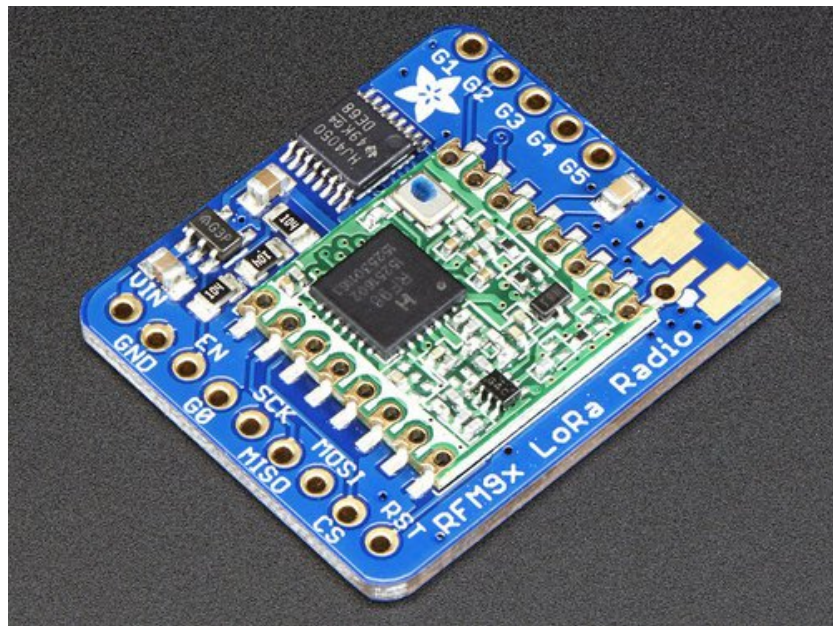


□

Adafruit RFM69HCW and RFM9X LoRa Packet Radio Breakouts

Created by lady ada



Last updated on 2016-09-13 05:09:05 PM UTC

Guide Contents

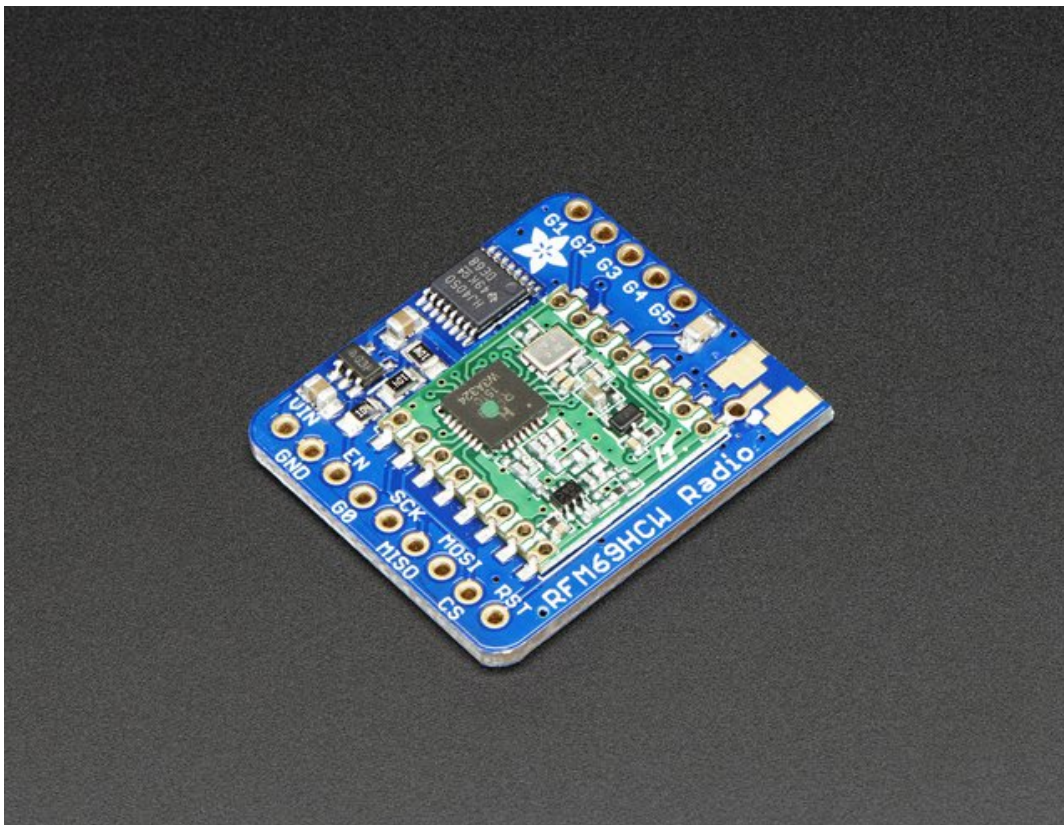
Guide Contents	2
Overview	4
Pinouts	8
Power Pins	8
SPI Logic pins:	9
Radio GPIO	10
Antenna Connection	11
Assembly	13
Prepare the header strip:	13
Add the breakout board:	13
And Solder!	14
Antenna Options	15
Wire Antenna	15
uFL Connector	17
SMA Edge-Mount Connector	19
Wiring	23
RFM69 Test	24
"Raw" vs Packetized	24
Arduino Libraries	24
LowPowerLab RFM69 Library example	25
Basic RX & TX example	25
Transmitter example code	25
Receiver example code	28
Radio Net & ID Configuration	31
Radio Type Config	32
Radio Pinout	32
Setup	32
Initializing Radio	33
Transmission Code	33
Receiver Code	33
Want more?	34
RFM9X Test	35
Arduino Library	35

RadioHead RFM9x Library example	35
Basic RX & TX example	36
Transmitter example code	36
Receiver example code	38
Radio Pinout	41
Frequency	41
Setup	41
Initializing Radio	42
Transmission Code	42
Receiver Code	43
Downloads	44
Datasheets & Files	44
Schematic	44
Fabrication Print	44

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.

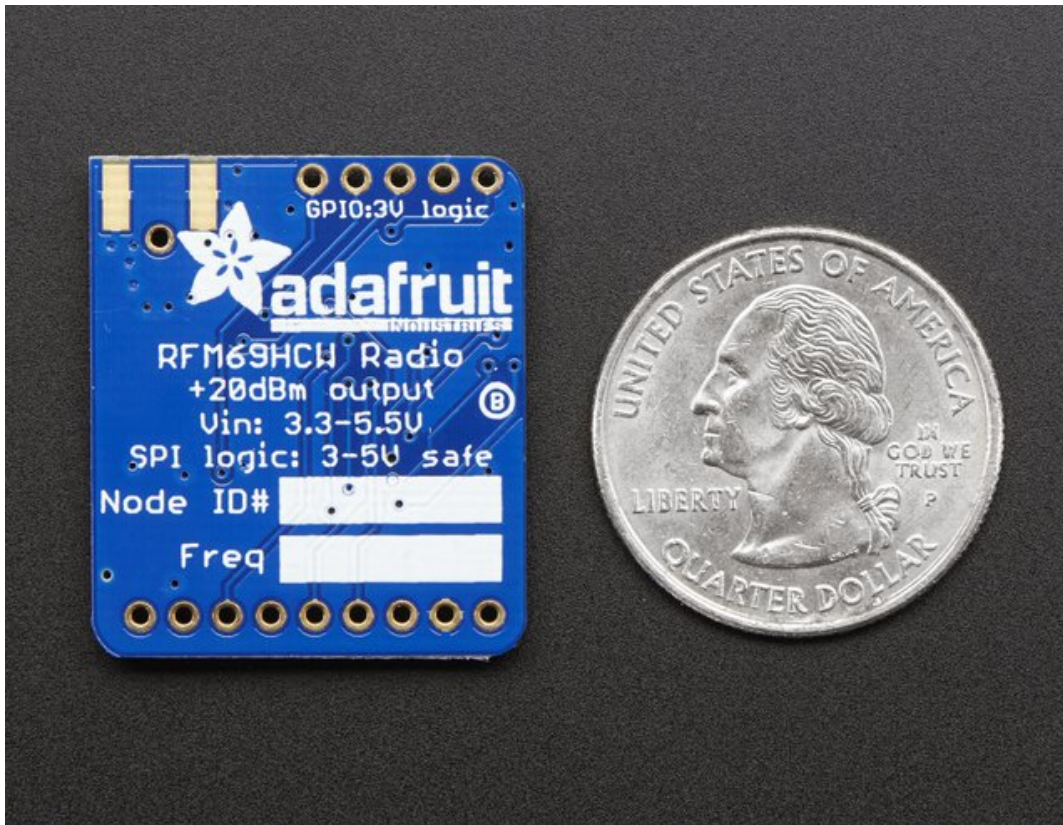
These modules are great for use with Arduinos or other microcontrollers, say if you want a sensor node network or transmit data over a campus or town. The trade off is you need two or more radios, with matching frequencies. WiFi and BT, on the other hand, are commonly included in computers and phones.

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:

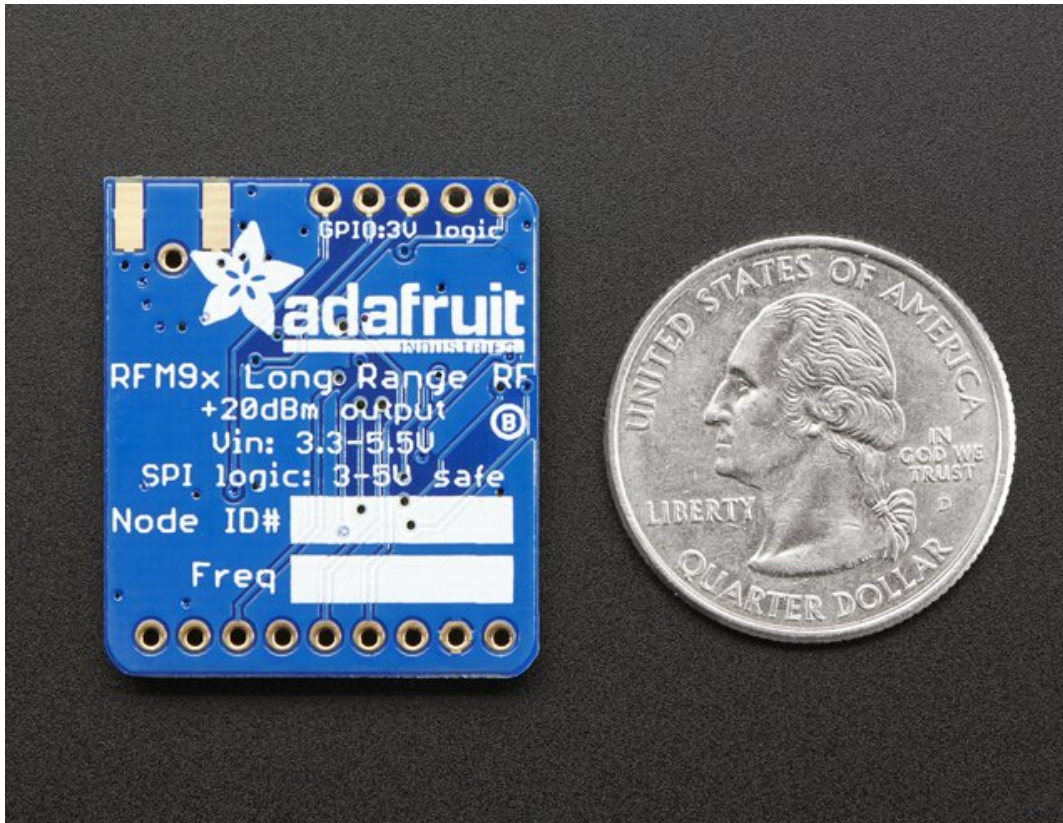
- Packet radio with ready-to-go Arduino libraries
- Uses the amateur or [license-free ISM bands](http://adafru.it/mOE) (<http://adafru.it/mOE>): 433MHz is ITU "Europe" license-free ISM or ITU "American" amateur with limitations. 900MHz is license free ISM for ITU "Americas"
- Use a simple wire antenna 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 at least 500 meters line of sight using simple wire antennas, probably up to 5Km with directional antennas and settings tweakings

