# RF24 Class Reference

```
#include <RF24.h>
```

## Public Member Functions

### Primary public interface

These are the main methods you need to operate the chip

| | |
|---|---|
| | **RF24** (uint8_t _cepin, uint8_t _cspin) |
| | **RF24** (uint8_t _cepin, uint8_t _cspin, uint32_t spispeed) |
| bool | **begin** (void) |
| void | **startListening** (void) |
| void | **stopListening** (void) |
| bool | **available** (void) |
| void | **read** (void *buf, uint8_t len) |
| bool | **write** (const void *buf, uint8_t len) |
| void | **openWritingPipe** (const uint8_t *address) |
| void | **openReadingPipe** (uint8_t number, const uint8_t *address) |

### Deprecated

Methods provided for backwards compability.

| | |
|---|---|
| void | **openReadingPipe** (uint8_t number, uint64_t address) |
| void | **openWritingPipe** (uint64_t address) |

## Protected Member Functions

| | |
|---|---|
| void | **beginTransaction** () |
| void | **endTransaction** () |

## Advanced Operation

Methods you can use to drive the chip in more advanced ways

| | |
|---|---|
| bool | **failureDetected** |
| void | **printDetails** (void) |
| bool | **available** (uint8_t *pipe_num) |
| bool | **rxFifoFull** () |
| void | **powerDown** (void) |

| | |
|---|---|
| void | **powerUp** (void) |
| bool | **write** (const void *buf, uint8_t len, const bool multicast) |
| bool | **writeFast** (const void *buf, uint8_t len) |
| bool | **writeFast** (const void *buf, uint8_t len, const bool multicast) |
| bool | **writeBlocking** (const void *buf, uint8_t len, uint32_t timeout) |
| bool | **txStandBy** () |
| bool | **txStandBy** (uint32_t timeout, bool startTx=0) |
| void | **writeAckPayload** (uint8_t pipe, const void *buf, uint8_t len) |
| bool | **isAckPayloadAvailable** (void) |
| void | **whatHappened** (bool &tx_ok, bool &tx_fail, bool &rx_ready) |
| void | **startFastWrite** (const void *buf, uint8_t len, const bool multicast, bool startTx=1) |
| void | **startWrite** (const void *buf, uint8_t len, const bool multicast) |
| void | **reUseTX** () |
| uint8_t | **flush_tx** (void) |
| bool | **testCarrier** (void) |
| bool | **testRPD** (void) |
| bool | **isValid** () |
| void | **closeReadingPipe** (uint8_t pipe) |

## Optional Configurators

Methods you can use to get or set the configuration of the chip. None are required. Calling **begin()** sets up a reasonable set of defaults.

| | |
|---|---|
| uint32_t | **txDelay** |
| uint32_t | **csDelay** =5 |
| void | **setAddressWidth** (uint8_t a_width) |
| void | **setRetries** (uint8_t **delay**, uint8_t count) |
| void | **setChannel** (uint8_t channel) |
| uint8_t | **getChannel** (void) |
| void | **setPayloadSize** (uint8_t size) |
| uint8_t | **getPayloadSize** (void) |
| uint8_t | **getDynamicPayloadSize** (void) |
| void | **enableAckPayload** (void) |
| void | **enableDynamicPayloads** (void) |
| void | **enableDynamicAck** () |
| bool | **isPVariant** (void) |
| void | **setAutoAck** (bool enable) |
| void | **setAutoAck** (uint8_t pipe, bool enable) |

| | |
|---|---|
| void | **setPALevel** (uint8_t level) |
| uint8_t | **getPALevel** (void) |
| bool | **setDataRate** (**rf24_datarate_e speed**) |
| **rf24_datarate_e** | **getDataRate** (void) |
| void | **setCRCLength** (**rf24_crclength_e** length) |
| **rf24_crclength_e** | **getCRCLength** (void) |
| void | **disableCRC** (void) |
| void | **maskIRQ** (bool tx_ok, bool tx_fail, bool rx_ready) |

## Detailed Description

Driver for nRF24L01(+) 2.4GHz Wireless Transceiver

**Examples:**

**gettingstarted.cpp**, **GettingStarted.ino**, **gettingstarted_call_response.cpp**, **GettingStarted_CallResponse.ino**, **GettingStarted_HandlingData.ino**, **pingpair_ack.ino**, **pingpair_dyn.cpp**, **pingpair_dyn.ino**, **pingpair_irq.ino**, **pingpair_irq_simple.ino**, **pingpair_sleepy.ino**, **rf24ping85.ino**, **scanner.ino**, **starping.pde**, **transfer.cpp**, **Transfer.ino**, and **TransferTimeouts.ino**.

