# nrf24l01 Library for PIC32MX250F128B

By Douglas Katz, Fred Kummer

# Table of Contents

# Introduction

The goal of this project was to develop a library for the PIC32MX250F128B to allow it to easily be used with the nrf24l01+ radio modules. These radio modules are cheap, widely available, and highly flexible, but using them without a library is a challenging task, requiring a large number of options and settings to be properly configured by adjusting the registers of the radio. This library removes the need for users to interact directly with the radio registers, instead allowing them to configure the radio through a series of simple functions. It allows the radio to be set to operate in a wide variety of modes, including varying power levels, different frequencies, static and dynamic payload sizes, and with or without auto-acknowledgement. The library also simplifies the process of sending and receiving radio payloads, allowing the user to send and receive strings of bytes of variable size without needing to consider the specifics of how the radio handles them. The ultimate goal of this library is to allow users to quickly and easily add high-speed wireless communication to any project.

The documentation for this project is divided into three sections. The first part is the User's Manual. This section assumes no familiarity with the radio modules, and focuses exclusively on how to use the library and not the technical details of how it works. In this section are descriptions of the radios features, explanations of the terminology used, schematics, descriptions of the functions available in the library, and example code along with detailed walkthroughs of the code. These are intended for a user who wants to quickly start using the library without concern for the technical details of its implementation.

The second section is the Technical Description. This section covers in greater depth the inner workings of the radio and the specifics of how the library was implemented. It includes details on the registers of the radio and the specifics bits that must be set to control each setting as well as greater detail on how transmitting and receiving are handled. This section is recommended for users interested in modifying this library or just gaining a deeper understanding of the nrf24l01+ module.

The final part is the API, generated with Doxygen. This contains a full list of all functions included in the library and descriptions of what their purpose is, their parameters, and their return values.

# User's Manual

## Hardware Setup

The radio requires an SPI connection, an interrupt pin, and a connection to Vdd and ground. The radio's SCK needs to be connected to SCK1 or SCK2 on the PIC, MOSI needs to be connected to an SPI Data In 1 or SPI Data In 2 pin, and MISO needs to be connected to an SPI Data In 1 or SPI Data In 2. The 1 or 2 for each of these three pins depends on whether SPI1 or SPI 2 is being used on the PIC and the number must be the same for all three. A list of the radio pins and their recommended connections is shown in Table 1 and a schematic of the recommended pins is shown in Figure 2. Figure 1 below shows a pinout of the radio module, showing which pins are referred to by which numbers.

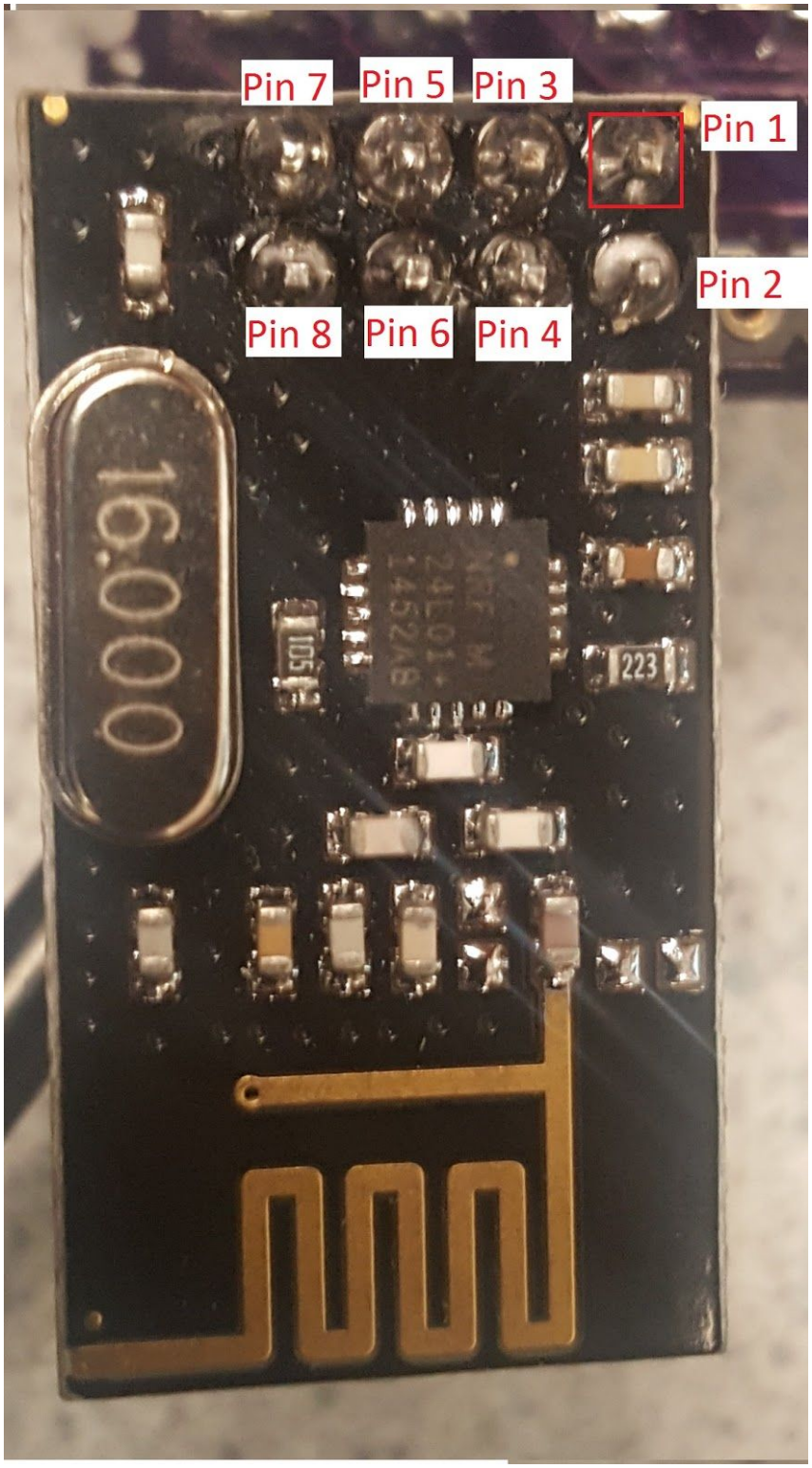


**Figure 1:** Radio Module Pinout

| Radio Pin Number | Name | PIC Connection |
|---|---|---|
| 1 | GND | GND |
| 2 | Vcc | Vcc |
| 3 | CE | Pin 17 (RB8 I/O) |
| 4 | CSN | Pin 18 (RB9 I/O) |
| 5 | SCK | Pin 26 (SPI SCK 2) |
| 6 | MOSI | Pin 9 (RPA2 SPI2 Data In) |
| 7 | MISO | Pin 12 (RPA4 SPI2 Data Out) |
| 8 | IRQ | Pin 21 (RPB10 External Interrupt 1) |

**Table 1:** Recommended Radio Pin Connections

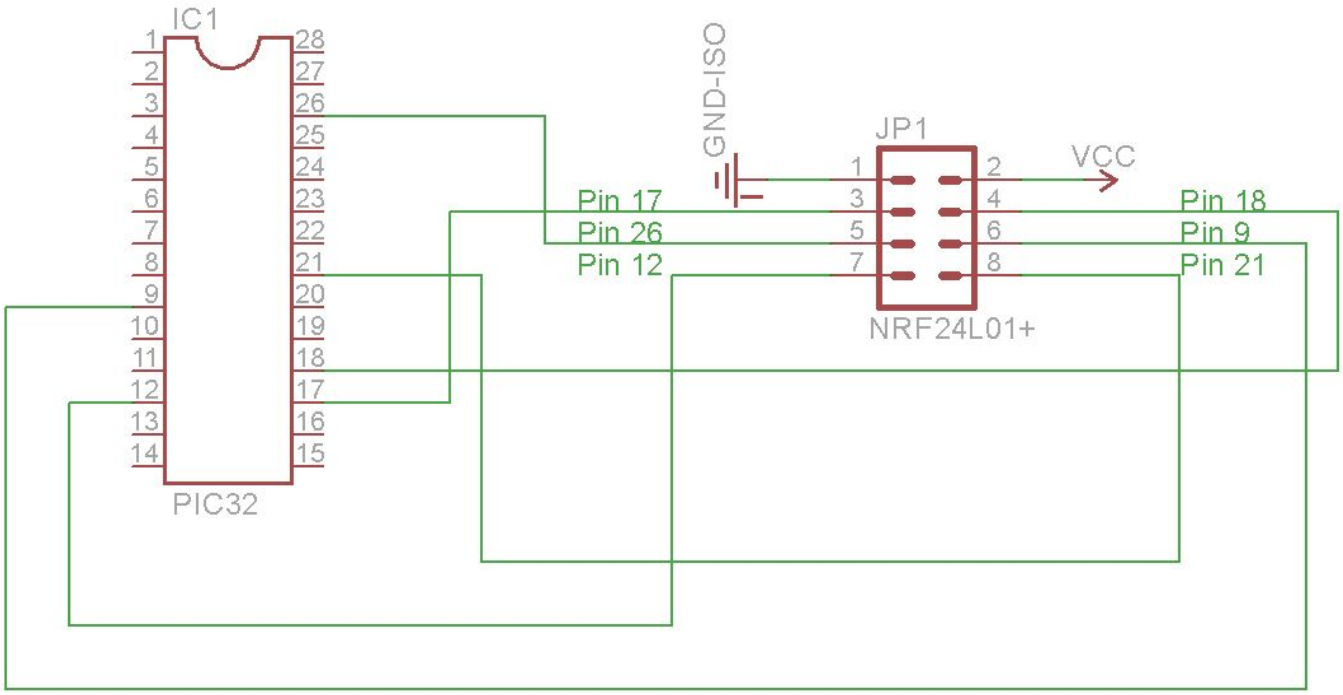


**Figure 2:** Schematic for Basic Radio Connections

Other pins can be used but various parts of the library code must be changed. To use SPI1 instead of the default SPI2 the following must be changed in the init_SPI and fr_spiwrite functions:

```
SpiChnOpen(2, SPI_OPEN_MSTEN | SPI_OPEN_MODE8 | SPI_OPEN_ON | SPI_OPEN_CKE_REV, 16);

PPSInput(3, SDI2, RPA4);

PPSOutput(3, RPA2, SDO2);

char rf_spiwrite(unsigned char c){
    while (TxBufFullSPI2());
    WriteSPI2(c);
    while( !SPI2STATbits.SPIRBF);
    return ReadSPI2();
}
```

This will allow the radio's SCK (pin 5) to be connected to a the SPI1 pin. The radio's MOSI (pin 6) can be connected to any pin that can be set to be an SPI Data In on the PIC. If this is done the following code must be changed in the init_SPI function:

```
PPSInput(3, SDI2, RPA4);
```

The radio's MISO (pin7) can be connected to any pin that can be set to SPI Data Out on the PIC. If this is done the following code must be changed in the init_SPI function:

```
PPSOutput(3, RPA2, SDO2);
```

The radio's IRQ (pin 8) can be changed by modifying the following code in the nrf_setup function:

```
PPSInput(4, INT1, RPB10);
```

More pins can be used by changing which interrupt the radio will trigger on the PIC. The default interrupt used is External Interrupt 1 but others can be used by changing the following code in the nrf_setup function and the INT1Handler ISR:

```
PPSInput(4, INT1, RPB10);

ConfigINT1(EXT_INT_PRI_2 | FALLING_EDGE_INT | EXT_INT_ENABLE);
EnableINT1;

void __ISR(_EXTERNAL_1_VECTOR, ipl2) INT1Handler(void){

mINT1ClearIntFlag();
```

The radio's CSN (pin 4) and CE (pin 3) can be set to any I/O pin on the PIC. To do this the following code must be changed in nrf24l01.h file:

```
#define _csn         LATBbits.LATB9
#define TRIS_csn     TRISBbits.TRISB9

#define _ce         LATBbits.LATB8
#define TRIS_ce     TRISBbits.TRISB8
```

To learn the details of how the code must be changed refer to the PIC's manual and datasheet.

