nRF24L01 Library Documentation

Release 1.2

Brendan Doherty

Oct 16, 2020

Introduction

CHAPTER 1

Getting Started

This is a Circuitpython driver library for the nRF24L01(+) transceiver.

Originally this code was a Micropython module written by Damien P. George & Peter Hinch which can still be found here

The Micropython source has since been rewritten to expose all the nRF24L01's features and for Circuitpython compatible devices (including linux-based SoC computers like the Raspberry Pi). Modified by Brendan Doherty & Rhys Thomas.

- Authors: Damien P. George, Peter Hinch, Rhys Thomas, Brendan Doherty

1.1 Features currently supported

- Change the address's length (can be 3 to 5 bytes long)
- Dynamically sized payloads (max 32 bytes each) or statically sized payloads
- Automatic responding acknowledgment (ACK) packets for verifying transmission success
- Append custom payloadsto the acknowledgment (ACK) packets for instant bi-directional communication
- Mark a single payload for no acknowledgment (ACK) from the receiving nRF24L01 (see `ask_no_ack` parameter for *`send()`* and *`write()`* functions)
- Invoke the "re-use the same payload" feature (for manually re-transmitting failed transmissions that remain in the TX FIFO buffer)
- Multiple payload transmissions with one function call (see documentation on the *`send()`* function and try out the Stream example)
- Context manager compatible for easily switching between different radio configurations using `The with statement` blocks (not available in `rf24_lite.py` version)
- Configure the interrupt (IRQ) pin to trigger (active low) on received, sent, and/or failed transmissions (these 3 events control 1 IRQ pin). There's also virtual representations of these interrupt events available (see *`irq_dr`*, *`irq_ds`*, & *`irq_df`* attributes)

- Invoke sleep mode (AKA power down mode) for ultra-low current consumption
- cyclic redundancy checking (CRC) up to 2 bytes long
- Adjust the nRF24L01's builtin automatic re-transmit feature's parameters (`arc`: number of attempts, `ard`: delay between attempts)
- Adjust the nRF24L01's frequency channel (2.4-2.525 GHz)
- Adjust the nRF24L01's power amplifier level (0, -6, -12, or -18 dBm)
- Adjust the nRF24L01's RF data rate (250kbps, 1Mbps, or 2Mbps)
- An nRF24L01 driven by this library can communicate with a nRF24L01 on an Arduino driven by the TMRh20 RF24 library. See the nrf24l01_2arduino_handling_data.py example.
- fake BLE module for sending BLE beacon advertisments from the nRF24L01 as outlined by Dmitry Grinberg in his write-up (including C source code).
- Multiceiver™ mode (up to 6 TX nRF24L01 "talking" to 1 RX nRF24L01 simultaneously). See the Multiceiver Example

1.2 Dependencies

This driver depends on:

- Adafruit CircuitPython
- Bus Device (specifically the `spi_device`)

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

Note: This library supports Python 3.4 or newer, but Python 3.7 introduced the function time.monotonic_ns() which returns an arbitrary time "counter" as an `int` of nanoseconds. However, this function is not used in the example scripts for backward compatibility reasons. Instead, we used `monotonic()` which returns an arbitrary time "counter" as a `float` of seconds. CircuitPython firmware supports both functions as of v4.0.

1.3 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 circuitpython-nrf24l01
```

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

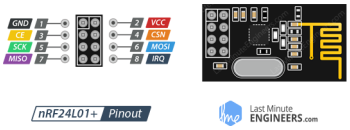
```
sudo pip3 install circuitpython-nrf24l01
```

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 circuitpython-nrf24l01
```

CHAPTER 2

Pinout



The nRF24L01 is controlled through SPI so there are 3 pins (SCK, MOSI, & MISO) that can only be connected to their counterparts on the MCU (microcontroller unit). The other 2 essential pins (CE & CSN) can be connected to any digital output pins. Lastly, the only optional pin on the nRf24L01 GPIOs is the IRQ (interrupt; a digital output that's active when low) pin and is only connected to the MCU via a digital input pin during the interrupt example. The following pinout is used in the example codes of this library's example directory.

nRF2401	Raspberry Pi	ItsyBitsy M4
GND	GND	GND
VCC	3V	3.3V
CE	GPIO4	D4
CSN	GPIO5	D5
SCK	GPIO11 (SCK)	SCK
MOSI	GPIO10 (MOSI)	MOSI
MISO	GPIO9 (MISO)	MISO
IRQ	GPIO12	D12

Tip: User reports and personal experiences have improved results if there is a capacitor of 100 mirofarads [+ another optional 0.1 microfarads capacitor for added stability] connected in parrallel to the VCC and GND pins.

CHAPTER 3

Using The Examples

See examples for testing certain features of this the library. The examples were developed and tested on both Raspberry Pi and ItsyBitsy M4. Pins have been hard coded in the examples for the corresponding device, so please adjust these accordingly to your circuitpython device if necessary.

To run the simple example, navigate to this repository's "examples" folder in the terminal. If you're working with a CircuitPython device (not a Raspberry Pi), copy the file named "nrf24l01_simple_test.py" from this repository's "examples" folder to the root directory of your CircuitPython device's CIRCUITPY drive. Now you're ready to open a python REPR and run the following commands:

```
>>> from nrf24l01_simple_test import *
    nRF24L01 Simple test.
    Run slave() on receiver
    Run master() on transmitter
>>> master()
Sending: 5 as struct: b'\x05\x00\x00\x00'
send() successful
Transmission took 36.0 ms
Sending: 4 as struct: b'\x04\x00\x00\x00'
send() successful
Transmission took 28.0 ms
Sending: 3 as struct: b'\x03\x00\x00\x00'
send() successful
Transmission took 24.0 ms
```

CHAPTER 4

What to purchase

See the store links on the sidebar or just google "nRF24L01+". It is worth noting that you generally want to buy more than 1 as you need 2 for testing – 1 to send & 1 to receive and vise versa. This library has been tested on a cheaply bought 6 pack from Amazon.com, but don't take Amazon or eBay for granted! There are other wireless transceivers that are NOT compatible with this library. For instance, the esp8266-01 (also sold in packs) is NOT compatible with this library, but looks very similar to the nRF24L01+ and could lead to an accidental purchase.

4.1 Power Stability

If you're not using a dedicated 3V regulator to supply power to the nRF24L01, then adding capcitor(s) (100 µF + an optional 0.1µF) in parrellel (& as close as possible) to the VCC and GND pins is highly recommended. Stablizing the power input provides significant performance increases. More finite details about the nRF24L01 are available from the datasheet (referenced here in the documentation as the nRF24L01+ Specification Sheet)

4.2 About the nRF24L01+PA+LNA modules

You may find variants of the nRF24L01 transceiver that are marketed as "nRF24L01+PA+LNA". These modules are distinct in the fact that they come with a detachable (SMA-type) antenna. They employ seperate RFX24C01 IC with the antenna for enhanced Power Amplification (PA) and Low Noise Amplification (LNA) features. While they boast greater range with the same functionality, they are subject to a couple lesser known (and lesser advertised) drawbacks:

1. Stronger power source. Below is a chart of advertised current requirements that many MCU boards' 3V regulators may not be able to provide (after supplying power to internal components).

Specification	Value
Emission mode current(peak)	115 mA
Receive Mode current(peak)	45 mA
Power-down mode current	4.2 µA

2. Needs shielding from electromagnetic interference. Shielding usually works best when it has a path to ground (GND pin), but this connection to the GND pin is not required.

See also the Testing nRF24L01+PA+LNA module

4.3 nRF24L01(+) clones and counterfeits

This library does not directly support clones/counterfeits as there is no way for the library to differentiate between an actual nRF24L01+ and a clone/counterfeit. To determine if your purchase is a counterfeit, please contact the retailer you purchased from (also reading this article and its links might help). The most notable clone is the Si24R1. I could not find the Si24R1 datasheet in english. Troubleshooting the SI24R1 may require replacing the onboard antenna with a wire. Furthermore, the Si24R1 has different power amplifier options as noted in the RF_PWR section (bits 0 through 2) of the RF_SETUP register (address 0x06) of the datasheet. While the options' values differ from those identified by this library's API, the underlying commands to configure those options are almost identical to the nRF24L01. Other known clones include the bk242x (also known as RFM7x).

CHAPTER 5

Contributing

Contributions are welcome! Please read our Code of Conduct before contributing to help this project stay welcoming. To contribute, all you need to do is fork this repository, develop your idea(s) and submit a pull request when stable. To initiate a discussion of idea(s), you need only open an issue on the aforementioned repository (doesn't have to be a bug report).

5.1 Future Project Ideas/Additions

The following are only ideas; they are not currently supported by this circuitpython library.

- There's a few blog posts by Nerd Ralph demonstrating how to use the nRF24L01 via 2 or 3 pins (uses custom bitbanging SPI functions and an external circuit involving a resistor and a capacitor)
- network linking layer, maybe something like TMRh20's RF24Network
- implement the Gazelle-based protocol used by the BBC micro-bit (makecode.com's radio blocks).

5.2 Sphinx documentation

Sphinx is used to build the documentation based on rST files and comments in the code. First, install dependencies (feel free to reuse the virtual environment from above):

```
python3 -m venv .env
source .env/bin/activate
pip install Sphinx sphinx-rtd-theme
```

Now, once you have the virtual environment activated:

```
cd docs
sphinx-build -E -W -b html . _build
```

This will output the documentation to `docs/_build`. Open the index.html in your browser to view them. It will also (due to -W) error out on any warning like the Github action, Build CI, does. This is a good way to locally verify it will pass.

CHAPTER 6

nRF24L01 Features

6.1 Simple test

Ensure your device works with this simple test.

Listing 1: examples/nrf24l01_simple_test.py

```
"""
Simple example of using the RF24 class.
"""
import time
import struct
import board
import digitalio as dio
# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# addresses needs to be in a buffer protocol object (bytearray)
address = b"1Node"

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# we'll be using the dynamic payload size feature (enabled by default)
# initialize the nRF24L01 on the spi bus object
nrf = RF24(spi, csn, ce)

# set the Power Amplifier level to -12 dBm since this test example is
```

(continues on next page)

(continued from previous page)

```
# usually run with nRF24L01 transceivers in close proximity
nrf.pa_level = -12


def master(count=5):  # count = 5 will only transmit 5 packets
    """Transmits an incrementing integer every second"""
    nrf.open_tx_pipe(address)  # set address of RX node into a TX pipe
    nrf.listen = False  # ensures the nRF24L01 is in TX mode

    while count:
        # use struct.pack to packetize your data
        # into a usable payload
        buffer = struct.pack("<i", count)
        # 'i' means a single 4 byte int value.
        # '<' means little endian byte order. this may be optional
        print("Sending: {} as struct: {}".format(count, buffer))
        start_timer = time.monotonic() * 1000  # start timer
        result = nrf.send(buffer)
        end_timer = time.monotonic() * 1000  # end timer
        if not result:
            print("send() failed or timed out")
        else:
            print("send() successful")
        # print timer results despite transmission success
        print("Transmission took", end_timer - start_timer, "ms")
        time.sleep(1)
        count -= 1


def slave(count=5):
    """Polls the radio and prints the received value. This method expires
    after 6 seconds of no received transmission"""
    # set address of TX node into an RX pipe. NOTE you MUST specify
    # which pipe number to use for RX, we'll be using pipe 0
    # pipe number options range [0,5]
    # the pipe numbers used during a transmission don't have to match
    nrf.open_rx_pipe(0, address)
    nrf.listen = True  # put radio into RX mode and power up

    start = time.monotonic()
    while count and (time.monotonic() - start) < 6:
        if nrf.update() and nrf.pipe is not None:
            # print details about the received packet
            print("{} bytes received on pipe {}".format(nrf.any(), nrf.pipe))
            # fetch 1 payload from RX FIFO
            rx = nrf.recv()  # also clears nrf.irq_dr status flag
            # expecting an int, thus the string format '<i'
            # the rx[:4] is just in case dynamic payloads were disabled
            buffer = struct.unpack("<i", rx[:4])  # [:4] truncates padded 0s
            # print the only item in the resulting tuple from
            # using `struct.unpack()`
            print("Received: {}, Raw: {}".format(buffer[0], rx))
            start = time.monotonic()
            count -= 1
            # this will listen indefinitely till count == 0

    # recommended behavior is to keep in TX mode while idle
```

(continues on next page)

(continued from previous page)

```
    nrf.listen = False  # put the nRF24L01 is in TX mode


print(
    """\
    nRF24L01 Simple test.\n\
    Run slave() on receiver\n\
    Run master() on transmitter"""
)
```

6.2 ACK Payloads Example

This is a test to show how to use custom acknowledgment payloads. See also documentation on *ack* and *load_ack()*.

