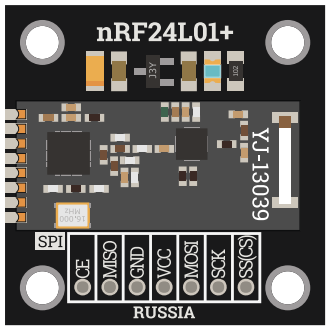
Радио модуль NRF24L01+ / PA+LNA 2.4G (Trema-модуль V2.0)



Общие сведения:

Trema радио модули NRF24L01+ 2.4G и NRF24L01+PA+LNA 2.4G предназначены для приёма и передачи данных по радиоканалу в ISM диапазоне радиочастот (Industrial, Scientific, Medical).

Преимуществом Trema радиомодулей от аналогичных модулей NRF24L01+ является то, что Trema радиомодули могут работать как от 3,3В, так и от 5В, не требуя дополнительных адаптеров для подключения к Aduino.

У модуля NRF24L01+PA+LNA 2.4G дальность действия больше чем у модуля NRF24L01+ 2.4G так как первый оснащён двумя дополнительными усилителями: «PA» (Power Amplifier) - усилитель мощности радиопередающего тракта и «LNA» (Low-Noise Amplifier) - малошумящий усилитель радиоприёмного тракта.

Оба модуля построены на базе чипа NRF24L01+ (Nordic Radio Frequency) радиочастотный приёмопередатчик производства «Nordic Semiconductor».

Модулю можно программно указать и менять в процессе работы такие параметры, как: канал (частоту), роль (приёмник или передатчик), уровень усиления мощности передатчика, скорость передачи данных по радиоканалу и многие другие параметры.

Спецификация:

- Частотный диапазон: ISM (2,400 … 2,525 ГГц).
- Доступные радиоканалы: 0-125 (от 2400 до 2525 МГц).
- Ширина одного радиоканала: 1МГц (на скорости 2 Мбит/сек используется 2 канала).
- Время переключения между каналами: 130 мкс.
- Модуляция: GFSK (Gaussian Frequency-Shift Keying).
- Девиация частоты: 156 кГц.
- Радиус действия модуля NRF24L01+: до 100 м (в пределах прямой видимости).
- Радиус действия модуля NRF24L01+PA+LNA: до 1 км (в пределах прямой видимости).
- Скорость передачи данных: 0.25, 1, 2 Мбит/с (задаётся программно).

- Мощность передатчика: -18, -12, -6, 0 дБм (задаётся программно).
- Чувствительность приемника: -82 дБм.
- Коэффициент усиления антенны: 2 дБм.
- Интерфейс: SPI.
- Напряжение питания: 3.3 или 5 В (поддерживаются оба варианта).
- Уровень логической «1»: 3,3 В (все выводы толерантны к 5 В).
- Ток потребляемый в режиме передачи данных: до 15 мА.
- Ток потребляемый в режиме приёма данных: до 21 мА.
- Ток потребляемый в режиме ожидания: до 8 мА.
- Рабочая температура: -40 ... 85 °C.
- Габариты: 30x30 мм.

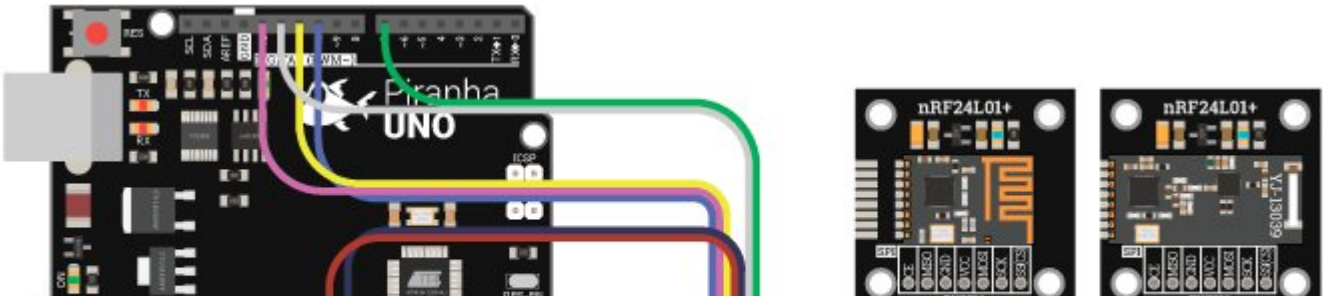
Подключение:

Модуль подключается к Arduino по шине SPI (можно использовать как аппаратную так и программную шину).

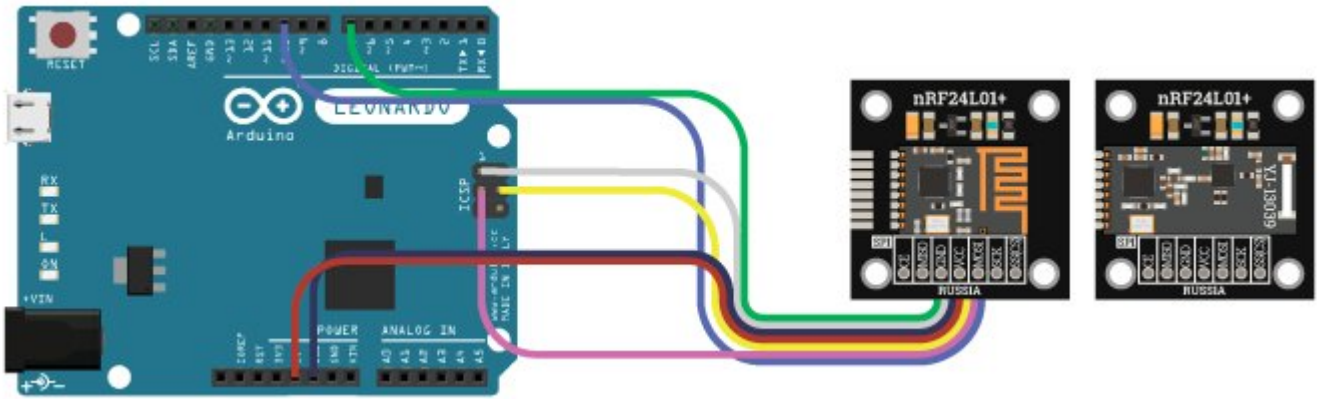
- Выводы модуля Vcc и GND подключаются к шине питания 3,3 или 5 В постоянного тока.
- Выводы модуля MISO, MOSI и SCK подключаются к одноименным выводам шины SPI на плате Arduino.
- Выводы SS (Slave Select) и CE (Chip Enable) назначаются при объявлении объекта библиотеки FR24 и подключаются к любым назначенным выводам Arduino. В перечисленных ниже примерах линии SS назначен вывод 10, а линии CE назначен вывод 7.

Способ - 1: Используя провода и плату контроллера

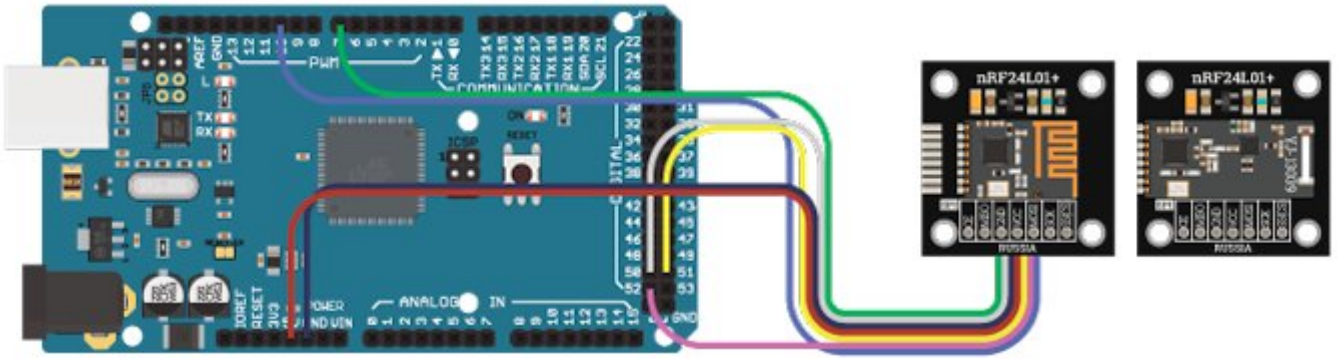
Используя провода «Папа - Мама», подключаем напрямую к контроллеру Piranha UNO или Arduino UNO:



Используя провода «Папа - Мама», подключаем напрямую к контроллеру Arduino Leonardo. Этот способ подключения подойдёт и для плат Piranha UNO, Arduino UNO и Arduino Mega:

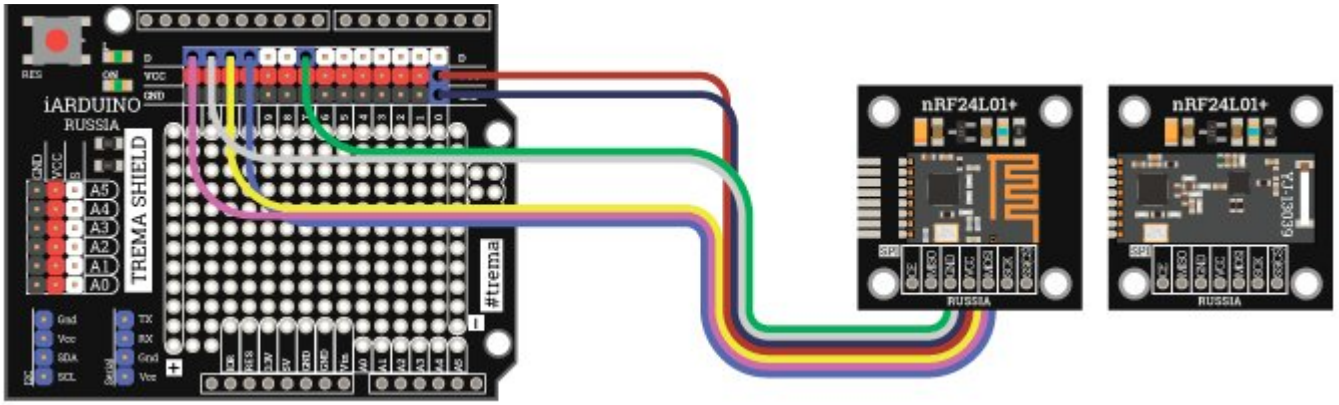


Используя провода «Папа - Мама», подключаем напрямую к контроллеру Arduino Mega:



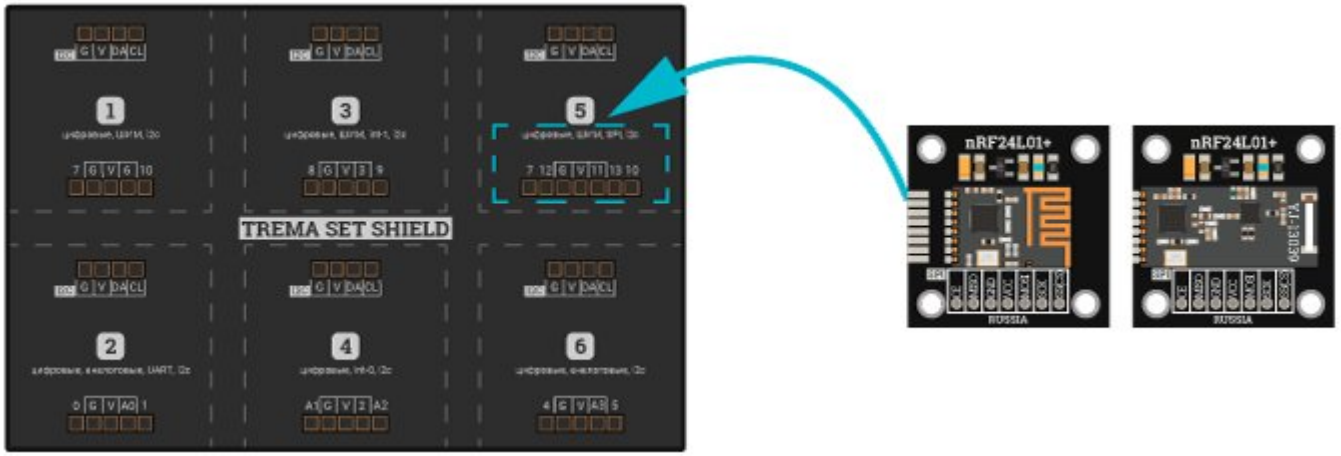
Способ - 2: Используя проводной шлейф и Shield

Используя провода «Папа - Мама», подключаем к Trema Shield, Trema-Power Shield, Motor Shield, Trema Shield NANO и тд.



