RF24Network library

Release 1.0.16

nRF24

Nov 28, 2021

API REFERENCE

This class implements an OSI Network Layer using nRF24L01(+) radios driven by the newly optimized RF24 library fork.

RF24 Library docs for general RF24 configuration and setup.

- Linux Installation and General Linux/RPi configuration and setup documentation

CHAPTER

ONE

PURPOSE/GOAL

Original: Create an alternative to ZigBee radios for Arduino communication.

New: Enhance the current functionality for maximum efficiency, reliability, and speed

Xbees are excellent little radios, backed up by a mature and robust standard protocol stack. They are also expensive.

For many Arduino uses, they seem like overkill. So I am working to improve the current standard for nRF24L01 radios. The best RF24 modules are available for less than $6 from many sources. With the RF24Network layer, I hope to cover many common communication scenarios.

Please see TMRh20's blog post for a comparison against the ZigBee protocols

CHAPTER

TWO

NEWS

Please see the recent changes listed in the github releases page

CHAPTER

THREE

FEATURES

3.1 The layer provides

- Network ACKs: Efficient acknowledgement of network-wide transmissions, via dynamic radio acks and network protocol acks.
- Updated addressing standard for optimal radio transmission.
- Extended timeouts and staggered timeout intervals. The new txTimeout variable allows fully automated extended timeout periods via auto-retry/auto-reUse of payloads.
- Optimization to the core library provides improvements to reliability, speed and efficiency. See RF24 library documentation for more info.
- Built in sleep mode using interrupts. (Still under development enable via RF24Network_config.h)
- Host Addressing. Each node has a logical address on the local network.
- Message Forwarding. Messages can be sent from one node to any other, and this layer will get them there no matter how many hops it takes.
- Ad-hoc Joining. A node can join a network without any changes to any existing nodes.

3.2 The layer does not provide

- Dynamic address assignment (See RF24Mesh)
- Layer 4 protocols (TCP/IP - See RF24Ethernet and RF24Gateway)

CHAPTER

FOUR

HOW TO LEARN MORE

- RF24Network Class Documentation
- Advanced Configuration Options
- Addressing format
- Topology and Overview
- Download Current Development Package
- Examples Page. Start with `helloworld_*` examples.

4.1 Additional Information & Add-ons

- RF24Mesh: Dynamic Mesh Layer for RF24Network Dev
- RF24Ethernet: TCP/IP over RF24Network
- TMRh20's Blog: RF24 Optimization Overview
- TMRh20's Blog: RF24 Wireless Audio
- RF24: Original Author

CHAPTER

FIVE

TOPOLOGY FOR MESH NETWORKS USING NRF24L01(+)

This network layer takes advantage of the fundamental capability of the nRF24L01(+) radio to listen actively to up to 6 other radios at once. The network is arranged in a Tree Topology, where one node is the base, and all other nodes are children either of that node, or of another. Unlike a true mesh network, multiple nodes are not connected together, so there is only one path to any given node.

CHAPTER

SIX

OCTAL ADDRESSING AND TOPOLOGY

Each node must be assigned an 15-bit address by the administrator. This address exactly describes the position of the node within the tree. The address is an octal number. Each digit in the address represents a position in the tree further from the base.

- Node 00 is the base node.
- Nodes 01-05 are nodes whose parent is the base.
- Node 021 is the second child of node 01.
- Node 0321 is the third child of node 021, an so on.
- The largest node address is 05555, so up to 781 nodes are allowed on a single channel. An example topology is shown below, with 5 nodes in direct communication with the master node, and multiple leaf nodes spread out at a distance, using intermediate nodes to reach other nodes.

CHAPTER

SEVEN

HOW ROUTING IS HANDLED

When sending a message using *RF24Network::write()*, you fill in the header with the logical node address. The network layer figures out the right path to find that node, and sends it through the system until it gets to the right place. This works even if the two nodes are far separated, as it will send the message down to the base node, and then back out to the final destination.

All of this work is handled by the *RF24Network::update()* method, so be sure to call it regularly or your network will miss packets.

CHAPTER

EIGHT

STARTING UP A NODE

When a node starts up, it only has to contact its parent to establish communication. No direct connection to the Base node is needed. This is useful in situations where relay nodes are being used to bridge the distance to the base, so leaf nodes are out of range of the base.

CHAPTER

NINE

DIRECTIONALITY

By default all nodes are always listening, so messages will quickly reach their destination.

You may choose to sleep any nodes on the network if using interrupts. This is useful in a case where the nodes are operating on batteries and need to sleep. This greatly decreases the power requirements for a sensor network. The leaf nodes can sleep most of the time, and wake every few minutes to send in a reading. Routing nodes can be triggered to wake up whenever a payload is received See *RF24Network::sleepNode()* in the class documentation, and RF24Network_config.h to enable sleep mode.

CHAPTER

TEN

SITE INDEX

Site index

10.1 RF24Network class

class **RF24Network**

RF24Network::**RF24Network**(RF24 &_radio)
Construct the network

Parameters **_radio** – The underlying radio driver instance

See also:

Use the RF24 class to create the radio object.

10.1.1 Basic API

inline void *RF24Network*::**begin**(uint16_t _node_address)
Bring up the network using the current radio frequency/channel. Calling begin brings up the network, and configures the address, which designates the location of the node within RF24Network topology.

Example 1: Begin on current radio channel with address 0 (master node)

```
network.begin(00);
```

Example 2: Begin with address 01 (child of master)

```
network.begin(01);
```

Example 3: Begin with address 011 (child of 01, grandchild of master)

```
network.begin(011);
```

See *begin(uint8_t _channel, uint16_t _node_address)*

Note: Node addresses are specified in Octal format, see RF24Network Addressing for more information. The address `04444` is reserved for RF24Mesh usage (when a mesh node is connecting to the network).

Warning: Be sure to first call `RF24::begin()` to initialize the radio properly.

Parameters **_node_address** – The logical address of this node.

void *RF24Network*::**begin**(uint8_t _channel, uint16_t _node_address)

Bring up the network on a specific radio frequency/channel.

Deprecated:

Use `RF24::setChannel()` to configure the radio channel. Use *RF24Network::begin(uint16_t _node_address)* to set the node address.

Example 1: Begin on channel 90 with address 0 (master node)

```
network.begin(90, 0);
```

Example 2: Begin on channel 90 with address 01 (child of master)

```
network.begin(90, 01);
```

Example 3: Begin on channel 90 with address 011 (child of 01, grandchild of master)

```
network.begin(90, 011);
```

Parameters

- **_channel** – The RF channel to operate on.
- **_node_address** – The logical address of this node.

uint8_t *RF24Network*::**update**(void)

Main layer loop

This function must be called regularly to keep the layer going. This is where payloads are re-routed, received, and all the action happens.

Returns Returns the *RF24NetworkHeader::type* of the last received payload.

bool *RF24Network*::**available**(void)

Test whether there is a message available for this node

Returns Whether there is a message available for this node

uint16_t *RF24Network*::**read**(*RF24NetworkHeader* &header, void *message, uint16_t maxlen = MAX_PAYLOAD_SIZE)

Read a message

Note: This function assumes there is a frame in the queue.

```
while (network.available()) {
  RF24NetworkHeader header;
  uint32_t time;
  uint16_t msg_size = network.peek(header);
  if (header.type == 'T') {
    network.read(header, &time, sizeof(time));
    Serial.print("Got time: ");
    Serial.println(time);
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Parameters

- **header** – **[out]** The *RF24NetworkHeader* (envelope) of this message
- **message** – **[out]** Pointer to memory where the message should be placed
- **maxlen** – The largest message size which can be held in `message` . If this parameter is left unspecified, the entire length of the message is fetched. Hint: Use *peek(RF24NetworkHeader &)* to get the length of next available message in the queue.

Returns The total number of bytes copied into `message`

bool *RF24Network*::**write**(*RF24NetworkHeader* &header, const void *message, uint16_t len)

Send a message

```
uint32_t time = millis();
uint16_t to = 00; // Send to master
RF24NetworkHeader header(to, 'T'); // Send header type 'T'
network.write(header, &time, sizeof(time));
```

Note: RF24Network now supports fragmentation for very long messages, send as normal. Fragmentation may need to be enabled or configured by editing the RF24Network_config.h file. Default max payload size is 120 bytes.

Parameters

- **header** – **[inout]** The header (envelope) of this message. The critical thing to fill in is the `to_node` field so we know where to send the message. It is then updated with the details of the actual header sent.
- **message** – Pointer to memory where the message is located
- **len** – The size of the message

Returns Whether the message was successfully received

uint16_t *RF24Network*::**peek**(*RF24NetworkHeader* &header)

Read the next available header

Reads the next available header without advancing to the next incoming message. Useful for doing a switch on the message type.

Parameters **header** – **[out]** The *RF24NetworkHeader* (envelope) of the next message. If there is no message available, the referenced `header` object is not touched

Returns The length of the next available message in the queue.

void *RF24Network*::**peek**(*RF24NetworkHeader* &header, void *message, uint16_t maxlen = MAX_PAYLOAD_SIZE)

Read the next available payload

Reads the next available payload without advancing to the next incoming message. Useful for doing a transparent packet manipulation layer on top of RF24Network.

Parameters

- **header** – **[out]** The *RF24NetworkHeader* (envelope) of this message
- **message** – **[out]** Pointer to memory where the message should be placed
- **maxlen** – Amount of bytes to copy to `message` . If this parameter is left unspecified, the entire length of the message is fetched. Hint: Use peek(RF24NetworkHeader) to get the length of next available message in the queue.

10.1.2 Advanced API

void *RF24Network*::**failures**(uint32_t *_fails, uint32_t *_ok)

Return the number of failures and successes for all transmitted payloads, routed or sent directly

```
bool fails, success;
network.failures(&fails, &success);
```

Note: This needs to be enabled via `#define ENABLE_NETWORK_STATS` in RF24Network_config.h

bool *RF24Network*::**multicast**(*RF24NetworkHeader* &header, const void *message, uint16_t len, uint8_t level = 7)

Send a multicast message to multiple nodes at once Allows messages to be rapidly broadcast through the network

Multicasting is arranged in levels, with all nodes on the same level listening to the same address Levels are assigned by network level ie: nodes 01-05: Level 1, nodes 011-055: Level 2

See

- *multicastLevel()*
- *multicastRelay*

Parameters

- **header** – reference to the *RF24NetworkHeader* object used for this `message`
- **message** – Pointer to memory where the message is located
- **len** – The size of the message
- **level** – Multicast level to broadcast to. If this parameter is unspecified, then the node's current *multicastLevel()* is used.

Returns Whether the message was successfully sent

bool *RF24Network*::**write**(*RF24NetworkHeader* &header, const void *message, uint16_t len, uint16_t writeDirect)

Writes a direct (unicast) payload. This allows routing or sending messages outside of the usual routing paths. The same as write, but a physical address is specified as the last option. The payload will be written to the physical address, and routed as necessary by the recipient.

bool *RF24Network*::**sleepNode**(unsigned int cycles, int interruptPin, uint8_t INTERRUPT_MODE = 0)

Sleep this node - For AVR devices only

The node can be awoken in two ways, both of which can be enabled simultaneously:

a. An interrupt - usually triggered by the radio receiving a payload. Must use pin 2 (interrupt 0) or 3 (interrupt 1) on Uno, Nano, etc.

b. The watchdog timer waking the MCU after a designated period of time, can also be used instead of delays to control transmission intervals.