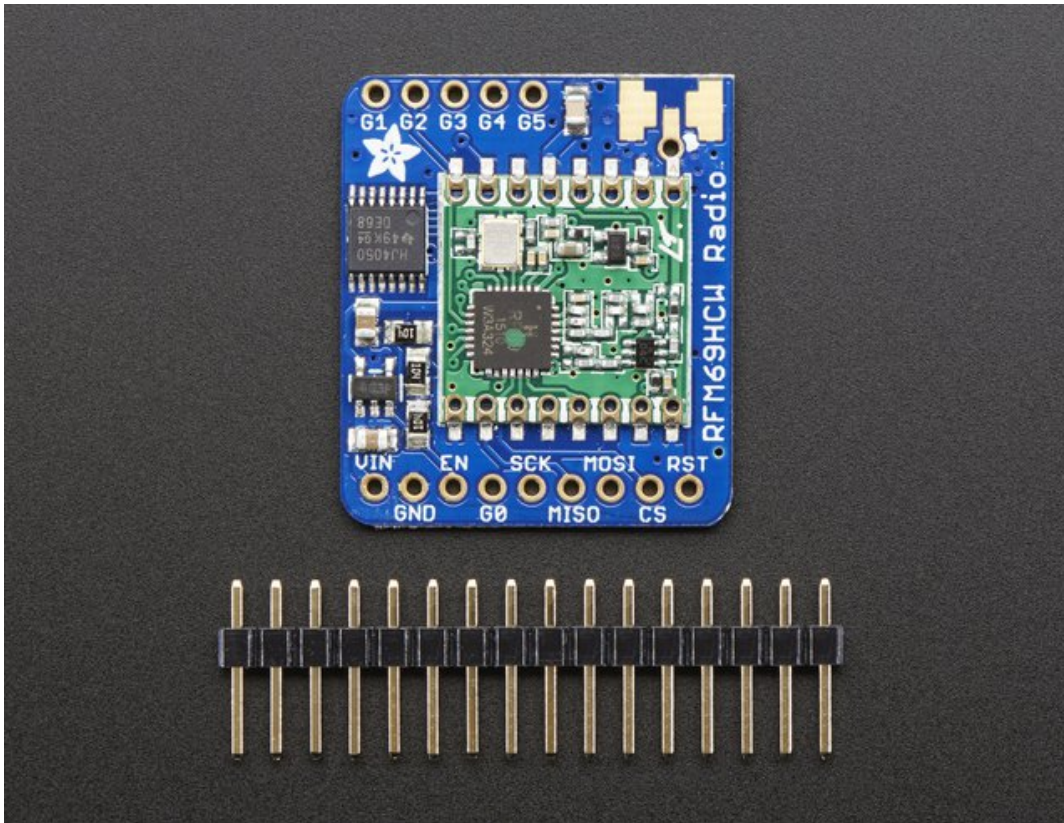
- SX1231 based module with SPI interface
- +13 to +20 dBm up to 100 mW Power Output Capability (power output selectable in software)
- 50mA (+13 dBm) to 150mA (+20dBm) current draw for transmissions, ~30mA during active radio listening.
- Range of approx. 500 meters, depending on obstructions, frequency, antenna and power output
- 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

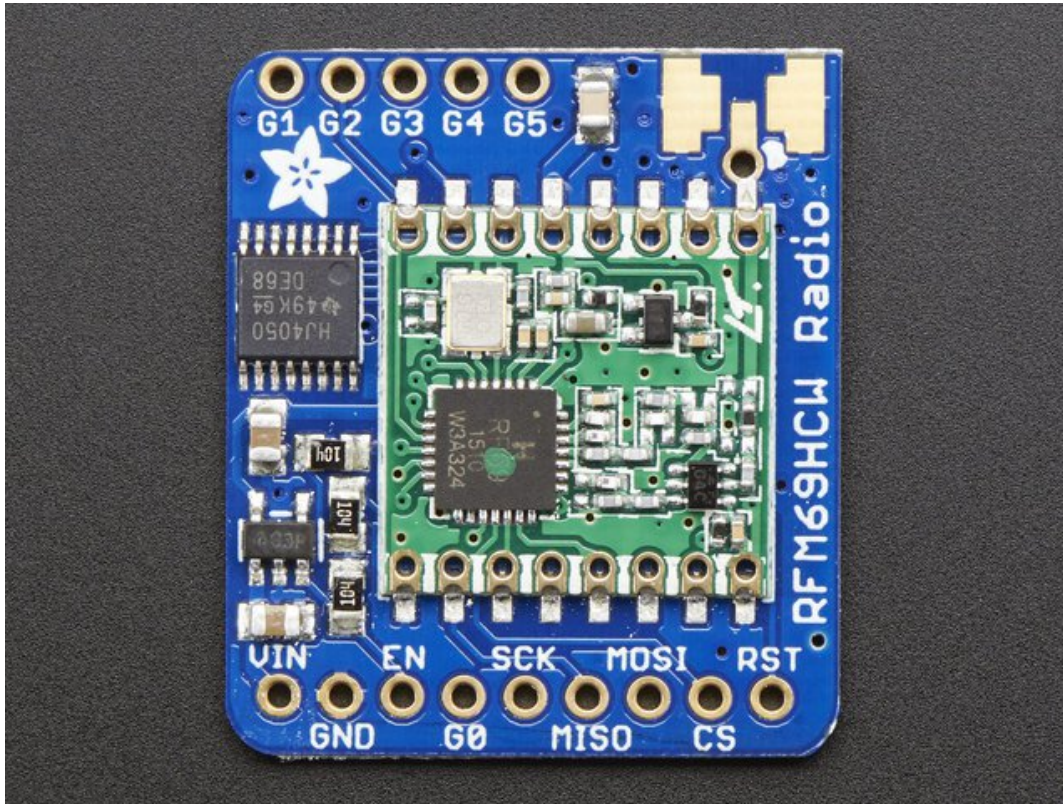
- SX1276 LoRa® based module with SPI interface
- +5 to +20 dBm up to 100 mW Power Output Capability (power output selectable in software)
- ~100mA peak during +20dBm transmit, ~30mA during active radio listening.
- Range of approx. 2Km, depending on obstructions, frequency, antenna and power output



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.

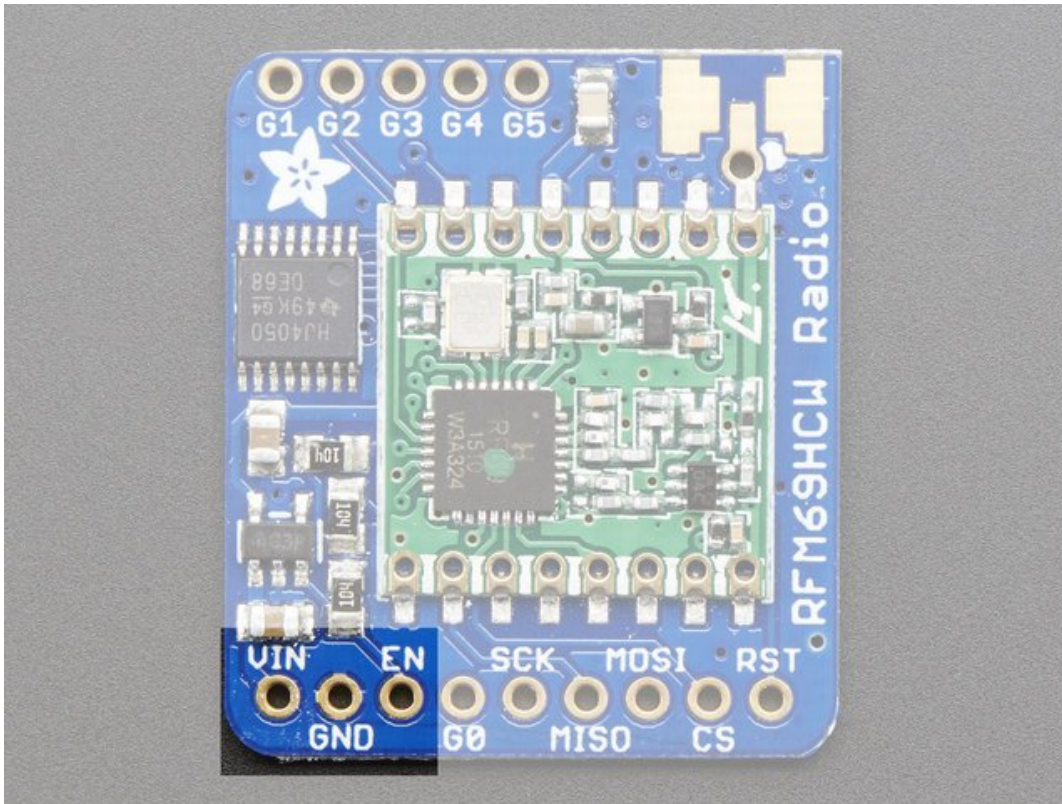
Each radio comes with some header, a 3.3V voltage regulator and levelshifter that can handle 3-5V DC power and logic so you can use it with 3V or 5V devices. Some soldering is required to attach the header. You will need to cut and solder on a small piece of wire (any solid or stranded core is fine) 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

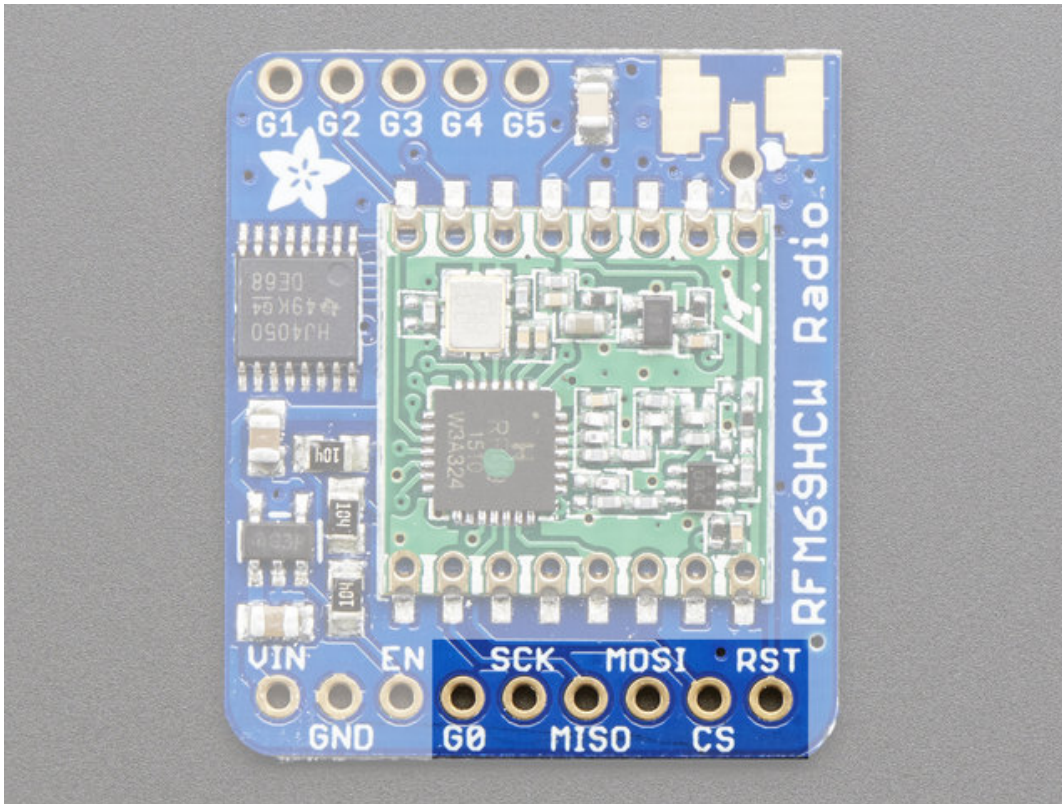
Power Pins



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.