Definition at line **51** of file **RF24.h**.

## Constructor & Destructor Documentation

```
RF24::RF24 ( uint8_t  _cepin,
             uint8_t  _cspin
           )
```

Arduino Constructor

Creates a new instance of this driver. Before using, you create an instance and send in the unique pins that this chip is connected to.

**Parameters**

| | |
|---|---|
| **_cepin** | The pin attached to Chip Enable on the RF module |
| **_cspin** | The pin attached to Chip Select |

Definition at line **418** of file **RF24.cpp**.

**RF24::RF24 ( uint8_t _cepin,**
**uint8_t _cspin,**
**uint32_t spispeed**
**)**

Optional Linux Constructor

Creates a new instance of this driver. Before using, you create an instance and send in the unique pins that this chip is connected to.

**Parameters**

| | |
|---|---|
| **_cepin** | The pin attached to Chip Enable on the RF module |
| **_cspin** | The pin attached to Chip Select |
| **spispeed** | For RPi, the SPI speed in MHZ ie: BCM2835_SPI_SPEED_8MHZ |

## Member Function Documentation

**void RF24::beginTransaction ( )** inline protected

SPI transactions

Common code for SPI transactions including CSN toggle

Definition at line **68** of file **RF24.cpp**.

**void RF24::endTransaction ( )** inline protected

Definition at line **77** of file **RF24.cpp**.

**bool RF24::begin ( void )**

Begin operation of the chip

Call this in setup(), before calling any other methods.

```
radio.begin()
```

Definition at line **571** of file **RF24.cpp**.

### void RF24::startListening ( void )

Start listening on the pipes opened for reading.

1. Be sure to call **openReadingPipe()** first.
2. Do not call **write()** while in this mode, without first calling **stopListening()**.
3. Call **available()** to check for incoming traffic, and **read()** to get it.

```
Open reading pipe 1 using address CCCECCCECC

byte address[] = { 0xCC,0xCE,0xCC,0xCE,0xCC };
radio.openReadingPipe(1,address);
radio.startListening();
```

Definition at line **696** of file **RF24.cpp**.

### void RF24::stopListening ( void )

Stop listening for incoming messages, and switch to transmit mode.

Do this before calling **write()**.

```
radio.stopListening();
radio.write(&data,sizeof(data));
```

Definition at line **727** of file **RF24.cpp**.

### bool RF24::available ( void )

Check whether there are bytes available to be read

```
if(radio.available()){
  radio.read(&data,sizeof(data));
}
```

**Returns**

True if there is a payload available, false if none is

Definition at line **1060** of file **RF24.cpp**.

**void RF24::read ( void * buf,**
**uint8_t len**
**)**

Read the available payload

The size of data read is the fixed payload size, see **getPayloadSize()**

**Note**

I specifically chose 'void*' as a data type to make it easier for beginners to use. No casting needed.

No longer boolean. Use available to determine if packets are available. Interrupt flags are now cleared during reads instead of when calling **available()**.

**Parameters**

| | |
|---|---|
| **buf** | Pointer to a buffer where the data should be written |
| **len** | Maximum number of bytes to read into the buffer |

```
if(radio.available()){
  radio.read(&data,sizeof(data));
}
```

**Returns**

No return value. Use **available()**.

Definition at line **1087** of file **RF24.cpp**.

### bool RF24::write ( const void * buf, uint8_t len )

Be sure to call **openWritingPipe()** first to set the destination of where to write to.

This blocks until the message is successfully acknowledged by the receiver or the timeout/retransmit maxima are reached. In the current configuration, the max delay here is 60-70ms.

The maximum size of data written is the fixed payload size, see **getPayloadSize()**. However, you can write less, and the remainder will just be filled with zeroes.

TX/RX/RT interrupt flags will be cleared every time write is called

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |

```
radio.stopListening();
radio.write(&data,sizeof(data));
```

**Returns**

True if the payload was delivered successfully false if not

Definition at line **831** of file **RF24.cpp**.

### void RF24::openWritingPipe ( const uint8_t * address )

New: Open a pipe for writing via byte array. Old addressing format retained for compatibility.