## Software Setup

Before calling any other radio functions nrf_setup must be called. This function will initialize SPI for communicating with the radio, setup the radio interrupt, setup I/O pins, and reset all register values on the radio. To reset the radio the nrf_reset can be called which will reset all the registers in the radio to their default values.

## Frequency

The nrf24l01+ can be set to operate on a range of frequencies from 2.4 GHz to 2.525 GHz. The frequency can be set with nrf_set_rf_ch function. The frequency is set according to the equation, frequency = 2400 + ch (MHz), where ch is the input to the nrf_set_rf_ch function. The transmitter and receiver must be set operate at the same frequency to communicate. As an example, the following line of code would set the frequency to 2.401 GHz.

```
nrf_set_rf_ch(0x01); // freq = 2.401 GHz
```

## Addresses and Pipes

The nrf24l01+ uses addresses on the receiver and transmitter to identify which nrf24l01+ module packets are meant for. This allows packets to be directed to certain modules instead of being read by every module in range. Addresses have to be set on both the receiver and transmitter to allow proper operation. The specifics for the setting addresses for receivers and transmitters are detailed below.

For receiving, the nrf24l01+ contains six pipes that data can be received on numbered 0-5. Each of these pipes has a 3-5 byte address field that determines which packets are received on that pipe. Each pipe can have a different address, allowing them each to receive data from a different module. All pipes have the same address length, so if you set the address length to 3, all pipes will have a length of 3. The length of the address can be set to be 3-5 bytes using the nrf_set_address_width function. The address of a pipe can be set using the nrf_set_rx_addr. The first two pipes allow all five bytes of their addresses to be set while the other four pipes only allow the least significant byte of their address to be set. Pipes 2-5 use the four most significant bytes of pipe 1's address. So when using the nrf_set_rx_addr function on Pipes 2-5, only one byte should be entered as the address. When a payload is received the pipe it was received on is stored and can be read using the nrf_get_pipe function.

For transmission, each radio has a TX address as well. This sets the address which this radio transmits with. If the TX address of a radio sending a packet matches the address of a pipe on a receiving radio the packet will be received by the pipe with the matching address. So in order to send to a pipe on a receiver that has its address set to 0xAAAAAA, the TX address of the transmitter must be set to 0xAAAAAA as well. The address of the transmitter be set using the nrf_set_tx_addr function.

## Dynamic and Static Payload Lengths

The payload is the data sent between the transmitting and receiving radios. The payload can vary in size from 1 to 32 bytes. There are two ways to configure the length of payloads. The length can be set to be static, in which case all payloads are of the same length,or the payloads can be set to be dynamic, in which case the length of each payload can change.

By default static payload lengths are enabled on the radio. With static lengths the length of the payload sent by the transmitter must match the static payload length set on the receiving pipe. The size of the payload does not need to be explicitly set on the transmitter, that is simply determined by the size of the data that is sent. On the receiving end the length must be set explicitly for each pipe. The payload size for a receiving pipe is set by using the nrf_set_pw function. The payload widths can be set to be 1-32 bytes. If the length of the payload sent does not match the length set on the receiver in static payload length mode packets will not be received correctly. Static payload widths are enabled by default, but to use them after using dynamic payload lengths the nrf_dis_dpl function is used which disables dynamic payload lengths and re-enables static payload lengths. Note that nrf_get_payload_width, which is detailed in the Receiving section, should not be used when static payloads are being used.

Dynamic payload lengths allow packets to be sent and received without having to specify the size of the packet beforehand. In order to use dynamic payloads auto-acknowledge mode must also be enabled, which is detailed in the Auto-Acknowledge section below. Because of this, the nrf_en_dpl function automatically enables auto-acknowledge mode on its pipe. Dynamic payload lengths should never be used in pipes that are not using auto-acknowledge mode. Dynamic payload lengths is enabled on a per pipe basis. The function nrf_en_dpl will enable dynamic payloads on a pipe and will enable auto-acknowledge on a pipe as necessary. For a transmitter to transmit dynamic payloads, nrf_en_dpl must be called on pipe 0 of the transmitter. In addition to this the transmitter must have its TX address and pipe 0 address set to be the same as the receiving pipe's address. The pipe 0 address on the transmitter must be explicitly set using the nrf_set_rx_addr function to allow auto-acknowledge, which is needed for dynamic payloads. Refer to the Auto-Acknowledge section below for more detail. On the receiving end, dynamic payloads must be set individually using nrf_en_dpl on each pipe that wishes to use them.

Dynamic payloads can be disabled on a pipe using the nrf_dis_dpl function which will not disable auto_acknowledge for a pipe.

When a dynamic payload is received its size in bytes is stored and can be read using the nrf_get_width function.

Below are snippets of code demonstrating how to set pipe 5 for static and dynamic payloads.

Static Payload

Receiver:

```
nrf_set_pw(1, 5); //set the payload width to be 1 byte
```

Dynamic Payload

Receiver:

```
nrf_set_rx_addr(5, 0xE7E7E7E7E7, 5); //Set pipe 0 address
nrf_en_dpl(5); //Set pipe 5 to accept dynamic payloads.
```

Transmitter:

```
nrf_set_rx_addr(0, 0xE7E7E7E7E7, 5); //Set pipe 0 address
nrf_set_tx_addr(0xE7E7E7E7E7);
nrf_en_dpl(0); //Pipe 0 must be set to auto-ack on transmitter
```

## Auto-Acknowledge

The nrf24l01 has an auto-acknowledge features that allows a transmitter to determine if a packet was successfully received. When auto-acknowledge is enabled the transmitter will send the packet and then wait to receive an acknowledgement packet from the transmitter. When the receiver successfully receives the packet it will automatically go into transmit mode and send an acknowledgement packet, and then return to receiving automatically . If the transmitter receives the acknowledgement packet it will signal that the packet was received successfully by returning a 1 from the nrf_send_payload function. If it does not receive an acknowledgement the nrf_send_payload function will return a 0.

The transmitter will retransmit packets multiple times if a packet is not acknowledged. The amount of times the packet is retransmitted is set using the nrf_set_arc function. The transmitter can be set to retransmit 0-15 times. There is a delay between retransmits that can be set by using the nrf_set_ard function. The delay is set to 250 + 250*ARD, where ARD is the number passed as a parameter to the nrf_set_ard function, μs and is measured from when the time a packet is finished transmitting to the time the next packet begins being transmitted. The

radio times out after transmitting the set amount of times and assumes the packet was not received.

Auto-acknowledge mode is enabled on a per pipe basis. Enabling or disabling auto-acknowledge for a pipe is done using the nrf_en_aa and nrf_dis_aa functions. The nrf_en_aa function will also enable receiving on a pipe automatically.

The transmitter must enable auto-acknowledge for pipe 0. In addition to this the transmitter must have its TX address and pipe 0 address set to be the same as the receiving pipe's address. Addresses for the necessary RX and TX pipes can be set using the nrf_set_tx_addr or nrf_set_rx_addr functions. For example if receiving on pipe 5 the addresses can be set up according to Table 2.

| | |
|---|---|
| Transmitter: TX address | 0xE7E7E7E7E7 |
| Transmitter: Pipe 0 address | 0xE7E7E7E7E7 |
| Receiver: Pipe 5 address | 0xE7E7E7E7E7 |

**Table 2:** Example Auto-Acknowledge Address Setup

This setup will allow a packet sent to pipe 5 on the receiver to be acknowledged. Below is some example code for setting up this configuration.

Receiver Code:

```
nrf_set_rx_addr(5, 0xE7E7E7E7E7, 5); //Set pipe 5 address
nrf_en_aa(5); // enable autoack on pipe 5
```

Transmitter Code:

```
nrf_set_arc(0x0A); // 10 retransmits
nrf_set_ard(0x00); //250 us delay
nrf_en_aa(0); //Transmitter must enable auto ack on pipe 0
nrf_set_rx_addr(0, 0xE7E7E7E7E7, 5); //Set pipe 0 address
nrf_set_tx_addr(0xE7E7E7E7E7);
```

## Transmit Power

The power the radio transmits at can also be setr using the nrf_set_transmit_pwr function. The possible power settings are 0 dBm, -6dBm, -12dBm, or -18dBm. Macros to set the different power levels are contained in the library, and their use is highly recommended. The name of each macro and the power it corresponds to is summarized in Table 3 below.

| Macro Name | Power Level |
|---|---|

| | |
|---|---|
| nrf24l01_RF_SETUP_RF_PWR_0 | 0dBm |
| nrf24l01_RF_SETUP_RF_PWR_6 | -6dBm |
| nrf24l01_RF_SETUP_RF_PWR_12 | -12dBm |
| nrf24l01_RF_SETUP_RF_PWR_18 | -18dBm |

**Table 3:** Macros for Power Levels

## Data Rate

The data rate can be set by using the nrf_set_transmit_rate function. The possible data rates are 250 kbps, 1 Mbps, or 2 Mbps. If the rate is set to 2 Mbps the radio will operate in a 2 Mhz bandwidth instead of a 1 Mhz bandwidth. Macros to set the different data rates are contained in the library, and their use is highly recommended. The name of each macro and the power it corresponds to is summarized in Table 4 below.

| Macro Name | Data Rate |
|---|---|
| nrf24l01_DR_LOW | 250 Kbps |
| nrf24l01_DR_MED | 1 Mbps |
| nrf24l01_DR_HIGH | 2 Mbps |

**Table 4:** Macros for Data Rates

## Transmitting

The nrf24l01+ can send up to 32 bytes of data in a packet. To send a packet, the desired number of bytes to send must be written into a buffer. A pointer to the buffer along with the length of the buffer must then be passed to the nrf_send_payload function. This function automatically handles transmitting. Note that after calling this function the radio will no longer be in receive mode, and so packets cannot be received without explicitly returning the module to receive mode using the nrf_state_rx_mode function. Below is a snippet of example code transmitting a single byte.

```
char pay = 0xAA; //Byte to be transmitted
nrf_send_payload(&pay,1);//Send payload in pay of length 1
```

## Receiving

The radio can receive up to 32 bytes of data per packet on any of its six pipes. To receive the radio must be put in receive mode using the nrf_state_rx_mode function. Once in this mode the radio will wait to receive packets. If a packet is received, the

nrf_payload_available function will return 1. If no packet is available, nrf_payload_available will return a 0.

When a payload is available, the nrf_get_pipe function will return what pipe the data came in on and the nrf_get_payload_width function will return the length of the received payload. If these characteristics of the received payload are required, they should be obtained using these functions prior to reading the payload. Note that nrf_get_payload_width should never be used when using static payloads, as it will not return the appropriate widths.

The payload can be read into a buffer using the nrf_get_payload function. Note that using the nrf_get_payload function sets the payload as no longer available, so nrf_payload_available will return a 0 prior to reading the payload.

Note that it is necessary to check nrf_payload_available before calling the nrf_get_payload function, as trying to read a payload when none is available can return meaningless or repeated data.

Also note that after a payload is received, that payload must be read using nrf_get_payload prior to the next payload being received. If a new payload arrives without the first payload being read, the first payload will be lost. So it is important to closely monitor payload availability and quickly read available packets. Below is a snippet of code showing a very simple setup for receiving:

```
int length;
int pipe;
nrf_state_rx_mode();
while(1){
	if(nrf_payload_available()){
		length = nrf_get_payload_width();//Using dynamic payloads
		pipe = nrf_get_pipe();
		char buff[length];
		nrf_get_payload(&buff, pipe);
	}
}
```

## Example Code

### *Counter.c*

This code implements a synchronous counter using two PIC32s both connected to TFT displays and nrf24l01+ radios. The code is driven by radio interrupts where each PIC will wait until data is received before any action is taken. The ID field for the two PICs must be set before running. The first PICs ID must be set to 0 and the second PICs ID must be set to 1.