```
if(!network.available())
    network.sleepNode(1, 0); // Sleep the node for 1 second or a payload is received

// Other options:
network.sleepNode(0, 0);     // Sleep this node for the designated time period, or␣
↪a payload is received.
network.sleepNode(1, 255);   // Sleep this node for 1 cycle. Do not wake up until␣
↪then, even if a payload is received ( no interrupt )
```

Note: NEW - Nodes can now be slept while the radio is not actively transmitting. This must be manually enabled by uncommenting the `#define ENABLE_SLEEP_MODE` in RF24Network_config.h

Note: The watchdog timer should be configured in the sketch's `setup()` if using sleep mode. This function will sleep the node, with the radio still active in receive mode. See *setup_watchdog()*.

Parameters

- **cycles** – The node will sleep in cycles of 1s. Using 2 will sleep 2 WDT cycles, 3 sleeps 3WDT cycles…
- **interruptPin** – The interrupt number to use (0, 1) for pins 2 and 3 on Uno & Nano. More available on Mega etc. Setting this parameter to 255 will disable interrupt wake-ups.
- **INTERRUPT_MODE** – an identifying number to indicate what type of state for which the `interrupt_pin` will be used to wake up the radio.

INTERRUPT_MODE	type of state
0	LOW
1	RISING
2	FALLING
3	CHANGE

Returns True if sleepNode completed normally, after the specified number of cycles. False if sleep was interrupted

uint16_t *RF24Network*::**parent**() const
This node's parent address

Returns This node's parent address, or 65535 (-1 when casted to a signed int16_t) if this is the master node.

uint16_t *RF24Network*::**addressOfPipe**(uint16_t node, uint8_t pipeNo)
Provided a node address and a pipe number, will return the RF24Network address of that child pipe for that node.

bool *RF24Network*::**is_valid_address**(uint16_t node)
Validate a network address as a proper logical address

Remark This function will validate an improper address of `0100` as it is the reserved *NETWORK_MULTICAST_ADDRESS* used for multicasted messages.

Note: Addresses are specified in octal form, ie 011, 034. Review RF24Nettwork addressing for more information.

Parameters `node` – The specified logical address of a network node.

Returns True if the specified `node` address is a valid network address, otherwise false.

10.1.3 Configuration API

bool *RF24Network*::**multicastRelay**
Enabling this will allow this node to automatically forward received multicast frames to the next highest multicast level. Forwarded frames will also be enqueued on the forwarding node as a received frame.

This is disabled by default.

See *multicastLevel()*

uint32_t *RF24Network*::**txTimeout**
Network timeout value.

Sets the timeout period for individual payloads in milliseconds at staggered intervals. Payloads will be retried automatically until success or timeout. Set to 0 to use the normal auto retry period defined by `radio.setRetries()`.

Note: This value is automatically assigned based on the node address to reduce errors and increase throughput of the network.

---

uint16_t *RF24Network*::**routeTimeout**
Timeout for routed payloads.

This only affects payloads that are routed through one or more nodes. This specifies how long to wait for an ack from across the network. Radios sending directly to their parent or children nodes do not utilize this value.

void *RF24Network*::**multicastLevel**(uint8_t level)
By default, multicast addresses are divided into 5 network levels:

- The master node is the only node on level 0 (the lowest level)
- Nodes 01-05 (level 1) share a multicast address
- Nodes 0n1-0n5 (level 2) share a multicast address
- Nodes 0n11-0n55 (level 3) share a multicast address
- Nodes 0n111-0n555 (level 4) share a multicast address

Notice "n" (used in the list above) stands for an octal digit in range [0, 5]

This optional function is used to override the default level set when a node's logical address changes, and it can be used to create custom multicast groups that all share a single address.

See

- *multicastRelay*
- *multicast()*
- The topology image

Parameters `level` – Levels 0 to 4 are available. All nodes at the same level will receive the same messages if in range. Messages will be routed in order of level, low to high, by default.

void *RF24Network*::**setup_watchdog**(uint8_t prescalar)

Set up the watchdog timer for sleep mode using the number 0 through 10 to represent the following time periods:

wdt_16ms = 0, wdt_32ms, wdt_64ms, wdt_128ms, wdt_250ms, wdt_500ms, wdt_1s, wdt_2s, wdt_4s, wdt_8s

```
setup_watchdog(7);   // Sets the WDT to trigger every second
```

Parameters `prescalar` – The WDT prescaler to define how often the node will wake up. When defining sleep mode cycles, this time period is 1 cycle.

10.1.4 External Applications/Systems

Interface for External Applications and Systems (RF24Mesh, RF24Ethernet)

uint8_t *RF24Network*::**frame_buffer**[MAX_FRAME_SIZE]
The raw system frame buffer.

This member can be accessed to retrieve the latest received data just after it is enqueued. This buffer is also used for outgoing data.

Note: The first 8 bytes of this buffer is latest handled frame's *RF24NetworkHeader* data.

Warning: Conditionally, this buffer may only contain fragments of a message (either outgoing or incoming).

std::queue<*RF24NetworkFrame*> *RF24Network*::**external_queue**
Linux platforms only

Data with a header type of *EXTERNAL_DATA_TYPE* will be loaded into a separate queue. The data can be accessed as follows:

```
RF24NetworkFrame f;
while(network.external_queue.size() > 0) {
  f = network.external_queue.front();
  uint16_t dataSize = f.message_size;
```

(continues on next page)

(continued from previous page)

```
  // read the frame message buffer
  memcpy(&myBuffer, &f.message_buffer, dataSize);
  network.external_queue.pop();
}
```

RF24NetworkFrame ******RF24Network*::**frag_ptr**

ARDUINO platforms only

The `frag_ptr` is only used with Arduino (not RPi/Linux) and is mainly used for external data systems like RF24Ethernet. When a payload of type *EXTERNAL_DATA_TYPE* is received, and returned from *update()*, the `frag_ptr` will always point to the starting memory location of the received frame.

This is used by external data systems (RF24Ethernet) to immediately copy the received data to a buffer, without using the user-cache.

```
uint8_t return_type = network.update();
if(return_type == EXTERNAL_DATA_TYPE) {
    memcpy(&myDataBuffer, network.frag_ptr->message_buffer, network.frag_ptr->
↪message_size);
}
```

Linux devices (defined as `RF24_LINUX`) currently cache all payload types, and do not utilize `frag_ptr`.

See *RF24NetworkFrame*

bool *RF24Network*::**returnSysMsgs**

Variable to determine whether *update()* will return after the radio buffers have been emptied (DEFAULT), or whether to return immediately when (most) system types are received.

As an example, this is used with RF24Mesh to catch and handle system messages without loading them into the user cache.

The following reserved/system message types are handled automatically, and not returned.

System Message Types (Not Returned)
NETWORK_ADDR_RESPONSE
NETWORK_ACK
NETWORK_PING
NETWORK_POLL (With multicast enabled)
NETWORK_REQ_ADDRESS

uint8_t *RF24Network*::**networkFlags**

Network Flags allow control of data flow

Incoming Blocking: If the network user-cache is full, lets radio cache fill up. Radio ACKs are not sent when radio internal cache is full.

This behaviour may seem to result in more failed sends, but the payloads would have otherwise been dropped due to the cache being full.

FLAGS	Value	Description
FLAG_FAST_FRAG	4 (bit 2 asserted)	INTERNAL: Replaces the fastFragTransfer variable, and allows for faster transfers between directly connected nodes.
FLAG_NO_POLL	8 (bit 3 asserted)	EXTERNAL/USER: Disables *NETWORK_POLL* responses on a node-by-node basis.

Note: Bit posistions 0 & 1 in the `networkFlags` byte are no longer used as they once were during experimental development.

Protected Members

These members are accessible by RF24Network derivatives.

uint8_t *RF24Network*::**_multicast_level**
The current node's network level (used for multicast TX/RX-ing).

See Use *multicastLevel()* to adjust this when needed.

uint16_t *RF24Network*::**node_address**
Logical node address of this unit, typically in range [0, 2925] (that's [0, 05555] in octal).

Note: The values 0 represents the network master node. Additionally, the value 1 is occupied when using RF24Ethernet layer.

10.2 Network Payload Structures

struct **RF24NetworkFrame**
Frame structure for internal message handling, and for use by external applications

The actual frame put over the air consists of a header (8-bytes) and a message payload (Up to 24-bytes)

When data is received, it is stored using the *RF24NetworkFrame* structure, which includes:

a. The header containing information about routing the message and the message type

b. The size of the included message

c. The 'message' or data being received

Public Functions

inline **RF24NetworkFrame**()
Default constructor

Simply constructs a blank frame. Frames are generally used internally. See *RF24NetworkHeader*.

inline **RF24NetworkFrame**(*RF24NetworkHeader* &_header, const void *_message = NULL, uint16_t _len = 0)
Constructor for Linux platforms - create a network frame with data Frames are constructed and handled differently on Arduino/AVR and Linux devices (`#if defined RF24_LINUX`)

Frames are used internally and by external systems. See *RF24NetworkHeader*.

Parameters

- **_header** – The RF24Network header to be stored in the frame
- **_message** – The 'message' or data.
- **_len** – The size of the 'message' or data.

inline **RF24NetworkFrame**(*RF24NetworkHeader* &_header, uint16_t _message_size)
Constructor for Arduino/AVR/etc. platforms - create a network frame with data Frames are constructed and handled differently on Arduino/AVR and Linux devices (`#if defined RF24_LINUX`)

Frames are used internally and by external systems. See *RF24NetworkHeader*.

See *RF24Network.frag_ptr*

Parameters

- **_header** – The RF24Network header to be stored in the frame
- **_message_size** – The size of the 'message' or data

Public Members

RF24NetworkHeader **header**
Header which is sent with each message

uint16_t **message_size**
The size in bytes of the payload length

uint8_t ***message_buffer**
On Arduino, the message buffer is just a pointer, and can be pointed to any memory location. On Linux the message buffer is a standard byte array, equal in size to the defined MAX_PAYLOAD_SIZE

struct **RF24NetworkHeader**
Header which is sent with each message

The frame put over the air consists of this header and a message

Headers are addressed to the appropriate node, and the network forwards them on to their final destination.

Public Functions

inline **RF24NetworkHeader**()
Default constructor

Simply constructs a blank header

inline **RF24NetworkHeader**(uint16_t _to, unsigned char _type = 0)
Send constructor

Fragmentation is enabled by default for all devices except ATTiny

Configure fragmentation and max payload size in RF24Network_config.h

Use this constructor to create a header and then send a message

```
uint16_t recipient_address = 011;