Listing 2: examples/nrf24l01_ack_payload_test.py

```
"""
Simple example of using the library to transmit
and retrieve custom automatic acknowledgment payloads.
"""
import time
import board
import digitalio as dio
# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# we'll be using the dynamic payload size feature (enabled by default)
# the custom ACK payload feature is disabled by default
# the custom ACK payload feature should not be enabled
# during instantiation due to its singular use nature
# meaning 1 ACK payload per 1 RX'd payload
nrf = RF24(spi, csn, ce)

# NOTE the the custom ACK payload feature will be enabled
# automatically when you call load_ack() passing:
# a buffer protocol object (bytearray) of
# length ranging [1,32]. And pipe number always needs
# to be an int ranging [0,5]

# to enable the custom ACK payload feature
nrf.ack = True  # False disables again

# set the Power Amplifier level to -12 dBm since this test example is
# usually run with nRF24L01 transceivers in close proximity
```

(continues on next page)

(continued from previous page)

```
nrf.pa_level = -12

# addresses needs to be in a buffer protocol object (bytearray)
address = b"1Node"

# NOTE ACK payloads (like regular payloads and addresses)
# need to be in a buffer protocol object (bytearray)
ACK = b"World "


def master(count=5):  # count = 5 will only transmit 5 packets
    """Transmits a payload every second and prints the ACK payload"""
    nrf.listen = False  # put radio in TX mode
    # set address of RX node into a TX pipe
    nrf.open_tx_pipe(address)

    while count:
        buffer = b"Hello " + bytes([count + 48])  # output buffer
        print("Sent:", buffer, end=" ")
        start_timer = time.monotonic() * 1000  # start timer
        result = nrf.send(buffer)  # save the response (ACK payload)
        end_timer = time.monotonic() * 1000  # stop timer
        if not result:
            print("send() failed or timed out")
        else:
            # print the received ACK that was automatically
            # fetched and saved to "result" via send()
            print("Received:", result)
        # print timer results despite transmission success
        print("Transmission took", end_timer - start_timer, "ms")
        time.sleep(1)  # let the RX node prepare a new ACK payload
        count -= 1


def slave(count=5):
    """Prints the received value and sends an ACK payload"""
    # set address of TX node into an RX pipe. NOTE you MUST specify
    # which pipe number to use for RX; we'll be using pipe 0
    nrf.open_rx_pipe(0, address)
    # put radio into RX mode, power it up
    nrf.listen = True

    # setup the first transmission's ACK payload
    buffer = ACK + bytes([count + 48])
    # we must set the ACK payload data and corresponding
    # pipe number [0,5]
    nrf.load_ack(buffer, 0)  # load ACK for first response

    start = time.monotonic()  # start timer
    while count and (time.monotonic() - start) < 6:  # use 6 second timeout
        if nrf.update() and nrf.pipe is not None:
            count -= 1
            # retreive the received packet's payload
            rx = nrf.recv()  # clears flags & empties RX FIFO
            print("Received: {} Sent: {}".format(rx, buffer))
            start = time.monotonic()  # reset timer
            if count:  # Going again?
```

(continues on next page)

(continued from previous page)

```
                buffer = ACK + bytes([count + 48])  # build a new ACK
                nrf.load_ack(buffer, 0)  # load ACK for next response

    # recommended behavior is to keep in TX mode while idle
    nrf.listen = False  # put radio in TX mode
    nrf.flush_tx()  # flush any ACK payloads that remain


print(
    """\
    nRF24L01 ACK test\n\
    Run slave() on receiver\n\
    Run master() on transmitter"""
)
```

6.3 Multiceiver Example

This example shows how use a group of 6 nRF24L01 transceivers to transmit to 1 nRF24L01 transceiver. This technique is called "Multiceiver" in the nRF24L01 Specifications Sheet

---

Note: This example follows the diagram illistrated in figure 12 of section 7.7 of the nRF24L01 Specifications Sheet Please note that if `auto_ack` (on the base station) and `arc` (on the trnasmitting nodes) are disabled, then figure 10 of section 7.7 of the nRF24L01 Specifications Sheet would be a better illustration.

Hint: A paraphrased note from the the nRF24L01 Specifications Sheet:

Only when a data pipe receives a complete packet can other data pipes begin to receive data. When multiple [nRF24L01]s are transmitting to [one nRF24L01], the `ard` can be used to skew the auto retransmission so that they only block each other once.

This basically means that it might help packets get received if the `ard` attribute is set to various values among multiple transmitting nRF24L01 transceivers.

Listing 3: examples/nrf24l01_multiceiver_test.py

```
"""
Simple example of using 1 nRF24L01 to receive data from up to 6 other
transceivers. This technique is called "multiceiver" in the datasheet.
For fun, this example also sends an ACK payload from the base station
to the node-1 transmitter.
"""
import time
import board
import digitalio as dio

# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
```

(continues on next page)

(continued from previous page)

```
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# we'll be using the dynamic payload size feature (enabled by default)
# initialize the nRF24L01 on the spi bus object
nrf = RF24(spi, csn, ce)

# set the Power Amplifier level to -12 dBm since this test example is
# usually run with nRF24L01 transceivers in close proximity
nrf.pa_level = -12

# setup the addresses for all transmitting nRF24L01 nodes
addresses = [
    b"\x78" * 5,
    b"\xF1\xB3\xB4\xB5\xB6",
    b"\xCD\xB3\xB4\xB5\xB6",
    b"\xA3\xB3\xB4\xB5\xB6",
    b"\x0F\xB3\xB4\xB5\xB6",
    b"\x05\xB3\xB4\xB5\xB6"
]

# to use custom ACK payloads, we must enable that feature
nrf.ack = True
# let this be the ACK payload
ACK = b"Yak Back ACK"


def base(timeout=10):
    """Use the nRF24L01 as a base station for lisening to all nodes"""
    # write the addresses to all pipes.
    for pipe_n, addr in enumerate(addresses):
        nrf.open_rx_pipe(pipe_n, addr)
    while nrf.fifo(True, False):  # fill TX FIFO with ACK payloads
        nrf.load_ack(ACK, 1)  # only send ACK payload to node 1
    nrf.listen = True  # put base station into RX mode
    start_timer = time.monotonic()  # start timer
    while time.monotonic() - start_timer < timeout:
        while not nrf.fifo(False, True):  # keep RX FIFO empty for reception
            # show the pipe number that received the payload
            print("node", nrf.pipe, "sent:", nrf.recv())
            start_timer = time.monotonic()  # reset timer with every payload
            if nrf.load_ack(ACK, 1):  # keep TX FIFO full with ACK payloads
                print("\t ACK re-loaded")
    nrf.listen = False


def node(node_number, count=6):
    """start transmitting to the base station.

        :param int node_number: the node's identifying index (from the
            the `addresses` list)
        :param int count: the number of times that the node will transmit
            to the base station.
    """
```

(continues on next page)

(continued from previous page)

```
    nrf.listen = False
    # set the TX address to the address of the base station.
    nrf.open_tx_pipe(addresses[node_number])
    counter = 0
    # use the node_number to identify where the payload came from
    node_id = b"PTX-" + bytes([node_number + 48])
    while counter < count:
        counter += 1
        # payloads will include the node_number and a payload ID character
        payload = node_id + b" payload-ID: " + bytes([node_number + 48])
        payload += bytes([counter + (65 if 0 <= counter < 26 else 71)])
        # show something to see it isn't frozen
        print("attempt {} returned {}".format(counter, nrf.send(payload)))
        time.sleep(0.5)  # slow down the test for readability


print(
    """\
    nRF24L01 Multiceiver test.\n\
    Run base() on the receiver\n\
        base() sends ACK payloads to node 1\n\
    Run node(node_number) on a transmitter\n\
        node()'s parameter, `node_number`, must be in range [0, 5]"""
)
```

6.4 IRQ Pin Example

This is a test to show how to use nRF24L01's interrupt pin. Be aware that *send()* clears all IRQ events on exit, so we use the non-blocking *write()* instead. Also the *ack* attribute is enabled to trigger the *irq_dr* event when the master node receives ACK payloads. Simply put, this example is the most advanced example script (in this library), and it runs VERY quickly.

Listing 4: examples/nrf24l01_interrupt_test.py

```
"""
Simple example of detecting (and verifying) the IRQ (interrupt) pin on the
nRF24L01
    .. note:: this script uses the non-blocking `write()` function because
        the function `send()` clears the IRQ flags upon returning
"""
import time
import board
import digitalio as dio
# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# address needs to be in a buffer protocol object (bytearray is preferred)
address = b"1Node"

# select your digital input pin that's connected to the IRQ pin on the nRF4L01
irq_pin = dio.DigitalInOut(board.D12)
irq_pin.switch_to_input()  # make sure its an input object
# change these (digital output) pins accordingly
```

(continues on next page)

(continued from previous page)

```
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# we'll be using the dynamic payload size feature (enabled by default)
# initialize the nRF24L01 on the spi bus object
nrf = RF24(spi, csn, ce)

# this example uses the ACK payload to trigger the IRQ pin active for
# the "on data received" event
nrf.ack = True  # enable ACK payloads

# set the Power Amplifier level to -12 dBm since this test example is
# usually run with nRF24L01 transceivers in close proximity
nrf.pa_level = -12


def _ping_and_prompt():
    """transmit 1 payload, wait till irq_pin goes active, print IRQ status
    flags."""
    ce.value = 1  # tell the nRF24L01 to prepare sending a single packet
    time.sleep(0.00001)  # mandatory 10 microsecond pulse starts transmission
    ce.value = 0  # end 10 us pulse; use only 1 buffer from TX FIFO
    while irq_pin.value:  # IRQ pin is active when LOW
        pass
    print("IRQ pin went active LOW.")
    nrf.update()  # update irq_d? status flags
    print(
        "\tirq_ds: {}, irq_dr: {}, irq_df: {}".format(
            nrf.irq_ds, nrf.irq_dr, nrf.irq_df
        )
    )

def master():
    """Transmits 3 times: successfully receive ACK payload first, successfully
    transmit on second, and intentionally fail transmit on the third"""
    # set address of RX node into a TX pipe
    nrf.open_tx_pipe(address)
    # ensures the nRF24L01 is in TX mode
    nrf.listen = False
    # NOTE nrf.power is automatically set to True on first call to nrf.write()
    # NOTE nrf.write() internally calls nrf.clear_status_flags() first

    # load 2 buffers into the TX FIFO; write_only=True leaves CE pin LOW
    nrf.write(b"Ping ", write_only=True)
    nrf.write(b"Pong ", write_only=True)

    # on data ready test
    print("\nConfiguring IRQ pin to only ignore 'on data sent' event")
    nrf.interrupt_config(data_sent=False)
    print("    Pinging slave node for an ACK payload...", end=" ")
    _ping_and_prompt()  # CE pin is managed by this function
    if nrf.irq_dr:
        print("\t'on data ready' event test successful")
```

(continues on next page)

(continued from previous page)

```
    else:
        print("\t'on data ready' event test unsucessful")

    # on data sent test
    print("\nConfiguring IRQ pin to only ignore 'on data ready' event")
    nrf.interrupt_config(data_recv=False)
    print("    Pinging slave node again...             ", end=" ")
    _ping_and_prompt()  # CE pin is managed by this function
    if nrf.irq_ds:
        print("\t'on data sent' event test successful")
    else:
        print("\t'on data sent' event test unsucessful")

    # trigger slave node to exit by filling the slave node's RX FIFO
    print("\nSending one extra payload to fill RX FIFO on slave node.")
    if nrf.send(b"Radio", send_only=True):
        # when send_only parameter is True, send() ignores RX FIFO usage
        print("Slave node should not be listening anymore.")
    else:
        print("Slave node was unresponsive.")

    # on data fail test
    print("\nConfiguring IRQ pin to go active for all events.")
    nrf.interrupt_config()
    print("    Sending a ping to inactive slave node...", end=" ")
    nrf.flush_tx()  # just in case any previous tests failed
    nrf.write(b"Dummy", write_only=True)  # CE pin is left LOW
    _ping_and_prompt()  # CE pin is managed by this function
    if nrf.irq_df:
        print("\t'on data failed' event test successful")
    else:
        print("\t'on data failed' event test unsucessful")
    nrf.flush_tx()  # flush artifact payload in TX FIFO from last test
    # all 3 ACK payloads received were 4 bytes each, and RX FIFO is full
    # so, fetching 12 bytes from the RX FIFO also flushes RX FIFO
    print("\nComplete RX FIFO:", nrf.recv(12))


def slave(timeout=6):  # will listen for 6 seconds before timing out
    """Only listen for 3 payload from the master node"""
    # setup radio to recieve pings, fill TX FIFO with ACK payloads
    nrf.open_rx_pipe(0, address)
    nrf.load_ack(b"Yak ", 0)
    nrf.load_ack(b"Back", 0)
    nrf.load_ack(b" ACK", 0)
    nrf.listen = True  # start listening & clear irq_dr flag
    start_timer = time.monotonic()  # start timer now
    while not nrf.fifo(0, 0) and time.monotonic() - start_timer < timeout:
        # if RX FIFO is not full and timeout is not reached, then keep going
        pass
    nrf.listen = False  # put nRF24L01 in Standby-I mode when idling
    if not nrf.fifo(False, True):  # if RX FIFO is not empty
        # all 3 payloads received were 5 bytes each, and RX FIFO is full
        # so, fetching 15 bytes from the RX FIFO also flushes RX FIFO
        print("Complete RX FIFO:", nrf.recv(15))
    nrf.flush_tx()  # discard any pending ACK payloads

```

(continues on next page)

(continued from previous page)

```
135
136 print(
137     """\
138     nRF24L01 Interrupt pin test.\n\
139     Make sure the IRQ pin is connected to the MCU\n\
140     Run slave() on receiver\n\
141     Run master() on transmitter"""
142 )
```

CHAPTER 7

Library-Specific Features

7.1 Stream Example

This is a test to show how to stream data. The `master()` uses the *`send()`* to transmit multiple payloads with 1 function call. However `master()` only uses 1 level of the nRF24L01's TX FIFO. An alternate function, called `master_fifo()` uses all 3 levels of the nRF24L01's TX FIFO to stream data, but it uses the *`write()`* function to do so.