Only one writing pipe can be open at once, but you can change the address you'll write to. Call **stopListening()** first.

Addresses are assigned via a byte array, default is 5 byte address length s *

```
uint8_t addresses[][6] = {"1Node","2Node"};
radio.openWritingPipe(addresses[0]);
```

```
uint8_t address[] = { 0xCC,0xCE,0xCC,0xCE,0xCC };
radio.openWritingPipe(address);
address[0] = 0x33;
radio.openReadingPipe(1,address);
```

**See also**

**setAddressWidth**

**Parameters**

| | |
|---|---|
| **address** | The address of the pipe to open. Coordinate these pipe addresses amongst nodes on the network. |

Definition at line **1128** of file **RF24.cpp**.

### void RF24::openReadingPipe ( uint8_t number, const uint8_t * address )

Open a pipe for reading

Up to 6 pipes can be open for reading at once. Open all the required reading pipes, and then call **startListening()**.

**See also**

openWritingPipe

setAddressWidth

**Note**

Pipes 0 and 1 will store a full 5-byte address. Pipes 2-5 will technically only store a single byte, borrowing up to 4 additional bytes from pipe #1 per the assigned address width.

**Warning**

Pipes 1-5 should share the same address, except the first byte. Only the first byte in the array should be unique, e.g.

```
uint8_t addresses[][6] = {"1Node","2Node"};
openReadingPipe(1,addresses[0]);
openReadingPipe(2,addresses[1]);
```

Pipe 0 is also used by the writing pipe. So if you open pipe 0 for reading, and then **startListening()**, it will overwrite the writing pipe. Ergo, do an **openWritingPipe()** again before **write()**.

**Parameters**

| | |
|---|---|
| **number** | Which pipe# to open, 0-5. |
| **address** | The 24, 32 or 40 bit address of the pipe to open. |

Definition at line **1190** of file **RF24.cpp**.

### void RF24::printDetails ( void )

Print a giant block of debugging information to stdout

**Warning**

Does nothing if stdout is not defined. See fdevopen in stdio.h The **printf.h** file is included with the library for Arduino.

```
#include <printf.h>
setup(){
 Serial.begin(115200);
 printf_begin();
 ...
}
```

Definition at line **512** of file **RF24.cpp**.

### bool RF24::available ( uint8_t * pipe_num )

Test whether there are bytes available to be read in the FIFO buffers.

**Parameters**

[out] **pipe_num** Which pipe has the payload available

```
uint8_t pipeNum;
if(radio.available(&pipeNum)){
  radio.read(&data,sizeof(data));
  Serial.print("Got data on pipe");
  Serial.println(pipeNum);
}
```

**Returns**

True if there is a payload available, false if none is

Definition at line **1067** of file **RF24.cpp**.

### bool RF24::rxFifoFull ( )

Check if the radio needs to be read. Can be used to prevent data loss

**Returns**

True if all three 32-byte radio buffers are full

Definition at line **956** of file **RF24.cpp**.

### void RF24::powerDown ( void )

Enter low-power mode

To return to normal power mode, call **powerUp()**.

**Note**

After calling **startListening()**, a basic radio will consume about 13.5mA at max PA level. During active transmission, the radio will consume about 11.5mA, but this will be reduced to 26uA (.026mA) between sending. In full powerDown mode, the radio will consume approximately 900nA (.0009mA)

```
radio.powerDown();
avr_enter_sleep_mode(); // Custom function to sleep the device
radio.powerUp();
```

Definition at line **755** of file **RF24.cpp**.

### void RF24::powerUp ( void )

Leave low-power mode - required for normal radio operation after calling **powerDown()**

To return to low power mode, call **powerDown()**.

**Note**

> This will take up to 5ms for maximum compatibility

Definition at line **764** of file **RF24.cpp**.

### bool RF24::write ( const void * buf, uint8_t len, const bool multicast )

```
bool RF24::write ( const void *  buf,
                   uint8_t       len,
                   const bool    multicast
                 )
```

Write for single NOACK writes. Optionally disables acknowledgements/autoretries for a single write.

**Note**

> **enableDynamicAck()** must be called to enable this feature

Can be used with **enableAckPayload()** to request a response

**See also**

- **enableDynamicAck()**
- **setAutoAck()**
- **write()**

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |
| **multicast** | Request ACK (0), NOACK (1) |

Definition at line **795** of file **RF24.cpp**.

