

---

# **nRF24L01 Library Documentation**

***Release 1.2***

**Brendan Doherty**

**Oct 06, 2020**



<b>1</b>	<b>Getting Started</b>	<b>1</b>
1.1	Features currently supported . . . . .	1
1.2	Features currently unsupported . . . . .	2
1.3	Dependencies . . . . .	2
1.4	Installing from PyPI . . . . .	2
<b>2</b>	<b>Pinout</b>	<b>5</b>
<b>3</b>	<b>Using The Examples</b>	<b>7</b>
<b>4</b>	<b>What to purchase</b>	<b>9</b>
4.1	Power Stability . . . . .	9
4.2	About the nRF24L01+PA+LNA modules . . . . .	9
4.3	nRF24L01(+) clones and counterfeits . . . . .	10
<b>5</b>	<b>Contributing</b>	<b>11</b>
5.1	Future Project Ideas/Additions . . . . .	11
5.2	Sphinx documentation . . . . .	11
<b>6</b>	<b>Simple test</b>	<b>13</b>
<b>7</b>	<b>ACK Payloads Example</b>	<b>17</b>
<b>8</b>	<b>Stream Example</b>	<b>21</b>
<b>9</b>	<b>Fake BLE Example</b>	<b>25</b>
<b>10</b>	<b>Context Example</b>	<b>29</b>
<b>11</b>	<b>Working with TMRh20's Arduino library</b>	<b>31</b>
<b>12</b>	<b>IRQ Pin Example</b>	<b>35</b>
<b>13</b>	<b>Troubleshooting info</b>	<b>39</b>
<b>14</b>	<b>About the lite version</b>	<b>41</b>
<b>15</b>	<b>Testing nRF24L01+PA+LNA module</b>	<b>43</b>
15.1	The Setup . . . . .	43

15.2	Results (ordered by pa_level settings)	43
15.3	Conclusion	44
<b>16</b>	<b>Basic API</b>	<b>45</b>
16.1	Constructor	45
16.2	open_tx_pipe()	45
16.3	close_rx_pipe()	46
16.4	open_rx_pipe()	46
16.5	listen	46
16.6	any()	47
16.7	recv()	47
16.8	send()	48
<b>17</b>	<b>Advanced API</b>	<b>51</b>
17.1	what_happened()	51
17.2	is_plus_variant	52
17.3	load_ack()	52
17.4	read_ack()	53
17.5	irq_dr	54
17.6	irq_df	54
17.7	irq_ds	54
17.8	clear_status_flags()	55
17.9	power	55
17.10	tx_full	56
17.11	update()	56
17.12	resend()	56
17.13	write()	56
17.14	flush_rx()	58
17.15	flush_tx()	58
17.16	fifo()	58
17.17	pipe	59
17.18	address_length	59
17.19	address()	59
17.20	rpd	59
17.21	start_carrier_wave()	60
17.22	stop_carrier_wave()	60
<b>18</b>	<b>Configuration API</b>	<b>61</b>
18.1	CSN_DELAY	61
18.2	dynamic_payloads	61
18.3	payload_length	62
18.4	auto_ack	62
18.5	arc	62
18.6	ard	63
18.7	ack	63
18.8	interrupt_config()	63
18.9	data_rate	64
18.10	channel	64
18.11	crc	65
18.12	pa_level	65
18.13	is_lna_enabled	65
<b>19</b>	<b>BLE Limitations</b>	<b>67</b>
<b>20</b>	<b>helpers</b>	<b>69</b>

20.1	swap_bits()	69
20.2	reverse_bits()	69
20.3	chunk()	69
20.4	crc24_ble()	70
20.5	BLE_FREQ	70
<b>21</b>	<b>FakeBLE class</b>	<b>71</b>
21.1	to_iphone	71
21.2	mac	72
21.3	name	72
21.4	show_pa_level	72
21.5	hop_channel()	72
21.6	whiten()	72
21.7	available()	73
21.8	advertise()	73
21.9	Available RF24 functionality	74
21.9.1	pa_level	74
21.9.2	channel	74
21.9.3	payload_length	74
21.9.4	power	74
21.9.5	is_lna_enabled	74
21.9.6	is_plus_variant	74
21.9.7	interrupt_config()	74
21.9.8	irq_ds	75
21.9.9	irq_dr	75
21.9.10	clear_status_flags()	75
21.9.11	update()	75
21.9.12	what_happened()	75
<b>22</b>	<b>Service related classes</b>	<b>77</b>
22.1	abstract parent	77
22.2	derivative children	77
<b>23</b>	<b>Indices and tables</b>	<b>79</b>
	<b>Index</b>	<b>81</b>



# 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 (MUST read documentation on the `send()` function)
- 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 advertisements from the nRF24L01 as outlined by [Dmitry Grinberg](#) in his write-up (including C source code).

## 1.2 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), but this has not been tested.

## 1.3 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.

---

## 1.4 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:

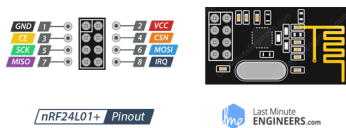


```
mkdir project-name && cd project-name
python3 -m venv .env
source .env/bin/activate
pip3 install circuitpython-nrf24l01
```



## 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 microfarads [+ another optional 0.1 microfarads capacitor for added stability] connected in parallel 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
```



---

## 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 vice 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 capacitor(s) (100  $\mu$ F + an optional 0.1 $\mu$ F) in parallel (& as close as possible) to the VCC and GND pins is highly recommended. Stabilizing 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 separate 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 $\mu$ A

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).



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

---

### Simple test

---

Ensure your device works with this simple test.

Listing 1: examples/nrf24l01\_simple\_test.py

```
1  """
2  Simple example of using the RF24 class.
3  """
4  import time
5  import struct
6  import board
7  import digitalio as dio
8  # if running this on a ATSAM21 M0 based board
9  # from circuitpython_nrf24l01.rf24_lite import RF24
10 from circuitpython_nrf24l01.rf24 import RF24
11
12 # addresses needs to be in a buffer protocol object (bytearray)
13 address = b"1Node"
14
15 # change these (digital output) pins accordingly
16 ce = dio.DigitalInOut(board.D4)
17 csn = dio.DigitalInOut(board.D5)
18
19 # using board.SPI() automatically selects the MCU's
20 # available SPI pins, board.SCK, board.MOSI, board.MISO
21 spi = board.SPI() # init spi bus object
22
23 # we'll be using the dynamic payload size feature (enabled by default)
24 # initialize the nRF24L01 on the spi bus object
25 nrf = RF24(spi, csn, ce)
26
27 # set the Power Amplifier level to -12 dBm since this test example is
28 # usually run with nRF24L01 transceivers in close proximity
29 nrf.pa_level = -12
30
```

(continues on next page)

(continued from previous page)

