

---

# **nRF24L01 Library Documentation**

***Release 1.0***

**Brendan Doherty**

**Oct 06, 2020**



---

## Contents

---

<b>1</b>	<b>Features currently supported</b>	<b>3</b>
<b>2</b>	<b>Features currently unsupported</b>	<b>5</b>
2.1	Dependencies . . . . .	5
2.2	Installing from PyPI . . . . .	5
2.3	Pinout . . . . .	6
2.4	Using The Examples . . . . .	6
2.5	About the nRF24L01 . . . . .	7
<b>3</b>	<b>Key Features:</b>	<b>9</b>
<b>4</b>	<b>Applications</b>	<b>11</b>
4.1	Where do I get 1? . . . . .	12
4.2	Contributing . . . . .	12
<b>5</b>	<b>Sphinx documentation</b>	<b>13</b>
5.1	Table of Contents . . . . .	13
5.1.1	Simple test . . . . .	13
5.1.2	ACK Payloads Example . . . . .	15
5.1.3	IRQ Pin Example . . . . .	17
5.1.4	Stream Example . . . . .	19
5.1.5	Context Example . . . . .	21
5.1.6	Working with TMRh20's Arduino library . . . . .	22
5.1.7	RF24 class . . . . .	24
5.1.7.1	Basic API . . . . .	26
5.1.7.2	Advanced API . . . . .	30
5.2	Indices and tables . . . . .	39
	<b>Index</b>	<b>41</b>



Circuitpython driver library for the nRF24L01 transceiver

CircuitPython port of the nRF24L01 library from Micropython. Original work by Damien P. George & Peter Hinch can be found [here](#)

The Micropython source has been rewritten to expose all the nRF24L01's features and for compatibilty with the Raspberry Pi and other Circuitpython compatible devices. Modified by Brendan Doherty, Rhys Thomas

- Author(s): Damien P. George, Peter Hinch, Rhys Thomas, Brendan Doherty



# CHAPTER 1

---

## Features currently supported

---

- change the addresses' length (can be 3 to 5 bytes long)
- dynamically sized payloads (max 32 bytes each) or statically sized payloads
- automatic responding acknowledgment (ACK) for verifying transmission success
- custom acknowledgment (ACK) payloads for bi-directional communication
- flag a single payload for no acknowledgment (ACK) from the receiving nRF24L01
- “re-use the same payload” feature (for manually re-transmitting failed transmissions that remain in the buffer)
- multiple payload transmissions with one function call (MUST read documentation on the `send()` function)
- context manager compatible for easily switching between different radio configurations using “with” statements
- configure the interrupt (IRQ) pin to trigger (active low) on received, sent, and/or failed transmissions (these 3 flags control the 1 IRQ pin). There's also virtual representations of these interrupt flags 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 is buggy due to hardware design, but 1Mbps and 2Mbps are reliable)
- a nRF24L01 driven by this library can communicate with a nRF24L01 on an Arduino driven by the [TMRh20 RF24 library](#). See the `nrf24l01_arduino_handling_data.py` code in the [examples folder](#) of this library's repository





---

### Features currently unsupported

---

- as of yet, no [intended] implementation for Multiceiver mode (up to 6 TX nRF24L01 “talking” to 1 RX nRF24L01 simultaneously). Although this might be acheived easily using the “automatic retry delay” (*ard*) and “automatic retry count” (*arc*) attributes set accordingly (varyingly high – this has not been tested).

## 2.1 Dependencies

This driver depends on:

- [Adafruit CircuitPython](#)
- [Bus Device](#)

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

## 2.2 Installing from PyPI

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

```
pip3 install 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
```

(continues on next page)

(continued from previous page)

```
source .env/bin/activate
pip3 install circuitpython-nrf24l01
```

## 2.3 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](#).

nRF24L01	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	GPIO4	D4

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

## 2.4 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 REPL and run the following commands:

```
>>> from nrf24l01_simple_test import *
      nRF24L01 Simple test
      Run slave() on receiver
      Run master() on transmitter
>>> master(3)
Sending: 3 as struct: b'\x03\x00\x00\x00'
send() successful
```

(continues on next page)

(continued from previous page)

```
Transmission took 86.0 ms
Sending: 2 as struct: b'\x02\x00\x00\x00'
send() successful
Transmission took 109.0 ms
Sending: 1 as struct: b'\x01\x00\x00\x00'
send() successful
Transmission took 109.0 ms
# these results were observed from a test on the Raspberry Pi 3
# transmissions from a CircuitPython device took 32 to 64 ms
```

## 2.5 About the nRF24L01

Here are the features listed directly from the datasheet (referenced here in the documentation as the [nRF24L01+ Specification Sheet](#)):



## CHAPTER 3

---

### Key Features:

---

- Worldwide 2.4GHz ISM band operation
- 250kbps, 1Mbps and 2Mbps on air data rates
- Ultra low power operation
- 11.3mA TX at 0dBm output power
- 13.5mA RX at 2Mbps air data rate
- 900nA in power down
- 26 $\mu$ A in standby-I
- On chip voltage regulator
- 1.9 to 3.6V supply range
- Enhanced ShockBurst™
- Automatic packet handling
- Auto packet transaction handling
- 6 data pipe MultiCeiver™
- Drop-in compatibility with nRF24L01
- On-air compatible in 250kbps and 1Mbps with nRF2401A, nRF2402, nRF24E1 and nRF24E2
- Low cost BOM
- $\pm 60$ ppm 16MHz crystal
- 5V tolerant inputs
- Compact 20-pin 4x4mm QFN package



# CHAPTER 4

---

## Applications

---

- Wireless PC Peripherals
- Mouse, keyboards and remotes
- 3-in-1 desktop bundles
- Advanced Media center remote controls
- VoIP headsets
- Game controllers
- Sports watches and sensors
- RF remote controls for consumer electronics
- Home and commercial automation
- Ultra low power sensor networks
- Active RFID
- Asset tracking systems
- Toys

Future Project Ideas/Additions using the nRF24L01 (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](#)
- add a fake BLE module for sending BLE beacon advertisements from the nRF24L01 as outlined by [Dmitry Grinberg](#) in his [write-up](#) (including C source code). We've started developing this, but fell short of success in the BLEfake branch of this library's repository

## 4.1 Where do I get 1?

See the store links on the sidebar or just google “nRF24L01”. It is worth noting that you generally don’t want to buy just 1 as you need 2 for testing – 1 to send & 1 to receive and vice versa. This library has been tested on a cheaply bought 10 pack from Amazon.com using a highly recommended capacitor (100  $\mu$ F) on the power pins. Don’t get lost on Amazon or eBay! 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.

