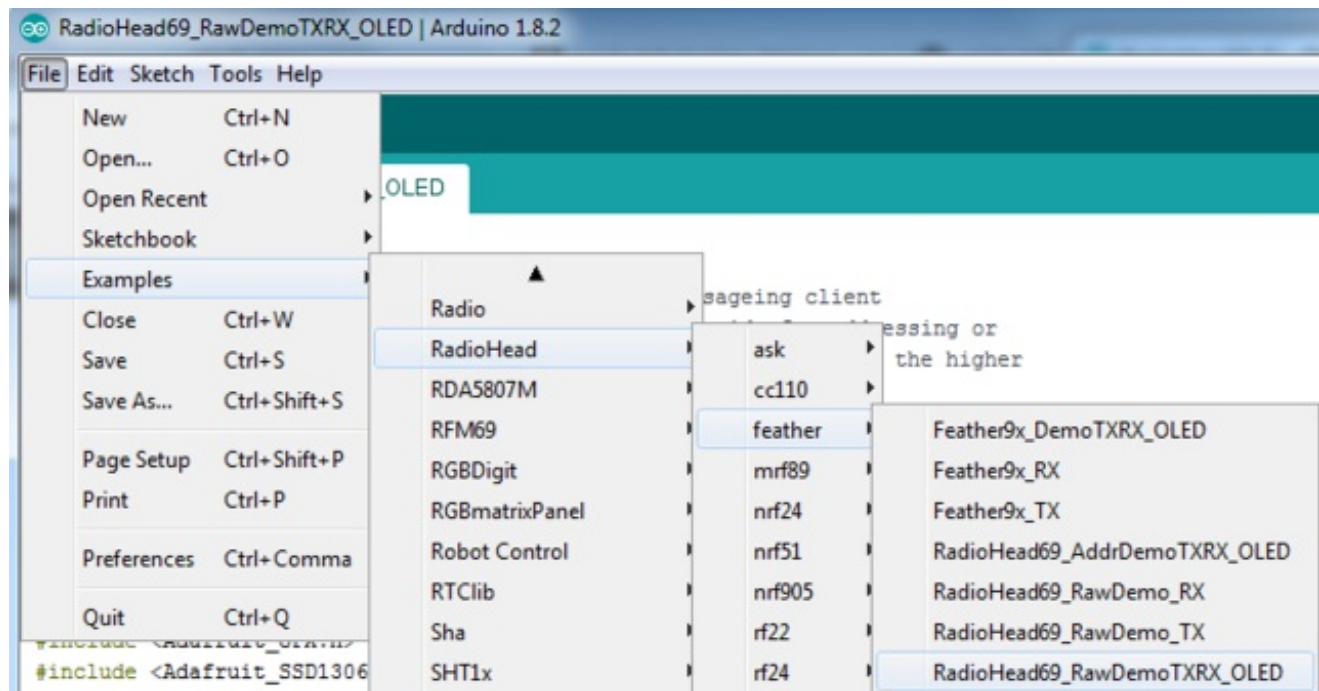
If the data contains the text "Hello World" it will also reply to the packet.

Once done it will continue waiting for a new packet

Basic Receiver/Transmitter Demo w/OLED

OK once you have that going you can try this example,

RadioHead69_RawDemoTXRX_OLED. We're using the Feather with an OLED wing but in theory you can run the code without the OLED and connect three buttons to GPIO #9, 6, and 5 on the Feathers. Upload the same code to each Feather. When you press buttons on one Feather they will be printed out on the other one, and vice versa. Very handy for testing bi-directional communication!



This demo code shows how you can listen for packets and also check for button presses (or sensor data or whatever you like) and send them back and forth between the two radios!

Addressed RX and TX Demo

OK so the basic demo is well and good but you have to do a lot of *management* of the connection to make sure packets were received. Instead of manually sending acknowledgements, you can have the RFM69 and library do it for you! Thus the **Reliable Datagram** part of the **RadioHead** library.

Load up the **RadioHead69_AddrDemo_RX** and **RadioHead69_AddrDemo_TX** sketches to each of your boards

Don't forget to check the frequency set in the example, and that the pinouts match your wiring!!!