Способ - 3: Используя Trema Set Shield

Модуль можно подключить к SPI входу Trema Set Shield который расположен в секции № 5.



Питание:

Входное напряжение питания 3,3 или 5 В постоянного тока, подаётся на выводы Vcc и GND модуля.

Подробнее о модуле:

Модуль общается с Arduino по шине SPI и позволяет не только передавать данные, но и программно задавать множество параметров

работы. Ему можно программно указать один из 126 каналов: от 0 (частота 2,400 ГГц) до 125 (частота 2,525 ГГц) на котором он будет работать. Так же модулю задаётся роль приёмника или передатчика. На одном канале можно создать сеть из 6 передатчиков и одного приёмника, при этом каждому передатчику присваивается уникальный номер (адрес трубы), а приемнику присваиваются адреса труб всех передатчиков, данные которых требуется принимать. Стоит отметить что на одном канале можно создать несколько сетей, в каждой из которых будут по одному приёмнику и до 6 передатчиков, главное что бы адреса труб передатчиков этих сетей не совпадали. Модулю можно задать уровень мощности передатчика и скорость передачи данных по радиоканалу.

Модуль может работать либо в режиме передатчика, либо в режиме приёмника, но передатчик способен запрашивать, а приёмник отправлять пакет подтверждения приёма данных. Происходит это по протоколу Enhanced Shockburst примерно так: передатчик отправляет данные приёмнику с запросом ответа, приёмник получает данные, проверяет их корректность (сверяет CRC), и если всё верно, то отвечает передатчику «Ок! я все получил, спасибо.». А если приёмник не ответил передатчику, то передатчик отправляет данные приёмнику повторно, пока не исчерпает заданное количество попыток отправки данных. При желании можно задать количество проверок, указать свою задержку между попытками отправки данных, отключить запрос подтверждения приема данных, отключить передачу CRC и его проверку, изменить длину CRC.

Модуль можно перевести в режим энергосбережения, при этом он сохранит возможность приёма и передачи данных.

Для работы с модулем предлагаем воспользоваться библиотекой RF24.

Примеры:

Рекомендуем начать работу с загрузки проверочного скетча, для определения правильности подключения модуля к Arduino.

Проверочный скетч:

```
#include <SPI.h>                                          // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                     // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                         // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                    // Создаём объект radio для работы с библиотекой RF24, указывая
                                                          //
void setup(){                                             //
    Serial.begin(9600);                                   // Инициируем передачу данных по шине UART в монитор последоват
```

```
    radio.begin();           Serial.print("Connected "    );   // Инициируем работу модуля nRF24L01+ и выводим текст «Connecte
    if(radio.isPVariant() ){ Serial.print("nRF24L01"     ); } // Если модуль поддерживается библиотекой RF24, то выводим текс
    else                   { Serial.print("unknown module"); } // Иначе, если модуль не поддерживается, то выводи текст «unkno
                             Serial.print("\r\n"          );   //
}                                                             //
                                                              //
void loop(){}                                                 //
```

Если после загрузки проверочного скетча Вы увидели, в окне монитора последовательного порта, надпись «Connected nRF24L01», значит Ваш модуль поддерживается библиотекой RF24 и можно приступать к проверке работы предложенных примеров...

Если Вы увидели надпись «Connected unknown module», проверьте подключение модуля к Arduino. В скетче указано что вывод «CE» (Chip Enable) модуля подключается к выводу 7 Arduino, а вывод SS (Slave Select) модуля подключается к выводу 10 Arduino. При необходимости измените выводы на другие. Если модуль подключён правильно, значит он собран на чипе отличном от nRF24L01.

Пример передачи данных:

```
#include <SPI.h>                                              // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                         // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                             // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                        // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                           // Объявляем массив для хранения и передачи данных (до 32 байт
                                                              //
void setup(){                                                 //
    radio.begin           ();                                 // Инициируем работу модуля nRF24L01+.
    radio.setChannel      (27);                               // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                       // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                      // Указываем уровень усиления передатчика (RF24_PA_MIN=-18dBm,
    radio.openWritingPipe (0xAABBCCDD11LL);                   // Открываем трубу с адресом 0xAABBCCDD11 для передачи данных (
}                                                             //
                                                              //
```

```
void loop(){                                                    //
    radio.write( &myData , sizeof(myData) );                    // Отправляем данные из массива myData указывая весь размер мас
    delay(50);                                                  // Устанавливаем задержку на 50 мс. В этом скетче нет смысла сл
}                                                               // Так же задержка нужна для того, что бы приёмник успел выполн
```

Скетч данного примера начинается с подключения файлов библиотек `SPI` , `RF24` и файла настроек `nRF24L01` . Далее создаётся объект `radio` с указанием выводов Arduino к которым подключены выводы модуля CE (Chip Enable) и SS (Slave Select). Можно указать любые выводы Arduino, но какие выводы Вы укажите, к тем выводам и следует подключать модуль. Далее в скетче объявляется массив `myData` из 5 элементов типа `int` , данные которого и будут передаваться. Библиотека RF24 позволяет передавать массивы любых типов, в т.ч. и строки, но за один раз можно передать не более 32 байт данных.

В коде `setup()` данного примера модулю задаются основные настройки: модуль работает в качестве передатчика (по умолчанию), на 27 канале, со скоростью 1 Мбит/сек (`RF24_1MBPS`), на максимальной мощности (`RF24_PA_MAX`), используя адрес трубы `0xAABBCCDD11` . На стороне приёмника нужно указать тот же номер канала, скорость передачи, мощность и адрес трубы.

В коде `loop()` осуществляется отправка данных функцией `write( данные , размер )` . В качестве данных для передачи указан адрес массива в памяти ОЗУ `&myData` , а в качестве размера передаваемых данных указан размер всего массива в байтах `sizeof(myData)` . Размер передаваемых данных указывается в байтах, а не в количестве элементов массива.

Если требуется отправить не весь массив `myData` а, например, его первые три элемента, то строку отправки данных можно было бы записать так `radio.write( &myData , 6 );` - отправить первые 3 элемента типа `int` (6 байт) массива `myData` .

Пример передачи данных с проверкой их доставки:

```
#include <SPI.h>                                                // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                           // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                               // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                          // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                             // Объявляем массив для хранения и передачи данных (до 32 байт
                                                                //
void setup(){                                                   //
```

```
    radio.begin           ();                                  // Инициируем работу модуля nRF24L01+.
    radio.setChannel      (27);                                // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                        // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                       // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.openWritingPipe (0xAABBCCDD11LL);                    // Открываем трубу с адресом 0xAABBCCDD11 для передачи данных (
}                                                              //
                                                               //
void loop(){                                                   //
    if( radio.write(&myData, sizeof(myData)) ){                // Если указанное количество байт массива myData было доставлен
    //  Данные передатчика были корректно приняты приёмником.  // Тут можно указать код который будет выполняться при получени
    }else{                                                     // Иначе (если данные не доставлены) ...
    //  Данные передатчика не приняты или дошли с ошибкой CRC. // Тут можно указать код который будет выполняться если приёмни
    }                                                          //
    delay(50);                                                 // Устанавливаем задержку на 50 мс. В этом скетче нет смысла сл
}                                                              // Так же задержка нужна для того, что бы приёмник успел выполн
```

Скетч данного примера отличается от предыдущего только кодом `loop()` где функция `write()` вызывается в условии оператора `if()` . Дело в том, что функция `write()` не только отправляет данные, но и возвращает `true` (если данные были доставлены) или `false` (если данные не доставлены). По умолчанию передача данных реализована так, что передатчик не только отправляет данные, но и запрашивает у приемника подтверждение их получения, а приёмник получив данные и проверив CRC, возвращает передатчику пакет подтверждения приема данных. Таким образом на стороне передатчика можно контролировать факт доставки данных приёмнику.

Пример получения данных от одного или нескольких передатчиков:

```
#include <SPI.h>                                               // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                          // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                              // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                         // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                            // Объявляем массив для приёма и хранения данных (до 32 байт вк
uint8_t  pipe;                                                 // Объявляем переменную в которую будет сохраняться номер трубы
                                                               //
```

```
void setup(){                                                    //
    radio.begin();                                               // Инициируем работу nRF24L01+.
    radio.setChannel      (27);                                  // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                          // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                         // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.openReadingPipe (1, 0xAABBCCDD11LL);                   // Открываем 1 трубу с адресом 1 передатчика 0xAABBCCDD11, для
    radio.openReadingPipe (2, 0xAABBCCDD22LL);                   // Открываем 2 трубу с адресом 2 передатчика 0xAABBCCDD22, для
    radio.openReadingPipe (3, 0xAABBCCDD33LL);                   // Открываем 3 трубу с адресом 3 передатчика 0xAABBCCDD33, для
    radio.openReadingPipe (4, 0xAABBCCDD96LL);                   // Открываем 4 трубу с адресом 4 передатчика 0xAABBCCDD96, для
    radio.openReadingPipe (5, 0xAABBCCDDFFLL);                   // Открываем 5 трубу с адресом 5 передатчика 0xAABBCCDDFF, для
    radio.startListening  ();                                    // Включаем приемник, начинаем прослушивать открытые трубы.
}                                                                //
                                                                 //
void loop(){                                                     //
    if(radio.available(&pipe)){                                  // Если в буфере имеются принятые данные, то получаем номер тру
        radio.read( &myData, sizeof(myData) );                   // Читаем данные из буфера в массив myData указывая сколько все
        if(pipe==1){ /* Данные пришли по 1 трубе */ ;}           // Если данные пришли от 1 передатчика (по 1 трубе), то можно в
        if(pipe==2){ /* Данные пришли по 2 трубе */ ;}           // Если данные пришли от 2 передатчика (по 2 трубе), то можно в
        if(pipe==3){ /* Данные пришли по 3 трубе */ ;}           // Если данные пришли от 3 передатчика (по 3 трубе), то можно в
        if(pipe==4){ /* Данные пришли по 4 трубе */ ;}           // Если данные пришли от 4 передатчика (по 4 трубе), то можно в
    }                                                            //
}                                                                //
```

Скетч приёмника начинается как и два предыдущих скетча передатчиков (подключаются библиотеки, создаётся объект, объявляется массив), но еще объявляется переменная `pipe` .

В коде `setup()` модулю задаются такие же настройки как и передатчику (27 канал, скорость 1 Мбит/сек, максимальная мощность передатчика). Тут стоит вспомнить про то, что приемник по умолчанию отправляет передатчику пакеты подтверждения приёма, а следовательно модулю выполняющему роль приёмника приходится указывать мощность передатчика (ведь если пакеты подтверждения от приёмника к передатчику будут отправляться с меньшей мощностью, то передатчик их может и не получить). Далее, в отличии от передатчика которому был задан один адрес трубы функцией `openWritingPipe()` , приёмнику можно задать до 6 труб функцией `openReadingPipe(`

`номер , адрес)` с номерами труб от 0 до 5 и адресами труб совпадающими с адресами труб передатчиков. Сколько труб Вы укажете, столько передатчиков будет слушать приёмник. И последняя функция которая вызвана в данном коде - `startListening()` включающая прослушивание труб, то есть переводящая модуль в режим работы приемника. Если далее вызвать функцию `stopListening()` - завершить прослушивание труб, то модуль перейдёт в режим работы передатчика.

В коде `loop()` осуществляется проверка получения данных функцией `available()`, которая возвращает `true` если в буфере есть принятые данные доступные для чтения. В качестве необязательного аргумента функции `available()` можно указать адрес переменной в которую будет помещён номер трубы по которой были приняты данные (в примере используется адрес переменной `&pipe`). Зная номер трубы мы знаем от какого передатчика пришли данные. Данные из буфера читаются функцией `read( адрес массива для данных , размер )`, в примере указан адрес ОЗУ массива `&myData`, а в качестве размера указывается размер массива `sizeof(myData)` в байтах, а не размер принимаемых данных.

Если приемник будет принимать данные только от одного передатчика, то объявлять переменную `pipe` в начале скетча не надо, функцию `openReadingPipe()` достаточно вызвать один раз (указав номер от 0 до 5 и адрес трубы передатчика), а функцию `available()` можно вызвать без параметра, так как в этом случае не требуется узнавать от какого передатчика приняты данные.