## 4.2 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).



# CHAPTER 5

---

## 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/html
```

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

## 5.1 Table of Contents

### 5.1.1 Simple test

Ensure your device works with this simple test.

Listing 1: `examples/nrf24l01_simple_test.py`

```
1 """
2 Simple example of library usage.
3 """
4 import time
5 import struct
6 import board
7 import digitalio as dio
8 from circuitpython_nrf24l01 import RF24
```

(continues on next page)

(continued from previous page)

```

9
10 # addresses needs to be in a buffer protocol object (bytearray)
11 address = b'1Node'
12
13 # change these (digital output) pins accordingly
14 ce = dio.DigitalInOut(board.D4)
15 csn = dio.DigitalInOut(board.D5)
16
17 # using board.SPI() automatically selects the MCU's
18 # available SPI pins, board.SCK, board.MOSI, board.MISO
19 spi = board.SPI() # init spi bus object
20
21 # we'll be using the dynamic payload size feature (enabled by default)
22 # initialize the nRF24L01 on the spi bus object
23 nrf = RF24(spi, csn, ce)
24
25 def master(count=5): # count = 5 will only transmit 5 packets
26     """Transmits an incrementing integer every second"""
27     # set address of RX node into a TX pipe
28     nrf.open_tx_pipe(address)
29     # ensures the nRF24L01 is in TX mode
30     nrf.listen = False
31
32     while count:
33         # use struct.pack to packetize your data
34         # into a usable payload
35         buffer = struct.pack('<i', count)
36         # 'i' means a single 4 byte int value.
37         # '<' means little endian byte order. this may be optional
38         print("Sending: {} as struct: {}".format(count, buffer))
39         now = time.monotonic() * 1000 # start timer
40         result = nrf.send(buffer)
41         if result is None:
42             print('send() timed out')
43         elif not result:
44             print('send() failed')
45         else:
46             print('send() successful')
47         # print timer results despite transmission success
48         print('Transmission took',
49             time.monotonic() * 1000 - now, 'ms')
50         time.sleep(1)
51         count -= 1
52
53 def slave(count=3):
54     """Polls the radio and prints the received value. This method expires
55     after 6 seconds of no received transmission"""
56     # set address of TX node into an RX pipe. NOTE you MUST specify
57     # which pipe number to use for RX, we'll be using pipe 0
58     # pipe number options range [0,5]
59     # the pipe numbers used during a transition don't have to match
60     nrf.open_rx_pipe(0, address)
61     nrf.listen = True # put radio into RX mode and power up
62
63     start = time.monotonic()
64     while count and (time.monotonic() - start) < 6:
65         if nrf.any():

```

(continues on next page)

(continued from previous page)

```

66     # print details about the received packet (if any)
67     print("Found {} bytes on pipe {}\\
68           ".format(repr(nrf.any()), nrf.pipe()))
69     # retrieve the received packet's payload
70     rx = nrf.recv() # clears flags & empties RX FIFO
71     # expecting an int, thus the string format '<i'
72     buffer = struct.unpack('<i', rx)
73     # print the only item in the resulting tuple from
74     # using `struct.unpack()`
75     print("Received: {}, Raw: {}".format(buffer[0], repr(rx)))
76     start = time.monotonic()
77     count -= 1
78     # this will listen indefinitely till count == 0
79     time.sleep(0.25)
80
81     # recommended behavior is to keep in TX mode while idle
82     nrf.listen = False # put the nRF24L01 in TX mode
83
84     print("""\\
85         nRF24L01 Simple test.\\n\\
86         Run slave() on receiver\\n\\
87         Run master() on transmitter""")

```

## 5.1.2 ACK Payloads Example

This is a test to show how to use custom acknowledgment payloads.

Listing 2: examples/nrf24l01\_ack\_payload\_test.py

```

1  """
2  Simple example of using the library to transmit
3  and retrieve custom automatic acknowledgment payloads.
4  """
5  import time
6  import board
7  import digitalio as dio
8  from circuitpython_nrf24l01 import RF24
9
10 # change these (digital output) pins accordingly
11 ce = dio.DigitalInOut(board.D4)
12 csn = dio.DigitalInOut(board.D5)
13
14 # using board.SPI() automatically selects the MCU's
15 # available SPI pins, board.SCK, board.MOSI, board.MISO
16 spi = board.SPI() # init spi bus object
17
18 # we'll be using the dynamic payload size feature (enabled by default)
19 # the custom ACK payload feature is disabled by default
20 # the custom ACK payload feature should not be enabled
21 # during instantiation due to its singular use nature
22 # meaning 1 ACK payload per 1 RX'd payload
23 nrf = RF24(spi, csn, ce)
24
25 # NOTE the the custom ACK payload feature will be enabled
26 # automatically when you call load_ack() passing:

```

(continues on next page)

(continued from previous page)

```

27 # a buffer protocol object (bytearray) of
28 # length ranging [1,32]. And pipe number always needs
29 # to be an int ranging [0,5]
30
31 # to enable the custom ACK payload feature
32 nrf.ack = True # False disables again
33
34 # addresses needs to be in a buffer protocol object (bytearray)
35 address = b'lNode'
36
37 # payloads need to be in a buffer protocol object (bytearray)
38 tx = b'Hello '
39
40 # NOTE ACK payloads (like regular payloads and addresses)
41 # need to be in a buffer protocol object (bytearray)
42 ACK = b'World '
43
44 def master(count=5): # count = 5 will only transmit 5 packets
45     """Transmits a dummy payload every second and prints the ACK payload"""
46     # recommended behavior is to keep in TX mode while idle
47     nrf.listen = False # put radio in TX mode
48
49     # set address of RX node into a TX pipe
50     nrf.open_tx_pipe(address)
51
52     while count:
53         buffer = tx + bytes([count + 48]) # output buffer
54         print("Sending (raw): {}".format(repr(buffer)))
55         # to read the ACK payload during TX mode we
56         # pass the parameter read_ack as True.
57         nrf.ack = True # enable feature before send()
58         now = time.monotonic() * 1000 # start timer
59         result = nrf.send(buffer) # becomes the response buffer
60         if result is None:
61             print('send() timed out')
62         elif not result:
63             print('send() failed')
64         else:
65             # print the received ACK that was automatically
66             # fetched and saved to "buffer" via send()
67             print('raw ACK: {}'.format(repr(result)))
68             # the ACK payload should now be in buffer
69             # print timer results despite transmission success
70             print('Transmission took',
71                   time.monotonic() * 1000 - now, 'ms')
72             time.sleep(1)
73             count -= 1
74
75 def slave(count=3):
76     """Prints the received value and sends a dummy ACK payload"""
77     # set address of TX node into an RX pipe. NOTE you MUST specify
78     # which pipe number to use for RX, we'll be using pipe 0
79     nrf.open_rx_pipe(0, address)
80
81     # put radio into RX mode, power it up, and set the first
82     # transmission's ACK payload and pipe number
83     nrf.listen = True

```

(continues on next page)

(continued from previous page)