The value of a counter is stored in variable counter and is initially set to zero. Radio 0 begins by attempting to send counter to radio 1. Until radio 0 receives an acknowledgement it

will continue to send counter and also blink a circle on screen to show that the program is still running.

Once radio 0 gets an acknowledgement from radio 1, radio 0 will go into receive mode and the counting will start. Radio 0 will increment the counter and send it to radio 1 which will display the count and send it back to radio 0. Radio 0 will receive the count back from radio 1 and will then display it, increment it, and send the updated count to radio 1. Radio 1 will repeat what it did before and the loop will then continue indefinitely.

The counter will then continue to increment as fast as the radios can send and the screens
can update.

Schematics for setting up the radios for this code using a Microstick or a standalone PIC is shown in Figure 3 and Figure 4.

Line-by-Line Explanation

It is recommended that a radio setup function is made that will setup any desired options on the radio. Counter.c starts this radio_setup function with:

```
nrf_setup();
```

Which initializes the radio and must be called before the radio can be used.

Next the function sets up how many times it should try to retransmit packets if they are not acknowledged:

```
nrf_set_arc(0x0A);
```

This will cause the radio to attempt to resend a packet 10 times before it will assume the packet will not be received and signals this to the PIC. This can be set higher for more reliable transmission or lower to find out that a packet didn't get sent sooner.

After this the delay between sending each of these packets is set using:

```
nrf_set_ard(0x00);
```

This sets the delay to 250 μs which is found using the equation $delay = 250 + arg \times 250\ \mu s$. This can be set higher to give the receiver more time to send the auto-ack packet or at a low amount find out that a packet was not sent sooner.

After this the frequency the radio will transmit on is set using:

```
nrf_set_rf_ch(0x01);
```

This function sets the frequency to 2.401 MHz which is found using the following equation $frequency = 2400 + arg\ MHz$. Different frequencies can be used to avoid interference with other devices transmitting on the same frequency. The value above is not chosen for any specific reason.

The radio enables auto-acknowledge next using:

```
nrf_en_aa(0);
```

This function will enable auto-acknowledge for pipe 0. This is done for two reasons. First, a transmitter must have auto-acknowledge enabled on pipe 0 for auto-acknowledgement to work. Second, the radio also will be receiving on pipe 0 and the receiving pipe must have auto-acknowledge enabled for it. If the code was going to receive on a pipe other than pipe 0 this function would need to be called with the argument being the receiving pipe.

The length a payload will be is set next using:

```
nrf_set_pw(1, 0);
```

This function sets up pipe 0 of the radio to receive payloads that are one byte long. Other pipes could be set to receive packets of different lengths if desired but for this code only pipe 0 is being used so the length is only set for the one pipe. The radio will ignore payloads that are not the length specified by this function so it is important to set this for any receiving pipes that will be used. The length is set to be one byte here because the counter that is being sent between the radios is only one byte long. Longer lengths can be used to send more data per packet or shorter lengths can allow packets to be transmitted faster.

The addresses of the radio are set up last by first using:

```
nrf_set_address_width(5);
```

This sets how long addresses can be for both the transmitter and the receiving pipes. The code sets the address width to be 5 bytes long. Because the width was set to 5 bytes 5 byte addresses can be written to pipes 0 or 1 after this. The other pipes still only allow one byte to be written and how this works is explains in the Addresses and Pipes section. The actual addresses are set last using:

```
nrf_set_rx_addr(0, 0xAABBCCDDFF, 5);
nrf_set_tx_addr(0xAABBCCDDFF);
```

The first function sets the address of the receiving pipe 0 to 0xAABBCCDDFF. Pipe 0 will ignore any messages that were sent from transmitters with different addresses. After this the TX address is also set to 0xAABBCCDDFF. This will cause receiving pipes with the same

address to grab any packets sent by this transmitter. Because auto-acknowledge is enabled pipe 0's address and the TX address must be set to be the same. Other pipes may have different address however. Both PIC and radio pairs are programmed to have the same address so that the currently receiving pair will capture the packet sent by the currently transmitting pair because their addresses match.

Main calls this setup function and a few other functions that set up the TFT screen. In the while loop the main part of the program runs. It starts by declaring an ID as 0 or 1. This is not part of the radio library operation but one of the PICs must set be 1 and the other to 0 for the operation of Counter.c. PIC 0 will start the communication between the two radios first by calling:

```
while (!nrf_send_payload(&counter, 1))
```

The send payload function will attempt to transmit the one byte of data stored in counter to the other radio. If the packet of data is successfully sent PIC 1's radio will acknowledge it by sending back an acknowledgement packet. PIC 0's radio will automatically check for the acknowledgement packet. If PIC 0 receives this acknowledgement packet it will know that the original transmitted packet was received and the send payload function will return 1 which will exit the while loop. If no acknowledgement packet is received the function will instead return 0 and then blink a circle on the screen before trying to send the data again. This will make sure the two PICs and their radios will switch between transmitter and receiver at the correct times in the future.

PIC 0 will skip this first while loop and instead immediately go into receive mode by calling:

```
nrf_state_rx_mode();
```

This function will put the radio in receive mode where it can detect packets sent and receive them if addresses, frequency, and various other options match. After receive mode is entered the code enters a while loop where it will do nothing but update the display until a packet is received. The code checks for a packet by polling using:

```
if (nrf_payload_available())
```

This function will return 1 when a payload is available to be read. If no payload is available the function will return 0 and the if statement body will not be entered.

In the if statement body the code first clears the display and then reads the new packet using:

```
nrf_get_payload(&counter, 1);
```

This function reads 1 byte of data into the variable counter and now the payload available function will not return 1 anymore until a new packet is received. The code then updates the display with the received count value and PIC 0 will increment its count if it is the one running this section of the code. After this the code will will send the updated counter value back to the other PIC by calling:

```
nrf_send_payload(&counter, 1);
```

This will send a packet and check for an acknowledgement packet in the same way as described above. This send only attempts the retransmissions that were set up in the radio setup function and there is no code in Counter.c that will check if this transmission was successful. This will cause the radio to deadlock if the message does not go through. This function will also cause the radio to exit receive mode so after it is called so after sending this the radio will need to be put into receive mode again using:

```
nrf_state_rx_mode();
```

After the code has done this it will wait for another packet to be received and check for it by polling the payload available function again. This operation will continue until a transmission does not go through in which case both radios will enter receive mode and neither will transmit causing a deadlock because no payload will ever be available. It may be desirable to use a while loop as the first send payload function did if a piece of code should block until a packet is successfully sent and received. The way the second send payload is implemented can be used if the code should continue whether or not the payload is received.