### bool RF24::writeFast ( const void * buf, uint8_t len )

This will not block until the 3 FIFO buffers are filled with data. Once the FIFOs are full, writeFast will simply wait for success or timeout, and return 1 or 0 respectively. From a user perspective, just keep trying to send the same data. The library will keep auto retrying the current payload using the built in functionality.

**Warning**

It is important to never keep the nRF24L01 in TX mode and FIFO full for more than 4ms at a time. If the auto retransmit is enabled, the nRF24L01 is never in TX mode long enough to disobey this rule. Allow the FIFO to clear by issuing **txStandBy()** or ensure appropriate time between transmissions.

```
Example (Partial blocking):

        radio.writeFast(&buf,32);  // Writes 1 payload to the buffers
        txStandBy();               // Returns 0 if failed. 1 if success. Blocks only until
       MAX_RT timeout or success. Data flushed on fail.

        radio.writeFast(&buf,32);  // Writes 1 payload to the buffers
        txStandBy(1000);           // Using extended timeouts, returns 1 if success. Retries
       failed payloads for 1 seconds before returning 0.
```

**See also**

**txStandBy()**

**write()**

**writeBlocking()**

**Parameters**

**buf** Pointer to the data to be sent

**len** Number of bytes to be sent

**Returns**

True if the payload was delivered successfully false if not

Definition at line **914** of file **RF24.cpp**.

```
bool RF24::writeFast ( const void *  buf,
                       uint8_t       len,
                       const bool    multicast
                     )
```

WriteFast for single NOACK writes. Disables acknowledgements/autoretries for a single write.

**Note**

enableDynamicAck() must be called to enable this feature

**See also**

enableDynamicAck()

setAutoAck()

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |
| **multicast** | Request ACK (0) or NOACK (1) |

Definition at line **880** of file **RF24.cpp**.

### bool RF24::writeBlocking ( const void * buf, uint8_t len, uint32_t timeout )

This function extends the auto-retry mechanism to any specified duration. It will not block until the 3 FIFO buffers are filled with data. If so the library will auto retry until a new payload is written or the user specified timeout period is reached.

**Warning**

It is important to never keep the nRF24L01 in TX mode and FIFO full for more than 4ms at a time. If the auto retransmit is enabled, the nRF24L01 is never in TX mode long enough to disobey this rule. Allow the FIFO to clear by issuing **txStandBy()** or ensure appropriate time between transmissions.

```
Example (Full blocking):

         radio.writeBlocking(&buf,32,1000); //Wait up to 1 second to write 1 payload to the
        buffers
         txStandBy(1000);                   //Wait up to 1 second for the payload to send. Return
        1 if ok, 0 if failed.
                                            //Blocks only until user timeout or success. Data
        flushed on fail.
```

**Note**

If used from within an interrupt, the interrupt should be disabled until completion, and sei(); called to enable **millis()**.

**See also**

**txStandBy()**

**write()**

**writeFast()**

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |
| **timeout** | User defined timeout in milliseconds. |

**Returns**

True if the payload was loaded into the buffer successfully false if not

Definition at line **837** of file **RF24.cpp**.

### bool RF24::txStandBy ( )

This function should be called as soon as transmission is finished to drop the radio back to STANDBY-I mode. If not issued, the radio will remain in STANDBY-II mode which, per the data sheet, is not a recommended operating mode.

**Note**

> When transmitting data in rapid succession, it is still recommended by the manufacturer to drop the radio out of TX or STANDBY-II mode if there is time enough between sends for the FIFOs to empty. This is not required if auto-ack is enabled.

Relies on built-in auto retry functionality.

```
Example (Partial blocking):

        radio.writeFast(&buf,32);
        radio.writeFast(&buf,32);
        radio.writeFast(&buf,32);  //Fills the FIFO buffers up
        bool ok = txStandBy();     //Returns 0 if failed. 1 if success.
                                   //Blocks only until MAX_RT timeout or success. Data flushed on fail.
```

**See also**

txStandBy(unsigned long timeout)

**Returns**

True if transmission is successful

Definition at line **961** of file **RF24.cpp**.

