nRF24L01 Library Documentation

Release 2.1.0

Brendan Doherty

Jun 16, 2022

CONTENTS

CHAPTER

ONE

NRF24L01 FEATURES

1.1 Simple test

Changed in version 2.0.0:

- uses 2 addresses on pipes 1 & 0 to demonstrate proper addressing convention.
- transmits an incrementing `float` instead of an `int`

Ensure your device works with this simple test.

Listing 1: examples/nrf24l01_simple_test.py

```
import time
import struct
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


```

(continues on next page)

(continued from previous page)

```
# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

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

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

# to use different addresses on a pair of radios, we need a variable to
# uniquely identify which address this radio will use to transmit
# 0 uses address[0] to transmit, 1 uses address[1] to transmit
radio_number = bool(
    int(input("Which radio is this? Enter '0' or '1'. Defaults to '0' ") or 0)
)

# set TX address of RX node into the TX pipe
nrf.open_tx_pipe(address[radio_number])  # always uses pipe 0

# set RX address of TX node into an RX pipe
nrf.open_rx_pipe(1, address[not radio_number])  # using pipe 1

# using the python keyword global is bad practice. Instead we'll use a 1 item
# list to store our float number for the payloads sent
payload = [0.0]

# uncomment the following 3 lines for compatibility with TMRh20 library
# nrf.allow_ask_no_ack = False
# nrf.dynamic_payloads = False
# nrf.payload_length = 4


def master(count=5):  # count = 5 will only transmit 5 packets
    """Transmits an incrementing integer every second"""
    nrf.listen = False  # ensures the nRF24L01 is in TX mode

    while count:
        # use struct.pack to structure your data
        # into a usable payload
        buffer = struct.pack("<f", payload[0])
        # "<f" means a single little endian (4 byte) float value.
        start_timer = time.monotonic_ns()  # start timer
        result = nrf.send(buffer)
        end_timer = time.monotonic_ns()  # end timer
        if not result:
            print("send() failed or timed out")
        else:
```

(continues on next page)

(continued from previous page)

```
            print(
                "Transmission successful! Time to Transmit:",
                "{} us. Sent: {}".format((end_timer - start_timer) / 1000, payload[0])
            )
            payload[0] += 0.01
        time.sleep(1)
        count -= 1


def slave(timeout=6):
    """Polls the radio and prints the received value. This method expires
    after 6 seconds of no received transmission"""
    nrf.listen = True  # put radio into RX mode and power up

    start = time.monotonic()
    while (time.monotonic() - start) < timeout:
        if nrf.available():
            # grab information about the received payload
            payload_size, pipe_number = (nrf.any(), nrf.pipe)
            # fetch 1 payload from RX FIFO
            buffer = nrf.read()  # also clears nrf.irq_dr status flag
            # expecting a little endian float, thus the format string "<f"
            # buffer[:4] truncates padded 0s if dynamic payloads are disabled
            payload[0] = struct.unpack("<f", buffer[:4])[0]
            # print details about the received packet
            print(
                "Received {} bytes on pipe {}: {}".format(
                    payload_size, pipe_number, payload[0]
                )
            )
            start = time.monotonic()

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


```

1.2 ACK Payloads Example

Changed in version 2.0.0:

- uses 2 addresses on pipes 1 & 0 to demonstrate proper addressing convention.
- changed payloads to show use of c-strings' NULL terminating character.

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

See also:

More details are found in the documentation on *ack* and *load_ack()*.

Listing 2: examples/nrf24l01_ack_payload_test.py

```
import time
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

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

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

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

# addresses needs to be in a buffer protocol object (bytearray)
```

(continues on next page)

(continued from previous page)

```
address = [b"1Node", b"2Node"]

# to use different addresses on a pair of radios, we need a variable to
# uniquely identify which address this radio will use to transmit
# 0 uses address[0] to transmit, 1 uses address[1] to transmit
radio_number = bool(
    int(input("Which radio is this? Enter '0' or '1'. Defaults to '0' ") or 0)
)

# set TX address of RX node into the TX pipe
nrf.open_tx_pipe(address[radio_number])  # always uses pipe 0

# set RX address of TX node into an RX pipe
nrf.open_rx_pipe(1, address[not radio_number])  # using pipe 1

# using the python keyword global is bad practice. Instead we'll use a 1 item
# list to store our integer number for the payloads' counter
counter = [0]


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

    while count:
        # construct a payload to send
        # add b"\0" as a c-string NULL terminating char
        buffer = b"Hello \0" + bytes([counter[0]])
        start_timer = time.monotonic_ns()  # start timer
        result = nrf.send(buffer)  # save the response (ACK payload)
        end_timer = time.monotonic_ns()  # stop timer
        if result:
            # print the received ACK that was automatically
            # fetched and saved to "result" via send()
            # print timer results upon transmission success
            print(
                "Transmission successful! Time to transmit:",
                int((end_timer - start_timer) / 1000),
                "us. Sent: {}{}".format(buffer[:6].decode("utf-8"), counter[0]),
                end=" ",
            )
            if isinstance(result, bool):
                print("Received an empty ACK packet")
            else:
                # result[:6] truncates c-string NULL termiating char
                # received counter is a unsigned byte, thus result[7:8][0]
                print(
                    "Received: {}{}".format(result[:6].decode("utf-8"), result[7:8][0])
                )
            counter[0] += 1  # increment payload counter
        elif not result:
            print("send() failed or timed out")
```

(continues on next page)

(continued from previous page)

```
        time.sleep(1)  # let the RX node prepare a new ACK payload
        count -= 1


def slave(timeout=6):
    """Prints the received value and sends an ACK payload"""
    nrf.listen = True  # put radio into RX mode, power it up

    # setup the first transmission's ACK payload
    # add b"\0" as a c-string NULL terminating char
    buffer = b"World \0" + bytes([counter[0]])
    # we must set the ACK payload data and corresponding
    # pipe number [0, 5]. We'll be acknowledging pipe 1
    nrf.load_ack(buffer, 1)  # load ACK for first response

    start = time.monotonic()  # start timer
    while (time.monotonic() - start) < timeout:
        if nrf.available():
            # grab information about the received payload
            length, pipe_number = (nrf.any(), nrf.pipe)
            # retreive the received packet's payload
            received = nrf.read()
            # increment counter from received payload
            # received counter is a unsigned byte, thus result[7:8][0]
            counter[0] = received[7:8][0] + 1
            # the [:6] truncates the c-string NULL termiating char
            print(
                "Received {} bytes on pipe {}:".format(length, pipe_number),
                "{}{}".format(received[:6].decode("utf-8"), received[7:8][0]),
                "Sent: {}{}".format(buffer[:6].decode("utf-8"), buffer[7:8][0]),
            )
            start = time.monotonic()  # reset timer
            buffer = b"World \0" + bytes([counter[0]])  # build new ACK
            nrf.load_ack(buffer, 1)  # load ACK for next response

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


```

1.3 Multiceiver Example

New in version 1.2.2.

Changed in version 2.0.0: no longer uses ACK payloads for responding to node 1.

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

Note: This example follows the diagram illustrated in figure 12 of section 7.7 of the nRF24L01 Specifications Sheet

Please note that if `auto_ack` (on the base station) and `arc` (on the transmitting nodes) are disabled, then figure 10 of section 7.7 of the nRF24L01 Specifications Sheet would be a better illustration.

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

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

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

Listing 3: examples/nrf24l01_multiceiver_test.py

```
import time
import struct
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

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

(continues on next page)

(continued from previous page)

```
nrf.pa_level = -12

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

# uncomment the following 3 lines for compatibility with TMRh20 library
# nrf.allow_ask_no_ack = False
# nrf.dynamic_payloads = False
# nrf.payload_length = 8


def base(timeout=10):
    """Use the nRF24L01 as a base station for listening to all nodes"""
    # write the addresses to all pipes.
    for pipe_n, addr in enumerate(addresses):
        nrf.open_rx_pipe(pipe_n, addr)
    nrf.listen = True  # put base station into RX mode
    start_timer = time.monotonic()  # start timer
    while time.monotonic() - start_timer < timeout:
        while not nrf.fifo(False, True):  # keep RX FIFO empty for reception
            # show the pipe number that received the payload
            # NOTE read() clears the pipe number and payload length data
            print("Received", nrf.any(), "on pipe", nrf.pipe, end=" ")
            node_id, payload_id = struct.unpack("<ii", nrf.read())
            print("from node {}. PayloadID: {}".format(node_id, payload_id))
            start_timer = time.monotonic()  # reset timer with every payload
    nrf.listen = False


def node(node_number=0, count=6):
    """start transmitting to the base station.

    :param int node_number: the node's identifying index (from the
        the `addresses` list)
    :param int count: the number of times that the node will transmit
        to the base station.
    """
    nrf.listen = False
    # set the TX address to the address of the base station.
    nrf.open_tx_pipe(addresses[node_number])
    counter = 0
    # use the node_number to identify where the payload came from
    while counter < count:
        counter += 1
        # payloads will include the node_number and a payload ID character
```

(continues on next page)

(continued from previous page)

```
        payload = struct.pack("<ii", node_number, counter)
        # show something to see it isn't frozen
        start_timer = time.monotonic_ns()
        report = nrf.send(payload)
        end_timer = time.monotonic_ns()
        # show something to see it isn't frozen
        if report:
            print(
                "Transmission of payloadID {} as node {} successfull!".format(
                    counter, node_number
                ),
                "Transmission time: {} us".format(int((end_timer - start_timer) / 1000))
            )
        else:
            print("Transmission failed or timed out")
        time.sleep(0.5)  # slow down the test for readability


```

1.4 Scanner Example

New in version 2.0.0.

This example simply scans the entire RF frequency (2.4 GHz to 2.525 GHz) and outputs a vertical graph of how many signals (per *channel*) were detected. This example can be used to find a frequency with the least ambient interference from other radio-emitting sources (i.e. WiFi, Bluetooth, or etc).

Listing 4: examples/nrf24l01_scanner_test.py

```
import time
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object
```

(continues on next page)

(continued from previous page)

```

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

# turn off RX features specific to the nRF24L01 module
nrf.auto_ack = False
nrf.dynamic_payloads = False
nrf.crc = 0
nrf.arc = 0
nrf.allow_ask_no_ack = False

# use reverse engineering tactics for a better "snapshot"
nrf.address_length = 2
nrf.open_rx_pipe(1, b"\0\x55")
nrf.open_rx_pipe(0, b"\0\xAA")


def scan(timeout=30):
    """Traverse the spectrum of accessible frequencies and print any detection
    of ambient signals.

    :param int timeout: The number of seconds in which scanning is performed.
    """
    # print the vertical header of channel numbers
    print("0" * 100 + "1" * 26)
    for i in range(13):
        print(str(i % 10) * (10 if i < 12 else 6), sep="", end="")
    print("")  # endl
    for i in range(126):
        print(str(i % 10), sep="", end="")
    print("\n" + "~" * 126)

    signals = [0] * 126  # store the signal count for each channel
    curr_channel = 0
    start_timer = time.monotonic()  # start the timer
    while time.monotonic() - start_timer < timeout:
        nrf.channel = curr_channel
        if nrf.available():
            nrf.flush_rx()  # flush the RX FIFO because it asserts the RPD flag
        nrf.listen = 1  # start a RX session
        time.sleep(0.00013)  # wait 130 microseconds
        signals[curr_channel] += nrf.rpd  # if interference is present
        nrf.listen = 0  # end the RX session
```

(continues on next page)

(continued from previous page)

```
        curr_channel = curr_channel + 1 if curr_channel < 125 else 0

        # output the signal counts per channel
        sig_cnt = signals[curr_channel]
        print(
            ("%X" % min(15, sig_cnt)) if sig_cnt else "-",
            sep="",
            end="" if curr_channel < 125 else "\r",
        )
    # finish printing results and end with a new line
    while curr_channel < len(signals) - 1:
        curr_channel += 1
        sig_cnt = signals[curr_channel]
        print(("%X" % min(15, sig_cnt)) if sig_cnt else "-", sep="", end="")
    print("")


def noise(timeout=1, channel=None):
    """print a stream of detected noise for duration of time.

    :param int timeout: The number of seconds to scan for ambient noise.
    :param int channel: The specific channel to focus on. If not provided, then the
        radio's current setting is used.
    """
    if channel is not None:
        nrf.channel = channel
    nrf.listen = True
    timeout += time.monotonic()
    while time.monotonic() < timeout:
        signal = nrf.read()
        if signal:
            print(address_repr(signal, False, " "))
    nrf.listen = False
    while not nrf.fifo(False, True):
        # dump the left overs in the RX FIFO
        print(address_repr(nrf.read(), False, " "))


```

1.4.1 Reading the scanner output

Hint: Make sure the terminal window used to run the scanner example is expanded to fit 125 characters. Otherwise the output will look weird.

The output of the scanner example is supposed to be read vertically (as columns). So, the following

```
000
111
789
~~~
```

13-

should be interpreted as

- 1 signal detected on channel 017
- 3 signals detected on channel 018
- no signal (-) detected on channel 019

The ~ is just a divider between the vertical header and the signal counts.

1.5 IRQ Pin Example

Changed in version 1.2.0: uses ACK payloads to trigger all 3 IRQ events.

Changed in version 2.0.0: uses 2 addresses on pipes 1 & 0 to demonstrate proper addressing convention.

This is a test to show how to use nRF24L01's interrupt pin using the non-blocking *write()*. 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 5: examples/nrf24l01_interrupt_test.py

```
import time
import board
import digitalio

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

# select your digital input pin that's connected to the IRQ pin on the nRF4L01
irq_pin = digitalio.DigitalInOut(board.D12)
irq_pin.switch_to_input()  # make sure its an input object
# change these (digital output) pins accordingly
CE_PIN = digitalio.DigitalInOut(board.D4)
CSN_PIN = digitalio.DigitalInOut(board.D5)

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

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

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

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

```

(continues on next page)

(continued from previous page)

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

# to use different addresses on a pair of radios, we need a variable to
# uniquely identify which address this radio will use to transmit
# 0 uses address[0] to transmit, 1 uses address[1] to transmit
radio_number = bool(
    int(input("Which radio is this? Enter '0' or '1'. Defaults to '0' ") or 0)
)

# set TX address of RX node into the TX pipe
nrf.open_tx_pipe(address[radio_number])  # always uses pipe 0

# set RX address of TX node into an RX pipe
nrf.open_rx_pipe(1, address[not radio_number])  # using pipe 1


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


def master():
    """Transmits 3 times: successfully receive ACK payload first, successfully
    transmit on second, and intentionally fail transmit on the third"""
    nrf.listen = False  # ensures the nRF24L01 is in TX mode
    # NOTE nrf.write() internally calls nrf.clear_status_flags() first

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

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

    # on data sent test
```

(continues on next page)

(continued from previous page)

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

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

    # on data fail test
    print("\nConfiguring IRQ pin to go active for all events.")
    nrf.interrupt_config()
    print("    Sending a ping to inactive slave node...", end=" ")
    nrf.flush_tx()  # just in case any previous tests failed
    nrf.write(b"Dummy", write_only=True)  # CE pin is left LOW
    _ping_and_prompt()  # CE pin is managed by this function
    print("\t\"on data failed\" event test {}successful".format("un" * nrf.irq_df))
    nrf.flush_tx()  # flush artifact payload in TX FIFO from last test
    # all 3 ACK payloads received were 4 bytes each, and RX FIFO is full
    # so, fetching 12 bytes from the RX FIFO also flushes RX FIFO
    print("\nComplete RX FIFO:", nrf.read(12))


def slave(timeout=6):  # will listen for 6 seconds before timing out
    """Only listen for 3 payload from the master node"""
    # setup radio to receive pings, fill TX FIFO with ACK payloads
    nrf.load_ack(b"Yak ", 1)
    nrf.load_ack(b"Back", 1)
    nrf.load_ack(b" ACK", 1)
    nrf.listen = True  # start listening & clear irq_dr flag
    start_timer = time.monotonic()  # start timer now
    while not nrf.fifo(0, 0) and time.monotonic() - start_timer < timeout:
        # if RX FIFO is not full and timeout is not reached, then keep going
        pass
    nrf.listen = False  # put nRF24L01 in Standby-I mode when idling
    if not nrf.fifo(False, True):  # if RX FIFO is not empty
        # all 3 payloads received were 5 bytes each, and RX FIFO is full
        # so, fetching 15 bytes from the RX FIFO also flushes RX FIFO
        print("Complete RX FIFO:", nrf.read(15))
    nrf.flush_tx()  # discard any pending ACK payloads


```

CHAPTER

TWO

LIBRARY-SPECIFIC FEATURES

2.1 Stream Example

Changed in version 1.2.3: added `master_fifo()` to demonstrate using full TX FIFO to stream data.

Changed in version 2.0.0: uses 2 addresses on pipes 1 & 0 to demonstrate proper addressing convention.