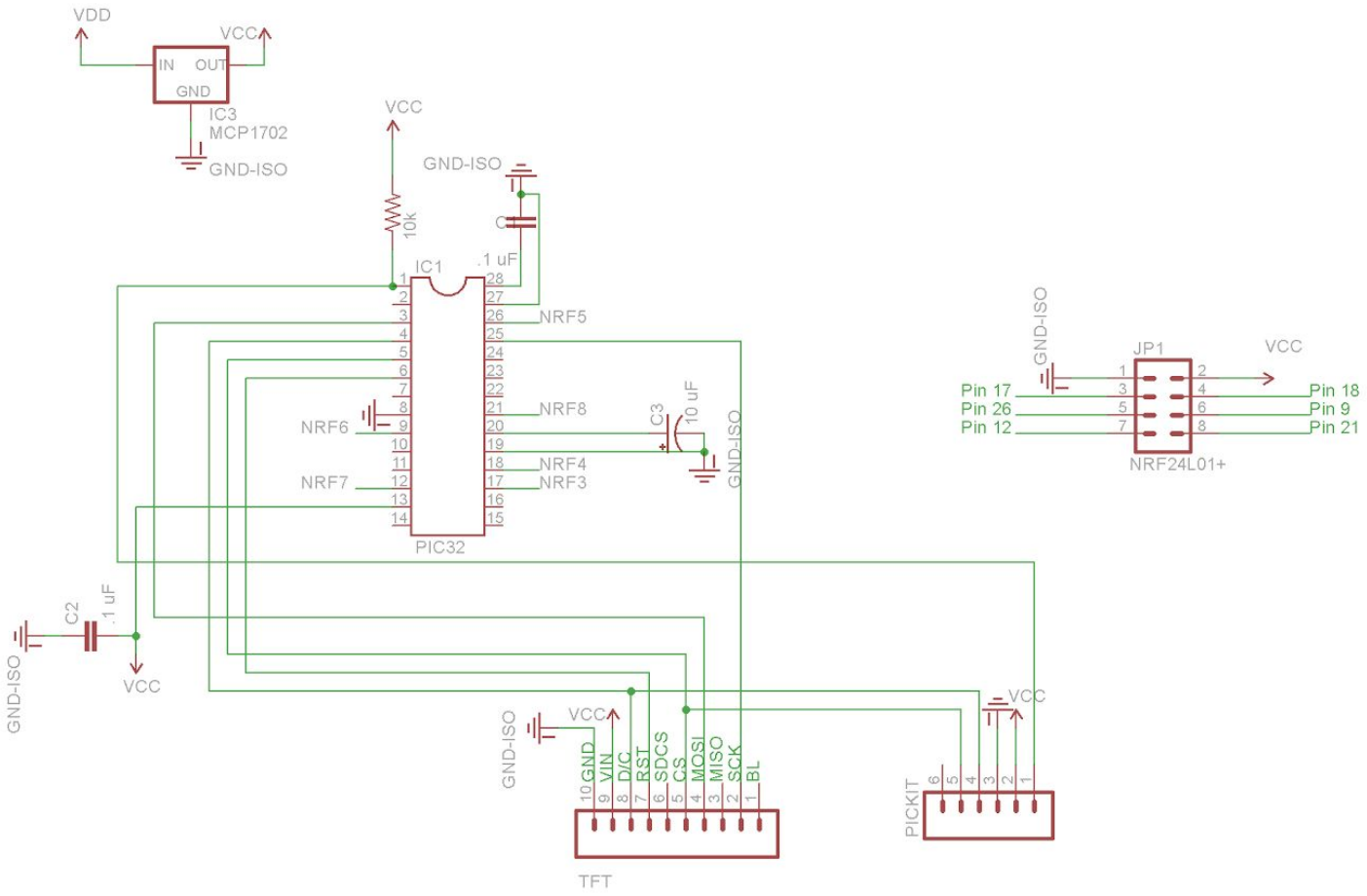


**Figure 3:** Counter Schematic with Standalone PIC and PICKIT

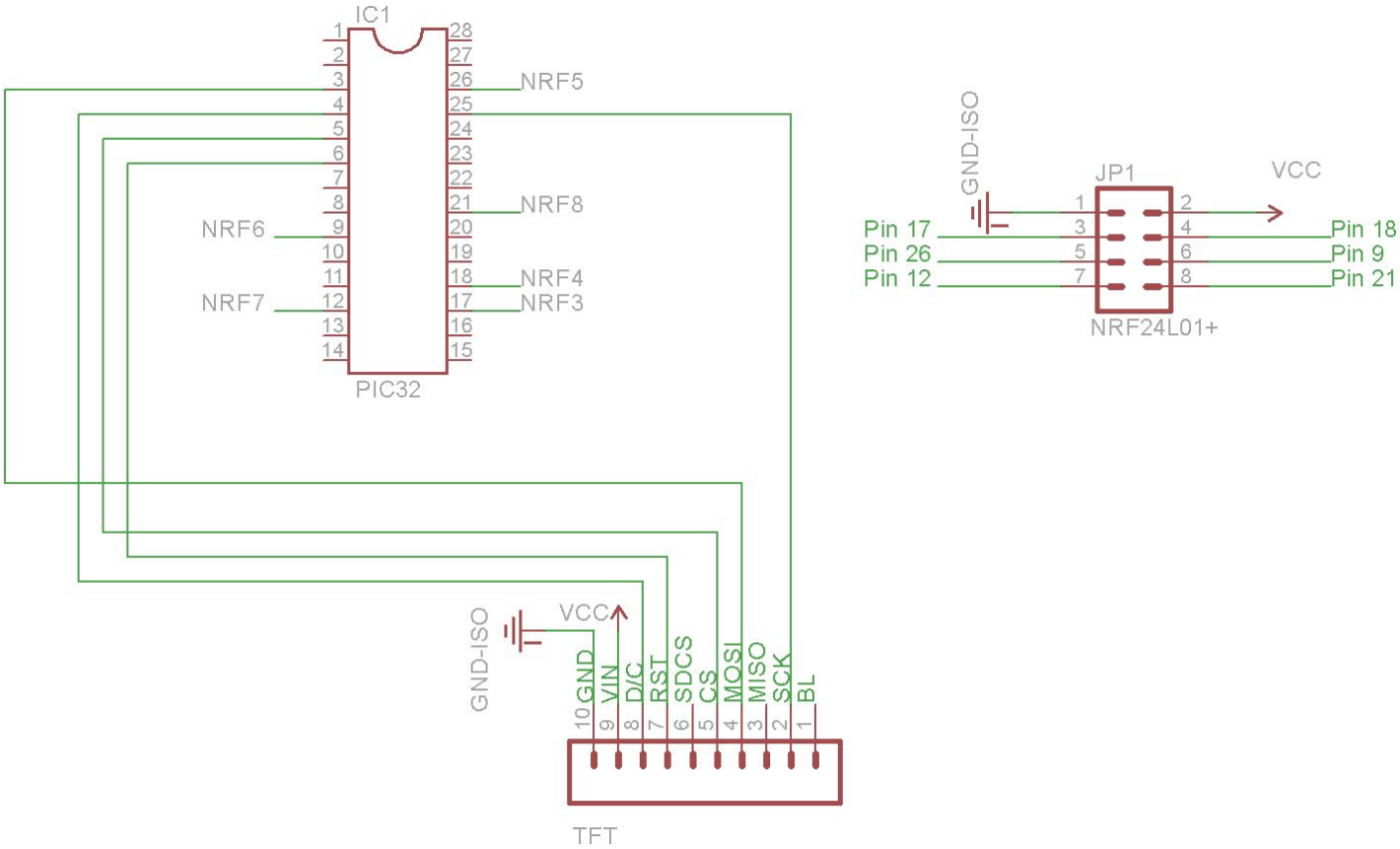


**Figure 4:** Counter Schematic for Microstick

***TX_carrier.c and RX_carrier.c***

TX_Carrier sends a continuous carrier wave out. RX_Carrier looks for the wave and turns on and off two LEDs to signal whether or not the wave is detected. If the wave is found LEDGREEN is turned on and LEDRED is turned off. If the wave is not found LEDGREEN is turned off and LEDRED is turned on. The radio only detects whether the received power is above -64 dB or not and a bit is set in the RPD register of the radio depending on this power level. This power is only checked for on a specific frequency.

Schematics for setting up the radios for this code using a Microstick or a standalone PIC is shown in Figure 5 **and** Figure 6**.**

The TX_carrier program starts by setting up the radio. It call nrf_setup to initialize the radio first. It then sets the frequency it will transmit at. This frequency must be the same on the transmitter and the receiver. This is all that is needed to set up the radio for sending the continuous carrier wave. The carrier wave is then sent with a power level of 0 dB. The wave will be sent until the PIC is powered down by having the code stay in an endless while loop.

The RX_carrier wave also starts by setting up the radio. It will initialize and set the frequency of the radio just as the transmitter does. After this is done the radio then goes into

RX mode to detect the carrier wave and also sets the two LEDs as outputs. The setup is finished at this point and the code begins to run. Nrf_received_pwr is called to check if the wave is being detected. If the wave is detected the green LED turns on and the red LED turns off. If the wave is not detected the green LED turns off and the red LED turns on. This will continue indefinitely.

This set of programs is helpful for checking if the radio is connected correctly as it only requires two LEDs to be connected to the PIC in addition to the radio. Even at the lowest power level the receiver should still detect the wave if the two radios are within a foot of each other. If the green LED is off in this case the radios are likely connected to the PICs incorrectly. These two programs can also be used for detecting the maximum reliable range of the radios at a given power level and frequency. The power level can be changed in the nrf_start_cont_wave function and the frequency can be changed in the nrf_set_rf_ch.

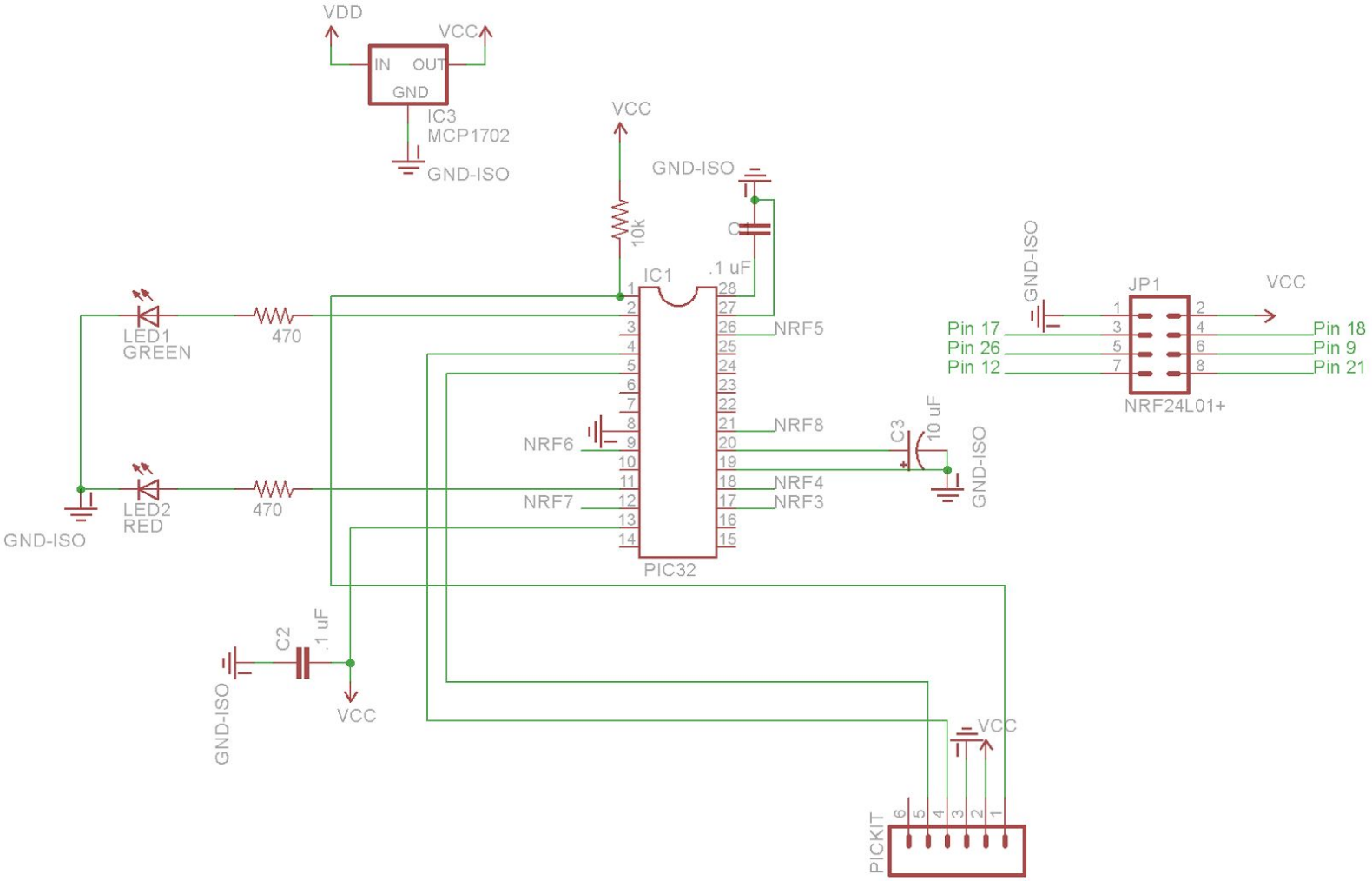


**Figure 5:** Carrier Schematic with Standalone PIC and PICKIT

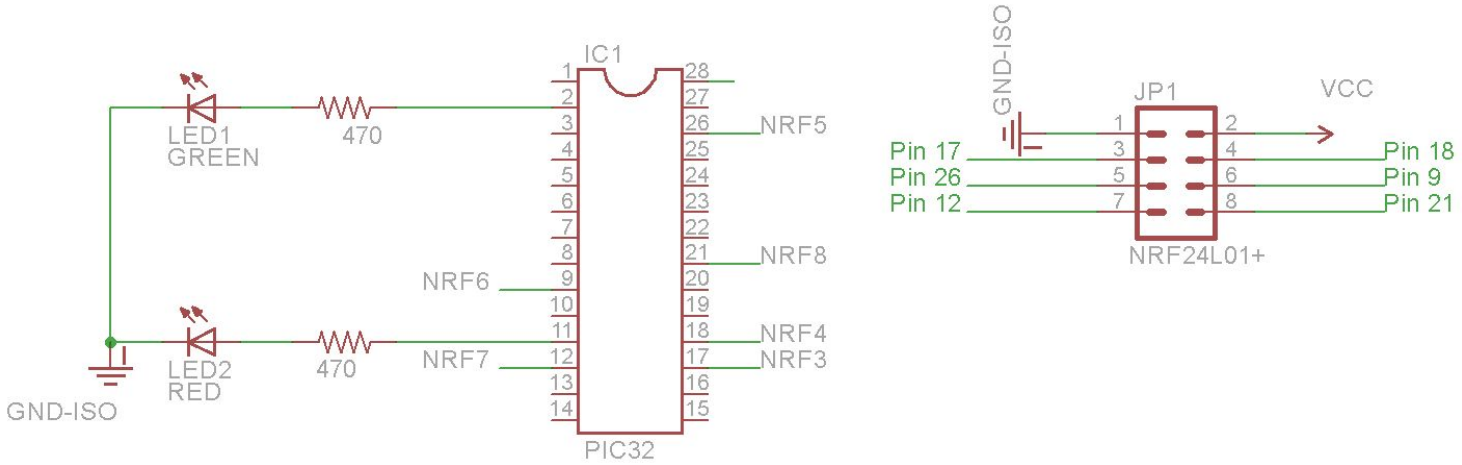


**Figure 6:** Carrier Schematic with Microstick

***dynPldRX.c and dynPldTX.c***

These two programs show off how to send and receive with dynamic payloads lengths enabled. The transmitter runs dynPldTX.c and sends packets of differing lengths made from the same array. The receiver runs dynPldRX which will receive these packets and display the received bytes of data and the length of the payload. The transmitter requires no external hardware other than the radio. The receiver requires a TFT display to be connected to the PIC.

The transmitter starts by running a function to set up the radio. This setup function first sets the radio to attempt to retransmit packets 5 time if an acknowledgement is not received and sets the radio to wait for 250 µs between each of these transmissions. Next the radio sets the frequency it will transmit on to 2401 GHz. After this the radio enables dynamic payload lengths on pipe 0 using:

```
nrf_en_dpl(0);
```

Dynamic payload lengths must be enabled on pipe zero in order to enable transmitting these types of payloads. No other pipes have this setting enabled as the radio running this code will only be transmitting and not receiving. This will also enable auto-acknowledgements because this must be enabled for dynamic payload lengths to work. Finally addresses are setup to use three byte addresses. The transmitting address and the pipe 0 address are set to be the same because auto-acknowledge mode is enabled.