RF24NetworkHeader header(recipient_address, 't');

network.write(header, &message, sizeof(message));
```

Note: Now supports automatic fragmentation for very long messages, which can be sent as usual if fragmentation is enabled.

Parameters

- **_to** – The Octal format, logical node address where the message is going
- **_type** – The type of message which follows. Only 0 - 127 are allowed for user messages. Types 1 - 64 will not receive a network acknowledgement.

const char ***toString**(void) const
Create debugging string

Useful for debugging. Dumps all members into a single string, using internal static memory. This memory will get overridden next time you call the method.

Returns String representation of the object's significant members.

Public Members

uint16_t **from_node**
Logical address where the message was generated

uint16_t **to_node**
Logical address where the message is going

uint16_t **id**
Sequential message ID, incremented every time a new frame is constructed

unsigned char **type**
Type of the packet. 0 - 127 are user-defined types, 128 - 255 are reserved for system.

User message types 1 through 64 will NOT be acknowledged by the network, while message types 65 through 127 will receive a network ACK. System message types 192 through 255 will NOT be acknowledged by the network. Message types 128 through 192 will receive a network ACK.

See Reserved System Message Types

unsigned char **reserved**
Reserved for system use

During fragmentation, it carries the fragment_id, and on the last fragment it carries the header_type.

Public Static Attributes

static uint16_t **next_id** = 1
The message ID of the next message to be sent. This attribute is not sent with the header.

10.3 RF24Network_config.h

Defines

NETWORK_DEFAULT_ADDRESS 04444
A reserved valid address for use with RF24Mesh (when a mesh node requests an assigned address)

NETWORK_MULTICAST_ADDRESS 0100
A sentinel address value for multicasting purposes.

NETWORK_AUTO_ROUTING 070
A sentinel value for internally indicating that the frame should be automatically routed as necessary.

SLOW_ADDR_POLL_RESPONSE 10
Adds a delay to node prior to transmitting *NETWORK_ADDR_RESPONSE* messages.

By default this is undefined for speed. This defined number of milliseconds is only applied to the master node when replying to a child trying to connect to the mesh network.

Note: It is advised to define this if any child node is running CircuitPython because the execution speed in pure python is inherently slower than it is in C++.

RF24NetworkMulticast
When defined, this will allow the use of multicasting messages.

MAX_PAYLOAD_SIZE 144
Maximum size of fragmented network frames and fragmentation cache.

Note: This buffer can now be any size > 24. Previously this needed to be a multiple of 24 (changed in v1.0.15).

Note: If used with RF24Ethernet, this value is used to set the buffer sizes.

Note: For nodes driven by an ATTiny based chip, this is set to 72. However, defining `DISABLE_FRAGMENTION` truncates the actual transmitted payload to 24 bytes (which is also the default behavior on ATTiny devices).

MAIN_BUFFER_SIZE (*MAX_PAYLOAD_SIZE* + FRAME_HEADER_SIZE)
The allocated size of the incoming frame buffer.

This is the user-cache, where incoming data is stored. Data is stored using Frames: Header (8 bytes) + Message_Size (2 bytes) + Message_Data (? bytes)

Note: Over-The-Air (OTA) transmissions don't include the message size in the transmitted packet.

ENABLE_DYNAMIC_PAYLOADS
Enable dynamic payloads - If using different types of nRF24L01 modules, some may be incompatible when using this feature

10.4 Deprecated API

Member *RF24Network::begin* (uint8_t _channel, uint16_t _node_address)
Use `RF24::setChannel()` to configure the radio channel. Use *RF24Network::begin(uint16_t _node_address)* to set the node address.

10.5 Related Pages

10.5.1 Contributing

These are the current requirements for getting your code included in RF24Network:

- Try your best to follow the rest of the code, if you're unsure then the NASA C style can help as it's closest to the current style: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19950022400.pdf
- Definetly follow PEP-8 if it's Python code.
- Follow the Arduino IDE formatting style for Arduino examples
- Add doxygen-compatible documentation to any new functions you add, or update existing documentation if you change behaviour
- CMake modules and CMakeLists.txt files should also have a uniform syntax.
  - Indentation is a mandatory 4 spaces (not a `\t` character).
  - Closing parenthesis for multi-line commands should have the same indentation as the line that opened the parenthesis.

- For other useful CMake syntax convention, please see CMake docs for developers and this useful best CMake practices article. The qiBuild project has some well-reasoned "Dos & Don'ts" guideline, but beware that the nRF24 organization is not related to the qiBuild project in any way.

10.5.2 Addressing Format: Understanding Addressing and Topology

An overview of addressing in RF24Network

Overview

The nrf24 radio modules typically use a 40-bit address format, requiring 5-bytes of storage space per address, and allowing a wide array of addresses to be utilized. In addition, the radios are limited to direct communication with 6 other nodes while using the Enhanced-Shock-Burst (ESB) functionality of the radios.

RF24Network uses a simple method of data compression to store the addresses using only 2 bytes, in a format designed to represent the network topology in an intuitive way. See the Topology and Overview page for more info regarding topology.

Decimal, Octal and Binary formats

Say we want to designate a logical address to a node, using a tree topology as defined by the manufacturer. In the simplest format, we could assign the first node the address of 1, the second 2, and so on. Since a single node can only connect to 6 other nodes (1 parent and 5 children) subnets need to be created if using more than 6 nodes. In this case, the

- children of node 1 could simply be designated as 11, 21, 31, 41, and 51
- children of node 2 could be designated as 12, 22, 32, 42, and 52

The above example is exactly how RF24Network manages the addresses, but they are represented in Octal format.

Decimal, Octal and Binary

Decimal	Octal	Binary
1	01	00000001
11	013	00001011
9	011	00001001
73	0111	01001001
111	0157	01101111

Since the numbers 0-7 can be represented in exactly three bits, each digit is represented by exactly 3 bits when viewed in octal format. This allows a very simple method of managing addresses via masking and bit shifting.

Displaying Addresses

When using Arduino devices, octal addresses can be printed in the following manner:

```
uint16_t address = 0111;
Serial.println(address, OCT);
```

Printf can also be used, if enabled, or if using linux/RPi

```
uint16_t address = 0111;
printf("0%o\n", address);
```

- This cplusplus.com tutorial for more information number bases.
- The Topology and Overview page for more information regarding network topology.

10.5.3 Advanced Configuration

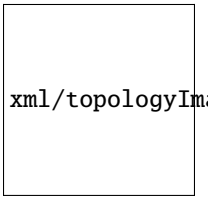
RF24Network offers many features, some of which can be configured by editing the RF24Network_config.h file

Configuration Option	Description
#define RF24NetworkMulticast	This option allows nodes to send and receive multicast payloads. Nodes with multicast enabled can also be configured to relay multicast payloads on to further multicast levels. See *RF24Network::multicastRelay*
#define DISABLE_FRAGMENTATION	Fragmentation is enabled by default, and uses an additional 144 bytes of memory.
#define MAX_PAYLOAD_SIZE 144	The maximum size of payloads defaults to 144 bytes. If used with RF24toTUN and two Raspberry Pi, set this to 1500
#define DISABLE_USER_PAYLOADS	This option will disable user-caching of payloads entirely. Use with RF24Ethernet to reduce memory usage. (TCP/IP is an external data type, and not cached)
#define ENABLE_SLEEP_MODE	Uncomment this option to enable sleep mode for AVR devices. (ATTiny,Uno, etc)
#define ENABLE_NETWORK_STATS	Enable counting of all successful or failed transmissions, routed or sent directly

10.5.4 Performance and Data Loss: Tuning the Network

Tips and examples for tuning the network and general operation.

Observe:



Understanding Radio Communication and Topology

When a transmission takes place from one radio module to another, the receiving radio will communicate back to the sender with an acknowledgement (ACK) packet, to indicate success. If the sender does not receive an ACK, the radio automatically engages in a series of timed retries, at set intervals. The radios use techniques like addressing and numbering of payloads to manage this, but it is all done automatically by the nrf chip, out of sight from the user.

When working over a radio network, some of these automated techniques can actually hinder data transmission to a degree. Retrying failed payloads over and over on a radio network can hinder communication for nearby nodes, or reduce throughput and errors on routing nodes.

Radios in this network are linked by **addresses** assigned to **pipes**. Each radio can listen to 6 addresses on 6 pipes, therefore each radio has a parent pipe and 4-5 child pipes, which are used to form a tree structure. Nodes communicate directly with their parent and children nodes. Any other traffic to or from a node must be routed through the network.

Topology of RF24Network

Anybody who is familiar at all with IP networking should be able to easily understand RF24Network topology. The master node can be seen as the gateway, with up to 4 directly connected nodes. Each of those nodes creates a subnet below it, with up to 4 additional child nodes. The numbering scheme can also be related to IP addresses, for purposes of understanding the topology via subnetting. Nodes can have 5 children if multicast is disabled.

Expressing RF24Network addresses in IP format

As an example, we could designate the master node in theory, as Address `10.10.10.10`

- The children nodes of the master would be `10.10.10.1`, `10.10.10.2`, `10.10.10.3`, `10.10.10.4` and `10.10.10.5`
- The children nodes of `10.10.10.1` would be `10.10.1.1`, `10.10.2.1`, `10.10.3.1`, `10.10.4.1` and `10.10.5.1`

In RF24Network, the master is just `00`

- Children of master are `01`, `02`, `03`, `04`, `05`
- Children of `01` are `011`, `021`, `031`, `041`, `051`

Multicast

Multicast is enabled by default, which limits the master node to 5 child pipes and other nodes to 4. Nodes are arranged in multicast 'levels' with the master node being level 0, nodes 01-05 are level 1, nodes n1-n5 are level 2, and so on. The multicast level of each node can be configured as desired by the user, or multicast can be disabled by editing RF24Network_config.h. For example, if all nodes are in range of the master node, all nodes can be configured to use multicast level 1, allowing the master node to contact all of them by sending a single payload. Multicasting is also used by the RF24Mesh layer for dynamic addressing requests.

Routing

Routing of traffic is handled invisibly to the user, by the network layer. If the network addresses are assigned in accordance with the physical layout of the network, nodes will route traffic automatically as required. Users simply constuct a header containing the appropriate destination address, and the network will forward it through to the correct node. Individual nodes only route individual fragments, so if using fragmentation, routing nodes do not need it enabled, unless sending or receiving fragmented payloads themselves.

If routing data between parent and child nodes (marked by direct links on the topology image above) the network uses built-in acknowledgement and retry functions of the chip to prevent data loss. When payloads are sent to other nodes, they need to be routed. Routing is managed using a combination of built in ACK requests, and software driven network ACKs. This allows all routing nodes to forward data very quickly, with only the final routing node confirming delivery and sending back an acknowledgement.

Example: Node 00 sends to node 01. The nodes will use the built in auto-retry and auto-ack functions.

Example: Node 00 sends to node 011. Node 00 will send to node 01 as before. Node 01 will forward the message to 011. If delivery was successful, node 01 will also forward a message back to node 00, indicating success.

Old Functionality: Node 00 sends to node 011 using auto-ack. Node 00 first sends to 01, 01 acknowledges. Node 01 forwards the payload to 011 using auto-ack. If the payload fails between 01 and 011, node 00 has no way of knowing.

Note: When retrying failed payloads that have been routed, there is a chance of duplicate payloads if the network-ack is not successful. In this case, it is left up to the user to manage retries and filtering of duplicate payloads.

Acknowledgements can and should be managed by the application or user. If requesting a response from another node, an acknowledgement is not required, so a user defined type of 0-64 should be used, to prevent the network from responding with an acknowledgement. If not requesting a response, and wanting to know if the payload was successful or not, users can utilize header types 65-127.

Tuning Overview

The RF24 radio modules are generally only capable of either sending or receiving data at any given time, but have built-in auto-retry mechanisms to prevent the loss of data. These values are adjusted automatically by the library on startup, but can be further adjusted to reduce data loss, and thus increase throughput of the network. This page is intended to provide a general overview of its operation within the context of the network library, and provide guidance for adjusting these values.

Auto-Retry Timing

The core radio library provides the functionality of adjusting the internal auto-retry interval of the radio modules. In the network configuration, the radios can be set to automatically retry failed transmissions at intervals ranging anywhere from 500us (0.5ms) up to 4000us (4ms). When operating any number of radios larger than two, it is important to stagger the assigned intervals, to prevent the radios from interfering with each other at the radio frequency (RF) layer.

The library should provide fairly good working values, as it simply staggers the assigned values within groups of radios in direct communication. This value can be set manually by calling `radio.setRetries(X, 15);` and adjusting the value of X from 1 to 15 (steps of 250us).

Auto-Retry Count and Extended Timeouts

The core radio library also provides the ability to adjust the internal auto-retry count of the radio modules. The default setting is 15 automatic retries per payload, and can be extended by configuring the network.txTimeout variable. This default retry count should generally be left at 15, as per the example in the above section. An interval/retry setting of (15,15) will provide 15 retrys at intervals of 4ms, taking up to 60ms per payload. The library now provides staggered timeout periods by default, but they can also be adjusted on a per-node basis.

The txTimeout variable is used to extend the retry count to a defined duration in milliseconds. See the network.txTimeout variable. Timeout periods of extended duration (500+) will generally not help when payloads are failing due to data collisions, it will only extend the duration of the errors. Extended duration timeouts should generally only be configured on leaf nodes that do not receive data.

Scenarios

Example 1

Network with master node and three leaf nodes that send data to the master node. None of the leaf nodes need to receive data.

1. Master node uses default configuration
2. Leaf nodes can be configured with extended timeout periods to ensure reception by the master.
3. The following configuration will provide a reduction in errors, as the timeouts have been extended and are staggered between devices.