```

84     buffer = ACK + bytes([count + 48])
85     # we must set the ACK payload data and corresponding
86     # pipe number [0,5]
87     nrf.load_ack(buffer, 0) # load ACK for first response
88
89     start = time.monotonic()
90     while count and (time.monotonic() - start) < (count * 2):
91         if nrf.any():
92             # this will listen indefinitely till count == 0
93             count -= 1
94             # print details about the received packet (if any)
95             print("Found {} bytes on pipe {} \
96                   ".format(repr(nrf.any()), nrf.pipe()))
97             # retrieve the received packet's payload
98             rx = nrf.recv() # clears flags & empties RX FIFO
99             print("Received (raw): {}".format(repr(rx)))
100            start = time.monotonic()
101            if count: # Going again?
102                # build new ACK
103                buffer = ACK + bytes([count + 48])
104                # load ACK for next response
105                nrf.load_ack(buffer, 0)
106
107            # recommended behavior is to keep in TX mode while idle
108            nrf.listen = False # put radio in TX mode
109            nrf.flush_tx() # flush any ACK payload
110
111
112    print("""\
113        nRF24L01 ACK test\n\
114        Run slave() on receiver\n\
115        Run master() on transmitter""")

```

### 5.1.3 IRQ Pin Example

This is a test to show how to use nRF24L01's interrupt pin.

Listing 3: examples/nrf24l01\_interrupt\_test.py

```

1  """
2  Simple example of detecting (and verifying) the IRQ
3  interrupt pin on the nRF24L01
4  """
5  import time
6  import board
7  import digitalio as dio
8  from circuitpython_nrf24l01 import RF24
9
10 # address needs to be in a buffer protocol object (bytearray is preferred)
11 address = b'1Node'
12
13 # select your digital input pin that's connected to the IRQ pin on the nRF4L01
14 irq = dio.DigitalInOut(board.D4)
15 irq.switch_to_input() # make sure its an input object
16 # change these (digital output) pins accordingly

```

(continues on next page)

(continued from previous page)

```

17 ce = dio.DigitalInOut(board.D4)
18 csn = dio.DigitalInOut(board.D5)
19
20 # using board.SPI() automatically selects the MCU's
21 # available SPI pins, board.SCK, board.MOSI, board.MISO
22 spi = board.SPI() # init spi bus object
23
24 # we'll be using the dynamic payload size feature (enabled by default)
25 # initialize the nRF24L01 on the spi bus object
26 nrf = RF24(spi, csn, ce)
27 nrf.arc = 15 # turn up automatic retries to the max. default is 3
28
29 def master(timeout=5): # will only wait 5 seconds for slave to respond
30     """Transmits once, receives once, and intentionally fails a transmit"""
31     # set address of RX node into a TX pipe
32     nrf.open_tx_pipe(address)
33     # ensures the nRF24L01 is in TX mode
34     nrf.listen = 0
35
36     # on data sent test
37     print("Pinging: enslaved nRF24L01 without auto_ack")
38     nrf.write(b'ping')
39     time.sleep(0.00001) # mandatory 10 microsecond pulse starts transmission
40     nrf.ce.value = 0 # end 10 us pulse; now in active TX
41     while not nrf.irq_DS and not nrf.irq_DF:
42         nrf.update() # updates the current status on IRQ flags
43     if nrf.irq_DS and not irq.value:
44         print('interrupt on data sent successful')
45     else:
46         print(
47             'IRQ on data sent is not active, check your wiring and call interrupt_
↳config()')
48     nrf.clear_status_flags() # clear all flags for next test
49
50     # on data ready test
51     nrf.listen = 1
52     nrf.open_rx_pipe(0, address)
53     start = time.monotonic()
54     while not nrf.any() and time.monotonic() - start < timeout: # wait for slave to
↳send
55         pass
56     if nrf.any():
57         print('Pong received')
58         if nrf.irq_DR and not irq.value:
59             print('interrupt on data ready successful')
60         else:
61             print(
62                 'IRQ on data ready is not active, check your wiring and call
↳interrupt_config()')
63         nrf.flush_rx()
64     else:
65         print('pong reception timed out!. make sure to run slave() on the other
↳nRF24L01')
66     nrf.clear_status_flags() # clear all flags for next test
67
68     # on data fail test
69     nrf.listen = False # put the nRF24L01 is in TX mode

```

(continues on next page)

(continued from previous page)

```

70     # the writing pipe should still be open since we didn't call close_tx_pipe()
71     nrf.flush_tx() # just in case the previous "on data sent" test failed
72     nrf.write(b'dummy') # slave isn't listening anymore
73     time.sleep(0.00001) # mandatory 10 microsecond pulse starts transmission
74     nrf.ce.value = 0 # end 10 us pulse; now in active TX
75     while not nrf.irq_DS and not nrf.irq_DF: # these attributes don't update_
↳ themselves
76         nrf.update() # updates the current status on all IRQ flags (irq_DR, irq_DF,
↳ irq_DS)
77         if nrf.irq_DF and not irq.value:
78             print('interrupt on data fail successful')
79         else:
80             print(
81                 'IRQ on data fail is not active, check your wiring and call interrupt_
↳ config()')
82         nrf.clear_status_flags() # clear all flags for next test
83
84 def slave(timeout=10): # will listen for 10 seconds before timing out
85     """Acts as a ponging RX node to successfully complete the tests on the master"""
86     # setup radio to receive ping
87     nrf.open_rx_pipe(0, address)
88     nrf.listen = 1
89     start = time.monotonic()
90     while not nrf.any() and time.monotonic() - start < timeout:
91         pass # nrf.any() also updates the status byte on every call
92     if nrf.any():
93         print("ping received. sending pong now.")
94     else:
95         print('listening timed out, please try again')
96     nrf.flush_rx()
97     nrf.listen = 0
98     nrf.open_tx_pipe(address)
99     nrf.send(b'pong') # send a payload to complete the on data ready test
100    # we're done on this side
101
102 print("""\
103     nRF24L01 Interrupt test\n\
104     Run master() to run IRQ pin tests\n\
105     Run slave() on the non-testing nRF24L01 to complete the test successfully""")

```

### 5.1.4 Stream Example

This is a test to show how to use the send() to transmit multiple payloads.

Listing 4: examples/nrf24l01\_stream\_test.py

```

1  """
2  Example of library usage for streaming multiple payloads.
3  """
4  import time
5  import board
6  import digitalio as dio
7  from circuitpython_nrf24l01 import RF24
8
9  # addresses needs to be in a buffer protocol object (bytearray)

```

(continues on next page)

(continued from previous page)