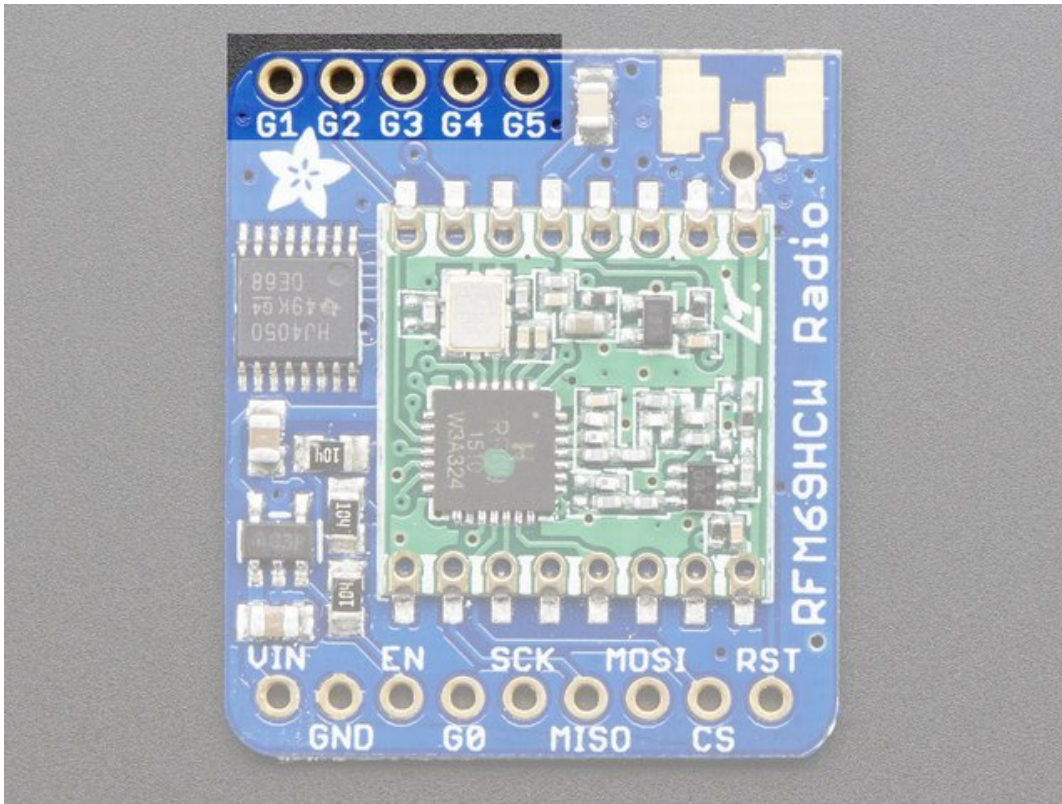
SPI Logic pins:



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

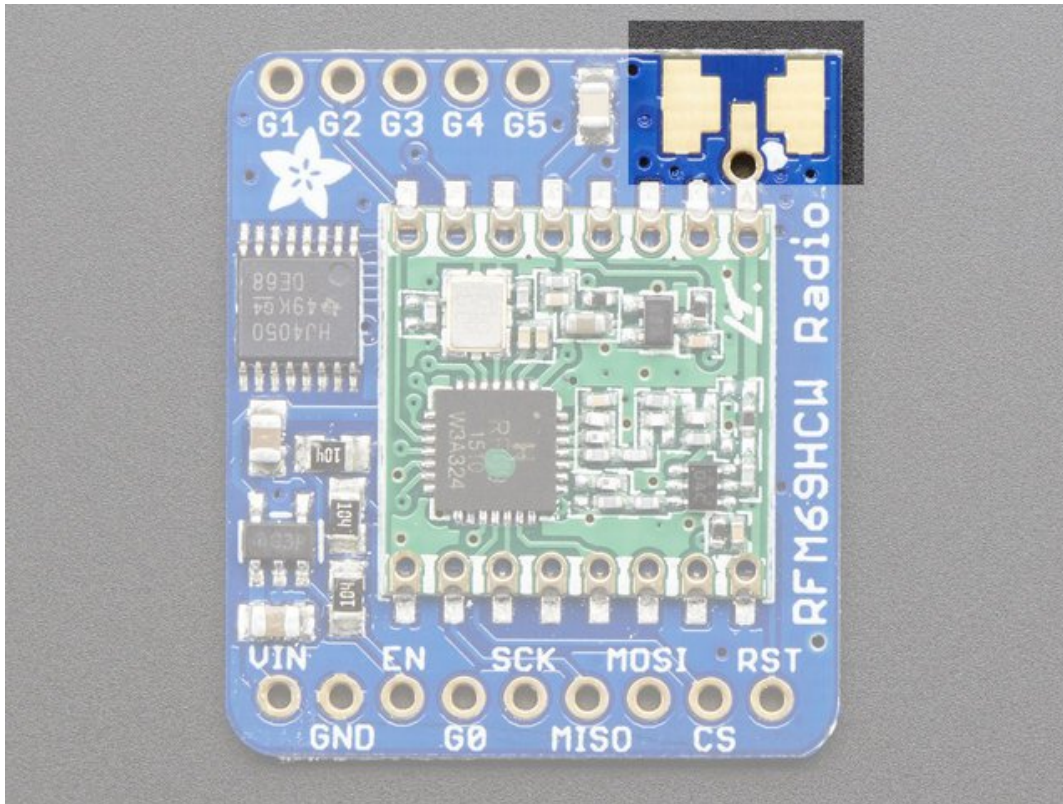
Radio GPIO



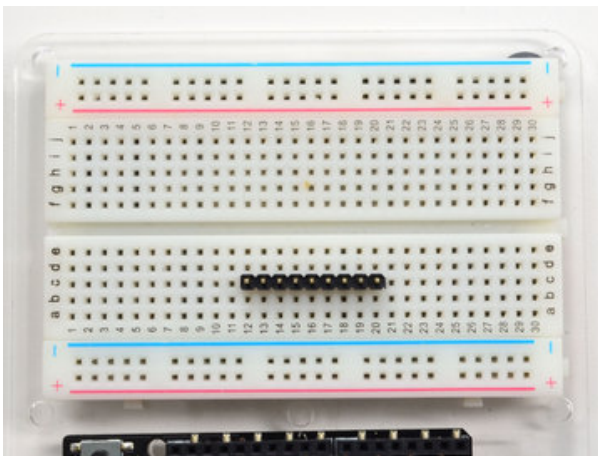
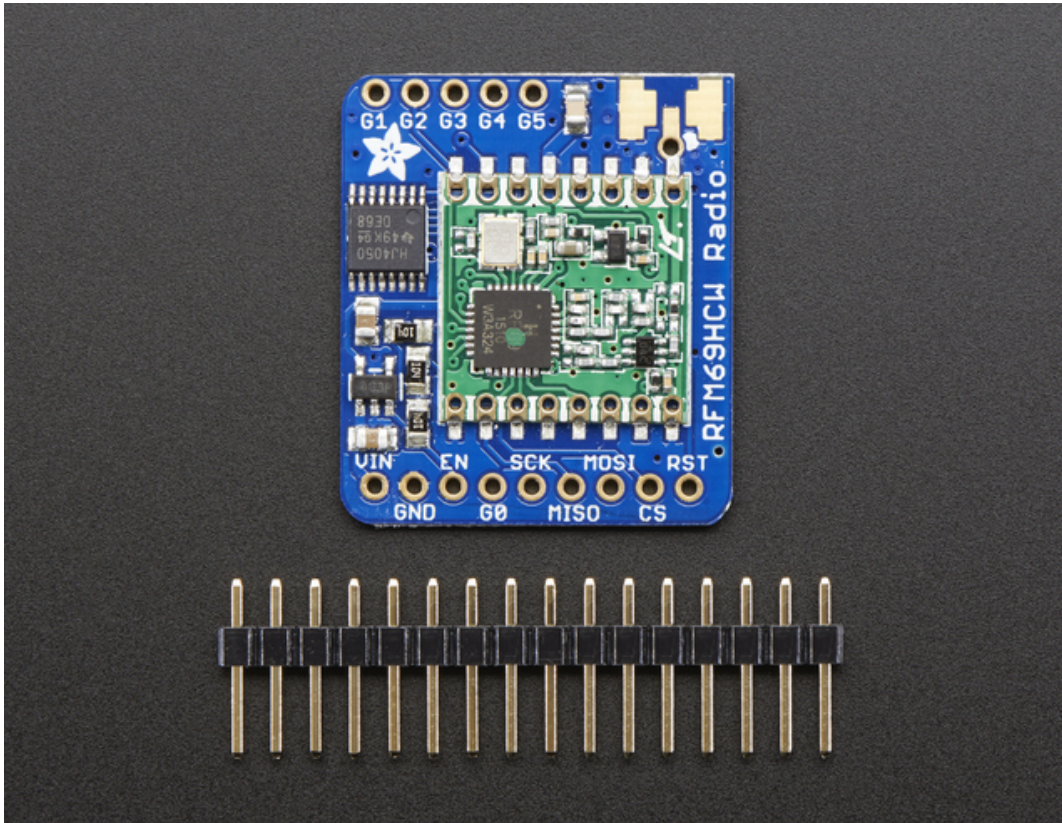
The radio's have another 5 GPIO pins that can be used for various notifications or radio functions. These aren't used for the majority of uses but are available in case you want them! All are 3.3V logic with no level shifting