Listing 1: examples/nrf24l01_stream_test.py

```
"""
Example of library usage for streaming multiple payloads.
"""
import time
import board
import digitalio as dio
# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# addresses needs to be in a buffer protocol object (bytearray)
address = b"1Node"

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# we'll be using the dynamic payload size feature (enabled by default)
# initialize the nRF24L01 on the spi bus object
nrf = RF24(spi, csn, ce)
```

(continues on next page)

(continued from previous page)

```
25
26  # set the Power Amplifier level to -12 dBm since this test example is
27  # usually run with nRF24L01 transceivers in close proximity
28  nrf.pa_level = -12
29
30
31  def make_buffers(size=32):
32      """private function to construct a large list of payloads"""
33      buffers = []
34      # we'll use `size` for the number of payloads in the list and the
35      # payloads' length
36      for i in range(size):
37          # prefix payload with a sequential letter to indicate which
38          # payloads were lost (if any)
39          buff = bytes([i + (65 if 0 <= i < 26 else 71)])
40          for j in range(size - 1):
41              char = bool(j >= (size - 1) / 2 + abs((size - 1) / 2 - i))
42              char |= bool(j < (size - 1) / 2 - abs((size - 1) / 2 - i))
43              buff += bytes([char + 48])
44          buffers.append(buff)
45          del buff
46      return buffers
47
48
49  def master(count=1, size=32):  # count = 5 will transmit the list 5 times
50      """Transmits multiple payloads using `RF24.send()` and `RF24.resend()`."""
51      buffers = make_buffers(size)  # make a list of payloads
52      nrf.open_tx_pipe(address)  # set address of RX node into a TX pipe
53      nrf.listen = False  # ensures the nRF24L01 is in TX mode
54      successful = 0  # keep track of success rate
55      for _ in range(count):
56          start_timer = time.monotonic() * 1000  # start timer
57          # NOTE force_retry=2 internally invokes `RF24.resend()` 2 times at
58          # most for payloads that fail to transmit.
59          result = nrf.send(buffers, force_retry=2)  # result is a list
60          end_timer = time.monotonic() * 1000  # end timer
61          print("Transmission took", end_timer - start_timer, "ms")
62          for r in result:  # tally the resulting success rate
63              successful += 1 if r else 0
64      print(
65          "successfully sent {}% ({}/{})".format(
66              successful / (size * count) * 100,
67              successful,
68              size * count
69          )
70      )
71
72
73  def master_fifo(count=1, size=32):
74      """Similar to the `master()` above except this function uses the full
75      TX FIFO and `RF24.write()` instead of `RF24.send()`"""
76      if size < 6:
77          print("setting size to 6;", size, "is not allowed for this test.")
78          size = 6
79      buf = make_buffers(size)  # make a list of payloads
80      nrf.open_tx_pipe(address)  # set address of RX node into a TX pipe
81      nrf.listen = False  # ensures the nRF24L01 is in TX mode
```

(continues on next page)

(continued from previous page)

```
    for c in range(count):  # transmit the same payloads this many times
        nrf.flush_tx()  # clear the TX FIFO so we can use all 3 levels
        # NOTE the write_only parameter does not initiate sending
        buf_iter = 0  # iterator of payloads for the while loop
        failures = 0  # keep track of manual retries
        start_timer = time.monotonic() * 1000  # start timer
        while buf_iter < size:  # cycle through all the payloads
            while buf_iter < size and nrf.write(buf[buf_iter], write_only=1):
                # NOTE write() returns False if TX FIFO is already full
                buf_iter += 1  # increment iterator of payloads
            ce.value = True  # start tranmission (after 10 microseconds)
            while not nrf.fifo(True, True):  # updates irq_df flag
                if nrf.irq_df:
                    # reception failed; we need to reset the irq_rf flag
                    ce.value = False  # fall back to Standby-I mode
                    failures += 1  # increment manual retries
                    if failures > 99 and buf_iter < 7 and c < 2:
                        # we need to prevent an infinite loop
                        print(
                            "Make sure slave() node is listening."
                            " Quiting master_fifo()"
                        )
                        buf_iter = size + 1  # be sure to exit the while loop
                        nrf.flush_tx()  # discard all payloads in TX FIFO
                        break
                    nrf.clear_status_flags()  # clear the irq_df flag
                    ce.value = True  # start re-transmitting
            ce.value = False
        end_timer = time.monotonic() * 1000  # end timer
        print(
            "Transmission took {} ms with {} failures detected.".format(
                end_timer - start_timer,
                failures
            ),
            end=" " if failures < 100 else "\n"
        )
        if 1 <= failures < 100:
            print(
                "\n\nHINT: Try playing with the 'ard' and 'arc' attributes to"
                " reduce the number of\nfailures detected. Tests were better"
                " with these attributes at higher values, but\nnotice the "
                "transmission time differences."
            )
        elif not failures:
            print("You Win!")


def slave(timeout=5):
    """Stops listening after a `timeout` with no response"""
    # set address of TX node into an RX pipe. NOTE you MUST specify
    # which pipe number to use for RX, we'll be using pipe 0
    nrf.open_rx_pipe(0, address)  # pipe number options range [0,5]
    nrf.listen = True  # put radio into RX mode and power up
    count = 0  # keep track of the number of received payloads
    start_timer = time.monotonic()  # start timer
    while time.monotonic() < start_timer + timeout:
        if nrf.update() and nrf.pipe is not None:
```

(continues on next page)

(continued from previous page)

```
            count += 1
            # retreive the received packet's payload
            rx = nrf.recv()  # clears flags & empties RX FIFO
            print("Received: {} - {}".format(rx, count))
            start_timer = time.monotonic()  # reset timer on every RX payload

    # recommended behavior is to keep in TX mode while idle
    nrf.listen = False  # put the nRF24L01 is in TX mode


print(
    """\
    nRF24L01 Stream test\n\
    Run slave() on receiver\n\
    Run master() on transmitter to use 1 level of the TX FIFO\n\
    Run master_fifo() on transmitter to use all 3 levels of the TX FIFO."""
)
```

7.2 Context Example

This is a test to show how to use `The with statement` blocks to manage multiple different nRF24L01 configurations on 1 transceiver.

Listing 2: examples/nrf24l01_context_test.py

```
"""
Simple example of library usage in context.
This will not transmit anything, but rather
display settings after changing contexts ( & thus configurations)

    .. warning:: This script is not compatible with the rf24_lite module
"""
import board
import digitalio as dio
from circuitpython_nrf24l01.rf24 import RF24
from circuitpython_nrf24l01.fake_ble import FakeBLE

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# initialize the nRF24L01 objects on the spi bus object
# the first object will have all the features enabled
nrf = RF24(spi, csn, ce)
# enable the option to use custom ACK payloads
nrf.ack = True
# set the static payload length to 8 bytes
nrf.payload_length = 8
# RF power amplifier is set to -18 dbm
nrf.pa_level = -18
```

(continues on next page)

(continued from previous page)

```

# the second object has most features disabled/altered
ble = FakeBLE(spi, csn, ce)
# the IRQ pin is configured to only go active on "data fail"
# NOTE BLE operations prevent the IRQ pin going active on "data fail" events
ble.interrupt_config(data_recv=False, data_sent=False)
# using a channel 2
ble.channel = 2
# RF power amplifier is set to -12 dbm
ble.pa_level = -12

print("\nsettings configured by the nrf object")
with nrf:
    # only the first character gets written because it is on a pipe_number > 1
    nrf.open_rx_pipe(5, b"1Node")  # NOTE we do this inside the "with" block

    # display current settings of the nrf object
    nrf.what_happened(True)  # True dumps pipe info

print("\nsettings configured by the ble object")
with ble as nerf:  # the "as nerf" part is optional
    nerf.what_happened(1)

# if you examine the outputs from what_happened() you'll see:
#   pipe 5 is opened using the nrf object, but closed using the ble object.
#   pipe 0 is closed using the nrf object, but opened using the ble object.
#   also notice the different addresses bound to the RX pipes
# this is because the "with" statements load the existing settings
# for the RF24 object specified after the word "with".

# NOTE it is not advised to manipulate seperate RF24 objects outside of the
# "with" block; you will encounter bugs about configurations when doing so.
# Be sure to use 1 "with" block per RF24 object when instantiating multiple
# RF24 objects in your program.
# NOTE exiting a "with" block will always power down the nRF24L01
# NOTE upon instantiation, this library closes all RX pipes &
# extracts the TX/RX addresses from the nRF24L01 registers
```

CHAPTER 8

OTA compatibility

8.1 Fake BLE Example

This is a test to show how to use the nRF24L01 as a BLE advertising beacon using the `FakeBLE` class.

Listing 1: examples/nrf24l01_fake_ble_test.py

```
"""
This example uses the nRF24L01 as a 'fake' BLE Beacon

    .. warning:: ATSAMD21 M0-based boards have memory allocation
        error when loading 'fake_ble.mpy'
"""
import time
import board
import digitalio as dio
from circuitpython_nrf24l01.fake_ble import (
    chunk,
    FakeBLE,
    UrlServiceData,
    BatteryServiceData,
    TemperatureServiceData,
)

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# initialize the nRF24L01 on the spi bus object as a BLE compliant radio
nrf = FakeBLE(spi, csn, ce)
```

(continues on next page)

(continued from previous page)

```
28
29  # the name parameter is going to be its broadcasted BLE name
30  # this can be changed at any time using the `name` attribute
31  # nrf.name = b"foobar"
32
33  # you can optionally set the arbitrary MAC address to be used as the
34  # BLE device's MAC address. Otherwise this is randomly generated upon
35  # instantiation of the FakeBLE object.
36  # nrf.mac = b"\x19\x12\x14\x26\x09\xE0"
37
38  # set the Power Amplifier level to -12 dBm since this test example is
39  # usually run with nRF24L01 transceiver in close proximity to the
40  # BLE scanning application
41  nrf.pa_level = -12
42
43
44  def _prompt(count, iterator):
45      if (count - iterator) % 5 == 0 or (count - iterator) < 5:
46          if count - iterator - 1:
47              print(count - iterator, "advertisments left to go!")
48          else:
49              print(count - iterator, "advertisment left to go!")
50
51
52  # create an object for manipulating the battery level data
53  battery_service = BatteryServiceData()
54  # battery level data is 1 unsigned byte representing a percentage
55  battery_service.data = 85
56
57
58  def master(count=50):
59      """Sends out the device information twice a second."""
60      # using the "with" statement is highly recommended if the nRF24L01 is
61      # to be used for more than a BLE configuration
62      with nrf as ble:
63          ble.name = b"nRF24L01"
64          # include the radio's pa_level attribute in the payload
65          ble.show_pa_level = True
66          print(
67              "available bytes in next payload:",
68              ble.available(chunk(battery_service.buffer))
69          )  # using chunk() gives an accurate estimate of available bytes
70          for i in range(count):  # advertise data this many times
71              if ble.available(chunk(battery_service.buffer)) >= 0:
72                  _prompt(count, i)  # something to show that it isn't frozen
73                  # broadcast the device name, MAC address, &
74                  # battery charge info; 0x16 means service data
75                  ble.advertise(battery_service.buffer, data_type=0x16)
76                  # channel hoping is recommended per BLE specs
77                  ble.hop_channel()
78                  time.sleep(0.5)  # wait till next broadcast
79      # nrf.show_pa_level & nrf.name both are set to false when
80      # exiting a with statement block
81
82
83  # create an object for manipulating temperature measurements
84  temperature_service = TemperatureServiceData()
```

(continues on next page)

(continued from previous page)

```
# temperature's float data has up to 2 decimal places of percision
temperature_service.data = 42.0


def send_temp(count=50):
    """Sends out a fake temperature twice a second."""
    with nrf as ble:
        ble.name = b"nRF24L01"
        print(
            "available bytes in next payload:",
            ble.available(chunk(temperature_service.buffer))
        )
        for i in range(count):
            if ble.available(chunk(temperature_service.buffer)) >= 0:
                _prompt(count, i)
                # broadcast a temperature measurement; 0x16 means service data
                ble.advertise(temperature_service.buffer, data_type=0x16)
                ble.hop_channel()
                time.sleep(0.2)


# use the Eddystone protocol from Google to broadcast a URL as
# service data. We'll need an object to manipulate that also
url_service = UrlServiceData()
# the data attribute converts a URL string into a simplified
# bytes object using byte codes defined by the Eddystone protocol.
url_service.data = "http://www.google.com"
# Eddystone protocol requires an estimated TX PA level at 1 meter
# lower this estimate since we lowered the actual `ble.pa_level`
url_service.pa_level_at_1_meter = -45  # defaults to -25 dBm

def send_url(count=50):
    """Sends out a URL twice a second."""
    with nrf as ble:
        print(
            "available bytes in next payload:",
            ble.available(chunk(url_service.buffer))
        )
        # NOTE we did NOT set a device name in this with block
        for i in range(count):
            # URLs easily exceed the nRF24L01's max payload length
            if ble.available(chunk(url_service.buffer)) >= 0:
                _prompt(count, i)
                ble.advertise(url_service.buffer, 0x16)
                ble.hop_channel()
                time.sleep(0.2)

print(
    """\
    nRF24L01 fake BLE beacon test.\n\
    Run master() to broadcast the device name, pa_level, & battery charge\n\
    Run send_temp() to broadcast the device name & a temperature\n\
    Run send_url() to broadcast a custom URL link"""
)
```

8.2 TMRh20's Arduino library

This test is meant to prove compatibility with the popular Arduino library for the nRF24L01 by TMRh20 (available for install via the Arduino IDE's Library Manager). The following code has been designed/tested with the TMRh20 library example named GettingStarted_HandlingData.ino. If you changed the `radioNumber` variable in the TMRh20 sketch, you will have to adjust the `radioNumber` variable this script so that it is opposite the value in the TMRh20 library's example.