Пример ретрансляторов:

Ретранслятор это устройство с приемником и передатчиком, в котором все что принял приемник передаётся на передатчик. Чаще всего ретрансляторы применяют для увеличения дальности передачи данных. Если модуль рассчитан на дальность в 1 км, то расстояние между приёмником и передатчиком не должно превышать указанного расстояния. Но если поставить передатчик, через 1 км ретранслятор, а еще через 1 км приёмник, то расстояние между конечными передатчиком и приёмником будет 2 км.

```
#include <SPI.h>                                        // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                   // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                       // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                  // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                     // Объявляем массив для приёма, хранения и передачи данных (до
                                                        //
                                                        //
void setup(){                                           //
    radio.begin();                                      // Инициируем работу nRF24L01+
```

```
    radio.setChannel      (27);                                 // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                         // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                        // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.openReadingPipe (1, 0xFEDCBA9876LL);                  // Открываем 1 трубу с адресом 0xFEDCBA9876 для приема   данных
    radio.openWritingPipe (   0xAABBCCDD11LL);                  // Открываем   трубу с адресом 0xAABBCCDD11 для передачи данных
    radio.startListening  ();                                   // Включаем приемник, начинаем прослушивать открытые трубы.
}                                                               //
                                                                //
void loop(){                                                    //
    if( radio.available() ){                                    // Если в буфере имеются принятые данные, то ...
        radio.read           (&myData, sizeof(myData) );        // Читаем данные из буфера в массив myData указывая сколько все
        radio.stopListening  ();                                // Выключаем приемник, завершаем прослушивание открытых труб.
        radio.write          (&myData, sizeof(myData) );        // Отправляем данные из массива myData указывая сколько байт ма
        radio.startListening ();                                // Включаем приемник, начинаем прослушивать открытые трубы.
    }                                                           //
}                                                               //
```

В коде `setup()` данного примера модулю задаются основные настройки: модуль работает на 27 канале, со скоростью 1 Мбит/сек (`RF24_1MBPS`), на максимальной мощности (`RF24_PA_MAX`). Далее модулю присваиваются два адреса труб: `0xFEDCBA9876` для приёма данных и `0xAABBCCDD11` для передачи, после чего вызывается функция `startListening()` включается режим прослушивания труб (режим приёма данных).

Алгоритм кода `loop()` заключается в том, что как только модуль получает данные `if(radio.available()){...}` , они читаются функцией `read()` в массив `myData` , далее модуль завершает прослушивание труб `stopListening()` , отправляет все полученные данные функцией `write()` и вновь начинает прослушивание труб `startListening()` .

Таким образом устройство с данным скетчем является приёмопередатчиком и может стать посредником между передатчиком, отправляющим данные по трубе с адресом `0xFEDCBA9876` и приёмником прослушивающим трубу с адресом `0xAABBCCDD11` , при этом все устройства работают на одной частоте (27 канал), с одной и той же скоростью (1 Мбит/сек) и мощностью.

Скетч представленный ниже является еще одним ретранслятором, но в отличии от предыдущего он использует один адрес трубы, но

принимает данные на одном канале, а передаёт на другом:

```
#include <SPI.h>                                        // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                   // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                       // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                  // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                     // Объявляем массив для приёма, хранения и передачи данных (до
                                                        //
void setup(){                                           //
    radio.begin();                                      // Инициируем работу nRF24L01+.
    radio.setChannel      (27);                         // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                 // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.openReadingPipe (0, 0xAABBCCDD11LL);          // Открываем 0 трубу с адресом 0xAABBCCDD11, для приема и перед
    radio.startListening  ();                           // Включаем приемник, начинаем прослушивать открытые трубы.
}                                                       //
                                                        //
void loop(){                                            //
    if(radio.available()){                              // Если в буфере имеются принятые данные, то ...
        radio.read            ( &myData, sizeof(myData) );  // Читаем данные из буфера в массив myData указывая сколько все
        radio.stopListening   ();                       // Выключаем приёмник, завершаем прослушивание открытых труб.
        radio.setChannel      (29);                     // Указываем канал передачи данных (от 0 до 125), 29 - значит г
        radio.write           ( &myData, sizeof(myData) );  // Отправляем данные из массива myData указывая весь размер мас
        radio.setChannel      (27);                     // Указываем канал передачи данных (от 0 до 125), 27 - значит г
        radio.startListening  ();                       // Включаем приемник, начинаем прослушивать открытые трубы.
    }                                                   //
}                                                       //
```

В коде `setup()` данного примера, функцией `openReadingPipe()`, модулю была указана труба для приёма данных с номером `0` и адресом `0xAABBCCDD11`. Адрес трубы с номером `0` (в отличии от труб с номерами 1-5) применяется как для приёма, так и для передачи данных, по этому функция `openWritingPipe()` в данном коде не участвует.

Алгоритм кода `loop()` аналогичен предыдущему скетчу, за исключением того, что для передачи данных модуль использует 29 канал, а для приёма 27. Не рекомендуется использовать соседние каналы (27 и 28, или 28 и 29, и т.д.) для работы с одинаковыми адресами труб.

При желании можно совместить алгоритм работы первого и второго варианта ретранслятора так, что бы он работал не только на разных каналах, но и с разными адресами труб.

При работе ретранслятора важно учитывать тот факт, что передатчик должен выдерживать паузы между передачей данных ретранслятору, достаточные для их получения, обработки и дальнейшей передачи ретранслятором.

Двунаправленная отправка данных:

Модуль может работать либо в режиме передатчика, либо в режиме приёмника, но передатчик способен запрашивать, а приёмник отправлять пакет подтверждения приёма данных. Происходит это по протоколу Enhanced Shockburst примерно так: передатчик отправляет данные приёмнику с запросом ответа, приёмник получает данные, проверяет их корректность (сверяет CRC), и если всё верно, то отвечает передатчику «Ок! я все получил, спасибо.». А если приёмник не ответил передатчику, то передатчик отправляет данные приёмнику повторно, пока не исчерпает заданное количество попыток отправки данных.

Так вот в ответе приёмника могут присутствовать и те данные которые Вы хотите отправить передатчику. Правда эти данные уже не будут проходить проверку, и они должны быть сформированы на стороне приёмника до того как он получит данные от передатчика. Но такой тип связи позволит приёмнику возвращать, а передатчику принимать, данные без изменения режимов радиопередач на обоих устройствах, приёмник остаётся приёмником, а передатчик передатчиком.

Скетч приёмника:

```
#include <SPI.h>                                    // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                               // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                   // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                              // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                 // Объявляем массив для приёма и хранения данных (до 32 байт вк
int      ackData[5];                                // Объявляем массив для передачи данных в пакете подтверждения
                                                    //
```

```
void setup(){                                                    //
    radio.begin();                                               // Инициируем работу nRF24L01+
    radio.setChannel      (27);                                  // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                          // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                         // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.enableAckPayload();                                    // Указываем что в пакетах подтверждения приёма есть блок с пол
//  radio.enableDynamicPayloads();                               // Разрешить динамически изменяемый размер блока данных на всех
    radio.openReadingPipe (1, 0xAABBCCDD11LL);                   // Открываем 1 трубу с адресом 0xAABBCCDD11, для приема данных.
    radio.startListening  ();                                    // Включаем приемник, начинаем прослушивать открытые трубы.
    radio.writeAckPayload (1, &ackData, sizeof(ackData) );       // Помещаем данные всего массива ackData в буфер FIFO. Как толь
}                                                                // В модуле имеется 3 буфера FIFO, значит в них одновременно мо
                                                                 // После отправки данных из буфера FIFO к передатчику, соответс
void loop(){                                                     //
    if(radio.available()){                                       // Если в буфере приёма имеются принятые данные от передатчика,
        radio.read            (   &myData,  sizeof(myData)  );   // Читаем данные из буфера приёма в массив myData указывая скол
        radio.writeAckPayload (1, &ackData, sizeof(ackData) );   // Помещаем данные всего массива ackData в буфер FIFO для их от
    }                                                            // Если все 3 буфера FIFO уже заполнены, то функция writeAckPay
}                                                                // Так как в данном скетче данные в буфер помещаются только пос
```

Обратите внимание на то, что ответ приёмника `ackData` сначала помещается в буфер FIFO функцией `writeAckPayload()`, а отправляется он аппаратно, при получении данных от передатчика, но ещё до того как функция `available()` вернёт `true`.

Скетч передатчика:

```
#include <SPI.h>                                                 // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                            // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                                // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                           // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                              // Объявляем массив для хранения и передачи данных.
int      ackData[5];                                             // Объявляем массив для получения данных из пакета подтвержден
                                                                 //
```

```
void setup(){                                                    //
    radio.begin           ();                                    // Инициируем работу модуля nRF24L01+.
    radio.setChannel      (27);                                  // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                          // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                         // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.enableAckPayload();                                    // Указываем что в пакетах подтверждения приёма есть блок с пол
//  radio.enableDynamicPayloads();                               // Разрешить динамически изменяемый размер блока данных на всех
    radio.openWritingPipe (0xAABBCCDD11LL);                      // Открываем трубу с адресом 0xAABBCCDD11 для передачи данных (
}                                                                //
                                                                 //
void loop(){                                                     //
    radio.write(&myData, sizeof(myData));                        // Отправляем данные из массива myData указывая сколько байт ма
    if( radio.isAckPayloadAvailable() ){                         // Если в буфере имеются принятые данные из пакета подтвержден
        radio.read(&ackData, sizeof(ackData));                   // Читаем данные из буфера в массив ackData указывая сколько во
    }                                                            //
    delay(50);                                                   // Устанавливаем задержку на 50 мс. В этом скетче нет смысла сл
}                                                                //
```

В этих скетчах одному модулю назначена роль приемника, а другому передатчика, но не смотря на это связь организована двухсторонняя, без изменения ролей модулей в радиопередаче.

Стоит отметить несколько особенностей данной реализации связи:

- На стороне передатчика, вместо функции `isAckPayloadAvailable()` можно использовать функцию `available()`, она будет работать точно так же.
- Блоки данных в пакетах подтверждения приёма имеют динамически изменяемый размер, он используется по умолчанию для отправки ответов по трубам 0 и 1, так что если данные приемника нужно отправить передатчику по трубам 2-5 (где по умолчанию используются статичный размер ответных данных), то необходимо раскомментировать функцию `enableDynamicPayloads()` в обоих скетчах.
- Если на стороне передатчика не читать данные из буферов, то буферы заполнятся и перестанут принимать пакеты подтверждения приёма, тогда функция `write()` будет всегда возвращать `false`, хотя данные будут и отправляться, и приниматься приемниками.
- Если на стороне приемника формировать ответы для нескольких передатчиков (а каждый из них должен использовать свою трубу), то

нужно учитывать тот факт что у модуля для этих целей имеется всего 3 буфера, которые могут хранить до трех ответов для труб с разными или одинаковыми номерами. Если буферы заполнены, то функция `writeAckPayload()` будет проигнорирована. Например, если в буферах уже есть ответы для труб с номерами 1,2 и 3, новые ответы записать не получится, а пришли данные от передатчика по трубе с номером 0, то он естественно не получит ответ.

Передача данных нескольким приёмникам:

В некоторых случаях может возникнуть необходимость отправки данных от одного передатчика к нескольким приёмникам, это можно сделать, но каждый приёмник будет отправлять передатчику пакеты подтверждения приёма. Эти пакеты будут отправляться почти одновременно и как следствие неизбежно «перемешаются / столкнутся».

Представленный ниже скетч передатчика отправляет данные приёмникам не запрашивая у них пакеты подтверждения приёма, так как функция `write()` вызывается с установленным флагом групповой передачи данных (третий параметр функции). Стоит отметить что и в коде `setup()` было обращение к функции `enableDynamicAck()`, без которой функция `write()` будет игнорировать флаг групповой передачи данных.

```
#include <SPI.h>                                    // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                               // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                   // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                              // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                 // Объявляем массив для хранения и передачи данных.
                                                    //
void setup(){                                       //
    radio.begin           ();                       // Инициируем работу модуля nRF24L01+.
    radio.setChannel      (27);                     // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);             // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);            // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.enableDynamicAck();                       // Разрешаем выборочно отключать запросы подтверждения приема д
    radio.openWritingPipe (0xAABBCCDD11LL);         // Открываем трубу с адресом 0xAABBCCDD11 для передачи данных (
}                                                   //
                                                    //
void loop(){                                        //
```

```
    radio.write(&myData, sizeof(myData), true);              // Отправляем данные из массива myData указывая сколько байт ма
    delay(50);                                               // Устанавливаем задержку на 50 мс. В этом скетче нет смысла сл
}                                                            // Так же задержка нужна для того, что бы приёмники успели выпо
```

Такой передатчик способен отправить данные сразу нескольким приёмникам на одном канале и по одной трубе. Корректность доставки данных по прежнему будет проверяться на стороне приемников путём сравнения CRC, но сам факт доставки данных проверить не получится, так как функция `write()` на стороне передатчика будет постоянно возвращать `true`, вне зависимости от того доставлены данные или нет.