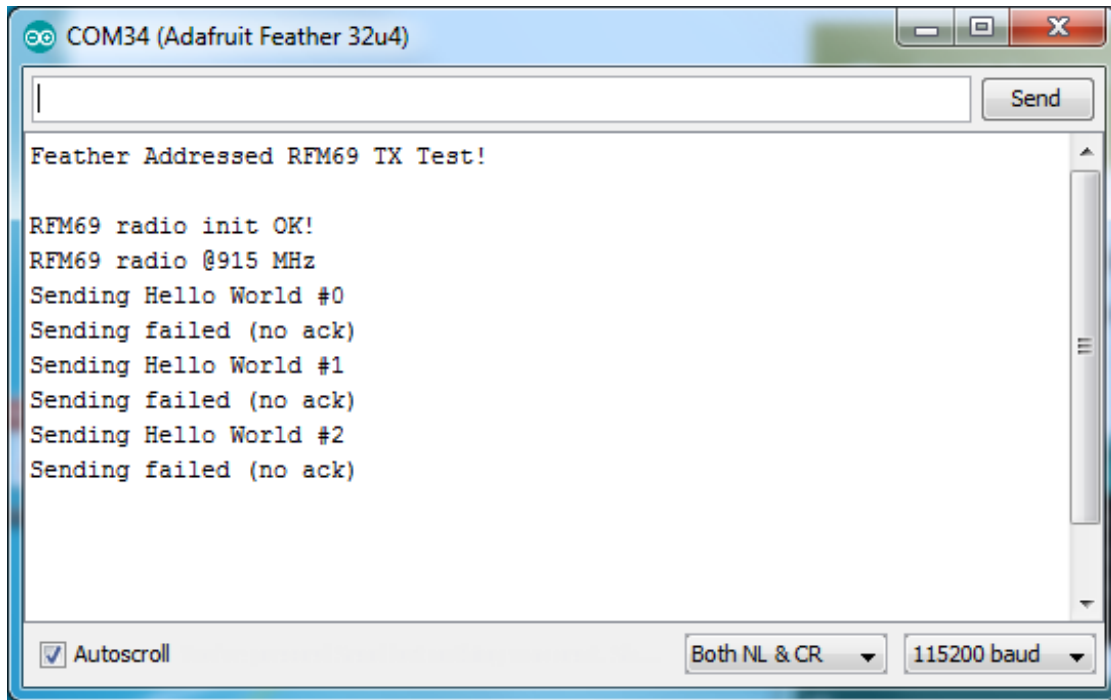
This example lets you have many 'client' RFM69's all sending data to one 'server'

Each client can have its own address set, as well as the server address. See this code at the beginning:

```
// Where to send packets to!  
#define DEST_ADDRESS 1  
// change addresses for each client board, any number :)  
#define MY_ADDRESS 2
```

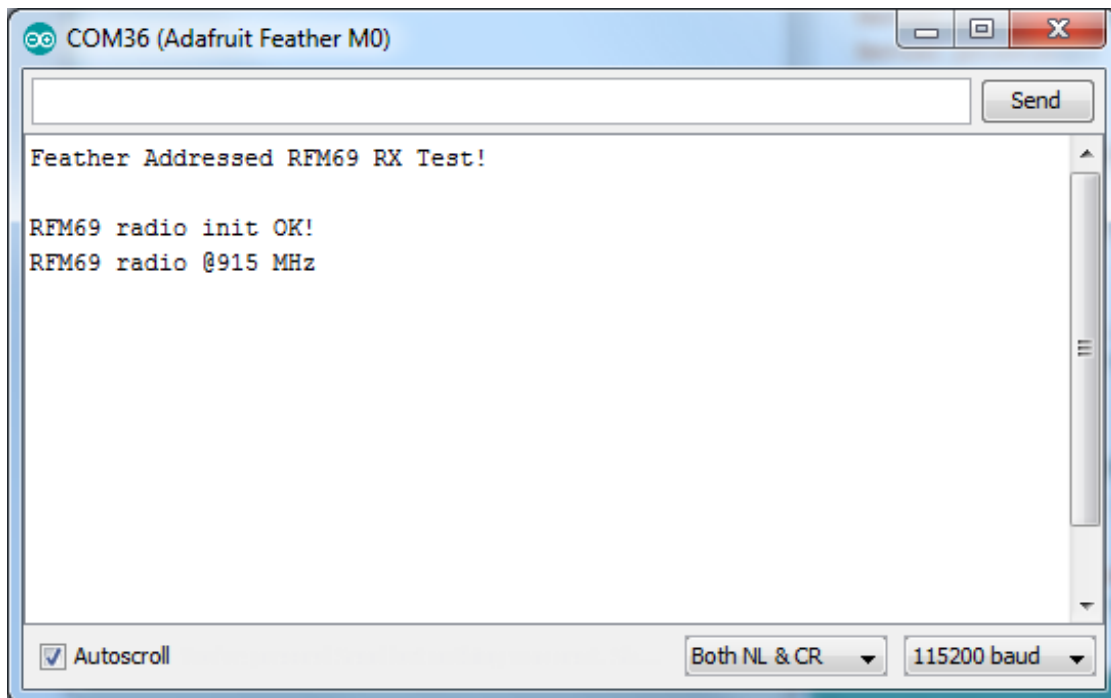
For each client, have a unique **MY_ADDRESS**. Then pick one server that will be address #1