Listing 2: examples/nrf24l01_2arduino_handling_data.py

```
"""
Example of library driving the nRF24L01 to communicate with a nRF24L01 driven by
the TMRh20 Arduino library. The Arduino program/sketch that this example was
designed for is named GettingStarted_HandlingData.ino and can be found in the "RF24"
examples after the TMRh20 library is installed from the Arduino Library Manager.
"""
import time
import struct
import board
import digitalio as dio
# if running this on a ATSAMD21 M0 based board
# from circuitpython_nrf24l01.rf24_lite import RF24
from circuitpython_nrf24l01.rf24 import RF24

# addresses needs to be in a buffer protocol object (bytearray)
address = [b"1Node", b"2Node"]

# change these (digital output) pins accordingly
ce = dio.DigitalInOut(board.D4)
csn = dio.DigitalInOut(board.D5)

# using board.SPI() automatically selects the MCU's
# available SPI pins, board.SCK, board.MOSI, board.MISO
spi = board.SPI()  # init spi bus object

# initialize the nRF24L01 on the spi bus object
nrf = RF24(spi, csn, ce)
nrf.dynamic_payloads = False  # the default in the TMRh20 arduino library

# set the Power Amplifier level to -12 dBm since this test example is
# usually run with nRF24L01 transceivers in close proximity
nrf.pa_level = -12

# change this variable to oppose the corresponding variable in the
# TMRh20 library's GettingStarted_HandlingData.ino example
radioNumber = True


# Create a data structure for transmitting and receiving data
# pylint: disable=too-few-public-methods
class DataStruct:
    """A data structure to hold transmitted values as the
    'HandlingData' part of the TMRh20 library example"""
    time = 0  # in milliseconds (used as start of timer)
    value = 1.22  # incremented  by 0.01 with every transmission
# pylint: enable=too-few-public-methods

```

(continues on next page)

(continued from previous page)

```
myData = DataStruct()


def master(count=5):  # count = 5 will only transmit 5 packets
    """Transmits an arbitrary unsigned long value every second"""
    # set address of TX node into a RX pipe, and
    # set address of RX node into a TX pipe
    if radioNumber:
        nrf.open_rx_pipe(1, address[0])
        nrf.open_tx_pipe(address[1])
    else:
        nrf.open_rx_pipe(1, address[1])
        nrf.open_tx_pipe(address[0])
    nrf.listen = False  # ensures the nRF24L01 is in TX mode
    while count:
        print("Now Sending")
        myData.time = int(time.monotonic() * 1000)  # start timer
        # use struct.pack to packetize your data into a usable payload
        # '<' means little endian byte order.
        # 'L' means a single 4 byte unsigned long value.
        # 'f' means a single 4 byte float value.
        buffer = struct.pack("<Lf", myData.time, myData.value)
        # send the payload. send_only=1 is default behavior in TMRh20 library
        result = nrf.send(buffer, send_only=1)
        if not result:
            print("send() failed or timed out")
        else:  # sent successful; listen for a response
            nrf.listen = True  # get nRF24L01 ready to receive a response
            timeout = True  # used to determine if response timed out
            while time.monotonic() * 1000 - myData.time < 200:
                # the arbitrary 200 ms timeout value is also used in the
                # TMRh20 library's GettingStarted_HandlingData sketch
                if nrf.update() and nrf.pipe is not None:
                    end_timer = time.monotonic() * 1000  # end timer
                    rx = nrf.recv(32)  # 32 mimics behavior in TMRh20 library
                    rx = struct.unpack("<Lf", rx[:8])
                    myData.value = rx[1]  # save the new float value
                    timeout = False  # skips timeout prompt
                    # print total time to send and receive data
                    print(
                        "Sent {} Got Response: {}".format(
                            struct.unpack("<Lf", buffer),
                            rx
                        )
                    )
                    print("Round-trip delay:", end_timer - myData.time, "ms")
                    break
            if timeout:
                print("failed to get a response; timed out")
            nrf.listen = False  # put the nRF24L01 back in TX mode
        count -= 1
        time.sleep(1)


def slave(count=3):
    """Polls the radio and prints the received value. This method expires
    after 6 seconds of no received transmission"""
```

(continues on next page)

(continued from previous page)

```
    # set address of TX node into a RX pipe, and
    # set address of RX node into a TX pipe
    if radioNumber:
        nrf.open_rx_pipe(1, address[0])
        nrf.open_tx_pipe(address[1])
    else:
        nrf.open_rx_pipe(1, address[1])
        nrf.open_tx_pipe(address[0])
    nrf.listen = True  # put radio into RX mode and power up
    myData.time = time.monotonic() * 1000  # in milliseconds
    while count and (time.monotonic() * 1000 - myData.time) < 6000:
        if nrf.update() and nrf.pipe is not None:
            # clear flags & fetch 1 payload in RX FIFO
            buffer = nrf.recv(32)  # 32 mimics behavior in TMRh20 library
            # increment floating value as part of the "HandlingData" test
            myData.value = struct.unpack("<f", buffer[4:8])[0] + 0.01
            nrf.listen = False  # ensures the nRF24L01 is in TX mode
            myData.time = time.monotonic() * 1000
            # echo buffer[:4] appended with incremented float
            # send_only=True is the default behavior in TMRh20 library
            result = nrf.send(
                buffer[:4] + struct.pack("<f", myData.value),
                send_only=True
            )
            end_timer = time.monotonic() * 1000  # in milliseconds
            # expecting an unsigned long & a float, thus the
            # string format "<Lf"; buffer[:8] ignores the padded 0s
            buffer = struct.unpack("<L", buffer[:4])
            # print the unsigned long and float data sent in the response
            print("Responding: {}, {}".format(buffer[0], myData.value))
            if not result:
                print("response failed or timed out")
            else:
                # print timer results on transmission success
                print(
                    "successful response took {} ms".format(
                        end_timer - myData.time
                    )
                )
            # this will listen indefinitely till counter == 0
            count -= 1
            nrf.listen = True  # put nRF24L01 back into RX mode
    # recommended behavior is to keep in TX mode when in idle
    nrf.listen = False  # put the nRF24L01 in TX mode + Standby-I power state


print(
    """\
    nRF24L01 communicating with an Arduino running the\n\
    TMRh20 library's "GettingStarted_HandlingData.ino" example.\n\
    Run slave() to receive\n\
    Run master() to transmit\n\

    radioNumber is {}. Change this variable to oppose the TMRh20\n\
    example's radioNumber value.""".format(radioNumber)
)
```

CHAPTER 9

Troubleshooting info

Important: The nRF24L01 has 3 key features that can be interdependent of each other. Their priority of dependence is as follows:

1. *auto_ack* feature provides transmission verification by using the RX nRF24L01 to automatically and imediatedly send an acknowledgment (ACK) packet in response to received payloads. *auto_ack* does not require *dynamic_payloads* to be enabled.
2. *dynamic_payloads* feature allows either TX/RX nRF24L01 to be able to send/receive payloads with their size written into the payloads' packet. With this disabled, both RX/TX nRF24L01 must use matching *payload_length* attributes. For *dynamic_payloads* to be enabled, the *auto_ack* feature must be enabled. Although, the *auto_ack* feature can be used when the *dynamic_payloads* feature is disabled.
3. *ack* feature allows the MCU to append a payload to the ACK packet, thus instant bi-directional communication. A transmitting ACK payload must be loaded into the nRF24L01's TX FIFO buffer (done using *load_ack()*) BEFORE receiving the payload that is to be acknowledged. Once transmitted, the payload is released from the TX FIFO buffer. This feature requires the *auto_ack* and *dynamic_payloads* features enabled.

Remeber that the nRF24L01's FIFO (first-in,first-out) buffer has 3 levels. This means that there can be up to 3 payloads waiting to be read (RX) and up to 3 payloads waiting to be transmit (TX).

With the *auto_ack* feature enabled, you get:

- cyclic redundancy checking (*crc*) automatically enabled
- to change amount of automatic re-transmit attempts and the delay time between them. See the *arc* and *ard* attributes.

Note: A word on pipes vs addresses vs channels.

You should think of the data pipes as a "parking spot" for your payload. There are only six data pipes on the nRF24L01, thus it can simultaneously "listen" to a maximum of 6 other nRF24L01 radios. However, it can only "talk" to 1 other nRF24L01 at a time).

The specified address is not the address of an nRF24L01 radio, rather it is more like a path that connects the endpoints. When assigning addresses to a data pipe, you can use any 5 byte long address you can think of (as long as the first byte is unique among simultaneously broadcasting addresses), so you're not limited to communicating with only the same 6 nRF24L01 radios.

Finnaly, the radio's channel is not be confused with the radio's pipes. Channel selection is a way of specifying a certain radio frequency (frequency = [2400 + channel] MHz). Channel defaults to 76 (like the arduino library), but options range from 0 to 125 – that's 2.4 GHz to 2.525 GHz. The channel can be tweaked to find a less occupied frequency amongst Bluetooth, WiFi, or other ambient signals that use the same spectrum of frequencies.

Warning: For successful transmissions, most of the endpoint trasceivers' settings/features must match. These settings/features include:

- The RX pipe's address on the receiving nRF24L01 (passed to *open_rx_pipe()*) MUST match the TX pipe's address on the transmitting nRF24L01 (passed to *open_tx_pipe()*)
- *address_length*
- *channel*
- *data_rate*
- *dynamic_payloads*
- *payload_length* only when *dynamic_payloads* is disabled
- *auto_ack* on the recieving nRF24L01 must be enabled if *arc* is greater than 0 on the transmitting nRF24L01
- custom *ack* payloads
- *crc*

In fact the only attributes that aren't required to match on both endpoint transceivers would be the identifying data pipe number (passed to *open_rx_pipe()* or *load_ack()*), *pa_level*, *arc*, & *ard* attributes. The `ask_no_ack` feature can be used despite the settings/features configuration (see *send()* & *write()* function parameters for more details).

CHAPTER 10

About the lite version

This library contains a "lite" version of `rf24.py` titled `rf24_lite.py`. It has been developed to save space on microcontrollers with limited amount of RAM and/or storage (like boards using the ATSAMD21 M0). The following functionality has been removed from the lite version:

- The *FakeBLE* class is not compatible with the `rf24_lite.py` module.
- *is_plus_variant* is removed, meaning the lite version is not compatibility with the older non-plus variants of the nRF24L01.
- *address()* removed.
- *what_happened()* removed. However you can use the following function to dump all available registers' values (for advanced users):

```
# let `nrf` be the instantiated RF24 object
def dump_registers(end=0x1e):
    for i in range(end):
        if i in (0xA, 0xB, 0x10):
            print(hex(i), "=", nrf._reg_read_bytes(i))
        elif i not in (0x18, 0x19, 0x1a, 0x1b):
            print(hex(i), "=", hex(nrf._reg_read(i)))
```

- *dynamic_payloads* applies to all pipes, not individual pipes.
- *payload_length* applies to all pipes, not individual pipes.
- *read_ack()* removed. This is deprecated on next major release anyway; use *recv()* instead.
- *load_ack()* is available, but it will not throw exceptions for malformed `buf` or invalid `pipe_number` parameters.
- *crc* removed. 2-bytes encoding scheme (CRC16) is always enabled.
- *auto_ack* removed. This is always enabled for all pipes. Pass `ask_no_ack` parameter as `True` to *send()* or *write()* to disable automatic acknowledgement for TX operations.
- *is_lna_enabled* removed as it only affects non-plus variants of the nRF24L01.

- *pa_level* is available, but it will not accept a `list` or `tuple`.
- *rpd*, *start_carrier_wave()*, & *stop_carrier_wave()* removed. These only perform a test of the nRF24L01's hardware.
- *CSN_DELAY* removed. This is hard-coded to 5 milliseconds
- All comments and docstrings removed, meaning `help()` will not provide any specific information. Exception prompts have also been reduced and adjusted accordingly.
- Cannot switch between different radio configurations using context manager (the `The with statement` blocks). It is advised that only one *RF24* object be instantiated when RAM is limited (less than or equal to 32KB).

CHAPTER 11

Testing nRF24L01+PA+LNA module

The following are semi-successful test results using a nRF24L01+PA+LNA module:

11.1 The Setup

I wrapped the PA/LNA module with electrical tape and then foil around that (for shielding) while being very careful to not let the foil touch any current carrying parts (like the GPIO pins and the soldier joints for the antenna mount). Then I wired up a PA/LNA module with a 3V regulator (L4931 with a 2.2 µF capacitor between V_{out} & GND) using my ItsyBitsy M4 5V (USB) pin going directly to the L4931 V_{in} pin. The following are experiences from running simple, ack, & stream examples with a reliable nRF24L01+ (no PA/LNA) on the other end (driven by a Raspberry Pi 2):

11.2 Results (ordered by `pa_level` settings)

- 0 dBm: `master()` worked the first time (during simple example) then continuously failed (during all examples). `slave()` worked on simple & stream examples, but the opposing `master()` node reporting that ACK packets (without payloads) were **not** received from the PA/LNA module; `slave()` failed to send ACK packet payloads during the ack example.
- -6 dBm: `master()` worked consistently on simple, ack, & stream example. `slave()` worked reliably on simple & stream examples, but failed to transmit **any** ACK packet payloads in the ack example.
- -12 dBm: `master()` worked consistently on simple, ack, & stream example. `slave()` worked reliably on simple & stream examples, but failed to transmit **some** ACK packet payloads in the ack example.
- -18 dBm: `master()` worked consistently on simple, ack, & stream example. `slave()` worked reliably on simple, ack, & stream examples, meaning **all** ACK packet payloads were successfully transmit in the ack example.

I should note that without shielding the PA/LNA module and using the L4931 3V regulator, no TX transmissions got sent (including ACK packets for the `auto-ack` feature).

11.3 Conclusion

The PA/LNA modules seem to require quite a bit more power to transmit. The L4931 regulator that I used in the tests boasts a 300 mA current limit and a typical current of 250 mA. While the ItsyBitsy M4 boasts a 500 mA max, it would seem that much of that is consumed internally. Since playing with the *pa_level* is a current saving hack (as noted in the datasheet), I can only imagine that a higher power 3V regulator may enable sending transmissions (including ACK packets – with or without ACK payloads attached) from PA/LNA modules using higher *pa_level* settings. More testing is called for, but I don't have an oscilloscope to measure the peak current draws.