### bool RF24::txStandBy ( uint32_t timeout, bool startTx = 0 )

This function allows extended blocking and auto-retries per a user defined timeout

```
Fully Blocking Example:

     radio.writeFast(&buf,32);
     radio.writeFast(&buf,32);
     radio.writeFast(&buf,32);   //Fills the FIFO buffers up
     bool ok = txStandBy(1000);  //Returns 0 if failed after 1 second of retries. 1 if success.
                                 //Blocks only until user defined timeout or success. Data
        flushed on fail.
```

**Note**

If used from within an interrupt, the interrupt should be disabled until completion, and sei(); called to enable **millis()**.

**Parameters**

**timeout** Number of milliseconds to retry failed payloads

**Returns**

True if transmission is successful

Definition at line **989** of file **RF24.cpp**.

```
void RF24::writeAckPayload ( uint8_t      pipe,
                             const void * buf,
                             uint8_t      len
                           )
```

Write an ack payload for the specified pipe

The next time a message is received on `pipe`, the data in `buf` will be sent back in the acknowledgement.

**See also**

**enableAckPayload()**

**enableDynamicPayloads()**

**Warning**

Only three of these can be pending at any time as there are only 3 FIFO buffers.
Dynamic payloads must be enabled.

**Note**

Ack payloads are handled automatically by the radio chip when a payload is received. Users should generally write an ack payload as soon as **startListening()** is called, so one is available when a regular payload is received.

Ack payloads are dynamic payloads. This only works on pipes 0&1 by default. Call **enableDynamicPayloads()** to enable on all pipes.

**Parameters**

| | |
|---|---|
| **pipe** | Which pipe# (typically 1-5) will get this response. |
| **buf** | Pointer to data that is sent |
| **len** | Length of the data to send, up to 32 bytes max. Not affected by the static payload set by **setPayloadSize()**. |

Definition at line **1291** of file **RF24.cpp**.

```
bool RF24::isAckPayloadAvailable ( void   )
```

Determine if an ack payload was received in the most recent call to **write()**. The regular **available()** can also be used.

Call **read()** to retrieve the ack payload.

**Returns**

True if an ack payload is available.

Definition at line **1322** of file **RF24.cpp**.

```
void RF24::whatHappened ( bool & tx_ok,
                          bool & tx_fail,
                          bool & rx_ready
                        )
```

Call this when you get an interrupt to find out why

Tells you what caused the interrupt, and clears the state of interrupts.

**Parameters**

| | | |
|---|---|---|
| `[out]` | **tx_ok** | The send was successful (TX_DS) |
| `[out]` | **tx_fail** | The send failed, too many retries (MAX_RT) |
| `[out]` | **rx_ready** | There is a message waiting to be read (RX_DS) |

Definition at line **1099** of file **RF24.cpp**.

```
void RF24::startFastWrite ( const void *  buf,
                            uint8_t       len,
                            const bool    multicast,
                            bool          startTx = 1
                          )
```

Non-blocking write to the open writing pipe used for buffered writes

**Note**

Optimization: This function now leaves the CE pin high, so the radio will remain in TX or STANDBY-II Mode until a **txStandBy()** command is issued. Can be used as an alternative to **startWrite()** if writing multiple payloads at once.

**Warning**

It is important to never keep the nRF24L01 in TX mode with FIFO full for more than 4ms at a time. If the auto retransmit/autoAck is enabled, the nRF24L01 is never in TX mode long enough to disobey this rule. Allow the FIFO to clear by issuing **txStandBy()** or ensure appropriate time between transmissions.

**See also**

**write()**

**writeFast()**

**startWrite()**

**writeBlocking()**

For single noAck writes see:

**See also**

**enableDynamicAck()**

**setAutoAck()**

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |
| **multicast** | Request ACK (0) or NOACK (1) |

**Returns**

True if the payload was delivered successfully false if not

Definition at line **925** of file **RF24.cpp**.

```
void RF24::startWrite ( const void *  buf,
                        uint8_t       len,
                        const bool    multicast
                      )
```

Non-blocking write to the open writing pipe

Just like **write()**, but it returns immediately. To find out what happened to the send, catch the IRQ and then call **whatHappened()**.

**See also**

- **write()**
- **writeFast()**
- **startFastWrite()**
- **whatHappened()**

For single noAck writes see:

**See also**

- **enableDynamicAck()**
- **setAutoAck()**

**Parameters**

| | |
|---|---|
| **buf** | Pointer to the data to be sent |
| **len** | Number of bytes to be sent |
| **multicast** | Request ACK (0) or NOACK (1) |

Definition at line **939** of file **RF24.cpp**.