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



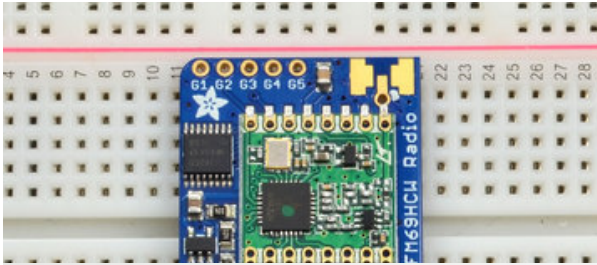
Assembly



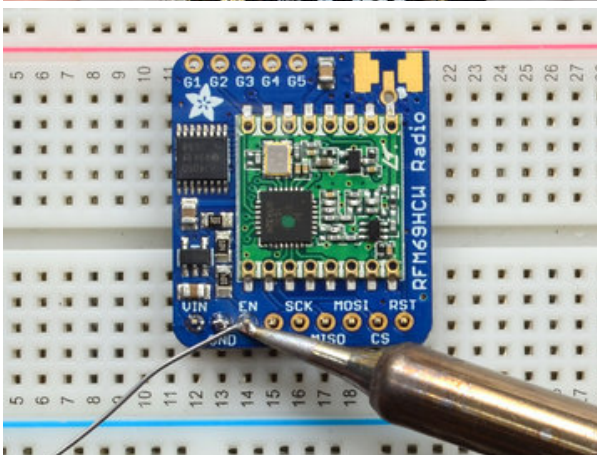
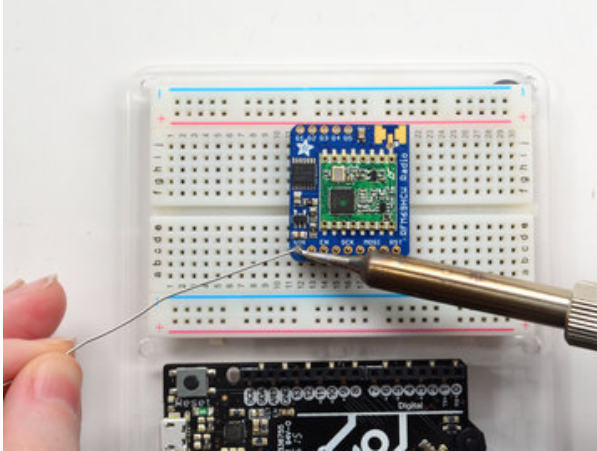
Prepare the header strip:

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

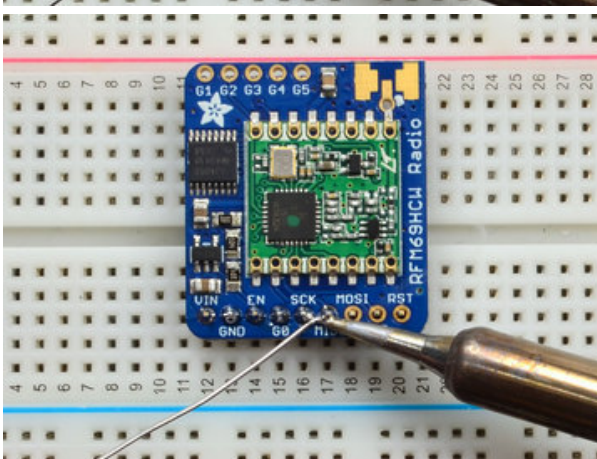
Add the breakout board:



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

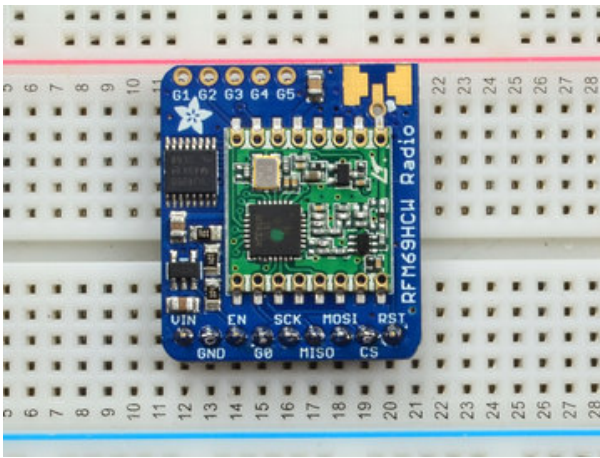
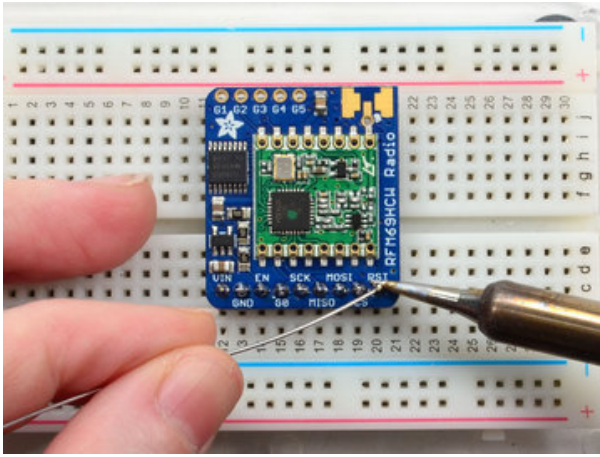


And Solder!



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://adafruit.it/aTk) (<http://adafruit.it/aTk>)).



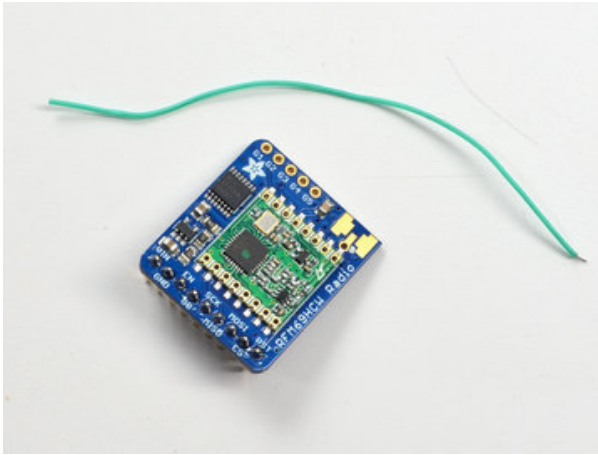
You're done! Check your solder joints visually and continue onto the next steps

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

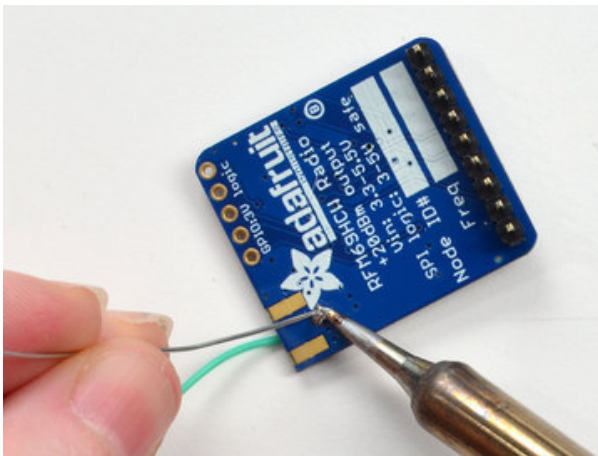
Wire Antenna

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

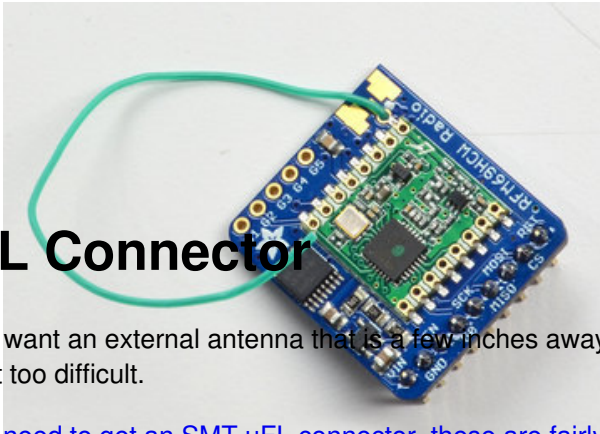


Strip a mm or two off the end of the wire, tin and solder into the **ANT** pad.



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

uFL Connector



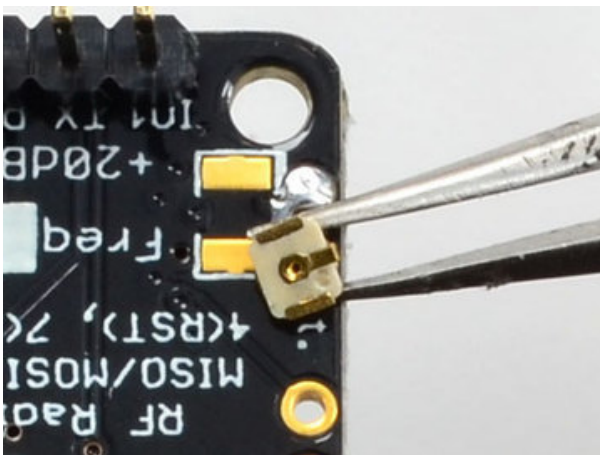
If you want an external antenna that is a few inches away from the radio, you need to do a tiny bit more work but its not too difficult.

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

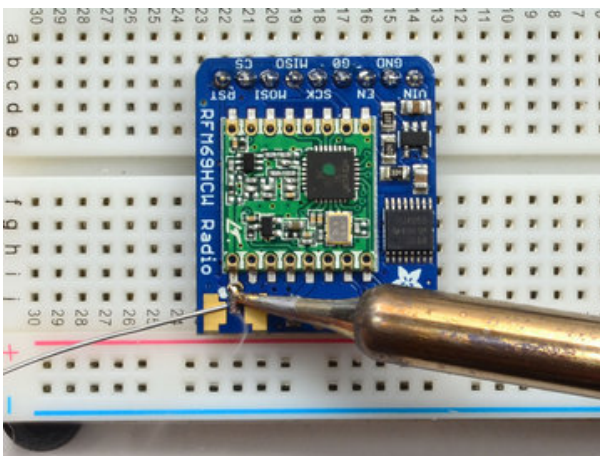
You'll also need a uFL to SMA adapter (<http://adafru.it/851>) (or whatever adapter you need for the antenna you'll be using, SMA is the most common

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

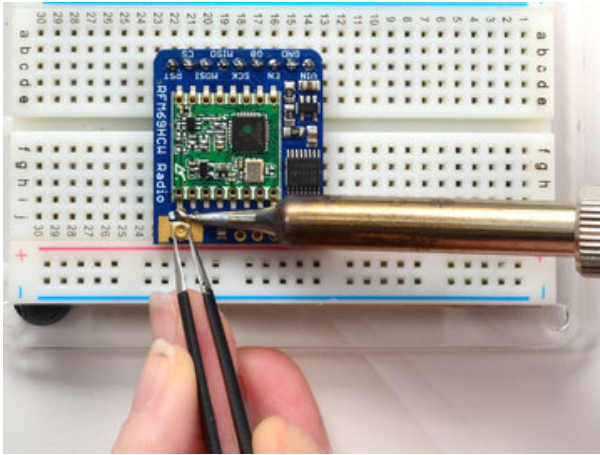
uFL connectors are rated for 30 connection cycles, but be careful when connecting/disconnecting to not rip the pads off the PCB. Once a uFL/SMA adapter is connected, use strain relief!