CHAPTER 12

Basic API

12.1 Constructor

class circuitpython_nrf24l01.rf24.**RF24**(*spi*, *csn*, *ce*, *spi_frequency=10000000*)

A driver class for the nRF24L01(+) transceiver radios.

This class aims to be compatible with other devices in the nRF24xxx product line that implement the Nordic proprietary Enhanced ShockBurst Protocol (and/or the legacy ShockBurst Protocol), but officially only supports (through testing) the nRF24L01 and nRF24L01+ devices.

Parameters

- **spi** (*SPI*) – The object for the SPI bus that the nRF24L01 is connected to.

  Tip: This object is meant to be shared amongst other driver classes (like adafruit_mcp3xxx.mcp3008 for example) that use the same SPI bus. Otherwise, multiple devices on the same SPI bus with different spi objects may produce errors or undesirable behavior.

- **csn** (*DigitalInOut*) – The digital output pin that is connected to the nRF24L01's CSN (Chip Select Not) pin. This is required.
- **ce** (*DigitalInOut*) – The digital output pin that is connected to the nRF24L01's CE (Chip Enable) pin. This is required.
- **spi_frequency** (*int*) – Specify which SPI frequency (in Hz) to use on the SPI bus. This parameter only applies to the instantiated object and is made persistent via SPIDevice.

12.2 open_tx_pipe()

RF24.**open_tx_pipe**(*address*)

This function is used to open a data pipe for OTA (over the air) TX transmissions.

Parameters **address** (*bytearray*, *bytes*) – The virtual address of the receiving nRF24L01. The address specified here must match the address set to one of the RX data pipes of the receiving nRF24L01. The existing address can be altered by writing a bytearray with a length less than 5. The nRF24L01 will use the first *address_length* number of bytes for the RX address on the specified data pipe.

Note: There is no option to specify which data pipe to use because the nRF24L01 only uses data pipe 0 in TX mode. Additionally, the nRF24L01 uses the same data pipe (pipe 0) for receiving acknowledgement (ACK) packets in TX mode when the *auto_ack* attribute is enabled for data pipe 0. Thus, RX pipe 0 is appropriated with the TX address (specified here) when *auto_ack* is enabled for data pipe 0.

12.3 close_rx_pipe()

RF24.**close_rx_pipe**(*pipe_number*)

This function is used to close a specific data pipe from OTA (over the air) RX transmissions.

Parameters **pipe_number** (*int*) – The data pipe to use for RX transactions. This must be in range [0, 5]. Otherwise a IndexError exception is thrown.

12.4 open_rx_pipe()

RF24.**open_rx_pipe**(*pipe_number*, *address*)

This function is used to open a specific data pipe for OTA (over the air) RX transmissions.

Parameters

- **pipe_number** (*int*) – The data pipe to use for RX transactions. This must be in range [0, 5]. Otherwise a IndexError exception is thrown.
- **address** (*bytearray*, *bytes*) – The virtual address to the receiving nRF24L01. If using a pipe_number greater than 1, then only the MSByte of the address is written, so make sure MSByte (first character) is unique among other simultaneously receiving addresses. The existing address can be altered by writing a bytearray with a length less than 5. The nRF24L01 will use the first *address_length* number of bytes for the RX address on the specified data pipe.

Note: The nRF24L01 shares the addresses' last 4 LSBytes on data pipes 2 through 5. These shared LSBytes are determined by the address set to data pipe 1.

12.5 listen

RF24.**listen**

An attribute to represent the nRF24L01 primary role as a radio. Setting this attribute incorporates the proper transitioning to/from RX mode as it involves playing with the *power* attribute and the nRF24L01's CE pin. This attribute does not power down the nRF24L01, but will power it up when needed; use *power* attribute set to False to put the nRF24L01 to sleep.

A valid input value is a bool in which:

- `True` enables RX mode. Additionally, per Appendix B of the nRF24L01+ Specifications Sheet, this attribute flushes the RX FIFO, clears the *irq_dr* status flag, and puts nRF24L01 in power up mode. Notice the CE pin is be held HIGH during RX mode.
- `False` disables RX mode. As mentioned in above link, this puts nRF24L01's power in Standby-I (CE pin is LOW meaning low current & no transmissions) mode which is ideal for post-reception work. Disabing RX mode doesn't flush the RX/TX FIFO buffers, so remember to flush your 3-level FIFO buffers when appropriate using *flush_tx()* or *flush_rx()* (see also the *recv()* function).

12.6 any()

RF24.**any**()

This function checks if the nRF24L01 has received any data at all, and then reports the next available payload's length (in bytes).

Returns

- `int` of the size (in bytes) of an available RX payload (if any).
- `0` if there is no payload in the RX FIFO buffer.

12.7 recv()

RF24.**recv**(*length=None*)

This function is used to retrieve the next available payload in the RX FIFO buffer, then clears the *irq_dr* status flag.

This function can also be used to fetch the last ACK packet's payload if *ack* is enabled.

Parameters **length** (*int*) – An optional parameter to specify how many bytes to read from the RX FIFO buffer. This parameter is not constrained in any way.

- If this parameter is less than the length of the first available payload in the RX FIFO buffer, then the payload will remain in the RX FIFO buffer until the entire payload is fetched by this function.
- If this parameter is greater than the next available payload's length, then additional data from other payload(s) in the RX FIFO buffer are returned.

Note: The nRF24L01 will repeatedly return the last byte fetched from the RX FIFO buffer when there is no data to return (even if the RX FIFO is empty). Be aware that a payload is only removed from the RX FIFO buffer when the entire payload has been fetched by this function. Notice that this function always starts reading data from the first byte of the first available payload (if any) in the RX FIFO buffer. Remember the RX FIFO buffer can hold up to 3 payloads at a maximum of 32 bytes each.

Returns

If the `length` parameter is not specified, then this function returns a `bytearray` of the RX payload data or `None` if there is no payload. This also depends on the setting of *dynamic_payloads* & *payload_length* attributes. Consider the following two scenarios:

- If the *dynamic_payloads* attribute is disabled, then the returned bytearray's length is equal to the user defined *payload_length* attribute for the data pipe that received the payload.
- If the *dynamic_payloads* attribute is enabled, then the returned bytearray's length is equal to the payload's length

When the `length` parameter is specified, this function strictly returns a `bytearray` of that length despite the contents of the RX FIFO.

12.8 send()

`RF24.`**`send`**(*buf*, *ask_no_ack=False*, *force_retry=0*, *send_only=False*)

This blocking function is used to transmit payload(s).

Returns

- `list` if a list or tuple of payloads was passed as the `buf` parameter. Each item in the returned list will contain the returned status for each corresponding payload in the list/tuple that was passed. The return statuses will be in one of the following forms:
- `False` if transmission fails. Transmission failure can only be detected if *arc* is greater than `0`.
- `True` if transmission succeeds.
- `bytearray` or `True` when the *ack* attribute is `True`. Because the payload expects a responding custom ACK payload, the response is returned (upon successful transmission) as a `bytearray` (or `True` if ACK payload is empty). Returning the ACK payload can be bypassed by setting the `send_only` parameter as `True`.

Parameters

- **`buf`** (*bytearray*, *bytes*, *list*, *tuple*) – The payload to transmit. This bytearray must have a length in range [1, 32], otherwise a `ValueError` exception is thrown. This can also be a list or tuple of payloads (`bytearray`); in which case, all items in the list/tuple are processed for consecutive transmissions.
  - If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is less than the *payload_length* attribute for pipe 0, then this bytearray is padded with zeros until its length is equal to the *payload_length* attribute for pipe 0.
  - If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is greater than *payload_length* attribute for pipe 0, then this bytearray's length is truncated to equal the *payload_length* attribute for pipe 0.
- **`ask_no_ack`** (*bool*) – Pass this parameter as `True` to tell the nRF24L01 not to wait for an acknowledgment from the receiving nRF24L01. This parameter directly controls a `NO_ACK` flag in the transmission's Packet Control Field (9 bits of information about the payload). Therefore, it takes advantage of an nRF24L01 feature specific to individual payloads, and its value is not saved anywhere. You do not need to specify this for every payload if the *arc* attribute is disabled, however setting this parameter to `True` will work despite the *arc* attribute's setting.

Note: Each transmission is in the form of a packet. This packet contains sections of data around and including the payload. See Chapter 7.3 in the nRF24L01 Specifications Sheet for more details.

- **force_retry** (*int*) – The number of brute-force attempts to *resend()* a failed transmission. Default is 0. This parameter has no affect on transmissions if *arc* is `0` or if `ask_no_ack` parameter is set to `True`. Each re-attempt still takes advantage of *arc* & *ard* attributes. During multi-payload processing, this parameter is meant to slow down CircuitPython devices just enough for the Raspberry Pi to catch up (due to the Raspberry Pi's seemingly slower SPI speeds).
- **send_only** (*bool*) – This parameter only applies when the *ack* attribute is set to `True`. Pass this parameter as `True` if the RX FIFO is not to be manipulated. Many other libraries' behave as though this parameter is `True` (e.g. The popular TMRh20 Arduino RF24 library). This parameter defaults to `False`. Use *recv()* to get the ACK payload (if there is any) from the RX FIFO.Remember that the RX FIFO can only hold up to 3 payloads at once.

Tip: It is highly recommended that *arc* attribute is enabled (greater than `0`) when sending multiple payloads. Test results with the *arc* attribute disabled were rather poor (less than 79% received by a Raspberry Pi). This same advice applies to the `ask_no_ack` parameter (leave it as `False` for multiple payloads).

Warning: The nRF24L01 will block usage of the TX FIFO buffer upon failed transmissions. Failed transmission's payloads stay in TX FIFO buffer until the MCU calls *flush_tx()* and *clear_status_flags()*. Therefore, this function will discard failed transmissions' payloads.

CHAPTER 13

Advanced API

13.1 what_happened()

RF24.**what_happened**(*dump_pipes=False*)

This debuggung function aggregates and outputs all status/condition related information from the nRF24L01.

Some information may be irrelevant depending on nRF24L01's state/condition.

Prints

- `Is a plus variant` True means the transceiver is a nRF24L01+. False means the transceiver is a nRF24L01 (not a plus variant).
- `Channel` The current setting of the *channel* attribute
- `RF Data Rate` The current setting of the RF *data_rate* attribute.
- `RF Power Amplifier` The current setting of the *pa_level* attribute.
- `CRC bytes` The current setting of the *crc* attribute
- `Address length` The current setting of the *address_length* attribute
- `TX Payload lengths` The current setting of the *payload_length* attribute for TX operations (concerning data pipe 0)
- `Auto retry delay` The current setting of the *ard* attribute
- `Auto retry attempts` The current setting of the *arc* attribute
- `Re-use TX FIFO` Are payloads in the TX FIFO to be re-used for subsequent transmissions (triggered by calling *resend()*)
- `Packets lost on current channel` Total amount of packets lost (transmission failures). This only resets when the *channel* is changed. This count will only go up to 15.
- `Retry attempts made for last transmission` Amount of attempts to re-transmit during last transmission (resets per payload)
- `IRQ - Data Ready` The current setting of the IRQ pin on "Data Ready" event

- `IRQ - Data Sent` The current setting of the IRQ pin on "Data Sent" event
- `IRQ - Data Fail` The current setting of the IRQ pin on "Data Fail" event
- `Data Ready` Is there RX data ready to be read? (state of the *irq_dr* flag)
- `Data Sent` Has the TX data been sent? (state of the *irq_ds* flag)
- `Data Failed` Has the maximum attempts to re-transmit been reached? (state of the *irq_df* flag)
- `TX FIFO full` Is the TX FIFO buffer full? (state of the *tx_full* flag)
- `TX FIFO empty` Is the TX FIFO buffer empty?
- `RX FIFO full` Is the RX FIFO buffer full?
- `RX FIFO empty` Is the RX FIFO buffer empty?
- `Custom ACK payload` Is the nRF24L01 setup to use an extra (user defined) payload attached to the acknowledgment packet? (state of the *ack* attribute)
- `Ask no ACK` Is the nRF24L01 setup to transmit individual packets that don't require acknowledgment?
- `Automatic Acknowledgment` The status of the *auto_ack* feature. If this value is a binary representation, then each bit represents the feature's status for each pipe.
- `Dynamic Payloads` The status of the *dynamic_payloads* feature. If this value is a binary representation, then each bit represents the feature's status for each pipe.
- `Primary Mode` The current mode (RX or TX) of communication of the nRF24L01 device.
- `Power Mode` The power state can be Off, Standby-I, Standby-II, or On.

Parameters **dump_pipes** (*bool*) – `True` appends the output and prints:

- the current address used for TX transmissions. This value is the entire content of the nRF24L01's register about the TX address (despite what *address_length* is set to).
- `Pipe [#] ([open/closed]) bound: [address]` where `#` represent the pipe number, the `open/closed` status is relative to the pipe's RX status, and `address` is the full value stored in the nRF24L01's RX address registers (despite what *address_length* is set to.
- if the pipe is open, then the output also prints `expecting [X] byte static payloads` where `X` is the *payload_length* (in bytes) the pipe is setup to receive when *dynamic_payloads* is disabled for that pipe.

This parameter's default is `False` and skips this extra information.

13.2 is_plus_variant

`RF24.`**`is_plus_variant`**

A `bool` attribute to descibe if the nRF24L01 is a plus variant or not (read-only). This information is detirmined upon instantiation.

13.3 load_ack()

RF24.**load_ack**(*buf*, *pipe_number*)

This allows the MCU to specify a payload to be allocated into the TX FIFO buffer for use on a specific data pipe.