```

10 address = b'1Node'
11
12 # change these (digital output) pins accordingly
13 ce = dio.DigitalInOut(board.D4)
14 csn = dio.DigitalInOut(board.D5)
15
16 # using board.SPI() automatically selects the MCU's
17 # available SPI pins, board.SCK, board.MOSI, board.MISO
18 spi = board.SPI() # init spi bus object
19
20 # we'll be using the dynamic payload size feature (enabled by default)
21 # initialize the nRF24L01 on the spi bus object
22 nrf = RF24(spi, csn, ce)
23
24 # lets create a list of payloads to be streamed to the nRF24L01 running slave()
25 buffers = []
26 SIZE = 32 # we'll use SIZE for the number of payloads in the list and the payloads'
27           ↳length
28 for i in range(SIZE):
29     buff = b''
30     for j in range(SIZE):
31         buff += bytes([(j >= SIZE / 2 + abs(SIZE / 2 - i) or j <
32                               SIZE / 2 - abs(SIZE / 2 - i)) + 48])
33     buffers.append(buff)
34     del buff
35
36 def master(count=1): # count = 5 will transmit the list 5 times
37     """Transmits a massive buffer of payloads"""
38     # set address of RX node into a TX pipe
39     nrf.open_tx_pipe(address)
40     # ensures the nRF24L01 is in TX mode
41     nrf.listen = False
42
43     success_percentage = 0
44     for _ in range(count):
45         now = time.monotonic() * 1000 # start timer
46         result = nrf.send(buffers)
47         print('Transmission took', time.monotonic() * 1000 - now, 'ms')
48         for r in result:
49             success_percentage += 1 if r else 0
50     success_percentage /= SIZE * count
51     print('successfully sent', success_percentage * 100, '%')
52
53 def slave(timeout=5):
54     """Stops listening after timeout with no response"""
55     # set address of TX node into an RX pipe. NOTE you MUST specify
56     # which pipe number to use for RX, we'll be using pipe 0
57     # pipe number options range [0,5]
58     # the pipe numbers used during a transition don't have to match
59     nrf.open_rx_pipe(0, address)
60     nrf.listen = True # put radio into RX mode and power up
61
62     count = 0
63     now = time.monotonic() # start timer
64     while time.monotonic() < now + timeout:
65         if nrf.any():
66             count += 1

```

(continues on next page)



(continued from previous page)

```

66         # retrieve the received packet's payload
67         rx = nrf.recv() # clears flags & empties RX FIFO
68         print("Received (raw): {} - {}".format(repr(rx), count))
69         now = time.monotonic()
70
71         # recommended behavior is to keep in TX mode while idle
72         nrf.listen = False # put the nRF24L01 in TX mode
73
74     print("""\
75         nRF24L01 Stream test\n\
76         Run slave() on receiver\n\
77         Run master() on transmitter""")

```

### 5.1.5 Context Example

This is a test to show how to use “with” statements to manage multiple different nRF24L01 configurations on 1 transceiver.

Listing 5: examples/nrf24l01\_context\_test.py

```

1  """
2  Simple example of library usage in context.
3  This will not transmit anything, but rather
4  display settings after changing contexts ( & thus configurations)
5  """
6  import board
7  import digitalio as dio
8  from circuitpython_nrf24l01 import RF24
9
10 # change these (digital output) pins accordingly
11 ce = dio.DigitalInOut(board.D4)
12 csn = dio.DigitalInOut(board.D5)
13
14 # using board.SPI() automatically selects the MCU's
15 # available SPI pins, board.SCK, board.MOSI, board.MISO
16 spi = board.SPI() # init spi bus object
17
18 # initialize the nRF24L01 objects on the spi bus object
19 nrf = RF24(spi, csn, ce, ack=True)
20 # the first object will have all the features enabled
21 # including the option to use custom ACK payloads
22
23 # the second object has most features disabled/altered
24 # disabled dynamic_payloads, but still using enabled auto_ack
25 # the IRQ pin is configured to only go active on "data fail"
26 # using a different channel: 2 (default is 76)
27 # CRC is set to 1 byte long
28 # data rate is set to 2 Mbps
29 # payload length is set to 8 bytes
30 # NOTE address length is set to 3 bytes
31 # RF power amplifier is set to -12 dbm
32 # automatic retry attempts is set to 15 (maximum allowed)
33 # automatic retry delay (between attempts) is set to 1000 microseconds
34 basicRF = RF24(spi, csn, ce,
35               dynamic_payloads=False, irq_DR=False, irq_DS=False,

```

(continues on next page)

(continued from previous page)

```

36         channel=2, crc=1, data_rate=2, payload_length=8,
37         address_length=3, pa_level=-12, ard=1000, arc=15)
38
39 print("\nsettings configured by the nrf object")
40 with nrf:
41     nrf.open_rx_pipe(5, b'1Node') # NOTE we do this inside the "with" block
42     # only the first character gets written because it is on a pipe_number > 1
43     # NOTE if opening pipes outside of the "with" block, you may encounter
44     # conflicts in the differences between address_length attributes.
45     # the address_length attribute must equal the length of addresses
46
47     # display current settings of the nrf object
48     nrf.what_happened(True) # True dumps pipe info
49
50 print("\nsettings configured by the basicRF object")
51 with basicRF as nerf: # the "as nerf" part is optional
52     nerf.open_rx_pipe(2, b'SOS') # again only uses the first character
53     nerf.what_happened(1)
54
55 # if you examine the outputs from what_happened() you'll see:
56 #   pipe 5 is opened using the nrf object, but closed using the basicRF object.
57 #   pipe 2 is closed using the nrf object, but opened using the basicRF object.
58 # this is because the "with" statements load the existing settings
59 # for the RF24 object specified after the word "with".
60
61 # the things that remain consistent despite the use of "with"
62 # statements includes the power mode (standby or sleep), and
63 # primary role (RX/TX mode)
64 # NOTE this library uses the addresses' reset values and closes all pipes upon
65 # instantiation

```

### 5.1.6 Working with 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/test with the TMRh20 library example named "GettingStarted\_HandlingData.ino".

Listing 6: examples/nrf24l01\_2arduino\_handling\_data.py

```

1  """
2  Example of library driving the nRF24L01 to communicate with a nRF24L01 driven by
3  the TMRh20 Arduino library. The Arduino program/sketch that this example was
4  designed for is named GettingStarted_HandlingData.ino and can be found in the "RF24"
5  examples after the TMRh20 library is installed from the Arduino Library Manager.
6  """
7  import time
8  import struct
9  import board
10 import digitalio as dio
11 from circuitpython_nrf24l01 import RF24
12
13 # addresses needs to be in a buffer protocol object (bytearray)
14 address = [b'1Node', b'2Node']
15
16 # change these (digital output) pins accordingly

```

(continues on next page)

(continued from previous page)

