RFM69Radio Documentation

Release 0.6.0

Jacob Kittley-Davies

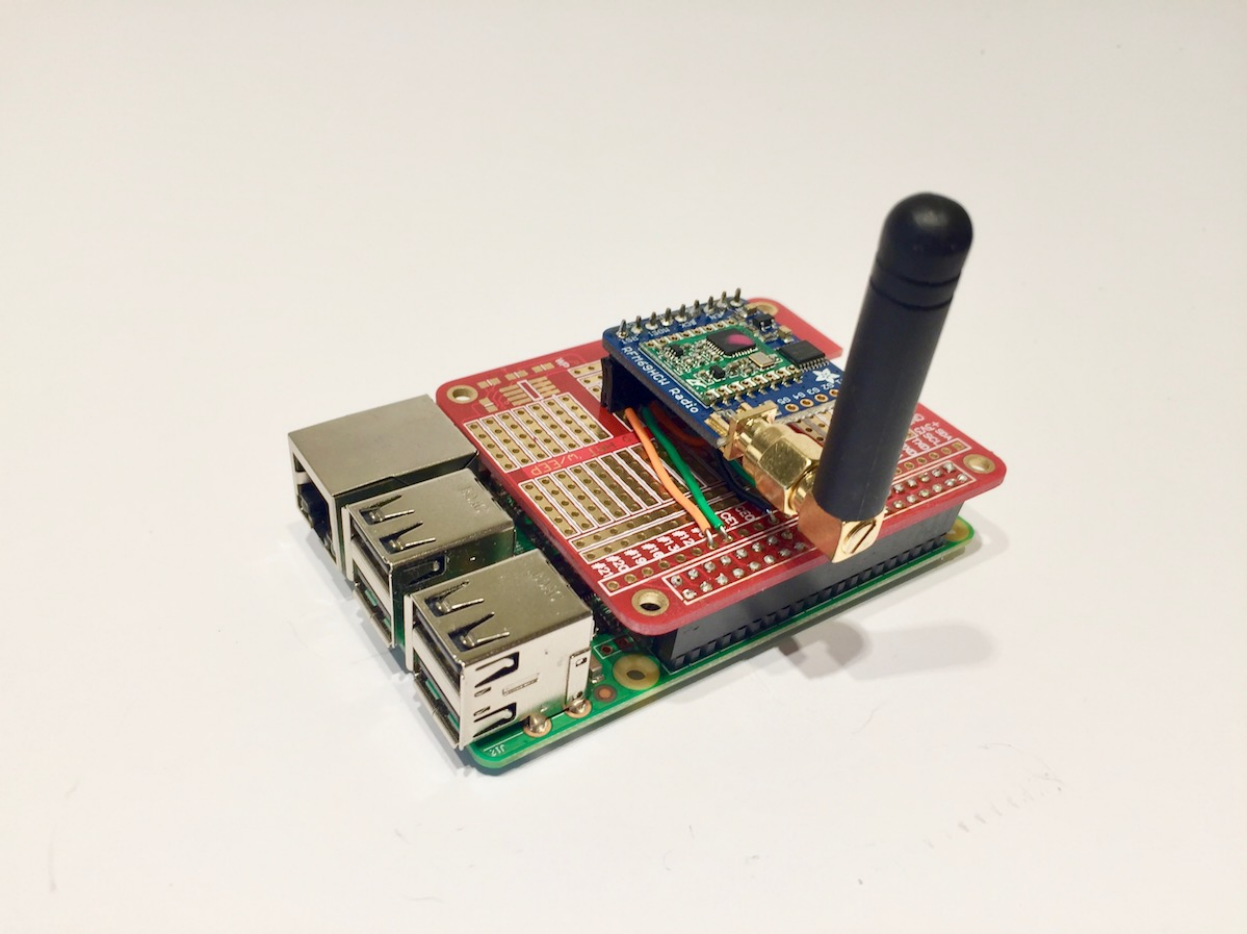
Aug 13, 2023

Contents:

Warning: This project is a Beta release and as such it may contain bugs. If you spot one then please report it as an issue on Github. We want to make this project awesome and to do that we need your help!