Once you upload the code to a client, you'll see the following in the serial console:



Because the data is being sent to address #1, but #1 is not acknowledging that data.

If you have the server running, with no clients, it will sit quietly:



Turn on the client and you'll see acknowledged packets!

```
COM34 (Adafruit Feather 32u4)
Feather Addressed RFM69 TX Test!
RFM69 radio init OK!
RFM69 radio @915 MHz
Sending Hello World #0
Got reply from #1 [RSSI :-26] : And hello back to you
Sending Hello World #1
Got reply from #1 [RSSI :-25] : And hello back to you
Sending Hello World #2
Got reply from #1 [RSSI :-26] : And hello back to you
Sending Hello World #3
Got reply from #1 [RSSI :-26] : And hello back to you
Sending Hello World #4
Got reply from #1 [RSSI :-26] : And hello back to you
```

And the server is also pretty happy

```
COM36 (Adafruit Feather M0)
Feather Addressed RFM69 RX Test!
RFM69 radio init OK!
RFM69 radio @915 MHz
Got packet from #2 [RSSI :-26] : Hello World #0
Got packet from #2 [RSSI :-31] : Hello World #1
Got packet from #2 [RSSI :-30] : Hello World #2
Got packet from #2 [RSSI :-31] : Hello World #3
Got packet from #2 [RSSI :-31] : Hello World #4
Got packet from #2 [RSSI :-30] : Hello World #5
Got packet from #2 [RSSI :-30] : Hello World #6
Got packet from #2 [RSSI :-31] : Hello World #7
Got packet from #2 [RSSI :-31] : Hello World #8
Got packet from #2 [RSSI :-30] : Hello World #9
```

The secret sauce is the addition of this new object:

```
// Class to manage message delivery and receipt, using the driver declared above
RHReliableDatagram rf69_manager(rf69, MY_ADDRESS);
```

Which as you can see, is the manager for the RFM69. **Insetup()** you'll need to init it, although you still configure the underlying rfm69 like before:

```
if (!rf69_manager.init()) {  
  Serial.println("RFM69 radio init failed");  
  while (1);  
}
```

And when transmitting, use **sendToWait** which will wait for an ack from the recipient (at DEST_ADDRESS)

```
if (rf69_manager.sendtoWait((uint8_t *)radiopacket, strlen(radiopacket), DEST_ADDRESS)) {
```