After setup is finished the code creates a 32 byte array and fills it with data. The data in the array is set to be incrementing integers from 0-31. After this the code begins sending pieces of this array over the radio. First the first byte of the array is sent meaning only a 1 will be received by the transmitter. Next, the first 5 bytes of the array are sent so {0,1,2,3,4} will be

received. Afterwards the first 10 bytes of the array are sent so {0,1,2,3,4,5,6,7,8,9,10} will be received. Finally all 32 bytes of the array will be sent over the radio and then the code will resend everything again in a loop.

The receiver also runs a setup function for the radio. This function first sets the frequency to be 2401 GHz to match the transmitter. Next the function enables dynamic payloads lengths on pipe 1 so it can receive dynamic payloads on this pipe. This also enables auto-acknowledge mode on pipe 1 automatically. After this addresses are set on pipe 1 to be the same as on the transmitter.

Once radio setup is finished the code sets up the display and declares an array for received data to be read into and a variable width for the width of data to be read into. After this the radio is put into receive mode to begin looking for packets. When a packet is found the code checks its width using:

```
width = nrf_get_width();
```

This width means that the radio and code do not need to know the length of the data they will receive beforehand and knowing this width is helpful for working with this data. The code then uses the found width to properly read the received packet into the array declared earlier using:

```
nrf_get_payload(&array, width);
```

The whole array will not necessarily be used and only the first “width” bytes contain the data received. The received data and the width of the data in bytes is then displayed on the TFT. The received data is displayed in blue although data contained in bytes greater than 14 is cut off because of display size. The width of the received packet is displayed in red. The code will then wait for the next packet to be sent.

Dynamic payload lengths can make sending and receiving data easier by not requiring the receiver to know the length of the data being transmitted. The receiver can check the length of the data coming in and work with it easily using this mode.

# Technical Description

## Hardware Setup

The nrf24l01+ has eight pins, SCK, SDI, SDO, IRQ, CSN, CE, PWR. GND, and VCC. VCC is connected to 1.9-3.3V and GND is connected to ground. SCK is the SPI clock pin is connected to one of the two SPI clock pins on the PIC32. MISO is connected to an SDI and MOSI is connected to an SDO pin on the MCU. IRQ is an interrupt pin and is connected to any pin that allows external interrupts. CSN is the SPI chip select and CE is the chip enable and each can be connected to any I/O pin. CSN is driven low whenever data is being sent over SPI and is set high again once the communication has finished. CE is driven high when the radio is supposed to be receiving or sending data over radio.

For our setup we connected CE to pin 17 and connected CSN to pin 18. We then used a define directive for each in our code to make it easier to set or clear the pin. We connected SCK to pin 26 which is the SCK2 pin on the MCU. MOSI was connected to pin 13 and MISO to pin 12 and these pins were set to be SPI data in and SPI data out. IRQ was connected to pin 21 and this pin was set to be external interrupt 1. Refer to Figure 7 below for a schematic detailing the connections to the PIC the radio must have. Note that this only shows the PIC to radio connections, and to run a stand-alone PIC would require additional supporting hardware for the PIC.

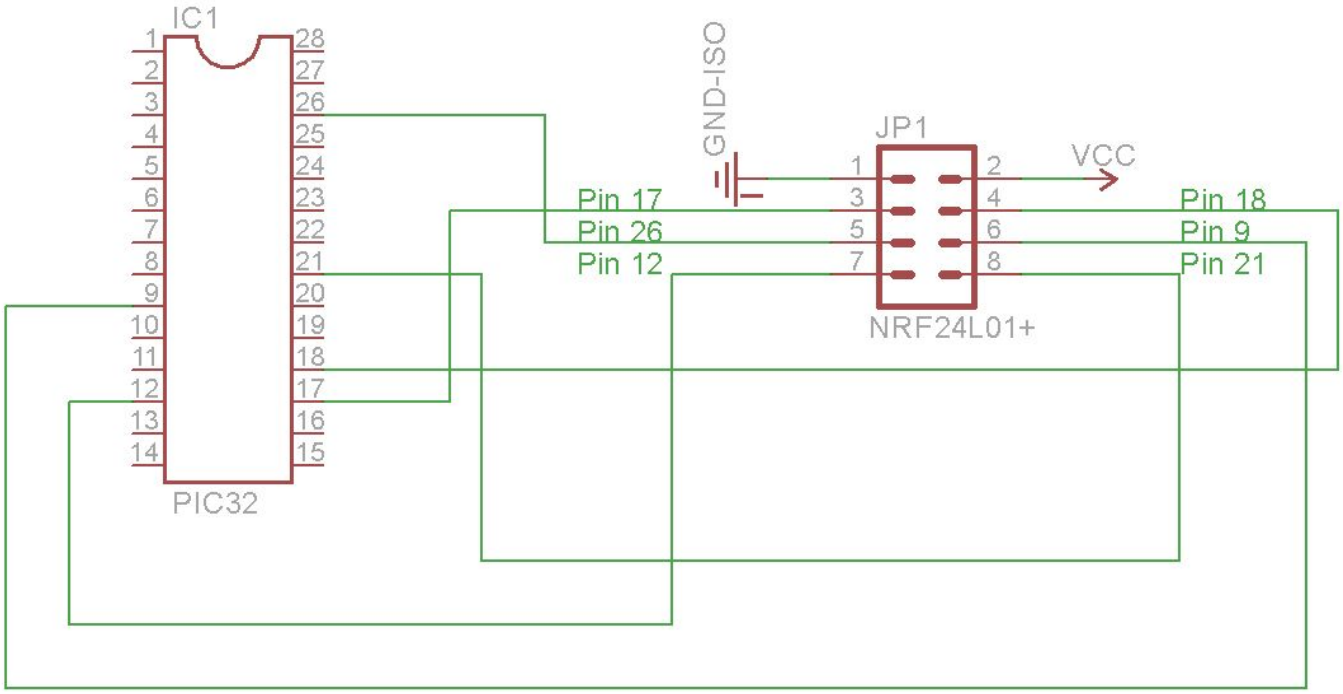


**Figure 7:** Basic Radio Setup Schematic (PIC Connections Only)

## SPI Commands

The PIC32 communicates with the radio chip through SPI sending one byte at a time. Because of this characters or arrays of characters are used for communication functions because characters are also one byte in size. On the MCU communication starts by sending a command from the list in Table 5. The nrf24l01+ always sends the contents of the STATUS register on the MISO line while a command is being sent, regardless of the command. For a write operation, the MCU then sends up to 32 bytes of data. For a read operation, the nrf24l01+ sends the contents sends up to 32 bytes of data depending on the command. Figure 8 and Figure 9 from the nrf24l01+ datasheet demonstrates the structure of the SPI communication between the nrf24l01+ and the MCU.

| | |
|---|---|
| R_REGISTER | 000A AAAA |
| W_REGISTER | 001A AAAA |
| R_RX_PAYLOAD | 0110 0001 |
| W_TX_PAYLOAD | 1010 0000 |

| | |
|---|---|
| FLUSH_TX | 1110 0001 |
| FLUSH_RX | 1110 0010 |
| REUSE_TX_PL | 1110 0011 |
| R_RX_PL_WID | 0110 0000 |
| W_ACK_PAYLOAD | 1010 1PPP |
| W_TX_PAYLOAD_NO_ACK | 1011 0000 |
| NOP | 1111 1111 |

**Table 5:** SPI Commands

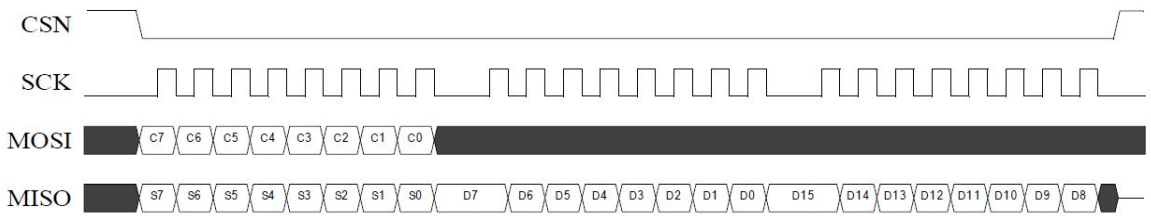


**Figure 8:** SPI Read Operation

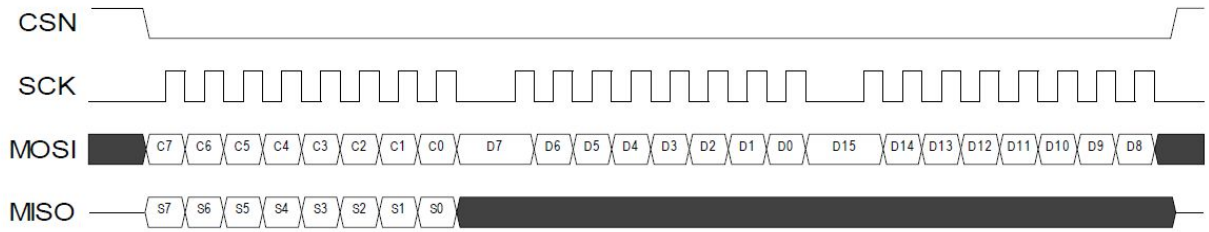


**Figure 9:** SPI Write Operation

The R_REGISTER command reads register AAAAA from the radio chip. The MCU will first send the command and then send the NOP command one time for each byte that needs to be read from the register. Register sizes range from one to five bytes. In the library this command is implemented in the nrf_read_reg function where the data is read into a buffer array of characters.

The W_REGISTER command writes data to register AAAAA on the radio chip. The MCU will first send the command and then send the data to be written. The data can be one to five bytes long. The radio chip will not send any data back during the write. This command is implemented in the library in in the nrf_write_reg function where data is written from an array of characters to the radio.

The R_RX_PAYLOAD function reads a received payload from a FIFO on the radio chip. After the MCU sends the command to read a payload it will send up to 32 NOP commands to the radio, one for each byte to be read. This means that the receiver must figure out how many bytes long the payload is before attempting to read it. After sending the status register while the command is sent the radio sends each byte of data to the MCU. This command is implemented in the nrf_read_payload function and works similarly to the read register function.

The W_TX_PAYLOAD function writes data to the TX FIFO on the radio that can be transmitted later. The MCU first sends the command to write and then sends data for up to 32 cycles, one for each byte of data to be written. This command is implemented in the nrf_write_payload function and works similarly to the write register function.

The FLUSH_TX and FLUSH_RX commands flush the TX and RX FIFOs respectively. No data needs to be written after this command so it only takes one cycle. These commands are implemented in the nrf_flush_tx and nrf_flush_rx functions.

The R_RX_PL_WID command gets the width of the first payload in the RX FIFO in bytes. The MCU first sends the command and then sends a NOP command while the width is being sent back. The radio chip sends the STATUS register while the command is being sent and then sends the width of the payload in bytes. This command is useful for checking the size of a payload when dynamic payload width is enabled. This command is implemented using the nrf_get_payload_width function.

The NOP command does nothing but is used to keep the SPI communication open when data is being sent back from the radio. If NOP is sent by itself and not after another command the radio will send the STATUS register back. This command is implemented using the normal rf_spiwrite function but giving it the argument, nrf24l01_SEND_CLOCK.

The remaining commands were not implemented because of time constraints and because they seemed less useful than the other commands.

## State Machine

The nrf24l01+ uses a state machine with five states. When the radio begins receiving power it will startup in the Power Down state. In this state registers can be written to and read and power consumption is reduced but the radio will take longer to begin sending or receiving packets compared to other states. This state can be transitioned to by clearing a PWR_UP bit in the CONFIG register and the nrf_pwrup function does transitions to this state from any other state. The CONFIG register map is shown in Table 6.

The next state the radio has is the Standby-I state where the radio consumes more power than the power down state but can start sending and receiving packets faster. In this state registers can also be read or written to. The radio will transition to Standby-I when the PWR_UP bit is set and the CE pin is driven low.

The next state the radio has is the RX Mode state where the radio will listen for packets being sent. If a packet is received in this mode it is put into an RX FIFO until it is read out. It will continually do this until it leaves the RX Mode state. To transition to the RX Mode state a PRIM_RX bit in the CONFIG register must be set and then CE must be driven high. The radio will remain in RX Mode until CE is driven low again or the PWR_UP bit is cleared.