```

31
32 def master(count=5): # count = 5 will only transmit 5 packets
33     """Transmits an incrementing integer every second"""
34     # set address of RX node into a TX pipe
35     nrf.open_tx_pipe(address)
36     # ensures the nRF24L01 is in TX mode
37     nrf.listen = False
38
39     while count:
40         # use struct.pack to packetize your data
41         # into a usable payload
42         buffer = struct.pack("<i", count)
43         # 'i' means a single 4 byte int value.
44         # '<' means little endian byte order. this may be optional
45         print("Sending: {} as struct: {}".format(count, buffer))
46         now = time.monotonic() * 1000 # start timer
47         result = nrf.send(buffer)
48         if not result:
49             print("send() failed or timed out")
50         else:
51             print("send() successful")
52             # print timer results despite transmission success
53             print("Transmission took", time.monotonic() * 1000 - now, "ms")
54             time.sleep(1)
55             count -= 1
56
57
58 def slave(count=5):
59     """Polls the radio and prints the received value. This method expires
60     after 6 seconds of no received transmission"""
61     # set address of TX node into an RX pipe. NOTE you MUST specify
62     # which pipe number to use for RX, we'll be using pipe 0
63     # pipe number options range [0,5]
64     # the pipe numbers used during a transition don't have to match
65     nrf.open_rx_pipe(0, address)
66     nrf.listen = True # put radio into RX mode and power up
67
68     start = time.monotonic()
69     while count and (time.monotonic() - start) < 6:
70         if nrf.any():
71             # print details about the received packet (if any)
72             print("Found {} bytes on pipe {}".format(nrf.any(), nrf.pipe))
73             # retrieve the received packet's payload
74             rx = nrf.recv() # clears flags & empties RX FIFO
75             # expecting an int, thus the string format '<i'
76             buffer = struct.unpack("<i", rx[:4])
77             # print the only item in the resulting tuple from
78             # using `struct.unpack()`
79             print("Received: {}, Raw: {}".format(buffer[0], repr(rx)))
80             start = time.monotonic()
81             count -= 1
82             # this will listen indefinitely till count == 0
83             time.sleep(0.25)
84
85     # recommended behavior is to keep in TX mode while idle
86     nrf.listen = False # put the nRF24L01 is in TX mode
87

```

(continues on next page)

(continued from previous page)

```
88
89 print(
90     """\
91     nRF24L01 Simple test.\n\
92     Run slave() on receiver\n\
93     Run master() on transmitter"""
94 )
```



## ACK Payloads Example

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

Listing 1: 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  # if running this on a ATSAM21 M0 based board
9  # from circuitpython_nrf24l01.rf24_lite import RF24
10 from circuitpython_nrf24l01.rf24 import RF24
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 # the custom ACK payload feature is disabled by default
22 # the custom ACK payload feature should not be enabled
23 # during instantiation due to its singular use nature
24 # meaning 1 ACK payload per 1 RX'd payload
25 nrf = RF24(spi, csn, ce)
26
27 # NOTE the the custom ACK payload feature will be enabled
28 # automatically when you call load_ack() passing:
29 # a buffer protocol object (bytearray) of
30 # length ranging [1,32]. And pipe number always needs

```

(continues on next page)

(continued from previous page)

```

31 # to be an int ranging [0,5]
32
33 # to enable the custom ACK payload feature
34 nrf.ack = True # False disables again
35
36 # set the Power Amplifier level to -12 dBm since this test example is
37 # usually run with nRF24L01 transceivers in close proximity
38 nrf.pa_level = -12
39
40 # addresses needs to be in a buffer protocol object (bytearray)
41 address = b"1Node"
42
43 # NOTE ACK payloads (like regular payloads and addresses)
44 # need to be in a buffer protocol object (bytearray)
45 ACK = b"World "
46
47
48 def master(count=5): # count = 5 will only transmit 5 packets
49     """Transmits a dummy payload every second and prints the ACK payload"""
50     # recommended behavior is to keep in TX mode while idle
51     nrf.listen = False # put radio in TX mode
52
53     # set address of RX node into a TX pipe
54     nrf.open_tx_pipe(address)
55
56     while count:
57         buffer = b"Hello " + bytes([count + 48]) # output buffer
58         print("Sending (raw): {}".format(repr(buffer)))
59         # to read the ACK payload during TX mode we
60         # pass the parameter read_ack as True.
61         nrf.ack = True # enable feature before send()
62         now = time.monotonic() * 1000 # start timer
63         result = nrf.send(buffer) # becomes the response buffer
64         if not result:
65             print("send() failed or timed out")
66         else:
67             # print the received ACK that was automatically
68             # fetched and saved to "buffer" via send()
69             print("raw ACK: {}".format(repr(result)))
70             # the ACK payload should now be in buffer
71             # print timer results despite transmission success
72             print("Transmission took", time.monotonic() * 1000 - now, "ms")
73             time.sleep(1)
74             count -= 1
75
76
77 def slave(count=5):
78     """Prints the received value and sends a dummy ACK payload"""
79     # set address of TX node into an RX pipe. NOTE you MUST specify
80     # which pipe number to use for RX, we'll be using pipe 0
81     nrf.open_rx_pipe(0, address)
82
83     # put radio into RX mode, power it up, and set the first
84     # transmission's ACK payload and pipe number
85     nrf.listen = True
86     buffer = ACK + bytes([count + 48])
87     # we must set the ACK payload data and corresponding

```

(continues on next page)



(continued from previous page)

```

88     # pipe number [0,5]
89     nrf.load_ack(buffer, 0)  # load ACK for first response
90
91     start = time.monotonic()
92     while count and (time.monotonic() - start) < (count * 2):
93         if nrf.any():
94             # this will listen indefinitely till count == 0
95             count -= 1
96             # print details about the received packet (if any)
97             print("Found {} bytes on pipe {}".format(nrf.any(), nrf.pipe))
98             # retrieve the received packet's payload
99             rx = nrf.recv()  # clears flags & empties RX FIFO
100            print("Received (raw): {}".format(repr(rx)))
101            start = time.monotonic()
102            if count:  # Going again?
103                # build new ACK
104                buffer = ACK + bytes([count + 48])
105                # load ACK for next response
106                nrf.load_ack(buffer, 0)
107
108            # recommended behavior is to keep in TX mode while idle
109            nrf.listen = False  # put radio in TX mode
110            nrf.flush_tx()  # flush any ACK payload
111
112
113 print(
114     """\
115     nRF24L01 ACK test\n\
116     Run slave() on receiver\n\
117     Run master() on transmitter"""
118 )