Запретить отправку пакетов подтверждения приёма можно и на стороне приёмников, вызвав у них функцию `setAutoAck( false )` или `setAutoAck( номер трубы, false )`. Но в таком случае и на стороне передатчика нужно вызвать функцию `setAutoAck( false )` иначе приёмник не будет понимать что ему прислал передатчик.

Быстрая передача данных:

Функция `write()` в скетче передатчика по умолчанию ждёт пока приёмник не подтвердит получение данных (пока приёмник не получит данные и не отправит пакет подтверждения приема) или пока не будут исчерпаны все попытки доставки данных, после чего функция возвращает флаг доставки данных `true` или `false`. Время ожидания в стандартной конфигурации может достигать 60 - 70 миллисекунд.

В представленном ниже скетче функция `write()` заменена на `writeFast()` которая принимает те же параметры что и `write()`, но новая функция возвращает не флаг доставки данных приёмнику, а флаг записи данных в буфер FIFO. Значит в большинстве случаев функция вернёт `true` даже до того как приёмник получит данные. Если же все три буфера FIFO заполнены, то функция `writeFast()` ждёт пока один из них не освободится или пока не истечёт время таймаута но и это ожидание на порядок меньше чем у функции `write()`.

```
#include <SPI.h>                                             // Подключаем библиотеку для работы с шиной SPI.
#include <nRF24L01.h>                                        // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>                                            // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);                                       // Создаём объект radio для работы с библиотекой RF24, указывая
int      myData[5];                                          // Объявляем массив для хранения и передачи данных.
                                                             //
```

```
void setup(){                                                    //
    radio.begin           ();                                    // Инициируем работу модуля nRF24L01+.
    radio.setChannel      (27);                                  // Указываем канал передачи данных (от 0 до 125), 27 - значит г
    radio.setDataRate     (RF24_1MBPS);                          // Указываем скорость передачи данных (RF24_250KBPS, RF24_1MBPS
    radio.setPALevel      (RF24_PA_MAX);                         // Указываем мощность передатчика (RF24_PA_MIN=-18dBm, RF24_PA_
    radio.enableDynamicAck();                                    // Разрешаем выборочно отключать запросы подтверждения приема д
    radio.openWritingPipe (0xAABBCCDD11LL);                      // Открываем трубу с адресом 0xAABBCCDD11 для передачи данных (
}                                                                //
                                                                 //
void loop(){                                                     //
    radio.writeFast(&myData, sizeof(myData));                    // Отправляем данные из массива myData указывая сколько байт ма
}                                                                //
```

К недостаткам данного скетча можно отнести то, что на передающей стороне нельзя определить факт доставки данных приёмнику.

Список функций библиотеки:

- RF24 **объект(** вывод CE , вывод SS **);** // создание объекта с указанием выводов Arduino.
- **begin();** // Инициализация модуля.
- **startListening();** // Начать прослушивание труб, открытых для приёма данных.
- **stopListening();** // Прекратить прослушивание труб и переключиться в режим передатчика.
- **available(** [№ трубы] **);** // Проверить наличие принятых данных доступных для чтения.
- **isAckPayloadAvailable();** // Проверить передатчиком наличие данных в ответе приёмника.
- **read(** адрес для чтения данных , размер **);** // Прочитать принятые данные.
- **write(** данные , размер [, флаг групповой передачи] **);** // Отправить данные по радиоканалу.
- **writeAckPayload(** № трубы , данные , размер **);** // Подготовить данные для ответа передатчику.
- **openWritingPipe(** адрес трубы **);** // Открыть трубу для передачи данных.
- **openReadingPipe(** № трубы , адрес трубы **);** // Открыть трубу для приёма данных.
- **closeReadingPipe(** № трубы **);** // Закрыть трубу открытую ранее для приёма данных.

- **setChannel(** № канала **);** // Установить радиочастотный канал связи.
- **getChannel();** // Получить текущий радиочастотный канал связи.
- **setDataRate(** скорость **);** // Установить скорость передачи данных по радиоканалу.
- **getDataRate();** // Получить текущую скорость передачи данных по радиоканалу.
- **setPALevel(** уровень **);** // Установить уровень усиления мощности передатчика.
- **getPALevel();** // Получить текущий уровень усиления мощности передатчика.
- **setCRCLength(** размер **);** // Установить размер CRC (циклически избыточный код).
- **getCRCLength();** // Получить текущий размер CRC (циклически избыточный код).
- **disableCRC();** // Отключить передачу CRC передатчиком и проверку данных приёмником.
- **setPayloadSize(** размер **);** // Установить статичный размер блока данных в байтах.
- **getPayloadSize();** // Получить текущий статичный размер блока данных в байтах.
- **getDynamicPayloadSize();** // Получить размер блока данных в последнем принятом пакете.
- **enableDynamicPayloads();** // Разрешить изменяемый размер блока данных для всех труб.
- **enableDynamicAck();** // Разрешить отказываться от запроса пакетов подтверждения приёма.
- **enableAckPayload();** // Разрешить размещать данные в пакеты подтверждения приёма.
- **setAutoAck(** [№ трубы ,] флаг **);** // Вкл/выкл пакеты подтверждения приёма данных.
- **setAddressWidth(** размер **);** // Указать какой длины использовать адреса труб в байтах.
- **setRetries(** время , количество **);** // Кол-о попыток отправки данных и задержка между ними.
- **powerDown();** // Перейти в режим пониженного энергопотребления.
- **powerUp();** // Выйти из режима пониженного энергопотребления.
- **isPVariant();** // Проверить аппаратную совместимость модуля с функциями nRF24L01.

- **writeFast(** данные , размер [, флаг] **);** // Быстро отправить данные по радиоканалу.
- **writeBlocking(** данные , размер , время **);** // Быстро отправить данные с указанием таймаута.
- **startFastWrite(** данные , размер , флаг , флаг **);** // Начать быструю отправку данных.
- **startWrite(** данные , размер , флаг групповой передачи **);** // Начать отправку данных.

- **txStandBy(** [время ожидания] **);** // Подождать пока передаются данные и вернуть результат.
- **rxFifoFull();** // Проверить не заполнены ли все три буфера FIFO.

- **flush_tx();** // Очистить буферы от данных для передачи, вернуть значение регистра состояния.
- **reUseTX();** // Повторная отправка данных из буфера FIFO, если они там есть.
- **testCarrier();** // Проверка наличия несущей частоты на выбранном канале (частоте).
- **testRPD();** // Проверка наличия любого сигнала выше -64 дБм на выбранном канале (частоте).
- **isValid();** // Проверить используется ли модуль или выполняется отладка кода.

Описание функций библиотеки:

Перед просмотром описаний функций библиотеки RF24 предлагаем Вам ознакомиться со следующими понятиями:

- **Канал** - номер от 0 до 125 определяющий частоту на которой работает модуль. Каждый канал имеет шаг в 1 МГц, а каналу 0 соответствует частота 2,4 ГГц = 2400 МГц, следовательно, каналу 1 соответствует частота 2401 МГц, каналу 2 - частота 2402 МГц и т.д. до канала 125 с частотой 2525 МГц.
- **Труба** - абстрактное понятие которое можно представить как невидимая труба в которую с одной стороны отправляет данные передатчик, а с другой стороны принимает данные приёмник, при этом они никому не мешают, им никто не мешает и их «не могут» подслушать.
- **Адрес трубы** - уникальный адрес позволяющий приёмнику понять, что данные назначаются именно ему, а передатчику отправлять данные для конкретного приемника. Общающимся передатчику и приемнику задаётся один и тот же адрес трубы. Адрес по умолчанию состоит из 5 байт и может быть представлен числом типа uint64_t или массивом из 5 однобайтных элементов.
- **Сеть** - система обеспечивающая обмен данными между несколькими радиомодулями. Передатчик может одновременно вещать только по одной трубе, а приёмник может одновременно прослушивать до 6 труб, по этому на одном канале можно создать сеть из одного приёмника и до 6 передатчиков. Правда никто не запрещает создать на одном канале более одной сети, главное что бы не пересекались адреса их труб. Так же нужно учитывать что чем больше передатчиков находятся на одном канале и чем чаще они передают данные, тем выше вероятность «столкновения» пакетов и как следствие недоставление их адресату. На одном канале, используя одну трубу, можно создать сеть состоящую из одного передатчика и неограниченного количества приёмников. И туту тоже никто не запрещает создать на одном канале несколько таких сетей. Но в таких сетях передатчик передаёт данные сразу всем приёмникам своего канала, а подтверждение доставки данных не проверяется.
- **Номер трубы** - Так как приёмник может прослушивать до 6 труб одновременно, то у приёмника каждая труба имеет не только адрес, но и номер от 0 до 5.
- **Пакет** - определённым образом оформленный блок данных, передаваемый по сети. Данные между передатчиком и приемником передаются пакетами. В пакете помимо данных пользователя имеются и иные блоки (стартовые биты, адрес трубы, биты управления, CRC

и т.д.).

- **Данные** - до 32 байт данных пользователя передаются пакетом, где блок данных пользователя имеет статичный (по умолчанию) или динамичный размер. Если блок данных пользователя в пакете имеет динамичный размер, то чем больше байтов Вы отправляете, тем длиннее отправляемый передатчиком пакет. Если блок данных пользователя в пакете имеет статичный размер (по умолчанию), то пакет отправляемый передатчиком имеет один и тот же размер, вне зависимости от количества отправляемых Вами байт данных.
- **Ответ** - пакет подтверждения приёма данных, отправляется от приёмника к передатчику. Передатчик способен запрашивать, а приёмник отправлять пакет подтверждения приёма данных. Происходит это по протоколу Enhanced Shockburst примерно так: передатчик отправляет данные приёмнику с запросом ответа, приёмник получает данные, проверяет их корректность (сверяет CRC), и если всё верно, то отвечает передатчику «Ок! я все получил, спасибо». А если приёмник не ответил передатчику, то передатчик отправляет данные приёмнику повторно, пока не исчерпает заданное количество попыток отправки данных. Но приёмник может не просто ответить передатчику, а вложить в ответ и свои данные которые примет передатчик. Таким образом можно организовать двухстороннюю связь. Блок данных в пакете подтверждения имеет динамический размер.
- **Назначение адресов туб** - имеет ограничения связанные с номерами труб приёмника:
  - Труба с номером 0 используется и для чтения, и для записи. Если приёмнику указать адрес 0 трубы после чего задать роль передатчика и отправить данные, то эти данные будут отправлены по трубе с адресом который был задан приемнику как труба с номером 0. Даже если до изначально был назначен другой адрес трубы для передачи данных.
  - Адрес трубы с номером 0 может полностью отличаться от адресов труб с номерами 1-5.
  - Если адреса трубам назначаются как числа типа uint64_t, то адреса труб с номерами 2-5 должны отличаться от адреса трубы с номером 1 только последним (младшим) байтом числа.
  - Если адреса трубам назначаются как массивы, то адреса труб с номерами 2-5 должны отличаться от адреса трубы с номером 1 только первым элементом массива.
  - Трубы с номерами 0 и 1 хранят полный 5 байтовый (по умолчанию) адрес, а трубы 2-5 технически хранят только 1 байт, заимствуя 4 дополнительных байта из адреса 1 трубы, не смотря на то, что вы задаёте им полный 5 байтовый адрес. По этому нельзя открывать трубы с номерами 2-5 если не открыта труба с номером 1.
  - Размер адреса труб можно уменьшить до 4 или 3 байт, как для приёмника, так и для передатчика.

Подключение библиотеки:

```
#include <SPI.h>          // Подключаем библиотеку для работы с шиной SPI.
```

```
#include <nRF24L01.h>    // Подключаем файл настроек из библиотеки RF24.
#include <RF24.h>        // Подключаем библиотеку для работы с nRF24L01+.
RF24     radio(7, 10);   // Создаём объект radio, указывая номера выводов Arduino к которым подключены выводы модуля (CE, SS).
```

Если в качестве одного из параметров указать не номер вывода, а число 255 (0xFF) то библиотека будет работать в режиме отладки кода, а функция isValid() будет возвращать false.

Функция begin();

- Назначение: Инициализация работы модуля.
- Синтаксис: begin();
- Параметры: Нет.
- Возвращаемое значение: bool - результат инициализации (true / false).
- Примечание:
  - Вызывается однократно в коде setup().
  - Функцию необходимо вызвать до обращения к остальным функциям библиотеки.
  - После вызова данной функции модуль будет работать в роли передатчика, для перевода модуля в режим приёмника обратитесь к функции startListening().
- Пример:

```
setup(){           //
    radio.begin(); // Инициализация работы модуля.
    ...            // Обращение к иным функциям библиотеки.
}                  //
```

Функция startListening();

- Назначение: Начать прослушивание труб, открытых для приёма данных.
- Синтаксис: startListening();
- Параметры: Нет.