The radio's next state is the TX Mode state where the radio will send packets out. Before entering the state data must be put in the TX FIFO. This data will be sent out as soon as the TX mode state is entered and the FIFO will be emptied as the data is sent out. Once data is in the TX FIFO the TX Mode state is entered by first clearing the PRIM_RX bit and then driving CE high for at least 10 us. The radio will leave TX Mode when CE is driven low again or when all the data in the FIFO has been transmitted. If CE goes low the radio will transition to the Standby-I state, if the FIFO is empty the radio will instead transition to the Standby-II state. The Standby-II uses more power than the Standby-II state but as soon as data is put in the TX FIFO in this state the radio will immediately transition to the TX Mode state and begin transmitting the data. The radio can transition out of the Standby-II state to the Standby-I state by driving CE low. The library code does not use the Standby-II state as it is not a recommended operating mode. A state diagram for the radio from the datasheet is shown in Figure 10.

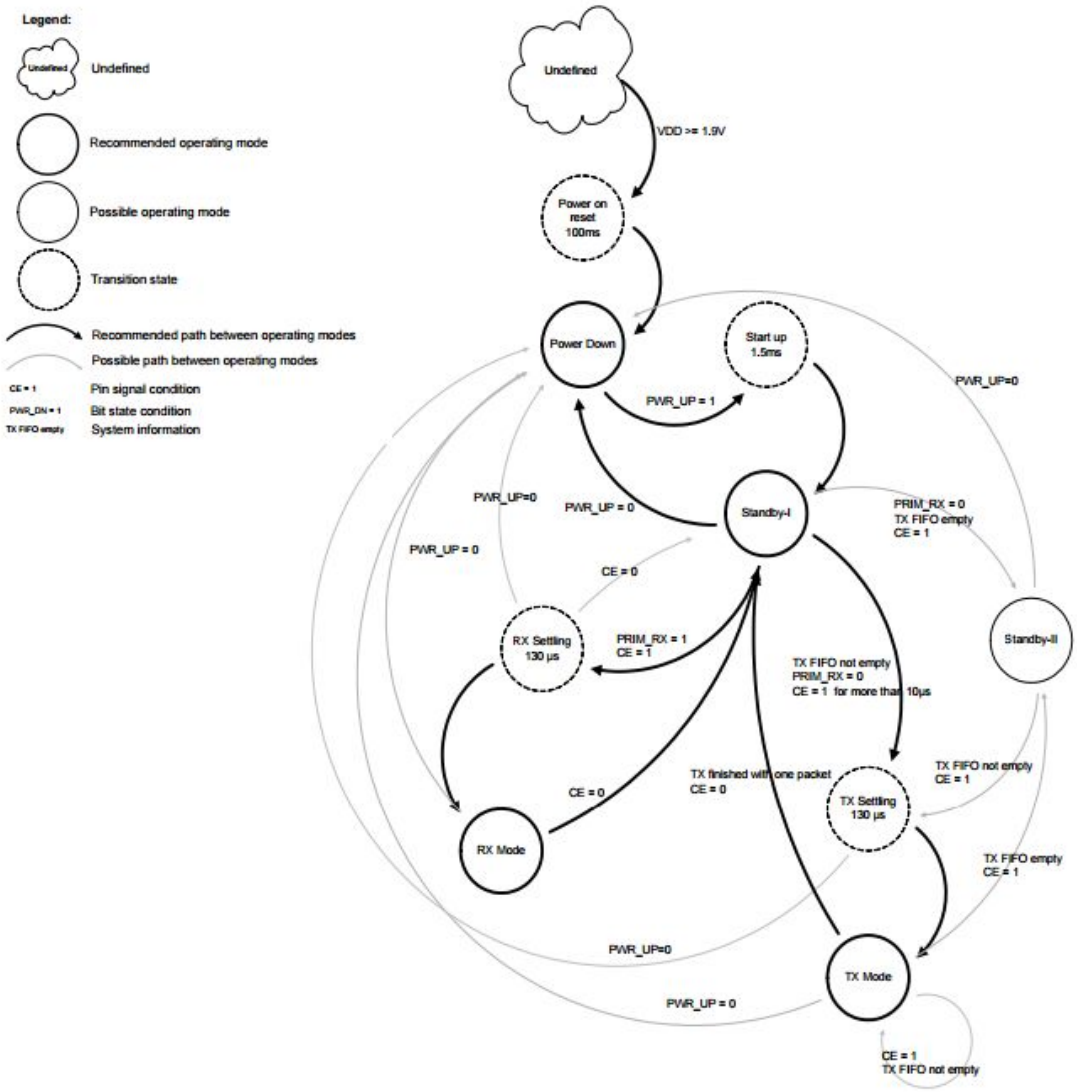


**Figure 10:** Radio State Diagram

| Name | Bits |
|---|---|
| Reserved | 7 |
| MASK_RX_DR | 6 |
| MASK_RX_DS | 5 |
| MASK_MAX_RT | 4 |
| EN_CRC | 3 |
| CRCO | 2 |

| | |
|---|---|
| PWR_UP | 1 |
| PRIM_RX | 0 |

**Table 6:** CONFIG Register Map

## Addresses and Pipes

For receiving, the nrf24l01+ contains six pipes that data can be received on numbered 0-5. Each of these pipes has a 3-5 byte address field that determines which packets are received on that pipe. Each pipe can have a different address, allowing them each to receive data from a different module. All pipes have the same address length, so if you set the address length to 3, all pipes will have a length of 3. The length of the address can be set to be 3-5 bytes using the nrf_set_address_width function. The address of a pipe can be set using the nrf_set_rx_addr. The first two pipes allow all five bytes of their addresses to be set while the other four pipes only allow the least significant byte of their address to be set. Pipes 2-5 use the four most significant bytes of pipe 1's address. So when using the nrf_set_rx_addr function on Pipes 2-5, only one byte should be entered as the address. When a payload is received the pipe it was received on is stored and can be read using the nrf_get_pipe function.

For transmission, each radio has a TX address as well. This sets the address which this radio transmits with. If the TX address of a radio sending a packet matches the address of a pipe on a receiving radio the packet will be received by the pipe with the matching address. So in order to send to a pipe on a receiver that has its address set to 0xAAAAAA, the TX address of the transmitter must be set to 0xAAAAAA as well. The address of the transmitter be set using the nrf_set_tx_addr function.

Receiving on certain pipes can be enabled or disabled by setting or clearing the ERX_Px bits in the EN_RXADDR register. For example, setting ENAA_P5 to 1 would enable pipe 5. This can be done using the nrf_en_rxaddr and nrf_dis_rxaddr functions. The EN_RXADDR register map is shown in Table 7.

| Name | Bits |
|---|---|
| Reserved | 7:6 |
| ENAA_P5 | 5 |
| ENAA_P4 | 4 |
| ENAA_P3 | 3 |

| | |
|---|---|
| ENAA_P2 | 2 |
| ENAA_P1 | 1 |
| ENAA_P0 | 0 |

**Table 7:** EN_RXADDR Register Map

When a payload is received the pipe it was received on is stored and can be read using the nrf_get_pipe function.

## Transmitting

The radio chip can send up to 32 bytes of data in a packet. To send a packet the data must be written into the TX FIFO which is done using the nrf_write_payload function. Once the data is in the FIFO it can be sent by setting PRIM_RX to 0 and driving CE high. If the radio is in the Standby-II state the radio will send the data right after it is written to the FIFO. If multiple packets are written to the TX FIFO the radio will send all of them out one after the other or will stop sending them if CE is driven low. When a packet is finished being sent the radio will trigger an interrupt. When auto-acknowledge mode is off this interrupt occurs as soon as the packet is sent regardless of if the packet was received or not. With auto-acknowledge on, the interrupt will trigger for several different cases which are covered in greater detail in the Auto-Acknowledge and Interrupts sections below.

## Transmit Settings

There are several minor settings that can be set when transmitting by setting certain bits in certain registers. A 1 or 2 byte CRC can be set in the config register by setting the EN_CRC bit and setting or clearing the CRCO bit in the CONFIG register. The frequency the radio will transmit at can be set by changing the RF_CH bits in the RF_CH registers using the nrf_set_rf_ch function. The frequency set will be 2400 + RF_CH (MHz). The data rate can be set by setting or clearing the RF_DR_LOW and RF_DR_HIGH bits in the RF_SETUP register using the nrf_set_transmit_rate function. The possible data rates are 250 kbps, 1 Mbps, or 2 Mbps. If the rate is set to 2 Mbps the radio will operate in a 2 Mhz bandwidth instead of a 1 Mhz bandwidth. The power the radio transmits at can also be set by setting the RF_PWR bits in the RF_SETUP register using the nrf_set_transmit_pwr function. The possible power settings are 0 dBm, -6dBm, -12dBm, or -18dBm. The RF_SETUP register map is shown in Table 8 and the RF_CH register map is shown in Table 9 for reference.

| Name | Bits |
|---|---|

| | |
|---|---|
| CONT_WAVE | 7 |
| Reserved | 6 |
| RF_DR_LOW | 5 |
| PLL_LOCK | 4 |
| RF_DR_HIGH | 3 |
| RF_PWR | 2:1 |
| Obsolete | 0 |

**Table 8:** RF_SETUP Register Map

| Name | Bits |
|---|---|
| Reserved | 7 |
| RF_CH | 6:0 |

**Table 9:** RF_CH Register Map

## Auto-Acknowledge

The nrf24l01+ has an auto-acknowledge features that allows a transmitter to determine if a packet was successfully received. When auto-acknowledge is enabled the transmitter will send the packet and then go into receive mode automatically. When the receiver gets the packet it will go into transmit mode and send an acknowledgement packet. If the transmitter receives the acknowledgement packet it will signal that the packet was received successfully, if it doesn't receive the packet it will signal the originally sent packet was not received. After this the transmitter goes back into transmit mode and the receiver goes back into receive mode automatically.

The radio will retransmit packets multiple times if a packet is not acknowledged. The amount of times the packet is retransmitted is set by setting the ARC bits in the SETUP_RETR register using the nrf_set_arc function. The radio can be set to retransmit 0-15 times. There is a delay between retransmits that can be set by setting the ARD bits in the SETUP_RETR register using the nrf_set_ard function. The delay is set to 250 + 250*ARD μs and is measured from when the time a packet is finished transmitting to the time the next packet begins being transmitted. The radio times out after transmitting the set amount of times and assumes the packet was not received. The SETUP_RETR register map is shown in Table 10.

| Name | Bits |
|---|---|
| ARD | 7:4 |
| ARC | 3:0 |

**Table 10:** SETUP_RETR Register Map

When a packet is sent in auto-acknowledge mode the radio will trigger an interrupt after the packet is acknowledged or after the radio times out. The same pin is used to signal both interrupts so in order to signal which interrupt occurred the radio sets a bit in the STATUS register. If the packet was acknowledged the TX_DS bit is set. If the radio times out the TX_RT bit is set. The nrf_send_payload function returns 1 if a packet was sent and acknowledged and 0 if the radio times out. The STATUS register map is shown in Table 11.

| Name | Bits |
|---|---|
| Reserved | 7 |
| RX_DR | 6 |
| TX_DS | 5 |
| MAX_RT | 4 |
| RX_P_NO | 3:1 |
| TX_FULL | 0 |

**Table 11:** STATUS Register Map

Auto-acknowledge mode is enabled on a per pipe basis. In order to enable auto-acknowledge on the receiver the EN_AA_Px bits must be set in the EN_AA register for whichever receiving pipes will have auto-acknowledge enabled. The transmitter must enable auto-acknowledge for pipe 0. In addition to this the transmitter must have its TX address and pipe 0 address set to be the same as the receiving pipe's address. For example if receiving on pipe 5 the addresses can be set up according to Table 12.

| | |
|---|---|
| Transmitter: TX address | 0xE7E7E7E7E7 |
| Transmitter: Pipe 0 address | 0xE7E7E7E7E7 |
| Receiver: Pipe 5 address | 0xE7E7E7E7E7 |

**Table 12:** Example Auto-Acknowledge Address Setup

This setup will allow a packet sent to pipe 5 on the receiver to be acknowledged. The EN_AA register map is shown in Table 13.

| Name | Bits |
|---|---|
| Reserved | 7:6 |
| ENAA_P5 | 5 |
| ENAA_P4 | 4 |
| ENAA_P3 | 3 |
| ENAA_P2 | 2 |
| ENAA_P1 | 1 |
| ENAA_P0 | 0 |

**Table 13:** EN_AA Register Map

Enabling or disabling auto-acknowledge for a pipe is done using the nrf_en_aa and nrf_dis_aa functions. The nrf_en_aa function will also enable receiving on a pipe automatically. Addresses for the necessary RX and TX pipes can be set using the nrf_set_tx_addr or nrf_set_rx_addr functions.