```



---

## Stream Example

---

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

Listing 1: `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  # if running this on a ATSAM21 M0 based board
8  # from circuitpython_nrf24l01.rf24_lite import RF24
9  from circuitpython_nrf24l01.rf24 import RF24
10
11 # addresses needs to be in a buffer protocol object (bytearray)
12 address = b"1Node"
13
14 # change these (digital output) pins accordingly
15 ce = dio.DigitalInOut(board.D4)
16 csn = dio.DigitalInOut(board.D5)
17
18 # using board.SPI() automatically selects the MCU's
19 # available SPI pins, board.SCK, board.MOSI, board.MISO
20 spi = board.SPI() # init spi bus object
21
22 # we'll be using the dynamic payload size feature (enabled by default)
23 # initialize the nRF24L01 on the spi bus object
24 nrf = RF24(spi, csn, ce)
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
```

(continues on next page)

(continued from previous page)

```

31 def master(count=1): # count = 5 will transmit the list 5 times
32     """Transmits a massive buffer of payloads"""
33     # lets create a `list` of payloads to be streamed to
34     # the nRF24L01 running slave()
35     buffers = []
36     # we'll use SIZE for the number of payloads in the list and the
37     # payloads' length
38     size = 32
39     for i in range(size):
40         # prefix payload with a sequential letter to indicate which
41         # payloads were lost (if any)
42         buff = bytes([i + (65 if 0 <= i < 26 else 71)])
43         for j in range(size - 1):
44             char = bool(j >= (size - 1) / 2 + abs((size - 1) / 2 - i))
45             char |= bool(j < (size - 1) / 2 - abs((size - 1) / 2 - i))
46             buff += bytes([char + 48])
47         buffers.append(buff)
48         del buff
49
50     # set address of RX node into a TX pipe
51     nrf.open_tx_pipe(address)
52     # ensures the nRF24L01 is in TX mode
53     nrf.listen = False
54
55     successful = 0
56     for _ in range(count):
57         now = time.monotonic() * 1000 # start timer
58         result = nrf.send(buffers, force_retry=2)
59         print("Transmission took", time.monotonic() * 1000 - now, "ms")
60         for r in result:
61             successful += 1 if r else 0
62     print(
63         "successfully sent {}% ({} / {})".format(
64             successful / (size * count) * 100, successful, size * count
65         )
66     )
67
68
69 def slave(timeout=5):
70     """Stops listening after timeout with no response"""
71     # set address of TX node into an RX pipe. NOTE you MUST specify
72     # which pipe number to use for RX, we'll be using pipe 0
73     # pipe number options range [0,5]
74     # the pipe numbers used during a transition don't have to match
75     nrf.open_rx_pipe(0, address)
76     nrf.listen = True # put radio into RX mode and power up
77
78     count = 0
79     now = time.monotonic() # start timer
80     while time.monotonic() < now + timeout:
81         if nrf.any():
82             count += 1
83             # retrieve the received packet's payload
84             rx = nrf.recv() # clears flags & empties RX FIFO
85             print("Received: {} - {}".format(repr(rx), count))
86             now = time.monotonic()
87

```

(continues on next page)

(continued from previous page)

```
88     # recommended behavior is to keep in TX mode while idle
89     nrf.listen = False # put the nRF24L01 is in TX mode
90
91
92 print(
93     """
94     nRF24L01 Stream test\n
95     Run slave() on receiver\n
96     Run master() on transmitter"""
97 )
```



---

Fake BLE Example

---

This is a test to show how to use the nRF24L01 as a BLE advertising beacon.

Listing 1: examples/nrf24l01\_fake\_ble\_test.py

```
1  """
2  This example of using the nRF24L01 as a 'fake' Bluetooth Beacon
3
4      .. warning:: ATSAM21 M0-based boards have memory allocation
5                  error when loading fake_ble.mpy
6  """
7  import time
8  import board
9  import digitalio as dio
10 from circuitpython_nrf24l01.fake_ble import (
11     chunk,
12     FakeBLE,
13     UrlServiceData,
14     BatteryServiceData,
15     TemperatureServiceData,
16 )
17
18 # change these (digital output) pins accordingly
19 ce = dio.DigitalInOut(board.D4)
20 csn = dio.DigitalInOut(board.D5)
21
22 # using board.SPI() automatically selects the MCU's
23 # available SPI pins, board.SCK, board.MOSI, board.MISO
24 spi = board.SPI() # init spi bus object
25
26 # initialize the nRF24L01 on the spi bus object as a BLE compliant radio
27 nrf = FakeBLE(spi, csn, ce)
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
```

(continues on next page)

(continued from previous page)

```

31 # nrf.name = b"foobar"
32
33 # if broadcasting to an Android, set the to_iphone attribute to False
34 # if broadcasting to an iPhone, set the to_iphone attribute to True
35 nrf.to_iphone = True # default value is False
36
37 # you can optionally set the arbitrary MAC address to be used as the
38 # BLE device's MAC address. Otherwise this is randomly generated upon
39 # instantiation of the FakeBLE object.
40 # nrf.mac = b"\x19\x12\x14\x26\x09\xE0"
41
42 # set the Power Amplifier level to -12 dBm since this test example is
43 # usually run with nRF24L01 transceiver in close proximity to the
44 # BLE scanning application
45 nrf.pa_level = -12
46
47
48 def _prompt(count, iterator):
49     if (count - iterator) % 5 == 0 or (count - iterator) < 5:
50         if count - iterator - 1:
51             print(count - iterator, "advertisements left to go!")
52         else:
53             print(count - iterator, "advertisement left to go!")
54
55
56 # create an object for manipulating the battery level data
57 battery_service = BatteryServiceData()
58 # battery level data is 1 unsigned byte representing a percentage
59 battery_service.data = 85
60
61
62 def master(count=50):
63     """Sends out the device information twice a second."""
64     # using the "with" statement is highly recommended if the nRF24L01 is
65     # to be used for more than a BLE configuration
66     with nrf as ble:
67         ble.name = b"nRF24L01"
68         # include the radio's pa_level attribute in the payload
69         ble.show_pa_level = True
70         print(
71             "available bytes in next payload:",
72             ble.available(chunk(battery_service.buffer))
73         ) # using chunk() gives an accurate estimate of available bytes
74         for i in range(count): # advertise data this many times
75             if ble.available(chunk(battery_service.buffer)) >= 0:
76                 _prompt(count, i) # something to show that it isn't frozen
77                 # broadcast the device name, MAC address, &
78                 # battery charge info; 0x16 means service data
79                 ble.advertise(battery_service.buffer, data_type=0x16)
80                 # channel hopping is recommended per BLE specs
81                 ble.hop_channel()
82                 # alternate advertisements to target all devices
83                 ble.to_iphone = not ble.to_iphone
84                 time.sleep(0.5) # wait till next broadcast
85             # nrf.show_pa_level & nrf.name both are set to false when
86             # exiting a with statement block
87

```

(continues on next page)



(continued from previous page)

