RF24G

0.9

Generated by Doxygen 1.8.12

Contents

Chapter 1

A simple interface for the RF24 radio that abstracts thmr20's driver.

1.1 About

The nRF24L01+ wireless transceiver board allow for wireless communication between two or more radios at distances greater than Bluetooth or standard WiFi. This tutorial includes an overview of the different types or radios available in the store, wiring the radios to an Arduino, an example sketch that allows for two way communication, and finally tips and tricks to increase your success with the radios.

1.2 Purchasing

There are two versions available in the UCSB ECE store. They also can be purchased on the yourduino website (`http://www.yourduino.com/sunshop/`) The high power transceiver has amplifiers and an external antenna. It has been tested to work at ranges in excess of 350 meters. The low power transceivers have an internal antenna and work at about 20 meters. The two different types can work together. Se the tips and tricks section for more info.

1.3 Installation

This library requires thmr20's radio driver. Both can be found in the Arduino repository.

First, go to sketch→Include Library→Manage Libraries...

The library manager will show as an additional window.

Search for rf24 and select version 1.1.7 of TMRh20's RF24 Library.

Press install.

Next, add version 0.9 of the RF24G library.

Press install.

Wiring

This tutorial assumes you are using the RF24 modules sold here: `http://yourduino.com/sunshop//index.php?l=product_detail&p=489`

recouses for this tutorial are based on Terry's instructions at `https://arduino-info.wikispaces.com/Nrf24L01-2.4GHz-HowTo`

First, attach the either the high power or low power radio module to the base module

Next, connect jumpers between the Arduino and the base module using this table

More in-depth instructions can be found at `https://arduino-info.wikispaces.com/Nrf24L01-2.4GHz-HowTo`

General concepts

This library provides an abstraction layer that allows the user identify each radio by an address and each transmission as a packet.

Up to 6 radios can be used in the network, with each having a unique address: (0, 1, 2, 3, 4, 5).

Each radio is initialized using an RF24_G object, which provides the ability to read and write packets.

This library uses the built in functions of the radio to ensure guaranteed delivery; However, like any practical guaranteed transmission network, there is a timeout.

The after 30 retransmit attempts, the radio gives up and returns that it has failed to transmit a packet. More info can be seen it the RF24_G::read() docs.

The packet class is an object that contains all the necessary information to bring data to and from each radio, as well as let each radio keep track of any dropped packets.

A packet allows for any playload that is 30 bytes long. The payload can be any type or array of types.

How to use this documentation.

Read the packet and RF24_G class documentation. It provides a description of what every class and object in the library is for.b

Check the examples to understand the way the two classes are used to send data from one radio to another.

Chapter 2

RF24G

This library provides a simple way for up to 6 nRF24L01 radios to communicate with each other.

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 packet Class Reference

Public Member Functions

Packet public interface

These are the main methods you need to set, modify, and retrieve data from packets.

- packet ()
- void setAddress (uint8_t _address)
- uint8_t getAddress ()
- uint8_t getCnt ()
- void setCnt (uint8_t _cnt)
- bool addPayload (const void ∗data, const uint8_t size)
- bool readPayload (void ∗data, const uint8_t size)

5.1.1 Detailed Description

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 packet()

```
packet::packet ( )
```

Default Constructor

Creates a new instance of the packet object. The packet is blank and will need to be modified with the methods below.

5.1.3 Member Function Documentation

5.1.3.1 setAddress()

```
void packet::setAddress (
            uint8_t _address )
```

Sets the address of a packet.

If you are sending a packet, set this to set the destination of the packet.

Examples:

RF24G_Send.cpp.

5.1.3.2 getAddress()

```
uint8_t packet::getAddress ( )
```

Gets the address of a packet.

If you receive a packet, call this on the packet to get what address the packet came from.

Returns

Current packet address.

5.1.3.3 getCnt()

```
uint8_t packet::getCnt ( )
```

Gets the counter of a packet.

This is used internally by the library to set the packet counter. This is used to detect duplicate packets.

The user does not need to use this method.

Returns

Current packet counter.

Examples:

RF24G_Receive.cpp.

5.1.3.4 setCnt()

```
void packet::setCnt (
            uint8_t _cnt )
```

Sets the counter of a packet.