## Dynamic and Static Payload Lengths

By default static payload lengths are enabled on the radio. With static lengths the length of the payload must be set to be the same on both the receiver and the transmitter. The size of the payload is set on the transmitter by putting data in the TX FIFO. The size of the payload will be the amount of bytes of data put in the TX FIFO. On the receiving end the size of the payload is set on a per pipe basis. The payload size for a receiving pipe is set by setting the bits in the RX_PW_Px registers using the nrf_set_pw function. The payload widths can be set to be 1-32 bytes. If the payload width is not set to be the same for both the transmitter and receiver in static payload length mode packets will not be received correctly. To enable static payload widths the nrf_dis_dpl function is used which disables dynamic payload lengths and reenables static payload lengths.

Dynamic payload lengths allow packets to be sent and received without having to specify the size of the packet beforehand. In order to use dynamic payloads auto-acknowledge mode must also be enabled. Dynamic payload lengths is enabled on a per pipe basis. To enable dynamic payload both the transmitter and receiver must set the EN_DPL bit in the FEATURE register. The receiver must set the DPL_Px bit in the DYNPD register for whichever pipe it will receive dynamic payloads on. The transmitter must enable dynamic payloads on pipe 0 by setting the DPL_P0 bit in the DYNPD register. The function nrf_en_dpl will enable dynamic

payloads on a pipe and will also set the EN_DPL bit and enable auto-acknowledge on a pipe as necessary. Dynamic payloads can be disabled on a pipe using the nrf_dis_dpl function which will not disable auto_acknowledge for a pipe, but will clear the EN_DPL bit if dynamic payloads are disabled on all pipes. The DYNPD register map is shown in Table 14 and the FEATURE register map is shown in Table 15.

| Name | Bits |
|---|---|
| Reserved | 7:6 |
| DPL_P5 | 5 |
| DPL_P4 | 4 |
| DPL_P3 | 3 |
| DPL_P2 | 2 |
| DPL_P1 | 1 |
| DPL_P0 | 0 |

**Table 14:** DYNPD Register Map

| Name | Bits |
|---|---|
| Reserved | 7:3 |
| EN_DPL | 2 |
| EN_ACK_PAY | 1 |
| EN_DYN_ACK | 0 |

**Table 15:** FEATURE Register Map

When a dynamic payload is received its size in bytes is stored and can be read using the nrf_get_width function.

## Receiving

The radio can receive up to 32 bytes of data per packet on any of its six pipes. To receive the radio must be put in RX Mode by clearing PRIM_RX and driving CE low. Once in this mode the radio will wait to receive packets. When a packet is received the radio will trigger an interrupt to signal that a new packet is available to be read. To differentiate this interrupt

from the data sent and max retransmissions interrupts which are triggered when sending the radio will also set the RX_DR bit in the status register. The code reads this packet, the pipe the packet was received on, and the width of the packet (if dynamic payload length is enabled) The pipe number can be checked by calling the nrf_get_pipe function. The width can be found by calling the nrf_get_width function. The payload can be read by calling the nrf_get_payload function. The payload is stored as an array of characters and so it must be read out into another array.

## Interrupt

The radio has a single interrupt pin that signals three different types of interrupts. The type of interrupt can be determined by checking the status register to see which of the RX_DR (data received), TX_DS (data sent), and TX_MR (max retransmits) bits was set. The first type is a data sent interrupt which signals that data was sent and acknowledged if auto-acknowledge mode was enabled. The second is a max retransmit interrupt which occurs when the radio is in auto-acknowledge mode and does not receive an acknowledgement packet before retransmitting the maximum amount of times as set in the ARC bits of the SETUP_RETR register. The third interrupt is the data received interrupt which is triggered when a new packet is received by the radio and is available to be read.

The ISR works by first reading the status register to determine which kind of interrupt occurred. Depending on what type of interrupt occurred flags will be set to signal the rest of the code what has happened. The sent flag is set high during a data sent interrupt and is used in the nrf_send_payload functions to block until data has finished sending and to determine whether data has been acknowledged or not. The error flag is set during a max retransmits interrupt and signals that no acknowledgement packet was received. This is used by the nrf_send_payload function to block until data has been sent and to determine if the data was not received and acknowledged. The received flag is set during a data received interrupt to signal that a new packet was received. During a data received interrupt the data is also read out of the radio into a buffer called RX_payload. The received flag can be checked by calling nrf_payload_available and the data can be read out of RX_payload by calling nrf_get_payload.

The received flag can be polled to check when a new payload is available by calling nrf_payload_available and the flag will automatically be cleared when nrf_get_payload is called and the payload is read. If data is not read using nrf_get_payload before a new payload arrives the buffer will be overwritten and the payload that was inside will be lost. The width of the packet and the pipe the packet was received on will also be overwritten at the same time.

## Continuous Carrier Wave and Received Power Detector

The radio can be set up to transmit a continuous carrier wave which can be helpful for testing that the radio is set up correctly and for testing how far apart the radios can communicate from. To send a continuous carrier wave the CONT_WAVE and PLL_LOCK bits must be set in the RF_SETUP register. After this is done the radio must be put in TX mode by putting the radio in Standby-I mode, clearing PRIM_RX, and then driving CE high. The radio will continue to send the wave until CE is driven low again.

A continuous carrier wave can be sent by calling the nrf_start_cont_wave function which will set all necessary register values and drive CE high and will transmit a wave at a specified power level. To stop sending the wave nrf_stop_cont_wave should be called which will clear all the bits for sending a carrier wave and drive CE low. The carrier wave should be stopped before attempting to change any radio settings.

The radio contains a register called the RPD (received power detector) register that indicates the signal strength of a packet or carrier wave. If the received power is more than -64 dBm the RPD bit in the RPD register will be set high. This can be checked using the nrf_received_pwr function. Before calling this function the radio must be in RX Mode for at least 170 μs to get accurate measurements. This can be used with the continuous carrier wave to make testing that the radio is set up correctly easier.