```
Leaf 01: network.txTimeout = 500;
Leaf 02: network.txTimeout = 573;
Leaf 03: network.txTimeout = 653;
```

Example 2

Network with master node and three leaf nodes that send data to the master node. The second leaf node needs to receive configuration data from the master at set intervals of 1 second, and send data back to the master node. The other leaf nodes will send basic sensor information every few seconds, and a few dropped payloads will not affect the operation greatly.

1. Master node configured with extended timeouts of 0.5 seconds, and increased retry delay:

```
radio.setRetries(11, 15);
network.txTimeout = 500;
```

2. Second leaf node configured with a similar timeout period and retry delay:

```
radio.setRetries(8, 15);
network.txTimeout = 553;
```

3. First and third leaf nodes configured with default timeout periods or slightly increased timout periods.

10.5.5 Comparison to ZigBee

This network layer is influenced by the design of ZigBee, but does not implement it directly.

Which is better?

ZigBee is a much more robust, feature-rich set of protocols, with many different vendors providing compatible chips.

RF24Network is cheap. While ZigBee radios are well over $20, nRF24L01 modules can be found for under $2.

Similiarities & Differences

Here are some comparisons between RF24Network and ZigBee.

- Both networks support Star and Tree topologies. Only Zigbee supports a true mesh.
- In ZigBee networks, only leaf nodes can sleep
- ZigBee nodes are configured using AT commands, or a separate Windows application. RF24 nodes are configured by recompiliing the firmware or writing to EEPROM.
- A paper was written comparing the performance of Zigbee vs nRF24l01+, see TMRh20s Blog for a detailed overview.

Node Naming

- Leaf node: A node at the outer edge of the network with no children. ZigBee calls it an End Device node.
- Relay node: A node which has both parents and children, and relays messages from one to the other. ZigBee calls it a Router.
- Base node. The top of the tree node with no parents, only children. Typically this node will bridge to another kind of network like Ethernet. ZigBee calls it a Co-ordinator node.

10.6 Reserved System Message Types

The network will determine whether to automatically acknowledge payloads based on their general *type*.

- **User types** (1 - 127) 1 - 64 will NOT be acknowledged
- **System types** (128 - 255) 192 - 255 will NOT be acknowledged

System types can also contain message data.

NETWORK_ADDR_RESPONSE 128
A *NETWORK_ADDR_RESPONSE* type is utilized to manually route custom messages containing a single RF24Network address.

Used by RF24Mesh

If a node receives a message of this type that is directly addressed to it, it will read the included message, and forward the payload on to the proper recipient.

This allows nodes to forward multicast messages to the master node, receive a response, and forward it back to the requester.

NETWORK_PING 130
Messages of type NETWORK_PING will be dropped automatically by the recipient. A NETWORK_ACK or automatic radio-ack will indicate to the sender whether the payload was successful. The time it takes to successfully send a NETWORK_PING is the round-trip-time.

EXTERNAL_DATA_TYPE 131
External data types are used to define messages that will be passed to an external data system. This allows RF24Network to route and pass any type of data, such as TCP/IP frames, while still being able to utilize standard RF24Network messages etc.

- **Linux**

  Linux devices (defined with `RF24_LINUX` macro) will buffer all data types in the user cache.

- **Arduino/AVR/Etc**

  Data transmitted with the type set to EXTERNAL_DATA_TYPE will not be loaded into the user cache.

External systems can extract external data using the following process, while internal data types are cached in the user buffer, and accessed using network.read() :

```
uint8_t return_type = network.update();
if(return_type == EXTERNAL_DATA_TYPE){
    uint16_t size = network.frag_ptr->message_size;
    memcpy(&myDataBuffer, network.frag_ptr->message_buffer, network.frag_ptr->
↪message_size);
}
```

NETWORK_FIRST_FRAGMENT 148
Messages of this type designate the first of two or more message fragments, and will be re-assembled automatically.

NETWORK_MORE_FRAGMENTS 149
Messages of this type indicate a fragmented payload with two or more message fragments.

NETWORK_LAST_FRAGMENT 150
Messages of this type indicate the last fragment in a sequence of message fragments. Messages of this type do not receive a *NETWORK_ACK*

NETWORK_ACK 193
Messages of this type signal the sender that a network-wide transmission has been completed.

- **Not fool-proof**

  RF24Network does not directly have a built-in transport layer protocol, so message delivery is not 100% guaranteed. Messages can be lost via corrupted dynamic payloads, or a NETWORK_ACK can fail (despite successful transmission of the message).

- **Traffic analysis**

  NETWORK_ACK messages can be utilized as a traffic/flow control mechanism. Transmitting nodes that emit NETWORK_ACK qualifying messages will be forced to wait, before sending additional data, until the payload is transmitted across the network and acknowledged.

- **Different from Radio ACK Packets**

  In the event that the transmitting device will be sending directly to a parent or child node, a NETWORK_ACK is not required. This is because the radio's auto-ack feature is utilized for connections between directly related network nodes. For example: nodes `01` and `011` use the radio's auto-ack feature for transmissions between them, but nodes `01` and `02` do not use the radio's auto-ack feature for transmissions between them as messages will be routed through other nodes.

  Multicasted messages do use the radio's auto-ack feature because of the hardware limitations of nRF24L01 transceivers. This applies to all multicasted messages (directly related nodes or otherwise).

  Remark Remember, user messages types with a decimal value of 64 or less will not be acknowledged across the network via NETWORK_ACK messages.

Note: NETWORK_ACK messages are only sent by the last node in the route to a target node. ie: When node `00` sends an instigating message to node `011`, node `01` will send the NETWORK_ACK message to `00` upon sucessful delivery of instigating message to node `011`.

NETWORK_POLL 194
Used by RF24Mesh

Messages of this type are used with multi-casting , to find active/available nodes. Any node receiving a NETWORK_POLL sent to a multicast address will respond directly to the sender with a blank message, indicating the address of the available node via the header.

NETWORK_REQ_ADDRESS 195
Used by RF24Mesh

Messages of this type are used to request information from the master node, generally via a unicast (direct) write. Any (non-master) node receiving a message of this type will manually forward it to the master node using a normal network write.

10.7 Examples

10.7.1 Arduino Examples

helloworld_tx.ino

```
/**
 * Copyright (C) 2012 James Coliz, Jr. <maniacbug@ymail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * Update 2014 - TMRh20
 */

/**
 * Simplest possible example of using RF24Network
 *
 * TRANSMITTER NODE
 * Every 2 seconds, send a payload to the receiver node.
 */

#include <SPI.h>
#include <RF24.h>
#include <RF24Network.h>

RF24 radio(7, 8);                    // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);          // Network uses that radio

const uint16_t this_node = 01;       // Address of our node in Octal format
const uint16_t other_node = 00;      // Address of the other node in Octal format

const unsigned long interval = 2000; // How often (in ms) to send 'hello world' to the other unit

unsigned long last_sent;             // When did we last send?
unsigned long packets_sent;          // How many have we sent already


struct payload_t {                   // Structure of our payload
  unsigned long ms;
  unsigned long counter;
};

void setup(void) {
  Serial.begin(115200);
  while (!Serial) {
    // some boards need this because of native USB capability
  }
```

(continues on next page)

(continued from previous page)

```
  Serial.println(F("RF24Network/examples/helloworld_tx/"));

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);
}

void loop() {

  network.update(); // Check the network regularly

  unsigned long now = millis();

  // If it's time to send a message, send it!
  if (now - last_sent >= interval) {
    last_sent = now;

    Serial.print(F("Sending... "));
    payload_t payload = { millis(), packets_sent++ };
    RF24NetworkHeader header(/*to node*/ other_node);
    bool ok = network.write(header, &payload, sizeof(payload));
    Serial.println(ok ? F("ok.") : F("failed."));
  }
}
```

helloworld_rx.ino

```
/**
 * Copyright (C) 2012 James Coliz, Jr. <maniacbug@ymail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * Update 2014 - TMRh20
 */

/**
 * Simplest possible example of using RF24Network,
 *
 * RECEIVER NODE
 * Listens for messages from the transmitter and prints them out.
 */

#include <SPI.h>
```

(continues on next page)

(continued from previous page)

```
#include <RF24.h>
#include <RF24Network.h>


RF24 radio(7, 8);               // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);     // Network uses that radio
const uint16_t this_node = 00;  // Address of our node in Octal format (04, 031, etc)
const uint16_t other_node = 01; // Address of the other node in Octal format

struct payload_t {              // Structure of our payload
  unsigned long ms;
  unsigned long counter;
};


void setup(void) {
  Serial.begin(115200);
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/helloworld_rx/"));

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);
}

void loop(void) {

  network.update();                  // Check the network regularly

  while (network.available()) {      // Is there anything ready for us?

    RF24NetworkHeader header;        // If so, grab it and print it out
    payload_t payload;
    network.read(header, &payload, sizeof(payload));
    Serial.print(F("Received packet: counter="));
    Serial.print(payload.counter);
    Serial.print(F(", origin timestamp="));
    Serial.println(payload.ms);
  }
}
```

helloworld_tx_advanced.ino

```
/**
 * Copyright (C) 2020 TMRh20(tmrh20@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 */

/**
 * More advanced example of using RF24Network:
 * Fragmentation and Reassembly:
 * - nrf24l01+ radios can tx/rx 32 bytes of data per transmission
 * - RF24Network will fragment and re-assemble payloads of any size
 * Demonstrates use of differing sized payloads using peek() function
 *
 * TRANSMITTER NODE
 * Every X milliseconds, send a payload to the receiver node.
 */

#include "printf.h"
#include <RF24.h>
#include <RF24Network.h>

RF24 radio(7, 8);                    // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);          // Network uses that radio

const uint16_t this_node = 01;       // Address of our node in Octal format
const uint16_t other_node = 00;      // Address of the other node in Octal format

const unsigned long interval = 500;  //ms  // How often to send 'hello world to the other unit

unsigned long last_sent;             // When did we last send?

