Adafruit RFM9x Library Documentation

Release 1.0

Tony DiCola

Dec 24, 2020

Contents

CircuitPython module for the RFM95/6/7/8 LoRa 433/915mhz radio modules.

CHAPTER 1

Dependencies

This driver depends on:

- Adafruit CircuitPython
- Bus Device

Please ensure all dependencies are available on the CircuitPython filesystem. This is easily achieved by downloading the Adafruit library and driver bundle.

CHAPTER 2

Installing from PyPI

On supported GNU/Linux systems like the Raspberry Pi, you can install the driver locally from PyPI. To install for current user:

```
pip3 install adafruit-circuitpython-rfm9x
```

To install system-wide (this may be required in some cases):

```
sudo pip3 install adafruit-circuitpython-rfm9x
```

To install in a virtual environment in your current project:

```
mkdir project-name && cd project-name
python3 -m venv .env
source .env/bin/activate
pip3 install adafruit-circuitpython-rfm9x
```

CHAPTER 3

Usage Example

Initialization of the RFM radio requires specifying a frequency appropriate to your radio hardware (i.e. 868-915 or 433 MHz) and specifying the pins used in your wiring from the controller board to the radio module.

This example code matches the wiring used in the LoRa and LoRaWAN Radio for Raspberry Pi project:

```
import digitalio
import board
import busio
import adafruit_rfm9x

RADIO_FREQ_MHZ = 915.0
CS = digitalio.DigitalInOut(board.CE1)
RESET = digitalio.DigitalInOut(board.D25)
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, RADIO_FREQ_MHZ)
```

Note: the default baudrate for the SPI is 50000000 (5MHz). The maximum setting is 10Mhz but transmission errors have been observed expecially when using breakout boards. For breakout boards or other configurations where the boards are separated, it may be necessary to reduce the baudrate for reliable data transmission. The baud rate may be specified as an keyword parameter when initializing the board. To set it to 1000000 use :

```
# Initialze RFM radio with a more conservative baudrate
rfm9x = adafruit_rfm9x.RFM9x(spi, CS, RESET, RADIO_FREQ_MHZ, baudrate=1000000)
```

Optional controls exist to alter the signal bandwidth, coding rate, and spreading factor settings used by the radio to achieve better performance in different environments. By default, settings compatible with RadioHead Bw125Cr45Sf128 mode are used, which can be altered in the following manner (continued from the above example):

```
# Apply new modem config settings to the radio to improve its effective range
rfm9x.signal_bandwidth = 62500
rfm9x.coding_rate = 6
rfm9x.spreading_factor = 8
rfm9x.enable_crc = True
```

See examples/rfm9x_simpletest.py for an expanded demo of the usage.

CHAPTER 4

Contributing

Contributions are welcome! Please read our Code of Conduct before contributing to help this project stay welcoming.

CHAPTER 5

Documentation

For information on building library documentation, please check out this guide.

CHAPTER 6

Table of Contents

6.1 Simple test

Ensure your device works with this simple test.

Listing 1: examples/rfm9x_simpletest.py

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

import adafruit_rfm9x


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

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

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

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

6.2 adafruit_rfm9x

CircuitPython module for the RFM95/6/7/8 LoRa 433/915mhz radio modules. This is adapted from the Radiohead library RF95 code from: http: www.airspayce.com/mikem/arduino/RadioHead/

- Author(s): Tony DiCola, Jerry Needell

class adafruit_rfm9x.**RFM9x**(*spi*, *cs*, *reset*, *frequency*, ***, *preamble_length=8*, *high_power=True*, *baudrate=5000000*, *agc=False*)

Interface to a RFM95/6/7/8 LoRa radio module. Allows sending and receivng bytes of data in long range LoRa mode at a support board frequency (433/915mhz).

You must specify the following parameters: - spi: The SPI bus connected to the radio. - cs: The CS pin DigitalInOut connected to the radio. - reset: The reset/RST pin DigialInOut connected to the radio. - frequency: The frequency (in mhz) of the radio module (433/915mhz typically).

You can optionally specify: - preamble_length: The length in bytes of the packet preamble (default 8). - high_power: Boolean to indicate a high power board (RFM95, etc.). Default is True for high power. - baudrate: Baud rate of the SPI connection, default is 10mhz but you might choose to lower to 1mhz if using long wires or a breadboard. - agc: Boolean to Enable/Disable Automatic Gain Control - Default=False (AGC off) Remember this library makes a best effort at receiving packets with pure Python code. Trying to receive packets too quickly will result in lost data so limit yourself to simple scenarios of sending and receiving single packets at a time.