### void RF24::reUseTX ( )

This function is mainly used internally to take advantage of the auto payload re-use functionality of the chip, but can be beneficial to users as well.

The function will instruct the radio to re-use the data in the FIFO buffers, and instructs the radio to re-send once the timeout limit has been reached. Used by writeFast and writeBlocking to initiate retries when a TX failure occurs. Retries are automatically initiated except with the standard **write()**. This way, data is not flushed from the buffer until switching between modes.

**Note**

> This is to be used AFTER auto-retry fails if wanting to resend using the built-in payload reuse features. After issuing **reUseTX()**, it will keep reending the same payload forever or until a payload is written to the FIFO, or a flush_tx command is given.

Definition at line **871** of file **RF24.cpp**.

### uint8_t RF24::flush_tx ( void )

Empty the transmit buffer. This is generally not required in standard operation. May be required in specific cases after **stopListening()** , if operating at 250KBPS data rate.

**Returns**

Current value of status register

Definition at line **326** of file **RF24.cpp**.

### bool RF24::testCarrier ( void )

Test whether there was a carrier on the line for the previous listening period.

Useful to check for interference on the current channel.

**Returns**

true if was carrier, false if not

Definition at line **1365** of file **RF24.cpp**.

### bool RF24::testRPD ( void )

Test whether a signal (carrier or otherwise) greater than or equal to -64dBm is present on the channel. Valid only on nRF24L01P (+) hardware. On nRF24L01, use **testCarrier()**.

Useful to check for interference on the current channel and channel hopping strategies.

```
bool goodSignal = radio.testRPD();
if(radio.available()){
   Serial.println(goodSignal ? "Strong signal > 64dBm" : "Weak signal < 64dBm" );
   radio.read(0,0);
}
```

**Returns**

true if signal => -64dBm, false if not

Definition at line **1372** of file **RF24.cpp**.

### bool RF24::isValid ( ) inline

Test whether this is a real radio, or a mock shim for debugging. Setting either pin to 0xff is the way to indicate that this is not a real radio.

**Returns**

true if this is a legitimate radio

Definition at line **643** of file **RF24.h**.

### void RF24::closeReadingPipe ( uint8_t pipe )

Close a pipe after it has been previously opened. Can be safely called without having previously opened a pipe.

**Parameters**

**pipe** Which pipe # to close, 0-5.

Definition at line **1218** of file **RF24.cpp**.

### void RF24::setAddressWidth ( uint8_t a_width )

Set the address width from 3 to 5 bytes (24, 32 or 40 bit)

**Parameters**

**a_width** The address width to use: 3,4 or 5

Definition at line **1179** of file **RF24.cpp**.

### void RF24::setRetries ( uint8_t delay, uint8_t count )

Set the number and delay of retries upon failed submit

**Parameters**

**delay** How long to wait between each retry, in multiples of 250us, max is 15. 0 means 250us, 15 means 4000us.

**count** How many retries before giving up, max 15

Definition at line **1531** of file **RF24.cpp**.

### void RF24::setChannel ( uint8_t channel )

Set RF communication channel

**Parameters**

**channel** Which RF channel to communicate on, 0-125

Definition at line **437** of file **RF24.cpp**.

### uint8_t RF24::getChannel ( void )

Get RF communication channel

**Returns**

The currently configured RF Channel

Definition at line **443** of file **RF24.cpp**.

### void RF24::setPayloadSize ( uint8_t size )

Set Static Payload Size

This implementation uses a pre-stablished fixed payload size for all transmissions. If this method is never called, the driver will always transmit the maximum payload size (32 bytes), no matter how much was sent to **write()**.

**Parameters**

**size** The number of bytes in the payload

Definition at line **450** of file **RF24.cpp**.

### uint8_t RF24::getPayloadSize ( void )

Get Static Payload Size

**See also**

setPayloadSize()

**Returns**

The number of bytes in the payload

Definition at line 457 of file RF24.cpp.

### uint8_t RF24::getDynamicPayloadSize ( void )

Get Dynamic Payload Size

For dynamic payloads, this pulls the size of the payload off the chip

**Note**

Corrupt packets are now detected and flushed per the manufacturer.

```
if(radio.available()){
  if(radio.getDynamicPayloadSize() < 1){
    // Corrupt payload has been flushed
    return;
  }
  radio.read(&data,sizeof(data));
}
```

**Returns**

Payload length of last-received dynamic payload

Definition at line 1036 of file RF24.cpp.