/**** Create a large array for data to be sent ****
* MAX_PAYLOAD_SIZE is defined in RF24Network_config.h
* Payload sizes of ~1-2 KBytes or more are practical when radio conditions are good
*/
uint8_t dataBuffer[MAX_PAYLOAD_SIZE];

void setup(void) {
  Serial.begin(115200);
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/helloworld_tx_advanced/"));
  printf_begin(); // needed for RF24* libs' internal printf() calls

  if (!radio.begin()) {
```

(continues on next page)

(continued from previous page)

```
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);
  radio.printDetails();

  // Load our data buffer with numbered data
  for (uint16_t i = 0; i < MAX_PAYLOAD_SIZE; i++) {
    dataBuffer[i] = i % 256; //Ensure the max value is 255
  }
}

uint16_t sizeofSend = 0; //Variable to indicate how much data to send
bool stopSending = 0;    //Used to stop/start sending of data

void loop() {

  //User input anything via Serial to stop/start data transmission
  if (Serial.available()) {
    Serial.read();
    stopSending = !stopSending;
  }

  network.update();                          // Check the network regularly

  unsigned long now = millis();              // If it's time to send a message, send it!
  if ( now - last_sent >= interval && !stopSending ) {
    last_sent = now;
    Serial.print(F("Sending size "));
    Serial.print(sizeofSend);

    // Fragmentation/reassembly is transparent. Just send payloads as usual.
    RF24NetworkHeader header(/*to node*/ other_node);
    bool ok = network.write(header, &dataBuffer, sizeofSend++);

    // If the size of data to be sent is larger than max payload size, reset at 0
    if (sizeofSend  > MAX_PAYLOAD_SIZE) {
      sizeofSend  = 0;
    }

    Serial.println(ok ? F(" ok.") : F(" failed."));
  }
}
```

helloworld_rx_advanced.ino

```
/**
 * Copyright (C) 2020 TMRh20(tmrh20@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 */

/**
 * More advanced example of using RF24Network:
 * Fragmentation and Reassembly:
 * - nrf24l01+ radios can tx/rx 32 bytes of data per transmission
 * - RF24Network will fragment and re-assemble payloads of any size
 * Demonstrates use of differing sized payloads using peek() function
 *
 * RECEIVER NODE
 * Every X milliseconds, send a payload to the receiver node.
 */

#include "printf.h"
#include <RF24.h>
#include <RF24Network.h>

RF24 radio(7, 8);                // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);      // Network uses that radio
const uint16_t this_node = 00;   // Address of our node in Octal format ( 04,031, etc)
const uint16_t other_node = 01;  // Address of the other node in Octal format

/**** Create a large array for data to be received ****
* MAX_PAYLOAD_SIZE is defined in RF24Network_config.h
* Payload sizes of ~1-2 KBytes or more are practical when radio conditions are good
*/
uint8_t dataBuffer[MAX_PAYLOAD_SIZE]; //MAX_PAYLOAD_SIZE is defined in RF24Network_config.h


void setup(void) {

  Serial.begin(115200);
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/helloworld_rx_advanced/"));

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
```

(continues on next page)

(continued from previous page)

```
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);

  printf_begin();       // needed for RF24* libs' internal printf() calls
  radio.printDetails(); // requires printf support
}

// Variable for calculating how long between RX
uint32_t timeBetweenPackets = 0;

void loop(void) {

  network.update();                                  // Check the network regularly

  while (network.available()) {                      // Is there anything ready for us?

    RF24NetworkHeader header;                        // If so, grab it and print it out
    uint16_t payloadSize = network.peek(header);     // Use peek() to get the size of the␣
↪payload
    network.read(header, &dataBuffer, payloadSize); // Get the data
    Serial.print("Received packet, size ");          // Print info about received data
    Serial.print(payloadSize);
    Serial.print("(");
    Serial.print(millis() - timeBetweenPackets);
    Serial.println("ms since last)");
    timeBetweenPackets = millis();

    // Uncomment below to print the entire payload
    /*
    for(uint32_t i = 0; i < payloadSize; i++) {
      Serial.print(dataBuffer[i]);
      Serial.print(F(": "));
      if(i % 50 == 49) {
        //Add a line break every 50 characters
        Serial.println();
      }
    }
    Serial.println();
    */
  }
}
```

Network_Ping.ino

```
/**
 * Copyright (C) 2011 James Coliz, Jr. <maniacbug@ymail.com>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 */

/**
 * Example: Network topology, and pinging across a tree/mesh network
 *
 * Using this sketch, each node will send a ping to every other node in the network every few seconds.
 * The RF24Network library will route the message across the mesh to the correct node.
 *
 * This sketch is greatly complicated by the fact that at startup time, each
 * node (including the base) has no clue what nodes are alive.  So,
 * each node builds an array of nodes it has heard about.  The base
 * periodically sends out its whole known list of nodes to everyone.
 *
 * To see the underlying frames being relayed, compile RF24Network with
 * #define SERIAL_DEBUG.
 *
 * Update: The logical node address of each node is set below, and are grouped in twos for demonstration.
 * Number 0 is the master node. Numbers 1-2 represent the 2nd layer in the tree (02,05).
 * Number 3 (012) is the first child of number 1 (02). Number 4 (015) is the first child of number 2.
 * Below that are children 5 (022) and 6 (025), and so on as shown below
 * The tree below represents the possible network topology with the addresses defined lower down
 *
 *     Addresses/Topology                          Node Numbers  (To simplify address assignment in this demonstration)
 *             00                  - Master Node         ( 0 )
 *          02  05                 - 1st Level children ( 1,2 )
 *    32 22 12    15 25 35 45    - 2nd Level children (7,5,3-4,6,8)
 *
 * eg:
 * For node 4 (Address 015) to contact node 1 (address 02), it will send through node 2 (address 05) which relays the payload
 * through the master (00), which sends it through to node 1 (02). This seems complicated, however, node 4 (015) can be a very
 * long way away from node 1 (02), with node 2 (05) bridging the gap between it and the master node.
 *
 * To use the sketch, upload it to two or more units and set the NODE_ADDRESS below. If configuring only a few
 * units, set the addresses to 0,1,3,5... to configure all nodes as children to each other. If using many nodes,
 * it is easiest just to increment the NODE_ADDRESS by 1 as the sketch is uploaded to each device.
```

(continues on next page)

(continued from previous page)

```
 */

#include <avr/pgmspace.h>
#include "printf.h"
#include <SPI.h>
#include <RF24.h>
#include <RF24Network.h>

/**********************************************************************
************* Set the Node Address ***********************************
/**********************************************************************/

// These are the Octal addresses that will be assigned
const uint16_t node_address_set[10] = { 00, 02, 05, 012, 015, 022, 025, 032, 035, 045 };

// 0 = Master
// 1-2 (02, 05)   = Children of Master(00)
// 3,5 (012, 022) = Children of (02)
// 4,6 (015, 025) = Children of (05)
// 7   (032)      = Child of (02)
// 8,9 (035, 045) = Children of (05)

uint8_t NODE_ADDRESS = 0;  // Use numbers 0 through to select an address from the array

/**********************************************************************/
/**********************************************************************/


RF24 radio(7, 8);                               // CE & CS pins to use (Using 7,8 on Uno,
↪Nano)
RF24Network network(radio);

uint16_t this_node;                             // Our node address

const unsigned long interval = 1000; // ms       // Delay manager to send pings␣
↪regularly.
unsigned long last_time_sent;


const short max_active_nodes = 10;              // Array of nodes we are aware of
uint16_t active_nodes[max_active_nodes];
short num_active_nodes = 0;
short next_ping_node_index = 0;


bool send_T(uint16_t to);                       // Prototypes for functions to send &␣
↪handle messages
bool send_N(uint16_t to);
void handle_T(RF24NetworkHeader& header);
void handle_N(RF24NetworkHeader& header);
void add_node(uint16_t node);

```

(continues on next page)

(continued from previous page)

```

void setup() {

  Serial.begin(115200);
  printf_begin(); // needed for RF24* libs' internal printf() calls
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/meshping/"));

  this_node = node_address_set[NODE_ADDRESS];            // Which node are we?

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setPALevel(RF24_PA_HIGH);
  radio.setChannel(100);
  network.begin(/*node address*/ this_node);
}

void loop() {

  network.update();                                      // Pump the network regularly

  while (network.available()) {                        // Is there anything ready for us?

    RF24NetworkHeader header;                            // If so, take a look at it
    network.peek(header);


    switch (header.type) {                              // Dispatch the message to the␣
↪correct handler.
      case 'T':
        handle_T(header);
        break;
      case 'N':
        handle_N(header);
        break;
      default:
        Serial.print(F("*** WARNING *** Unknown message type "));
        Serial.println(header.type);
        network.read(header, 0, 0);
        break;
    };
  }


  unsigned long now = millis();                          // Send a ping to the next node␣
↪every 'interval' ms
```

(continues on next page)

(continued from previous page)

```
  if (now - last_time_sent >= interval) {
    last_time_sent = now;


    uint16_t to = 00;                                   // Who should we send to? By default, send to base


    if (num_active_nodes) {                             // Or if we have active nodes,
      to = active_nodes[next_ping_node_index++];        // Send to the next active node
      if (next_ping_node_index > num_active_nodes) { // Have we rolled over?
        next_ping_node_index = 0;                     // Next time start at the beginning
        to = 00;                                      // This time, send to node 00.
      }
    }

    bool ok;

    if (this_node > 00 || to == 00) {                   // Normal nodes send a 'T' ping
      ok = send_T(to);
    } else {                                               // Base node sends the current active nodes out
      ok = send_N(to);
    }

    if (ok) {                                             // Notify us of the result
      Serial.print(millis());
      Serial.println(F(": APP Send ok"));
    } else {
      Serial.print(millis());
      Serial.println(F(": APP Send failed"));
      last_time_sent -= 100;                              // Try sending at a different time next time
    }
  }


  //  delay(50);                          // Delay to allow completion of any serial printing
  //  if(!network.available()){
  //      network.sleepNode(2,0);         // Sleep this node for 2 seconds or a payload is received (interrupt 0 triggered), whichever comes first
  //  }
}

/**
 * Send a 'T' message, the current time
 */
bool send_T(uint16_t to) {
  RF24NetworkHeader header(/*to node*/ to, /*type*/ 'T' /*Time*/);

  // The 'T' message that we send is just a ulong, containing the time
```

(continues on next page)

(continued from previous page)

```
  unsigned long message = millis();
  Serial.println(F("---------------------------------"));
  Serial.print(millis());
  Serial.print(F(": APP Sending "));
  Serial.print(message);
  Serial.print(F(" to "));
  Serial.print(to);
  Serial.println(F("..."));
  return network.write(header, &message, sizeof(unsigned long));
}

/**
 * Send an 'N' message, the active node list
 */
bool send_N(uint16_t to) {
  RF24NetworkHeader header(/*to node*/ to, /*type*/ 'N' /*Time*/);

  Serial.println(F("---------------------------------"));
  Serial.print(millis());
  Serial.print(F(": APP Sending active nodes to "));
  Serial.print(to);
  Serial.println(F("..."));
  return network.write(header, active_nodes, sizeof(active_nodes));
}

/**
 * Handle a 'T' message
 * Add the node to the list of active nodes
 */
void handle_T(RF24NetworkHeader& header) {

  unsigned long message;                                                    ␣
↪     // The 'T' message is just a ulong, containing the time
  network.read(header, &message, sizeof(unsigned long));
  Serial.print(millis());
  Serial.print(F(": APP Received "));
  Serial.print(message);
  Serial.print(F(" from "));
  Serial.println(header.from_node);

  if (header.from_node != this_node || header.from_node > 00)               ␣
↪    // If this message is from ourselves or the base, don't bother adding it to the␣
↪active nodes.
    add_node(header.from_node);
}

/**
 * Handle an 'N' message, the active node list
 */
void handle_N(RF24NetworkHeader& header) {
  static uint16_t incoming_nodes[max_active_nodes];

```

(continues on next page)

(continued from previous page)

```
  network.read(header, &incoming_nodes, sizeof(incoming_nodes));
  Serial.print(millis());
  Serial.print(F(": APP Received nodes from "));
  Serial.println(header.from_node);

  int i = 0;
  while ( i < max_active_nodes && incoming_nodes[i] > 00 )
    add_node(incoming_nodes[i++]);
}

/**
 * Add a particular node to the current list of active nodes
 */
void add_node(uint16_t node) {

  short i = num_active_nodes;                          // Do we already know about this node?
  while (i--) {
    if (active_nodes[i] == node)
      break;
  }

  if (i == -1 && num_active_nodes < max_active_nodes) { // If not, add it to the table
    active_nodes[num_active_nodes++] = node;
    Serial.print(millis());
    Serial.print(F(": APP Added "));
    Serial.print(node);
    Serial.println(F(" to list of active nodes."));
  }
}
```

Network_Ping_Sleep.ino

This example demonstrates how nodes on the network utilize sleep mode to conserve power. For example, the radio itself will draw about 13.5mA in receive mode. In sleep mode, it will use as little as 22ua (.000022mA) of power when not actively transmitting or receiving data. In addition, the Arduino is powered down as well, dropping network power consumption dramatically compared to previous capabilities.

Note: Sleeping nodes generate traffic that will wake other nodes up. This may be mitigated with further modifications. Sleep payloads are currently always routed to the master node, which will wake up intermediary nodes. Routing nodes can be configured to go back to sleep immediately.

The displayed `millis()` count will give an indication of how much a node has been sleeping compared to the others, as `millis()` will not increment while a node sleeps.

- Using this sketch, each node will send a ping to every other node in the network every few seconds.
- The RF24Network library will route the message across the mesh to the correct node.