- Возвращаемое значение: Нет.
- Примечание:
  - Перед началом прослушивания труб нужно указать модулю какие именно трубы прослушивать, для этого обратитесь к функции openReadingPipe().
  - Не вызывайте функцию write() в режиме прослушивания труб.
  - Модуль находясь в режиме прослушивания труб является приёмником.
  - Для выхода из режима прослушивания труб вызовите функцию stopListening(), модуль перейдёт в режим передатчика.
  - Функцией available() можно проверить наличие принятых данных доступных для чтения.
  - Функцией read() можно прочитать принятые данные.
- Пример:

```
radio.openReadingPipe( 1, 0xAABBCCDDE0LL ); // Задаём номер и адрес трубы для прослушивания.
radio.openReadingPipe( 2, 0xAABBCCDDE1LL ); // Задаём номер и адрес трубы для прослушивания.
radio.startListening();                     // Указываем модулю начать прослушивание заданных труб (начать работать в режиме г
```

Функция stopListening();

- Назначение: Прекратить прослушивание труб и переключиться в режим передатчика.
- Синтаксис: stopListening();
- Параметры: Нет
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме приёмника.
  - После вызова данной функции модуль начнёт работать в режиме передатчика, в котором можно вызывать функцию write() для отправки данных.
  - Роль модуля (приёмник/передатчик) можно неоднократно менять обращаясь к функциям startListening() / stopListening() в процессе работы модуля.
- Пример:

```
radio.stopListening();           // Указываем модулю прекратить прослушивание труб и переключиться в режим передатчика.
radio.write(&data,sizeof(data)); // Отправляем данные data указав их размер в байтах.
```

Функция available();

- Назначение: Проверить наличие принятых данных доступных для чтения.
- Синтаксис: available([№ ТРУБЫ]);
- Параметры:
  - № ТРУБЫ - адрес переменной типа uint8_t в которую требуется поместить номер трубы по которой были приняты данные.
- Возвращаемое значение: bool - флаг наличия принятых данных (true / false).
- Примечание:
  - Функция используется модулем в режиме приёмника, но может быть использована и в режиме передатчика.
  - Единственный параметр данной функции является необязательным.
  - Указывать параметр целесообразно только если приёмник получает данные по нескольким трубам (от нескольких передатчиков), тогда по номеру трубы можно определить от какого именно передатчика получены данные.
  - Для того что бы в качестве параметра функции указать адрес переменной, а не её значение, перед именем переменной указывается символ «&».
  - Если функция вернула true - есть принятые данные доступные для чтения, то их можно прочитать функцией read().
  - Размер полученных данных можно узнать функцией getDynamicPayloadSize().
- Пример: (получение данных без проверки номера трубы по которой они пришли)

```
if( radio.available() ){           // Проверяем наличие принятых данных без получения номера трубы по которой они пришли.
    radio.read(&data,sizeof(data)); // Читаем принятые данные в массив data указав размер этого массива в байтах.
}                                  //
```

- Пример: (получение данных и получение номера трубы по которой пришли данные)

```
uint8_t pipe;                      // Объявляем переменную для хранения номера трубы по которой приняты данные.
if( radio.available( &pipe ) ){    // Проверяем наличие принятых данных и получаем номер трубы по которой они пришли в переме
    radio.read(&data,sizeof(data)); // Читаем принятые данные в массив data указав размер этого массива в байтах.
```

```
    if( pipe==0 ){...}else          // Выполняем действия с полученными данными, зная что они пришли по трубе с номером 0.
    if( pipe==1 ){...}else          // Выполняем действия с полученными данными, зная что они пришли по трубе с номером 1.
    if( pipe==2 ){...}else ...      // Выполняем действия с полученными данными, зная что они пришли по трубе с номером 2.
}                                   //
```

Функция isAckPayloadAvailable();

- Назначение: Проверить передатчиком наличие данных в ответе приёмника.
- Синтаксис: isAckPayloadAvailable();
- Параметры:
  - ПАРАМЕТР - назначение редактируется.
- Возвращаемое значение: bool - флаг наличия принятых данных от приёмника (true / false).
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - По умолчанию, передатчик запрашивает у приёмника подтверждение получения данных. Приёмник получив данные от передатчика отправляет в ответ пакет подтверждения приёма, сигнализируя передатчику о том что его данные получены. Если на стороне приёмника вызвать функцию writeAckPayload(ТРУБА,ДАННЫЕ,РАЗМЕР), то приёмник не просто ответит передатчику, а вложит в ответ свои данные которые примет передатчик.
  - Функция isAckPayloadAvailable() позволяет на стороне передатчика проверить не получил ли он данные от приёмника, а функция read() позволяет прочитать принятые данные.
  - Вместо данной функции на стороне передатчика можно использовать функцию available() без параметра, она так же вернёт флаг наличия принятых данных доступных для чтения.
  - В разделе «Примеры» данной статьи есть пример «Двунаправленная отправка данных:» с использованием функции isAckPayloadAvailable().
- Пример:

```
if( radio.isAckPayloadAvailable() ){ // Проверяем наличие принятых данных в ответе приёмника.
    radio.read(&data,sizeof(data));  // Читаем принятые данные в массив data указав размер этого массива в байтах.
}                                    //
```

Функция read();

- Назначение: Прочитать принятые данные.
- Синтаксис: read(АДРЕС ДЛЯ ЧТЕНИЯ ДАННЫХ , РАЗМЕР);
- Параметры:
  - АДРЕС ДЛЯ ЧТЕНИЯ ДАННЫХ - адрес массива, строки или переменной в которую требуется поместить принятые данные.
  - РАЗМЕР - количество байт занимаемое массивом, строкой или переменной в которую требуется поместить принятые данные.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме приёмника.
  - Перед вызовом данной функции на стороне приёмника, проверьте наличие принятых данных обратившись к функции available().
  - Перед вызовом данной функции на стороне передатчика, проверьте наличие принятых данных из ответа приёмника обратившись к функции isAckPayloadAvailable().
  - Для того что бы в качестве параметра функции указать адрес переменной, а не её значение, перед именем переменной указывается символ «&».
  - Если указанный размер меньше размера принятых данных, то данные будут урезаны до указанного размера.
  - Если указанный размер больше размера принятых данных, то оставшиеся байты массива, строки или переменной останутся без изменений.
  - Максимальный размер принятых данных не может превышать 32 байта.
  - Реальный размер принятых данных можно узнать функцией getDynamicPayloadSize().
- Пример:

```
тип data[количество];               // Объявляем массив для получения данных. Можно указать любой тип и количество элементов м
if( radio.available() ){            // Проверяем наличие принятых данных без получения номера трубы по которой они пришли.
    radio.read(&data,sizeof(data)); // Читаем принятые данные в массив data указав размер этого массива в байтах.
}                                   //
```

Функция write();

- Назначение: Отправить данные по радиоканалу.
- Синтаксис: write(ДАННЫЕ , РАЗМЕР [, ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ]);
- Параметры:
  - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить.
  - РАЗМЕР - отправляемых данных в байтах.
  - ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ - установите в true если требуется отправить данные нескольким приёмникам.
- Возвращаемое значение: bool - результат доставки данных приёмнику (true / false).
- Примечание:
  - Функция используется модулем в режиме передатчика, обязательными являются только два первых параметра функции.
  - Не отправляйте данные если не было ни одного обращения к функции openWritingPipe() указывающей модулю адрес трубы для отправки данных.
  - Если была вызвана функция startListening() - модуль находится в роли приёмника, то перед отправкой данных нужно вызвать функцию stopListening() задав модулю роль передатчика.
  - Функция ждёт пока данные не будут доставлены приёмнику, или пока не будут исчерпаны все попытки доставки данных, что может занять до 60-70 миллисекунд.
  - Максимальное количество попыток доставки данных можно указать функцией setRetries().
  - Максимальный размер передаваемых данных не может превышать 32 байта. Этот размер можно уменьшить функцией setPayloadSize() или узнать функцией getPayloadSize().
  - При попытке передать массив или строку превышающую максимальный размер, будут переданы данные урезанные до установленного максимального размера.
  - Если вызвать функцию с третьим параметром установленным в true, то передатчик отправит данные без запроса ответа от приёмника. Таким образом можно отправить данные сразу нескольким приёмникам по одной трубе, при этом ни один из приёмников не ответит передатчику, но и передатчик не сможет узнать доставлены ли данные приёмнику, а функция всегда будет возвращать true, как будто данные доставлены.
  - Отключить запросы подтверждения приёма можно только если в скетче уже было обращение к функции enableDynamicAck() разрешающей отказываться от запроса пакетов подтверждения приёма.
  - Вместо функции write() можно воспользоваться функциями: writeFast(), writeBlocking(), startWrite(), startFastWrite().
  - В разделе «Примеры» данной статьи есть пример «Передача данных нескольким приёмникам» с использованием функции write()

которая вызывается с третьим параметром установленным в true.

- Пример:

```
int data[16] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5}; // Определяем массив для передачи данных.
radio.write( &data , sizeof(data) );               // Отправляем данные из массива data указывая весь размер массива в байтах.
```

Функция writeAckPayload();

- Назначение: Подготовить данные для ответа передатчику.
- Синтаксис: writeAckPayload(№ ТРУБЫ , ДАННЫЕ , РАЗМЕР);
- Параметры:
  - № ТРУБЫ - передатчика которому требуется ответить данными.
  - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить вместе с ответом передатчику.
  - РАЗМЕР - отправляемых данных в байтах.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме приёмника.
  - Функция не будет работать если не было ни одного обращения к функции enableAckPayload() которая разрешает размещать данные в ответы приёмника (пакеты подтверждения получения данных).
  - Функция помещает данные в один из трёх буферов FIFO модуля. Если приёмник получает по трубе с указанным номером, любые данные от передатчика с запросом подтверждения приёма, то на аппаратном уровне в ответ передатчику добавляются и данные из буфера FIFO, при этом сам буфер очищается.
  - Так как в модуле доступно 3 буфера FIFO то в них одновременно можно загрузить до трёх разных или одинаковых данных, для одной или разных труб, которые будут отправлены вместе с подтверждением приёма данных по этим трубам.
  - Если при обращении к функции writeAckPayload() все три буфера FIFO уже имеют данные, то новые данные не будут записаны в буферы FIFO.
  - Факт заполненности всех трёх буферов FIFO можно проверить функцией rxFifoFull() и при необходимости очистить все буферы функцией flush_tx().
  - Максимальный размер данных принимаемых функцией writeAckPayload() не может превышать 32 байта (размер одного буфера FIFO).

Если указать размер более 32 байт, то данные будут урезаны до 32 байт.
  - Для передачи данных в пакетах подтверждения приёма по трубам с номерами 2-5 необходимо предварительно вызвать функцию enableDynamicPayloads() которая разрешает динамически изменяемый размер блока данных на всех трубах.
  - В разделе «Примеры» данной статьи есть пример «Двунаправленная отправка данных:» с использованием функции writeAckPayload().
- Пример:

```
int dataAck[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};     // Определяем массив с данными для ответа передатчику.
radio.writeAckPayload( 1, &dataAck , sizeof(dataAck) );  // Отправляем данные для ответа в буфер FIFO приёмника. Передатчик 1
dataAck[2]=1;                                            // Меняем значение второго элемента массива dataAck.
radio.writeAckPayload( 1, &dataAck , sizeof(dataAck) );  // Отправляем данные для ответа в буфер FIFO приёмника. Передатчик 1
dataAck[2]=2;                                            // Меняем значение второго элемента массива dataAck.
radio.writeAckPayload( 2, &dataAck , sizeof(dataAck) );  // Отправляем данные для ответа в буфер FIFO приёмника. Передатчик 2
                                                         // Дальнейшее обращение к функции writeAckPayload() не имеет смысла г
```

Функция openWritingPipe();

- Назначение: Открыть трубу для передачи данных.
- Синтаксис: openWritingPipe(АДРЕС ТРУБЫ);
- Параметры:
  - АДРЕС ТРУБЫ - состоит из 5 байт (по умолчанию) и может быть представлен числом типа uint64_t или массивом из 5 однобайтных элементов. Адрес трубы передатчика должен совпадать с одним из адресов труб приёмника.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - Открыть трубу необходимо до передачи данных функцией write().
  - Одновременно может быть открыта только одна труба для передачи данных, но допускается менять адрес трубы, что позволит отправлять данные поочерёдно разным получателям.
  - Не допускайте совпадение адресов труб передатчиков находящихся на одном канале. Так же не рекомендуется использование одинаковых адресов для рядом стоящих каналов, например, одинаковые адреса труб на каналах 25 и 26.