This package provides a Python wrapper of the LowPowerLabs RFM69 library and is largely based on the work of Eric Trombly who ported the library from C.

The package expects to be installed on a Raspberry Pi and depends on the RPI.GPIO and spidev libraries. In addition you need to have an RFM69 radio module directly attached to the Pi. For details on how to connect such a module checkout this guide *Connecting the Radio*.



CHAPTER 1

Contribute

- Issue Tracker: https://github.com/jgillula/rpi-rfm69/issues
- Source Code: https://github.com/jgillula/rpi-rfm69/issues

CHAPTER 2

Support

If you are having issues, please let us know by submitting an issue.

CHAPTER 3

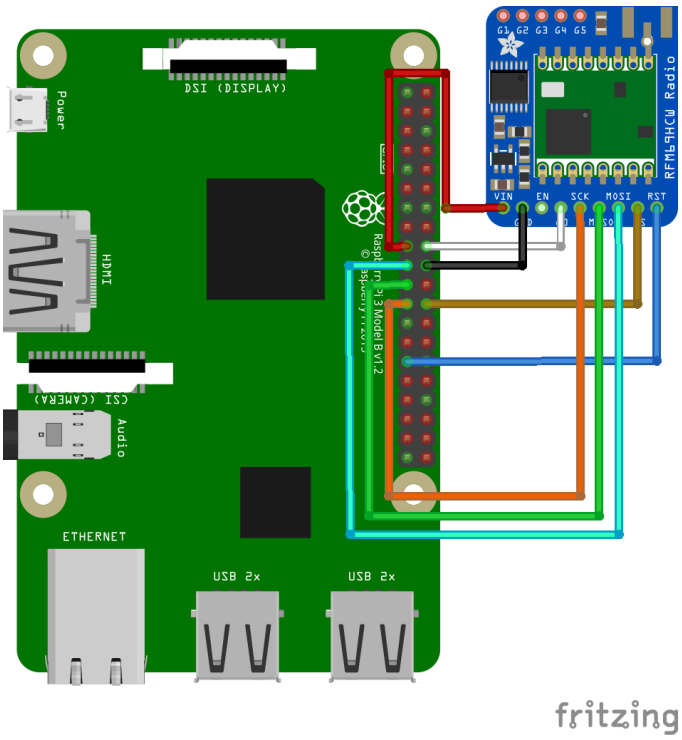
License

The project is licensed under the GPL v3 license.

3.1 Connecting the Radio

3.1.1 Wiring

The pins of the Raspberry Pi can be confusing and figuring out which pin is the right one and which one is going to make a loud popping noise can often mean repeated counting. If you struggle like I do, to mentally map from diagrams to the real world then I suggest buying or downloading a template card (a.k.a. a leaf).



So that is how we are going to connect it all up. Different makes of breakout board use different terminology. For example the Sparkfun boards use NSS rather than CS and DIO0 instead of G0. The table below is my best effort at making the wiring clear. If you come across any different terms then let me know and I will include them.