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

Listing 1: examples/nrf24l01_stream_test.py

```
import time
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


```

(continues on next page)

(continued from previous page)

```
# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

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

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

# to use different addresses on a pair of radios, we need a variable to
# uniquely identify which address this radio will use to transmit
# 0 uses address[0] to transmit, 1 uses address[1] to transmit
radio_number = bool(
    int(input("Which radio is this? Enter '0' or '1'. Defaults to '0' ") or 0)
)

# set TX address of RX node into the TX pipe
nrf.open_tx_pipe(address[radio_number])  # always uses pipe 0

# set RX address of TX node into an RX pipe
nrf.open_rx_pipe(1, address[not radio_number])  # using pipe 1

# uncomment the following 2 lines for compatibility with TMRh20 library
# nrf.allow_ask_no_ack = False
nrf.dynamic_payloads = False


def make_buffers(size=32):
    """return a list of payloads"""
    buffers = []
    # we'll use `size` for the number of payloads in the list and the
    # payloads' length
    for i in range(size):
        # prefix payload with a sequential letter to indicate which
        # payloads were lost (if any)
        buff = bytes([i + (65 if 0 <= i < 26 else 71)])
        for j in range(size - 1):
            char = j >= (size - 1) / 2 + abs((size - 1) / 2 - i)
            char |= j < (size - 1) / 2 - abs((size - 1) / 2 - i)
            buff += bytes([char + 48])
        buffers.append(buff)
        del buff
    return buffers


def master(count=1, size=32):  # count = 5 will transmit the list 5 times
    """Transmits multiple payloads using `RF24.send()` and `RF24.resend()`."""
```

(continues on next page)

(continued from previous page)

```
    buffers = make_buffers(size)  # make a list of payloads
    nrf.listen = False  # ensures the nRF24L01 is in TX mode
    successful = 0  # keep track of success rate
    for _ in range(count):
        start_timer = time.monotonic_ns()  # start timer
        # NOTE force_retry=2 internally invokes `RF24.resend()` 2 times at
        # most for payloads that fail to transmit.
        result = nrf.send(buffers, force_retry=2)  # result is a list
        end_timer = time.monotonic_ns()  # end timer
        print("Transmission took", (end_timer - start_timer) / 1000, "us")
        for r in result:  # tally the resulting success rate
            successful += 1 if r else 0
    print(
        "successfully sent {}%".format(successful / (size * count) * 100),
        "({}/{})".format(successful, size * count)
    )


def master_fifo(count=1, size=32):
    """Similar to the `master()` above except this function uses the full
    TX FIFO and `RF24.write()` instead of `RF24.send()`"""
    buf = make_buffers(size)  # make a list of payloads
    nrf.listen = False  # ensures the nRF24L01 is in TX mode
    for cnt in range(count):  # transmit the same payloads this many times
        nrf.flush_tx()  # clear the TX FIFO so we can use all 3 levels
        # NOTE the write_only parameter does not initiate sending
        buf_iter = 0  # iterator of payloads for the while loop
        failures = 0  # keep track of manual retries
        start_timer = time.monotonic_ns()  # start timer
        while buf_iter < size:  # cycle through all the payloads
            nrf.ce_pin = False
            while buf_iter < size and nrf.write(buf[buf_iter], write_only=1):
                # NOTE write() returns False if TX FIFO is already full
                buf_iter += 1  # increment iterator of payloads
            nrf.ce_pin = True
            while not nrf.fifo(True, True):  # updates irq_df flag
                if nrf.irq_df:
                    # reception failed; we need to reset the irq_rf flag
                    nrf.ce_pin = False  # fall back to Standby-I mode
                    failures += 1  # increment manual retries
                    nrf.clear_status_flags()  # clear the irq_df flag
                    if failures > 99 and buf_iter < 7 and cnt < 2:
                        # we need to prevent an infinite loop
                        print(
                            "Make sure slave() node is listening."
                            " Quiting master_fifo()"
                        )
                        buf_iter = size + 1  # be sure to exit the while loop
                        nrf.flush_tx()  # discard all payloads in TX FIFO
                    else:
                        nrf.ce_pin = True  # start re-transmitting
        nrf.ce_pin = False
```

(continues on next page)

(continued from previous page)

```
        end_timer = time.monotonic_ns()  # end timer
        print(
            "Transmission took {} us".format((end_timer - start_timer) / 1000),
            "with {} failures detected.".format(failures)
        )


def slave(timeout=5):
    """Stops listening after a `timeout` with no response"""
    nrf.listen = True  # put radio into RX mode and power up
    count = 0  # keep track of the number of received payloads
    start_timer = time.monotonic()  # start timer
    while time.monotonic() < start_timer + timeout:
        if nrf.available():
            count += 1
            # retreive the received packet's payload
            buffer = nrf.read()  # clears flags & empties RX FIFO
            print("Received:", buffer,"-", count)
            start_timer = time.monotonic()  # reset timer on every RX payload

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


```

2.2 Context Example

Changed in version 1.2.0: demonstrates switching between *FakeBLE* object & *RF24* object with the same nRF24L01

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

Listing 2: examples/nrf24l01_context_test.py

```
from circuitpython_nrf24l01.rf24 import RF24
from circuitpython_nrf24l01.fake_ble import FakeBLE

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object
```

(continues on next page)

(continued from previous page)

```

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


# initialize the nRF24L01 objects on the spi bus object
# the first object will have all the features enabled
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                21 = bus 2, CE1  # enable SPI bus 1 prior to running this

# enable the option to use custom ACK payloads
nrf.ack = True
# set the static payload length to 8 bytes
nrf.payload_length = 8
# RF power amplifier is set to -18 dbm
nrf.pa_level = -18

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

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

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

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

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

# NOTE it is not advised to manipulate separate RF24 objects outside of the
# "with" block; you will encounter bugs about configurations when doing so.
```

(continues on next page)

(continued from previous page)

```
# Be sure to use 1 "with" block per RF24 object when instantiating multiple
# RF24 objects in your program.
# NOTE exiting a "with" block will always power down the nRF24L01
# NOTE upon instantiation, this library closes all RX pipes &
# extracts the TX/RX addresses from the nRF24L01 registers
```

2.3 Manual ACK Example

New in version 2.0.0: Previously, this example was strictly made for TMRh20's RF24 library example titled "GettingStarted_HandlingData.ino". With the latest addition of new examples to the TMRh20 RF24 library, this example was renamed from "nrf24l01_2arduino_handling_data.py" and adapted for both this library and TMRh20's RF24 library.

This is a test to show how to use the library for acknowledgement (ACK) responses without using the automatic ACK packets (like the *ACK Payloads Example* does). Beware, that this technique is not faster and can be more prone to communication failure. However, This technique has the advantage of using more updated information in the responding payload as information in ACK payloads are always outdated by 1 transmission.

Listing 3: examples/nrf24l01_manual_ack_test.py

```
import time
import board
from digitalio import DigitalInOut

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

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)

# initialize the nRF24L01 on the spi bus object
nrf = RF24(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                 0 = bus 0, CE0  # SPI bus 0 is enabled by default
```

(continues on next page)

(continued from previous page)

```
#                 10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                 21 = bus 2, CE1  # enable SPI bus 1 prior to running this

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

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

# to use different addresses on a pair of radios, we need a variable to
# uniquely identify which address this radio will use to transmit
# 0 uses address[0] to transmit, 1 uses address[1] to transmit
radio_number = bool(
    int(input("Which radio is this? Enter '0' or '1'. Defaults to '0' ") or 0)
)

# set TX address of RX node into the TX pipe
nrf.open_tx_pipe(address[radio_number])  # always uses pipe 0

# set RX address of TX node into an RX pipe
nrf.open_rx_pipe(1, address[not radio_number])  # using pipe 1
# nrf.open_rx_pipe(2, address[radio_number])  # for getting responses on pipe 2

# using the python keyword global is bad practice. Instead we'll use a 1 item
# list to store our integer number for the payloads' counter
counter = [0]

# uncomment the following 3 lines for compatibility with TMRh20 library
# nrf.allow_ask_no_ack = False
# nrf.dynamic_payloads = False
# nrf.payload_length = 8


def master(count=5):  # count = 5 will only transmit 5 packets
    """Transmits an arbitrary unsigned long value every second"""
    nrf.listen = False  # ensures the nRF24L01 is in TX mode
    while count:
        # construct a payload to send
        # add b"\0" as a c-string NULL terminating char
        buffer = b"Hello \0" + bytes([counter[0]])
        start_timer = time.monotonic_ns()  # start timer
        result = nrf.send(buffer)  # save the response (ACK payload)
        if not result:
            print("send() failed or timed out")
        else:  # sent successful; listen for a response
            nrf.listen = True  # get radio ready to receive a response
            timeout = time.monotonic_ns() + 200000000  # set sentinel for timeout
            while not nrf.available() and time.monotonic_ns() < timeout:
                # this loop hangs for 200 ms or until response is received
                pass
            nrf.listen = False  # put the radio back in TX mode
```

(continues on next page)

(continued from previous page)

```
            end_timer = time.monotonic_ns()  # stop timer
            print(
                "Transmission successful! Time to transmit:",
                int((end_timer - start_timer) / 1000),
                "us. Sent: {}{}".format(buffer[:6].decode("utf-8"), counter[0]),
                end=" ",
            )
            if nrf.pipe is None:  # is there a payload?
                # nrf.pipe is also updated using `nrf.listen = False`
                print("Received no response.")
            else:
                length = nrf.any()  # reset with read()
                pipe_number = nrf.pipe  # reset with read()
                received = nrf.read()  # grab the response
                # save new counter from response
                counter[0] = received[7:8][0]
                print(
                    "Received {} bytes with pipe {}:".format(length, pipe_number),
                    "{}{}".format(bytes(received[:6]).decode("utf-8"), counter[0]),
                )
        count -= 1
        # make example readable in REPL by slowing down transmissions
        time.sleep(1)


def slave(timeout=6):
    """Polls the radio and prints the received value. This method expires
    after 6 seconds of no received transmission"""
    nrf.listen = True  # put radio into RX mode and power up
    start_timer = time.monotonic()  # used as a timeout
    while (time.monotonic() - start_timer) < timeout:
        if nrf.available():
            length = nrf.any()  # grab payload length info
            pipe = nrf.pipe  # grab pipe number info
            received = nrf.read(length)  # clears info from any() and nrf.pipe
            # increment counter before sending it back in responding payload
            counter[0] = received[7:8][0] + 1
            nrf.listen = False  # put the radio in TX mode
            result = False
            ack_timeout = time.monotonic_ns() + 200000000
            while not result and time.monotonic_ns() < ack_timeout:
                # try to send reply for 200 milliseconds (at most)
                result = nrf.send(b"World \0" + bytes([counter[0]]))
            nrf.listen = True  # put the radio back in RX mode
            print(
                "Received {} on pipe {}:".format(length, pipe),
                "{}{}".format(bytes(received[:6]).decode("utf-8"), received[7:8][0]),
                end=" Sent: ",
            )
            if not result:
                print("Response failed or timed out")
            else:
```

(continues on next page)

(continued from previous page)

```
                print("World", counter[0])
            start_timer = time.monotonic()  # reset timeout

    # recommended behavior is to keep in TX mode when in idle
    nrf.listen = False  # put the nRF24L01 in TX mode + Standby-I power state


```

2.4 Network Test

New in version 2.1.0.

The following network example is designed to be compatible with most of TMRh20's C++ examples for the RF24Mesh and RF24Network libraries. However, due to some slight differences this example prompts for user input which can cover a broader spectrum of usage scenarios.

Listing 4: examples/nrf24l01_network_test.py

```
import time
import struct
import board
from digitalio import DigitalInOut
from circuitpython_nrf24l01.network.constants import MAX_FRAG_SIZE, NETWORK_DEFAULT_ADDR

IS_MESH = (
    input(
        "    nrf24l01_network_test example\n"
        "Would you like to run as a mesh network node (y/n)? Defaults to 'Y' "
    ) or "Y"
).upper().startswith("Y")


# to use different addresses on a set of radios, we need a variable to
# uniquely identify which address this radio will use
THIS_NODE = 0
print(
    "Remember, the master node always uses `0` as the node_address and node_id."
    "\nWhich node is this? Enter",
    end=" ",
)
if IS_MESH:
    # node_id must be less than 256
    THIS_NODE = int(input("a unique int. Defaults to '0' ") or "0") & 0xFF
else:
    # logical node_address is in octal
    THIS_NODE = int(input("an octal int. Defaults to '0' ") or "0", 8)

if IS_MESH:
    if THIS_NODE:  # if this is not a mesh network master node
        from circuitpython_nrf24l01.rf24_mesh import RF24MeshNoMaster as Network
    else:
```

(continues on next page)

(continued from previous page)

```
        from circuitpython_nrf24l01.rf24_mesh import RF24Mesh as Network
    print("Using RF24Mesh{} class".format("" if not THIS_NODE else "NoMaster"))
else:
    from circuitpython_nrf24l01.rf24_network import RF24Network as Network

    # we need to construct frame headers for RF24Network.send()
    from circuitpython_nrf24l01.network.structs import RF24NetworkHeader

    # we need to construct entire frames for RF24Network.write() (not for this example)
    # from circuitpython_nrf24l01.network.structs import RF24NetworkFrame
    print("Using RF24Network class")

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)


# initialize this node as the network
nrf = Network(SPI_BUS, CSN_PIN, CE_PIN, THIS_NODE)

# TMRh20 examples use channel 97 for RF24Mesh library
# TMRh20 examples use channel 90 for RF24Network library
nrf.channel = 90 + IS_MESH * 7

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

# using the python keyword global is bad practice. Instead we'll use a 1 item
# list to store our number of the payloads sent
packets_sent = [0]

if THIS_NODE:  # if this node is not the network master node
    if IS_MESH:  # mesh nodes need to bond with the master node
        print("Connecting to mesh network...", end=" ")

        # get this node's assigned address and connect to network
```

(continues on next page)

(continued from previous page)

```
        if nrf.renew_address() is None:
            print("failed. Please try again manually with `nrf.renew_address()`")
        else:
            print("assigned address:", oct(nrf.node_address))
else:
    print("Acting as network master node.")


def idle(timeout: int = 30, strict_timeout: bool = False):
    """Listen for any payloads and print the transaction

    :param int timeout: The number of seconds to wait (with no transmission)
        until exiting function.
    :param bool strict_timeout: If set to True, then the timer is not reset when
        processing incoming traffic
    """
    print("idling for", timeout, "seconds")
    start_timer = time.monotonic()
    while (time.monotonic() - start_timer) < timeout:
        nrf.update()  # keep the network layer current
        while nrf.available():
            if not strict_timeout:
                start_timer = time.monotonic()  # reset timer
            frame = nrf.read()
            message_len = len(frame.message)
            print("Received payload", end=" ")
            # TMRh20 examples only use 1 or 2 long ints as small messages
            if message_len < MAX_FRAG_SIZE and message_len % 4 == 0:
                # if not a large fragmented message and multiple of 4 bytes
                fmt = "<" + "L" * int(message_len / 4)
                print(struct.unpack(fmt, bytes(frame.message)), end=" ")
            print(frame.header.to_string(), "length", message_len)


def emit(
    node: int = not THIS_NODE, frag: bool = False, count: int = 5, interval: int = 1
):
    """Transmits 1 (or 2) integers or a large buffer

    :param int node: The target node for network transmissions.
        If using RF24Mesh, this is a unique node_id.
        If using RF24Network, this is the node's logical address.
    :param bool frag: Only use fragmented messages?
    :param int count: The max number of messages to transmit.
    :param int interval: time (in seconds) between transmitting messages.
    """
    while count:
        idle(interval, True)  # idle till its time to emit
        count -= 1
        packets_sent[0] += 1
        # TMRh20's RF24Mesh examples use 1 long int containing a timestamp (in ms)
        message = struct.pack("<L", int(time.monotonic() * 1000))
```

(continues on next page)

(continued from previous page)

```
        if frag:
            message = bytes(
                range((packets_sent[0] + MAX_FRAG_SIZE) % nrf.max_message_length)
            )
        elif not IS_MESH:  # if using RF24Network
            # TMRh20's RF24Network examples use 2 long ints, so add another
            message += struct.pack("<L", packets_sent[0])
        result = False
        start = time.monotonic_ns()
        # pylint: disable=no-value-for-parameter
        if IS_MESH:  # send() is a little different for RF24Mesh vs RF24Network
            result = nrf.send(node, "M", message)
        else:
            result = nrf.send(RF24NetworkHeader(node, "T"), message)
        # pylint: enable=no-value-for-parameter
        end = time.monotonic_ns()
        print(
            "Sending {} (len {})...".format(packets_sent[0], len(message)),
            "ok." if result else "failed.",
            "Transmission took {} ms".format(int((end - start) / 1000000)),
        )


```

CHAPTER

THREE

OTA COMPATIBILITY

3.1 Fake BLE Example

New in version 1.2.0.

Changed in version 2.1.0: A new `slave()` function was added to demonstrate receiving BLE data.