```

88
89 # create an object for manipulating temperature measurements
90 temperature_service = TemperatureServiceData()
91 # temperature's float data has up to 2 decimal places of percision
92 temperature_service.data = 42.0
93
94
95 def send_temp(count=50):
96     """Sends out a fake temperature twice a second."""
97     with nrf as ble:
98         ble.name = b"nRF24L01"
99         print(
100             "available bytes in next payload:",
101             ble.available(chunk(temperature_service.buffer))
102         )
103         for i in range(count):
104             if ble.available(chunk(temperature_service.buffer)) >= 0:
105                 _prompt(count, i)
106                 # broadcast a temperature measurement; 0x16 means service data
107                 ble.advertise(temperature_service.buffer, data_type=0x16)
108                 # channel hopping is recommended per BLE specs
109                 ble.hop_channel()
110                 ble.to_iphone = not ble.to_iphone
111                 time.sleep(0.2)
112
113
114 # use the Eddystone protocol from Google to broadcast a URL as
115 # service data. We'll need an object to manipulate that also
116 url_service = UrlServiceData()
117 # the data attribute converts a URL string into a simplified
118 # bytes object using byte codes defined by the Eddystone protocol.
119 url_service.data = "http://www.google.com"
120 # Eddystone protocol requires an estimated TX PA level at 1 meter
121 # lower this estimate since we lowered the actual `ble.pa_level`
122 url_service.pa_level_at_1_meter = -45 # defaults to -25 dBm
123
124 def send_url(count=50):
125     """Sends out a URL twice a second."""
126     with nrf as ble:
127         print(
128             "available bytes in next payload:",
129             ble.available(chunk(url_service.buffer))
130         )
131         # NOTE we did NOT set a device name in this with block
132         for i in range(count):
133             # URLs easily exceed the nRF24L01's max payload length
134             if ble.available(chunk(url_service.buffer)) >= 0:
135                 _prompt(count, i)
136                 ble.advertise(url_service.buffer, 0x16)
137                 ble.hop_channel()
138                 ble.to_iphone = not ble.to_iphone
139                 time.sleep(0.2)
140
141 print(
142     """\
143     nRF24L01 fake BLE beacon test.\n\
144     Run master() to broadcast the device name, pa_level, & battery charge\n\

```

(continues on next page)

(continued from previous page)

```
145   Run send_temp() to broadcast the device name & a temperature\n\  
146   Run send_url() to broadcast a custom URL link"""  
147 )
```

# CHAPTER 10

## Context Example

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

Listing 1: 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  .. warning:: This script is not compatible with the rf24_lite module
7  """
8  import board
9  import digitalio as dio
10 from circuitpython_nrf24l01.rf24 import RF24
11 from circuitpython_nrf24l01.fake_ble import FakeBLE
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 # initialize the nRF24L01 objects on the spi bus object
22 # the first object will have all the features enabled
23 nrf = RF24(spi, csn, ce)
24 # enable the option to use custom ACK payloads
25 nrf.ack = True
26 # set the static payload length to 8 bytes
27 nrf.payload_length = 8
28 # RF power amplifier is set to -18 dbm
29 nrf.pa_level = -18
```

(continues on next page)

(continued from previous page)

```

30
31 # the second object has most features disabled/alterd
32 ble = FakeBLE(spi, csn, ce)
33 # the IRQ pin is configured to only go active on "data fail"
34 # NOTE BLE operations prevent the IRQ pin going active on "data fail" events
35 ble.interrupt_config(data_rcv=False, data_sent=False)
36 # using a channel 2
37 ble.channel = 2
38 # RF power amplifier is set to -12 dbm
39 ble.pa_level = -12
40
41 print("\nsettings configured by the nrf object")
42 with nrf:
43     # only the first character gets written because it is on a pipe_number > 1
44     nrf.open_rx_pipe(5, b"1Node") # NOTE we do this inside the "with" block
45
46     # display current settings of the nrf object
47     nrf.what_happened(True) # True dumps pipe info
48
49 print("\nsettings configured by the ble object")
50 with ble as nerf: # the "as nerf" part is optional
51     nerf.what_happened(1)
52
53 # if you examine the outputs from what_happened() you'll see:
54 #   pipe 5 is opened using the nrf object, but closed using the ble object.
55 #   pipe 0 is closed using the nrf object, but opened using the ble object.
56 #   also notice the different addresses bound to the RX pipes
57 # this is because the "with" statements load the existing settings
58 # for the RF24 object specified after the word "with".
59
60 # NOTE it is not advised to manipulate seperate RF24 objects outside of the
61 # "with" block; you will encounter bugs about configurations when doing so.
62 # Be sure to use 1 "with" block per RF24 object when instantiating multiple
63 # RF24 objects in your program.
64 # NOTE exiting a "with" block will always power down the nRF24L01
65 # NOTE upon instantiation, this library closes all RX pipes &
66 # extracts the TX/RX addresses from the nRF24L01 registers

```

---

## 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". If you changed the `role` variable in the TMRh20 sketch, you will have to adjust the addresses assigned to the pipes in this script.