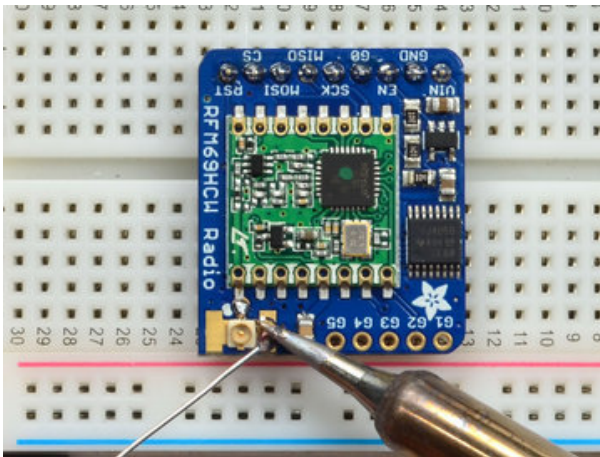
Check the bottom of the uFL connector, note that there's two large side pads (ground) and a little inlet pad. The other small pad is not used!



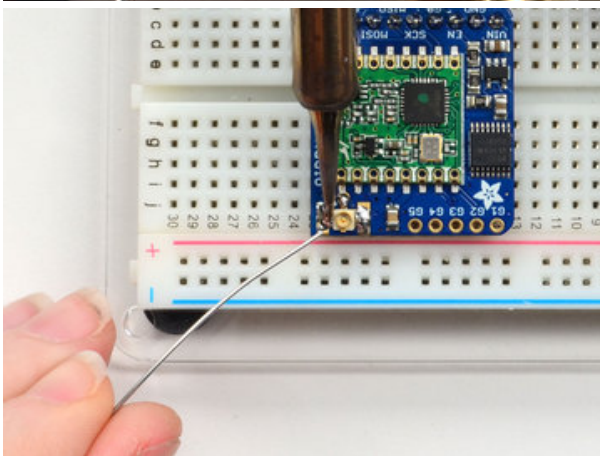
Put down a touch of solder on the signal pad



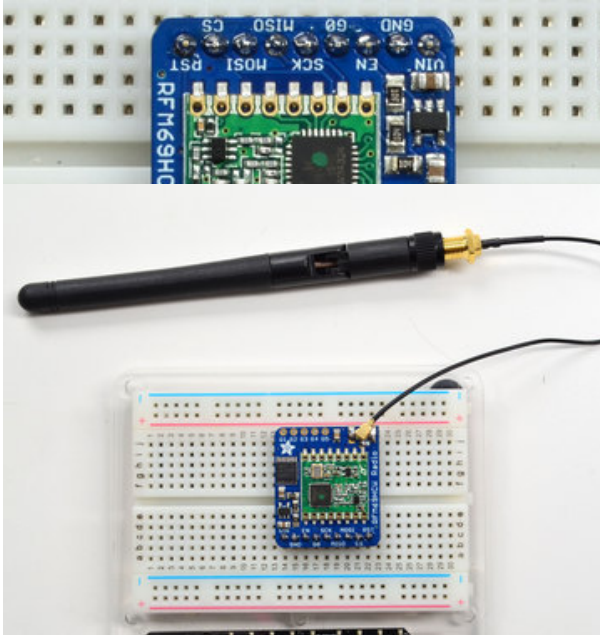
Solder in the first pad while holding the uFL steady



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



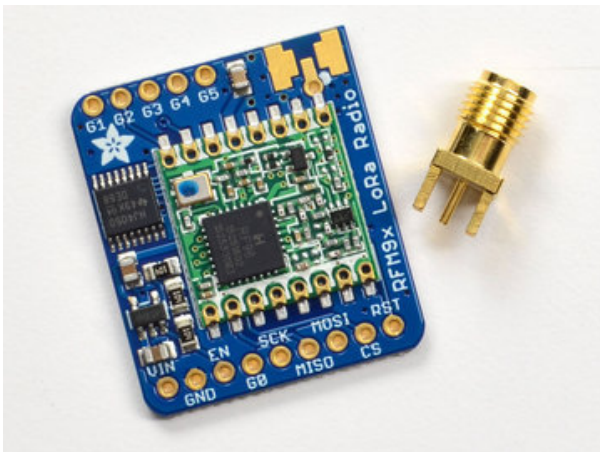
Once done, check your work visually



Once done attach your uFL adapter and antenna!

SMA Edge-Mount Connector

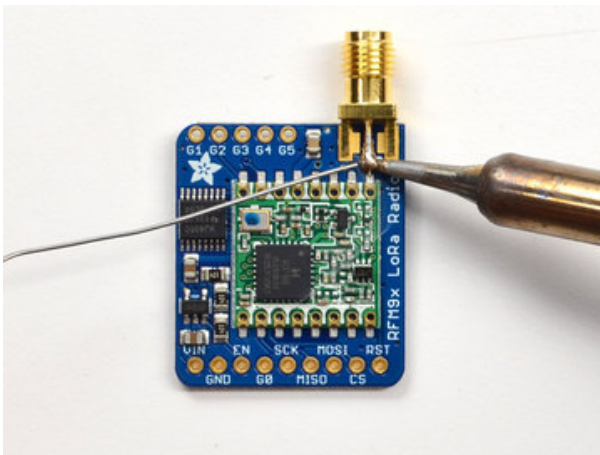
OK so



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

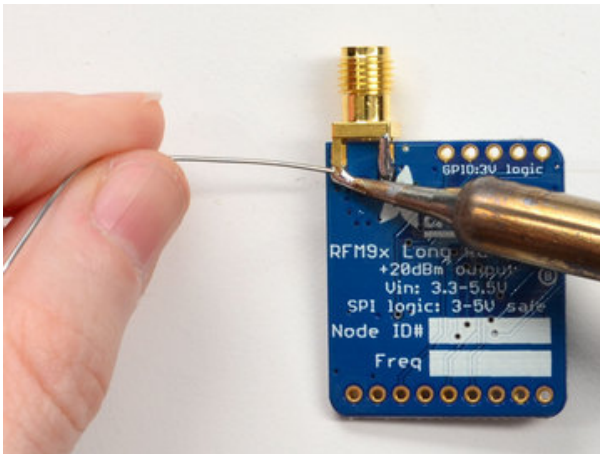
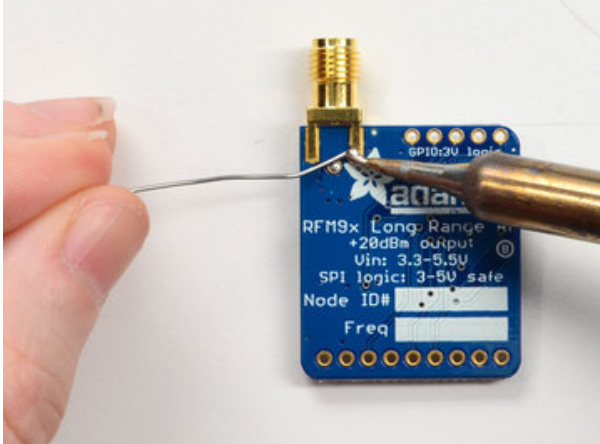
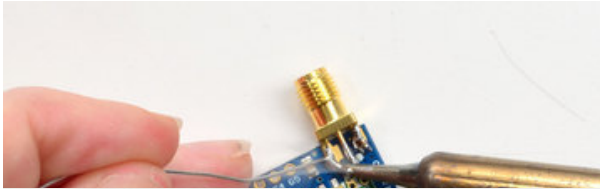
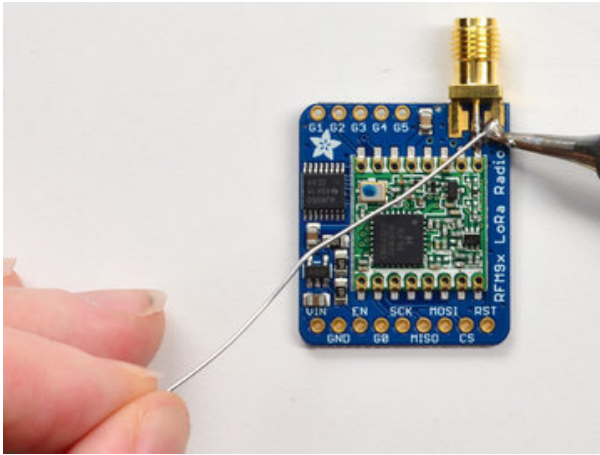


The SMA connector 'slides on' the top of the PCB



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



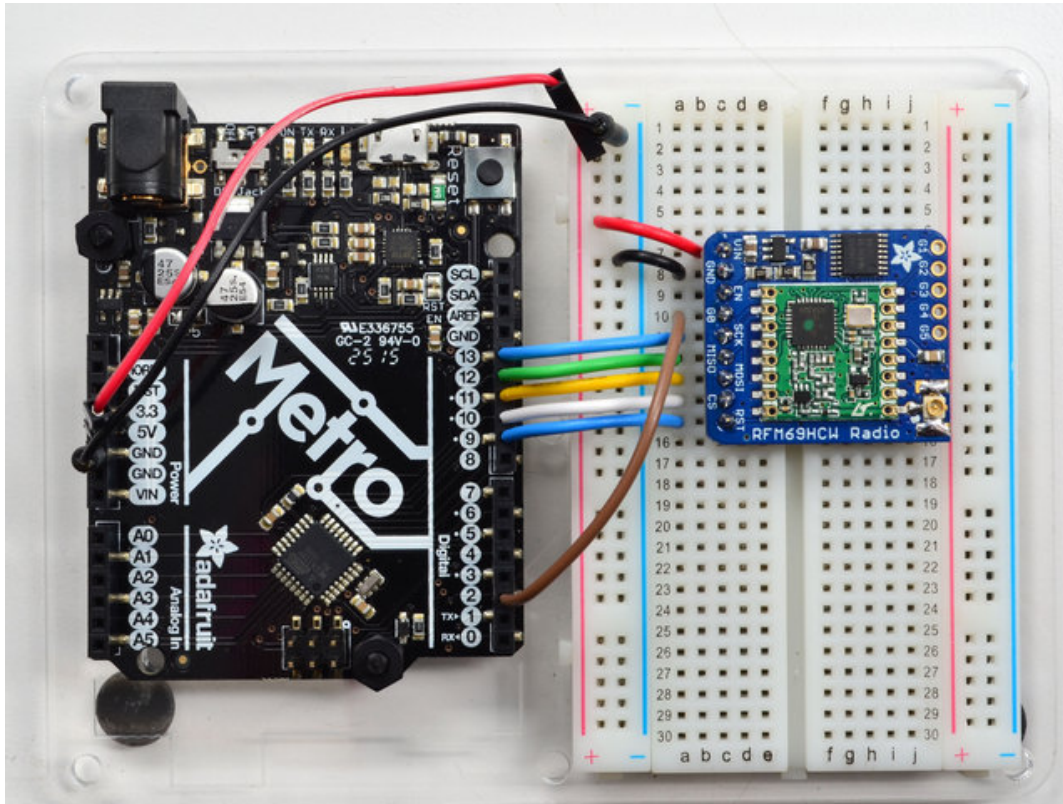
Flip over and also do the other side ground/mechanical contacts



Attach on your antenna, you're done!