### void RF24::enableAckPayload ( void )

Enable custom payloads on the acknowledge packets

Ack payloads are a handy way to return data back to senders without manually changing the radio modes on both units.

**Note**

Ack payloads are dynamic payloads. This only works on pipes 0&1 by default. Call **enableDynamicPayloads()** to enable on all pipes.

Definition at line 1256 of file RF24.cpp.

### void RF24::enableDynamicPayloads ( void )

Enable dynamically-sized payloads

This way you don't always have to send large packets just to send them once in a while. This enables dynamic payloads on ALL pipes.

Definition at line **1235** of file **RF24.cpp**.

### void RF24::enableDynamicAck ( void )

Enable dynamic ACKs (single write multicast or unicast) for chosen messages

**Note**

To enable full multicast or per-pipe multicast, use **setAutoAck()**

**Warning**

This MUST be called prior to attempting single write NOACK calls

```
radio.enableDynamicAck();
radio.write(&data,32,1);  // Sends a payload with no acknowledgement requested
radio.write(&data,32,0);  // Sends a payload using auto-retry/autoACK
```

Definition at line **1277** of file **RF24.cpp**.

### bool RF24::isPVariant ( void )

Determine whether the hardware is an nRF24L01+ or not.

**Returns**

true if the hardware is nRF24L01+ (or compatible) and false if its not.

Definition at line **1329** of file **RF24.cpp**.

### void RF24::setAutoAck ( bool enable )

Enable or disable auto-acknowlede packets

This is enabled by default, so it's only needed if you want to turn it off for some reason.

**Parameters**

**enable** Whether to enable (true) or disable (false) auto-acks

Definition at line **1336** of file **RF24.cpp**.

### void RF24::setAutoAck ( uint8_t pipe, bool enable )

Enable or disable auto-acknowlede packets on a per pipeline basis.

AA is enabled by default, so it's only needed if you want to turn it off/on for some reason on a per pipeline basis.

**Parameters**

| | |
|---|---|
| **pipe** | Which pipeline to modify |
| **enable** | Whether to enable (true) or disable (false) auto-acks |

Definition at line **1346** of file **RF24.cpp**.

### void RF24::setPALevel ( uint8_t level )

Set Power Amplifier (PA) level to one of four levels: RF24_PA_MIN, RF24_PA_LOW, RF24_PA_HIGH and RF24_PA_MAX

The power levels correspond to the following output levels respectively: NRF24L01: -18dBm, -12dBm,-6dBM, and 0dBm

SI24R1: -6dBm, 0dBm, 3dBM, and 7dBm.

**Parameters**

| | |
|---|---|
| **level** | Desired PA level. |

Definition at line **1379** of file **RF24.cpp**.

### uint8_t RF24::getPALevel ( void )

Fetches the current PA level.

NRF24L01: -18dBm, -12dBm, -6dBm and 0dBm SI24R1: -6dBm, 0dBm, 3dBm, 7dBm

**Returns**

Returns values 0 to 3 representing the PA Level.

Definition at line **1396** of file **RF24.cpp**.

### bool RF24::setDataRate ( rf24_datarate_e speed )

Set the transmission data rate

**Warning**

setting RF24_250KBPS will fail for non-plus units

**Parameters**

| | |
|---|---|
| **speed** | RF24_250KBPS for 250kbs, RF24_1MBPS for 1Mbps, or RF24_2MBPS for 2Mbps |

**Returns**

true if the change was successful

Definition at line **1404** of file **RF24.cpp**.

### rf24_datarate_e RF24::getDataRate ( void )

Fetches the transmission data rate

**Returns**

Returns the hardware's currently configured datarate. The value is one of 250kbs, RF24_1MBPS for 1Mbps, or RF24_2MBPS, as defined in the rf24_datarate_e enum.

Definition at line **1454** of file **RF24.cpp**.

### void RF24::setCRCLength ( rf24_crclength_e length )

Set the CRC length
CRC checking cannot be disabled if auto-ack is enabled

**Parameters**

| | |
|---|---|
| **length** | RF24_CRC_8 for 8-bit or RF24_CRC_16 for 16-bit |

Definition at line **1481** of file **RF24.cpp**.