Listing 1: 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 # if running this on a ATSAMD21 M0 based board
12 # from circuitpython_nrf24l01.rf24_lite import RF24
13 from circuitpython_nrf24l01.rf24 import RF24
14
15 # addresses needs to be in a buffer protocol object (bytearray)
16 address = [b"1Node", b"2Node"]
17
18 # change these (digital output) pins accordingly
19 ce = dio.DigitalInOut(board.D4)
20 csn = dio.DigitalInOut(board.D5)
21
22 # using board.SPI() automatically selects the MCU's
23 # available SPI pins, board.SCK, board.MOSI, board.MISO
24 spi = board.SPI() # init spi bus object
25
26 # initialize the nRF24L01 on the spi bus object
27 nrf = RF24(spi, csn, ce)
```

(continues on next page)

(continued from previous page)

```

28 nrf.dynamic_payloads = False # the default in the TMRh20 arduino library
29
30 # set the Power Amplifier level to -12 dBm since this test example is
31 # usually run with nRF24L01 transceivers in close proximity
32 nrf.pa_level = -12
33
34 # set address of TX node into a RX pipe
35 nrf.open_rx_pipe(1, address[1])
36 # set address of RX node into a TX pipe
37 nrf.open_tx_pipe(address[0])
38
39 # pylint: disable=too-few-public-methods
40 class DataStruct:
41     """A data structure to hold transmitted values as the
42     'HandlingData' part of the TMRh20 library example"""
43     time = 0 # in milliseconds (used as start of timer)
44     value = 1.22 # incremented by 0.01 with every transmission
45     # pylint: enable=too-few-public-methods
46
47 myData = DataStruct()
48
49
50 def master(count=5): # count = 5 will only transmit 5 packets
51     """Transmits an arbitrary unsigned long value every second"""
52     while count:
53         nrf.listen = False # ensures the nRF24L01 is in TX mode
54         print("Now Sending")
55         myData.time = int(time.monotonic() * 1000) # start timer
56         # use struct.pack to packetize your data into a usable payload
57         # '<' means little endian byte order.
58         # 'L' means a single 4 byte unsigned long value.
59         # 'f' means a single 4 byte float value.
60         buffer = struct.pack("<Lf", myData.time, myData.value)
61         result = nrf.send(buffer)
62         if not result:
63             print("send() failed or timed out")
64         else:
65             nrf.listen = True # get radio ready to receive a response
66             timeout = True # used to determine if response timed out
67             while time.monotonic() * 1000 - myData.time < 200:
68                 # the arbitrary 200 ms timeout value is also used in the
69                 # TMRh20 library's GettingStarted_HandlingData sketch
70                 if nrf.any():
71                     end_timer = time.monotonic() * 1000 # end timer
72                     rx = nrf.recv()
73                     rx = struct.unpack("<Lf", rx[:8])
74                     myData.value = rx[1] # save the new float value
75                     timeout = False # skips timeout prompt
76                     # print total time to send and receive data
77                     print(
78                         "Sent {} Got Response: {}".format(
79                             struct.unpack("<Lf", buffer),
80                             rx
81                         )
82                     )
83                     print("Round-trip delay:", end_timer - myData.time, "ms")
84                     break

```

(continues on next page)

(continued from previous page)

```

85         if timeout:
86             print("failed to get a response; timed out")
87         count -= 1
88         time.sleep(1)
89
90
91 def slave(count=3):
92     """Polls the radio and prints the received value. This method expires
93     after 6 seconds of no received transmission"""
94     myData.time = time.monotonic() * 1000 # in milliseconds
95     while count and (time.monotonic() * 1000 - myData.time) < 6000:
96         nrf.listen = True # put radio into RX mode and power up
97         if nrf.any():
98             # retrieve the received packet's payload
99             buffer = nrf.recv() # clears flags & empties RX FIFO
100             # increment floating value as part of the "HandlingData" test
101             myData.value = struct.unpack("<f", buffer[4:8])[0] + 0.01
102             nrf.listen = False # ensures the nRF24L01 is in TX mode
103             myData.time = time.monotonic() * 1000
104             # echo buffer[:4] appended with incremented float
105             result = nrf.send(buffer[:4] + struct.pack("<f", myData.value))
106             end_timer = time.monotonic() * 1000 # in milliseconds
107             # expecting an unsigned long & a float, thus the
108             # string format "<Lf"; buffer[:8] ignores the padded 0s
109             rx = struct.unpack("<Lf", buffer[:8])
110             # print the unsigned long and float data sent in the response
111             print("Responding: {}, {}".format(rx[0], rx[1] + 0.01))
112             if not result:
113                 print("response failed or timed out")
114             else:
115                 # print timer results on transmission success
116                 print(
117                     "successful response took {} ms".format(
118                         end_timer - myData.time
119                     )
120                 )
121             # this will listen indefinitely till counter == 0
122             count -= 1
123     # recommended behavior is to keep in TX mode when in idle
124     nrf.listen = False # put the nRF24L01 in TX mode + Standby-I power state
125
126
127 print(
128     """
129     nRF24L01 communicating with an Arduino running the
130     TMRh20 library's "GettingStarted_HandlingData.ino" example.
131     Run slave() on receiver
132     Run master() on transmitter"""
133 )

```





## CHAPTER 12

### 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 1: examples/nrf24l01\_interrupt\_test.py

```
1  """
2  Simple example of detecting (and verifying) the IRQ (interrupt) pin on the
3  nRF24L01
4      .. note:: this script uses the non-blocking `write()` function because
5                 the function `send()` clears the IRQ flags upon returning
6  """
7  import time
8  import board
9  import digitalio as dio
10 # if running this on a ATSAM21 M0 based board
11 # from circuitpython_nrf24l01.rf24_lite import RF24
12 from circuitpython_nrf24l01.rf24 import RF24
13
14 # address needs to be in a buffer protocol object (bytearray is preferred)
15 address = b"1Node"
16
17 # select your digital input pin that's connected to the IRQ pin on the nRF4L01
18 irq_pin = dio.DigitalInOut(board.D12)
19 irq_pin.switch_to_input() # make sure its an input object
20 # change these (digital output) pins accordingly
21 ce = dio.DigitalInOut(board.D4)
22 csn = dio.DigitalInOut(board.D5)
23
24 # using board.SPI() automatically selects the MCU's
25 # available SPI pins, board.SCK, board.MOSI, board.MISO
26 spi = board.SPI() # init spi bus object
27
```

(continues on next page)

(continued from previous page)