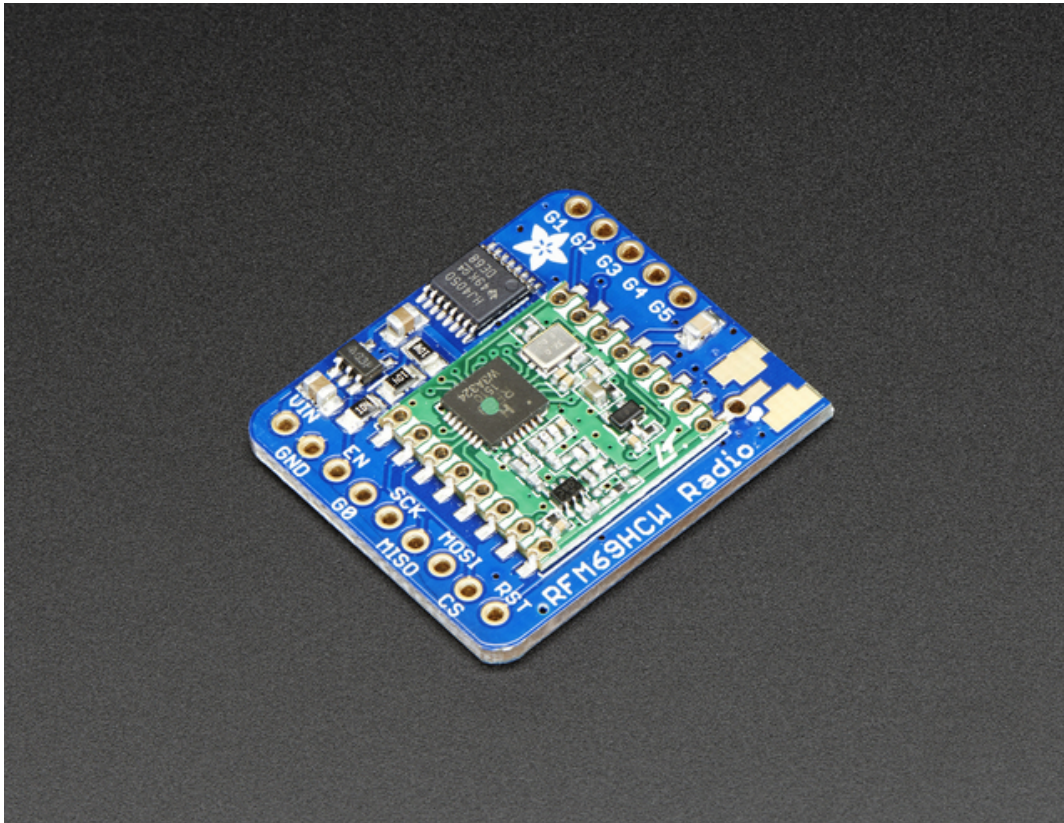
Wiring



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))
- **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 10** but you can later change this to any pin
- **RST** connects to our radio reset pin. We'll be using **Digital 9** but you can later change this pin too.
- **G0 (IRQ)** connects to an interrupt-capable pin. We'll be using **Digital 2** 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 2 is interrupt #0

RFM69 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 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.

"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 wont 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!

LowPowerLab RFM69 Library example

To begin talking to the radio, you will need to [download RFM69 from their github repository](http://adafru.it/mCz) (<http://adafru.it/mCz>). You can do that by visiting the github repo and manually downloading or, easier, just click this button to download the zip

[Download RFM69 Library](http://adafru.it/mCB)
<http://adafru.it/mCB>

Rename the uncompressed folder **RFM69** and check that the **RFM69** folder contains **RFM69.cpp** and **RFM69.h**

Place the **RFM69** library folder your **arduinofolder/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 radio 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!

```
/* RFM69 library and code by Felix Rusu - felix@lowpowerlab.com
// Get libraries at: https://github.com/LowPowerLab/
// Make sure you adjust the settings in the configuration section below !!!
// *****
// Copyright Felix Rusu, LowPowerLab.com
// Library and code by Felix Rusu - felix@lowpowerlab.com
// *****
// License
// *****
// This program is free software; you can redistribute it
// and/or modify it under the terms of the GNU General
// Public License as published by the Free Software
// Foundation; either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will
// be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public
// License for more details.
//
```

```

// You should have received a copy of the GNU General
// Public License along with this program.
// If not, see <http://www.gnu.org/licenses></http>:.
//
// Licence can be viewed at
// http://www.gnu.org/licenses/gpl-3.0.txt
//
// Please maintain this license information along with authorship
// and copyright notices in any redistribution of this code
// *****/

#include <RFM69.h> //get it here: https://www.github.com/lowpowerlab/rfm69
#include <SPI.h>

//*****
// ***** IMPORTANT SETTINGS - YOU MUST CHANGE/ONFIGURE TO FIT YOUR HARDWARE *****
//*****

#define NETWORKID 100 // The same on all nodes that talk to each other
#define NODEID 2 // The unique identifier of this node
#define RECEIVER 1 // The recipient of packets

//Match frequency to the hardware version of the radio on your Feather
//#define FREQUENCY RF69_433MHZ
//#define FREQUENCY RF69_868MHZ
#define FREQUENCY RF69_915MHZ
#define ENCRYPTKEY "sampleEncryptKey" //exactly the same 16 characters/bytes on all nodes!
#define IS_RFM69HCW true // set to 'true' if you are using an RFM69HCW module

//*****
#define SERIAL_BAUD 115200

#define RFM69_CS 10
#define RFM69_IRQ 2
#define RFM69_IRQN 0 // Pin 2 is IRQ 0!
#define RFM69_RST 9

#define LED 13 // onboard blinky

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

RFM69 radio = RFM69(RFM69_CS, RFM69_IRQ, IS_RFM69HCW, RFM69_IRQN);

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

  Serial.println("Arduino RFM69HCW Transmitter");

  // Hard Reset the RFM module
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, HIGH);
  delay(100);
  digitalWrite(RFM69_RST, LOW);
  delay(100);

  // Initialize radio
  radio.initialize(FREQUENCY,NODEID,NETWORKID);
  if (IS_RFM69HCW) {
    radio.setHighPower(); // Only for RFM69HCW & HW!
  }
  radio.setPowerLevel(31); // power output ranges from 0 (5dBm) to 31 (20dBm)

```



```

radio.encrypt(ENCRYPTKEY);

pinMode(LED, OUTPUT);
Serial.print("\nTransmitting at ");
Serial.print(FREQUENCY==RF69_433MHZ ? 433 : FREQUENCY==RF69_868MHZ ? 868 : 915);
Serial.println(" MHz");
}

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);

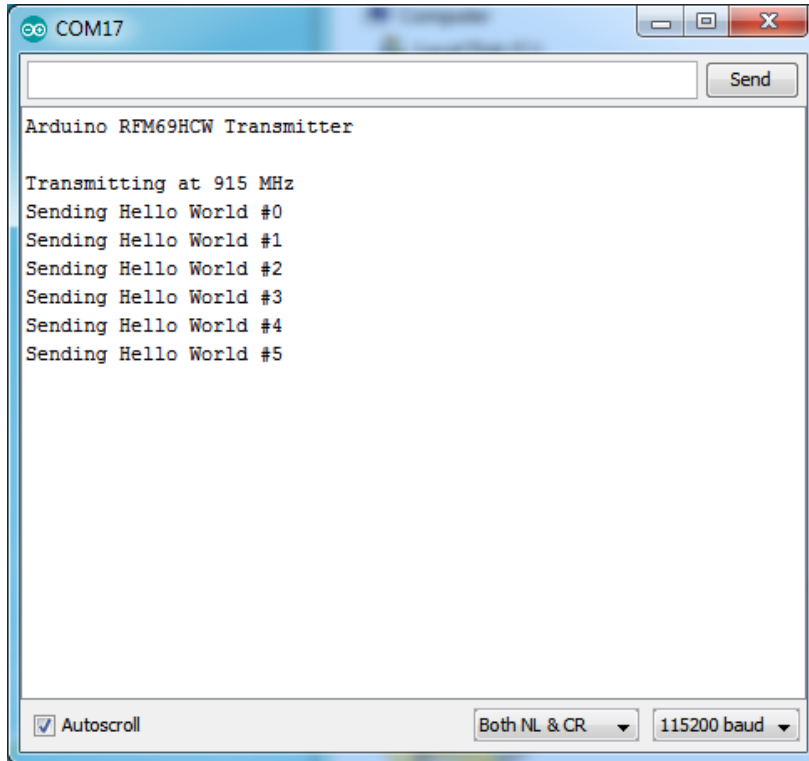
  if (radio.sendWithRetry(RECEIVER, radiopacket, strlen(radiopacket))) { //target node Id, message as string or byte array, message length
    Serial.println("OK");
    Blink(LED, 50, 3); //blink LED 3 times, 50ms between blinks
  }

  radio.receiveDone(); //put radio in RX mode
  Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
}

void Blink(byte PIN, byte DELAY_MS, byte loops)
{
  for (byte i=0; i<loops; i++)
  {
    digitalWrite(PIN,HIGH);
    delay(DELAY_MS);
    digitalWrite(PIN,LOW);
    delay(DELAY_MS);
  }
}

```

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 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!

```
/* RFM69 library and code by Felix Rusu - felix@lowpowerlab.com
// Get libraries at: https://github.com/LowPowerLab/
// Make sure you adjust the settings in the configuration section below !!!
// *****
// Copyright Felix Rusu, LowPowerLab.com
// Library and code by Felix Rusu - felix@lowpowerlab.com
// *****
// License
// *****
// This program is free software; you can redistribute it
// and/or modify it under the terms of the GNU General
// Public License as published by the Free Software
// Foundation; either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will
// be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A
// PARTICULAR PURPOSE. See the GNU General Public
// License for more details.
//
// You should have received a copy of the GNU General
// Public License along with this program.
// If not, see <http://www.gnu.org/licenses></http>.>
```

```

//
// Licence can be viewed at
// http://www.gnu.org/licenses/gpl-3.0.txt
//
// Please maintain this license information along with authorship
// and copyright notices in any redistribution of this code
// *****/

#include <RFM69.h> //get it here: https://www.github.com/lowpowerlab/rfm69
#include <SPI.h>

//*****
// ***** IMPORTANT SETTINGS - YOU MUST CHANGE/ONFIGURE TO FIT YOUR HARDWARE *****
//*****

#define NETWORKID 100 //the same on all nodes that talk to each other
#define NODEID 1

//Match frequency to the hardware version of the radio on your Feather
//#define FREQUENCY RF69_433MHZ
//#define FREQUENCY RF69_868MHZ
#define FREQUENCY RF69_915MHZ
#define ENCRYPTKEY "sampleEncryptKey" //exactly the same 16 characters/bytes on all nodes!
#define IS_RFM69HCW true // set to 'true' if you are using an RFM69HCW module

//*****
#define SERIAL_BAUD 115200

#define RFM69_CS 10
#define RFM69_IRQ 2
#define RFM69_IRQN 0 // Pin 2 is IRQ 0!
#define RFM69_RST 9

#define LED 13 // onboard blinky

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

RFM69 radio = RFM69(RFM69_CS, RFM69_IRQ, IS_RFM69HCW, RFM69_IRQN);

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

  Serial.println("Feather RFM69HCW Receiver");

  // Hard Reset the RFM module
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, HIGH);
  delay(100);
  digitalWrite(RFM69_RST, LOW);
  delay(100);

  // Initialize radio
  radio.initialize(FREQUENCY, NODEID, NETWORKID);
  if (IS_RFM69HCW) {
    radio.setHighPower(); // Only for RFM69HCW & HW!
  }
  radio.setPowerLevel(31); // power output ranges from 0 (5dBm) to 31 (20dBm)

  radio.encrypt(ENCRYPTKEY);

  pinMode(LED, OUTPUT);

  Serial.print("\nListening at ");