This payload will then be appended to the automatic acknowledgment (ACK) packet that is sent when *new* data is received on the specified pipe. See *recv()* on how to fetch a received custom ACK payloads.

Parameters

- **buf** (*bytearray*, *bytes*) – This will be the data attached to an automatic ACK packet on the incoming transmission about the specified `pipe_number` parameter. This must have a length in range [1, 32] bytes, otherwise a `ValueError` exception is thrown. Any ACK payloads will remain in the TX FIFO buffer until transmitted successfully or *flush_tx()* is called.
- **pipe_number** (*int*) – This will be the pipe number to use for deciding which transmissions get a response with the specified `buf` parameter's data. This number must be in range [0, 5], otherwise a `IndexError` exception is thrown.

Returns `True` if payload was successfully loaded onto the TX FIFO buffer. `False` if it wasn't because TX FIFO buffer is full.

Note: this function takes advantage of a special feature on the nRF24L01 and needs to be called for every time a customized ACK payload is to be used (not for every automatic ACK packet – this just appends a payload to the ACK packet). The *ack*, *auto_ack*, and *dynamic_payloads* attributes are also automatically enabled (with respect to data pipe 0) by this function when necessary.

Tip: The ACK payload must be set prior to receiving a transmission. It is also worth noting that the nRF24L01 can hold up to 3 ACK payloads pending transmission. Using this function does not over-write existing ACK payloads pending; it only adds to the queue (TX FIFO buffer) if it can. Use *flush_tx()* to discard unused ACK payloads when done listening.

13.4 read_ack()

RF24.**read_ack**()

Allows user to read the automatic acknowledgement (ACK) payload (if any).

This function is an alias of *recv()* and remains for backward compatibility with older versions of this library.

> **Warning:** This function will be deprecated on next major release. Use *recv()* instead.

13.5 irq_dr

RF24.**irq_dr**

A `bool` that represents the "Data Ready" interrupted flag. (read-only) .

Returns

- `True` represents Data is in the RX FIFO buffer
- `False` represents anything depending on context (state/condition of FIFO buffers); usually this means the flag's been reset.

Pass `data_recv` parameter as `True` to *`clear_status_flags()`* and reset this. As this is a virtual representation of the interrupt event, this attribute will always be updated despite what the actual IRQ pin is configured to do about this event.

Calling this does not execute an SPI transaction. It only exposes that latest data contained in the STATUS byte that's always returned from any other SPI transactions. Use the *`update()`* function to manually refresh this data when needed (especially after calling *`clear_status_flags()`*).

13.6 irq_df

`RF24`.**`irq_df`**

A `bool` that represents the "Data Failed" interrupted flag. (read-only) .

Returns

- `True` signifies the nRF24L01 attemped all configured retries
- `False` represents anything depending on context (state/condition); usually this means the flag's been reset.

Pass `data_fail` parameter as `True` to *`clear_status_flags()`* and reset this. As this is a virtual representation of the interrupt event, this attribute will always be updated despite what the actual IRQ pin is configured to do about this event.

Calling this does not execute an SPI transaction. It only exposes that latest data contained in the STATUS byte that's always returned from any other SPI transactions. Use the *`update()`* function to manually refresh this data when needed (especially after calling *`clear_status_flags()`*).

13.7 irq_ds

`RF24`.**`irq_ds`**

A `bool` that represents the "Data Sent" interrupted flag. (read-only) .

Returns

- `True` represents a successful transmission
- `False` represents anything depending on context (state/condition of FIFO buffers); usually this means the flag's been reset.

Pass `data_sent` parameter as `True` to *`clear_status_flags()`* and reset this. As this is a virtual representation of the interrupt event, this attribute will always be updated despite what the actual IRQ pin is configured to do about this event.

Calling this does not execute an SPI transaction. It only exposes that latest data contained in the STATUS byte that's always returned from any other SPI transactions. Use the *`update()`* function to manually refresh this data when needed (especially after calling *`clear_status_flags()`*).

13.8 clear_status_flags()

RF24.**clear_status_flags**(*data_recv=True*, *data_sent=True*, *data_fail=True*)

This clears the interrupt flags in the status register.

Internally, this is automatically called by *send()*, *write()*, *recv()*, and when *listen* changes from False to True.

Parameters

- **data_recv** (*bool*) – specifies wheather to clear the "RX Data Ready" (*irq_dr*) flag.
- **data_sent** (*bool*) – specifies wheather to clear the "TX Data Sent" (*irq_ds*) flag.
- **data_fail** (*bool*) – specifies wheather to clear the "Max Re-transmit reached" (*irq_df*) flag.

Note: Clearing the data_fail flag is necessary for continued transmissions from the nRF24L01 (locks the TX FIFO buffer when *irq_df* is True) despite wheather or not the MCU is taking advantage of the interrupt (IRQ) pin. Call this function only when there is an antiquated status flag (after you've dealt with the specific payload related to the staus flags that were set), otherwise it can cause payloads to be ignored and occupy the RX/TX FIFO buffers. See Appendix A of the nRF24L01+ Specifications Sheet for an outline of proper behavior.

13.9 power

RF24.**power**

This bool attribute controls the power state of the nRF24L01. This is exposed for convenience.

- False basically puts the nRF24L01 to sleep (AKA power down mode) with ultra-low current consumption. No transmissions are executed when sleeping, but the nRF24L01 can still be accessed through SPI. Upon instantiation, this driver class puts the nRF24L01 to sleep until the MCU invokes RX/TX modes. This driver class will only power down the nRF24L01 after exiting a The with statement block.
- True powers up the nRF24L01. This is the first step towards entering RX/TX modes (see also *listen* attribute). Powering up is automatically handled by the *listen* attribute as well as the *send()* and *write()* functions.

Note: This attribute needs to be True if you want to put radio on Standby-II (highest current consumption) or Standby-I (moderate current consumption) modes. The state of the CE pin determines which Standby mode is acheived. See Chapter 6.1.2-7 of the nRF24L01+ Specifications Sheet for more details.

13.10 tx_full

RF24.**tx_full**

An attribute to represent the nRF24L01's status flag signaling that the TX FIFO buffer is full. (read-only) .

Calling this does not execute an SPI transaction. It only exposes that latest data contained in the STATUS byte that's always returned from any other SPI transactions. Use the *update()* function to manually refresh this data when needed (especially after calling *flush_tx()*).

Returns

- `True` for TX FIFO buffer is full
- `False` for TX FIFO buffer is not full. This doesn't mean the TX FIFO buffer is empty.

13.11 update()

`RF24.`**`update`**`()`

This function is only used to get an updated status byte over SPI from the nRF24L01.

Refreshing the status byte is vital to checking status of the interrupt flags, RX pipe number related to current RX payload, and if the TX FIFO buffer is full. This function returns nothing, but internally updates the *`irq_dr`*, *`irq_ds`*, *`irq_df`*, *`pipe`*, and *`tx_full`* attributes. Internally this is a helper function to *`send()`*, and *`resend()`* functions.

Returns

`True` for every call. This value is meant to allow this function to be used in `if` statements in conjunction with attributes related to the refreshed status byte.

```
# let ``nrf`` be the instantiated object of the RF24 class

# the following if statement is faster than using ``if nrf.any():``
if nrf.update() and nrf.pipe is not None:
    nrf.recv()
```

13.12 resend()

`RF24.`**`resend`**`(`*send_only=False*`)`

Use this function to maunally re-send the previous payload in the top level (first out) of the TX FIFO buffer.

This function is meant to be used for payloads that failed to transmit using the *`send()`* function. If a payload failed to transmit using the *`write()`* function, just call *`clear_status_flags()`* and re-start the pulse on the nRF24L01's CE pin.

Returns Data returned from this function follows the same pattern that *`send()`* returns with the added condition that this function will return `False` if the TX FIFO buffer is empty.

Parameters **`send_only`** (*`bool`*) – This parameter only applies when the *`ack`* attribute is set to `True`. Pass this parameter as `True` if the RX FIFO is not to be manipulated. Many other libraries' behave as though this parameter is `True` (e.g. The popular TMRh20 Arduino RF24 library). This parameter defaults to `False`. Use *`recv()`* to get the ACK payload (if there is any) from the RX FIFO. Remember that the RX FIFO can only hold up to 3 payloads at once.

Note: The nRF24L01 normally removes a payload from the TX FIFO buffer after successful transmission, but not when this function is called. The payload (successfully transmitted or not) will remain in the TX FIFO buffer until *`flush_tx()`* is called to remove them. Alternatively, using this function also allows the failed payload to be over-written by using *`send()`* or *`write()`* to load a new payload into the TX FIFO buffer.

13.13 write()

RF24.**write**(*buf*, *ask_no_ack=False*, *write_only=False*)

This non-blocking function (when used as alternative to *send()*) is meant for asynchronous applications and can only handle one payload at a time as it is a helper function to *send()*.

This function isn't completely non-blocking as we still need to wait 5 ms (*CSN_DELAY*) for the CSN pin to settle (allowing an accurate SPI write transaction). Example usage of this function can be seen in the IRQ pin example and in the Stream example's "master_fifo()" function

Returns `True` if the payload was added to the TX FIFO buffer. `False` if the TX FIFO buffer is already full, and no payload could be added to it.

Parameters

- **buf** (*bytearray*) – The payload to transmit. This bytearray must have a length greater than 0 and less than 32 bytes, otherwise a `ValueError` exception is thrown.
  - If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is less than the *payload_length* attribute for data pipe 0, then this bytearray is padded with zeros until its length is equal to the *payload_length* attribute for data pipe 0.
  - If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is greater than *payload_length* attribute for data pipe 0, then this bytearray's length is truncated to equal the *payload_length* attribute for data pipe 0.
- **ask_no_ack** (*bool*) – Pass this parameter as `True` to tell the nRF24L01 not to wait for an acknowledgment from the receiving nRF24L01. This parameter directly controls a `NO_ACK` flag in the transmission's Packet Control Field (9 bits of information about the payload). Therefore, it takes advantage of an nRF24L01 feature specific to individual payloads, and its value is not saved anywhere. You do not need to specify this for every payload if the *arc* attribute is disabled, however setting this parameter to `True` will work despite the *arc* attribute's setting.

  Note: Each transmission is in the form of a packet. This packet contains sections of data around and including the payload. See Chapter 7.3 in the nRF24L01 Specifications Sheet for more details.

- **write_only** (*bool*) – This function will not manipulate the nRF24L01's CE pin if this parameter is `True`. The default value of `False` will ensure that the CE pin is HIGH upon exiting this function. This function does not set the CE pin LOW at any time. Use this parameter as `True` to fill the TX FIFO buffer before beginning transmissions.

  Note: The nRF24L01 doesn't initiate sending until a mandatory minimum 10 µs pulse on the CE pin is acheived. If the `write_only` parameter is `False`, then that pulse is initiated before this function exits. However, we have left that 10 µs wait time to be managed by the MCU in cases of asychronous application, or it is managed by using *send()* instead of this function. According to the Specification sheet, if the CE pin remains HIGH for longer than 10 µs, then the nRF24L01 will continue to transmit all payloads found in the TX FIFO buffer.

Warning: A note paraphrased from the nRF24L01+ Specifications Sheet:

It is important to NEVER to keep the nRF24L01+ in TX mode for more than 4 ms at a time. If the [*arc* attribute is] enabled, nRF24L01+ is never in TX mode longer than 4 ms.

Tip: Use this function at your own risk. Because of the underlying "Enhanced ShockBurst Protocol", disobeying the 4 ms rule is easily avoided if the *arc* attribute is greater than `0`. Alternatively, you MUST use nRF24L01's IRQ pin and/or user-defined timer(s) to AVOID breaking the 4 ms rule. If the nRF24L01+ Specifications Sheet explicitly states this, we have to assume radio damage or misbehavior as a result of disobeying the 4 ms rule. See also table 18 in the nRF24L01 specification sheet for calculating an adequate transmission timeout sentinal.

13.14 flush_rx()

`RF24.`**`flush_rx`**`()`
A helper function to flush the nRF24L01's RX FIFO buffer.

Note: The nRF24L01 RX FIFO is 3 level stack that holds payload data. This means that there can be up to 3 received payloads (each of a maximum length equal to 32 bytes) waiting to be read (and removed from the stack) by *recv()* or *read_ack()*. This function clears all 3 levels.

13.15 flush_tx()

`RF24.`**`flush_tx`**`()`
A helper function to flush the nRF24L01's TX FIFO buffer.

Note: The nRF24L01 TX FIFO is 3 level stack that holds payload data. This means that there can be up to 3 payloads (each of a maximum length equal to 32 bytes) waiting to be transmit by *send()*, *resend()* or *write()*. This function clears all 3 levels. It is worth noting that the payload data is only removed from the TX FIFO stack upon successful transmission (see also *resend()* as the handling of failed transmissions can be altered).

13.16 fifo()

`RF24.`**`fifo`**`(`*about_tx=False*, *check_empty=None*`)`
This provides *some* precision determining the status of the TX/RX FIFO buffers. (read-only)

Parameters

- **about_tx** (*bool*) –
  - `True` means the information returned is about the TX FIFO buffer.
  - `False` means the information returned is about the RX FIFO buffer. This parameter defaults to `False` when not specified.

- **check_empty** (*bool*) –
  - `True` tests if the specified FIFO buffer is empty.
  - `False` tests if the specified FIFO buffer is full.
  - `None` (when not specified) returns a 2 bit number representing both empty (bit 1) & full (bit 0) tests related to the FIFO buffer specified using the `about_tx` parameter.

Returns

- A `bool` answer to the question:

  "Is the [TX/RX](`about_tx`) FIFO buffer [empty/full](`check_empty`)?
- If the `check_empty` parameter is not specified: an `int` in range [0,2] for which:
  - `1` means the specified FIFO buffer is empty
  - `2` means the specified FIFO buffer is full
  - `0` means the specified FIFO buffer is neither full nor empty

13.17 pipe

`RF24.`**`pipe`**

The identifying number of the data pipe that received the next available payload in the RX FIFO buffer. (read only) .