This is used internally by the library to set the packet counter. This is used to detect duplicate packets.

The user does not need to use this method.

5.1.3.5 addPayload()

```
bool packet::addPayload (
            const void * data,
            const uint8_t size )
```

Adds any datatype smaller than 30 bytes to the packet.

Note

> There is no way to determine what kind of datatype is in this packet without prior knowledge.
> If you want to send different types of payloads, use a struct or class similar to this packet within the payload that contains metadata on what type of data it is.

This needs the address of an object and it's size to work correctly.

```
//addPayload() example:

    int var = 23;
    if (packet.addPayload( &value, sizeof(var)) == false) {
        Serial.println("Datatype is too large!")
    }
```

Returns

> True if the size is within 30 bytes, false if it is not.

Warning

> This does not allow for you to overwrite the packet. But it is possible to overread from locations in memory that are adjacent to an object! Always use sizeof(yourObject) to prevent this.

Examples:

> RF24G_Send.cpp.

5.1.3.6 readPayload()

```
bool packet::readPayload (
             void * data,
             const uint8_t size )
```

Retrieves any datatype smaller than 30 bytes from the packet.

Note

There is no way to determine what kind of datatype is in this packet.
If you want to send multiple values, use a struct or class similar to this packet within the payload.

This needs the address of an object and it's size to work correctly.

```
//readPayload() example:

    int var;
    if (packet.readPayload( &var, sizeof(var)) == false) {
        Serial.println("Datatype is too large!")
    }
```

Note

The variable *var* will have a new value from the packet.

Returns

True if the size is within 30 bytes, false if it is not.

Warning

If you specify a size that is larger than the object you wish to write to, you can write into adjacent memory! This *probably* will crash your program and/or give you junk data in other parts of your code! Always use sizeof(yourObject) to prevent this.

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

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

- RF24G.h
- RF24G.cpp

5.2 RF24_G Class Reference

Public Member Functions

Primary public interface

These are the main methods you need to send and receive data.

- RF24_G ()
- RF24_G (uint8_t address, uint8_t _cepin, uint8_t _cspin)
- bool available ()
- bool write (const packet ∗_packet)
- bool read (packet ∗_packet)
- bool setChannel (uint8_t channel)

5.2.1 Detailed Description

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 RF24_G() [1/2]

```
RF24_G::RF24_G ( )
```

Default Constructor

Creates a new instance of the radio object. This configures tmrh20's driver to default settings. Use this if you want to instantiate the radio class, but initialize it later.

5.2.2.2 RF24_G() [2/2]

```
RF24_G::RF24_G (
             uint8_t address,
             uint8_t _cepin,
             uint8_t _cspin )
```

Constructor

Creates a new instance of the radio object. This configures tmrh20's driver. Before using, you create an instance and send in the unique pins that this chip is connected to. If you have followed the wiring diagram on the first page, the CE pin should be 7 and the CS pin should be 8.

Parameters

address	The address of tis radio instance
_cepin	The pin attached to Chip Enable (CE) pin on the RF module
_cspin	The pin attached to Chip Select (CS) pin

5.2.3 Member Function Documentation

5.2.3.1 available()

```
bool RF24_G::available ( )
```

Checks if there is a packet received packet to be read

Returns

True if a packet is available, false if not.

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

5.2.3.2 write()

```
bool RF24_G::write (
             const packet * _packet )
```

Writes data to a packet. The packet is passed by reference, this means we need to use the & operator.

```
//write() example:

     int var;
     if (radio.write( &packet) == false) {
         Serial.println("Transmission failed!")
     }
```

Returns

True if a packet was sent successfully, false if not.

Note

Just because a packet was not sent successfully, it does not mean a packet was not received by the target radio!
This could be due to the sender not receiving the confirmation that the target radio has received the packet.
This could be fixed with a 3 way handshake, but that is not supported in hardware and would be slow in software.

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

5.2.3.3 read()

```
bool RF24_G::read (
             packet * _packet )
```

Reads a packet. The packet is passed by reference, this means we need to use the & operator.

```
//read() example:

     int var;
     if (radio.read( &packet) == false) {
         Serial.println("Receive failed!")
     }
```

Returns

True if a packet was read successfully, false if not.

Examples:

RF24G_Receive.cpp, and RF24G_Send.cpp.