```



```

Serial.print(FREQUENCY==RF69_433MHZ ? 433 : FREQUENCY==RF69_868MHZ ? 868 : 915);
Serial.println(" MHz");
}

void loop() {
  //check if something was received (could be an interrupt from the radio)
  if (radio.receiveDone())
  {
    //print message received to serial
    Serial.print("[");Serial.print(radio.SENDERID);Serial.print("] ");
    Serial.print((char*)radio.DATA);
    Serial.print("  [RX_RSSI:");Serial.print(radio.RSSI);Serial.print("]");

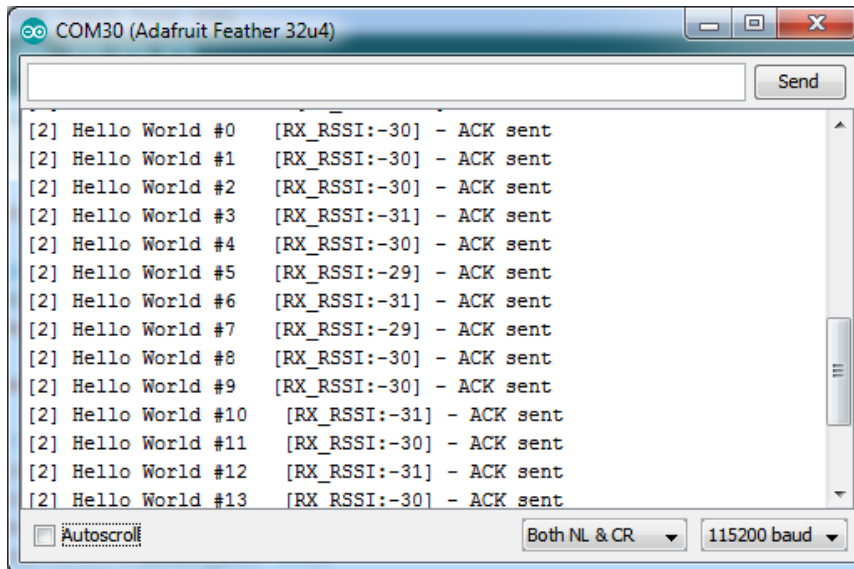
    //check if received message contains Hello World
    if (strstr((char *)radio.DATA, "Hello World"))
    {
      //check if sender wanted an ACK
      if (radio.ACKRequested())
      {
        radio.sendACK();
        Serial.println(" - ACK sent");
      }
      Blink(LED, 40, 3); //blink LED 3 times, 40ms between blinks
    }
  }

  radio.receiveDone(); //put radio in RX mode
  Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
}

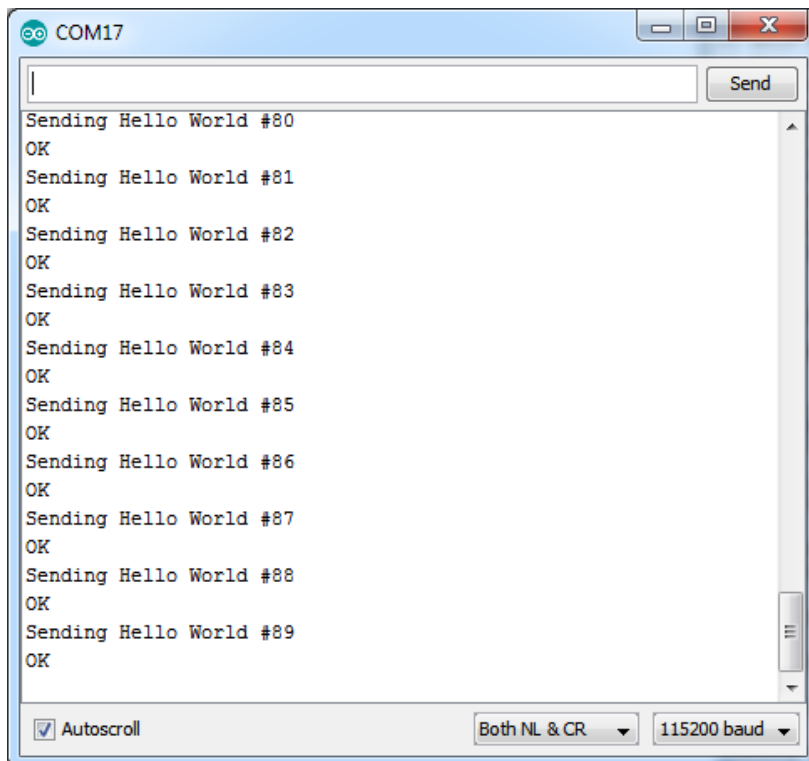
void Blink(byte PIN, byte DELAY_MS, byte loops)
{
  for (byte i=0; i<loops; i++)
  {
    digitalWrite(PIN,HIGH);
    delay(DELAY_MS);
    digitalWrite(PIN,LOW);
    delay(DELAY_MS);
  }
}

```

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 **OK** after each transmission because it got an acknowledgement 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 Net & ID Configuration

The first thing you need to do for *all* radio nodes on your network is to configure the network ID (the identifier shared by all radio nodes you are using) and individual ID's for each node.

```

//*****
// ***** IMPORTANT SETTINGS - YOU MUST CHANGE/ONFIGURE TO FIT YOUR HARDWARE *****
//*****
#define NETWORKID 100 // The same on all nodes that talk to each other
#define NODEID 2 // The unique identifier of this node

```

Make sure all radios you are using are sharing the same network ID and all have different/unique Node ID's!

Radio Type Config

You will also have to set up type of radio you are using, an encryption key if you're using it, and the type of radio (high or low power)

```

//Match frequency to the hardware version of the radio on your Feather
//#define FREQUENCY RF69_433MHZ
//#define FREQUENCY RF69_868MHZ
#define FREQUENCY RF69_915MHZ
#define ENCRYPTKEY "sampleEncryptKey" //exactly the same 16 characters/bytes on all nodes!
#define IS_RFM69HCW true // set to 'true' if you are using an RFM69HCW module

```

For all radios they will need to be on the same frequency. If you have a 433MHz radio you will have 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

The **ENCRYPTKEY** is 16 characters long and keeps your messages a little more secret than just plain text

IS_RFM69HCW is used for telling the library that we are using the high power version of the radio, make sure its set to **true**

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 (IRQ 0) or pin #3 (IRQ 1). Each chipset has different interrupt pins!

```

#define RFM69_CS 10
#define RFM69_IRQ 2
#define RFM69_IRQN 0 // Pin 2 is IRQ 0!
#define RFM69_RST 9

```

You can then instantiate the radio object with our custom pin numbers. Note that the IRQ is defined by the IRQ *number* not the pin.

```
RFM69 radio = RFM69(RFM69_CS, RFM69_IRQ, IS_RFM69HCW, RFM69_IRQN);
```

Setup

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

```

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

  Serial.println("RFM69HCW Transmitter");

  // Hard Reset the RFM module
  pinMode(RFM69_RST, OUTPUT);
  digitalWrite(RFM69_RST, HIGH);
}

```



```

delay(100);
digitalWrite(RFM69_RST, LOW);
delay(100);

```

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 the frequency, unique identifier and network identifier. For HCW type modules (which you are using) it will also turn on the amplifier. You can also configure the output power level, the number ranges from 0 to 31. Start with the highest power level (31) and then scale down as necessary

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

```

// Initialize radio
radio.initialize(FREQUENCY,NODEID,NETWORKID);
if (IS_RFM69HCW) {
  radio.setHighPower(); // Only for RFM69HCW & HW!
}
radio.setPowerLevel(31); // power output ranges from 0 (5dBm) to 31 (20dBm)

radio.encrypt(ENCRYPTKEY);

```

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 an acknowledgement

```

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);

  if (radio.sendWithRetry(RECEIVER, radiopacket, strlen(radiopacket))) { //target node Id, message as string or byte array, message length
    Serial.println("OK");
    Blink(LED, 50, 3); //blink LED 3 times, 50ms between blinks
  }

  radio.receiveDone(); //put radio in RX mode
  Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
}

```

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 **sendWithRetry** to the address in the first argument (RECEIVER), and passes in the array of data and the length of the data.

If you receive a **true** from that call it means an acknowledgement was received and print**OK** - either way the transmitter will continue the loop and sleep for a second until the next TX.

Receiver Code

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

```

void loop() {

```

```

//check if something was received (could be an interrupt from the radio)
if (radio.receiveDone())
{
  //print message received to serial
  Serial.print("[");Serial.print(radio.SENDERID);Serial.print("] ");
  Serial.print((char*)radio.DATA);
  Serial.print("  [RX_RSSI:");Serial.print(radio.RSSI);Serial.print("]");

  //check if received message contains Hello World
  if (strstr((char *)radio.DATA, "Hello World"))
  {
    //check if sender wanted an ACK
    if (radio.ACKRequested())
    {
      radio.sendACK();
      Serial.println(" - ACK sent");
    }
    Blink(LED, 40, 3); //blink LED 3 times, 40ms between blinks
  }
}

Serial.flush(); //make sure all serial data is clocked out before sleeping the MCU
}

```

Instead of transmitting, it is constantly checking if there's any data packets that have been received. **receiveDone()** will return true if a packet for the current node, in the right network and 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.