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

Listing 1: examples/nrf24l01_fake_ble_test.py

```
import time
import board
from digitalio import DigitalInOut
from circuitpython_nrf24l01.fake_ble import (
    chunk,
    FakeBLE,
    UrlServiceData,
    BatteryServiceData,
    TemperatureServiceData,
)
from circuitpython_nrf24l01.rf24 import address_repr

# invalid default values for scoping
SPI_BUS, CSN_PIN, CE_PIN = (None, None, None)

try:  # on Linux
    import spidev

    SPI_BUS = spidev.SpiDev()  # for a faster interface on linux
    CSN_PIN = 0  # use CE0 on default bus (even faster than using any pin)
    CE_PIN = DigitalInOut(board.D22)  # using pin gpio22 (BCM numbering)

except ImportError:  # on CircuitPython only
    # using board.SPI() automatically selects the MCU's
    # available SPI pins, board.SCK, board.MOSI, board.MISO
    SPI_BUS = board.SPI()  # init spi bus object

    # change these (digital output) pins accordingly
    CE_PIN = DigitalInOut(board.D4)
    CSN_PIN = DigitalInOut(board.D5)

```

(continues on next page)

(continued from previous page)

```

# initialize the nRF24L01 on the spi bus object as a BLE compliant radio
nrf = FakeBLE(SPI_BUS, CSN_PIN, CE_PIN)
# On Linux, csn value is a bit coded
#                  0 = bus 0, CE0  # SPI bus 0 is enabled by default
#                 10 = bus 1, CE0  # enable SPI bus 2 prior to running this
#                 21 = bus 2, CE1  # enable SPI bus 1 prior to running this

# the name parameter is going to be its broadcasted BLE name
# this can be changed at any time using the `name` attribute
# nrf.name = b"foobar"

# you can optionally set the arbitrary MAC address to be used as the
# BLE device's MAC address. Otherwise this is randomly generated upon
# instantiation of the FakeBLE object.
# nrf.mac = b"\x19\x12\x14\x26\x09\xE0"

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


def _prompt(remaining):
    if remaining % 5 == 0 or remaining < 5:
        if remaining - 1:
            print(remaining, "advertisements left to go!")
        else:
            print(remaining, "advertisement left to go!")


# create an object for manipulating the battery level data
battery_service = BatteryServiceData()
# battery level data is 1 unsigned byte representing a percentage
battery_service.data = 85


def master(count=50):
    """Sends out the device information."""
    # using the "with" statement is highly recommended if the nRF24L01 is
    # to be used for more than a BLE configuration
    with nrf as ble:
        ble.name = b"nRF24L01"
        # include the radio's pa_level attribute in the payload
        ble.show_pa_level = True
        print(
            "available bytes in next payload:",
            ble.len_available(chunk(battery_service.buffer)),
        )  # using chunk() gives an accurate estimate of available bytes
        for i in range(count):  # advertise data this many times
            if ble.len_available(chunk(battery_service.buffer)) >= 0:
                _prompt(count - i)  # something to show that it isn't frozen
```

(continues on next page)

(continued from previous page)

```
                # broadcast the device name, MAC address, &
                # battery charge info; 0x16 means service data
                ble.advertise(battery_service.buffer, data_type=0x16)
                # channel hoping is recommended per BLE specs
                ble.hop_channel()
                time.sleep(0.5)  # wait till next broadcast
    # nrf.show_pa_level & nrf.name both are set to false when
    # exiting a with statement block


# create an object for manipulating temperature measurements
temperature_service = TemperatureServiceData()
# temperature's float data has up to 2 decimal places of precision
temperature_service.data = 42.0


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


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


def send_url(count=50):
    """Sends out a URL."""
    with nrf as ble:
        print(
            "available bytes in next payload:",
            ble.len_available(chunk(url_service.buffer)),
        )
        # NOTE we did NOT set a device name in this with block
```

(continues on next page)

(continued from previous page)

```
        for i in range(count):
            # URLs easily exceed the nRF24L01's max payload length
            if ble.len_available(chunk(url_service.buffer)) >= 0:
                _prompt(count - i)
                ble.advertise(url_service.buffer, 0x16)
                ble.hop_channel()
                time.sleep(0.2)


def slave(timeout=6):
    """read and decipher BLE payloads for `timeout` seconds."""
    nrf.listen = True
    end_timer = time.monotonic() + timeout
    while time.monotonic() <= end_timer:
        if nrf.available():
            result = nrf.read()
            print(
                "received payload from MAC address",
                address_repr(result.mac, delimit=":")
            )
            if result.name is not None:
                print("\tdevice name:", result.name)
            if result.pa_level is not None:
                print("\tdevice transmitting PA Level:", result.pa_level, "dbm")
            for service_data in result.data:
                if isinstance(service_data, (bytearray, bytes)):
                    print("\traw buffer:", address_repr(service_data, False, " "))
                else:
                    print("\t" + repr(service_data))
    nrf.listen = False
    nrf.flush_rx()  # discard any received raw BLE data


```

3.2 TMRh20's C++ libraries

All examples are designed to work with TMRh20's RF24, RF24Network, and RF24Mesh libraries' examples. This Circuitpython library uses dynamic payloads enabled by default. TMRh20's RF24 library uses static payload lengths by default.

To make this circuitpython library compatible with TMRh20's RF24 library:

1. set *dynamic_payloads* to `False`.
2. set *allow_ask_no_ack* to `False`.
3. set *payload_length* to the value that is passed to TMRh20's `RF24::setPayloadSize()`. 32 is the default (& maximum) payload length/size for both libraries.

> **Warning:** Certain C++ datatypes allocate a different amount of memory depending on the board being used in the Arduino IDE. For example, `uint8_t` isn't always allocated to 1 byte of memory for certain boards.

> Make sure you understand the amount of memory that different datatypes occupy in C++. This will help you comprehend how to configure `payload_length`.

For completeness, TMRh20's RF24 library uses a default value of 15 for the `ard` attribute, but this Circuitpython library uses a default value of 3.

Table 1: Corresponding examples

circuitpython_nrf24l01	TMRh20's C++ examples
nrf24l01_simple_test ([1])	RF24 gettingStarted
nrf24l01_ack_payload_test	RF24 acknowledgementPayloads
nrf24l01_manual_ack_test ([1])	RF24 manualAcknowledgements
nrf24l01_multiceiver_test ([1])	RF24 multiceiverDemo
nrf24l01_stream_test ([1])	RF24 streamingData
nrf24l01_interrupt_test	RF24 interruptConfigure
nrf24l01_context_test	feature is not available in C++
nrf24l01_fake_ble_test	feature is available via floe's BTLE library
nrf24l01_network_test ([2])	• all RF24Network examples except Network_Ping & Network_Ping_Sleep • all RF24Mesh examples except RF24Mesh_Example_Node2NodeExtra (which may still work but the data is not interpreted as a string)

[1] Some of the Circuitpython examples (that are compatible with TMRh20's examples) contain 2 or 3 lines of code that are commented out for easy modification. These lines look like this in the examples' source code:

```
# uncomment the following 3 lines for compatibility with TMRh20 library
# nrf.allow_ask_no_ack = False
# nrf.dynamic_payloads = False
# nrf.payload_length = 4
```

[2] When running the network examples, it is important to understand the typical network topology. Otherwise, entering incorrect answers to the example's user prompts may result in seemingly bad connections.

CHAPTER

FOUR

BASIC RF24 API

class circuitpython_nrf24l01.rf24.**RF24**(spi: *busio.SPI*, csn: *digitalio.DigitalInOut*, ce_pin: *digitalio.DigitalInOut*, 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_pin : **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 *RF24* object and is made persistent via `SPIDevice`.

Changed in version 1.2.0:

- new `spi_frequency` parameter
- removed all keyword arguments in favor of using the provided corresponding attributes.

RF24.**open_tx_pipe**(address: *bytes*, *bytearray*) → None

Open a data pipe for 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 1) 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.

`RF24.close_rx_pipe(pipe_number: int) → None`

Close a specific data pipe from RX transmissions.

Parameters

pipe_number : int
The data pipe to use for RX transactions. This must be in range [0, 5]. Otherwise a `IndexError` exception is thrown.

Changed in version 1.2.0: removed the `reset` parameter. Addresses assigned to pipes will persist until changed or power to the nRF24L01 is discontinued.

`RF24.open_rx_pipe(pipe_number: int, address: bytes, bytearray) → None`

Open a specific data pipe for RX transmissions.

Parameters

pipe_number : int
The data pipe to use for RX transactions. This must be in range [0, 5]. Otherwise a `IndexError` exception is thrown.

address : bytearray,bytes
The virtual address to the receiving nRF24L01. If using a `pipe_number` greater than 1, then only the MSByte of the address is written, so make sure MSByte (first character) is unique among other simultaneously receiving addresses. The existing address can be altered by writing a bytearray with a length less than 5. The nRF24L01 will use the first *address_length* number of bytes for the RX address on the specified data pipe.

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

`RF24.listen`

This attribute is the 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 if 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, puts nRF24L01 in power up mode. Notice the CE pin is be held HIGH during RX mode.
- `False` disables RX mode. As mentioned in above link, this puts nRF24L01's power in Standby-I mode (CE pin is LOW meaning low current & no transmissions) which is ideal for post-reception work. Disabling RX mode doesn't flush the RX FIFO buffers, so remember to flush your 3-level FIFO buffers when appropriate using *flush_tx()* or *flush_rx()* (see also the *read()* function).

Note: When *ack* payloads are enabled, this attribute flushes the TX FIFO buffers upon exiting RX mode. However, this attribute does not flush the TX FIFO buffers when entering RX mode. This is done to better manage the ACK payloads loaded into the TX FIFO.

Changed in version 2.1.0: Prior to v2.1.0 this attribute would clear the status flags when entering RX mode. This was removed to expedite applications that use manually transmitted acknowledgement payloads.

`RF24.any()` → int

This function 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.

`RF24.available()` → bool

A `bool` describing if there is a payload in the RX FIFO.

This function is provided for convenience and is synonymous with the following statement:

```
# let `nrf` be the instantiated RF24 object
nrf.update() and nrf.pipe is not None
```

New in version 2.0.0.

`RF24.read(length: int | None = None)` → bytearray

This function is used to retrieve data from the RX FIFO.

The *irq_dr* status flag is reset automatically. This function can also be used to fetch the last ACK packet's payload if *ack* is enabled.

Parameters

length : int

An optional parameter to specify how many bytes to read from the RX FIFO buffer. This parameter is not constrained in any way.

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

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

Returns

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

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

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

New in version 1.2.0: `length` parameter

Changed in version 2.0.0: renamed this method from `recv()` to `read()` because it isn't doing any actual receiving. Rather, it is only reading data from the RX FIFO that was already received/validated by the radio.

`RF24.send`(buf: *bytes, bytearray, Sequence[bytes, bytearray]*, ask_no_ack: *bool* = ***False***, force_retry: *int* = *0*, send_only: *bool* = ***False***) → bool, bytearray, list[bool, bytearray]

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 *auto_ack* is enabled for data pipe 0.
- `True` if transmission succeeds.
- `bytearray` or `True` when the *ack* attribute is `True`. Because the payload expects a responding custom ACK payload, the response is returned (upon successful transmission) as a `bytearray` (or `True` if ACK payload is empty). Returning the ACK payload can be bypassed by setting the `send_only` parameter as `True`.

Parameters

buf : bytearray,bytes,list,tuple

The payload to transmit. This bytearray must have a length in range [1, 32], otherwise a `ValueError` exception is thrown. This can also be a list or tuple of payloads (`bytearray`); in which case, all items in the list/tuple are processed for consecutive transmissions.

- **If the *dynamic_payloads* attribute is disabled for data pipe 0 and this** bytearray's length is less than the *payload_length* attribute for pipe 0, then this bytearray is padded with zeros until its length is equal to the *payload_length* attribute for pipe 0.
- **If the *dynamic_payloads* attribute is disabled for data pipe 0 and this** bytearray's length is greater than *payload_length* attribute for pipe 0, then this bytearray's length is truncated to equal the *payload_length* attribute for pipe 0.

ask_no_ack : bool

Pass this parameter as `True` to tell the nRF24L01 not to wait for an acknowledgment from the receiving nRF24L01. This parameter directly controls a `NO_ACK` flag in the transmission's Packet Control Field (9 bits of information about the payload). Therefore, it takes advantage of an nRF24L01 feature specific to individual payloads, and its value is not saved anywhere. You do not need to specify this for every payload if the *auto_ack* attribute is disabled (for data pipe 0), however setting this parameter to `True` will work despite the *auto_ack* attribute's setting.

Important: If the *allow_ask_no_ack* attribute is disabled (set to `False`), then this parameter will have no affect at all. By default the *allow_ask_no_ack* attribute is enabled.

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 *auto_ack* is disabled or if `ask_no_ack` parameter is set to `True`. Each re-attempt still takes advantage of Auto-Retry feature. During multi-payload processing, this parameter is meant to slow down CircuitPython devices just enough for the Raspberry Pi to catch up (due to the Raspberry Pi's seemingly slower SPI speeds).

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

Tip: It is highly recommended that *auto_ack* attribute is enabled when sending multiple payloads. Test results with the *auto_ack* 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 any payloads in the TX FIFO when called, but failed transmissions' payloads will remain in the TX FIFO until *send()* or *flush_tx()* is called after failed transmissions.

New in version 1.2.0: `send_only` parameter

CHAPTER

FIVE

ADVANCED RF24 API

`RF24.resend(send_only: bool = False)`

Manually re-send the first-out payload from TX FIFO buffers.

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

Returns

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

Parameters

send_only : bool

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

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

`RF24.write(buf: bytes, bytearray, ask_no_ack: bool = False, write_only: bool = False) → bool`

This non-blocking and helper function to *send()* can only handle one payload at a time.

This function isn't completely non-blocking as we still need to wait for the necessary SPI transactions to complete. Example usage of this function can be seen in the IRQ pin example and in the Stream example's "master_fifo()" function

Returns

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

Parameters

buf : bytearray

The payload to transmit. This bytearray must have a length greater than 0 and less than 32 bytes, otherwise a `ValueError` exception is thrown.

- If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is less than the *payload_length* attribute for data pipe 0, then this bytearray is padded with zeros until its length is equal to the *payload_length* attribute for data pipe 0.
- If the *dynamic_payloads* attribute is disabled for data pipe 0 and this bytearray's length is greater than *payload_length* attribute for data pipe 0, then this bytearray's length is truncated to equal the *payload_length* attribute for data pipe 0.

ask_no_ack : bool

Pass this parameter as `True` to tell the nRF24L01 not to wait for an acknowledgment from the receiving nRF24L01. This parameter directly controls a `NO_ACK` flag in the transmission's Packet Control Field (9 bits of information about the payload). Therefore, it takes advantage of an nRF24L01 feature specific to individual payloads, and its value is not saved anywhere. You do not need to specify this for every payload if the *auto_ack* attribute is disabled, however setting this parameter to `True` will work despite the *auto_ack* attribute's setting.

Important: If the *allow_ask_no_ack* attribute is disabled (set to `False`), then this parameter will have no affect at all. By default the *allow_ask_no_ack* attribute is enabled.

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

write_only : bool

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

Note: The nRF24L01 doesn't initiate sending until a mandatory minimum 10 µs pulse on the CE pin is achieved. If the `write_only` parameter is `False`, then that pulse is initiated before this function exits. However, we have left that 10 µs wait time to be managed by the MCU in cases of 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 µs, then the nRF24L01 will continue to transmit all payloads found in the TX FIFO buffer.

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

It is important to NEVER to keep the nRF24L01+ in TX mode for more than 4 ms at a time. If the [*auto_ack* 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 *auto_ack* 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.

New in version 1.2.0: `write_only` parameter

`RF24.load_ack(buf, pipe_number: int) → bool`

Load a payload into the TX FIFO for use on a specific data pipe.

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

Parameters

buf : bytearray,bytes
This will be the data attached to an automatic ACK packet on the incoming transmission about the specified `pipe_number` parameter. This must have a length in range [1, 32] bytes, otherwise a `ValueError` exception is thrown. Any ACK payloads will remain in the TX FIFO buffer until transmitted successfully or *flush_tx()* is called.

pipe_number : int
This will be the pipe number to use for deciding which transmissions get a response with the specified `buf` parameter's data. This number must be in range [0, 5], otherwise a `IndexError` exception is thrown.

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

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

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

`RF24.power`

This `bool` attribute controls the power state of the nRF24L01.

This is exposed for convenience.

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

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

`RF24.address_length`

This `int` is the length (in bytes) used of RX/TX addresses.

A valid input value must be an `int` in range [3, 5]. Default is set to the nRF24L01's maximum of 5. Any invalid input value results in a address length of 2 bytes.

Changed in version 2.1.0: A `ValueError` exception was thrown when an invalid input value was encountered. This changed to setting the address length to 2 bytes (for possible reverse engineering protocol purposes).

`RF24.address(index: int = -1)`

Returns the current TX address or optionally RX address. (read-only)

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

Parameters

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

New in version 1.2.0.

`RF24.last_tx_arc`

Return the number of attempts made for last transmission (read-only).

This attribute resets to 0 at the beginning of every transmission in TX mode. Remember that the number of automatic retry attempts made for each transmission is configured with the *arc* attribute or the *set_auto_retries()* function.

`RF24.is_plus_variant`

A `bool` describing if the nRF24L01 is a plus variant or not (read-only).

This information is determined upon instantiation.

New in version 1.2.0.

5.1 Debugging Output

`RF24.print_details(dump_pipes: bool = False) → None`

This debugging function outputs all details about the nRF24L01.

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

Prints

- `Is a plus variant` True means the transceiver is a nRF24L01+. False means the transceiver is a nRF24L01 (not a plus variant).
- `Channel` The current setting of the *channel* attribute
- `RF Data Rate` The current setting of the RF *data_rate* attribute.
- `RF Power Amplifier` The current setting of the *pa_level* attribute.
- `CRC bytes` The current setting of the *crc* attribute
- `Address length` The current setting of the *address_length* attribute
- `TX Payload lengths` The current setting of the *payload_length* attribute for TX operations (concerning data pipe 0)

- `Auto retry delay` The current setting of the *ard* attribute
- `Auto retry attempts` The current setting of the *arc* attribute
- `Re-use TX FIFO` Is the first payload in the TX FIFO to be re-used for subsequent transmissions (this flag is set to `True` when entering *resend()* and reset to `False` when *resend()* exits)
- `Packets lost on current channel` Total amount of packets lost (transmission failures). This only resets when the *channel* is changed. This count will only go up to 15.
- `Retry attempts made for last transmission` Amount of attempts to re-transmit during last transmission (resets per payload)
- `IRQ on Data Ready` The current setting of the IRQ pin on "Data Ready" event
- `IRQ on Data Sent` The current setting of the IRQ pin on "Data Sent" event
- `IRQ on 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` The current setting of the *allow_ask_no_ack* attribute.
- `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.

Set this parameter to `False` (it default value) to skips this extra information.

Changed in version v2.1.0: Changed the default value for the `dump_pipes` parameter to `True`

`RF24.print_pipes()` → None

Prints all information specific to pipe's addresses, RX state, & expected static payload sizes (if configured to use static payloads).