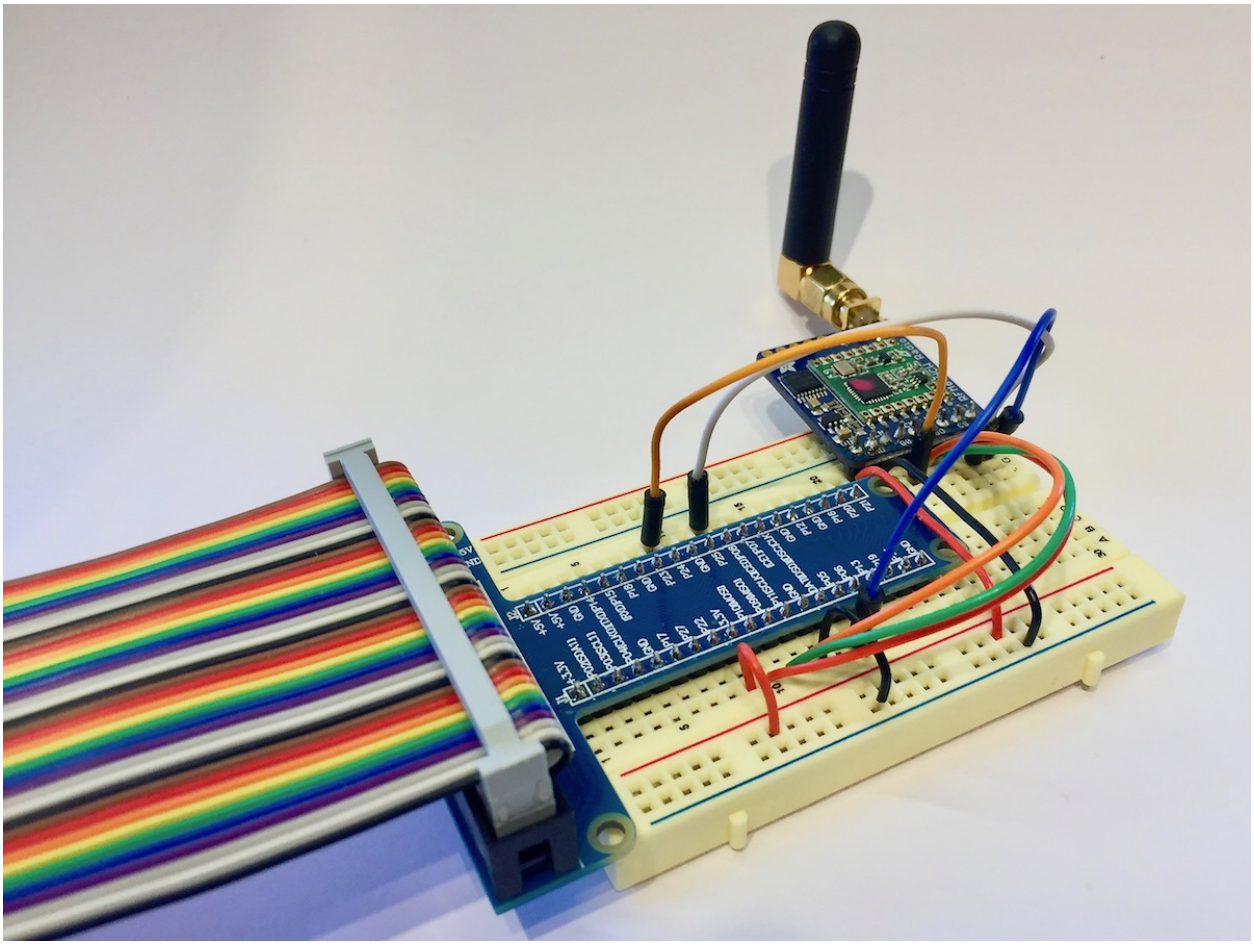
3.1.2 Pinout guide

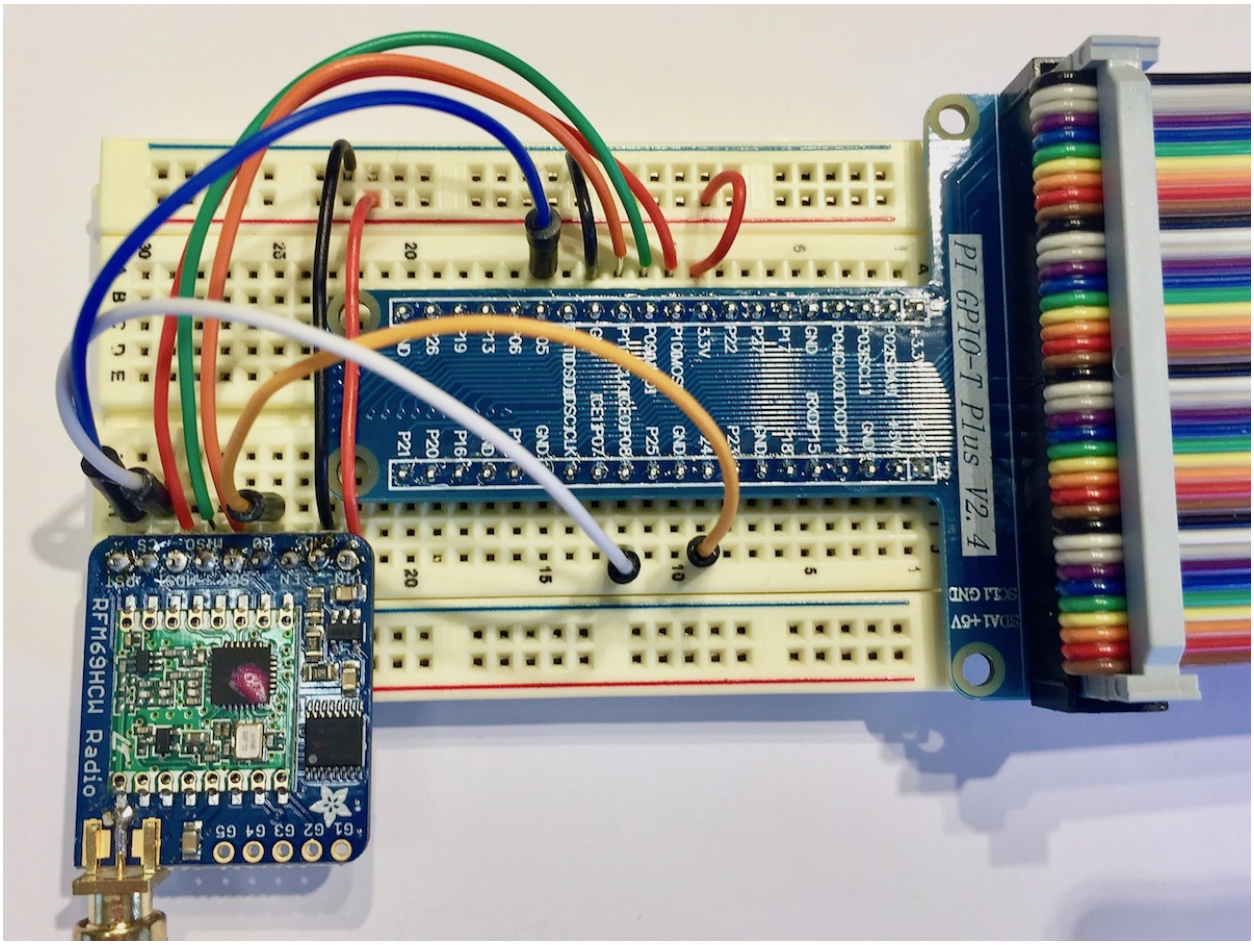
Table 1: Pin Guide Raspberry Pi to RFM69HCW (Adafruit and Spark Fun breakouts)

PI Name	3v3[1]	Ground	MOSI	MISO	SCLK	ID_SC[2]	CE0	
PI GPIO[3]			10	9	11		8	5
PI Pin	17	20	19	21	23	18	24	29
Adafruit	Vin	GND	MOSI	MISO	CLK	G0	CS	RST
Sparkfun	3.3v	GND	MOSI	MISO	SCK	DIO0	NSS	RESET

[1] This pin can only support low power mode. Use different supply if you want to use the RFM69HCW in high power mode.
[2] This is the interrupt pin. The RFM69HCW module calls an interrupt on the RPI when there is new data to process.
[3] These numbers refer to the pin index when you could from top left to bottom right i.e. odd number on the left and evens on the right. These are not the GPIO numbers. pinout.xyz is an amazingly helpful tool.

3.1.3 Setup Examples





3.2 Installing the library

3.2.1 Install

To install the library from Pypi:

```
pip3 install rpi-rfm69
```

Note: The library is dependant on "spidev" and "RPI.GPIO" and therefore will only install correctly on a Raspberry Pi. Also, since Python 2 is EOL only Python 3 is supported.