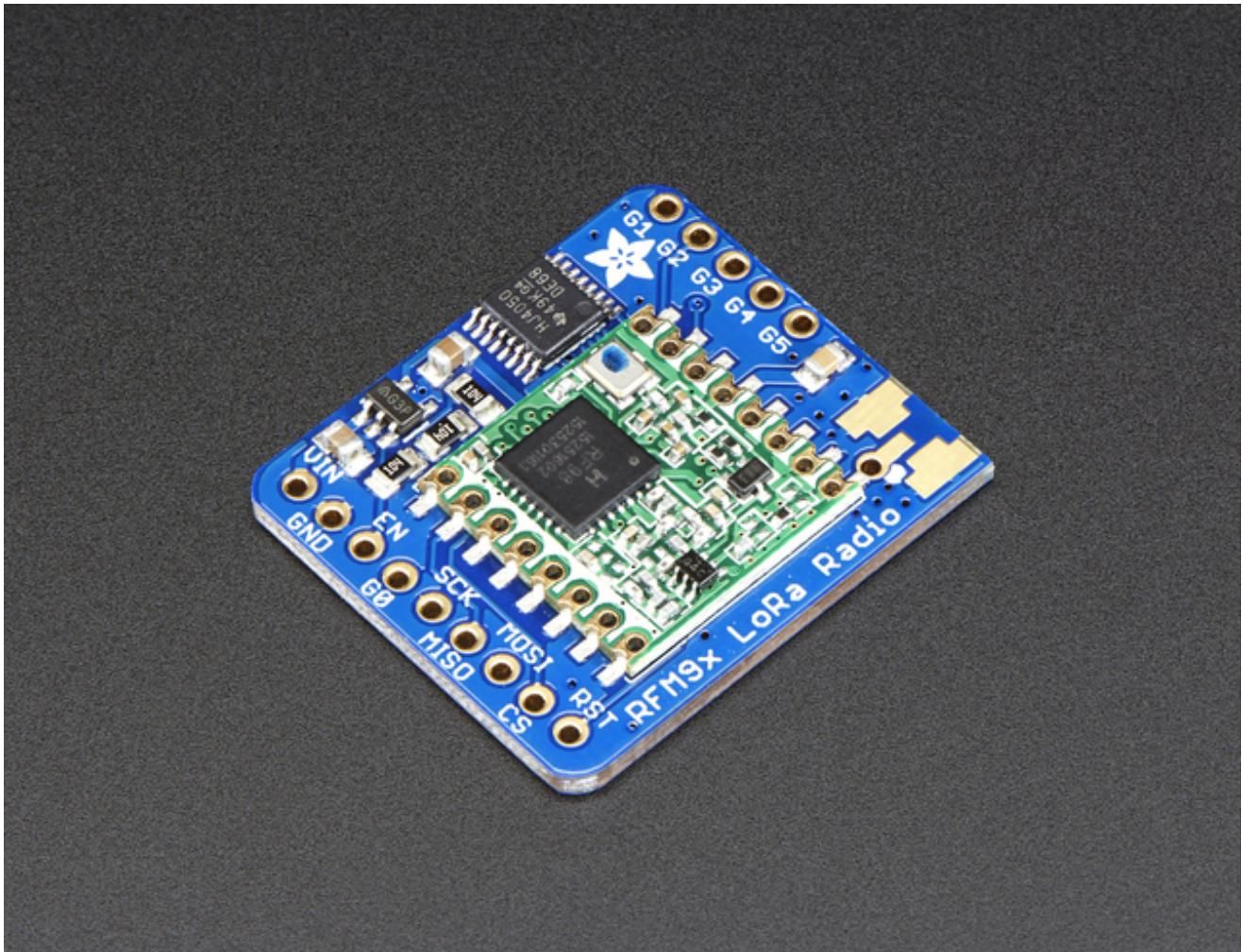
on the 'other side' use the **recvFromAck** which will receive and acknowledge a packet

```
// Wait for a message addressed to us from the client  
uint8_t len = sizeof(buf);  
uint8_t from;  
if (rf69_manager.recvfromAck(buf, &len, &from)) {
```

That function will wait forever. If you'd like to timeout while waiting for a packet, use **recvfromAckTimeout** which will wait an indicated # of milliseconds

```
if (rf69_manager.recvfromAckTimeout(buf, &len, 2000, &from))
```

RFM9X Test



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 adjustable 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 radiosto 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](http://adafru.it/mCA) (<http://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](http://adafru.it/mCA) (<http://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 1.59

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](http://adafru.it/mCA) (<http://adafru.it/mCA>)

[Download RadioHead v1.59](http://adafru.it/mHC)

<http://adafru.it/mHC>

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 **arduinodesketchfolder/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> (<http://adafru.it/aYM>)