```

28 # we'll be using the dynamic payload size feature (enabled by default)
29 # initialize the nRF24L01 on the spi bus object
30 nrf = RF24(spi, csn, ce)
31
32 # this example uses the ACK payload to trigger the IRQ pin active for
33 # the "on data received" event
34 nrf.ack = True # enable ACK payloads
35
36 # set the Power Amplifier level to -12 dBm since this test example is
37 # usually run with nRF24L01 transceivers in close proximity
38 nrf.pa_level = -12
39
40
41 def _ping_and_prompt():
42     """transmit 1 payload, wait till irq_pin goes active, print IRQ status
43     flags."""
44     ce.value = 1 # tell the nRF24L01 to prepare sending a single packet
45     time.sleep(0.00001) # mandatory 10 microsecond pulse starts transmission
46     ce.value = 0 # end 10 us pulse; use only 1 buffer from TX FIFO
47     while irq_pin.value: # IRQ pin is active when LOW
48         pass
49     print("IRQ pin went active LOW.")
50     nrf.update() # update irq_d? status flags
51     print(
52         "\tirq_ds: {}, irq_dr: {}, irq_df: {}".format(
53             nrf.irq_ds, nrf.irq_dr, nrf.irq_df
54         )
55     )
56
57 def master():
58     """Transmits 3 times: successfully receive ACK payload first, successfully
59     transmit on second, and intentionally fail transmit on the third"""
60     # set address of RX node into a TX pipe
61     nrf.open_tx_pipe(address)
62     # ensures the nRF24L01 is in TX mode
63     nrf.listen = False
64     # NOTE nrf.power is automatically set to True on first call to nrf.write()
65     # NOTE nrf.write() internally calls nrf.clear_status_flags() first
66
67     # load 2 buffers into the TX FIFO; write_only=True leaves CE pin LOW
68     nrf.write(b"Ping ", write_only=True)
69     nrf.write(b"Pong ", write_only=True)
70
71     # on data ready test
72     print("\nConfiguring IRQ pin to only ignore 'on data sent' event")
73     nrf.interrupt_config(data_sent=False)
74     print("    Pinging slave node for an ACK payload...", end=" ")
75     _ping_and_prompt() # CE pin is managed by this function
76     if nrf.irq_dr:
77         print("\t'on data ready' event test successful")
78     else:
79         print("\t'on data ready' event test unsuccessful")
80
81     # on data sent test
82     print("\nConfiguring IRQ pin to only ignore 'on data ready' event")
83     nrf.interrupt_config(data_rcv=False)
84     print("    Pinging slave node again... ", end=" ")

```

(continues on next page)

(continued from previous page)