```
/**
 * Copyright (C) 2011 James Coliz, Jr. <maniacbug@ymail.com>
```

(continues on next page)

(continued from previous page)

```
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * 2014 - TMRh20: New sketch included with updated library
 */

/**
 * Example: Network topology, and pinging across a tree/mesh network with sleeping nodes
 *
 * Using this sketch, each node will send a ping to every other node in the network every few seconds.
 * The RF24Network library will route the message across the mesh to the correct node.
 *
 * This sketch demonstrates the new functionality of nodes sleeping in STANDBY-I mode. In receive mode,
 * the radio will draw about 13.5 mA. In STANDBY-I mode, the radio draws .000022mA, and is able to awake
 * when payloads are received.
 *
 * How it Works:
 * The enhanced sleep mode utilizes the ACK payload functionality, as radios that are in Primary Transmitter
 * mode (PTX) are able to receive ACK payloads while in STANDBY-I mode.
 * 1. The radio is configured to use Dynamic Payloads and ACK payloads with Auto-Ack enabled
 * 2. The radio enters PTX mode and attaches an interrupt handler to the radio interrupt input pin (pin 2)
 * 3. The radio uses the Watchdog Timer to awake at set 1 second intervals in this example
 * 4. Every interval, it sends out a 'sleep' payload and goes back to sleep. Incoming payloads will then be treated as ACK payloads, while the radio remains in STANDBY-I mode.
 * 5. If an interrupt is triggered, the radio wakes up
 * 6. When a message is sent to the sleeping node, the interrupt triggers a wake up, the MCU
 * grabs the payload, and switches back to receive mode in case more data is on its way.
 *
 * The node (Arduino) power use can be reduced further by disabling unnessessary systems via the Power Reduction Register(s) (PRR).
 */

#include <avr/pgmspace.h>
#include <avr/sleep.h>
#include <avr/power.h>
#include "printf.h"
#include <SPI.h>
#include <RF24.h>
#include <RF24Network.h>


```

(continues on next page)

(continued from previous page)

```
/**********************************************************************
************* Set the Node Address *************************************
**********************************************************************/

// These are the Octal addresses that will be assigned
const uint16_t node_address_set[10] = { 00, 02, 05, 012, 015, 022, 025, 032, 035, 045 };

// 0 = Master
// 1-2  (02, 05)   = Children of Master(00)
// 3, 5 (012, 022) = Children of (02)
// 4, 6 (015, 025) = Children of (05)
// 7    (032)      = Child of (02)
// 8, 9 (035, 045) = Children of (05)

uint8_t NODE_ADDRESS = 1; // Use numbers 0 through 9 to select an address from the array

/**********************************************************************/
/**********************************************************************/


RF24 radio(7, 8);                           // CE & CS pins to use (Using 7,8 on Uno,Nano)
RF24Network network(radio);

uint16_t this_node;                         // Our node address

const unsigned long interval = 1000;        // Delay manager to send pings regularly (in ms). Because of sleepNode(), this is largely irrelevant.
unsigned long last_time_sent;

const short max_active_nodes = 10;          // Array of nodes we are aware of
uint16_t active_nodes[max_active_nodes];
short num_active_nodes = 0;
short next_ping_node_index = 0;


bool send_T(uint16_t to);                   // Prototypes for functions to send & handle messages
bool send_N(uint16_t to);
void handle_T(RF24NetworkHeader& header);
void handle_N(RF24NetworkHeader& header);
void add_node(uint16_t node);


//This is for sleep mode. It is not really required, as users could just use the number 0 through 10
typedef enum { wdt_16ms = 0, wdt_32ms, wdt_64ms, wdt_128ms, wdt_250ms, wdt_500ms, wdt_1s, wdt_2s, wdt_4s, wdt_8s } wdt_prescalar_e;

unsigned long awakeTime = 500;                          // How long in ms the radio will stay awake after leaving sleep mode
unsigned long sleepTimer = 0;                           // Used to keep track of how long the system has been awake
```

(continues on next page)

(continued from previous page)

```
90
91  void setup() {
92
93    Serial.begin(115200);
94    printf_begin(); // needed for RF24* libs' internal printf() calls
95    while (!Serial) {
96      // some boards need this because of native USB capability
97    }
98    Serial.println(F("RF24Network/examples/meshping/"));
99
100   this_node = node_address_set[NODE_ADDRESS];            // Which node are we?
101
102   if (!radio.begin()) {
103     Serial.println(F("Radio hardware not responding!"));
104     while (1) {
105       // hold in infinite loop
106     }
107   }
108   radio.setPALevel(RF24_PA_HIGH);
109   radio.setChannel(100);
110   network.begin(/*node address*/ this_node);
111
112   /********************************* This is the configuration for sleep mode␣
↪************************/
113   network.setup_watchdog(wdt_1s);                        //The watchdog timer will wake␣
↪the MCU and radio every second to send a sleep payload, then go back to sleep
114 }
115
116 void loop() {
117
118   network.update();             // Pump the network regularly
119
120   while (network.available()) { // Is there anything ready for us?
121
122     RF24NetworkHeader header;   // If so, take a look at it
123     network.peek(header);
124
125
126     switch (header.type) {      // Dispatch the message to the correct handler.
127       case 'T':
128         handle_T(header);
129         break;
130       case 'N':
131         handle_N(header);
132         break;
133
134       /************** SLEEP MODE *********/
135       // Note: A 'sleep' header has been defined, and should only need to be ignored if a␣
↪node is routing traffic to itself
136       // The header is defined as:  RF24NetworkHeader sleepHeader(/*to node*/ 00, /
↪*type*/ 'S' /*Sleep*/);
137       case 'S':
```

(continues on next page)

(continued from previous page)

```
        /*This is a sleep payload, do nothing*/
        break;

      default:
        Serial.print(F("*** WARNING *** Unknown message type "));
        Serial.println(header.type);
        network.read(header, 0, 0);
        break;
    };
  }

  /************************** CALLING THE NEW SLEEP FUNCTION ************************/

  if (millis() - sleepTimer > awakeTime && NODE_ADDRESS) {
    // Want to make sure the Arduino stays awake for a little while when data comes in.
    // Do NOT sleep if master node.
    Serial.println(F("Sleep"));
    sleepTimer = millis();      // Reset the timer value
    delay(100);                 // Give the Serial print some time to finish up
    radio.stopListening();      // Switch to PTX mode. Payloads will be seen as ACK␣
↪payloads, and the radio will wake up
    network.sleepNode(8, 0);    // Sleep the node for 8 cycles of 1second intervals
    Serial.println(F("Awake"));
  }

  //Examples:
  // network.sleepNode(cycles, interrupt-pin);
  // network.sleepNode(0, 0);         // The WDT is configured in this example to sleep␣
↪in cycles of 1 second. This will sleep 1 second, or until a payload is received
  // network.sleepNode(1, 255);       // Sleep this node for 1 second. Do not wake up␣
↪until then, even if a payload is received ( no interrupt ) Payloads will be lost.

  /****  end sleep section ***/


  unsigned long now = millis();                          // Send a ping to the next node␣
↪every 'interval' ms
  if (now - last_time_sent >= interval) {
    last_time_sent = now;

    uint16_t to = 00;                                    // Who should we send to? By␣
↪default, send to base

    if (num_active_nodes) {                              // Or if we have active nodes,
      to = active_nodes[next_ping_node_index++];         // Send to the next active node
      if (next_ping_node_index > num_active_nodes) {     // Have we rolled over?
        next_ping_node_index = 0;                        // Next time start at the beginning
        to = 00;                                         // This time, send to node 00.
      }
    }

    bool ok;
```

(continues on next page)

(continued from previous page)

```
185
186     if (this_node > 00 || to == 00) {                // Normal nodes send a 'T' ping
187       ok = send_T(to);
188     } else {                                         // Base node sends the current␣
    ↪active nodes out
189       ok = send_N(to);
190     }
191
192     if (ok) {                                        // Notify us of the result
193       Serial.print(millis());
194       Serial.println(F(": APP Send ok"));
195     } else {
196       Serial.print(millis());
197       Serial.println(F(": APP Send failed"));
198       last_time_sent -= 100;                         // Try sending at a different time␣
    ↪next time
199     }
200   }
201 }
202
203 /**
204  * Send a 'T' message, the current time
205  */
206 bool send_T(uint16_t to) {
207   RF24NetworkHeader header(/*to node*/ to, /*type*/ 'T' /*Time*/);
208
209   // The 'T' message that we send is just a ulong, containing the time
210   unsigned long message = millis();
211   Serial.println(F("-----------------------------------"));
212   Serial.print(millis());
213   Serial.print(F(": APP Sending "));
214   Serial.print(message);
215   Serial.print(F(" to "));
216   Serial.print(to);
217   Serial.println(F("..."));
218   return network.write(header, &message, sizeof(unsigned long));
219 }
220
221 /**
222  * Send an 'N' message, the active node list
223  */
224 bool send_N(uint16_t to) {
225   RF24NetworkHeader header(/*to node*/ to, /*type*/ 'N' /*Time*/);
226
227   Serial.println(F("-----------------------------------"));
228   Serial.print(millis());
229   Serial.print(F(": APP Sending active nodes to "));
230   Serial.print(to);
231   Serial.println(F("..."));
232   return network.write(header, active_nodes, sizeof(active_nodes));
233 }
234
```

(continues on next page)

(continued from previous page)

```
/**
 * Handle a 'T' message
 * Add the node to the list of active nodes
 */
void handle_T(RF24NetworkHeader& header) {

  unsigned long message;                                    // The 'T' message is just a ulong, containing the time
  network.read(header, &message, sizeof(unsigned long));
  Serial.print(millis());
  Serial.print(F(": APP Received "));
  Serial.print(message);
  Serial.print(F(" from "));
  Serial.print(header.from_node);

  if (header.from_node != this_node || header.from_node > 00) // If this message is from ourselves or the base, don't bother adding it to the active nodes.
    add_node(header.from_node);
}