Basic RX & TX example

Lets get a basic demo going, where one Arduino 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!

```

// LoRa 9x_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 LoRa9x_RX

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

#define RFM95_CS 10
#define RFM95_RST 9
#define RFM95_INT 2

// 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);

  while (!Serial);
  Serial.begin(9600);
  delay(100);

  Serial.println("Arduino 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");
    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()
{
  Serial.println("Sending to rf95_server");
  // 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;

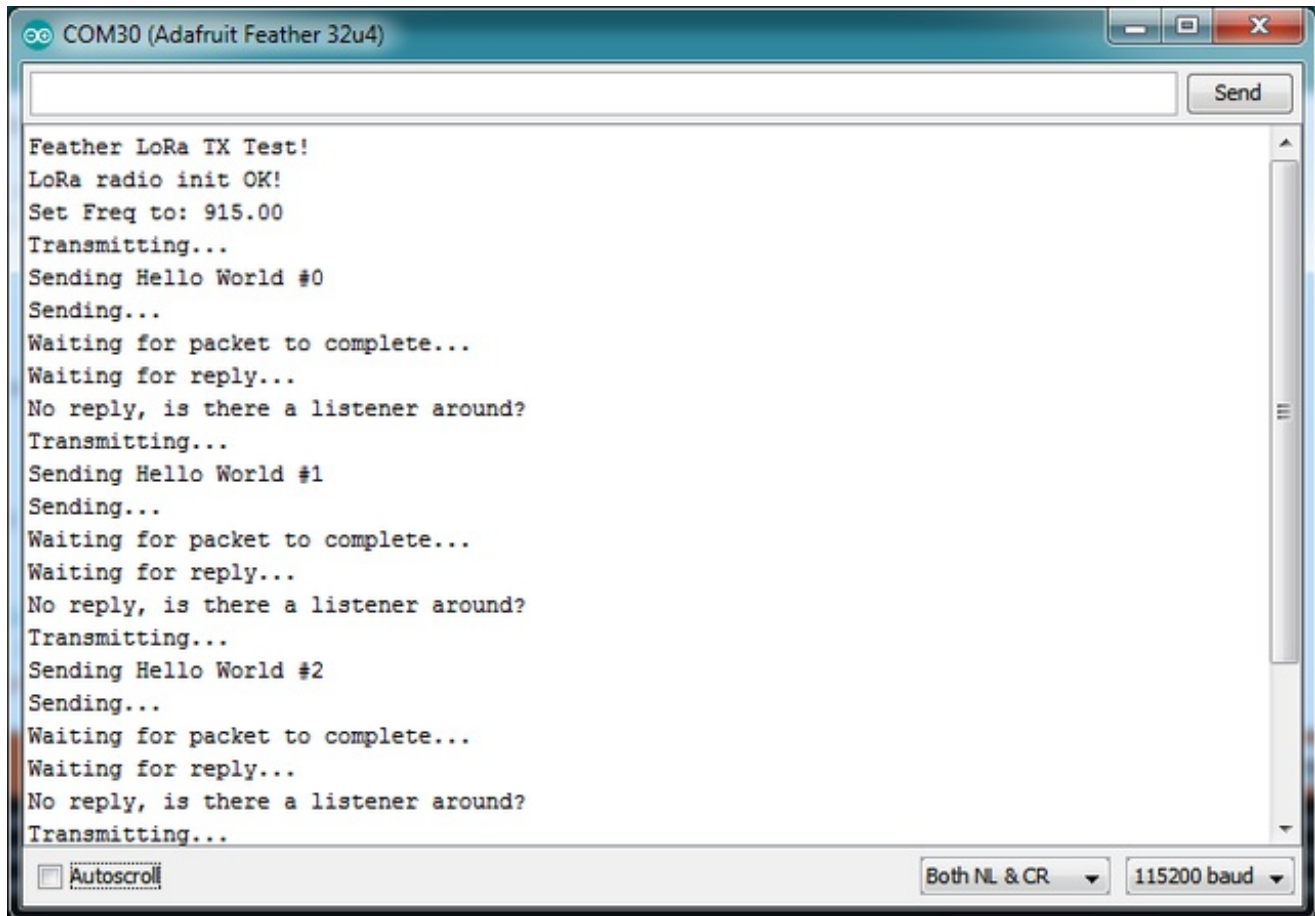
  Serial.println("Sending..."); delay(10);
  rf95.send((uint8_t *)radiopacket, 20);

  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..."); delay(10);
  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?");
  }
  delay(1000);
}