This method is called from *`print_details()`* if the `dump_pipes` parameter is set to `True`.

Changed in version v2.1.0: Changed this method's name from the private method `_dump_pipes()` to a public method `print_pipes()`.

`circuitpython_nrf24l01.rf24.address_repr`(buf, reverse: *bool* = ***True***, delimit: *str* = '') → str

Convert a buffer into a hexlified string.

This method is primarily used in *`print_pipes()`* to display how the address is used by the radio.

```
>>> from circuitpython_nrf24l01.rf24 import address_repr
>>> address_repr(b"1Node")
'65646F4E31'
```

Parameters

buf : bytes,bytearray
The buffer of bytes to convert into a hexlified string.

reverse : bool
A `bool` to control the resulting endianess. `True` outputs the result as big endian. `False` outputs the result as little endian. This parameter defaults to `True` since `bytearray` and `bytes` objects are stored in big endian but written in little endian.

delimit : str
A `chr` or `str` to use as a delimiter between bytes. Defaults to an empty string.

Returns
A string of hexadecimal characters in big endian form of the specified `buf` parameter.

Changed in version 2.1.0: Added parameters `reverse` and `delimit` as this function proved vital to debugging and developing *`RF24NetworkHeader`* & *`RF24NetworkFrame`*.

5.2 Status Byte

`RF24.tx_full`

An `bool` to represent if the TX FIFO is full. (read-only)

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

Returns

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

`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 has been reset.

Important: It is recommended that this flag is only used when the IRQ pin is active. To determine if there is a payload in the RX FIFO, use *`fifo()`*, *`any()`*, or *`pipe`*. Notice that calling *`read()`* also resets this status flag.

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

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

`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 has been reset.

Important: This can only return `True` if *`auto_ack`* is enabled, otherwise this will always be `False`.

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

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

`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 has been reset.

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

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

`RF24.update()` → typing_extensions.Literal[True]

This function gets an updated status byte over SPI.

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 *available()*, *send()*, and *resend()* functions.

Returns

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

Changed in version 1.2.3: Arbitrarily returns `True`.

`RF24.pipe`

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

Changed in version 1.2.0: In previous versions of this library, this attribute was a read-only function (`pipe()`).

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

Returns

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

`RF24.clear_status_flags(data_recv: bool = True, data_sent: bool = True, data_fail: bool = True)`

This clears the interrupt flags in the status register.

Internally, this is automatically called by *send()*, *write()*, *read()*.

Parameters

data_recv : bool
specifies whether to clear the "RX Data Ready" (*irq_dr*) flag.

data_sent : bool
specifies whether to clear the "TX Data Sent" (*irq_ds*) flag.

data_fail : bool
specifies whether to clear the "Max Re-transmit reached" (*irq_df*) flag.

Note: Clearing the `data_fail` flag is necessary for continued transmissions from the nRF24L01 (locks the TX FIFO buffer when *irq_df* is `True`) despite whether 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 status 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.

5.3 FIFO management

RF24.**flush_rx**()

Flush all 3 levels of the RX FIFO.

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 *read()*. This function clears all 3 levels.

RF24.**flush_tx**()

Flush all 3 levels of the TX FIFO.

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).

RF24.**fifo**(about_tx: *bool* = ***False***, check_empty: *bool* | *None* = ***None***)

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

Parameters

about_tx : bool

- True means the information returned is about the TX FIFO buffer.
- False means the information returned is about the RX FIFO buffer. This parameter defaults to False when not specified.

check_empty : bool

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

Returns

- A bool answer to the question:

  "Is the [TX/RX](about_tx) FIFO buffer [empty/full](check_empty)?

- If the check_empty parameter is not specified: an int in range [0, 2] for which:
  - 1 means the specified FIFO buffer is empty
  - 2 means the specified FIFO buffer is full
  - 0 means the specified FIFO buffer is neither full nor empty

5.4 Ambiguous Signal Detection

RF24.**rpd**

Returns True if signal was detected or False if not. (read-only)

The RPD (Received Power Detector) flag is triggered in the following cases:

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

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

New in version 1.2.0.

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* and setting *ard* to 250 microseconds
- changing the TX address to b"\xFF\xFF\xFF\xFF\xFF"
- loading a 32-byte payload (each byte is 0xFF) into the TX FIFO buffer

Finally the radio continuously behaves like using *resend()* to establish the constant carrier wave. If *is_plus_variant* is True, then none of these changes are needed nor applied.

New in version 1.2.0.

`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).

Hint: If the radio is a non-plus variant (*is_plus_variant* returns `False`), then use `The with statement` to re-establish the previous settings:

```
# let `nrf` be the instantiated RF24 object
with nrf:
    pass  # settings are now restored
```

New in version 1.2.0.

CHAPTER

SIX

CONFIGURABLE RF24 API

`RF24.ack`

Represents use of custom payloads as part of the ACK packet.

Use this attribute to set/check if the custom ACK payloads feature is enabled (`True`) or disabled (`False`). Default setting is `False`.

Note: This attribute differs from the *auto_ack* attribute because the *auto_ack* attribute enables or disables the use of automatic ACK *packets*. By default, ACK *packets* have no *payload*. This attribute enables or disables attaching payloads to the ACK packets.

See also:

Use *load_ack()* attach ACK payloads.

Use *read()*, *send()*, *resend()* to retrieve ACK payloads.

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.

`RF24.allow_ask_no_ack`

Allow or disable `ask_no_ack` parameter to *send()* & *write()*.

This attribute is enabled by default, and it only exists to provide support for the Si24R1. The designers of the Si24R1 (a cheap chinese clone of the nRF24L01) happened to clone a typo from the first version of the nRF24L01 specification sheet. Disable this attribute for the Si24R1.

`RF24.interrupt_config(data_recv: bool = True, data_sent: bool = True, data_fail: bool = True)`

Sets the configuration of the nRF24L01's IRQ pin. (write-only)

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

Parameters

data_recv : bool
If this is `True`, then IRQ pin goes active when new data is put into the RX FIFO buffer. Default setting is `True`

data_sent : bool
If this is `True`, then IRQ pin goes active when a payload from TX buffer is successfully transmit. Default setting is `True`

data_fail : bool
If this is `True`, then IRQ pin goes active when the maximum number of attempts to re-transmit the packet have been reached. If *auto_ack* attribute is disabled for pipe 0, then this IRQ event is not used. Default setting is `True`

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

Tip: Paraphrased from nRF24L01+ Specification Sheet:

The procedure for handling *irq_dr* IRQ should be:

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

`RF24.`**data_rate**

This `int` attribute specifies the RF data rate.

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 all variants of transceivers based on the nRF24L01. This library will assume that the transceiver being used does support 250 kbps, but there is no way to determine (via software) if that is actually the case. Please refer to your transceiver's manufacturer information to determine if 250 kbps is supposed to be supported.

Hint: You can perform a carrier wave test on 250 kbps to see if your transceiver hardware does support that data rate. See *start_carrier_wave()*, *stop_carrier_wave()*, and *rpd* to execute a hardware test.

Changed in version 2.2.0: Blindly allow configuring the radio for 250 kbps as support is marginally dependent on the hardware being used.

`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).

`RF24.crc`

This `int` attribute specifies the CRC checksum length in bytes.

CRC (cyclic redundancy checking) 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 will be clamped to range [0, 2]. Default is enabled using 2 bytes.

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

Changed in version 2.0.0: Invalid input values are clamped to proper range instead of throwing a `ValueError` exception.

`RF24.pa_level`

This `int` is the power amplifier level (in dBm).

Higher levels mean the transmission will cover a longer distance. Use this attribute to tweak the nRF24L01 current consumption on projects that don't span large areas.

A valid input value is:

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

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

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

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

`RF24.is_lna_enabled`

A read-only `bool` attribute about the LNA gain feature.

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

6.1 dynamic_payloads

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

`RF24.dynamic_payloads`

This `int` attribute is the dynamic payload length feature for any/all pipes.

Default setting is enabled on all pipes. A valid input is:

- A `bool` to enable (`True`) or disable (`False`) the dynamic payload length feature for all 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.
- An `int` where each bit in the integer represents the dynamic payload feature per pipe. Bit position 0 controls this feature for data pipe 0, and bit position 5 controls this feature for data pipe 5. All bits in positions greater than 5 are ignored.

Note:

- The *`payload_length`* attribute is ignored when this feature is enabled for any respective data pipes.
- Be sure to adjust the *`payload_length`* attribute accordingly when this feature is disabled for any respective data pipes.

Returns

An `int` (1 unsigned byte) where each bit in the integer represents the dynamic payload length feature per pipe.

Changed in version 1.2.0: Accepts a list or tuple for control of the dynamic payload length feature per pipe.

Changed in version 2.0.0:

- Returns a integer instead of a boolean
- Accepts an integer for binary control of the dynamic payload length feature per pipe

`RF24.set_dynamic_payloads(enable: bool, pipe_number: int | None = None)`

Control the dynamic payload feature for a specific data pipe.

Parameters

`enable` : bool
The state of the dynamic payload feature about a specified data pipe.

`pipe_number` : int
The specific data pipe number in range [0, 5] to apply the `enable` parameter. If this parameter is not specified the `enable` parameter is applied to all data pipes. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown.

New in version 2.0.0.

`RF24.get_dynamic_payloads(pipe_number: int = 0) → bool`

Returns a `bool` describing the dynamic payload feature about a pipe.

Parameters

pipe_number : int
The specific data pipe number in range [0, 5] concerning the dynamic payload length feature. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown. If this parameter is not specified, then the data returned is about data pipe 0.

6.2 payload_length

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

`RF24.payload_length`

This `int` attribute is the length of static payloads for any/all pipes.

This attribute can be used to specify the static payload length used for all data pipes in which the `dynamic_payloads` attribute is *disabled*

A valid input value must be:

- an `int` in which the value that will be clamped to the range [1, 32]. Setting this attribute to a single `int` configures all 6 data pipes.
- A `list` or `tuple` containing integers can be used control this feature per data pipe. Index 0 controls this feature on data pipe 0. Indices greater than 5 will be ignored since there are only 6 data pipes. If any index's value is less than or equal to``0``, then the existing setting for the corresponding data pipe will persist (not be changed).

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

Returns
The current setting of the expected static payload length feature for pipe 0 only.

Changed in version 1.2.0: Return a list of all payload length settings for all pipes. This implementation introduced a couple bugs:

1. The settings could be changed improperly in a way that was not written to the nRF24L01 registers.
2. There was no way to catch an invalid setting if configured improperly via the first bug. This led to errors in using other functions that handle payloads or the length of payloads.

Changed in version 2.0.0: This attribute returns the configuration about static payload length for data pipe 0 only. Use `get_payload_length()` to fetch the configuration of the static payload length feature for any data pipe.

`RF24.set_payload_length(length: int, pipe_number: int | None = None)`

Sets the static payload length feature for each/all data pipes.

This function only affects data pipes for which the `dynamic_payloads` attribute is *disabled*.

Parameters

length : int
The number of bytes in range [1, 32] for to be used for static payload lengths. If this number is not in range [1, 32], then it will be clamped to that range.

pipe_number : int
The specific data pipe number in range [0, 5] to apply the `length` parameter. If this parameter is not specified the `length` parameter is applied to all data pipes. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown.

New in version 2.0.0.

`RF24.get_payload_length(pipe_number: int = 0) → int`

Returns an `int` describing the specified data pipe's static payload length.

The data returned by this function is only relevant for data pipes in which the *dynamic_payloads* attribute is *disabled*.

Parameters

pipe_number : int
The specific data pipe number in range [0, 5] to concerning the static payload length feature. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown. If this parameter is not specified, then the data returned is about data pipe 0.

New in version 2.0.0.

6.3 auto_ack

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

- This attribute will intuitively disable the acknowledgement payload feature (*ack* attribute) when the automatic acknowledgement feature is disabled for data pipe 0.
- When entering in TX mode, the *listen* attribute will ensure data pipe 0 is open to receive automatic acknowledgments for outgoing transmissions.
- Be sure to configure this attribute for data pipe 0 before calling *open_tx_pipe()* because the RX address for pipe 0 needs to be overwritten for automatic acknowledgments to be received in TX mode. The *listen* attribute will re-write the RX address for data pipe 0 when entering RX mode if needed.

`RF24.auto_ack`

This `int` attribute is the automatic acknowledgment feature for any/all pipes.

Default setting is enabled on all data pipes. A valid input is:

- A `bool` to enable (`True`) or disable (`False`) transmitting automatic acknowledgment packets for all 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.
- An `int` where each bit in the integer represents the automatic acknowledgement feature per pipe. Bit position 0 controls this feature for data pipe 0, and bit position 5 controls this feature for data pipe 5. All bits in positions greater than 5 are ignored.

Note: The CRC (cyclic redundancy checking) is enabled (for all transmissions) automatically by the nRF24L01 if this attribute is enabled for any data pipe (see also *crc* attribute). The *crc* attribute will remain unaffected when disabling this attribute for any data pipes.

Returns
An `int` (1 unsigned byte) where each bit in the integer represents the automatic acknowledgement feature per pipe.

Changed in version 1.2.0: Accepts a list or tuple for control of the automatic acknowledgement feature per pipe.

Changed in version 2.0.0:

- Returns an integer instead of a boolean
- Accepts an integer for binary control of the automatic acknowledgement feature per pipe

`RF24.set_auto_ack(enable: bool, pipe_number: int)`

Control the *auto_ack* feature for a specific data pipe.

Parameters

enable : bool
The state of the automatic acknowledgement feature about a specified data pipe.

pipe_number : int
The specific data pipe number in range [0, 5] to apply the `enable` parameter. If this parameter is not specified the `enable` parameter is applied to all data pipes. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown.

New in version 2.0.0.

`RF24.get_auto_ack(pipe_number: int) → bool`

Returns a `bool` describing the *auto_ack* feature about a data pipe.

Parameters

pipe_number : int
The specific data pipe number in range [0, 5] concerning the setting for the automatic acknowledgment feature. If this parameter is not in range [0, 5], then a `IndexError` exception is thrown. If this parameter is not specified, then the data returned is about data pipe 0.

New in version 2.0.0.

6.4 Auto-Retry feature

`RF24.arc`

This `int` attribute specifies the number of attempts to re-transmit TX payload when ACK packet is not received.

The *auto_ack* attribute must be enabled on the receiving nRF24L01's pipe 0 & the RX data pipe and the transmitting nRF24L01's pipe 0 to properly use this attribute. If *auto_ack* is disabled on the transmitting nRF24L01's pipe 0, then this attribute is ignored when calling *send()*.

A valid input value will be clamped to range [0, 15]. Default is set to 15. A value of `0` disables the automatic re-transmit feature, but the sending nRF24L01 will still wait the number of microseconds specified by *ard* for an Acknowledgement (ACK) packet response (assuming *auto_ack* is enabled).

Changed in version 2.0.0: Invalid input values are clamped to proper range instead of throwing a `ValueError` exception.

Changed in version 2.2.0: Default value changed from 3 to the maximum 15. This only affects performance in scenarios that experience unreliable reception.

`RF24.ard`

This `int` attribute specifies the delay (in microseconds) between attempts to automatically re-transmit the TX payload when no ACK packet is received.

During this time, the nRF24L01 is listening for the ACK packet. If the *auto_ack* attribute is disabled for pipe 0, then this attribute is not applied.

A valid input value will be clamped to range [250, 4000]. 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.

Changed in version 2.0.0: Invalid input values are clamped to proper range instead of throwing a `ValueError` exception.

`RF24.set_auto_retries(delay: int, count: int)`

set the *ard* & *arc* attributes with 1 function.

Parameters

delay : int
accepts the same input as the *ard* attribute.

count : int
accepts the same input as the *arc* attribute.

`RF24.get_auto_retries() → tuple`

get the *ard* & *arc* attributes with 1 function.

Return
A tuple containing 2 items; index 0 will be the *ard* attribute, and index 1 will be the *arc* attribute.

CHAPTER

SEVEN

BLE API

New in version 1.2.0: BLE API added

7.1 BLE Limitations

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

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

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

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

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

   Use the helper function *len_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 *auto_ack* attribute is disabled.

7.2 fake_ble module helpers

circuitpython_nrf24l01.fake_ble.**swap_bits**(original: *int*) → int

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 is truncated to a single unsigned byte, meaning this parameter's value can only range from 0 to 255.

circuitpython_nrf24l01.fake_ble.**reverse_bits**(original: *bytes, bytearray*) → bytearray

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.

circuitpython_nrf24l01.fake_ble.**chunk**(buf: *bytes, bytearray*, data_type: *int* = *22*) → bytearray

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*.

---

circuitpython_nrf24l01.fake_ble.**crc24_ble**(data: *bytes, bytearray*, deg_poly: *int* = *1627*, init_val: *int* = *5592405*) → bytearray

This function calculates a checksum of various sized buffers.

This is exposed for convenience and 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).

`circuitpython_nrf24l01.fake_ble.whitener(buf: bytes, bytearray, coef: int) → bytearray`

Whiten and de-whiten data according to the given coefficient.

This is a helper function to *`FakeBLE.whiten()`*. It has been broken out of the *`FakeBLE`* class to allow whitening and dewhitening a BLE payload without the hardcoded coefficient.

Parameters

buf : bytes,bytearray
The BLE payloads data. This data should include the CRC24 checksum.

coef : int

The whitening coefficient used to avoid repeating binary patterns. This is the index of *`BLE_FREQ`* tuple for nRF24L01 channel that the payload transits (plus 37).

```
coef = None  # placeholder for the coefficient
rx_channel = nrf.channel
for index, chl in enumerate(BLE_FREQ):
    if chl == rx_channel:
        coef = index + 37
        break
```

Note: If currently used nRF24L01 channel is different from the channel in which the payload was received, then set this parameter accordingly.