```

85  _ping_and_prompt() # CE pin is managed by this function
86  if nrf.irq_ds:
87      print("\t'on data sent' event test successful")
88  else:
89      print("\t'on data sent' event test unsuccessful")
90
91  # trigger slave node to exit by filling the slave node's RX FIFO
92  print("\nSending one extra payload to fill RX FIFO on slave node.")
93  if nrf.send(b"Radio", send_only=True):
94      # when send_only parameter is True, send() ignores RX FIFO usage
95      print("Slave node should not be listening anymore.")
96  else:
97      print("Slave node was unresponsive.")
98
99  # on data fail test
100 print("\nConfiguring IRQ pin to go active for all events.")
101 nrf.interrupt_config()
102 print("    Sending a ping to inactive slave node...", end=" ")
103 nrf.flush_tx() # just in case any previous tests failed
104 nrf.write(b"Dummy", write_only=True) # CE pin is left LOW
105 _ping_and_prompt() # CE pin is managed by this function
106 if nrf.irq_df:
107     print("\t'on data failed' event test successful")
108 else:
109     print("\t'on data failed' event test unsuccessful")
110 nrf.flush_tx() # flush artifact payload in TX FIFO from last test
111 # all 3 ACK payloads received were 4 bytes each, and RX FIFO is full
112 # so, fetching 12 bytes from the RX FIFO also flushes RX FIFO
113 print("\nComplete RX FIFO:", nrf.recv(12))
114
115
116 def slave(timeout=6): # will listen for 6 seconds before timing out
117     """Only listen for 3 payload from the master node"""
118     # setup radio to receive pings, fill TX FIFO with ACK payloads
119     nrf.open_rx_pipe(0, address)
120     nrf.load_ack(b"Yak ", 0)
121     nrf.load_ack(b"Back", 0)
122     nrf.load_ack(b" ACK", 0)
123     nrf.listen = True # start listening & clear irq_dr flag
124     start_timer = time.monotonic() # start timer now
125     while not nrf.fifo(0, 0) and time.monotonic() - start_timer < timeout:
126         # if RX FIFO is not full and timeout is not reached, then keep going
127         pass
128     nrf.listen = False # put nRF24L01 in Standby-I mode when idling
129     if not nrf.fifo(False, True): # if RX FIFO is not empty
130         # all 3 payloads received were 5 bytes each, and RX FIFO is full
131         # so, fetching 15 bytes from the RX FIFO also flushes RX FIFO
132         print("Complete RX FIFO:", nrf.recv(15))
133     nrf.flush_tx() # discard any pending ACK payloads
134
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"""

```

(continues on next page)

(continued from previous page)

142

)

---

## 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 immediately 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.
- 

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:

- 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 (more on this when we officially support "Multiciever" mode).

Finally, 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 transceivers' 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 receiving 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 14

---

## 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 ATSAM21 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`.
- `rpdc`, `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).



---

## Testing nRF24L01+PA+LNA module

---

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

### 15.1 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 solder joints for the antenna mount). Then I wired up a PA/LNA module with a 3V regulator (L4931 with a 2.2  $\mu$ F 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):

### 15.2 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).

## 15.3 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.

## 16.1 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*.

## 16.2 `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.

---

## 16.3 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 *ValueError* exception is thrown.

## 16.4 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.

If *dynamic\_payloads* attribute is disabled for the specified data pipe, then the *payload\_length* attribute is used to define the expected length of the static 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, 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.

---

## 16.5 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 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).

## 16.6 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.

## 16.7 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 `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.

## 16.8 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). This functionality 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 `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). See also notes on `resend()` as using this parameter carries the same implications documented there. This parameter has no effect if the `ask_no_ack` parameter is set to `True` or if `arc` is disabled.
- **send\_only** (*bool*) – This parameter only applies when the `ack` attribute is set to `True`. Pass this parameter as `True` if you want to handle fetching the ACK payload (from the RX FIFO) separately from the sending transmission that recieved the ACK payload. Many other libraries' behave as though this parameter is `True` (e.g. The popular TMRh20 Arduino RF24 library). Use `recv()` to get the ACK payload (if there is any) from the RX FIFO. This parameter defaults to `False`. 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 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.





## 17.1 what\_happened()

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

- 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
- 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 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.

Default is `False` and skips this extra information.

## 17.2 `is_plus_variant`

### `RF24.is_plus_variant`

A `bool` attribute to describe if the nRF24L01 is a plus variant or not (read-only). Upon instantiation, this class determines if the nRF24L01 is a plus variant or not.

## 17.3 `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 fresh data is received on the specified pipe. See `read_ack()` 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 `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.

---

## 17.4 read\_ack()

nRF24L01 . `read_ack()`

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

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

---

**Note:** See also the `ack`, `dynamic_payloads`, and `auto_ack` attributes as they must be enabled to use custom ACK payloads.

---

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

## 17.5 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 `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.

## 17.6 irq\_df

### RF24.irq\_df

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

#### Returns

- `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()` 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.

## 17.7 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 `dataSent` 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.

## 17.8 clear\_status\_flags()

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.

---

## 17.9 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 transmissions. This driver class doesn't power down the nRF24L01 after RX/TX transmissions are complete (avoiding the required power up/down 150 µs wait time), that preference is left to the application.
- `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, which Standby mode depends on the state of the CE pin. 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).

---

## 17.10 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.

#### 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.

## 17.11 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.

## 17.12 resend()

### RF24.resend(send\_only=False)

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

All returned data 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 you want to handle fetching the ACK payload (from the RX FIFO) separately from the sending transmission that received the ACK payload. Many other libraries' behave as though this parameter is `True` (e.g. The popular TMRh20 Arduino RF24 library). Use `recv()` to get the ACK payload (if there is any) from the RX FIFO. This parameter defaults to `False`. 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.

---

## 17.13 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](#)

### 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  $\mu$ s pulse on the CE pin is achieved. If the `write_only` parameter is `False`, then 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. According to the Specification sheet, 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 [`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 sentinel.

---

## 17.14 flush\_rx()

`nRF24.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.

---

## 17.15 flush\_tx()

`nRF24.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).

---

## 17.16 fifo()

`nRF24.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 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.
- **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]:[`True/False`] FIFO buffer [empty/full]:[`True/False`]?”



- If the `check_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

## 17.17 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.

#### 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.

## 17.18 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.

## 17.19 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. Defaults to -1. A valid index ranges [0,5] for RX addresses or any negative `int` for the TX address. Otherwise an `IndexError` is thrown.

## 17.20 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 (`listen = 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.

---

## 17.21 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 separate 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 dummy payload 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 to settings are needed nor applied.

---

## 17.22 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).

---

### 18.1 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.

### 18.2 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 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 set accordingly for 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.

---

## 18.3 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 in RX mode.

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 to control this attribute 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 a 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.

---

## 18.4 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 automatically by the nRF24L01 if the `auto_ack` 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.

---