/**
 * Handle an 'N' message, the active node list
 */
void handle_N(RF24NetworkHeader& header)
{
  static uint16_t incoming_nodes[max_active_nodes];

  network.read(header, &incoming_nodes, sizeof(incoming_nodes));
  Serial.print(millis());
  Serial.print(F(": APP Received nodes from "));
  Serial.println(header.from_node);

  int i = 0;
  while (i < max_active_nodes && incoming_nodes[i] > 00)
    add_node(incoming_nodes[i++]);
}

/**
 * Add a particular node to the current list of active nodes
 */
void add_node(uint16_t node) {

  short i = num_active_nodes;                               // Do we already know about this node?
  while (i--) {
    if (active_nodes[i] == node)
      break;
  }

  if (i == -1 && num_active_nodes < max_active_nodes) { // If not, add it to the table
    active_nodes[num_active_nodes++] = node;
    Serial.print(millis());
```

(continues on next page)

(continued from previous page)

```
    Serial.print(F(": APP Added "));
    Serial.print(node);
    Serial.print(F(" to list of active nodes."));
  }
}
```

Network_Priority_TX.ino

```
/**
 * Copyright (C) 2020 TMRh20(tmrh20@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 */

/**
 * This sketch demonstrates handling of external data
 *
 * RF24Network contains a buffer for storing user payloads that have been received via the network.update()
 * function. If using protocols like TCP/IP over RF24Network, the memory on small devices is very limited.
 * Instead of using the user-payload buffer for such large payloads, they can be designated as an
 * EXTERNAL_DATA_TYPE in the header.type field. This allows users to prioritize these payloads, as they are
 * often very large, and would take up most or all of the user data buffer.
 *
 * The network.update function will return immediately upon receiving a payload marked as EXTERNAL_DATA_TYPE
 * Users can then process the data immediately.
 * All other payload types are handled via the network.available() and network.read() functionality.
 *
 * Functionality:
 * The TX node will send normal user data designated with header.type = 33, along with additional data
 * marked as header.type = EXTERNAL_DATA_TYPE.
 * The RX node demonstrates how to handle such data, allowing separation of standard data that is processed
 * normally vs data that needs to be passed elsewhere, like network interface for TCP/IP packets.
 * These methods are used in RF24Gateway & RF24Ethernet TCP/IP libraries for nrf24l01+.
 */

#include <RF24.h>
#include <RF24Network.h>
#include "printf.h"

```

(continues on next page)

(continued from previous page)

```
RF24 radio(7, 8);                    // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);          // Network uses that radio

const uint16_t this_node = 01;       // Address of our node in Octal format
const uint16_t other_node = 00;      // Address of the other node in Octal format

uint8_t dataBuffer[33];

void setup() {

  Serial.begin(115200);
  printf_begin(); // needed for RF24* libs' internal printf() calls
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/Network_Separation_TX/"));

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);
  radio.printDetails();

  // Load our data buffer with numbered data
  for (uint16_t i = 0; i < 33; i++) {
    dataBuffer[i] = i;
  }

}//setup


uint32_t sendTimer = 0;

/*
 * The main loop sends two types of data to be processed with different priority per the RX
 * example
 */

void loop() {

  network.update();

  if (millis() - sendTimer > 1000) {
    sendTimer = millis();

```

(continues on next page)

(continued from previous page)

```
    Serial.println(F("Sending data..."));

    // Sending of External data, which will be handled immediately
    RF24NetworkHeader header(other_node, EXTERNAL_DATA_TYPE);
    bool ok = network.write(header, &dataBuffer, 33);
    Serial.println(ok ? F("OK 1") : F("Fail 1"));

    // Sending normal user data, which may be buffered and handled later
    RF24NetworkHeader header2(other_node, 32);
    uint32_t someVariable = 1234;
    ok = network.write(header2, &someVariable, sizeof(someVariable));
    Serial.println(ok ? F("OK 2") : F("Fail 2"));
  }

  // Dummy operation to read 0 bytes from all incoming user payloads
  // Ensures the buffer doesnt fill up
  if (network.available()) {
    RF24NetworkHeader header;
    network.read(header, &dataBuffer, 0);
  }

}//loop
```

Network_Priority_RX.ino

```
/**
 * Copyright (C) 2020 TMRh20(tmrh20@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 */

/**
 * This sketch demonstrates handling of external data
 *
 * RF24Network contains a buffer for storing user payloads that have been received via␣
↪the network.update()
 * function. If using protocols like TCP/IP over RF24Network, the memory on small␣
↪devices is very limited.
 * Instead of using the user-payload buffer for such large payloads, they can be␣
↪designated as an
 * EXTERNAL_DATA_TYPE in the header.type field. This allows users to prioritize these␣
↪payloads, as they are
 * often very large, and would take up most or all of the user data buffer.
 *
 * The network.update function will return immediately upon receiving a payload marked␣
↪as EXTERNAL_DATA_TYPE
 * Users can then process the data immediately.
 * All other payload types are handled via the network.available() and network.read()␣
↪functionality.
```

(continues on next page)

(continued from previous page)

```
 *
 * Functionality:
 * The TX node will send normal user data designated with header.type = 33, along with additional data
 * marked as header.type = EXTERNAL_DATA_TYPE.
 * The RX node demonstrates how to handle such data, allowing separation of standard data that is processed
 * normally vs data that needs to be passed elsewhere, like network interface for TCP/IP packets.
 * These methods are used in RF24Gateway & RF24Ethernet TCP/IP libraries for nrf24l01+.
 */

#include "printf.h"
#include <RF24.h>
#include <RF24Network.h>

RF24 radio(7, 8);                    // nRF24L01(+) radio attached using Getting Started board

RF24Network network(radio);          // Network uses that radio

const uint16_t this_node = 00;       // Address of our node in Octal format
const uint16_t other_node = 01;      // Address of the other node in Octal format

uint32_t myVariable = 0;

void setup() {

  Serial.begin(115200);
  printf_begin(); // needed for RF24* libs' internal printf() calls
  while (!Serial) {
    // some boards need this because of native USB capability
  }
  Serial.println(F("RF24Network/examples/Network_Separation_RX/"));

  if (!radio.begin()) {
    Serial.println(F("Radio hardware not responding!"));
    while (1) {
      // hold in infinite loop
    }
  }
  radio.setChannel(90);
  network.begin(/*node address*/ this_node);
  radio.printDetails();

}//setup


uint32_t sendTimer = 0;

/* **** Create a large array for data to be received ****
 * MAX_PAYLOAD_SIZE is defined in RF24Network_config.h
```

(continues on next page)

(continued from previous page)

```
 * Payload sizes of ~1-2 KBytes or more are practical when radio conditions are good
 */
#define EXTERNAL_DATA_MAX_SIZE  MAX_PAYLOAD_SIZE

uint8_t dataBuffer[EXTERNAL_DATA_MAX_SIZE];

uint32_t userDataTimer = 0;


/*
 * The main loop behaviour demonstrates the different prioritization of handling data
 * External data is handled immediately upon reception, with the network.update() function being
 * called very regularly to handle incoming/outgoing radio traffic.
 *
 * The network.available() function is only called every 5 seconds, to simulate a busy microcontroller,
 * so the user payloads will only print out every 5 seconds
 *
 * The radio has 3, 32-byte FIFO buffers operating independantly of the MCU, and RF24Network will buffer
 * up to MAX_PAYLOAD_SIZE (see RF24Network_config.h) of user data.
 */
void loop() {

  // Immediate handling of data with header type EXTERNAL_DATA_TYPE

  if (network.update() == EXTERNAL_DATA_TYPE) {
    uint16_t size = network.frag_ptr->message_size;
    memcpy(&dataBuffer, network.frag_ptr->message_buffer, network.frag_ptr->message_size);

    // Handle the external data however...
    Serial.print(F("External Data RX, size: "));
    Serial.println(network.frag_ptr->message_size);

    for (uint16_t i = 0; i < network.frag_ptr->message_size; i++) {
      Serial.print(dataBuffer[i]);
      Serial.print(F(":"));
    }
    Serial.println();
  }


  // Use a timer to simulate a busy MCU where normal network data cannot be processed in a timely manner
  if (millis() - userDataTimer > 5000) {
    userDataTimer = millis();

    // Handling of standard RF24Network User Data
    while (network.available()) {

```

(continues on next page)

(continued from previous page)

```
      RF24NetworkHeader header;                                   // Create an empty␣
↪header
      uint16_t dataSize = network.peek(header);                   // Peek to get the␣
↪size of the data
      uint32_t someVariable;
      if (header.type = 32) {                                     // If a certain␣
↪header type is recieved
        network.read(header, &someVariable, sizeof(someVariable)); // Handle the data a␣
↪specific way
        Serial.print(F("RX User Data:\nHeader Type "));
        Serial.print(header.type);
        Serial.print(F(" Value "));
        Serial.println(someVariable);
      } else {
        // Clear the user data from the buffer if some other header type is received
        network.read(header, &someVariable, 0);
      }
    }
  }
}//loop
```

10.7.2 Linux Examples

helloworld_tx.cpp

Listing 1: examples_RPi/helloworld_rx.cpp

```
/*
 Update 2014 - TMRh20
 */

/**
 * Simplest possible example of using RF24Network,
 *
 * RECEIVER NODE
 * Listens for messages from the transmitter and prints them out.
 */

//#include <cstdlib>
#include <RF24/RF24.h>
#include <RF24Network/RF24Network.h>
#include <iostream>
#include <ctime>
#include <stdio.h>
#include <time.h>

using namespace std;

RF24 radio(22, 0); // (CE Pin, CSN Pin, [SPI Speed (in Hz)])

```

(continues on next page)

(continued from previous page)

```
RF24Network network(radio);

// Address of our node in Octal format (01,021, etc)
const uint16_t this_node = 01;

// Address of the other node
const uint16_t other_node = 00;

// How often (in milliseconds) to send a message to the `other_node`
const unsigned long interval = 2000;

unsigned long last_sent;    // When did we last send?
unsigned long packets_sent; // How many have we sent already

struct payload_t { // Structure of our payload
    unsigned long ms;
    unsigned long counter;
};

int main(int argc, char **argv)
{
    // Refer to RF24 docs or nRF24L01 Datasheet for settings

    if (!radio.begin()) {
        printf("Radio hardware not responding!\n");
        return 0;
    }

    delay(5);
    radio.setChannel(90);
    network.begin(/*node address*/ this_node);
    radio.printDetails();

    while (1) {

        network.update();
        unsigned long now = millis(); // If it's time to send a message, send it!
        if (now - last_sent >= interval) {
            last_sent = now;

            printf("Sending ..\n");
            payload_t payload = {millis(), packets_sent++};
            RF24NetworkHeader header(/*to node*/ other_node);
            bool ok = network.write(header, &payload, sizeof(payload));
            printf("%s.\n", ok ? "ok" : "failed");
        }
    }

    return 0;
}
```

helloworld_rx.cpp

Listing 2: examples_RPi/helloworld_rx.cpp

```
/*
 Update 2014 - TMRh20
 */

/**
 * Simplest possible example of using RF24Network,
 *
 * RECEIVER NODE
 * Listens for messages from the transmitter and prints them out.
 */

#include <RF24/RF24.h>
#include <RF24Network/RF24Network.h>
#include <iostream>
#include <ctime>
#include <stdio.h>
#include <time.h>


// CE Pin, CSN Pin, SPI Speed (Hz)
RF24 radio(22, 0);

RF24Network network(radio);

// Address of our node in Octal format
const uint16_t this_node = 00;

// Address of the other node in Octal format (01, 021, etc)
const uint16_t other_node = 01;

struct payload_t { // Structure of our payload
    unsigned long ms;
    unsigned long counter;
};

int main(int argc, char **argv)
{
    // Refer to RF24 docs or nRF24L01 Datasheet for settings

    if (!radio.begin()) {
        printf("Radio hardware not responding!\n");
        return 0;
    }

    delay(5);
    radio.setChannel(90);
    network.begin(/*node address*/ this_node);
    radio.printDetails();

```

(continues on next page)

(continued from previous page)

```
    while (1) {

        network.update();
        while (network.available()) { // Is there anything ready for us?