`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:

nRF24L01 channel	BLE channel
2	37
26	38
80	39

7.3 QueueElement class

New in version 2.1.0: This class was added when implementing BLE signal sniffing.

class circuitpython_nrf24l01.fake_ble.**QueueElement**(buffer: *bytearray*)

A data structure used for storing received & decoded BLE payloads in the *rx_queue*.

Parameters

buffer : bytes,bytearray
the validated BLE payload (not including the CRC checksum). The buffer passed here is decoded into this class's properties.

mac

The transmitting BLE device's MAC address as a `bytes` object.

name

The transmitting BLE device's name. This will be a `str`, `bytes` object (if a `UnicodeError` was caught), or `None` (if not included in the received payload).

pa_level

The transmitting device's PA Level (if included in the received payload) as an `int`.

Note: This value does not represent the received signal strength. The nRF24L01 will receive anything over a -64 dbm threshold.

data : list[bytearray, 'ServiceData']

A `list` of the transmitting device's data structures (if any). If an element in this `list` is not an instance (or descendant) of the *ServiceData* class, then it is likely a custom, user-defined, or unsupported specification - in which case it will be a `bytearray` object.

7.4 FakeBLE class

class circuitpython_nrf24l01.fake_ble.**FakeBLE**(*spi: busio.SPI*, *csn: digitalio.DigitalInOut*, *ce_pin: digitalio.DigitalInOut*, *spi_frequency: int = 10000000*)

Bases: *RF24*

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 *Unavailable RF24 functionality* for more details.

See also:

For all parameters' descriptions, see the *RF24* class' constructor documentation.

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.

`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.

Valid inputs are `str`, `bytes`, `bytearray`, or `None`. A `str` will be converted to a `bytes` object automatically.

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

Changed in version 2.2.0: This attribute can also be set with a `str`, but it must be UTF-8 compatible.

`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.

`FakeBLE.channel`

This `int` attribute specifies the nRF24L01's frequency.

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

Changed in version 2.1.0: Any invalid input value (that is not found in *`BLE_FREQ`*) had raised a `ValueError` exception. This behavior changed to ignoring invalid input values, and the exception is no longer raised.

`FakeBLE.hop_channel()`

Trigger an automatic change of BLE compliant channels.

`FakeBLE.whiten(data: bytes, bytearray) → bytearray`

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.

Note: *`advertise()`* and *`available()`* uses this function internally to prevent improper usage.

Warning: This function uses the currently set BLE channel as a base case for the whitening coefficient.

Do not call *`hop_channel()`* before calling *`available()`* because this function needs to know the correct BLE channel to properly de-whiten received payloads.

`FakeBLE.len_available(hypothetical: bytes, bytearray = b'') → int`

This function will calculates 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 a single payload.

Changed in version 2.0.0: The name of this function changed from "available" to "len_available" to avoid confusion with *circuitpython_nrf24l01.rf24.RF24.available()*. This change also allows providing the underlying *RF24* class' *available()* method in the *FakeBLE* API.

`FakeBLE.advertise(buf: bytes, bytearray = b'', data_type: int = 255)`

This blocking function is used to broadcast a payload.

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

Parameters

buf : bytearray
The payload to transmit. This bytearray must have a length greater than 0 and less than 22 bytes Otherwise a `ValueError` exception is thrown whose prompt will tell you the maximum length allowed under the current configuration. This can also be a list or tuple of payloads (`bytearray`); in which case, all items in the list/tuple are processed are packed into 1 payload for a single transmissions. See example code below about passing a `list` or `tuple` to this parameter.

data_type : int
This is used to describe the buffer data passed to the `buf` parameter. `0x16` describes all service data. The default value `0xFF` describes manufacturer information. This parameter is ignored when a `tuple` or `list` is passed to the `buf` parameter. Any other values are not applicable to BLE advertisements.

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

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

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

# make a tuple of the buffers
```

(continues on next page)

(continued from previous page)

```
buffers = (
    chunk(service1.buffer),
    chunk(service2.buffer)
)

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

FakeBLE.**available**() → bool

A `bool` describing if there is a payload in the *rx_queue*.

This method will take the first available data from the radio's RX FIFO and validate the payload using the 24bit CRC checksum at the end of the payload. If the payload is indeed a valid BLE transmission that fit within the 32 bytes that the nRF24L01 can capture, then this method will decipher the data within the payload and enqueue the resulting *QueueElement* in the *rx_queue*.

Tip: Use *read()* to fetch the decoded data.

Returns

- `True` if payload was received *and* validated
- `False` if no payload was received or the received payload could not be deciphered.

Changed in version 2.1.0: This was an added override to validate & decipher received BLE data.

FakeBLE.**rx_queue**

The internal queue of received BLE payloads' data.

Each Element in this queue is a *QueueElement* object whose members are set according to the its internal decoding algorithm. The *read()* function will remove & return the first element in this queue.

Hint: This attribute is exposed for debugging purposes, but it can also be used by applications.

New in version 2.1.0.

FakeBLE.**rx_cache**

The internal cache used when validating received BLE payloads.

This attribute is only used by *available()* to cache the data from the top level of the radio's RX FIFO then validate & decode it.

Hint: This attribute is exposed for debugging purposes.

New in version 2.1.0.

FakeBLE.**read**() → *QueueElement*

Get the First Out element from the queue.

Returns

- `None` if nothing is the internal *rx_queue*

- A *QueueElement* object from the front of the *rx_queue* (like a FIFO buffer)

Changed in version 2.1.0: This was an added override to fetch deciphered BLE data from the *rx_queue*.

FakeBLE.**interrupt_config**(*data_recv: bool* = **True**, *data_sent: bool* = **True**, *data_fail: bool* = **True**)

Sets the configuration of the nRF24L01's IRQ pin. (write-only)

> **Warning:** The *irq_df* attribute is not implemented for BLE operations.

See also:

interrupt_config()

7.4.1 Unavailable RF24 functionality

The following *RF24* functionality is not available in *FakeBLE* objects:

- *dynamic_payloads*
- *set_dynamic_payloads()*
- *data_rate*
- *address_length*
- *auto_ack*
- *set_auto_ack()*
- *ack*
- *crc*
- *open_rx_pipe()*
- *open_tx_pipe()*

7.5 Service related classes

7.5.1 Abstract Parent

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

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.

property uuid : bytes

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

property data : bytes, bytearray

This attribute is a `bytearray` or `bytes` object.

property buffer : bytes

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

__len__() → int

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.

__repr__() → str

For convenience, this class is compatible with python's builtin `repr()` method. In this context, it will return a string of data with applicable suffixed units.

7.5.2 Service data UUID numbers

These are the 16-bit UUID numbers used by the Derivative Children of the ServiceData class

circuitpython_nrf24l01.fake_ble.**TEMPERATURE_UUID = 0x1809**

The Temperature Service UUID number

circuitpython_nrf24l01.fake_ble.**BATTERY_UUID = 0x180F**

The Battery Service UUID number

circuitpython_nrf24l01.fake_ble.**EDDYSTONE_UUID = 0xFEAA**

The Eddystone Service UUID number

7.5.3 Derivative Children

class circuitpython_nrf24l01.fake_ble.**TemperatureServiceData**

Bases: *ServiceData*

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

See also:

Bluetooth Health Thermometer Measurement format as defined in the GATT Specifications Supplement.

property data : float

This attribute is a `float` value.

class circuitpython_nrf24l01.fake_ble.**BatteryServiceData**

Bases: *ServiceData*

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

See also:

The Bluetooth Battery Level format as defined in the GATT Specifications Supplement.

property data : int

The attribute is a 1-byte unsigned `int` value.

class circuitpython_nrf24l01.fake_ble.**UrlServiceData**

Bases: *ServiceData*

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

See also:

Google's Eddystone-URL specifications.

property pa_level_at_1_meter : int

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`.

property data : str

This attribute is a `str` of URL data.

CHAPTER

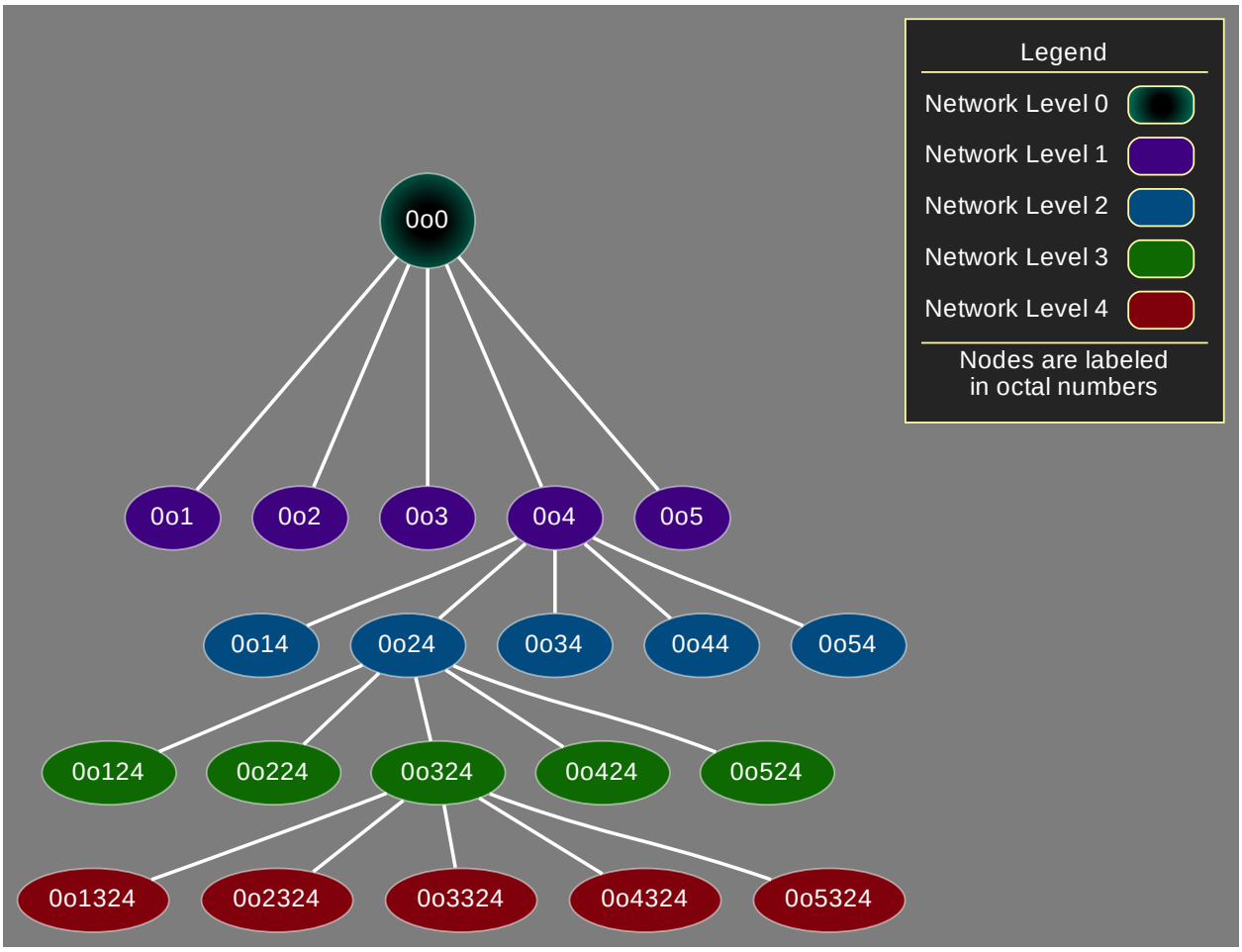
EIGHT

NETWORK TOPOLOGY

8.1 Network Levels

Because of the hardware limitation's of the nRF24L01 transceiver, each network is arranged in a levels where a parent can have up to 5 children. And each child can also have up to 5 other children. This is not limitless because this network is designed for low-memory devices. Consequently, all node's *Logical Address* are limited to 12-bit integers and use an octal counting scheme.

- The master node (designated with the *Logical Address* `00`) is always the only node in the lowest level (denoted as level 0).
- Child nodes are designated by the most significant octal digit in their *Logical Address*. A child node address' least significant digits are the inherited address of it's parent node. Nodes on level 1 only have 1 digit because they are children of the master node.

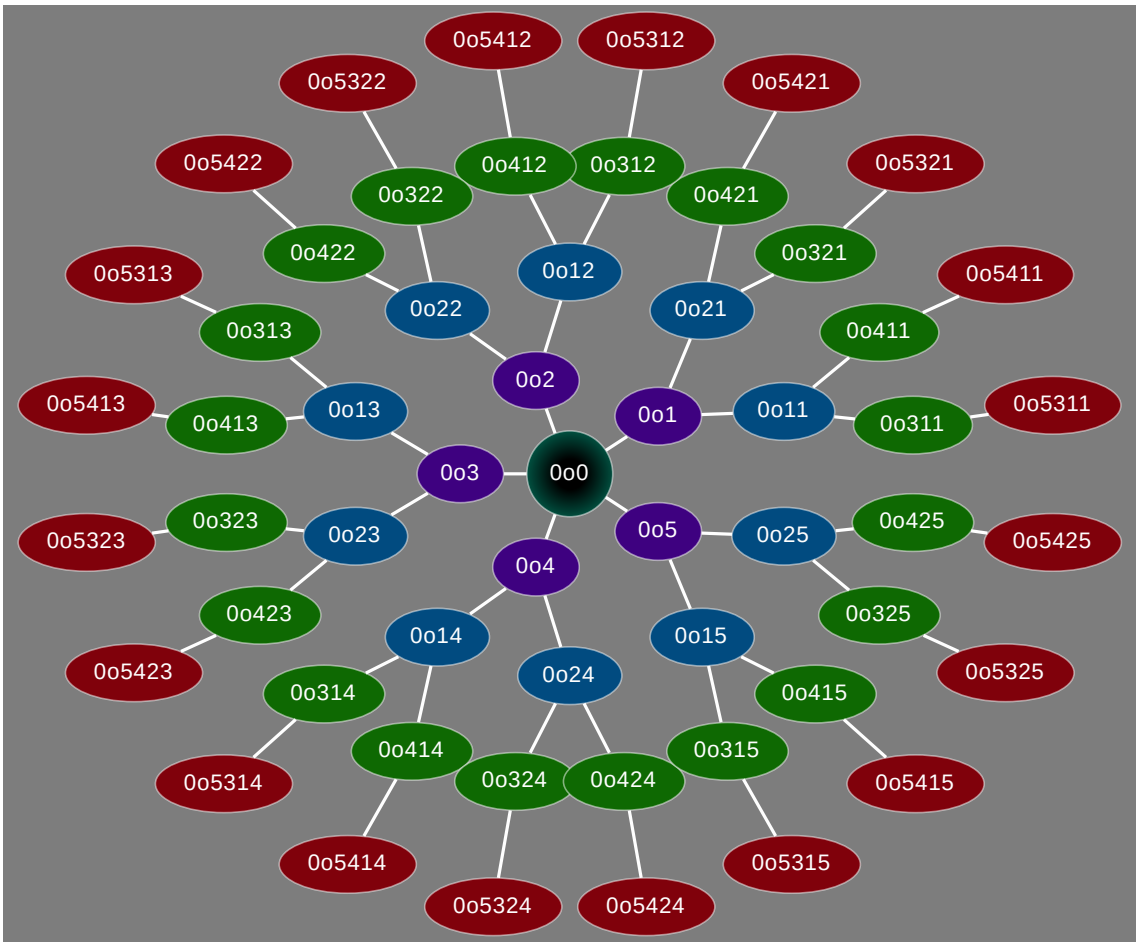


Hopefully, you should see the pattern. There can be up to a maximum of 5 network levels (that's 0-4 ordered from lowest to highest).

For a message to travel from node `0o124` to node `0o3`, it must be passed through any applicable network levels. So, the message flows `0o124` -> `0o24` -> `0o4` -> `0o0` -> `0o3`.

A single network can potentially have a maximum of 781 nodes (all operating on the same *channel*), but for readability reasons, the following graph only demonstrates

- the master node (level 0) and it's 5 children (level 1)
- level 2 only shows the 1st and 2nd children of parents on level 1
- level 3 only shows the 3rd and 4th children of parents on level 2
- level 4 only shows the 5th children of parents on level 3



8.2 Physical addresses vs Logical addresses

- The Physical address is the 5-byte address assigned to the radio's data pipes.
- The Logical address is the 12-bit integer representing a network node. The Logical address uses an octal counting scheme. A valid Logical Address must only contain octal digits in range [1, 5]. The master node is the exception for it uses the number `0`

Tip: Use the `is_address_valid()` function to programmatically check a Logical Address for validity.

Note: Remember that the nRF24L01 only has 6 data pipes for which to receive or transmit. Since only data pipe 0 can be used to transmit, the other other data pipes 1-5 are devoted to receiving transmissions from other network nodes; data pipe 0 also receives multicasted messages about the node's network level).

8.2.1 Translating Logical to Physical

Before translating the Logical address, a single byte is used repetitively as the base case for all bytes of any Physical Address. This byte is the *address_prefix* attribute (stored as a mutable `bytearray`) in the *RF24Network* class. By default the *address_prefix* has a single byte value of `b"\xCC"`.

The *RF24Network* class also has a predefined list of bytes used for translating unique Logical addresses into unique Physical addresses. This list is called *address_suffix* (also stored as a mutable `bytearray`). By default the *address_suffix* has 6-byte value of `b"\xC3\x3C\x33\xCE\x3E\xE3"` where the order of bytes pertains to the data pipe number and child node's most significant byte in its Physical Address.