## 18.5 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.

## 18.6 ard

### RF24.ard

This `int` attribute specifies the nRF24L01's delay (in microseconds) 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 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.

---

## 18.7 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.

---

## 18.8 interrupt\_config()

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

Sets the configuration of the nRF24L01's IRQ (interrupt) pin.

The digital 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. 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 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. 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 `data_recv` 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` (may require `update()` to be called), `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
- 

## 18.9 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 models of the nRF24L01 product line. Trying to set the data rate to 250 kbps when `is_plus_variant` is `True` will throw a `NotImplementedError`.

## 18.10 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).

## 18.11 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).

---

## 18.12 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 only applies to the nRF24L01 (non-plus variant). This includes boards with the RFX24C01-based PA/LNA muxing IC attached to an SMA-type detachable antenna.

---

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

## 18.13 is\_lna\_enabled

### RF24.is\_lna\_enabled

A read-only `bool` attribute about the LNA (Low Noise Amplifier) gain feature used in the nRF24L01-PA/LNA modules. 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. Use `is_plus_variant` to see if it can toggle the Low Noise Amplifier feature.





# CHAPTER 19

---

## 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:  
$$32 \text{ (nRF24L01 maximum)} - 6 \text{ (MAC address)} - 5 \text{ (required flags)} - 3 \text{ (CRC checksum)} = 18$$
  
Use the helper function `available()` to determine 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 transmitting 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.



## 20.1 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.

## 20.2 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.

## 20.3 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*.

---

## 20.4 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 specifications 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 specifications require `0x555555` (default value).

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

## 20.5 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 21

---

## 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*.

## 21.1 to\_iphone

### `FakeBLE.to_iphone`

A *bool* to specify if advertisements should be compatible with the iPhone. A value of `False` should still be compatible with other Apple devices. Testing with this attribute as `False` showed compatibility with a Mac desktop.

## 21.2 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.

## 21.3 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.

---

## 21.4 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.

---

## 21.5 hop\_channel()

### `FakeBLE.hop_channel()`

Trigger an automatic change of BLE compliant channels.

## 21.6 whiten()

### `FakeBLE.whiten(data)`

Whitening the BLE packet data ensures there's no long repetition 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 algorithm 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.

## 21.7 available()

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

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

This is determined 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.

## 21.8 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 and are packed into 1 payload for a single transmission. 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"

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

(continues on next page)

(continued from previous page)

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

## 21.9 Available RF24 functionality

### 21.9.1 `pa_level`

`FakeBLE.pa_level`

See `pa_level` for more details.

### 21.9.2 `channel`

`FakeBLE.channel`

The only allowed channels are those contained in the `BLE_FREQ` tuple.

### 21.9.3 `payload_length`

`FakeBLE.payload_length`

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

### 21.9.4 `power`

`FakeBLE.power`

See `power` for more details.

### 21.9.5 `is_lna_enabled`

`FakeBLE.is_lna_enabled`

See `is_lna_enabled` for more details.

### 21.9.6 `is_plus_variant`

`FakeBLE.is_plus_variant`

See `is_plus_variant` for more details.

### 21.9.7 `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.



### 21.9.8 `irq_ds`

`FakeBLE.irq_ds`

See *irq\_ds* for more details.

### 21.9.9 `irq_dr`

`FakeBLE.irq_dr`

See *irq\_dr* for more details.

### 21.9.10 `clear_status_flags()`

`FakeBLE.clear_status_flags()`

See *clear\_status\_flags()* for more details.

### 21.9.11 `update()`

`FakeBLE.update()`

See *update()* for more details.

### 21.9.12 `what_happened()`

`FakeBLE.what_happened()`

See *what\_happened()* for more details.



---

## Service related classes

---

### 22.1 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.

### 22.2 derivative children

**class** `circuitpython_nrf24l01.fake_ble.TemperatureServiceData`

Bases: `circuitpython_nrf24l01.fake_ble.ServiceData`

This derivative 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 derivative 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 derivative 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 23

---

### Indices and tables

---

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



## Symbols

`__len__()` (*circuitpython\_nrf24l01.fake\_ble.ServiceData* method), 77

## A

`ack()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 63

`address()` (*circuitpython\_nrf24l01.rf24.RF24* method), 59

`address_length()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 59

`advertise()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* method), 73

`any()` (*circuitpython\_nrf24l01.rf24.RF24* method), 47

`arc()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 62

`ard()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 63

`auto_ack()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 62

`available()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* method), 73

## B

`BatteryServiceData` (class in *circuitpython\_nrf24l01.fake\_ble*), 77

`BLE_FREQ` (in module *circuitpython\_nrf24l01.fake\_ble*), 70

`buffer()` (*circuitpython\_nrf24l01.fake\_ble.ServiceData* attribute), 77

## C

`channel()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* attribute), 74

`channel()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 64

`chunk()` (in module *circuitpython\_nrf24l01.fake\_ble*), 69

`clear_status_flags()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* method), 75

`clear_status_flags()` (*circuitpython\_nrf24l01.rf24.RF24* method), 55

`close_rx_pipe()` (*circuitpython\_nrf24l01.rf24.RF24* method), 46

`crc()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 65

`crc24_ble()` (in module *circuitpython\_nrf24l01.fake\_ble*), 70

`CSN_DELAY` (in module *circuitpython\_nrf24l01.rf24*), 61

## D

`data()` (*circuitpython\_nrf24l01.fake\_ble.ServiceData* attribute), 77

`data_rate()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 64

`dynamic_payloads()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 61

## F

`FakeBLE` (class in *circuitpython\_nrf24l01.fake\_ble*), 71

`fifo()` (*circuitpython\_nrf24l01.rf24.RF24* method), 58

`flush_rx()` (*circuitpython\_nrf24l01.rf24.RF24* method), 58

`flush_tx()` (*circuitpython\_nrf24l01.rf24.RF24* method), 58

## H

`hop_channel()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* method), 72

## I

`interrupt_config()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* method), 74

`interrupt_config()` (*circuitpython\_nrf24l01.rf24.RF24* method), 63

`irq_df()` (*circuitpython\_nrf24l01.rf24.RF24* attribute), 54

`irq_dr()` (*circuitpython\_nrf24l01.fake\_ble.FakeBLE* attribute), 75

`irq_dr` (`circuitpython_nrf24l01.rf24.RF24` attribute), 54  
`irq_ds` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 75  
`irq_ds` (`circuitpython_nrf24l01.rf24.RF24` attribute), 54  
`is_lna_enabled` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 74  
`is_lna_enabled` (`circuitpython_nrf24l01.rf24.RF24` attribute), 65  
`is_plus_variant` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 74  
`is_plus_variant` (`circuitpython_nrf24l01.rf24.RF24` attribute), 52

## L

`listen` (`circuitpython_nrf24l01.rf24.RF24` attribute), 46  
`load_ack()` (`circuitpython_nrf24l01.rf24.RF24` method), 52

## M

`mac` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 72

## N

`name` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 72

## O

`open_rx_pipe()` (`circuitpython_nrf24l01.rf24.RF24` method), 46  
`open_tx_pipe()` (`circuitpython_nrf24l01.rf24.RF24` method), 45

## P

`pa_level` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 74  
`pa_level` (`circuitpython_nrf24l01.rf24.RF24` attribute), 65  
`pa_level_at_1_meter` (`circuitpython_nrf24l01.fake_ble.UrlServiceData` attribute), 78  
`payload_length` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 74  
`payload_length` (`circuitpython_nrf24l01.rf24.RF24` attribute), 62  
`pipe` (`circuitpython_nrf24l01.rf24.RF24` attribute), 59  
`power` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 74

`power` (`circuitpython_nrf24l01.rf24.RF24` attribute), 55

## R

`read_ack()` (`circuitpython_nrf24l01.rf24.RF24` method), 53  
`recv()` (`circuitpython_nrf24l01.rf24.RF24` method), 47  
`resend()` (`circuitpython_nrf24l01.rf24.RF24` method), 56  
`reverse_bits()` (in module `circuitpython_nrf24l01.fake_ble`), 69  
`RF24` (class in `circuitpython_nrf24l01.rf24`), 45  
`rpd` (`circuitpython_nrf24l01.rf24.RF24` attribute), 59

## S

`send()` (`circuitpython_nrf24l01.rf24.RF24` method), 48  
`ServiceData` (class in `circuitpython_nrf24l01.fake_ble`), 77  
`show_pa_level` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 72  
`start_carrier_wave()` (`circuitpython_nrf24l01.rf24.RF24` method), 60  
`stop_carrier_wave()` (`circuitpython_nrf24l01.rf24.RF24` method), 60  
`swap_bits()` (in module `circuitpython_nrf24l01.fake_ble`), 69

## T

`TemperatureServiceData` (class in `circuitpython_nrf24l01.fake_ble`), 77  
`to_iphone` (`circuitpython_nrf24l01.fake_ble.FakeBLE` attribute), 71  
`tx_full` (`circuitpython_nrf24l01.rf24.RF24` attribute), 56

## U

`update()` (`circuitpython_nrf24l01.fake_ble.FakeBLE` method), 75  
`update()` (`circuitpython_nrf24l01.rf24.RF24` method), 56  
`UrlServiceData` (class in `circuitpython_nrf24l01.fake_ble`), 78  
`uuid` (`circuitpython_nrf24l01.fake_ble.ServiceData` attribute), 77

## W

`what_happened()` (`circuitpython_nrf24l01.fake_ble.FakeBLE` method), 75  
`what_happened()` (`circuitpython_nrf24l01.rf24.RF24` method), 51  
`whiten()` (`circuitpython_nrf24l01.fake_ble.FakeBLE` method), 72



```
write() (circuitpython_nrf24l01.rf24.RF24 method),  
56
```