3.2.2 Get the source

The source code is available on Github: https://github.com/jgillula/rpi-rfm69

3.3 Example Code

3.3.1 Example - The Basics

Initialise

To initialise the radio you create a context:

```
from RFM69 import Radio, FREQ_433MHZ

this_node_id = 1
with Radio(FREQ_433MHZ, this_node_id) as radio:
    ... your code here ...
```

This ensures that the necessary clean-up code is executed when you exit the context.

Note: Frequency selection: FREQ_315MHZ, FREQ_433MHZ, FREQ_868MHZ or FREQ_915MHZ. Select the band appropriate to the radio you have.

Simple Transceiver

Here's a simple transceiver example.

```
# pylint: disable=unused-import

import time
from RFM69 import Radio, FREQ_315MHZ, FREQ_433MHZ, FREQ_868MHZ, FREQ_915MHZ

node_id = 1
network_id = 100
recipient_id = 2

# The following are for an Adafruit RFM69HCW Transceiver Radio
# Bonnet https://www.adafruit.com/product/4072
# You should adjust them to whatever matches your radio
board = {'isHighPower': True, 'interruptPin': 15, 'resetPin': 22, 'spiDevice': 1}

# The following are for an RaspyRFM RFM69CW Module #1
# http://www.seegel-systeme.de/2015/09/02/ein-funkmodul-fuer-den-raspberry-raspyrfm/
# board = {'isHighPower': False, 'interruptPin': 22, 'resetPin': None, 'spiDevice': 0}

# The following are for an RaspyRFM RFM69CW Module #2
# http://www.seegel-systeme.de/2015/09/02/ein-funkmodul-fuer-den-raspberry-raspyrfm/
# board = {'isHighPower': False, 'interruptPin': 18, 'resetPin': None, 'spiDevice': 1}

with Radio(FREQ_915MHZ, node_id, network_id, verbose=False, **board) as radio:
    print ("Starting loop...")

    while True:
        startTime = time.time()
        # Get packets for at most 5 seconds
        while time.time() - startTime < 5:
            # We end at (startTime+5), so we have (startTime+5 - time.time())
```

(continues on next page)

(continued from previous page)

```
            # seconds left
            timeRemaining = max(0, startTime + 5 - time.time())

            # This call will block until a packet is received,
            # or timeout in however much time we have left
            packet = radio.get_packet(timeout = timeRemaining)

            # If radio.get_packet times out, it will return None
            if packet is not None:
                # Process packet
                print (packet)

        # After 5 seconds send a message
        print ("Sending")
        if radio.send(recipient_id, "TEST", attempts=3, waitTime=100):
            print ("Acknowledgement received")
        else:
            print ("No Acknowledgement")
```

Better Transceiver

Here's an even better way to do deal with sending and receiving packets: start up a separate thread to handle incoming packets. Now we've eliminated the complex timing code altogether, and the result is much more readable.

```
# pylint: disable=missing-function-docstring,unused-import,redefined-outer-name

import time
import threading
from RFM69 import Radio, FREQ_315MHZ, FREQ_433MHZ, FREQ_868MHZ, FREQ_915MHZ

node_id = 1
network_id = 100
recipient_id = 2

# We'll run this function in a separate thread
def receiveFunction(radio):
    while True:
        # This call will block until a packet is received
        packet = radio.get_packet()
        print("Got a packet: ", end="")
        # Process packet
        print(packet)

# The following are for an Adafruit RFM69HCW Transceiver Radio
# Bonnet https://www.adafruit.com/product/4072
# You should adjust them to whatever matches your radio
with Radio(FREQ_915MHZ, node_id, network_id, isHighPower=True, verbose=False,
           interruptPin=15, resetPin=22, spiDevice=1) as radio:
    print ("Starting loop...")

    # Create a thread to run receiveFunction in the background and start it
    receiveThread = threading.Thread(target = receiveFunction, args=(radio,))
    receiveThread.start()

    while True:
```

(continues on next page)

(continued from previous page)

```
        # After 5 seconds send a message
        time.sleep(5)
        print ("Sending")
        if radio.send(recipient_id, "TEST", attempts=3, waitTime=100):
            print ("Acknowledgement received")
        else:
            print ("No Acknowledgement")
```

3.3.2 Example - Asyncio

A common requirement is to forward received RFM69 packets onward to a web API. However HTTP requests can be slow and we need to consider how to manage possible delays. If we block the radio receiver loop while making the necessary HTTP request, then time critical messages will be forced to wait!