```

Once uploaded you should see the following on the serial console



```
COM30 (Adafruit Feather 32u4)
Send
Feather LoRa TX Test!
LoRa radio init OK!
Set Freq to: 915.00
Transmitting...
Sending Hello World #0
Sending...
Waiting for packet to complete...
Waiting for reply...
No reply, is there a listener around?
Transmitting...
Sending Hello World #1
Sending...
Waiting for packet to complete...
Waiting for reply...
No reply, is there a listener around?
Transmitting...
Sending Hello World #2
Sending...
Waiting for packet to complete...
Waiting for reply...
No reply, is there a listener around?
Transmitting...
Autoscroll
Both NL & CR
115200 baud
```

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

Receiver example code

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

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

```
// Arduino9x_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 Arduino9x_TX

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

```

#define RFM95_CS 10
#define RFM95_RST 9
#define RFM95_INT 2

// 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);

  while (!Serial);
  Serial.begin(9600);
  delay(100);

  Serial.println("Arduino 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");
    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);
}

```

```

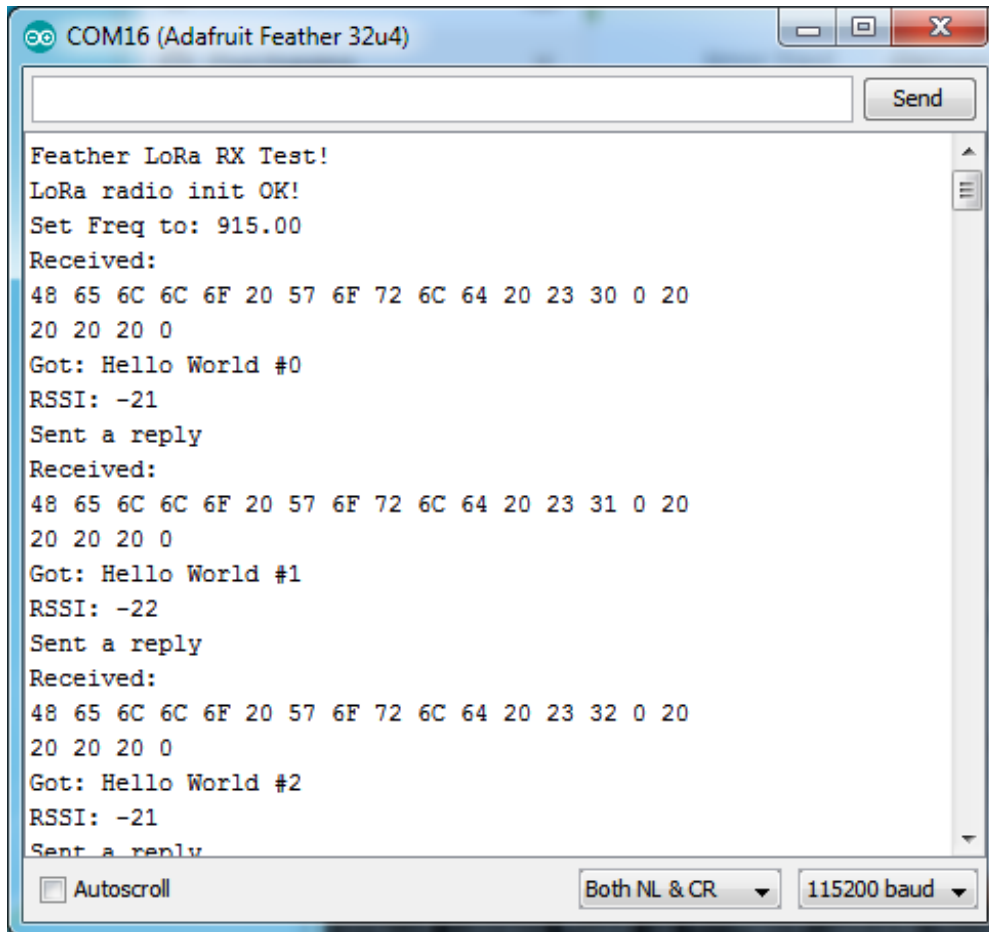
void loop()
{
  if (rf95.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);