### rf24_crclength_e RF24::getCRCLength ( void )

Get the CRC length
CRC checking cannot be disabled if auto-ack is enabled

**Returns**

RF24_CRC_DISABLED if disabled or RF24_CRC_8 for 8-bit or RF24_CRC_16 for 16-bit

Definition at line **1504** of file **RF24.cpp**.

### void RF24::disableCRC ( void )

Disable CRC validation

**Warning**

CRC cannot be disabled if auto-ack/ESB is enabled.

Definition at line **1524** of file **RF24.cpp**.

### void RF24::maskIRQ ( bool tx_ok, bool tx_fail, bool rx_ready )

The radio will generate interrupt signals when a transmission is complete, a transmission fails, or a payload is received. This allows users to mask those interrupts to prevent them from generating a signal on the interrupt pin. Interrupts are enabled on the radio chip by default.

```
Mask all interrupts except the receive interrupt:

radio.maskIRQ(1,1,0);
```

**Parameters**

| | |
|---|---|
| **tx_ok** | Mask transmission complete interrupts |
| **tx_fail** | Mask transmit failure interrupts |
| **rx_ready** | Mask payload received interrupts |

Definition at line **1024** of file **RF24.cpp**.

### void RF24::openReadingPipe ( uint8_t number, uint64_t address )

Open a pipe for reading

**Note**

For compatibility with old code only, see new function

**Warning**

Pipes 1-5 should share the first 32 bits. Only the least significant byte should be unique, e.g.

```
openReadingPipe(1,0xF0F0F0F0AA);
openReadingPipe(2,0xF0F0F0F066);
```

Pipe 0 is also used by the writing pipe. So if you open pipe 0 for reading, and then **startListening()**, it will overwrite the writing pipe. Ergo, do an **openWritingPipe()** again before **write()**.

**Parameters**

| | |
|---|---|
| **number** | Which pipe# to open, 0-5. |
| **address** | The 40-bit address of the pipe to open. |

Definition at line **1152** of file **RF24.cpp**.

### void RF24::openWritingPipe ( uint64_t address )

Open a pipe for writing

**Note**

For compatibility with old code only, see new function

Addresses are 40-bit hex values, e.g.:

```
openWritingPipe(0xF0F0F0F0F0);
```

**Parameters**

| | |
|---|---|
| **address** | The 40-bit address of the pipe to open. |

Definition at line **1113** of file **RF24.cpp**.

## Member Data Documentation

### bool RF24::failureDetected

Enable error detection by un-commenting #define FAILURE_HANDLING in **RF24_config.h** If a failure has been detected, it usually indicates a hardware issue. By default the library will cease operation when a failure is detected. This should allow advanced users to detect and resolve intermittent hardware issues.

In most cases, the radio must be re-enabled via radio.begin(); and the appropriate settings applied after a failure occurs, if wanting to re-enable the device immediately.

Usage: (Failure handling must be enabled per above)

```
if(radio.failureDetected){
  radio.begin();                       // Attempt to re-configure the radio with defaults
  radio.failureDetected = 0;           // Reset the detection value
  radio.openWritingPipe(addresses[1]); // Re-configure pipe addresses
  radio.openReadingPipe(1,addresses[0]);
  report_failure();                    // Blink leds, send a message, etc. to indicate failure
}
```

Definition at line **673** of file **RF24.h**.

### uint32_t RF24::txDelay

The driver will delay for this duration when **stopListening()** is called

When responding to payloads, faster devices like ARM(RPi) are much faster than Arduino:

1. Arduino sends data to RPi, switches to RX mode
2. The RPi receives the data, switches to TX mode and sends before the Arduino radio is in RX mode
3. If AutoACK is disabled, this can be set as low as 0. If AA/ESB enabled, set to 100uS minimum on RPi

**Warning**

If set to 0, ensure 130uS delay after **stopListening()** and before any sends

Definition at line **920** of file **RF24.h**.

### uint32_t RF24::csDelay =5

On all devices but Linux and ATTiny, a small delay is added to the CSN toggling function

This is intended to minimise the speed of SPI polling due to radio commands

If using interrupts or timed requests, this can be set to 0 Default:5

Definition at line **931** of file **RF24.h**.

The documentation for this class was generated from the following files:

- **RF24.h**
- **RF24.cpp**

Generated on Sat Nov 5 2016 10:02:27 for Optimized High Speed NRF24L01+ Driver Class Documenation by doxygen 1.8.10