    - Альтернативным способом открытия трубы для передачи данных является обращение к функции openReadingPipe(), если в качестве номера трубы указать 0.
- Пример:

```
radio.openWritingPipe(0x1234567890);            // Открыть трубу для передачи данных указав её адрес числом.
```

- Пример:

```
uint8_t myAddr[5] = {0x12,0x34,0x56,0x78,0x90}; // Объявляем массив из 5 однобайтных элементов.
radio.openWritingPipe(myAddr);                  // Открыть трубу для передачи данных указав её адрес массивом.
```

- Пример:

```
uint8_t myAddr[6] = "1Node";                    // Объявляем массив из 6 однобайтных элементов в качестве которых выступают ко
radio.openWritingPipe(myAddr);                  // Открыть трубу для передачи данных указав её адрес массивом.
```

Функция openReadingPipe();

- Назначение: Открыть трубу для приёма данных.
- Синтаксис: openReadingPipe(№ ТРУБЫ , АДРЕС ТРУБЫ);
- Параметры:
    - № ТРУБЫ - число от 0 до 5.
    - АДРЕС ТРУБЫ - состоит из 5 байт (по умолчанию) и может быть представлен числом типа uint64_t или массивом из 5 однобайтных элементов. Адрес трубы приёмника должен совпадать с адресом трубы передатчика.
- Возвращаемое значение: Нет.
- Примечание:
    - Функция используется модулем в режиме приёмника.
    - До начала прослушивания труб функцией startListening() необходимо открыть хотя бы одну трубу для приёма данных.
    - Одновременно можно открыть до 6 труб для приёма данных, назначив им адреса и номера от 0 до 5.

- Труба с номером 0 используется как для чтения, так и для записи. Её адрес затирает адрес заданный функцией openWritingPipe().
- Адрес трубы с номером 0 может полностью отличаться от адресов труб с номерами 1-5.
- Если адреса трубам назначаются как числа типа uint64_t, то адреса труб с номерами 2-5 должны отличаться от адреса трубы с номером 1 только последним (младшим) байтом числа.
- Если адреса трубам назначаются как массивы, то адреса труб с номерами 2-5 должны отличаться от адреса трубы с номером 1 только первым элементом массива.
- Трубы с номерами 0 и 1 хранят полный 5 байтовый (по умолчанию) адрес, а трубы 2-5 технически хранят только 1 байт, заимствуя 4 дополнительных байта из адреса 1 трубы, не смотря на то, что вы задаёте им полный 5 байтовый адрес. По этому нельзя открывать трубы с номерами 2-5 если не открыта труба с номером 1.
- Трубы открытые для прослушивания можно закрывать функцией closeReadingPipe() и заново открывать указывая новый или старый адрес.
- Размер адресов труб можно уменьшить до 4 или 3 байт вызвав функцию setAddressWidth().

- Пример:

```
radio.openReadingPipe(0, 0x1234567890LL);              // Открыть трубу номер 0 с адресом 0x1234567890 для прослушивания (пр
radio.openReadingPipe(1, 0xAABBCCDD01LL);              // Открыть трубу номер 1 с адресом 0xAABBCCDD01 для прослушивания (пр
radio.openReadingPipe(2, 0xAABBCCDD0BLL);              // Открыть трубу номер 2 с адресом 0xAABBCCDD0B для прослушивания (пр
radio.openReadingPipe(3, 0xAABBCCDDC3LL);              // Открыть трубу номер 3 с адресом 0xAABBCCDDC3 для прослушивания (пр
```

- Пример:

```
uint8_t myAddr_0[5] = {0x12,0x34,0x56,0x78,0x90};      // Определяем массив состоящий из 5 однобайтных элементов. Все значен
uint8_t myAddr_1[5] = {0x01,0xAA,0xBB,0xCC,0xDD};      // Определяем массив состоящий из 5 однобайтных элементов. Массив отл
uint8_t myAddr_2[5] = {0x0B,0xAA,0xBB,0xCC,0xDD};      // Определяем массив состоящий из 5 однобайтных элементов. Массив отл
uint8_t myAddr_3[5] = {0xC3,0xAA,0xBB,0xCC,0xDD};      // Определяем массив состоящий из 5 однобайтных элементов. Массив отл
                                                       //
radio.openReadingPipe(0, myAddr_0);                    // Открыть трубу номер 0 с адресом myAddr_0 для прослушивания (приёма
radio.openReadingPipe(1, myAddr_1);                    // Открыть трубу номер 1 с адресом myAddr_1 для прослушивания (приёма
radio.openReadingPipe(2, myAddr_2);                    // Открыть трубу номер 2 с адресом myAddr_2 для прослушивания (приёма
```

```
radio.openReadingPipe(3, myAddr_3);                    // Открыть трубу номер 3 с адресом myAddr_3 для прослушивания (приёма
```

- Пример:

```
uint8_t myAddr[][6] = {"ABCDE","1Node","2Node","3Node"}; // Определяем массив состоящий из 4 строк по 6 байт, где первая строк
                                                        //
radio.openReadingPipe(0, myAddr[0]);                    // Открыть трубу номер 0 с адресом myAddr[0] для прослушивания (приём
radio.openReadingPipe(1, myAddr[1]);                    // Открыть трубу номер 1 с адресом myAddr[1] для прослушивания (приём
radio.openReadingPipe(2, myAddr[2]);                    // Открыть трубу номер 2 с адресом myAddr[2] для прослушивания (приём
radio.openReadingPipe(3, myAddr[3]);                    // Открыть трубу номер 3 с адресом myAddr[3] для прослушивания (приём
```

Функция closeReadingPipe();

- Назначение: Закрыть трубу открытую ранее для прослушивания (приёма данных).
- Синтаксис: closeReadingPipe(№ ТРУБЫ);
- Параметры:
  - № ТРУБЫ - от 0 до 5, которую более не требуется прослушивать.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме приёмника.
  - После закрытия одной из труб её можно снова открыть функцией openReadingPipe(), задав новый или старый адрес и номер.
- Пример:

```
radio.openReadingPipe (1, 0x1234567890LL); // Открыть трубу номер 1 с адресом 0x1234567890 для прослушивания (приёма данных).
radio.openReadingPipe (2, 0x1234567891LL); // Открыть трубу номер 2 с адресом 0x1234567891 для прослушивания (приёма данных).
radio.closeReadingPipe(2);                 // Закрыть трубу номер 2
radio.openReadingPipe (2, 0x1234567892LL); // Открыть трубу номер 2 с адресом 0x1234567892 для прослушивания (приёма данных).
```

Функция setChannel();

- Назначение: Установить радиочастотный канал связи.
- Синтаксис: setChannel(№ КАНАЛА);
- Параметры:
  - № КАНАЛА - указывается числом от 0 до 125.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - У общающихся приёмника и передатчика номер канала должен совпадать.
  - Номер канала определяет частоту на которой работает модуль. Каждый канал имеет шаг в 1 МГц, а каналу 0 соответствует частота 2,4 ГГц = 2400 МГц, следовательно, каналу 1 соответствует частота 2401 МГц, каналу 2 - частота 2402 МГц и т.д. до канала 125 с частотой 2525 МГц.
  - Канал приёма/передачи данных можно менять в процессе работы модуля.
  - Узнать номер используемого в данный момент времени канала можно обратившись к функции getChannel().
  - Стоит учитывать что если функцией setDataRate() установлена скорость передачи данных по радиоканалу в 2 Мбит/сек, то модуль использует сразу два канала, установленный и следующий за ним.
  - Перед выбором канала для приема/передачи данных рекомендуем воспользоваться функцией testCarrier() и/или testRPD(), которые позволят Вам убедиться что указанные каналы не используются другими (сторонними) устройствами. Ведь некоторые устройства, WiFi и даже микроволновая печь создаёт сильные помехи на определённых каналах используемого модулем ISM диапазона.
- Пример:

```
radio.setChannel(35); // Указываем модулю использовать 35 канал (частота 2,435 ГГц).
```

Функция getChannel();

- Назначение: Получить номер текущего радиочастотного канала связи.
- Синтаксис: getChannel();
- Параметры: Нет.
- Возвращаемое значение: uint8_t - номер канала, число от 0 до 125.

- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
- Пример:

```
uint8_t i = radio.getChannel(); // Получить номер используемого канала в переменную i.
```

Функция setDataRate();

- Назначение: Установить скорость передачи данных по радиоканалу.
- Синтаксис: setDataRate(СКОРОСТЬ);
- Параметры:
  - СКОРОСТЬ - задаётся одной из констант:
    - RF24_1MBPS - 1 Мбит/сек.
    - RF24_2MBPS - 2 Мбит/сек.
    - RF24_250KBPS - 250 Кбит/сек (только для модуля NRF24L01+PA+LNA).
- Возвращаемое значение: bool - флаг успешной установки новой скорости (true / false).
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - У общающихся приёмника и передатчика скорость приёма/передачи данных должна совпадать.
  - Узнать текущую скорость передачи данных можно обратившись к функции getDataRate().
  - Константы принимаемые в качестве параметра функции определены в библиотеке как элементы перечисления с типом rf24_datarate_e.
- Пример:

```
radio.setDataRate(RF24_1MBPS); // Установить скорость приёма/передачи данных в 1 Мбит/сек.
```

Функция getDataRate();

- Назначение: Получить текущую скорость передачи данных по радиоканалу.
- Синтаксис: getDataRate();
- Параметры: Нет.

- Возвращаемое значение: значение одной из констант сопоставленной скорости:
  - RF24_1MBPS - 1 Мбит/сек.
  - RF24_2MBPS - 2 Мбит/сек.
  - RF24_250KBPS - 250 Кбит/сек (только для модуля NRF24L01+PA+LNA).
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функция возвращает значение одной из констант которые определены в библиотеке как элементы перечисления с типом rf24_datarate_e.
- Пример:

```
switch( radio.getDataRate() ){                              // Выбрать действие зависящее от значения возвращённого функцией ra
    case RF24_1MBPS   : Serial.println("  1 MBPS"); break; // Если полученное значение совпало с константой RF24_1MBPS  , то в
    case RF24_2MBPS   : Serial.println("  2 MBPS"); break; // Если полученное значение совпало с константой RF24_2MBPS  , то в
    case RF24_250KBPS : Serial.println("250 KBPS"); break; // Если полученное значение совпало с константой RF24_250KBPS, то в
}                                                           //
```

Функция setPALevel();

- Назначение: Установить уровень усиления мощности передатчика.
- Синтаксис: setPALevel(УРОВЕНЬ);
- Параметры:
  - УРОВЕНЬ - задаётся одной из констант:
    - RF24_PA_MIN - минимальный уровень усиления = -18 дБм.
    - RF24_PA_LOW - низкий уровень усиления = -12 дБм.
    - RF24_PA_HIGH - высокий уровень усиления = -6 дБм.
    - RF24_PA_MAX - максимальный уровень усиления = 0 дБм.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.

- Если модуль находится в режиме приёмника, то по умолчанию он отправляет передатчику пакеты подтверждения приёма, а следовательно приёмнику приходится указывать мощность передатчика.
- У общающихся приёмника и передатчика уровень усиления усилителя мощности должен совпадать, ведь если пакеты подтверждения от приёмника к передатчику будут отправляться с меньшей мощностью, то передатчик их может и не получить.
- Узнать текущий уровень усиления мощности можно обратившись к функции getPALevel().
- Константы принимаемые в качестве параметра функции определены в библиотеке как элементы перечисления с типом rf24_pa_dbm_e.