For example:

The Logical Address of the network's master node is `0`. The radio's pipes 1-5 start with the *address_prefix*. To make each pipe's Physical address unique to a child node's Physical address, the *address_suffix* is used.

The Logical address of the master node: `0o0`

pipe	Physical Address (hexadecimal)
1	`CC CC CC CC 3C`
2	`CC CC CC CC 33`
3	`CC CC CC CC CE`
4	`CC CC CC CC 3E`
5	`CC CC CC CC E3`

The Logical address of the master node's first child: `0o1`

pipe	Physical Address (hexadecimal)
1	`CC CC CC 3C 3C`
2	`CC CC CC 3C 33`
3	`CC CC CC 3C CE`
4	`CC CC CC 3C 3E`
5	`CC CC CC 3C E3`

The Logical address of the master node's second child: `0o2`

pipe	Physical Address (hexadecimal)
1	`CC CC CC 33 3C`
2	`CC CC CC 33 33`
3	`CC CC CC 33 CE`
4	`CC CC CC 33 3E`
5	`CC CC CC 33 E3`

The Logical address of the master node's third child's second child's first child: `0o123`

pipe	Physical Address (hexadecimal)
1	`CC 3C 33 CE 3C`
2	`CC 3C 33 CE 33`
3	`CC 3C 33 CE CE`
4	`CC 3C 33 CE 3E`
5	`CC 3C 33 CE E3`

8.2.2 Two networks coexisting on the same channel

Warning: The following section is an advanced tutorial. The default values for *address_prefix* and *address_suffix* were carefully chosen by TMRh20 to demonstrate best practices in terms of choosing a data pipe's address for transmissions. Bad practices can be avoided by heeding ManiacBug's advice in his detailed blog post about the topic.

In theory, the *address_prefix* and *address_suffix* attributes could be changed to allow 2 separate networks to coexist on the same *channel*. The following are example code snippets to use as a template for such a scenario.

Listing 1: Master node for `network_a`

```
from circuitpython_nrf24l01.rf24_network import RF24Network

# ... declare SPI_BUS, CE_PIN, and CSN_PIN objects
network_a_master = RF24Network(SPI_BUS, CSN_PIN, CE_PIN, 0)

# let network_a use the default values for address_prefix and address_suffix

while True:
    network_a_master.update()
    if network_a_master.available():
        recv_frame = network_a_master.read()
        print(
            "received {}: {}".format(
                recv_frame.header.to_string(), recv_frame.message.decode()
            )
        )
    # emit frames as needed
```

Listing 2: Master node for `network_b`

```
from circuitpython_nrf24l01.rf24_network import RF24Network

# ... declare SPI_BUS, CE_PIN, and CSN_PIN objects
network_b_master = RF24Network(SPI_BUS, CSN_PIN, CE_PIN, 0)

# let network_b use different values for address_prefix and address_suffix
network_b_master.address_prefix = bytearray([0xDB])
network_b_master.address_suffix = bytearray([0xDD, 0x99, 0xB6, 0xD9, 0x9D, 0x66])

# re-assign the node_address for the different physical addresses to be used
network_b_master.node_address = 0

while True:
    network_b_master.update()
    if network_b_master.available():
        recv_frame = network_b_master.read()
        print(
            "received {}: {}".format(
                recv_frame.header.to_string(), recv_frame.message.decode()
```

(continues on next page)

(continued from previous page)

```
            )
        )
    # emit frames as needed
```

Listing 3: A single network node for hoping between network_a & network_b

```
from circuitpython_nrf24l01.rf24_network import RF24Network

# ... declare SPI_BUS, CE_PIN, and CSN_PIN objects
network_b_node = RF24Network(SPI_BUS, CSN_PIN, CE_PIN, 5)
network_a_node = RF24Network(SPI_BUS, CSN_PIN, CE_PIN, 1)

# let network_b use different values for address_prefix and address_suffix
with network_b_node as net_b:
    net_b.address_prefix = bytearray([0xDB])
    net_b.address_suffix = bytearray([0xDD, 0x99, 0xB6, 0xD9, 0x9D, 0x66])

    # re-assign the node_address for the different physical addresses to be used
    net_b.node_address = 5

while True:
    # do something with network_a
    with network_a_node as net_a:
        net_a.update()
        net_a.send(RF24NetworkHeader(0, "T"), b"data for net A master")

    # do something with network_b
    with network_b_node as net_b:
        net_b.update()
        net_b.send(RF24NetworkHeader(0, "T"), b"data for net B master")
```

8.3 RF24Mesh connecting process

As noted above, a single network *can* have up to 781 nodes. This number also includes up to 255 RF24Mesh nodes. The key difference from the user's perspective is that RF24Mesh API does not use a *Logical Address*. Instead the RF24Mesh API relies on a `node_id` number to identify a RF24Mesh node that may use a different *Logical Address* (which can change based on the node's physical location).

Important: Any network that will use RF24mesh for a child node needs to have a RF24Mesh master node. This will not interfere with RF24Network nodes since the RF24Mesh API is layered on top of the RF24Network API.

To better explain the difference between a node's `node_address` vs a node's `node_id`, we will examine the connecting process for a RF24Mesh node. These are the steps performed when calling `renew_address()`:

1. Any RF24Mesh node not connected to a network will use the *Logical Address* `0o444` (that's `2340` in decimal). It is up to the network administrator to ensure that each RF24Mesh node has a unique `node_id` (which is limited to the range [0, 255]).

Hint: Remember that `0` is reserved the master node's `node_id`.

2. To get assigned a *Logical Address*, an unconnected node must poll the network for a response (using a `NETWORK_POLL` message). Initially this happens on the network level 0, but consecutive attempts will poll higher network levels (in order of low to high) if this process fails.
3. When a polling transmission is responded, the connecting mesh node sends an address request which gets forwarded to the master node when necessary (using a `MESH_ADDR_REQUEST` message).
4. The master node will process the address request and respond with a `node_address` (using a `MESH_ADDR_RESPONSE` message). If there is no available occupancy on the network level from which the address request originated, then the master node will respond with an invalid *Logical Address*.
5. Once the requesting node receives the address response (and the assigned address is valid), it assumes that as the `node_address` while maintaining its `node_id`.
   - The connecting node will verify its new address by calling `check_connection`.
   - If the assigned address is invalid or `check_connection()` returns `False`, then the connecting node will re-start the process (step 1) on a different network level.

8.3.1 Points of failure

This process happens over a span of a few milliseconds. However,

- If the connecting node is physically moving throughout the network very quickly, then this process will take longer and is likely to fail.
- If a master node is able to respond faster than the connecting node can prepare itself to receive, then the process will fail entirely. This failure about faster master nodes often results in some slower RF24Mesh nodes only being able to connect to the network through another non-master node.

If you run into trouble with this connection process, then please open an issue on github and describe the situation with as much detail as possible.

CHAPTER

NINE

NETWORK DATA STRUCTURES

New in version 2.1.0.

These classes are used to structure the payload data for wireless network transactions.

9.1 Header

class circuitpython_nrf24l01.network.structs.**RF24NetworkHeader**(to_node: *int* | *None* = ***None***, message_type: *str*, *int* | *None* = ***None***)

The header information used for routing network messages.

Parameters

to_node : int
The *Logical Address* designating the message's destination.

message_type : int,str
A 1-byte `int` representing the `message_type`. If a `str` is passed, then the first character's numeric ASCII representation is used.

Note: These parameters can be left unspecified to create a blank object that can be augmented after instantiation.

RF24NetworkHeader.**to_node**

This value is truncated to a 2-byte unsigned `int`.

Describes the message destination using a *Logical Address*.

RF24NetworkHeader.**from_node**

This value is truncated to a 2-byte unsigned `int`.

Describes the message origin using a *Logical Address*.

RF24NetworkHeader.**message_type**

The type of message.

This `int` must be less than 256. When set using a `str`, this attribute's `int` value is derived from the ASCII number of the string's first character (see `ord()`). Non-ASCII characters' values are truncated to 1 byte (see `str.isascii()`). A blank `str` sets this attribute's value to `0`.

Hint: Users are encouraged to specify a number in range [0, 127] (basically less than or equal to *MAX_USR_DEF_MSG_TYPE*) as there are Reserved Message Types.

`RF24NetworkHeader.frame_id`

This value is truncated to a 2-byte unsigned `int`.

The sequential identifying number for the frame (relative to the originating network node). Each sequential frame's ID is incremented, but frames containing fragmented messages have the same ID number.

`RF24NetworkHeader.reserved`

A single byte reserved for network usage.

This will be the sequential ID number for fragmented messages, but on the last message fragment, this will be the *message_type*. *RF24Mesh* will also use this attribute to hold a newly assigned network *Logical Address* for *MESH_ADDR_RESPONSE* messages.

`RF24NetworkHeader.unpack(buffer) → bool`

Decode header data from the first 8 bytes of a frame's buffer.

This function is meant for library internal usage.

Parameters

buffer : bytes,bytearray
The buffer to unpack. All resulting data is stored in the objects attributes accordingly.

Returns
`True` if successful; otherwise `False`.

`RF24NetworkHeader.pack() → bytes`

This function is meant for library internal usage.

Returns
The entire header as a `bytes` object.

`RF24NetworkHeader.to_string() → str`

Returns
A `str` describing all of the header's attributes.

9.2 Frame

`class circuitpython_nrf24l01.network.structs.RF24NetworkFrame(header: RF24NetworkHeader | None = None, message: bytes, bytearray | None = None)`

Structure of a single frame.

This is used for either a single fragment of an individually large message (greater than 24 bytes) or a single message that is less than 25 bytes.

Parameters

header : *RF24NetworkHeader*
The header describing the frame's *message*.

message : bytes,bytearray
The actual *message* containing the payload or a fragment of a payload.

Note: These parameters can be left unspecified to create a blank object that can be augmented after instantiation.

`RF24NetworkFrame.header`

The *RF24NetworkHeader* about the frame's *message*.

`RF24NetworkFrame.message`

The entire message or a fragment of a message allocated to the frame.

This attribute is typically a `bytearray` or `bytes` object.

`RF24NetworkFrame.unpack(buffer: bytes, bytearray) → bool`

Decode the *header* & *message* from a `buffer`.

This function is meant for library internal usage.

Parameters

buffer : bytes,bytearray
The buffer to unpack. All resulting data is stored in the objects attributes accordingly.

Returns
`True` if successful; otherwise `False`.

`RF24NetworkFrame.pack() → bytes`

This attribute is meant for library internal usage.

Returns
The entire object as a `bytes` object.

`RF24NetworkFrame.is_ack_type() → bool`

Check if the frame is to expect a *NETWORK_ACK* message.

This function is meant for library internal usage.

9.3 FrameQueue

`class circuitpython_nrf24l01.network.structs.FrameQueue(queue=None)`

A class that wraps a `list` with RF24Network Queue behavior.

Parameters

queue : *FrameQueue*,*FrameQueueFrag*
To move (not copy) the contents of another *FrameQueue* based object, you can pass the object to this parameter. Doing so will also copy the object's *max_queue_size* attribute.

`FrameQueue.max_queue_size`

The maximum number of frames that can be enqueued at once. Defaults to 6.

`FrameQueue.enqueue(frame: RF24NetworkFrame) → bool`

Add a *RF24NetworkFrame* to the queue.

Returns
`True` if the frame was added to the queue, or `False` if it was not.

`FrameQueue.dequeue()` → *RF24NetworkFrame*

Returns
The First Out element and removes it from the queue.

`FrameQueue.peek()` → *RF24NetworkFrame*

Returns
The First Out element without removing it from the queue.

`FrameQueue.__len__()` → int

Returns
The number of the enqueued frames.

For use with Python's builtin `len()`.

9.4 FrameQueueFrag

class `circuitpython_nrf24l01.network.structs.FrameQueueFrag(queue=None)`

Bases: *FrameQueue*

A specialized *FrameQueue* with an additional cache for fragmented frames.

Note: This class will only cache 1 fragmented message at a time. If parts of the fragmented message are missing (or duplicate fragments are received), then the fragment is discarded. If a new fragmented message is received (before a previous fragmented message is completed and reassembled), then the cache is reused for the new fragmented message to avoid memory leaks.

9.5 Logical Address Validation

`circuitpython_nrf24l01.network.structs.is_address_valid(address)` → bool

Test if a given address is a valid *Logical Address*.

Parameters

address : int
The *Logical Address* to validate.

Returns
`True` if the given address can be used as a *node_address* or *to_node* destination. Otherwise, this function returns `False`.

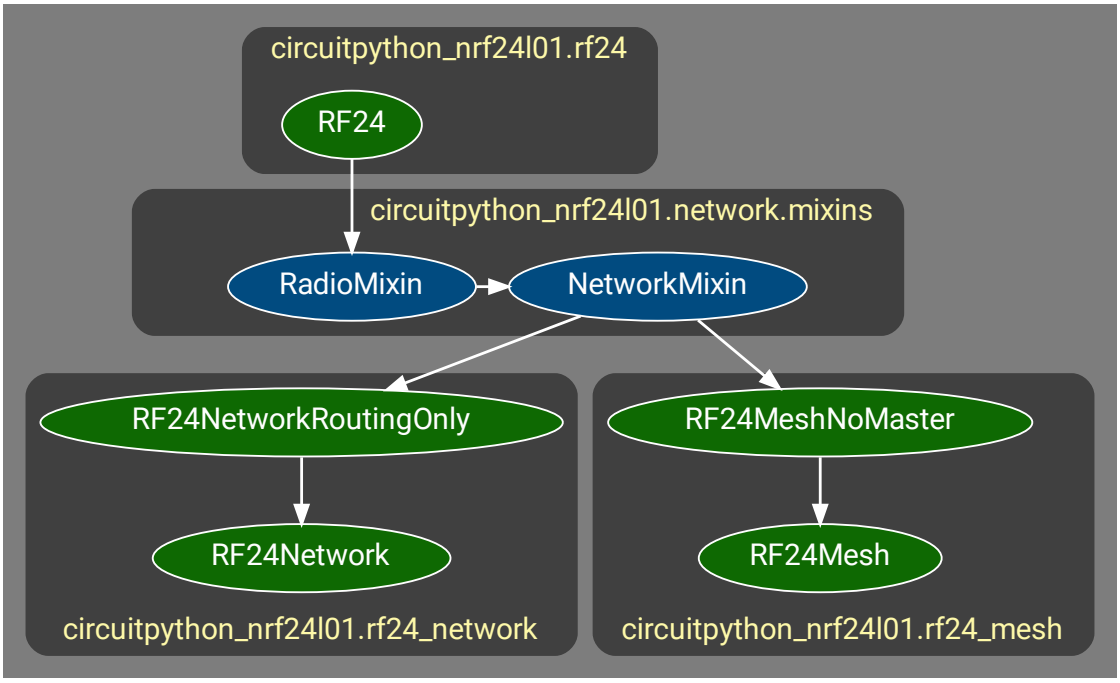
Warning: Please note that this function also allows the value `0o100` to validate because it is used as the *NETWORK_MULTICAST_ADDR* for multicasted messages. Technically, `0o100` is an invalid address.

CHAPTER

TEN

SHARED NETWORKING API

10.1 Order of Inheritance



The `RadioMixin` and `NetworkMixin` classes are not documented directly. Instead, this documentation follows the OSI (Open Systems Interconnection) model. This is done to mimic how the TMRh20 C++ libraries and documentation are structured.

Consequentially, all functions and members inherited from the `NetworkMixin` class are documented here as part of the *`RF24Network`* class. Note that the *`RF24MeshNoMaster`*, *`RF24Mesh`*, and *`RF24NetworkRoutingOnly`* classes all share the same API inherited from the `NetworkMixin` class.

10.2 Accessible RF24 API

The purpose of the `RadioMixin` class is

1. to provide a networking layer its own instantiated *RF24* object
2. to prevent applications from changing the radio's configuration in a way that breaks the networking layer's behavior

The following list of *RF24* functions and attributes are exposed in the RF24Network API and RF24Mesh API.

- *channel*
- *flush_rx()*
- *flush_tx()*
- *fifo()*
- *power*
- *set_dynamic_payloads()*
- *get_dynamic_payloads()*
- *listen*
- *pa_level*
- *is_lna_enabled*
- *data_rate*
- *crc*
- *set_auto_retries()*
- *get_auto_retries()*
- *last_tx_arc*
- *address()*
- *interrupt_config()*
- *print_pipes()*
- *print_details()*

  For the `print_details()` function, an additional keyword parameter named `network_only` can be used to filter out all the core details from the *RF24* object. The `dump_pipes` parameter still exists and defaults to `False`. Usage is as follows:

```
>>> # the following command is the same as `nrf.print_details(0, 1)`
>>> nrf.print_details(dump_pipes=False, network_only=True)
Network frame_buf contents:
    Header is from 0o7777 to 0o0 type 0 id 1 reserved 0. Message contains:
        an empty buffer
Return on system messages__False
Allow network multicasts___True
Multicast relay____________Disabled
Network fragmentation______Enabled
Network max message length_144 bytes
Network TX timeout_________25 milliseconds
```

(continues on next page)

(continued from previous page)

```
Network Routing timeout___75 milliseconds
Network node address_______0o0
```

Note: The address `0o7777` (seen in output above) is an invalid address used as a sentinel when the frame is unpopulated with a proper *from_node* address.

10.3 External Systems API

The following attributes are exposed in the *RF24Network* and *RF24Mesh* API for extensibility via external applications or systems.

`RF24Network.`**`address_prefix = b"\xCC"`**

The base case for all pipes' address' bytes before mutating with *address_suffix*.

See also:

The usage of this attribute is more explained in the Topology page

`RF24Network.`**`address_suffix = b"\xC3\x3C\x33\xCE\x3E\xE3"`**

Each byte in this `bytearray` corresponds to the unique byte per pipe and child node.