```

17 ce = dio.DigitalInOut(board.D4)
18 csn = dio.DigitalInOut(board.D5)
19
20 # using board.SPI() automatically selects the MCU's
21 # available SPI pins, board.SCK, board.MOSI, board.MISO
22 spi = board.SPI() # init spi bus object
23
24 # initialize the nRF24L01 on the spi bus object
25 nrf = RF24(spi, csn, ce, ask_no_ack=False)
26 nrf.dynamic_payloads = False # this is the default in the TMRh20 arduino library
27
28 # set address of TX node into a RX pipe
29 nrf.open_rx_pipe(1, address[1])
30 # set address of RX node into a TX pipe
31 nrf.open_tx_pipe(address[0])
32
33 def master(count=5): # count = 5 will only transmit 5 packets
34     """Transmits an arbitrary unsigned long value every second. This method
35     will only try to transmit (count) number of attempts"""
36
37     # for the "HandlingData" part of the test from the TMRh20 library example
38     float_value = 0.01
39     while count:
40         nrf.listen = False # ensures the nRF24L01 is in TX mode
41         print("Now Sending")
42         start_timer = int(time.monotonic() * 1000) # start timer
43         # use struct.pack to packetize your data into a usable payload
44         # '<' means little endian byte order.
45         # 'L' means a single 4 byte unsigned long value.
46         # 'f' means a single 4 byte float value.
47         buffer = struct.pack('<Lf', start_timer, float_value)
48         result = nrf.send(buffer)
49         if result is None:
50             print('send() timed out')
51         elif not result:
52             print('send() failed')
53         else:
54             nrf.listen = True # get radio ready to receive a response
55             timeout = True # used to determine if response timed out
56             while time.monotonic() * 1000 - start_timer < 200:
57                 # the arbitrary 200 ms timeout value is also used in the TMRh20_
58
59         example
60         if nrf.any():
61             end_timer = time.monotonic() * 1000 # end timer
62             rx = nrf.recv()
63             rx = struct.unpack('<Lf', rx[:8])
64             timeout = False # skips timeout prompt
65             # print total time to send and receive data
66             print('Sent', struct.unpack('<Lf', buffer), 'Got Response:', rx)
67             print('Round-trip delay:', end_timer - start_timer, 'ms')
68             float_value = rx[1] # save float value for next iteration
69             break
70         if timeout:
71             print("failed to get a response; timed out")
72         count -= 1
73         time.sleep(1)

```

(continues on next page)

(continued from previous page)

```

73 def slave(count=3):
74     """Polls the radio and prints the received value. This method expires
75     after 6 seconds of no received transmission"""
76     start_timer = time.monotonic()
77     while count and (time.monotonic() - start_timer) < 6:
78         nrf.listen = True # put radio into RX mode and power up
79         if nrf.any():
80             # retrieve the received packet's payload
81             buffer = nrf.recv() # clears flags & empties RX FIFO
82             # increment floating value as part of the "HandlingData" test
83             float_value = struct.unpack('<f', buffer[4:8])[0] + 0.01
84             nrf.listen = False # ensures the nRF24L01 is in TX mode
85             start_timer = time.monotonic() # in seconds
86             # echo buffer[:4] appended with incremented float
87             result = nrf.send(buffer[:4] + struct.pack('<f', float_value))
88             end_timer = time.monotonic() * 1000 # in milliseconds
89             # expecting an unsigned long & a float, thus the string format '<Lf'
90             rx = struct.unpack('<Lf', buffer[:8]) # "[:8]" ignores the padded 0s
91             # print the unsigned long and float data sent in the response
92             print("Responding: {}, {}".format(rx[0], rx[1] + 0.01))
93             if result is None:
94                 print('response timed out')
95             elif not result:
96                 print('response failed')
97             else:
98                 # print timer results on transmission success
99                 print('successful response took', end_timer - start_timer * 1000, 'ms
100 → ')
101             # this will listen indefinitely till counter == 0
102             count -= 1
103             # recommended behavior is to keep in TX mode when in idle
104             nrf.listen = False # put the nRF24L01 in TX mode + Standby-I power state
105
106 print("""\
107 nRF24L01 communicating with an Arduino running the\n\
108 TMRh20 library's "GettingStarted_HandlingData.ino" example.\n\
109 Run slave() on receiver\n\
110 Run master() on transmitter""")

```

### 5.1.7 RF24 class

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

1. *dynamic\_payloads* feature allowing 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.
2. *auto\_ack* feature provides transmission verification by using the RX nRF24L01 to automatically and immediately send an acknowledgment (ACK) packet in response to freshly received payloads. *auto\_ack* does not require *dynamic\_payloads* to be enabled.
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.

---

Remember 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:

- cycle 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 vehicle that you (the payload) get into. Continuing the analogy, the specified address is not the address of an nRF24L01 radio, rather it is more like a route that connects the endpoints. There are only six data pipes on the nRF24L01, thus it can simultaneously listen to a maximum of 6 other nRF24L01 radios (can only talk to 1 at a time). When assigning addresses to a data pipe, you can use any 5 byte long address you can think of (as long as the last byte is unique among simultaneously broadcasting addresses), so you're not limited to communicating to the same 6 radios (more on this when we support "Multiciever" mode). Also 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) ambient signals.

---

**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 MUST match the TX pipe's address on the transmitting nRF24L01
- `address_length`
- `channel`
- `data_rate`
- `dynamic_payloads`
- `payload_length` only when `dynamic_payloads` is disabled
- `auto_ack`
- 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()`), `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).

### 5.1.7.1 Basic API