We could solve this problem in a number of ways. In this example we are going to use Asyncio. It's worth mentioning here that although Asyncio is often touted as the wonder child of Python 3, asynchronous processing is not new to Python. More importantly, Asyncio is not a silver bullet and depending on your task may not be the best solution. I am not going to go over old ground talking about the pros and cons of async vs sync or concurrency and parallelism, Abu Ashraf Masnun has a nice article called Async Python: The Different Forms of Concurrency which I think covers this topic well.

Install the additional dependencies

```
pip install aiohttp cchardet aiodns
```

Asyncio RESTful API Gateway

The destination url is set to http://httpbin.org/post. This is a free online service which will echo back the post data sent to the service. It has a whole host (pardon the pun) of other tools for testing HTTP clients.

```
# pylint: disable=missing-function-docstring,redefined-outer-name

import asyncio
from aiohttp import ClientSession
from RFM69 import Radio, FREQ_433MHZ

async def call_API(url, packet):
    async with ClientSession() as session:
        print("Sending packet to server")
        async with session.post(url, json=packet.to_dict('%c')) as response:
            response = await response.read()
            print("Server responded", response)

async def receiver(radio):
    while True:
        print("Receiver")
        for packet in radio.get_packets():
            print("Packet received", packet.to_dict())
            await call_API("http://httpbin.org/post", packet)
        await asyncio.sleep(10)
```

(continues on next page)

(continued from previous page)

```
async def send(radio, to, message):
    print ("Sending")
    if radio.send(to, message, attempts=3, waitTime=100):
        print ("Acknowledgement received")
    else:
        print ("No Acknowledgement")

async def pinger(radio):
    print("Pinger")
    counter = 0
    while True:
        await send(radio, 2, "ping {}".format(counter))
        counter += 1
        await asyncio.sleep(5)


loop = asyncio.get_event_loop()
node_id = 1
network_id = 100
with Radio(FREQ_433MHZ, node_id, network_id, isHighPower=True, verbose=False) as␣
↪radio:
    print ("Started radio")
    loop.create_task(receiver(radio))
    loop.create_task(pinger(radio))
    loop.run_forever()

loop.close()
```

3.3.3 Example - Arduino Test Sketch

This Arduino sketch can be used to test the TX and RX of the radio. Make sure to uncomment the right frequency for your radio.

```
// **********************************************************************************
//
// Test RFM69 Radio.
//
// **********************************************************************************

#include <RFM69.h>              // https://www.github.com/lowpowerlab/rfm69
#include <RFM69_ATC.h>          // https://www.github.com/lowpowerlab/rfm69
#include <LowPower.h>           // https://github.com/LowPowerLab/LowPower
#include <SPI.h>                // Included with Arduino IDE

// Node and network config
#define NODEID        2    // The ID of this node (must be different for every node␣
↪on network)
#define NETWORKID     100  // The network ID

// Are you using the RFM69 Wing? Uncomment if you are.
//#define USING_RFM69_WING

// The transmision frequency of the board. Change as needed.
//#define FREQUENCY      RF69_433MHZ
//#define FREQUENCY      RF69_868MHZ
```

(continues on next page)