Calling this does not execute an SPI transaction. It only exposes that latest data contained in the STATUS byte that's always returned from any other SPI transactions. Use the *`update()`* function to manually refresh this data when needed (especially after calling *`flush_rx()`*).

Returns

- `None` if there is no payload in RX FIFO.
- The `int` identifying pipe number [0,5] that received the next available payload in the RX FIFO buffer.

13.18 address_length

`RF24.`**`address_length`**

This `int` attribute specifies the length (in bytes) of addresses to be used for RX/TX pipes. A valid input value must be an `int` in range [3, 5]. Otherwise a `ValueError` exception is thrown. Default is set to the nRF24L01's maximum of 5.

13.19 address()

`RF24.`**`address`**(*index=-1*)

Returns the current address set to a specified data pipe or the TX address. (read-only)

This function returns the full content of the nRF24L01's registers about RX/TX addresses despite what *`address_length`* is set to.

Parameters **index** (*int*) – the number of the data pipe whose address is to be returned. A valid index ranges [0,5] for RX addresses or any negative number for the TX address. Otherwise an `IndexError` is thown. This parameter defaults to `-1`.

13.20 rpd

`RF24.`**`rpd`**

This read-only attribute returns `True` if RPD (Received Power Detector) is triggered or `False` if not triggered. The RPD flag is triggered in the following cases:

1. During RX mode (when *`listen`* is `True`) and an arbitrary RF transmission with a gain above -64 dBm threshold is/was present.
2. When a packet is received (instigated by the nRF24L01 used to detect/"listen" for incoming packets).

Note: See also section 6.4 of the Specification Sheet concerning the RPD flag. Ambient temperature affects the -64 dBm threshold. The latching of this flag happens differently under certain conditions.

13.21 start_carrier_wave()

`RF24.`**`start_carrier_wave`**`()`

Starts a continuous carrier wave test.

This is a basic test of the nRF24L01's TX output. It is a commonly required test for telecommunication regulations. Calling this function may introduce interference with other transceivers that use frequencies in range [2.4, 2.525] GHz. To verify that this test is working properly, use the following code on a seperate nRF24L01 transceiver:

```
# declare objects for SPI bus and CSN pin and CE pin
nrf. = RF24(spi, csn, ce)
# set nrf.pa_level, nrf.channel, & nrf.data_rate values to
# match the corresponding attributes on the device that is
# transmitting the carrier wave
nrf.listen = True
if nrf.rpd:
    print("carrier wave detected")
```

The *`pa_level`*, *`channel`* & *`data_rate`* attributes are vital factors to the success of this test. Be sure these attributes are set to the desired test conditions before calling this function. See also the *`rpd`* attribute.

Note: To preserve backward compatibility with non-plus variants of the nRF24L01, this function will also change certain settings if *`is_plus_variant`* is `False`. These settings changes include disabling *`crc`*, disabling *`auto_ack`*, disabling *`arc`*, setting *`ard`* to 250 microseconds, changing the TX address to `b"\xFF\xFF\xFF\xFF\xFF"`, and loading a 32-byte payload (each byte is `0xFF`) into the TX FIFO buffer while continuously behaving like *`resend()`* to establish the constant carrier wave. If *`is_plus_variant`* is `True`, then none of these changes are needed nor applied.

13.22 stop_carrier_wave()

RF24.**stop_carrier_wave**()

Stops a continuous carrier wave test.

See *start_carrier_wave()* for more details.

Note: Calling this function puts the nRF24L01 to sleep (AKA power down mode).

CHAPTER 14

Configuration API

14.1 CSN_DELAY

circuitpython_nrf24l01.rf24.**CSN_DELAY = 0.005**

The delay time (in seconds) used to let the CSN pin settle, allowing a clean SPI transaction.

14.2 dynamic_payloads

RF24.**dynamic_payloads**

This `bool` attribute controls the nRF24L01's dynamic payload length feature for each pipe. Default setting is enabled on all pipes.

- `True` or `1` enables nRF24L01's dynamic payload length feature for all data pipes. The *`payload_length`* attribute is ignored when this feature is enabled for all respective data pipes.
- `False` or `0` disables nRF24L01's dynamic payload length feature for all data pipes. Be sure to adjust the *`payload_length`* attribute accordingly when this feature is disabled for any respective data pipes.
- A `list` or `tuple` containing booleans or integers can be used control this feature per data pipe. Index 0 controls this feature on data pipe 0. Indices greater than 5 will be ignored since there are only 6 data pipes. If any index's value is less than 0 (a negative value), then the pipe corresponding to that index will remain unaffected.

Note: This attribute mostly relates to RX operations, but data pipe 0 applies to TX operations also. The *`auto_ack`* attribute is automatically enabled by this attribute for any data pipes that have this feature enabled. Disabling this feature for any data pipe will not affect the *`auto_ack`* feature for the corresponding data pipes.

14.3 payload_length

RF24.**payload_length**

This int attribute specifies the length (in bytes) of static payloads for each pipe. If the *dynamic_payloads* attribute is *enabled* for a certain data pipe, this attribute has no affect on that data pipe. When *dynamic_payloads* is *disabled* for a certain data pipe, this attribute is used to specify the payload length on that data pipe.

A valid input value must be:

- an int in range [1, 32]. Otherwise a ValueError exception is thrown.
- A list or tuple containing integers can be used control this feature per data pipe. Index 0 controls this feature on data pipe 0. Indices greater than 5 will be ignored since there are only 6 data pipes. If any index's value is 0, then the existing setting will persist (not be changed).

Default is set to the nRF24L01's maximum of 32 (on all data pipes).

Note: This attribute mostly relates to RX operations, but data pipe 0 applies to TX operations also.

14.4 auto_ack

RF24.**auto_ack**

This bool attribute controls the nRF24L01's automatic acknowledgment feature during the process of receiving a packet. Default setting is enabled on all data pipes.

- True or 1 enables transmitting automatic acknowledgment packets for all data pipes. The CRC (cyclic redundancy checking) is enabled (for all transmissions) automatically by the nRF24L01 if this attribute is enabled for any data pipe (see also *crc* attribute).
- False or 0 disables transmitting automatic acknowledgment packets for all data pipes. The *crc* attribute will remain unaffected when disabling this attribute for any data pipes.
- A list or tuple containing booleans or integers can be used control this feature per data pipe. Index 0 controls this feature on data pipe 0. Indices greater than 5 will be ignored since there are only 6 data pipes. If any index's value is less than 0 (a negative value), then the pipe corresponding to that index will remain unaffected.

Note: This attribute mostly relates to RX operations, but data pipe 0 applies to TX operations also.

14.5 arc

RF24.**arc**

This int attribute specifies the nRF24L01's number of attempts to re-transmit TX payload when acknowledgment packet is not received. The *auto_ack* attribute must be enabled on the receiving nRF24L01 respective data pipe, otherwise this attribute will make *send()* seem like it failed.

A valid input value must be in range [0, 15]. Otherwise a ValueError exception is thrown. Default is set to 3. A value of 0 disables the automatic re-transmit feature and considers all payload transmissions a success.

14.6 ard

RF24.**ard**
This int attribute specifies the nRF24L01's delay (in microseconds) between attempts to automatically retransmit the TX payload when an expected acknowledgement (ACK) packet is not received. During this time, the nRF24L01 is listening for the ACK packet. If the *arc* attribute is disabled, this attribute is not applied.

A valid input value must be in range [250, 4000]. Otherwise a ValueError exception is thrown. Default is 1500 for reliability. If this is set to a value that is not multiple of 250, then the highest multiple of 250 that is no greater than the input value is used.

Note: Paraphrased from nRF24L01 specifications sheet:

Please take care when setting this parameter. If the custom ACK payload is more than 15 bytes in 2 Mbps data rate, the *ard* must be 500µS or more. If the custom ACK payload is more than 5 bytes in 1 Mbps data rate, the *ard* must be 500µS or more. In 250kbps data rate (even when there is no custom ACK payload) the *ard* must be 500µS or more.

See *data_rate* attribute on how to set the data rate of the nRF24L01's transmissions.

14.7 ack

RF24.**ack**
This bool attribute represents the status of the nRF24L01's capability to use custom payloads as part of the automatic acknowledgment (ACK) packet. Use this attribute to set/check if the custom ACK payloads feature is enabled. Default setting is False.

- True enables the use of custom ACK payloads in the ACK packet when responding to receiving transmissions.
- False disables the use of custom ACK payloads in the ACK packet when responding to receiving transmissions.

Important: As *dynamic_payloads* and *auto_ack* attributes are required for this feature to work, they are automatically enabled (on data pipe 0) as needed. However, it is required to enable the *auto_ack* and *dynamic_payloads* features on all applicable pipes. Disabling this feature does not disable the *auto_ack* and *dynamic_payloads* attributes for any data pipe; they work just fine without this feature.

14.8 interrupt_config()

RF24.**interrupt_config**(*data_recv=True*, *data_sent=True*, *data_fail=True*)
Sets the configuration of the nRF24L01's IRQ pin. (write-only)

The digital signal from the nRF24L01's IRQ (Interrupt ReQuest) pin is active LOW.

Parameters

- **data_recv** (*bool*) – If this is True, then IRQ pin goes active when new data is put into the RX FIFO buffer. Default setting is True
- **data_sent** (*bool*) – If this is True, then IRQ pin goes active when a payload from TX buffer is successfully transmit. Default setting is True

- **data_fail** (*bool*) – If this is `True`, then IRQ pin goes active when the maximum number of attempts to re-transmit the packet have been reached. If *arc* attribute is disabled, then this IRQ event is not used. Default setting is `True`

Note: To fetch the status (not configuration) of these IRQ flags, use the *irq_df*, *irq_ds*, *irq_dr* attributes respectively.

Tip: Paraphrased from nRF24L01+ Specification Sheet:

The procedure for handling *irq_dr* IRQ should be:

1. retreive the payload from RX FIFO using *recv()*
2. clear *irq_dr* status flag (taken care of by using *recv()* in previous step)
3. read FIFO_STATUS register to check if there are more payloads available in RX FIFO buffer. A call to *pipe* (may require *update()* to be called beforehand), *any()* or even `(False, True)` as parameters to *fifo()* will get this result.
4. if there is more data in RX FIFO, repeat from step 1

14.9 data_rate

`RF24.`**`data_rate`**

This `int` attribute specifies the nRF24L01's frequency data rate for OTA (over the air) transmissions. A valid input value is:

- `1` sets the frequency data rate to 1 Mbps
- `2` sets the frequency data rate to 2 Mbps
- `250` sets the frequency data rate to 250 Kbps (see warning below)

Any invalid input throws a `ValueError` exception. Default is 1 Mbps.

Warning: 250 Kbps is not available for the non-plus variants of the nRF24L01 transceivers. Trying to set the data rate to 250 kpbs when *is_plus_variant* is `True` will throw a `NotImplementedError`.

14.10 channel

`RF24.`**`channel`**

This `int` attribute specifies the nRF24L01's frequency. A valid input value must be in range [0, 125] (that means [2.4, 2.525] GHz). Otherwise a `ValueError` exception is thrown. Default is `76` (2.476 GHz).

14.11 crc

`RF24.`**`crc`**

This `int` attribute specifies the nRF24L01's CRC (cyclic redundancy checking) encoding scheme in terms of byte length. CRC is a way of making sure that the transmission didn't get corrupted over the air.

A valid input value must be:

- `0` disables CRC (no anti-corruption of data)
- `1` enables CRC encoding scheme using 1 byte (weak anti-corruption of data)
- `2` enables CRC encoding scheme using 2 bytes (better anti-corruption of data)

Any invalid input throws a `ValueError` exception. Default is enabled using 2 bytes.

Note: The nRF24L01 automatically enables CRC if automatic acknowledgment feature is enabled (see *`auto_ack`* attribute) for any data pipe.

14.12 pa_level

`RF24`**.pa_level**

This `int` attribute specifies the nRF24L01's power amplifier level (in dBm). Higher levels mean the transmission will cover a longer distance. Use this attribute to tweak the nRF24L01 current consumption on projects that don't span large areas.

A valid input value is:

- `-18` sets the nRF24L01's power amplifier to -18 dBm (lowest)
- `-12` sets the nRF24L01's power amplifier to -12 dBm
- `-6` sets the nRF24L01's power amplifier to -6 dBm
- `0` sets the nRF24L01's power amplifier to 0 dBm (highest)

If this attribute is set to a `list` or `tuple`, then the list/tuple must contain the desired power amplifier level (from list above) at index 0 and a `bool` to control the Low Noise Amplifier (LNA) feature at index 1. All other indices will be discarded.

Note: The LNA feature setting only applies to the nRF24L01 (non-plus variant).

Any invalid input will invoke the default of 0 dBm with LNA enabled.

14.13 is_lna_enabled

`RF24`**.is_lna_enabled**

A read-only `bool` attribute about the LNA (Low Noise Amplifier) gain feature. See *`pa_level`* attribute about how to set this. Default is always enabled, but this feature is specific to non-plus variants of nRF24L01 transceivers. If *`is_plus_variant`* attribute is `True`, then setting feature in any way has no affect.

CHAPTER 15

BLE Limitations

This module uses the *RF24* class to make the nRF24L01 imitate a Bluetooth-Low-Emissions (BLE) beacon. A BLE beacon can send data (referred to as advertisements) to any BLE compatible device (ie smart devices with Bluetooth 4.0 or later) that is listening.

Original research was done by Dmitry Grinberg and his write-up (including C source code) can be found here As this technique can prove invaluable in certain project designs, the code here has been adapted to work with CircuitPython.