```
class circuitpython_nrf24l01.rf24.RF24(spi, csn, ce, channel=76, payload_length=32,
                                         address_length=5, ard=1500, arc=3, crc=2,
                                         data_rate=1, pa_level=0, dynamic_payloads=True,
                                         auto_ack=True, ask_no_ack=True, ack=False,
                                         irq_DR=True, irq_DS=True, irq_DF=True)
```

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.
- **channel** (*int*) – This is used to specify a certain radio frequency that the nRF24L01 uses. Defaults to 76 and can be changed at any time by using the `channel` attribute.
- **payload\_length** (*int*) – This is the length (in bytes) of a single payload to be transmitted or received. This is ignored if the `dynamic_payloads` attribute is enabled. Defaults to 32 and must be in range [1,32]. This can be changed at any time by using the `payload_length` attribute.
- **address\_length** (*int*) – This is the length (in bytes) of the addresses that are assigned to the data pipes for transmitting/receiving. Defaults to 5 and must be in range [3,5]. This can be changed at any time by using the `address_length` attribute.
- **ard** (*int*) – This specifies the delay time (in  $\mu$ s) between attempts to automatically re-transmit. This can be changed at any time by using the `ard` attribute. This parameter must be a multiple of 250 in the range [250,4000]. Defaults to 1500  $\mu$ s.
- **arc** (*int*) – This specifies the automatic re-transmit count (maximum number of automatically attempts to re-transmit). This can be changed at any time by using the `arc` attribute. This parameter must be in the range [0,15]. Defaults to 3.
- **crc** (*int*) – This parameter controls the CRC setting of transmitted packets. Options are 0 (off), 1 or 2 (byte long CRC enabled). This can be changed at any time by using the `crc` attribute. Defaults to 2.
- **data\_rate** (*int*) – This parameter controls the RF data rate setting of transmissions. Options are 1 (Mbps), 2 (Mbps), or 250 (Kbps). This can be changed at any time by using the `data_rate` attribute. Defaults to 1.
- **pa\_level** (*int*) – This parameter controls the RF power amplifier setting of transmissions. Options are 0 (dBm), -6 (dBm), -12 (dBm), or -18 (dBm). This can be changed at any time by using the `pa_level` attribute. Defaults to 0.

- **dynamic\_payloads** (*bool*) – This parameter enables/disables the dynamic payload length feature of the nRF24L01. Defaults to enabled. This can be changed at any time by using the *dynamic\_payloads* attribute.
- **auto\_ack** (*bool*) – This parameter enables/disables the automatic acknowledgment (ACK) feature of the nRF24L01. Defaults to enabled if *dynamic\_payloads* is enabled. This can be changed at any time by using the *auto\_ack* attribute.
- **ask\_no\_ack** (*bool*) – This represents a special flag that has to be thrown to enable a feature specific to individual payloads. Setting this parameter only enables access to this feature; it does not invoke it (see parameters for *send()* or *write()* functions). Enabling/Disabling this does not affect *auto\_ack* attribute.
- **ack** (*bool*) – This represents a special flag that has to be thrown to enable a feature allowing custom response payloads appended to the ACK packets. Enabling this also requires the *auto\_ack* attribute enabled. This can be changed at any time by using the *ack* attribute.
- **irq\_DR** (*bool*) – When “Data is Ready”, this configures the interrupt (IRQ) trigger of the nRF24L01’s IRQ pin (active low). Defaults to enabled. This can be changed at any time by using the *interrupt\_config()* function.
- **irq\_DS** (*bool*) – When “Data is Sent”, this configures the interrupt (IRQ) trigger of the nRF24L01’s IRQ pin (active low). Defaults to enabled. This can be changed at any time by using the *interrupt\_config()* function.
- **irq\_DF** (*bool*) – When “max retry attempts are reached” (specified by the *arc* attribute), this configures the interrupt (IRQ) trigger of the nRF24L01’s IRQ pin (active low) and represents transmission failure. Defaults to enabled. This can be changed at any time by using the *interrupt\_config()* function.

#### address\_length

This *int* attribute specifies the length (in bytes) of addresses to be used for RX/TX pipes. The addresses assigned to the data pipes must have byte length equal to the value set for this attribute.

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.

#### open\_tx\_pipe (address)

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

**Parameters** **address** (*bytearray*) – The virtual address of the receiving nRF24L01. This must have a length equal to the *address\_length* attribute (see *address\_length* attribute). Otherwise a *ValueError* exception is thrown. The address specified here must match the address set to one of the RX data pipes of the receiving nRF24L01.

---

**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. Thus, RX pipe 0 is appropriated with the TX address (specified here) when *auto\_ack* is set to *True*.

---

#### close\_rx\_pipe (pipe\_number, reset=True)

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 *ValueError* exception is thrown.



- **reset** (*bool*) – `True` resets the address for the specified `pipe_number` to the factory address (different for each pipe). `False` leaves the address on the specified `pipe_number` alone. Be aware that the addresses will remain despite loss of power.

#### **open\_rx\_pipe** (*pipe\_number*, *address*)

This function is used to open a specific data pipe for OTA (over the air) RX transmissions. If `dynamic_payloads` attribute is `False`, then the `payload_length` attribute is used to specify the expected length of the RX payload on the specified data pipe.

##### **Parameters**

- **pipe\_number** (*int*) – The data pipe to use for RX transactions. This must be in range [0,5]. Otherwise a `ValueError` exception is thrown.
- **address** (*bytearray*) – The virtual address to the receiving nRF24L01. This must have a byte length equal to the `address_length` attribute. Otherwise a `ValueError` exception is thrown. 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).

---

**Note:** The nRF24L01 shares the addresses' LSBytes (address[1:5]) on data pipes 2 through 5. These shared LSBytes are determined by the address set to pipe 1.

---

#### **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 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. Disabling 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).

#### **any()**

This function checks if the nRF24L01 has received any data at all. Internally, this function uses `pipe()` then reports the next available payload's length (in bytes) – if there is any.

##### **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.

#### **recv()**

This function is used to retrieve the next available payload in the RX FIFO buffer, then clears the `irq_DR` status flag. This function also serves as a helper function to `read_ack()` in TX mode to acquire any custom payload in the automatic acknowledgement (ACK) packet – only when the `ack` attribute is enabled.

##### **Returns**

A `bytearray` of the RX payload data



- If the `dynamic_payloads` attribute is disabled, then the returned bytearray's length is equal to the user defined `payload_length` attribute (which defaults to 32).
- If the `dynamic_payloads` attribute is enabled, then the returned bytearray's length is equal to the payload's length

---

**Tip:** Call the `any()` function before calling `recv()` to verify that there is data to fetch. If there's no data to fetch, then the nRF24L01 returns bogus data and should not be regarded as a valid payload.

---

**send** (*buf*, *ask\_no\_ack=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.
- `True` if transmission succeeds.
- `bytearray` when the `ack` attribute is `True`, the payload expects a responding custom ACK payload; the response is returned (upon successful transmission) as a `bytearray`. Empty ACK payloads (upon successful transmission) when the `ack` attribute is set `True` are replaced with an error message `b'NO ACK RETURNED'`.
- `None` if transmission times out meaning nRF24L01 has malfunctioned. This condition is very rare. The allowed time for transmission is calculated using [table 18 in the nRF24L01 specification sheet](#)

#### Parameters

- **buf** (*bytearray*, *list*, *tuple*) – The payload to transmit. This bytearray must have a length greater than 0 and less than 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 and this bytearray's length is less than the `payload_length` attribute, then this bytearray is padded with zeros until its length is equal to the `payload_length` attribute.
  - If the `dynamic_payloads` attribute is disabled and this bytearray's length is greater than `payload_length` attribute, then this bytearray's length is truncated to equal the `payload_length` attribute.
- **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 `auto_ack` attribute is disabled, however this parameter should work despite the `auto_ack` 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.

---

**Tip:** It is highly recommended that `auto_ack` attribute is enabled when sending multiple payloads. Test results with the `auto_ack` attribute disabled were very poor (much < 50% received). 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 when sending a list or tuple of payloads, so it can continue to process through the list/tuple even if any payload fails to be acknowledged.

**Note:** We've tried very hard to keep nRF24L01s driven by CircuitPython devices compliant with nRF24L01s driven by the Raspberry Pi. But due to the Raspberry Pi's seemingly slower SPI speeds, we've had to resort to internally deploying `resend()` twice (at most when needed) for payloads that failed during multi-payload processing. This tactic is meant to slow down CircuitPython devices just enough for the Raspberry Pi to catch up. Transmission failures are less possible this way.

---

### 5.1.7.2 Advanced API

**class** `circuitpython_nrf24l01.rf24.RF24`

`RF24.what_happened(dump_pipes=False)`

This debugging function aggregates and outputs all status/condition related information from the nRF24L01. Some information may be irrelevant depending on nRF24L01's state/condition.

#### Prints

- 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
- Payload lengths The current setting of the `payload_length` attribute
- Auto retry delay The current setting of the `ard` attribute
- Auto retry attempts The current setting of the `arc` attribute
- Packets Lost Total amount of packets lost (transmission failures)
- Retry Attempts Made Maximum 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` Is the `auto_ack` attribute enabled?
- `Dynamic Payloads` Is the `dynamic_payloads` attribute enabled?
- `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
- 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 read directly from the nRF24L01 registers.
- 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.

Default is `False` and skips this extra information.

#### RF24.`dynamic_payloads`

This `bool` attribute controls the nRF24L01's dynamic payload length feature.

- `True` enables nRF24L01's dynamic payload length feature. The `payload_length` attribute is ignored when this feature is enabled.
- `False` disables nRF24L01's dynamic payload length feature. Be sure to adjust the `payload_length` attribute accordingly when `dynamic_payloads` feature is disabled.

#### RF24.`payload_length`

This `int` attribute specifies the length (in bytes) of payload that is regarded, meaning "how big of a payload should the radio care about?" If the `dynamic_payloads` attribute is enabled, this attribute has no affect. When `dynamic_payloads` is disabled, this attribute is used to specify the payload length when entering RX mode.