See also:

The usage of this attribute is more explained in the Topology page

`RF24Network.`**`frame_buf`**

A buffer containing the last frame handled by the network node

`RF24Network.`**`queue`**

The queue (FIFO) of received frames for this node

This attribute will be an instantiated *FrameQueue* or *FrameQueueFrag* object depending on the state of the *fragmentation* attribute.

`RF24Network.`**`ret_sys_msg`**

Force *update()* to return on system message types.

This `bool` attribute is asserted on mesh network nodes.

CHAPTER

ELEVEN

RF24NETWORK API

New in version 2.1.0.

See also:

Documentation for:

1. Network Topology
2. Shared Networking API
3. Network Data Structures
4. Network Constants

11.1 RF24NetworkRoutingOnly class

class circuitpython_nrf24l01.rf24_network.**RF24NetworkRoutingOnly**(*spi: busio.SPI*, *csn_pin: digitalio.DigitalInOut*, *ce_pin: digitalio.DigitalInOut*, node_address*: int*, *spi_frequency=10000000*)

A minimal Networking implementation for nodes that are meant for strictly routing data amidst a network of nodes.

This class is a minimal variant of the *RF24Network* class. The API is almost identical to *RF24Network* except that it has no *RF24Network.write()* or *RF24Network.send()* functions. This is meant to be the python equivalent to TMRh20's DISABLE_USER_PAYLOADS macro in the C++ RF24Network library.

Parameters

node_address : int
The octal int for this node's *Logical Address*

See also:

For all other parameters' descriptions, see the *RF24* class' constructor documentation.

11.2 RF24Network class

class circuitpython_nrf24l01.rf24_network.**RF24Network**(*spi: busio.SPI*, *csn_pin: digitalio.DigitalInOut*, *ce_pin: digitalio.DigitalInOut*, node_address*: int*, *spi_frequency=10000000*)

Bases: *RF24NetworkRoutingOnly*

The object used to instantiate the nRF24L01 as a network node.

Parameters

node_address : int
The octal `int` for this node's *Logical Address*

See also:

For all other parameters' descriptions, see the *RF24* class' constructor documentation.

11.3 Basic API

RF24Network.**node_address**

get/set the node's *Logical Address* for the *RF24Network* object.

Setting this attribute will alter

1. The *Physical Addresses* used on the radio's data pipes
2. The *parent* attribute
3. The *multicast_level* attribute's default value.

Warning:

1. If this attribute is set to an invalid network *Logical Address*, then nothing is done and the invalid address is ignored.
2. A *RF24Mesh* object cannot set this attribute because the *Logical Address* is assigned by the mesh network's master node. Therefore, this attribute is read-only for *RF24Mesh* objects.

   See also:

   Please review the tip documented in *RF24Mesh.node_id* for more details.

RF24Network.**update**() → int

This function is used to keep the network layer current.

Important: It is imperative that this function be called at least once during the application's main loop. For applications that perform long operations on each iteration of its main loop, it is encouraged to call this function more than once when possible.

Returns
The latest received message's *message_type*. The returned value is not gotten from frame's in the *queue*, but rather it is only gotten from the messages handled during the function's operation.

`RF24Network.available()` → bool

Returns
A `bool` describing if there is a frame waiting in the *queue*.

`RF24Network.peek()` → *RF24NetworkFrame*

Get (from *queue*) the next available frame.

Returns
A *RF24NetworkFrame* object. However, the data returned is not removed from the *queue*. If there is nothing in the *queue*, this method will return `None`.

`RF24Network.read()` → *RF24NetworkFrame*

Get (from *queue*) the next available frame.

This function differs from *peek()* because this function also removes the header & message from the *queue*.

Returns
A *RF24NetworkFrame* object. If there is nothing in the *queue*, this method will return `None`.

`RF24Network.send(header: RF24NetworkHeader, message: bytes, bytearray)` → bool

Deliver a message according to the header information.

Parameters

header : *RF24NetworkHeader*
The outgoing frame's *header*. It is important to have the header's *to_node* attribute set to the target network node's *Logical Address*.

message : bytes,bytearray

The outgoing frame's *message*.

Note: Be mindful of the message's size as this cannot exceed *MAX_FRAG_SIZE* (24 bytes) if *fragmentation* is disabled. If *fragmentation* is enabled (it is by default), then the message's size must be less than *max_message_length*

Returns
A `bool` describing if the message has been transmitted. This does not necessarily describe if the message has been received at its target destination.

Tip: To ensure a message has been delivered to its target destination, set the frame's header's *message_type* to an `int` in range [65, 127]. This will invoke a *NETWORK_ACK* response message.

11.4 Advanced API

`RF24Network.multicast(message: bytes, bytearray, message_type: str, int, level: int | None = None)` → bool

Broadcast a message to all nodes on a certain network level.

Parameters

message : bytes,bytearray
The outgoing frame's *message*.

message_type : str,int
The outgoing frame's *message_type*.

level : int
The network level of nodes to broadcast to. If this optional parameter is not specified, then the node's *multicast_level* is used.

See also:

multicast_level, *multicast_relay*, and *allow_multicast*

Returns
A bool describing if the message has been transmitted. This does not necessarily describe if the message has been received at its target destination.

Note: For multicasted messages, the radio's *auto_ack* feature is not used.

This function will always return True if a message is directed to a node's pipe that does not have *auto_ack* enabled (which will likely be pipe 0 in most network contexts).

Tip: To ensure a message has been delivered to its target destination, set the header's *message_type* to an int in range [65, 127]. This will invoke a *NETWORK_ACK* response message.

RF24Network.**write**(frame: RF24NetworkFrame, traffic_direct: *int* = *56*) → bool

Deliver a network frame.

Hint: This function can be used to transmit entire frames accumulated in a user-defined *FrameQueue* object.

```
from circuitpython_nrf24l01.network.structs import FrameQueue, RF24NetworkFrame,␣
↪RF24NetworkHeader

my_q = FrameQueue()
for i in range(my_q.max_queue_size):
    my_q.enqueue(
        RF24NetworkFrame(
            RF24NetworkHeader(0, "1"), bytes(range(i + 5))
        )
    )

# when it's time to send the queue
while len(my_q):
    # let `nrf` be the instantiated RF24Network object
    nrf.write(my_q.dequeue())
```

Parameters

frame : *RF24NetworkFrame*
The complete frame to send. It is important to have the header's *to_node* attribute set to the target network node's address.

traffic_direct : int
The specified direction of the frame. By default, this will invoke the automatic routing mechanisms. However, this parameter can be set to a network node's *Logical Address* for direct

transmission to the specified node - meaning the transmission's automatic routing will begin at the network node that is specified with this parameter instead of being automatically routed from the actual origin of the transmission.

Returns

- `True` if the `frame` has been transmitted. This does not necessarily describe if the message has been received at its target destination.
- `False` if the `frame` has failed to transmit.

Note: This function will always return `True` if the `traffic_direct` parameter is set to anything other than its default value. Using the `traffic_direct` parameter assumes there is a reliable/open connection to the *`node_address`* passed to `traffic_direct`.

Tip: To ensure a message has been delivered to its target destination, set the frame's header's *`message_type`* to an `int` in range [65, 127]. This will invoke a *`NETWORK_ACK`* response message.

`RF24Network.`**`parent`**

Get address for the parent node

Returns `0` if called on the network's master node.

11.5 Configuration API

`RF24Network.`**`max_message_length`**

The maximum length of a frame's message.

By default this is set to `144`. If a network node is driven by the TMRh20 RF24Network library on a ATTiny-based board, set this to `72` (as per TMRh20's RF24Network library default behavior).

Configuring the *`fragmentation`* attribute will automatically change the value that *`max_message_length`* attribute is set to.

`RF24Network.`**`fragmentation`**

Enable/disable (`True`/`False`) the message fragmentation feature.

Changing this attribute's state will also appropriately changes the type of *`FrameQueue`* (or *`FrameQueueFrag`*) object used for storing incoming network packets. Disabling fragmentation can save some memory (not as much as TMRh20's RF24Network library's `DISABLE_FRAGMENTATION` macro), but *`max_message_length`* will be limited to `24` bytes (*`MAX_FRAG_SIZE`*) maximum. Enabling this attribute will set *`max_message_length`* attribute to `144` bytes.

`RF24Network.`**`multicast_relay`**

Enabling this attribute will automatically forward received multicasted frames to the next highest network level.

Forwarded frames will also be enqueued on the forwarding node as a received frame.

`RF24Network.`**`multicast_level`**

Override the default multicasting network level which is set by the *`node_address`* attribute.

Setting this attribute will also change the *physical address* on the radio's RX data pipe 0.

See also:

The network levels are explained in more detail on the topology document.

`RF24Network.allow_multicast`

enable/disable (`True`/`False`) multicasting

This attribute affects

- the *Physical Address* translation (for data pipe 0) when setting the `node_address`
- all incoming multicasted frames (including `multicast_relay` behavior).

`RF24Network.tx_timeout`

The timeout (in milliseconds) to wait for successful transmission.

Defaults to 25.

`RF24Network.route_timeout`

The timeout (in milliseconds) to wait for transmission's `NETWORK_ACK`.

Defaults to 75.

CHAPTER

TWELVE

RF24MESH API

New in version 2.1.0.

See also:

Documentation for:

1. Shared Networking API (API common to *RF24Mesh* and *RF24Network*)
2. RF24Network API (*RF24Mesh* inherits from the same mixin class that *RF24Network* inherits from)

12.1 RF24MeshNoMaster class

class circuitpython_nrf24l01.rf24_mesh.**RF24MeshNoMaster**(*spi: busio.SPI*, *csn_pin: digitalio.DigitalInOut*, *ce_pin: digitalio.DigitalInOut*, node_id*: int*, *spi_frequency: int = 10000000*)

A descendant of the same mixin class that *RF24Network* inherits from. This class adds easy Mesh networking capability (non-master nodes only).

This class exists to save memory for nodes that don't behave like mesh network master nodes. It is the python equivalent to TMRh20's `MESH_NO_MASTER` macro in the C++ RF24Mesh library. All the API is the same as *RF24Mesh* class.

Parameters

node_id : int
The unique identifying *node_id* number for the instantiated mesh node.

See also:

For all parameters' descriptions, see the *RF24* class' constructor documentation.

12.2 RF24Mesh class

class circuitpython_nrf24l01.rf24_mesh.**RF24Mesh**(*spi: busio.SPI*, *csn_pin: digitalio.DigitalInOut*, *ce_pin: digitalio.DigitalInOut*, node_id*: int*, *spi_frequency: int = 10000000*)

Bases: *RF24MeshNoMaster*

A descendant of the base class *RF24MeshNoMaster* that adds algorithms needed for Mesh network master nodes.

Parameters

node_id : int
The unique identifying *node_id* number for the instantiated mesh node.

See also:

For all parameters' descriptions, see the *RF24* class' constructor documentation.

12.3 Basic API

`RF24Mesh.send(to_node: int, message_type: int, str, message: bytes, bytearray) → bool`

Send a message to a mesh *node_id*.

This function will use *lookup_address()* to fetch the necessary *Logical Address* to set the frame's header's *to_node* attribute.

Hint: If you already know the destination node's *Logical Address*, then you can use *write()* for quicker operation.

Parameters

to_node : int
The unique mesh network *node_id* of the frame's destination. Defaults to `0` (which is reserved for the master node).

message_type : str,int
The `int` that describes the frame header's *message_type*.

message : bytes,bytearray

The frame's *message* to be transmitted.

Note: Be mindful of the message's size as this cannot exceed *MAX_FRAG_SIZE* (24 bytes) if *fragmentation* is disabled. If *fragmentation* is enabled (it is by default), then the message's size must be less than *max_message_length*.

Returns

- `True` if the `frame` has been transmitted. This does not necessarily describe if the message has been received at its target destination.
- `False` if the `frame` has not been transmitted.

Tip: To ensure a message has been delivered to its target destination, set the `message_type` parameter to an `int` in range [65, 127]. This will invoke a *NETWORK_ACK* response message.

`RF24Mesh.node_id`

The unique ID number (1 byte long) of the mesh network node.

This is not to be confused with the network node's *node_address*. This attribute is meant to distinguish different mesh network nodes that may, at separate instances, use the same *node_address*. It is up to the developer to make sure each mesh network node uses a different ID number.

Warning: Changing this attributes value after instantiation will automatically call `release_address()` which disconnects the node from the mesh network. Notice the `node_address` is set to `NETWORK_DEFAULT_ADDR` when consciously not connected to the mesh network.

Tip: When a mesh node becomes disconnected from the mesh network, use `renew_address()` to fetch (from the master node) an assigned logical address to be used as the mesh node's `node_address`.

`RF24Mesh.renew_address(timeout: int = 7.5)`

Connect to the mesh network and request a new `node_address`.

Parameters

timeout : float,int
The amount of time (in seconds) to continue trying to connect and get an assigned *Logical Address*. Defaults to 7.5 seconds.

Note: This function automatically sets the `node_address` accordingly.

Returns

- If successful: The `node_address` that was set to the newly assigned *Logical Address*.
- If unsuccessful: `None`, and the `node_address` attribute will be set to `NETWORK_DEFAULT_ADDR` (`0o4444` in octal or `2340` in decimal).

12.4 Advanced API

`RF24Mesh.lookup_node_id(address: int | None = None) → int`

Convert a node's *Logical Address* into its corresponding unique ID number.

Parameters

address : int
The *Logical Address* for which a unique `node_id` is assigned from network master node.

Returns

- The unique `node_id` assigned to the specified `address`.
- Error codes include
  - `-2` means the specified `address` has not been assigned a unique `node_id` from the master node or the requesting network node's `node_address` is equal to `NETWORK_DEFAULT_ADDR`.
  - `-1` means the address lookup operation failed due to no network connection or the master node has not assigned a unique `node_id` for the specified `address`.

`RF24Mesh.lookup_address(node_id: int | None = None) → int`

Convert a node's unique ID number into its corresponding *Logical Address*.

Parameters

node_id : int
The unique *node_id* for which a *Logical Address* is assigned from network master node.

Returns

- The *Logical Address* assigned to the specified `node_id`.
- Error codes include
  - `-2` means the specified `node_id` has not been assigned a *Logical Address* from the master node or the requesting network node's *node_address* is equal to *NETWORK_DEFAULT_ADDR*.
  - `-1` means the address lookup operation failed due to no network connection or the master node has not assigned a *Logical Address* for the specified `node_id`.

`RF24Mesh.write(to_node: int, message_type: int, str, message: bytes, bytearray) → bool`

Send a message to a network *node_address*.

Parameters

to_node : int
The network node's *Logical Address*. of the frame's destination. This must be the destination's network *node_address* which is not be confused with a mesh node's *node_id*.

message_type : str,int

The `int` that describes the frame header's *message_type*.

> **Note:** Be mindful of the message's size as this cannot exceed *MAX_FRAG_SIZE* (24 bytes) if *fragmentation* is disabled. If *fragmentation* is enabled (it is by default), then the message's size must be less than *max_message_length*.

message : bytes,bytearray
The frame's *message* to be transmitted.

Returns

- `True` if the `frame` has been transmitted. This does not necessarily describe if the message has been received at its target destination.
- `False` if the `frame` has not been transmitted.

> **Tip:** To ensure a message has been delivered to its target destination, set the `message_type` parameter to an `int` in range [65, 127]. This will invoke a *NETWORK_ACK* response message.

`RF24Mesh.check_connection() → bool`

Check for network connectivity (not for use on master node).

`RF24Mesh.release_address() → bool`

Forces an address lease to expire from the master.

> **Hint:** This should be called from a mesh network node that is disconnecting from the network. This is also recommended for mesh network nodes that are entering a powered down (or sleep) mode.

`RF24Mesh.allow_children`

Allow/disallow child node to connect to this network node.

`RF24Mesh.block_less_callback`

This variable can be assigned a function to perform during long operations.

Note: Requesting a new address (via `renew_address()`) can take a while since it sequentially attempts to get re-assigned to the first available *Logical Address* on the highest possible network level.

The assigned function will be called during `renew_address()`, `lookup_address()` and `lookup_node_id()`.

`RF24Mesh.dhcp_dict`

A `dict` that enables master nodes to act as a DNS.

This `dict` stores the assigned *Logical Addresses* to the connected mesh node's `node_id`.

- The keys in this `dict` are the unique `node_id` of a mesh network node.
- The values in this `dict` (corresponding to each key) are the `node_address` assigned to the `node_id`.

`RF24Mesh.save_dhcp(filename: str = 'dhcplist.json', as_bin: bool = False)`

Save the `dhcp_dict` to a JSON file (meant for master nodes only).

Warning: This function will likely throw a `OSError` on boards running CircuitPython firmware because the file system is by default read-only.

Calling this function on a Linux device (like the Raspberry Pi) will save the `dhcp_dict` to a JSON file located in the program's working directory.

Parameters

filename : str
The name of the json file to be used. This value should include a file extension (like ".json" or ".txt").

as_bin : bool
Set this parameter to `True` to save the DHCP list to a binary text file. Defaults to `False` which saves the DHCP list as JSON syntax.

Changed in version 2.1.1: Added `as_bin` parameter to make use of binary text files.

`RF24Mesh.load_dhcp(filename: str = 'dhcplist.json', as_bin: bool = False)`

Load the `dhcp_dict` from a JSON file (meant for master nodes only).

Parameters

filename : str
The name of the json file to be used. This value should include a file extension (like ".json" or ".txt").

as_bin : bool
Set this parameter to `True` to load the DHCP list from a binary text file. Defaults to `False` which loads the DHCP list from JSON syntax.

Warning: This function will raise an `OSError` exception if no file exists.