- Пример:

```
radio.setPALevel(RF24_PA_MAX); // Установить максимальный уровень усиления усилителя мощности передатчика.
```

Функция getPALevel();

- Назначение: Получить текущий уровень усиления мощности передатчика.
- Синтаксис: getPALevel();
- Параметры: Нет.
- Возвращаемое значение: значение одной из констант сопоставленной мощности:
  - RF24_PA_MIN - минимальный уровень усиления = -18 дБм.
  - RF24_PA_LOW - низкий уровень усиления = -12 дБм.
  - RF24_PA_HIGH - высокий уровень усиления = -6 дБм.
  - RF24_PA_MAX - максимальный уровень усиления = 0 дБм.
  - RF24_PA_ERROR - уровень усиления не определён.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функция возвращает значение одной из констант которые определены в библиотеке как элементы перечисления с типом rf24_pa_dbm_e.
- Пример:

```
switch( radio.getPALevel() ){                       // Выбрать действие зависящее от значения возвращённого функцией radio.
    case RF24_PA_MIN   : Serial.println("MIN"); break; // Если полученное значение совпало с константой RF24_PA_MIN   , то выв
```

```
    case RF24_PA_LOW   : Serial.println("LOW"); break; // Если полученное значение совпало с константой RF24_PA_LOW   , то выв
    case RF24_PA_HIGH  : Serial.println("HIG"); break; // Если полученное значение совпало с константой RF24_PA_HIGH  , то выв
    case RF24_PA_MAX   : Serial.println("MAX"); break; // Если полученное значение совпало с константой RF24_PA_MAX   , то выв
    case RF24_PA_ERROR : Serial.println("ERR"); break; // Если полученное значение совпало с константой RF24_PA_ERROR , то выв
}                                                     //
```

Функция setCRCLength();

- Назначение: Установить размер CRC (циклически избыточный код).
- Синтаксис: setCRCLength(РАЗМЕР);
- Параметры:
  - РАЗМЕР - задаётся одной из констант:
    - RF24_CRC_8 - под CRC отводится 8 бит (CRC-8).
    - RF24_CRC_16 - под CRC отводится 16 бит (CRC-16).
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - У общающихся приёмника и передатчика размер CRC должен совпадать.
  - CRC (Cyclic redundancy check) алгоритм нахождения контрольной суммы, предназначенный для проверки целостности данных.
  - Узнать текущий размер CRC можно обратившись к функции getCRCLength().
  - Отключить отправку и проверку CRC можно обратившись к функции disableCRC().
  - Константы принимаемые в качестве параметра функции определены в библиотеке как элементы перечисления с типом rf24_crclength_e.
- Пример:

```
radio.setCRCLength(RF24_CRC_16); // Использовать CRC-8.
```

Функция getCRCLength();

- Назначение: Получить текущий размер CRC (циклически избыточный код).

- Синтаксис: getCRCLength();
- Параметры: Нет.
- Возвращаемое значение: значение одной из констант сопоставленной размеру CRC:
  - RF24_CRC_8 - под CRC отводится 8 бит (CRC-8).
  - RF24_CRC_16 - под CRC отводится 16 бит (CRC-16).
  - RF24_CRC_DISABLED - передача и проверка CRC отключены.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функция возвращает значение одной из констант которые определены в библиотеке как элементы перечисления с типом rf24_crclength_e.
- Пример:

```
switch( radio.getCRCLength() ){                                  // Выбрать действие зависящее от значения возвращённого функци
    case RF24_CRC_8        : Serial.println("  CRC-8 "); break; // Если полученное значение совпало с константой RF24_CRC_8
    case RF24_CRC_16       : Serial.println("  CRC-16"); break; // Если полученное значение совпало с константой RF24_CRC_16
    case RF24_CRC_DISABLED : Serial.println("DISABLED"); break; // Если полученное значение совпало с константой RF24_CRC_DISA
}                                                                //
```

Функция disableCRC();

- Назначение: Отключить передачу CRC передатчиком и проверку данных приёмником.
- Синтаксис: disableCRC();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика.
- Пример:

```
radio.disableCRC(); // Отключить передачу CRC передатчиком и проверку данных приёмником.
```

Функция setPayloadSize();

- Назначение: Установить статичный размер блока данных пользователя в байтах.
- Синтаксис: setPayloadSize(РАЗМЕР);
- Параметры:
  - РАЗМЕР - блока данных пользователя в байтах.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика.
  - По умолчанию используется максимальный статичный размер блока данных в 32 байта.
  - Чем меньше статичный размер блока данных пользователя, тем короче пакет данных передаваемый по радиочастотному каналу и, как следствие, его передача происходит быстрее. Размер данных передаваемых функцией write() не может превышать установленный статичный размер. Если же в функции write() указать размер выше установленного максимального статичного размера, то передаваемые данные будут урезаны.
  - Если вызвать функцию enableDynamicPayloads() разрешающую динамически изменяемый размер блока данных пользователя для всех труб, то статичный размер заданный функцией setPayloadSize() учитываться не будет.
- Пример:

```
radio.setPayloadSize(24); // Установить размер блока данных в 24 байта. Теперь это максимальный размер передаваемых данных.
```

Функция getPayloadSize();

- Назначение: Получить текущий статичный размер блока данных пользователя в байтах.
- Синтаксис: getPayloadSize();
- Параметры: Нет.
- Возвращаемое значение: uint8_t - текущий статичный размер блока данных от 0 до 32 байт.
- Примечание:

    - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
- Пример:

```
uint8_t i = radio.getPayloadSize(); // Получить текущий размер блока данных пользователя в переменную i.
```

Функция getDynamicPayloadSize();

- Назначение: Получить размер блока данных в последнем принятом пакете.
- Синтаксис: getDynamicPayloadSize();
- Параметры: Нет.
- Возвращаемое значение: uint8_t - размер данных последнего принятого пакета в байтах.
- Примечание:
    - Функция используется модулем в режиме приёмника.
    - Функция возвращает размер блока данных пользователя в последнем принятом пакете, только если разрешено использование динамически изменяемых размеров блока данных функцией enableDynamicPayloads().
    - Максимальный динамический размер блока данных пользователя не может превышать 32 байта.
    - Если на стороне приёмника функция available() вернула true, а функция getDynamicPayloadSize() вернула значение меньше 1, значит блок данных принятого пакета был повреждён.
- Пример:

```
byte  myData[32];                                   // Объявляем массив для получения данных.
uint8 mySize;                                       // Объявляем переменную для получения размера принятого блока пользовате
if( radio.available() ){                            // Если в буфере имеются принятые данные, то ...
    mySize = radio.getDynamicPayloadSize();         // Читаем размер блока данных принятого пакета в переменную mySize.
    radio.read( &myData, sizeof(myData) );          // Читаем данные из буфера в массив myData указывая сколько всего байт м
}                                                   //
```

Функция enableDynamicPayloads();

- Назначение: Разрешить динамически изменяемый размер блока данных для всех труб.

- Синтаксис: enableDynamicPayloads();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика.
  - Действие функции распространяется как на данные пользователя отправляемые передатчиком функцией write(), так и на данные пользователя отправляемые приёмником в качестве ответа передатчику функцией writeAckPayload().
  - Разрешив динамически изменяемый размер блока данных, размер отправляемых пакетов будет зависеть от размера передаваемых данных.
- Пример:

```
radio.enableDynamicPayloads(); // Разрешаем динамически изменяемый размер блока данных для всех труб.
```

Функция enableDynamicAck();

- Назначение: Разрешить отказываться от запроса пакетов подтверждения приёма.
- Синтаксис: enableDynamicAck();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - Если разрешить отказываться от запроса пакетов подтверждения приёма, то далее на стороне передатчика можно обращаться к функциям write(), writeFast(), startWrite() или startFastWrite() с установленным флагом групповой передачи данных (с указанием последнего параметра установленного в true). При этом передатчик отправляет данные одним пакетом (без повторов), а приёмник получивший данные отправленные таким образом не ответит передатчику пакетом подтверждения приёма данных. Следовательно, можно организовать сеть состоящую из одного передатчика и неограниченного количества приёмников на одном канале и одной трубе.
- Пример:

```
int data[16] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5}; // Определяем массив для передачи данных.
radio.enableDynamicAck();                          // Разрешаем отказываться от запроса пакетов подтверждения приёма.
radio.write( &data , sizeof(data) , true );        // Отправляем данные из массива data, указывая весь размер массива в байтах,
```

Функция enableAckPayload();

- Назначение: Разрешить размещать данные пользователя в пакете подтверждения приёма.
- Синтаксис: enableAckPayload();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика.
  - Обращение к функции enableAckPayload() разрешает приёмнику размещать данные пользователя в ответе передатчику используя функцию writeAckPayload(), а передатчику сообщает что в ответах приёмника могут быть данные пользователя.
- Пример:

```
radio.enableAckPayload();                                  // Указываем что в пакетах подтверждения приёма есть блок с пользов
radio.openReadingPipe (1, 0xAABBCCDD11LL);                 // Открываем 1 трубу с адресом 0xAABBCCDD11, для приема данных.
radio.startListening  ();                                  // Включаем приемник, начинаем прослушивать открытые трубы.
radio.writeAckPayload (1, &ackData, sizeof(ackData) );     // Помещаем данные всего массива ackData в буфер FIFO. Как только б
```

Функция setAutoAck();

- Назначение: Управление автоматической отправкой пакетов подтверждения приёма данных.
- Синтаксис: setAutoAck([№ ТРУБЫ ,] ФЛАГ);
- Параметры:
  - № ТРУБЫ - для которой разрешается / запрещается автоматическая отправка пакетов подтверждения приема. Указывается только на

стороне приёмника. Если № трубы на стороне приёмника не указан, то действие функции распространяется на все трубы.
    - ФЛАГ - разрешающий автоматическую отправку пакетов подтверждения приёма данных. true - разрешить / false - запретить.
- Возвращаемое значение: Нет.
- Примечание:
    - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
    - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика с указанием одинакового флага.
    - Если приёмник получает от передатчика данные с запросом подтверждения приёма, то приёмник, по умолчанию, автоматически отправляет в ответ передатчику пакет подтверждения приема его данных. Данная функция способна запретить приемнику отправлять пакеты подтверждения приема, как на все, так и только на указанные трубы. Эту же функцию необходимо вызвать и на стороне передатчика с тем же флагом что и у приёмника, согласовав их работу.
- Пример:

```
radio.setAutoAck( 1 , false ); // Запретить приёмнику отправлять пакеты подтверждения приема передатчику использующему адрес 1
```

Функция setAddressWidth();

- Назначение: Указать длину адресов труб в байтах.
- Синтаксис: setAddressWidth(РАЗМЕР);
- Параметры:
    - РАЗМЕР - адреса трубы в байтах, представлен числом 3, 4 или 5.
- Возвращаемое значение: Нет.
- Примечание:
    - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
    - Функцию необходимо вызвать и у приёмника, и у общающегося с ним передатчика с указанием одинакового размера, до указания адресов труб функциями openWritingPipe() и openReadingPipe().
    - По умолчанию используются адреса имеющие размер 5 байт (40 бит).
    - Установка ширины адреса 3 байта (24 бита) или 4 байта (32 бита) приведёт к уменьшению размера пакета на 2 или 1 байт соответственно.

- Пример:

```
radio.setAddressWidth(3);           // Использовать адреса размером 3 байта.
radio.openReadingPipe(1, 0x123456); // Открыть трубу номер 1 с адресом 0x123456 для прослушивания (приёма данных). Адрес данно
radio.openReadingPipe(2, 0x1234AA); // Открыть трубу номер 2 с адресом 0x1234AA для прослушивания (приёма данных). Адрес данно
```

Функция setRetries();

- Назначение: Указать максимальное количество попыток отправки данных и время ожидания.
- Синтаксис: setRetries(ВРЕМЯ , КОЛИЧЕСТВО);
- Параметры:
  - ВРЕМЯ - целое число от 0 до 15 определяющее время ожидания подтверждения приема.
  - КОЛИЧЕСТВО - целое число от 1 до 15 определяющее максимальное количество попыток доставить данные передатчику.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - По умолчанию передатчик отправляет данные приёмнику с CRC и запросом подтверждения приема данных. Приёмник получив данные проверяет CRC и если он совпал рассчитанным, то отправляет передатчику пакет подтверждения приема, на этом передача считается успешно завершена. Но если приемник не получил данные (выключен или находится вне радиуса действия), или рассчитанный CRC не совпал с отправленным передатчиком, то приёмник не отправит передатчику пакет подтверждения приема. Если в течении времени ожидания подтверждения приема передатчик не получит ответ от приёмника, то он повторит отправку данных, продолжая свои попытки пока не получит ответ от приёмника, или пока не исчерпает количество попыток отправки данных.
  - Время ожидания пакета подтверждения приема задается целым числом от 0 до 15 из которого реальное время рассчитывается по формуле: время = (число+1) * 250 мкс.
  - Количество попыток доставки данных соответствует указанному целому числу от 1 до 15.
- Пример:

```
radio.setRetries(5,10); // Указываем передатчику что он должен ждать пакет подтверждения приёма в течении (5+1)*250 = 1500 мкс
```

Функция powerDown();

- Назначение: Перейти в режим пониженного энергопотребления.
- Синтаксис: powerDown();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - В режиме пониженного энергопотребления модуль сохраняет способность принимать и передавать данные, но с меньшим энергопотреблением. Для выхода из режима пониженного энергопотребления вызовите функцию powerUp().
- Пример:

```
radio.powerDown(); // Перейти в режим пониженного энергопотребления.
```

Функция powerUp();

- Назначение: Выйти из режима пониженного энергопотребления.
- Синтаксис: powerUp();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Для входа в режим пониженного энергопотребления вызовите функцию powerDown().
- Пример:

```
radio.powerUp(); // Выйти из режима пониженного энергопотребления.
```

Функция isPVariant();

- Назначение: Проверить аппаратную совместимость модуля с функциями nRF24L01.