5.2.3.4 setChannel()

```
bool RF24_G::setChannel (
            uint8_t channel )
```

Sets the channel to use

Note

The available channels are 0-125, but channels 108+ are out of the wifi band and recommended.

Returns

True if the channel was set successfully, false if not.

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

- RF24G.h
- RF24G.cpp

Chapter 6

File Documentation

6.1 RF24G.h File Reference

A simple interface for the RF24 radio that abstracts thmr20's Driver.

```
#include "RF24.h"
```

Classes

- class packet
- class RF24_G

Macros

- #define **PACKET_CNTER** 32
- #define **MAX_NODES** 6
- #define **BASE_ADDRESS** 0xDEADBEEF00LL
- #define **TIMEOUT** 5

6.1.1 Detailed Description

A simple interface for the RF24 radio that abstracts thmr20's Driver.

Author

Caio Motta

Date

19 Sep 2016 This library provides a simple way for up to 6 nRF24L01 radios to communicate with each other.

See also

`http://tmrh20.github.io/RF24/`
`https://arduino-info.wikispaces.com/Nrf24L01-2.4GHz-HowTo`

Chapter 7

Example Documentation

7.1 RF24G_Receive.cpp

This is an example on how to receive using the RF24_G class

```
/* This is a small sketch that listens
 * for packets and forwards them back to the sender.
 */

#include <RF24G.h>
// we must instantiate the RF24_G object outside of the setup function so it is available in the loop
       function
RF24_G test;
int i = 0;
void setup() {
    Serial.begin(9600);
    // create the RF24G object with an address of 1, using pins 7 and 8
    test = RF24_G(1, 7, 8);
    // print out the details of the radio's configuration (useful for debug)
    test.radio.printDetails();
}

void loop() {
    // declare packet variable
    packet receiver;
    // declare string to place the packet payload in
    char payload[30]
    // check if the radio has any packets in the receive queue
    if (test.available() == true) {
        Serial.println("packet received!")
        // read the data into the packet
        test.read(&receiver);
        // print the packet number of the received packet
        // if these are not consecutive packets are being lost due to timeouts.
        Serial.print("count: ")
        Serial.println(receiver.getCnt());
        // print the source address of the received packet
        Serial.print("address: ")
        Serial.println(receiver.getaddress());
        // load the payload into the payload string
        receiver.readPayload(payload, 30)
        // print the payload
        Serial.print("payload: ")
        Serial.println(payload);
        // since the address in the packet object is already
        // set to the address of the receiver, it doesn't need to be changed
        // hence, we can write the packet back to the receiver
        // we may check to see if the transmission failed, if so we just drop the packet
        if (test.write(receiver) == false) {
            Serial.println("transmit back failed!");
            Serial.println("dropping packet...");
        }
    }
}
```

7.2 RF24G_Send.cpp

```
/* This sketch sends a packet with random data to another radio and waits for
 * the packet to be sent back.  It prints out the random data and the received data, which should be the
       same.
 */

#include <RF24G.h>
// We must instantiate the RF24_G object outside of the setup function so it is available in the loop
       function
RF24_G test;
void setup() {
    Serial.begin(9600);
    // create the RF24G object with an address of 4, using pins 7 and 8
    test = RF24_G(4, 7, 8);
    // print out the details of the radio's configuration (useful for debug)
    test.radio.printDetails();

}

void loop() {
    // create a random number
    uint8_t randNumber = random(300);
    // create a variable to store the received number
    uint8_t actual;
    // declare the sender packet variable
    packet sender;
    // declare the receiver packet variable
    packet receiver;
    // set the destination of the packet to address 1
    sender.setAddress(1);
    // write the payload to the packet
    sender.addPayload(&randNumber, sizeof(int));
    // print out the original payload
    Serial.print("original number:");
    Serial.println(randNumber);
    // send the packet, if it is successful try to read back the packet
    if (test.write(sender) == true) {
        // wait until a packet is received
        while (test.available() != true);
        // copy the packet into the receiver object
        test.read(&receiver);
        // copy the payload into the actual value
        receiver.readPayload(actual, sizeof(int));
        // print out the actual value received
        Serial.print("received number:");
        Serial.println(actual);

    }

}
```

Index