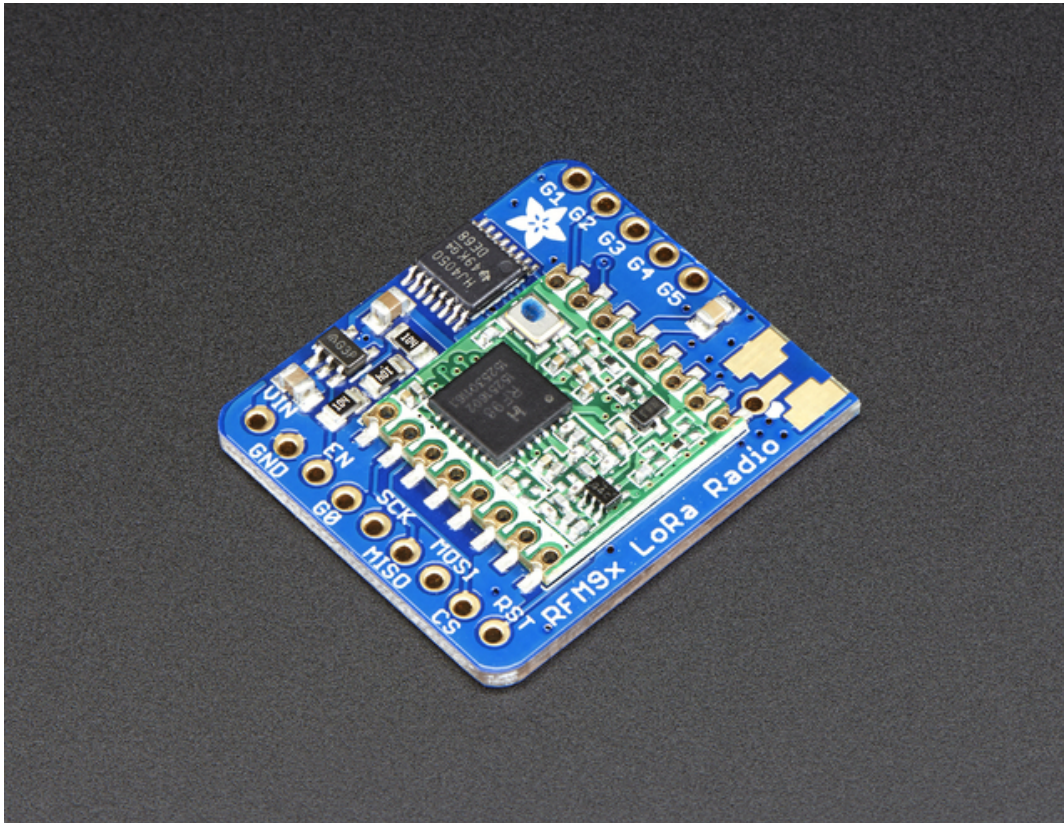
If the data contains the text "Hello World" it will also acknowledge the packet.

Once done it will continue waiting for a new packet

Want more?

[We have an example for Feather Radio that shows bidirectional button press and OLED displays, check it out here \(http://adafru.it/mOB\)](http://adafru.it/mOB)

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 it's common to stick to around 19.2 Kbps (that's 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](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 **arduinorsketchfolder/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

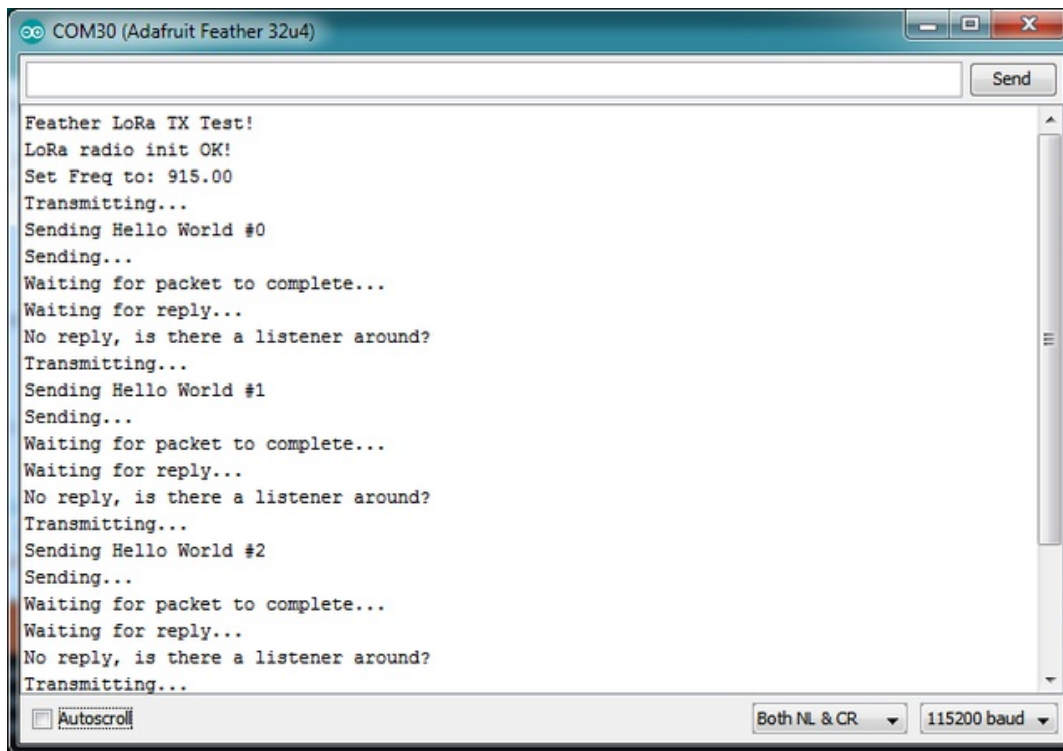
```

```

{
  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



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);
    }
  }
}

```

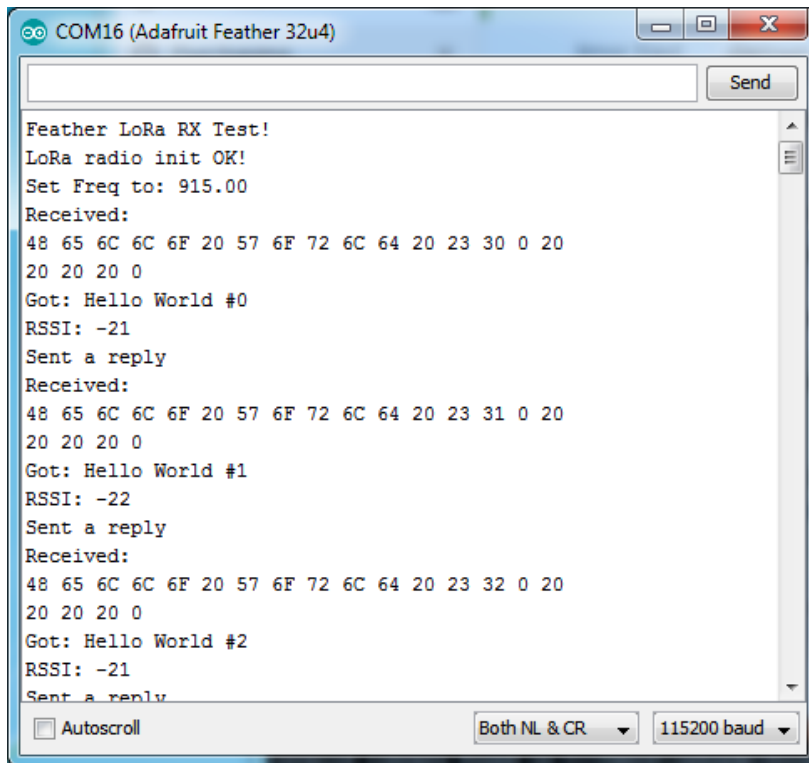
```

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");
}
}
}
}

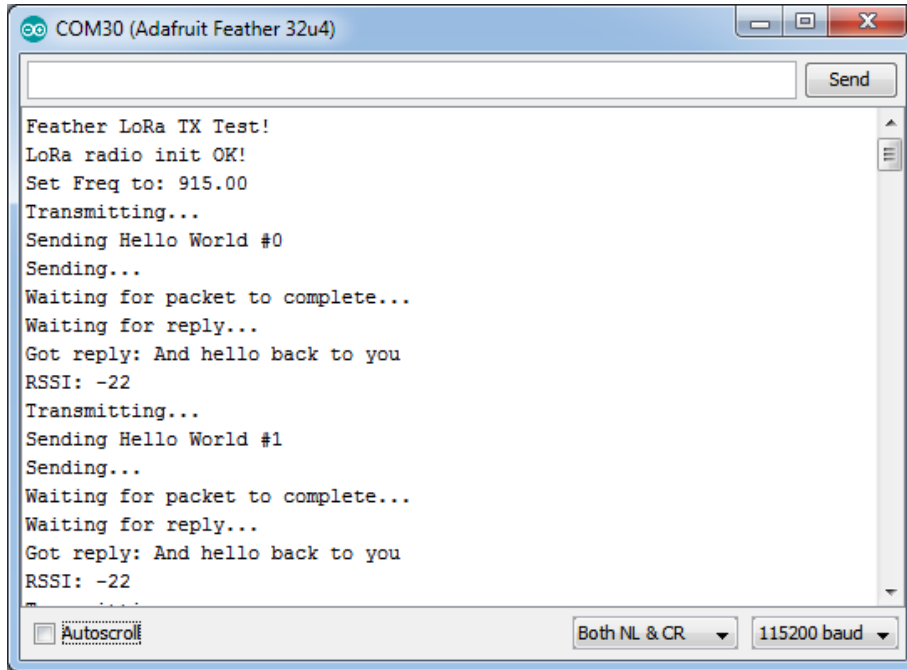
```

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 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

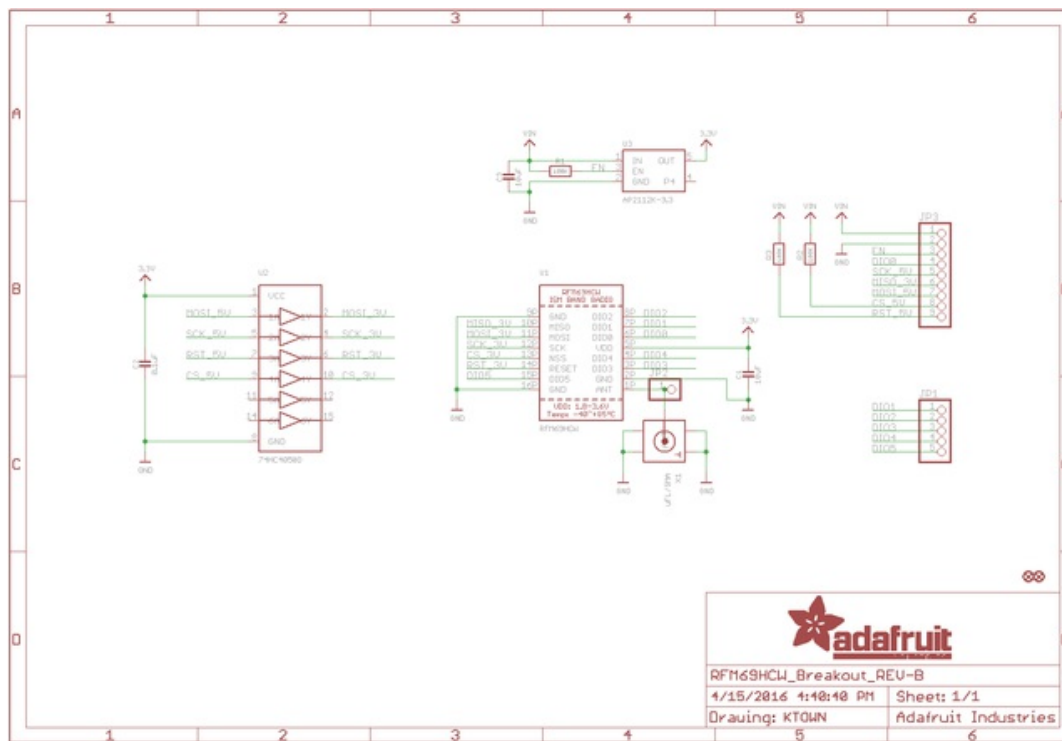
Downloads

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



Fabrication Print

RFM69 and RFM9X have the same layout so the same board is used