A valid input value must be an `int` in range [1,32]. Otherwise a `ValueError` exception is thrown. Default is set to the nRF24L01's maximum of 32.

---

**Note:** When `dynamic_payloads` is disabled during transmissions:

- Payloads' size of greater than this attribute's value will be truncated to match.
- Payloads' size of less than this attribute's value will be padded with zeros to match.

**RF24.auto\_ack**

This `bool` attribute controls the nRF24L01's automatic acknowledgment feature.

- `True` enables automatic acknowledgment packets. The CRC (cyclic redundancy checking) is enabled automatically by the nRF24L01 if the `auto_ack` attribute is enabled (see also `crc` attribute).
- `False` disables automatic acknowledgment packets. The `crc` attribute will remain unaffected (remains enabled) when disabling the `auto_ack` attribute.

**RF24.irq\_DR**

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

- `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 `dataReady` 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.

**RF24.irq\_DF**

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

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

Pass `dataFail` parameter as `True` to `clear_status_flags()` to 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. see also the `arc` and `ard` attributes.

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.

**RF24.irq\_DS**

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

- `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 `dataSent` parameter as `True` to `clear_status_flags()` to 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.

**RF24.clear\_status\_flags** (`data_rcv=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\_rcv** (*bool*) – specifies wheather to clear the “RX Data Ready” flag.
- **data\_sent** (*bool*) – specifies wheather to clear the “TX Data Sent” flag.
- **data\_fail** (*bool*) – specifies wheather to clear the “Max Re-transmit reached” 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.

---

RF24.**interrupt\_config** (*data\_rcv=True, data\_sent=True, data\_fail=True*)

Sets the configuration of the nRF24L01’s IRQ (interrupt) pin. The signal from the nRF24L01’s IRQ pin is active LOW. (write-only)

#### Parameters

- **data\_rcv** (*bool*) – If this is `True`, then IRQ pin goes active when there is new data to read in the RX FIFO buffer.
- **data\_sent** (*bool*) – If this is `True`, then IRQ pin goes active when a payload from TX buffer is successfully transmit.
- **data\_fail** (*bool*) – If this is `True`, then IRQ pin goes active when maximum number of attempts to re-transmit the packet have been reached. If `auto_ack` attribute is disabled, then this IRQ event is not used.

---

**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 `data_rcv` IRQ should be:

1. read payload through `recv()`
  2. clear `dataReady` 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()`, `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
- 

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.

- `True` enables the use of custom ACK payloads in the ACK packet when responding to receiving transmissions. As `dynamic_payloads` and `auto_ack` attributes are required for this feature to work, they are automatically enabled as needed.
- `False` disables the use of custom ACK payloads. Disabling this feature does not disable the `auto_ack` and `dynamic_payloads` attributes (they work just fine without this feature).

**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 fresh data is received on the specified pipe. See `read_ack()` on how to fetch a received custom ACK payloads.

**Parameters**

- **buf** (*bytearray*) – 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 `ValueError` 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 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.

---

**RF24.read\_ack()**

Allows user to read the automatic acknowledgement (ACK) payload (if any) when nRF24L01 is in TX mode. This function is called from a blocking `send()` call if the `ack` attribute is enabled. Alternatively, this function can be called directly in case of calling the non-blocking `write()` function during asynchronous applications.

**Warning:** In the case of asynchronous applications, this function will do nothing if the status flags are cleared after calling `write()` and before calling this function. See also the `ack`, `dynamic_payloads`, and `auto_ack` attributes as they must be enabled to use custom ACK payloads.

**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

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

**Warning:** 250 Kbps is be buggy on the non-plus models of the nRF24L01 product line. If you use 250 Kbps data rate, and some transmissions report failed by the transmitting nRF24L01, even though the same packet in question actually reports received by the receiving nRF24L01, then try a higher data rate. CAUTION: Higher data rates mean less maximum distance between nRF24L01 transceivers (and vise versa).

#### RF24.**channel**

This `int` attribute specifies the nRF24L01's frequency (in  $2400 + \text{channel}$  MHz).

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.

#### RF24.**crc**

This `int` attribute specifies the nRF24L01's CRC (cyclic redundancy checking) encoding scheme in terms of byte length.

A valid input value is in range [0,2]:

- 0 disables CRC
- 1 enables CRC encoding scheme using 1 byte
- 2 enables CRC encoding scheme using 2 bytes

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).

---

#### RF24.**power**

This `bool` attribute controls the power state of the nRF24L01. This is exposed for asynchronous applications and user preference.

- `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 transmissions. This driver class doesn't power down the nRF24L01 after RX/TX transmissions are complete (avoiding the required power up/down 130  $\mu$ s wait time), that preference is left to the user.
- `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. TX transmissions are only executed during Standby-II by calling `send()` or `write()`. RX transmissions are received during Standby-II by setting `listen` attribute to `True` (see Chapter 6.1.2-7 of the nRF24L01+ Specifications Sheet). After using `send()` or setting `listen` to `False`, the nRF24L01 is left in Standby-I mode (see also notes on the `write()` function).

---

#### RF24.**arc**

"This `int` attribute specifies the nRF24L01's number of attempts to re-transmit TX payload when acknowledgment packet is not received. The nRF24L01 does not attempt to re-transmit if `auto_ack` attribute is disabled.



A valid input value must be in range [0,15]. Otherwise a `ValueError` exception is thrown. Default is set to 3.

#### RF24.ard

This `int` attribute specifies the nRF24L01's delay (in  $\mu$ s) between attempts to automatically re-transmit 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 `auto_ack` attribute is disabled, this attribute is not applied.

A valid input value must be a multiple of 250 in range [250,4000]. Otherwise a `ValueError` exception is thrown. Default is 1500 for reliability.

---

**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 500 $\mu$ s or more. If the custom ACK payload is more than 5 bytes in 1 Mbps data rate, the `ard` must be 500 $\mu$ s or more. In 250kbps data rate (even when there is no custom ACK payload) the `ard` must be 500 $\mu$ s or more.

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

---

#### 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)

Any invalid input throws a `ValueError` exception. Default is 0 dBm.

#### 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 SPI transactions with the nRF24L01. Use the `update()` function to manually refresh this data when needed.

##### 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.

#### RF24.update()

This function is only used to get an updated status byte over SPI from the nRF24L01 and is exposed to the MCU for asynchronous applications. Refreshing the status byte is vital to checking status of the interrupts, 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`, and `tx_full` attributes. Internally this is a helper function to `pipe()`, `send()`, and `resend()` functions

#### RF24.resend()

Use this function to manually re-send the previously failed-to-transmit payload in the top level (first out) of the TX FIFO buffer.



---

**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.

---

RF24.**write** (*buf=None, ask\_no\_ack=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()`.

#### 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 and this bytearray’s length is less than the `payload_length` attribute, then this bytearray is padded with zeros until its length is equal to the `payload_length` attribute.
  - If the `dynamic_payloads` attribute is disabled and this bytearray’s length is greater than `payload_length` attribute, then this bytearray’s length is truncated to equal the `payload_length` attribute.
- **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 `auto_ack` attribute is disabled, however this parameter should work despite the `auto_ack` 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.

---

This function isn’t completely non-blocking as we still need to wait just under 5 ms for the CSN pin to settle (allowing a clean SPI transaction).

---

**Note:** The nRF24L01 doesn’t initiate sending until a mandatory minimum 10  $\mu$ s pulse on the CE pin is achieved. That pulse is initiated before this function exits. However, we have left that 10  $\mu$ s wait time to be managed by the MCU in cases of asynchronous application, or it is managed by using `send()` instead of this function. If the CE pin remains HIGH for longer than 10  $\mu$ s, 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 [`auto_ack` and `dynamic_payloads`] features are 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 you enable the `dynamic_payloads` and `auto_ack`

attributes. Alternatively, you MUST use interrupt flags or IRQ pin with 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 necessary transmission time (these calculations are used in the `send()` function).

---

#### RF24.`flush_rx()`

A helper function to flush the nRF24L01's internal RX FIFO buffer. (write-only)

---

**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 popped from the stack) by `recv()` or `read_ack()`. This function clears all 3 levels.

---

#### RF24.`flush_tx()`

A helper function to flush the nRF24L01's internal TX FIFO buffer. (write-only)

---

**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 popped from the TX FIFO stack upon successful transmission (see also `resend()` as the handling of failed transmissions can be altered).

---

#### RF24.`fifo` (*tx=False, empty=None*)

This provides some precision determining the status of the TX/RX FIFO buffers. (read-only)

##### Parameters

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

##### Returns

- A `bool` answer to the question: “Is the [TX/RX]:[`True/False`] FIFO buffer [empty/full]:[`True/False`]?”
- If the `empty` parameter is not specified: an `int` in range [0,2] for which:
  - 1 means the specified FIFO buffer is full
  - 2 means the specified FIFO buffer is empty
  - 0 means the specified FIFO buffer is neither full nor empty

#### RF24.`pipe()`

This function returns information about the data pipe that received the next available payload in the RX FIFO buffer.

**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.

## 5.2 Indices and tables

- `genindex`
- `modindex`
- `search`



**A**