- Синтаксис: isPVariant();
- Параметры: Нет.
- Возвращаемое значение: bool - (true / false) флаг указывающий на совместимость аппаратного обеспечения модуля с функциями чипа nRF24L01+.
- Примечание:
  - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Приобретая модуль nRF24L01+ не из линейки Trema, существует вероятность получить модуль на базе чипа отличного от nRF24L01+, который может не поддерживаться настоящей библиотекой. Если функция isPVariant() вернёт true значит можно с большой долей вероятности сказать что Ваш модуль собран на базе чипа nRF24L01+.
- Пример:

```
if( radio.isPVariant() ){ Serial.println("OK"); } // Вывести текст "OK" если чип nRF24L01+.
```

Функция writeFast();

- Назначение: Быстро отправить данные по радиоканалу.
- Синтаксис: writeFast(ДАННЫЕ , РАЗМЕР [, ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ]);
- Параметры:
  - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить.
  - РАЗМЕР - отправляемых данных в байтах.
  - ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ - установите в true если требуется отправить данные нескольким приёмникам.
- Возвращаемое значение: bool - результат записи данных в буфер для передачи (true / false).
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - Данная функция принимает те же параметры что и функция write() и назначение обеих функций одинаково - отправить данные. Но функция writeFast() в отличии от функции write() не ждёт доставки данных приёмнику, а помещает их в один из трёх буферов FIFO для передачи данных. Сама передача выполняется аппаратно.
  - Если все три буфера FIFO заполнены, то функция writeFast() будет ждать пока не освободится один из буферов или не истечёт время ожидания, которое много меньше ожидания функции write(). Время ожидания можно указать, если вместо функции writeFast()

использовать функцию writeBlocking().
    - Функция writeFast(), в отличии от функции write(), возвращает не флаг успешной доставки данных, а флаг успешной записи данных в один из трёх буферов FIFO.
    - Узнать статус отправки данных по радиоканалу можно обратившись к функции txStandBy() сразу после функции writeFast().
- Пример:

```
int data[16] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5}; // Определяем массив для передачи данных.
radio.writeFast( &data , sizeof(data) );          // Отправляем в буфер для передачи данные из массива data указывая весь разм
```

Функция writeBlocking();

- Назначение: Быстро отправить данные по радиоканалу с указанием таймаута.
- Синтаксис: writeBlocking(ДАННЫЕ , РАЗМЕР , ВРЕМЯ);
- Параметры:
    - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить.
    - РАЗМЕР - отправляемых данных в байтах.
    - ВРЕМЯ - максимальное время ожидания освобождения буфера FIFO в миллисекундах.
- Возвращаемое значение: bool - результат записи данных в буфер для передачи (true / false).
- Примечание:
    - Функция используется модулем в режиме передатчика.
    - Первые два параметра данной функции совпадают с параметрами функции writeFast() и назначение обеих функций одинаково - поместить данные в один из трёх буферов FIFO для передачи. Сама передача выполняется аппаратно.
    - Отличием функции writeBlocking() от функции writeFast() является то, что функция writeBlocking() ждёт освобождения буфера (если они заняты) в течении указанного ей времени ожидания.
    - Узнать статус отправки данных по радиоканалу можно обратившись к функции txStandBy() сразу после функции writeBlocking().
- Пример:

```
int data[16] = {0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5}; // Определяем массив для передачи данных.
radio.writeBlocking( &data , sizeof(data) ,  10); // Отправляем в буфер для передачи данные из массива data указывая весь разм
```

Функция startFastWrite();

- Назначение: Начать быструю отправку данных.
- Синтаксис: startFastWrite(ДАННЫЕ , РАЗМЕР , ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ [, startTx]);
- Параметры:
  - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить.
  - РАЗМЕР - отправляемых данных в байтах.
  - ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ - установите в true если требуется отправить данные нескольким приёмникам.
  - startTx - флаг перехода в режим TX или STANDBY-II. Если не указан, значит установлен.
- Возвращаемое значение: Нет.
- Примечание:
  - Функция является дополнительной и используется модулем в режиме передатчика.
  - Данная функция используется функциями write(), writeFast() и writeBlocking() для начала отправки данных. Но разработчик библиотеки решил дать доступ к этой функции в тестовых и отладочных целях.
  - Если флаг startTx установлен или отсутствует, то функция осуществляет переход в режим TX или STANDBY-II, для выхода из которого нужно вызвать функцию txStandBy().
- Пример: отсутствует.

Функция startWrite();

- Назначение: Начать отправку данных.
- Синтаксис: startWrite(ДАННЫЕ , РАЗМЕР , ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ);
- Параметры:
  - ДАННЫЕ - адрес массива, строки или переменной, данные которой требуется отправить.
  - РАЗМЕР - отправляемых данных в байтах.
  - ФЛАГ ГРУППОВОЙ ПЕРЕДАЧИ - установите в true если требуется отправить данные нескольким приёмникам.
- Возвращаемое значение: Нет.
- Примечание:

- Функция является дополнительной и используется модулем в режиме передатчика.
- Функция работает как startFastWrite() с установленным флагом startTx, но в отличии от неё, выходит из режима TX или STANDBY-II.

- Пример: отсутствует.

Функция txStandBy();

- Назначение: Подождать пока передаются данные и вернуть результат.
- Синтаксис: txStandBy([ВРЕМЯ ОЖИДАНИЯ]);
- Параметры:
  - ВРЕМЯ ОЖИДАНИЯ - максимальное время ожидания указывается в миллисекундах.
- Возвращаемое значение: bool - результат передачи данных из буферов FIFO в радиоканал (true / false).
- Примечание:
  - Функция используется модулем в режиме передатчика.
  - Функция ожидает завершение передачи данных из буферов FIFO в радиоканал, но не дольше указанного времени ожидания. Если время ожидания не указано, то за максимальное время ожидания берётся время MAX_RT.
  - По завершении функция очищает буферы FIFO и переходит в режим STANDBY-I (выходит из режимов TX или STANDBY-II).
- Пример:

```
radio.writeFast( &data , sizeof(data) ); // Отправляем в буфер для передачи данные из массива data указывая весь размер массив
radio.writeFast( &data , sizeof(data) ); // Отправляем в буфер для передачи данные из массива data указывая весь размер массив
radio.writeFast( &data , sizeof(data) ); // Отправляем в буфер для передачи данные из массива data указывая весь размер массив
bool i = txStandBy(1000);                // К моменту вызова данной функции все три буфера FIFO заполнены, ждём завершения пер
```

Функция rxFifoFull();

- Назначение: Проверить не заполнены ли все три буфера FIFO.
- Синтаксис: rxFifoFull();
- Параметры: Нет.
- Возвращаемое значение: bool - флаг указывающий на то что все буферы FIFO заполнены.
- Примечание:

    - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
    - Функция возвращает treu только если все три 32-байтных радиобуфера FIFO заполнены.
    - Функция может использоваться для предотвращения потери данных связанных с заполненностью буферов.
- Пример:

```
bool i = radio.rxFifoFull(); // Устанавливаем i в значение true если все три буфера FIFO заполнены.
```

Функция flush_tx();

- Назначение: Очистка буферов FIFO.
- Синтаксис: flush_tx();
- Параметры: Нет.
- Возвращаемое значение: uint8_t - значение регистра состояния.
- Примечание:
    - Функция используется модулем как в режиме приёмника, так и в режиме передатчика.
- Пример:

```
radio.flush_tx(); // Очистить буферы FIFO.
```

Функция reUseTX();

- Назначение: Повторная отправка данных из буфера FIFO, если они там есть.
- Синтаксис: reUseTX();
- Параметры: Нет.
- Возвращаемое значение: Нет.
- Примечание:
    - Функция является дополнительной и используется модулем в режиме передатчика.
    - Данная функция используется функциями writeFast() и writeBlocking() для повторных попыток отправки данных. Но разработчик библиотеки решил дать доступ к этой функции в тестовых и отладочных целях.
- Пример: отсутствует.

Функция testCarrier();

- Назначение: Проверка наличия несущей частоты на выбранном канале (частоте).
- Синтаксис: testCarrier();
- Параметры: Нет.
- Возвращаемое значение: bool - наличие несущей на выбранном канале за все время его прослушивания.
- Примечание:
  - Функция используется модулем в режиме приёмника, или после режима приёмника.
  - Функцию можно использовать для проверки помех при выборе или смене канала. Если несущая обнаружена, значит на данном канале работает кто то другой и этот канал выбирать не стоит.
  - При прослушивании канала для определения на нём сторонних устройств рекомендуется отключить автоматическую отправку подтверждения приёма данных вызвав функцию setAutoAck(false), чтоб не нарушать работу этих устройств.
  - Функция не определяет факт наличия несущей в момент её вызова, а возвращает внутренний флаг библиотеки который устанавливается если за всё время прослушивания хоть раз была обнаружена несущая частота выбранного канала.
  - Функция будет постоянно возвращать true если несущая была хоть раз обнаружена, пока не выйти из режима приёма функцией stopListening() и опять не начать прослушивание функцией startListening().
  - Для обнаружения сторонних устройств можно использовать и функцию testRPD(), которая отличается от данной функции тем, что возвращает true при обнаружении только мощного сигнала на выбранном канале, а не любой несущей.
- Пример:

```
radio.setAutoAck(false);      // Указываем модулю не отправлять пакеты подтверждения приёма данных. Чтоб не «удивлять» передат
radio.setChannel(27);         // Устанавливаем номер канала.
radio.startListening();       // Начинаем прослушивание любых труб, так как их адрес не указан.
delayMicroseconds(225);       // Ждём 255 микросекунд, чтоб модуль успел принять несущую от других устройств (время подбираетс
radio.stopListening();        // Завершаем прослушивание.
bool i = radio.testCarrier(); // Получаем флаг наличия несущей на выбранном канале в переменную i.
```

Функция testRPD();

- Назначение: Проверка наличия любого сигнала выше -64 дБм на выбранном канале (частоте).
- Синтаксис: testRPD();
- Параметры: Нет.
- Возвращаемое значение: bool - наличие сигнала мощностью выше -64 дБм на выбранном канале за все время его прослушивания.
- Примечание:
  - Функция используется модулем в режиме приёмника, или после режима приёмника.
  - Функцию можно использовать для проверки помех при выборе или смене канала. Если на выбранном канале обнаружен относительно мощный сигнал (выше -64 дБм), значит на данном канале работает кто то другой и этот канал выбирать не стоит.
  - При прослушивании канала для определения на нём сторонних устройств рекомендуется отключить автоматическую отправку подтверждения приёма данных вызвав функцию setAutoAck(false), чтоб не нарушать работу этих устройств.
  - Функция не определяет факт наличия мощного сигнала в момент её вызова, а возвращает внутренний флаг библиотеки который устанавливается если за всё время прослушивания канала на его частоте был хоть раз обнаружен сигнал мощностью выше -64 дБм.
  - Функция будет постоянно возвращать true если сигнал с уровнем более -64 дБм был хоть раз обнаружен, пока не выйти из режима приёма функцией stopListening() и опять не начать прослушивание функцией startListening().
  - Данная функция, в отличии от функции testCarrier(), работает только с модулями на базе чипа nRF24L01+ и не работает с nRF24L01.
- Пример:

```
radio.функция(параметр); // Комментарий редактируется.
```

Функция isValid();

- Назначение: Проверить используется ли модуль или выполняется отладка кода.
- Синтаксис: isValid();
- Параметры: Нет.
- Возвращаемое значение: bool - назначение редактируется (true / false).
- Примечание:
  - Функция является дополнительной и используется модулем как в режиме приёмника, так и в режиме передатчика.
  - Функция используется во время отладки кода и возвращает true если код не отлаживается, а передача и приём данных ведутся в нормальном режиме (по радиоканалу).

    - Если при объявлении объекта библиотеки вместо номера вывода CE или SS было указано число 255 или 0xFF, то библиотека будет работать в режиме отладки кода (данные не будут приниматься или передаваться модулем по радиоканалу, в то время как все функции библиотеки будут вести себя так как будто модуль корректно подключён).
- Пример:

```
if( !radio.isValid() }{ /* Выполняется отладка кода */ ;}
```

Применение:

Модуль можно использовать для создания беспроводных клавиатур, манипуляторов, джойстиков, систем безопасности и оповещения, систем домашней автоматизации, систем телеметрии, беспроводных датчиков и т.д.

Ссылки:

- Библиотека RF24.