Changed in version 2.1.1: Added `as_bin` parameter to make use of binary text files.

`RF24Mesh.set_address(node_id: int, node_address: int, search_by_address: bool = False)`

Set/change a *node_id* and *node_address* pair in the *dhcp_dict*.

This function is only meant to be called on the mesh network's master node. Use this function to manually assign a *node_id* to a *RF24Network.node_address*.

Parameters

node_id : int
A unique identifying number ranging [1, 255].

node_address : int
A *Logical Address*

search_by_address : bool
A flag to traverse the *dhcp_dict* by value instead of by key.

CHAPTER

THIRTEEN

NETWORK CONSTANTS

New in version 2.1.0.

13.1 Sending Behavior Types

`circuitpython_nrf24l01.network.constants.AUTO_ROUTING` = 56

Send a message with automatic network routing.

`circuitpython_nrf24l01.network.constants.TX_NORMAL` = 0

Send a routed message.

This is used for most outgoing message types.

`circuitpython_nrf24l01.network.constants.TX_ROUTED` = 1

Send a routed message.

This is internally used for *NETWORK_ACK* message routing.

`circuitpython_nrf24l01.network.constants.TX_PHYSICAL` = 2

Send a message directly to network node.

These usually take 1 transmission, so they don't get a network ACK because the radio's *auto_ack* will serve the ACK.

`circuitpython_nrf24l01.network.constants.TX_LOGICAL` = 3

Similar to *TX_NORMAL*.

This allows the user to define the routed transmission's first path (these can still get a *NETWORK_ACK*).

`circuitpython_nrf24l01.network.constants.TX_MULTICAST` = 4

Broadcast a message to a network level of nodes.

See also:

- Network Levels
- *multicast_relay*
- *multicast()*
- *multicast_level*

13.2 Reserved Network Message Types

circuitpython_nrf24l01.network.constants.**MESH_ADDR_RESPONSE** = 128

Primarily for RF24Mesh

This *message_type* is used to in the final step of *renew_address()* route a messages containing a newly allocated *node_address*. The header's *reserved* attribute for this *message_type* will store the requesting mesh node's *node_id* related to the newly assigned *node_address*. Any non-requesting network node receiving this *message_type* will forward it to the requesting node using normal network routing.

circuitpython_nrf24l01.network.constants.**NETWORK_PING** = 130

Used for network pings

This *message_type* is automatically discarded because the radio's *auto_ack* feature will serve up the response.

circuitpython_nrf24l01.network.constants.**NETWORK_EXT_DATA** = 131

Unsupported at this time as this operation requires a new implementation.

Used for bridging different network protocols between an RF24Network and LAN/WLAN networks.

circuitpython_nrf24l01.network.constants.**NETWORK_ACK** = 193

Used for network-wide acknowledgements.

The message type used when forwarding acknowledgements directed to the instigating message's origin. This is not be confused with the radio's *auto_ack* attribute. In fact, all messages (except multicasted ones) take advantage of the radio's *auto_ack* feature when transmitting between directly related nodes (ie between a transmitting node's parent or child node).

Important: NETWORK_ACK messages are only sent by the last node in the route to a destination. For example: Node `0o0` sends an instigating message to node `0o11`. The NETWORK_ACK message is sent from node `0o1` when it confirms node `0o11` received the instigating message.

Hint: This feature is not flawless because it assumes a reliable connection between all necessary network nodes.

circuitpython_nrf24l01.network.constants.**NETWORK_POLL** = 194

Primarily for RF24Mesh

This *message_type* is used with *NETWORK_MULTICAST_ADDR* to find active/available nodes. Any node receiving a *NETWORK_POLL* sent to a *NETWORK_MULTICAST_ADDR* will respond directly to the sender with a blank message, indicating the address of the available node via the header's *from_node* attribute.

circuitpython_nrf24l01.network.constants.**MESH_ADDR_REQUEST** = 195

Primarily for RF24Mesh

This *message_type* is used for requesting *Logical Address* data from the mesh network's master node. Any non-master node receiving this *message_type* will manually forward it to the master node using normal network routing.

circuitpython_nrf24l01.network.constants.**MESH_ADDR_LOOKUP** = 196

The *message_type* to request a mesh node's network address from its unique ID.

circuitpython_nrf24l01.network.constants.**MESH_ADDR_RELEASE** = 197

The *message_type* when manually expiring a leased address.

circuitpython_nrf24l01.network.constants.**MESH_ID_LOOKUP** = 198

The *message_type* to request a mesh node's unique ID number from its node address.

13.3 Generic Network constants

circuitpython_nrf24l01.network.constants.**MAX_USR_DEF_MSG_TYPE** = 127

A convenient sentinel value.

Any message type above 127 (but cannot exceed 255) are reserved for internal network usage.

circuitpython_nrf24l01.network.constants.**NETWORK_DEFAULT_ADDR** = 2340

Primarily used by RF24Mesh.

Any mesh node that disconnects or is trying to connect to a mesh network will use this value until it is assigned a *Logical Address* from the master node.

circuitpython_nrf24l01.network.constants.**NETWORK_MULTICAST_ADDR** = 64

A reserved address for multicast messages.

circuitpython_nrf24l01.network.constants.**MAX_FRAG_SIZE** = 24

Maximum message size for a single frame's message.

This does not including header's byte length (which is always 8 bytes).

> **Warning:** Do not increase this value in the source code. Adjust *max_message_length* instead.

13.4 Message Fragment Types

Message fragments will use these values in the *message_type* attribute. The sequential fragment id number will be stored in the *reserved* attribute, but the actual message type is transmitted in the *reserved* attribute of the last fragment.

circuitpython_nrf24l01.network.constants.**MSG_FRAG_FIRST** = 148

Used to indicate the first frame of a fragmented message.

circuitpython_nrf24l01.network.constants.**MSG_FRAG_MORE** = 149

Used to indicate a middle frame of a fragmented message.

circuitpython_nrf24l01.network.constants.**MSG_FRAG_LAST** = 150

Used to indicate the last frame of a fragmented message.

13.5 RF24Mesh specific constants

circuitpython_nrf24l01.network.constants.**MESH_LOOKUP_TIMEOUT** = 135

Used for *lookup_address()* & *lookup_node_id()*

The time (in milliseconds) that a non-master mesh node will wait for a response when requesting a node's relative *Logical Address* or unique ID number from the master node.

circuitpython_nrf24l01.network.constants.**MESH_MAX_POLL** = 4

The max number of contacts made during *renew_address()*.

A mesh node polls the first 4 network levels (0-3) looking for a response. This value is used to used when aggregating a list of responding nodes (per level).

circuitpython_nrf24l01.network.constants.**MESH_MAX_CHILDREN** = 4

The max number of children for 1 mesh node.

This information is only used by mesh network master nodes when allocating a possible *Logical Address* for the requesting node.

circuitpython_nrf24l01.network.constants.**MESH_WRITE_TIMEOUT** = 115

The time (in milliseconds) used to send messages.

When *RF24Mesh.send()* is called, This value is only used when getting the *node_address* assigned to a *node_id* from the mesh network's master node.

CHAPTER

FOURTEEN

TROUBLESHOOTING INFO

14.1 Common Problems

14.1.1 Attribute dependency

The nRF24L01 has 3 key features.

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.

   Note: 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.

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. *dynamic_payloads* does not require *auto_ack* to be enabled.

3. *ack* feature allows the MCU to append a payload to the ACK packet, thus instant bi-directional communication. A transmitting ACK payload must be loaded into the nRF24L01's TX FIFO buffer (done using *load_ack()*) BEFORE receiving the payload that is to be acknowledged. Once transmitted, the payload is released from the TX FIFO buffer.

   Important: This *ack* feature requires the *auto_ack* and *dynamic_payloads* features enabled.

14.1.2 FIFO Capacity

Remember that the nRF24L01's FIFO (First-In, First-Out) buffers have 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). Notice there are separate FIFO buffers sending & receiving (respectively mentioned in this documentation as TX FIFO & RX FIFO).

Each of the 3 levels in the FIFO buffers can only store a *maximum* of 32 bytes. If you receive 2 payloads with a length of 4 bytes each, then there is only 1 level of the RX FIFO buffers left unoccupied.

14.1.3 Pipes vs Addresses vs Channels

Hint: Please review the Multiceiver example as a demonstration of proper addressing using all pipes (on the same channel).

Pipes

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.

Addresses

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 of the `bytearray` is unique among simultaneously broadcasting addresses), so you're not limited to communicating with only the same 6 nRF24L01 radios.

Channels

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.

14.1.4 Settings that must Match

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*
- custom *ack* payloads
- *crc*

Settings that do not need to Match

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()* (as long as the corresponding addresses match)
- *pa_level*
- *arc*
- *ard*

The `ask_no_ack` feature can be used despite the settings/features configuration (see *send()* & *write()* function parameters for more details).

14.2 About the lite version

New in version 1.2.0.

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

- The *FakeBLE*, *RF24Network*, and *RF24Mesh* classes are 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.
- *print_details()* 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. This attribute will return a `bool` instead of an `int`. *set_dynamic_payloads()* and *get_dynamic_payloads()* have been removed.
- *payload_length* applies to all pipes, not individual pipes. *set_payload_length()* and *get_payload_length()* have been removed.
- *load_ack()* is available, but it will not throw exceptions for malformed `buf` or invalid `pipe_number` parameters. Rather any call to *load_ack()* with invalid parameters will have no affect on the TX FIFO.
- *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`.

- *start_carrier_wave()*, & *stop_carrier_wave()* removed. These only perform a test of the nRF24L01's hardware. *rpd* is still available.
- 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).
- *last_tx_arc* attribute removed because it is only meant for troubleshooting.
- *allow_ask_no_ack* attribute removed because it is only provided for the Si24R1 chinese clone.
- *set_auto_retries()* & *get_auto_retries()* removed. Use *ard* & *arc* attributes instead.

14.3 Testing nRF24L01+PA+LNA module

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

14.3.1 The Setup

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

14.3.2 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).

14.3.3 Conclusion

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

CHAPTER

FIFTEEN

GETTING STARTED

15.1 Introduction

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

15.1.1 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 payloads to the acknowledgment (ACK) packets for instant bi-directional communication
- Mark a single payload for no acknowledgment (ACK) from the receiving nRF24L01 (see `ask_no_ack` parameter for *`send()`* and *`write()`* functions)
- Invoke the "re-use the same payload" feature (for manually re-transmitting failed transmissions that remain in the TX FIFO buffer)
- Multiple payload transmissions with one function call (see documentation on the *`send()`* function and try out the Stream example)
- Context manager compatible for easily switching between different radio configurations using `The with statement` blocks (not available in `rf24_lite.py` version)
- Configure the interrupt (IRQ) pin to trigger (active low) on received, sent, and/or failed transmissions (these 3 events control 1 IRQ pin). There's also virtual representations of these interrupt events available (see *`irq_dr`*, *`irq_ds`*, & *`irq_df`* attributes)
- Invoke sleep mode (AKA power down mode) for ultra-low current consumption
- cyclic redundancy checking (CRC) up to 2 bytes long
- Adjust the nRF24L01's builtin automatic re-transmit feature's parameters (*`arc`*: number of attempts, *`ard`*: delay between attempts)
- Adjust the nRF24L01's frequency channel (2.4 - 2.525 GHz)

- Adjust the nRF24L01's power amplifier level (0, -6, -12, or -18 dBm)
- Adjust the nRF24L01's RF data rate (250kbps, 1Mbps, or 2Mbps)
- An nRF24L01 driven by this library can communicate with a nRF24L01 on an Arduino driven by the TMRh20 RF24 library.
- fake BLE module for sending BLE beacon advertisements from the nRF24L01 as outlined by Dmitry Grinberg in his write-up (including C source code).
- Multiceiver™ mode (up to 6 TX nRF24L01 "talking" to 1 RX nRF24L01 simultaneously). See the Multiceiver Example
- Networking capability that allows up to 781 transceivers to interact with each other.
  - This does not mean the radio can connect to WiFi. The networking implementation is a custom protocol ported from TMRh20's RF24Network & RF24Mesh libraries.

15.1.2 Dependencies

This driver depends on:

- Adafruit CircuitPython Firmware or the Adafruit_Blinka library for Linux SoC boards like Raspberry Pi
- `adafruit_bus_device` (specifically the `SPIDevice` class)

  Tip: Use CircuitPython v6.3.0 or newer because faster SPI execution yields faster transmissions.

- The SpiDev module is a C-extension that executes SPI transactions faster than Adafruit's PureIO library (a dependency of the Adafruit_Blinka library).

The `adafruit_bus_device`, Adafruit_Blinka library, and SpiDev libraries are installed automatically on Linux when installing this library.

New in version 2.1.0: Added support for the SpiDev module

Important: This library supports Python 3.7 or newer because it uses the function `time.monotonic_ns()` which returns an arbitrary time "counter" as an `int` of nanoseconds. CircuitPython firmware also supports `time.monotonic_ns()`.

15.1.3 Installing from PyPI

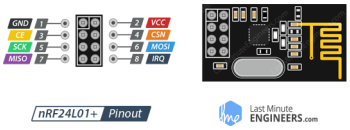
On supported GNU/Linux systems like the Raspberry Pi, you can install the driver locally from PyPI. To install for current user:

```
pip3 install circuitpython-nrf24l01
```

To install 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
```

15.2 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 GPIO pin on the nRF24L01 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.

Table 1: The pins used in this library's examples.

nRF24L01	Itsy Bitsy M4	Raspberry Pi
GND	GND	GND
VCC	3.3V	3V
CE	D4	• GPIO4 if using CircuitPython's `SPIDevice` • GPIO22 if using the SpiDev module
CSN	D5	• GPIO5 if using CircuitPython's `SPIDevice` • GPIO8 (CE0) if using the SpiDev module
SCK	SCK	GPIO11 (SCK)
MOSI	MOSI	GPIO10 (MOSI)
MISO	MISO	GPIO9 (MISO)
IRQ	D12	GPIO12

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.

Important: The nRF24L01's VCC pin is not 5V compliant. All other nRF24L01 pins *should* be 5V compliant, but it is safer to assume they are not.

15.3 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.

15.3.1 For an interactive REPL

All examples can be imported from within an interactive python REPL.

1. Make sure the examples are located in the current working directory. On CircuitPython devices, this will be the root directory of the CIRCUITPY drive.
2. Import everything from the desired example. The following code snippet demonstrates running the Simple Test example

```
>>> from nrf24l01_simple_test import *
Which radio is this? Enter '0' or '1'. Defaults to '0'
    nRF24L01 Simple test.
    Run slave() on receiver
    Run master() on transmitter
>>> master()
Transmission successful! Time to Transmit: 3906.25 us. Sent: 0.0
Transmission successful! Time to Transmit: 2929.69 us. Sent: 0.01
Transmission successful! Time to Transmit: 2929.69 us. Sent: 0.02
Transmission successful! Time to Transmit: 3906.25 us. Sent: 0.03
Transmission successful! Time to Transmit: 4882.81 us. Sent: 0.04
```

15.3.2 For CircuitPython devices

1. Copy the examples to the root directory of the CIRCUITPY device.
2. Rename the desired example file to `main.py`.
3. If the REPL is not already running, then the example should start automatically. If the REPL is already running in interactive mode, then press `ctrl+d` to do a soft reset, and the example should start automatically.

15.3.3 For CPython in Linux

1. Clone the library repository, then navigate to the repository's example directory.

```
git clone https://github.com/2bndy5/CircuitPython_nRF24L01.git
cd CircuitPython_nRF24L01/examples
```

2. Run the example as a normal python program

```
python3 nrf24l01_simple_test.py
```

15.4 What to purchase

See the following links to Sparkfun or just google "nRF24L01+".

- 2.4GHz Transceiver IC - nRF24L01+
- SparkFun Transceiver Breakout - nRF24L01+
- SparkFun Transceiver Breakout - nRF24L01+ (RP-SMA)

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

See also:

Beware, there are also *nrf24l01(+) clones and counterfeits* that may not work the same.

15.4.1 Power Stability

If you're not using a dedicated 3V regulator to supply power to the nRF24L01, then adding capacitor(s) (100 µF + an optional 0.1µF) 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)

15.4.2 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 additional circuitry 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:

Additional requirements for the PA/LNA modules

These requirements are dependent on what manufacturer produced the radio module.

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

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

Important: These values may be different depending on what manufacturer produced the radio module. Please consult the manufacturer's specifications or datasheet.

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:

I have documented Testing nRF24L01+PA+LNA module

15.4.3 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. The Si24R1 is also famous for not supporting `auto_ack` correctly because the designers "cloned" a typo from the 1st version of the nRF24L01 (non-plus) datasheet into the Si24R1 firmware. Other known clones include the bk242x (also known as RFM7x).

See also:

Read this article about using clones with missing capacitors (includes pictures).

15.5 Contributing

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

15.5.1 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)
- TCI/IP OSI layer, maybe something like TMRh20's RF24Ethernet
- implement the Gazelle-based protocol used by the BBC micro-bit (makecode.com's radio blocks) Additional resources can be found at the MicroPython firmware source code and its related documentation.

15.5.2 Sphinx documentation

Sphinx and Graphviz are used to build the documentation based on rST files and comments in the code.

Install Graphviz

On Windows, installing Graphviz library is done differently. Check out the Graphviz downloads page. Be sure that the `graphviz/bin` directory is in the `PATH` environment variable (there's an option in the installer for this). After Graphviz is installed, reboot the PC so the updated `PATH` environment variable takes affect.

On Linux, just run:

```
sudo apt-get install graphviz
```

Installing Sphinx necessities

First, install dependencies (feel free to reuse the virtual environment from above):

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

Building the documentation

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` directory. 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.

15.6 Site Index

genindex

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W