(continued from previous page)

```
//#define FREQUENCY      RF69_915MHZ

// Uncomment if this board is the RFM69HW/HCW not the RFM69W/CW
//#define IS_RFM69HW_HCW

// Serial board rate - just used to print debug messages
#define SERIAL_BAUD   57600

// Board and radio specific config - You should not need to edit
#if defined (__AVR_ATmega32U4__) && defined (USING_RFM69_WING)
#define RF69_SPI_CS  10
#define RF69_RESET   11
#define RF69_IRQ_PIN 2
#elif defined (__AVR_ATmega32U4__)
#define RF69_RESET    4
#define RF69_SPI_CS   8
#define RF69_IRQ_PIN  7
#elif defined(ARDUINO_SAMD_FEATHER_M0) && defined (USING_RFM69_WING)
#define RF69_RESET    11
#define RF69_SPI_CS   10
#define RF69_IRQ_PIN  6
#elif defined(ARDUINO_SAMD_FEATHER_M0)
#define RF69_RESET    4
#define RF69_SPI_CS   8
#define RF69_IRQ_PIN  3
#endif


RFM69 radio(RF69_SPI_CS, RF69_IRQ_PIN, false);

void setup() {
  Serial.begin(SERIAL_BAUD);

  // Initialize the radio
  radio.initialize(FREQUENCY, NODEID, NETWORKID);
#if defined(RF69_LISTENMODE_ENABLE)
  radio.listenModeEnd();
#endif

#ifdef IS_RFM69HW_HCW
  radio.setHighPower(); //must include this only for RFM69HW/HCW!
#endif

  Serial.println("Setup complete");
  Serial.println();
}


// Main loop
unsigned long previousMillis = 0;
const long sendInterval = 3000;
char* data = null;
uint8_t datalen = 0;

void loop() {
  // Receive
  if (radio.receiveDone()) {
```

(continues on next page)

(continued from previous page)

```
    getMessage(data, datalen);
    if (radio.ACKRequested()) {
      radio.sendACK(radio.SENDERID);
    }
    delay(100);
  }


  // Send
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= sendInterval) {
    previousMillis = currentMillis;

    Serial.println("Sending");
    char payload[] = "hello from test node";
    if (radio.sendWithRetry(1, payload, sizeof(payload), 3, 200)) {
      Serial.println("ACK received");
    } else {
      Serial.println("No ACK");
    }
  }
}

bool getMessage(char*& data, uint8_t& datalen) {
  if (data != null) {
    delete data;
    data = null;
  }
  datalen = 0;
  if (radio.DATALEN > 0 && radio.DATA != null) {
    datalen = radio.DATALEN;
    data = new char[datalen];
    memcpy(data, radio.DATA, datalen);
    Serial.println("Received message '" + bufferToString(data, datalen) + "' of␣
↪length " + String(datalen, DEC));
  }
  return data != null;
}

String bufferToString(char* data, uint8_t datalen) {
  bool all_ascii = true;
  String result = String("");
  for (uint8_t i = 0; i < datalen; i++) all_ascii &= isAscii(data[i]);

  for (uint8_t i = 0; i < datalen; i++) {
    result += all_ascii ? String((char)data[i]) : (String(data[i] < 16 ? "0" : "") +␣
↪String((uint8_t)data[i], HEX) + String(" "));
  }

  return result;
}
```

3.4 API Documentation

3.4.1 Radio

class RFM69.**Radio**(*freqBand*, *nodeID*, *networkID=100*, ***kwargs*)

RFM69 Radio interface for the Raspberry PI.

An RFM69 module is expected to be connected to the SPI interface of the Raspberry Pi. The class is as a context manager so you can instantiate it using the 'with' keyword.

Parameters

- **freqBand** – Frequency band of radio - 315MHz, 868Mhz, 433MHz or 915MHz.
- **nodeID** (*int*) – The node ID of this device.
- **networkID** (*int*) – The network ID

Keyword Arguments

- **auto_acknowledge** (*bool*) – Automatically send acknowledgements
- **isHighPower** (*bool*) – Is this a high power radio model
- **power** (*int*) – Power level - a percentage in range 10 to 100.
- **use_board_pin_numbers** (*bool*) – Use BOARD (not BCM) pin numbers. Defaults to True.
- **interruptPin** (*int*) – Pin number of interrupt pin. This is a pin index not a GPIO number.
- **resetPin** (*int*) – Pin number of reset pin. This is a pin index not a GPIO number.
- **spiBus** (*int*) – SPI bus number.
- **spiDevice** (*int*) – SPI device number.
- **promiscuousMode** (*bool*) – Listen to all messages not just those addressed to this node ID.
- **enableATC** (*bool*) – Enable ATC mode. Defaults to False.
- **encryptionKey** (*str*) – 16 character encryption key.
- **verbose** (*bool*) – Verbose mode - Activates logging to console.