```
flush_tx() (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 33
address_length (circuitpython_nrf24l01.rf24.RF24
attribute), 27
any() (circuitpython_nrf24l01.rf24.RF24 method), 28
arc (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24
attribute), 35
ard (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 36
auto_ack (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 32
interrupt_config() (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
method), 33
```

**C**

```
channel (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 35
circuitpython_nrf24l01.rf24.RF24 (class in listen (circuitpython_nrf24l01.rf24.RF24 attribute),
circuitpython_nrf24l01.rf24), 30
clear_status_flags() (circuit- load_ack() (circuit-
python_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24 python_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
method), 32 method), 33
close_rx_pipe() (circuit-
python_nrf24l01.rf24.RF24 method), 27
crc (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 35
```

D

```
open_tx_pipe() (circuitpython_nrf24l01.rf24.RF24
method), 27
data_rate(circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24
attribute), 34
dynamic_payloads (circuit- pa_level(circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24
python_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24#attribute), 36
attribute), 31
payload_length (circuit-
```

```
F
python_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 31
fifo() (circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
method), 38
flush_rx() (circuit- power(circuitpython_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
python_nrf24l01.rf24.circuitpython_nrf24l01.rf24.RF24.RF24
attribute), 35
method), 38
```

## R

`read_ack()` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 method*), 34

`recv()` (*circuitpython\_nrf24l01.rf24.RF24 method*), 28

`resend()` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 method*), 36

`RF24` (*class in circuitpython\_nrf24l01.rf24*), 26

## S

`send()` (*circuitpython\_nrf24l01.rf24.RF24 method*), 29

## T

`tx_full` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 attribute*), 36

## U

`update()` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 method*), 36

## W

`what_happened()` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 method*), 30

`write()` (*circuitpython\_nrf24l01.rf24.circuitpython\_nrf24l01.rf24.RF24.RF24 method*), 37