    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");
    }
  }
}

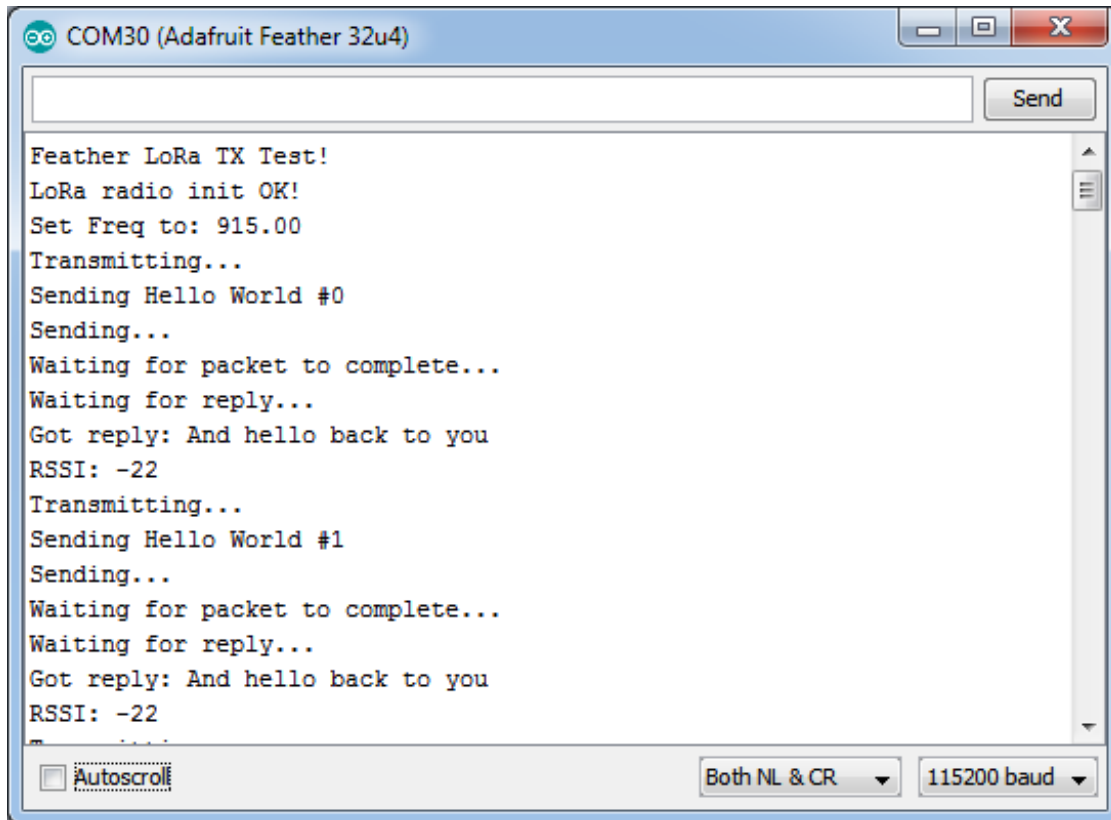
```

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 transmission. 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

Radio Pinout

This is the pinout setup - you can change around the reset and CS pins to any pin. the IRQ pin should be an interrupt pin. On an UNO this is pin #2 or pin #3. Each chipset has different interrupt pins!