__init__(*freqBand*, *nodeID*, *networkID=100*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

begin_receive()

Begin listening for packets

broadcast(*buff=''*)

Broadcast a message to network

Parameters **buff** (*str*) – Message buffer to send

calibrate_radio()

Calibrate the internal RC oscillator for use in wide temperature variations.

See RFM69 datasheet section [4.3.5. RC Timer Accuracy] for more information.

get_frequency_in_Hz()

Get the radio frequency in Hertz

get_packet(*block=True*, *timeout=None*)
Gets a single packet (thread-safe)

Parameters

- **block** (*bool*) – Block until a packet is available
- **timeout** (*int*) – Time to wait if blocking. Set to None to wait forever

Returns The oldest packet received if available, or None if no packet is available

Return type *Packet*

get_packets()
Get newly received packets.

Returns Returns a list of RFM69.Packet objects.

Return type list

has_received_packet()
Check if packet received

Returns True if packet has been received

Return type bool

listen_mode_send_burst(*toAddress*, *buff*)
Send a message to nodes in listen mode as a burst

Parameters

- **toAddress** (*int*) – Recipient node's ID
- **buff** (*str*) – Message buffer to send

listen_mode_set_durations(*rxDuration*, *idleDuration*)
Set the duty cycle for listen mode

The values used may be slightly different to accomodate what is allowed by the radio. This function returns the actual values used.

Parameters

- **rxDuration** (*int*) – number of microseconds to be in receive mode
- **idleDuration** (*int*) – number of microseconds to be sleeping

Returns the actual (rxDuration, idleDuration) used

Return type (int, int)

num_packets()
Returns the number of received packets

Returns Number of packets in the received queue

Return type int

read_registers()
Get all register values.

Returns Register values

Return type list

read_temperature(*calFactor=0*)
Read the temperature of the radios CMOS chip.

Parameters **calFactor** – Additional correction to corrects the slope, rising temp = rising val

Returns Temperature in centigrade

Return type int

send(*toAddress*, *buff=''*, ***kwargs*)
Send a message

Parameters

- **toAddress** (*int*) – Recipient node's ID
- **buff** (*str*) – Message buffer to send

Keyword Arguments

- **attempts** (*int*) – Number of attempts
- **wait** (*int*) – Milliseconds to wait for acknowledgement
- **require_ack** (*bool*) – Require Acknowledgement. If Attempts > 1 this is auto set to True.

Returns If acknowledgement received or None is no acknowledgement requested

Return type bool

send_ack(*toAddress*, *buff=[]*)
Send an acknowledgement packet

Parameters **toAddress** (*int*) – Recipient node's ID

set_frequency(*FRF*)
Set the radio frequency

set_frequency_in_Hz(*frequency_in_Hz*)
Set the radio frequency in Hertz

Parameters **frequency_in_Hz** (*int*) – Value between 315000000 to 915000000 Hz.

set_network(*network_id*)
Set the network ID (sync)

Parameters **network_id** (*int*) – Value between 1 and 254.

set_power_level(*percent*)
Set the transmit power level

Parameters **percent** (*int*) – Value between 0 and 100.

sleep()
Put the radio into sleep mode

3.4.2 Packet

class RFM69.**Packet**(*receiver*, *sender*, *RSSI*, *data*)
Object to represent received packet. Created internally and returned by radio when getPackets() is called.

Parameters

- **receiver** (*int*) – Node ID of receiver
- **sender** (*int*) – Node ID of sender

- **RSSI** (*int*) – Received Signal Strength Indicator i.e. the power present in a received radio signal
- **data** (*list*) – Raw transmitted data

__init__(*receiver*, *sender*, *RSSI*, *data*)
Initialize self. See help(type(self)) for accurate signature.

data_string
Returns the data as a string

to_dict(*dateFormat=None*)
Returns a dictionary representation of the class data

Index

Symbols

B

C

D

G

H

L

N

P

R

S

T