Important: Because the nRF24L01 wasn't designed for BLE advertising, it has some limitations that helps to be aware of.

1. The maximum payload length is shortened to **18** bytes (when not broadcasting a device *name* nor the nRF24L01 *show_pa_level*). This is calculated as:

   32 (nRF24L01 maximum) - **6** (MAC address) - **5** (required flags) - **3** (CRC checksum) = **18**

   Use the helper function *available()* to detirmine if your payload can be transmit.

2. the channels that BLE use are limited to the following three: 2.402 GHz, 2.426 GHz, and 2.480 GHz. We have provided a tuple of these specific channels for convenience (See *BLE_FREQ* and *hop_channel()*).

3. *crc* is disabled in the nRF24L01 firmware because BLE specifications require 3 bytes (*crc24_ble()*), and the nRF24L01 firmware can only handle a maximum of 2. Thus, we have appended the required 3 bytes of CRC24 into the payload.

4. *address_length* of BLE packet only uses 4 bytes, so we have set that accordingly.

5. The *auto_ack* (automatic acknowledgment) feature of the nRF24L01 is useless when tranmitting to BLE devices, thus it is disabled as well as automatic re-transmit (*arc*) and custom ACK payloads (*ack*) features which both depend on the automatic acknowledgments feature.

6. The *dynamic_payloads* feature of the nRF24L01 isn't compatible with BLE specifications. Thus, we have disabled it.

7. BLE specifications only allow using 1 Mbps RF *data_rate*, so that too has been hard coded.

8. Only the "on data sent" (*irq_ds*) & "on data ready" (*irq_dr*) events will have an effect on the interrupt (IRQ) pin. The "on data fail" (*irq_df*) is never triggered because *arc* attribute is disabled.

CHAPTER 16

helpers

16.1 swap_bits()

circuitpython_nrf24l01.fake_ble.**swap_bits**(*original*)

This function reverses the bit order for a single byte.

Returns An `int` containing the byte whose bits are reversed compared to the value passed to the `original` parameter.

Parameters **original** (*int*) – This should be a single unsigned byte, meaning the parameters value can only range from 0 to 255.

16.2 reverse_bits()

circuitpython_nrf24l01.fake_ble.**reverse_bits**(*original*)

This function reverses the bit order for an entire buffer protocol object.

Returns A `bytearray` whose byte order remains the same, but each byte's bit order is reversed.

Parameters **original** (*bytearray*, *bytes*) – The original buffer whose bits are to be reversed.

16.3 chunk()

circuitpython_nrf24l01.fake_ble.**chunk**(*buf*, *data_type=22*)

This function is used to pack data values into a block of data that make up part of the BLE payload per Bluetooth Core Specifications.

Parameters

- **buf** (*bytearray*, *bytes*) – The actual data contained in the block.

- **data_type** (*int*) – The type of data contained in the chunk. This is a predefined number according to BLE specifications. The default value `0x16` describes all service data. `0xFF` describes manufacturer information. Any other values are not applicable to BLE advertisements.

Important: This function is called internally by *`advertise()`*. To pack multiple data values into a single payload, use this function for each data value and pass a `list` or `tuple` of the returned results to *`advertise()`* (see example code in documentation about *`advertise()`* for more detail). Remember that broadcasting multiple data values may require the *`name`* be set to `None` and/or the *`show_pa_level`* be set to `False` for reasons about the payload size with *BLE Limitations*.

16.4 crc24_ble()

`circuitpython_nrf24l01.fake_ble.`**`crc24_ble`**(*data*, *deg_poly=1627*, *init_val=5592405*)

This function calculates a checksum of various sized buffers.

This is exposed for convenience but should not be used for other buffer protocols that require big endian CRC24 format.

Parameters

- **data** (*bytearray*, *bytes*) – The buffer of data to be uncorrupted.
- **deg_poly** (*int*) – A preset "degree polynomial" in which each bit represents a degree who's coefficient is 1. BLE specfications require `0x00065b` (default value).
- **init_val** (*int*) – This will be the initial value that the checksum will use while shifting in the buffer data. BLE specfications require `0x555555` (default value).

Returns A 24-bit `bytearray` representing the checksum of the data (in proper little endian).

16.5 BLE_FREQ

`circuitpython_nrf24l01.fake_ble.`**`BLE_FREQ = (2, 26, 80)`**

The BLE channel number is different from the nRF channel number. This tuple contains the relative predefined channels used:

- nRF channel 2 == BLE channel 37
- nRF channel 26 == BLE channel 38
- nRF channel 80 == BLE channel 39

CHAPTER 17

FakeBLE class

class circuitpython_nrf24l01.fake_ble.**FakeBLE**(*spi*, *csn*, *ce*, *spi_frequency=10000000*)

A class to implement BLE advertisements using the nRF24L01.

Per the limitations of this technique, only some of underlying *RF24* functionality is available for configuration when implementing BLE transmissions. See the *Available RF24 functionality* for more details.

Parameters

- **spi** (*SPI*) – The object for the SPI bus that the nRF24L01 is connected to.

  Tip: This object is meant to be shared amongst other driver classes (like adafruit_mcp3xxx.mcp3008 for example) that use the same SPI bus. Otherwise, multiple devices on the same SPI bus with different spi objects may produce errors or undesirable behavior.

- **csn** (*DigitalInOut*) – The digital output pin that is connected to the nRF24L01's CSN (Chip Select Not) pin. This is required.
- **ce** (*DigitalInOut*) – The digital output pin that is connected to the nRF24L01's CE (Chip Enable) pin. This is required.
- **spi_frequency** (*int*) – Specify which SPI frequency (in Hz) to use on the SPI bus. This parameter only applies to the instantiated object and is made persistent via `SPIDevice`.

17.1 to_android

FakeBLE.**to_android**

A `bool` attribute to specify if advertisements should be compatible with Android smartphones. A value of `True` allows advertisements to be compatible with Android smartphones. Setting this attribute to `False` still allows advertisements to be compatible with anything else except Android smartphones. Default Value is `True`.

> **Warning:** This attribute will be deprecated on the next major release because it is not necessary to change this attribute. Changing this attribute to `False` only breaks compatibility with Android smartphones.

17.2 mac

`FakeBLE.`**`mac`**

This attribute returns a 6-byte buffer that is used as the arbitrary mac address of the BLE device being emulated. You can set this attribute using a 6-byte `int` or `bytearray`. If this is set to `None`, then a random 6-byte address is generated.

17.3 name

`FakeBLE.`**`name`**

The broadcasted BLE name of the nRF24L01. This is not required. In fact setting this attribute will subtract from the available payload length (in bytes). Set this attribute to `None` to disable advertising the device name.

> **Note:** This information occupies (in the TX FIFO) an extra 2 bytes plus the length of the name set by this attribute.

17.4 show_pa_level

`FakeBLE.`**`show_pa_level`**

If this attribute is `True`, the payload will automatically include the nRF24L01's pa_level in the advertisement. The default value of `False` will exclude this optional information.

> **Note:** This information occupies (in the TX FIFO) an extra 3 bytes, and is really only useful for some applications to calculate proximity to the nRF24L01 transceiver.

17.5 hop_channel()

`FakeBLE.`**`hop_channel`**`()`

Trigger an automatic change of BLE compliant channels.

17.6 whiten()

`FakeBLE.`**`whiten`**`(`*`data`*`)`

Whitening the BLE packet data ensures there's no long repeatition of bits.

This is done according to BLE specifications.

Parameters **`data`** (*`bytearray`*`,`*`bytes`*) – The packet to whiten.

Returns A `bytearray` of the `data` with the whitening algorythm applied.

Warning: This function uses the currently set BLE channel as a base case for the whitening coefficient. Do not call *hop_channel()* before using this function to de-whiten received payloads (which isn't officially supported yet). Note that *advertise()* uses this function internally to prevent such improper usage.

17.7 available()

FakeBLE.**available**(*hypothetical=b''*)

This function will calculates how much length (in bytes) is available in the next payload.

This is detirmined from the current state of *name* and *show_pa_level* attributes.

Parameters **hypothetical** (*bytearray*, *bytes*) – Pass a potential *chunk()* of data to this parameter to calculate the resulting left over length in bytes. This parameter is optional.

Returns An `int` representing the length of available bytes for the a single payload.

17.8 advertise()

FakeBLE.**advertise**(*buf=b''*, *data_type=255*)

This blocking function is used to broadcast a payload.

Returns Nothing as every transmission will register as a success under the required settings for BLE beacons.

Parameters

- **buf** (*bytearray*) – The payload to transmit. This bytearray must have a length greater than 0 and less than 22 bytes Otherwise a `ValueError` exception is thrown whose prompt will tell you the maximum length allowed under the current configuration. This can also be a list or tuple of payloads (`bytearray`); in which case, all items in the list/tuple are processed are packed into 1 payload for a single transmissions. See example code below about passing a `list` or `tuple` to this parameter.
- **data_type** (*int*) – This is used to describe the buffer data passed to the `buf` parameter. `0x16` describes all service data. The default value `0xFF` describes manufacturer information. This parameter is ignored when a `tuple` or `list` is passed to the `buf` parameter. Any other values are not applicable to BLE advertisements.

Important: If the name and/or TX power level of the emulated BLE device is also to be broadcast, then the *name* and/or *show_pa_level* attribute(s) should be set prior to calling *advertise()*.

To pass multiple data values to the `buf` parameter see the following code as an example:

```
# let UUIDs be the 16-bit identifier that corresponds to the
# BLE service type. The following values are not compatible with
# BLE advertisements.
UUID_1 = 0x1805
UUID_2 = 0x1806
service1 = ServiceData(UUID_1)
service2 = ServiceData(UUID_2)
service1.data = b"some value 1"
service2.data = b"some value 2"
```

(continues on next page)

(continued from previous page)

```
# make a tuple of the buffers
buffers = (
    chunk(service1.buffer),
    chunk(service2.buffer)
)

# let `ble` be the instantiated object of the FakeBLE class
ble.advertise(buffers)
ble.hop_channel()
```

17.9 Available RF24 functionality

17.9.1 pa_level

FakeBLE.**pa_level**
See *pa_level* for more details.

17.9.2 channel

FakeBLE.**channel**
The only allowed channels are those contained in the *BLE_FREQ* tuple.

17.9.3 payload_length

FakeBLE.**payload_length**
This attribute is best left at 32 bytes for all BLE operations.

17.9.4 power

FakeBLE.**power**
See *power* for more details.

17.9.5 is_lna_enabled

FakeBLE.**is_lna_enabled**
See *is_lna_enabled* for more details.

17.9.6 is_plus_variant

FakeBLE.**is_plus_variant**
See *is_plus_variant* for more details.

17.9.7 interrupt_config()

FakeBLE.**interrupt_config**()
See *interrupt_config()* for more details.

> **Warning:** The *irq_df* attribute (and also this function's data_fail parameter) is not implemented for BLE operations.

17.9.8 irq_ds

FakeBLE.**irq_ds**
See *irq_ds* for more details.

17.9.9 irq_dr

FakeBLE.**irq_dr**
See *irq_dr* for more details.

17.9.10 clear_status_flags()

FakeBLE.**clear_status_flags**()
See *clear_status_flags()* for more details.

17.9.11 update()

FakeBLE.**update**()
See *update()* for more details.

17.9.12 what_happened()

FakeBLE.**what_happened**()
See *what_happened()* for more details.

CHAPTER 18

Service related classes

18.1 abstract parent

class circuitpython_nrf24l01.fake_ble.**ServiceData**(*uuid*)

An abstract helper class to package specific service data using Bluetooth SIG defined 16-bit UUID flags to describe the data type.

Parameters **uuid** (*int*) – The 16-bit UUID "GATT Service assigned number" defined by the Bluetooth SIG to describe the service data. This parameter is required.

uuid

This returns the 16-bit Service UUID as a `bytearray` in little endian. (read-only)

data

The service's data. This is a `bytearray`, and its format is defined by relative Bluetooth Service Specifications (and GATT supplemental specifications).

buffer

Get the representation of the instantiated object as an immutable `bytes` object (read-only).

__len__()

For convenience, this class is compatible with python's builtin `len()` method. In this context, this will return the length of the object (in bytes) as it would appear in the advertisement payload.

18.2 derivitive children

class circuitpython_nrf24l01.fake_ble.**TemperatureServiceData**

Bases: *circuitpython_nrf24l01.fake_ble.ServiceData*

This derivitive of the *ServiceData* class can be used to represent temperature data values as a `float` value.

This class's *data* attribute accepts a `float` value as input and returns a `bytes` object that conforms to the Bluetooth Health Thermometer Measurement format as defined in the GATT Specifications Supplement.

class circuitpython_nrf24l01.fake_ble.**BatteryServiceData**
Bases: *circuitpython_nrf24l01.fake_ble.ServiceData*

This derivitive of the *ServiceData* class can be used to represent battery charge percentage as a 1-byte value.

The class's *data* attribute accepts a `int` value as input and returns a `bytes` object that conforms to the Bluetooth Battery Level format as defined in the GATT Specifications Supplement.

class circuitpython_nrf24l01.fake_ble.**UrlServiceData**
Bases: *circuitpython_nrf24l01.fake_ble.ServiceData*

This derivitive of the *ServiceData* class can be used to represent URL data as a `bytes` value.

This class's *data* attribute accepts a `str` of URL data as input, and returns the URL as a `bytes` object where some of the URL parts are encoded using Eddystone byte codes as defined by the specifications.

pa_level_at_1_meter
The TX power level (in dBm) at 1 meter from the nRF24L01. This defaults to -25 (due to testing when broadcasting with 0 dBm) and must be a 1-byte signed `int`.

CHAPTER 19

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

D

F

H

I