            RF24NetworkHeader header; // If so, grab it and print it out
            payload_t payload;
            network.read(header, &payload, sizeof(payload));

            printf("Received payload: counter=%lu, origin timestamp=%lu\n", payload.
↪counter, payload.ms);
        }
        //sleep(2);
        delay(2000);
    }

    return 0;
}
```

10.7.3 PicoSDK Examples

helloworld_tx

See also:

defaultPins.h

Listing 3: examples_pico/helloworld_tx.cpp

```
/**
 * Simplest possible example of using RF24Network,
 *
 * TRANSMITTER NODE
 * Transmits messages to the reciever every 2 seconds.
 */
#include "pico/stdlib.h"  // printf(), sleep_ms(), to_ms_since_boot(), get_absolute_
↪time()
#include <tusb.h>         // tud_cdc_connected()
#include <RF24.h>         // RF24 radio object
#include <RF24Network.h>  // RF24Network network object
#include "defaultPins.h"  // board presumptive default pin numbers for CE_PIN and CSN_PIN

// instantiate an object for the nRF24L01 transceiver
RF24 radio(CE_PIN, CSN_PIN);

RF24Network network(radio);

// Address of our node in Octal format (01,021, etc)
const uint16_t this_node = 01;

// Address of the other node
```

(continues on next page)

(continued from previous page)

```
const uint16_t other_node = 00;

// How often (in milliseconds) to send a message to the `other_node`
const unsigned long interval = 2000;

unsigned long last_sent;    // When did we last send?
unsigned long packets_sent; // How many have we sent already

struct payload_t { // Structure of our payload
    unsigned long ms;
    unsigned long counter;
};


bool setup()
{
    // wait here until the CDC ACM (serial port emulation) is connected
    while (!tud_cdc_connected()) {
        sleep_ms(10);
    }

    // initialize the transceiver on the SPI bus
    if (!radio.begin()) {
        printf("radio hardware is not responding!!\n");
        return false;
    }

    radio.setChannel(90);
    network.begin(/*node address*/ this_node);

    // print example's introductory prompt
    printf("RF24Network/examples_pico/helloworld_tx\n");

    // For debugging info
    // radio.printDetails();       // (smaller) function that prints raw register values
    // radio.printPrettyDetails(); // (larger) function that prints human readable data

    return true;
} // setup


void loop()
{
    network.update();
    unsigned long now = to_ms_since_boot(get_absolute_time());
    if (now - last_sent >= interval) { // If it's time to send a message, send it!
        last_sent = now;

        printf("Sending ..\n");
        payload_t payload = {now, packets_sent++};
        RF24NetworkHeader header(/*to node*/ other_node);
        bool ok = network.write(header, &payload, sizeof(payload));
```

(continues on next page)

(continued from previous page)

```
        printf("%s.\n", ok ? "ok" : "failed");
    }
}

int main()
{
    stdio_init_all(); // init necessary IO for the RP2040

    while (!setup()) { // if radio.begin() failed
        // hold program in infinite attempts to initialize radio
    }
    while (true) {
        loop();
    }
    return 0; // we will never reach this
}
```

helloworld_rx

See also:

defaultPins.h

Listing 4: examples_pico/helloworld_rx.cpp

```
/**
 * Simplest possible example of using RF24Network,
 *
 * RECEIVER NODE
 * Listens for messages from the transmitter and prints them out.
 */
#include "pico/stdlib.h"  // printf(), sleep_ms(), to_ms_since_boot(), get_absolute_
↪time()
#include <tusb.h>         // tud_cdc_connected()
#include <RF24.h>         // RF24 radio object
#include <RF24Network.h>  // RF24Network network object
#include "defaultPins.h"  // board presumptive default pin numbers for CE_PIN and CSN_PIN

// instantiate an object for the nRF24L01 transceiver
RF24 radio(CE_PIN, CSN_PIN);

RF24Network network(radio);

// Address of our node in Octal format
const uint16_t this_node = 00;

// Address of the other node in Octal format (01, 021, etc)
const uint16_t other_node = 01;

struct payload_t { // Structure of our payload
    unsigned long ms;
    unsigned long counter;
```

(continues on next page)

(continued from previous page)

```
};


bool setup()
{
    // wait here until the CDC ACM (serial port emulation) is connected
    while (!tud_cdc_connected()) {
        sleep_ms(10);
    }

    // initialize the transceiver on the SPI bus
    if (!radio.begin()) {
        printf("radio hardware is not responding!!\n");
        return false;
    }

    radio.setChannel(90);
    network.begin(/*node address*/ this_node);

    // print example's introductory prompt
    printf("RF24Network/examples_pico/helloworld_rx\n");

    // For debugging info
    // radio.printDetails();       // (smaller) function that prints raw register values
    // radio.printPrettyDetails(); // (larger) function that prints human readable data

    return true;
} // setup


void loop()
{
    network.update();
    while (network.available()) { // Is there anything ready for us?
        // If so, grab it and print it out
        RF24NetworkHeader header;
        payload_t payload;
        network.read(header, &payload, sizeof(payload));

        printf("Received payload: counter=%lu, origin timestamp=%lu\n", payload.counter,␣
↪payload.ms);
    }
}

int main()
{
    stdio_init_all(); // init necessary IO for the RP2040

    while (!setup()) { // if radio.begin() failed
        // hold program in infinite attempts to initialize radio
    }
    while (true) {
```

(continues on next page)

(continued from previous page)

```
        loop();
    }
    return 0; // we will never reach this
}
```

PicoSDK Examples' Default Pins

```
// pre-chossen pins for different boards
#ifndef DEFAULTPINS_H
#define DEFAULTPINS_H

#if defined (ADAFRUIT_QTPY_RP2040)
// for this board, you can still use the Stemma QT connector as a separate I2C bus (`i2c1`)
#define CE_PIN PICO_DEFAULT_I2C_SDA_PIN  // the pin labeled SDA
#define CSN_PIN PICO_DEFAULT_I2C_SCL_PIN // the pin labeled SCL

#elif defined (PIMORONI_TINY2040)
// default SPI_SCK_PIN = 6
// default SPI_TX_PIN = 7
// default SPI_RX_PIN = 4
#define CE_PIN PICO_DEFAULT_I2C_SCL_PIN  // pin 3
#define CSN_PIN PICO_DEFAULT_SPI_CSN_PIN // pin 5


#elif defined (SPARFUN_THINGPLUS)
#define CSN_PIN 16 // the pin labeled 16
#define CE_PIN 7   // the pin labeled SCL

#else
// pins available on (ADAFRUIT_ITSYBITSY_RP2040 || ADAFRUIT_FEATHER_RP2040 || Pico_board || Sparkfun_ProMicro || SparkFun MicroMod)

#define CE_PIN 7
#define CSN_PIN 8
#endif // board detection macro defs

#endif // DEFAULTPINS_H
```

10.7.4 Python Examples

helloworld_tx.py

Listing 5: RPi/pyRF24Network/examples/helloworld_tx.py

```
"""Simplest possible example of using RF24Network in RX role.
Sends messages from to receiver.
"""
import time
```

(continues on next page)

(continued from previous page)

```
import struct
from RF24 import RF24
from RF24Network import RF24Network, RF24NetworkHeader


########### USER CONFIGURATION ###########
# See https://github.com/TMRh20/RF24/blob/master/pyRF24/readme.md
# Radio CE Pin, CSN Pin, SPI Speed
# CE Pin uses GPIO number with BCM and SPIDEV drivers, other platforms use
# their own pin numbering
# CS Pin addresses the SPI bus number at /dev/spidev<a>.<b>
# ie: RF24 radio(<ce_pin>, <a>*10+<b>); spidev1.0 is 10, spidev1.1 is 11 etc..

# Generic:
radio = RF24(22, 0)
################## Linux (BBB,x86,etc) #########################
# See http://nRF24.github.io/RF24/pages.html for more information on usage
# See http://iotdk.intel.com/docs/master/mraa/ for more information on MRAA
# See https://www.kernel.org/doc/Documentation/spi/spidev for more
# information on SPIDEV

# instantiate the network node using `radio` object
network = RF24Network(radio)

# Address of our node in Octal format (01,021, etc)
this_node = 0o1

# Address of the other node
other_node = 0o0

# How long to wait before sending the next message
interval = 2000  # in milliseconds

# initialize the radio
if not radio.begin():
    raise RuntimeError("radio hardware not responding")

radio.channel = 90

# initialize the network node
network.begin(this_node)

# radio.printDetails()
radio.printPrettyDetails()
packets_sent = 0
last_sent = 0

while 1:
    network.update()
    now = time.monotonic_ns() / 1000
    # If it's time to send a message, send it!
    if now - last_sent >= interval:
```

(continues on next page)

(continued from previous page)

```
        last_sent = now
        payload = struct.pack('<LL', now, packets_sent)
        packets_sent += 1
        ok = network.write(RF24NetworkHeader(other_node), payload)
        print("Sending %d..." % packets_sent, "ok." if ok else "failed.")
```

helloworld_rx.py

Listing 6: RPi/pyRF24Network/examples/helloworld_rx.py

```
"""Simplest possible example of using RF24Network in RX role.
Listens for messages from the transmitter and prints them out.
"""
import time
import struct
from RF24 import RF24
from RF24Network import RF24Network


########### USER CONFIGURATION ###########
# See https://github.com/TMRh20/RF24/blob/master/pyRF24/readme.md
# Radio CE Pin, CSN Pin, SPI Speed
# CE Pin uses GPIO number with BCM and SPIDEV drivers, other platforms use
# their own pin numbering
# CS Pin addresses the SPI bus number at /dev/spidev<a>.<b>
# ie: RF24 radio(<ce_pin>, <a>*10+<b>); spidev1.0 is 10, spidev1.1 is 11 etc..

# Generic:
radio = RF24(22, 0)
################## Linux (BBB,x86,etc) #########################
# See http://nRF24.github.io/RF24/pages.html for more information on usage
# See http://iotdk.intel.com/docs/master/mraa/ for more information on MRAA
# See https://www.kernel.org/doc/Documentation/spi/spidev for more
# information on SPIDEV

# instantiate the network node using `radio` object
network = RF24Network(radio)

# Address of our node in Octal format (01, 021, etc)
this_node = 0o0

# Address of the other node
other_node = 0o1

# initialize the radio
if not radio.begin():
    raise RuntimeError("radio hardware not responding")

radio.channel = 90

# initialize the network node
```

(continues on next page)

(continued from previous page)

```
network.begin(this_node)

# radio.printDetails()
radio.printPrettyDetails()

radio.startListening()  # put radio in RX mode
start = time.monotonic()
while time.monotonic() - start <= 6:  # listen for 6 seconds
    network.update()
    while network.available():
        header, payload = network.read(8)
        print("payload length ", len(payload))
        millis, number = struct.unpack('<LL', bytes(payload))
        print(
            "Received payload {} from {} to {} at (origin's timestamp) {}".format(
                number,
                oct(header.from_node),
                oct(header.to_node),
                millis,
            )
        )
        start = time.monotonic()
    time.sleep(0.05)
```

INDEX