Also note this library tries to be compatible with raw RadioHead Arduino library communication. This means the library sets up the radio modulation to match RadioHead's defaults and assumes that each packet contains a 4 byte header compatible with RadioHead's implementation. Advanced RadioHead features like address/node specific packets or "reliable datagram" delivery are supported however due to the limitations noted, "reliable datagram" is still subject to missed packets but with it, sender is notified if a packet has potentially been missed.

ack_delay = None
The delay time before attemting to send an ACK. If ACKs are being missed try setting this to .1 or .2.

ack_retries = None
The number of ACK retries before reporting a failure.

ack_wait = None
The delay time before attempting a retry after not receiving an ACK

auto_agc
Automatic Gain Control state

coding_rate
The coding rate used by the radio to control forward error correction (try setting to a higher value to increase tolerance of short bursts of interference or to a lower value to increase bit rate). Valid values are limited to 5, 6, 7, or 8.

crc_error()
crc status

destination = None
The default destination address for packet transmissions. (0-255). If 255 (0xff) then any receiving node should accept the packet. Second byte of the RadioHead header.

enable_crc
Set to True to enable hardware CRC checking of incoming packets. Incoming packets that fail the CRC check are not processed. Set to False to disable CRC checking and process all incoming packets.

flags = None
Upper 4 bits reserved for use by Reliable Datagram Mode. Lower 4 bits may be used to pass information. Fourth byte of the RadioHead header.

frequency_mhz
The frequency of the radio in Megahertz. Only the allowed values for your radio must be specified (i.e. 433 vs. 915 mhz)!

identifier = None
Automatically set to the sequence number when send_with_ack() used. Third byte of the RadioHead header.

idle()
Enter idle standby mode.

last_rssi = None
The RSSI of the last received packet. Stored when the packet was received. The instantaneous RSSI value may not be accurate once the operating mode has been changed.

listen()
Listen for packets to be received by the chip. Use *receive()* to listen, wait and retrieve packets as they're available.

node = None
The default address of this Node. (0-255). If not 255 (0xff) then only packets address to this node will be accepted. First byte of the RadioHead header.

preamble_length
The length of the preamble for sent and received packets, an unsigned 16-bit value. Received packets must match this length or they are ignored! Set to 8 to match the RadioHead RFM95 library.

receive(*, *keep_listening=True*, *with_header=False*, *with_ack=False*, *timeout=None*)
Wait to receive a packet from the receiver. If a packet is found the payload bytes are returned, otherwise None is returned (which indicates the timeout elapsed with no reception). If keep_listening is True (the default) the chip will immediately enter listening mode after reception of a packet, otherwise it will fall back to idle mode and ignore any future reception. All packets must have a 4-byte header for compatibilty with the RadioHead library. The header consists of 4 bytes (To,From,ID,Flags). The default setting will strip the header before returning the packet to the caller. If with_header is True then the 4 byte header will be returned with the packet. The payload then begins at packet[4]. If with_ack is True, send an ACK after receipt (Reliable Datagram mode)

receive_timeout = None
The amount of time to poll for a received packet. If no packet is received, the returned packet will be None

reset()
Perform a reset of the chip.

rssi
The received strength indicator (in dBm) of the last received message.

rx_done()
Receive status

send(*data*, *, *keep_listening=False*, *destination=None*, *node=None*, *identifier=None*, *flags=None*)
Send a string of data using the transmitter. You can only send 252 bytes at a time (limited by chip's FIFO size and appended headers). This appends a 4 byte header to be compatible with the RadioHead library. The header defaults to using the initialized attributes: (destination,node,identifier,flags) It may be temporarily overidden via the kwargs - destination,node,identifier,flags. Values passed via kwargs do not alter the attribute settings. The keep_listening argument should be set to True if you want to start listening automatically after the packet is sent. The default setting is False.

Returns: True if success or False if the send timed out.

send_with_ack(*data*)
Reliable Datagram mode: Send a packet with data and wait for an ACK response. The packet header is automatically generated. If enabled, the packet transmission will be retried on failure

signal_bandwidth
The signal bandwidth used by the radio (try setting to a higher value to increase throughput or to a

lower value to increase the likelihood of successfully received payloads). Valid values are listed in RFM9x.bw_bins.

sleep()
Enter sleep mode.

spreading_factor
The spreading factor used by the radio (try setting to a higher value to increase the receiver's ability to distinguish signal from noise or to a lower value to increase the data transmission rate). Valid values are limited to 6, 7, 8, 9, 10, 11, or 12.

transmit()
Transmit a packet which is queued in the FIFO. This is a low level function for entering transmit mode and more. For generating and transmitting a packet of data use *send()* instead.

tx_done()
Transmit status

tx_power
The transmit power in dBm. Can be set to a value from 5 to 23 for high power devices (RFM95/96/97/98, high_power=True) or -1 to 14 for low power devices. Only integer power levels are actually set (i.e. 12.5 will result in a value of 12 dBm). The actual maximum setting for high_power=True is 20dBm but for values > 20 the PA_BOOST will be enabled resulting in an additional gain of 3dBm. The actual setting is reduced by 3dBm. The reported value will reflect the reduced setting.

xmit_timeout = None
The amount of time to wait for the HW to transmit the packet. This is mainly used to prevent a hang due to a HW issue

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index