```
#define RFM95_CS 10
#define RFM95_RST 9
#define RFM95_INT 2
```

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("Arduino 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 Arduino 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";
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

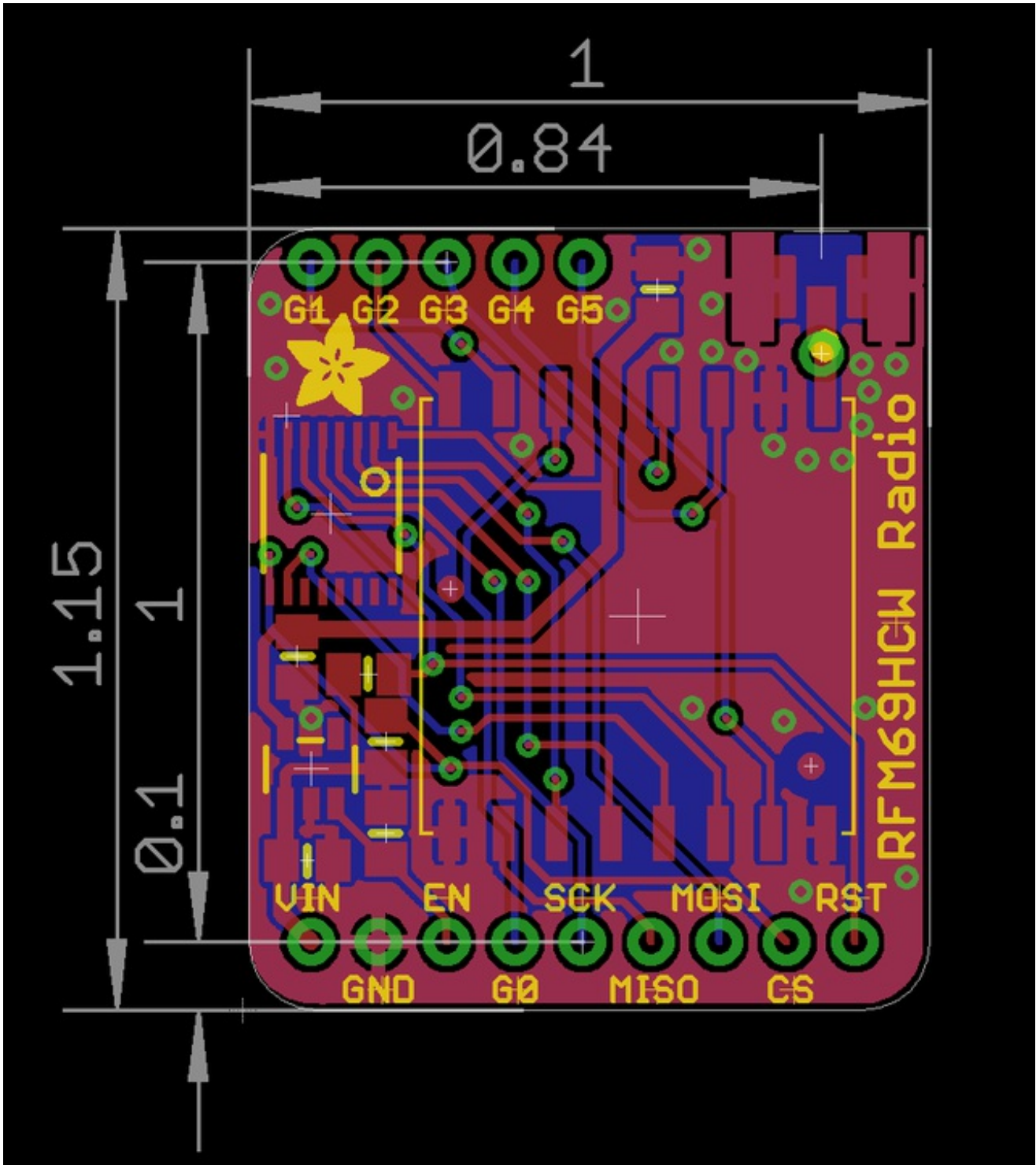
Downloads

Datasheets & Files

- [SX127x Datasheet](http://adafru.it/oBm) (<http://adafru.it/oBm>)- The RFM9X LoRa radio chip itself
- [SX1231 Datasheet](http://adafru.it/mCv) (<http://adafru.it/mCv>) - The RFM69 radio chip itself
- [RFM69HCW datasheet](http://adafru.it/mCu) (<http://adafru.it/mCu>)- [contains the SX1231 datasheet plus details about the module](http://adafru.it/mFX) (<http://adafru.it/mFX>)
- [RFM9X](http://adafru.it/mFX) (<http://adafru.it/mFX>) - The radio module, which contains the SX1272 chipset
- [FCC Test Report](http://adafru.it/r6d) (<http://adafru.it/r6d>)
- [EagleCAD PCB files on GitHub](http://adafru.it/oem) (<http://adafru.it/oem>)
- [Fritzing objects in the Adafruit Fritzing library](http://adafru.it/c7M) (<http://adafru.it/c7M>)

Schematic

RFM69 and RFM9X have the same pinout so the same schematic is used



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 advertised! 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 different 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 antenna book is recommended if you want to learn how to do the modeling and analysis \(http://adafru.it/sdN\)](http://adafru.it/sdN)

But nothing beats actual tests in your environment!