# Appendix

## Hardware Pictures

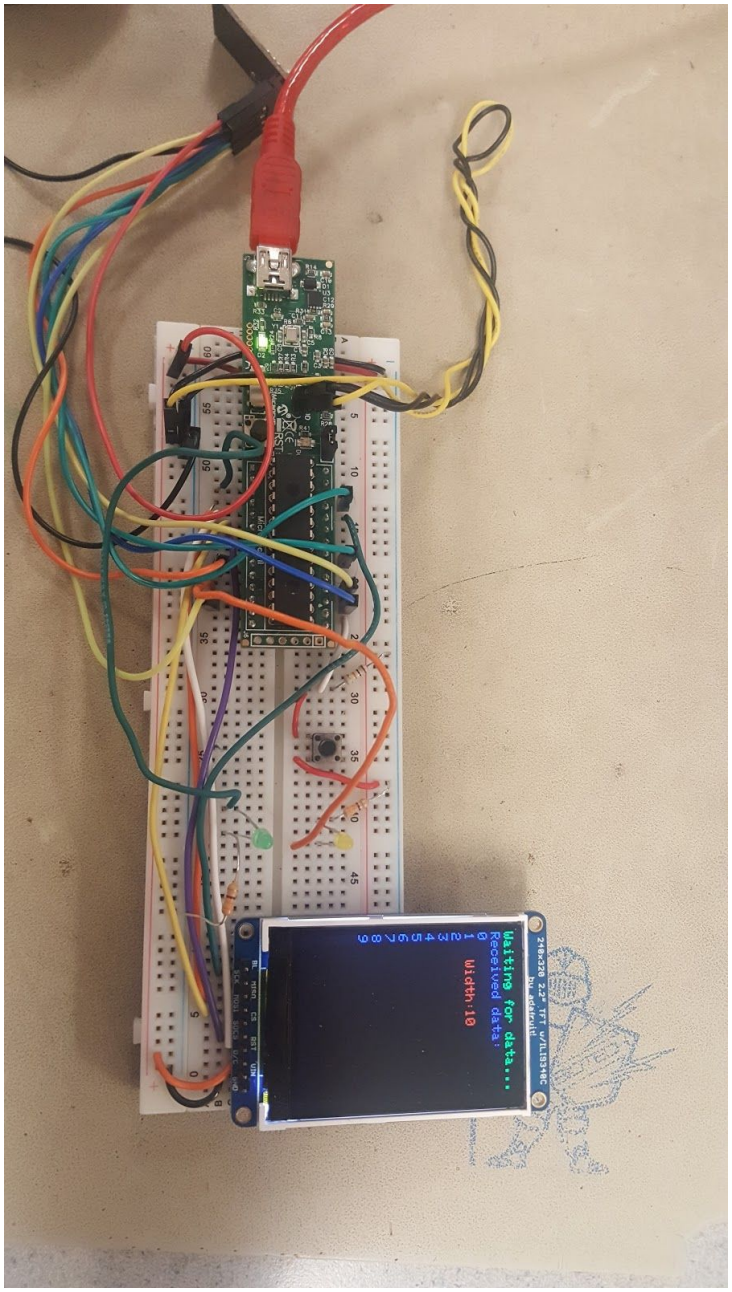


**Figure 11**: Microstick Setup

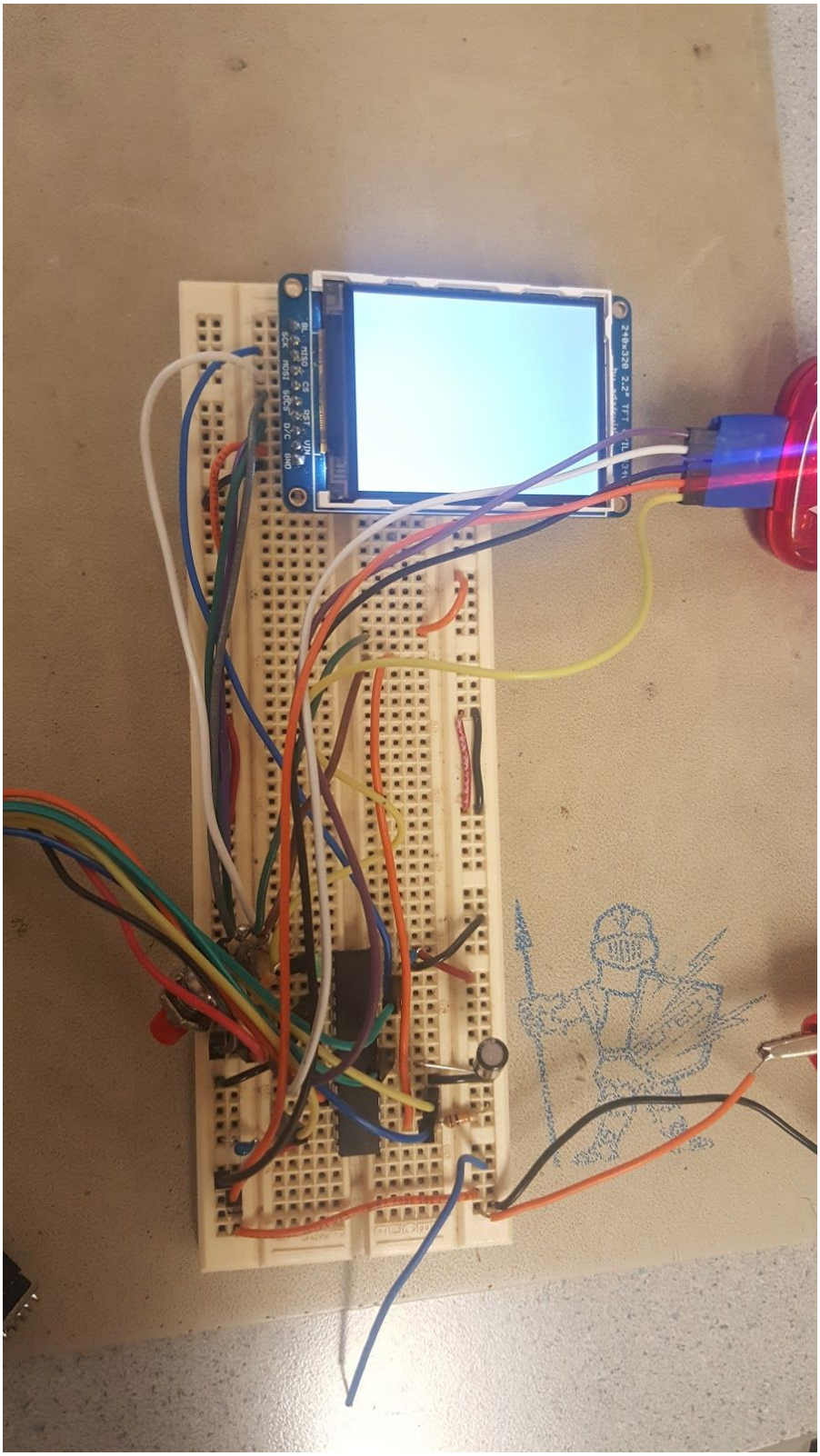


**Figure 12**: PICkit Setup


**Figure 13**: Custom PCB Setup

## Code

### *Nrf24l01.h*

```
/**
 * @file nrf24l01.h
 * @author Douglas Katz and Frederick Kummer
 * @date February 7 2016
 * @brief Library for use of the nrf24l01+ radio module with a PIC32
 *
 *
 */

 #define _SUPPRESS_PLIB_WARNING 1
 #include <plib.h>
 #include "inttypes.h"

#define _csn         LATBbits.LATB9
#define TRIS_csn     TRISBbits.TRISB9

#define _ce         LATBbits.LATB8
#define TRIS_ce     TRISBbits.TRISB8

// Credit to S. Brennen Ball for the register definitions
//
https://github.com/fffaraz/Introduction-to-Microprocessors/blob/master/material/atmel/
sample_code/nordic1/nrf24l01.h

// SPI Commands
#define nrf24l01_R_REGISTER            0x00
#define nrf24l01_W_REGISTER            0x20
#define nrf24l01_R_REGISTER_WID 0x61
#define nrf24l01_R_RX_PL_WID    0x60
#define nrf24l01_R_RX_PAYLOAD    0x61
#define nrf24l01_W_TX_PAYLOAD    0xA0
#define nrf24l01_FLUSH_TX        0xE1
#define nrf24l01_FLUSH_RX        0xE2
#define nrf24l01_REUSE_TX_PL     0xE3
#define nrf24l01_NOP                   0xFF


// Register definitions
#define nrf24l01_CONFIG                0x00
#define nrf24l01_EN_AA                 0x01
#define nrf24l01_EN_RXADDR       0x02
#define nrf24l01_SETUP_AW        0x03
#define nrf24l01_SETUP_RETR            0x04
#define nrf24l01_RF_CH                 0x05
#define nrf24l01_RF_SETUP        0x06
#define nrf24l01_STATUS                0x07
```

```
#define nrf24l01_OBSERVE_TX             0x08
#define nrf24l01_RPD            0x09
#define nrf24l01_RX_ADDR_P0             0x0A
#define nrf24l01_RX_ADDR_P1             0x0B
#define nrf24l01_RX_ADDR_P2             0x0C
#define nrf24l01_RX_ADDR_P3             0x0D
#define nrf24l01_RX_ADDR_P4             0x0E
#define nrf24l01_RX_ADDR_P5             0x0F
#define nrf24l01_TX_ADDR         0x10
#define nrf24l01_RX_PW_P0        0x11
#define nrf24l01_RX_PW_P1        0x12
#define nrf24l01_RX_PW_P2        0x13
#define nrf24l01_RX_PW_P3        0x14
#define nrf24l01_RX_PW_P4        0x15
#define nrf24l01_RX_PW_P5        0x16
#define nrf24l01_FIFO_STATUS     0x17
#define nrf24l01_DYNPD          0x1C
#define nrf24l01_FEATURE        0x1D

//CONFIG register bitwise definitions
#define nrf24l01_CONFIG_RESERVED 0x80
#define      nrf24l01_CONFIG_MASK_RX_DR 0x40
#define      nrf24l01_CONFIG_MASK_TX_DS 0x20
#define      nrf24l01_CONFIG_MASK_MAX_RT      0x10
#define      nrf24l01_CONFIG_EN_CRC           0x08
#define      nrf24l01_CONFIG_CRCO             0x04
#define      nrf24l01_CONFIG_PWR_UP           0x02
#define      nrf24l01_CONFIG_PRIM_RX          0x01

//EN_AA register bitwise definitions
#define nrf24l01_EN_AA_RESERVED         0xC0
#define nrf24l01_EN_AA_ENAA_ALL         0x3F
#define nrf24l01_EN_AA_ENAA_P5          0x20
#define nrf24l01_EN_AA_ENAA_P4          0x10
#define nrf24l01_EN_AA_ENAA_P3          0x08
#define nrf24l01_EN_AA_ENAA_P2          0x04
#define nrf24l01_EN_AA_ENAA_P1          0x02
#define nrf24l01_EN_AA_ENAA_P0          0x01
#define nrf24l01_EN_AA_ENAA_NONE 0x00

//EN_RXADDR register bitwise definitions
#define nrf24l01_EN_RXADDR_RESERVED     0xC0
#define nrf24l01_EN_RXADDR_ERX_ALL      0x3F
#define nrf24l01_EN_RXADDR_ERX_P5 0x20
#define nrf24l01_EN_RXADDR_ERX_P4 0x10
#define nrf24l01_EN_RXADDR_ERX_P3 0x08
#define nrf24l01_EN_RXADDR_ERX_P2 0x04
#define nrf24l01_EN_RXADDR_ERX_P1 0x02
#define nrf24l01_EN_RXADDR_ERX_P0 0x01
#define nrf24l01_EN_RXADDR_ERX_NONE     0x00
```

```
//SETUP_AW register bitwise definitions
#define nrf24l01_SETUP_AW_RESERVED      0xFC
#define nrf24l01_SETUP_AW               0x03
#define nrf24l01_SETUP_AW_5BYTES  0x03
#define nrf24l01_SETUP_AW_4BYTES  0x02
#define nrf24l01_SETUP_AW_3BYTES  0x01
#define nrf24l01_SETUP_AW_ILLEGAL 0x00

//SETUP_RETR register bitwise definitions
#define nrf24l01_SETUP_RETR_ARD                 0xF0
#define nrf24l01_SETUP_RETR_ARD_4000    0xF0
#define nrf24l01_SETUP_RETR_ARD_3750    0xE0
#define nrf24l01_SETUP_RETR_ARD_3500    0xD0
#define nrf24l01_SETUP_RETR_ARD_3250    0xC0
#define nrf24l01_SETUP_RETR_ARD_3000    0xB0
#define nrf24l01_SETUP_RETR_ARD_2750    0xA0
#define nrf24l01_SETUP_RETR_ARD_2500    0x90
#define nrf24l01_SETUP_RETR_ARD_2250    0x80
#define nrf24l01_SETUP_RETR_ARD_2000    0x70
#define nrf24l01_SETUP_RETR_ARD_1750    0x60
#define nrf24l01_SETUP_RETR_ARD_1500    0x50
#define nrf24l01_SETUP_RETR_ARD_1250    0x40
#define nrf24l01_SETUP_RETR_ARD_1000    0x30
#define nrf24l01_SETUP_RETR_ARD_750             0x20
#define nrf24l01_SETUP_RETR_ARD_500             0x10
#define nrf24l01_SETUP_RETR_ARD_250             0x00
#define nrf24l01_SETUP_RETR_ARC                 0x0F
#define nrf24l01_SETUP_RETR_ARC_15              0x0F
#define nrf24l01_SETUP_RETR_ARC_14              0x0E
#define nrf24l01_SETUP_RETR_ARC_13              0x0D
#define nrf24l01_SETUP_RETR_ARC_12              0x0C
#define nrf24l01_SETUP_RETR_ARC_11              0x0B
#define nrf24l01_SETUP_RETR_ARC_10              0x0A
#define nrf24l01_SETUP_RETR_ARC_9       0x09
#define nrf24l01_SETUP_RETR_ARC_8       0x08
#define nrf24l01_SETUP_RETR_ARC_7       0x07
#define nrf24l01_SETUP_RETR_ARC_6       0x06
#define nrf24l01_SETUP_RETR_ARC_5       0x05
#define nrf24l01_SETUP_RETR_ARC_4       0x04
#define nrf24l01_SETUP_RETR_ARC_3       0x03
#define nrf24l01_SETUP_RETR_ARC_2       0x02
#define nrf24l01_SETUP_RETR_ARC_1       0x01
#define nrf24l01_SETUP_RETR_ARC_0       0x00

//RF_CH register bitwise definitions
#define nrf24l01_RF_CH_RESERVED  0x80

//RF_SETUP register bitwise definitions
#define nrf24l01_RD_SETUP_CONT_WAVE 0x80
```

```
#define nrf24l01_RF_SETUP_RESERVED      0xE0
#define nrf24l01_RF_SETUP_PLL_LOCK      0x10
#define nrf24l01_RF_SETUP_RF_DR         0x08
#define nrf24l01_RF_SETUP_RF_PWR 0x06
#define nrf24l01_RF_SETUP_RF_PWR_0      0x06
#define nrf24l01_RF_SETUP_RF_PWR_6      0x04
#define nrf24l01_RF_SETUP_RF_PWR_12     0x02
#define nrf24l01_RF_SETUP_RF_PWR_18     0x00
#define nrf24l01_RF_SETUP_LNA_HCURR     0x01

//STATUS register bitwise definitions
#define nrf24l01_STATUS_RESERVED                            0x80
#define nrf24l01_STATUS_RX_DR                                     0x40
#define nrf24l01_STATUS_TX_DS                                     0x20
#define nrf24l01_STATUS_MAX_RT                                    0x10
#define nrf24l01_STATUS_RX_P_NO                                   0x0E
#define nrf24l01_STATUS_RX_P_NO_RX_FIFO_NOT_EMPTY    0x0E
#define nrf24l01_STATUS_RX_P_NO_UNUSED                      0x0C
#define nrf24l01_STATUS_RX_P_NO_5                           0x0A
#define nrf24l01_STATUS_RX_P_NO_4                           0x08
#define nrf24l01_STATUS_RX_P_NO_3                           0x06
#define nrf24l01_STATUS_RX_P_NO_2                           0x04
#define nrf24l01_STATUS_RX_P_NO_1                           0x02
#define nrf24l01_STATUS_RX_P_NO_0                           0x00
#define nrf24l01_STATUS_TX_FULL                                   0x01

//OBSERVE_TX register bitwise definitions
#define nrf24l01_OBSERVE_TX_PLOS_CNT    0xF0
#define nrf24l01_OBSERVE_TX_ARC_CNT           0x0F

//CD register bitwise definitions
#define nrf24l01_CD_RESERVED     0xFE
#define nrf24l01_CD_CD                  0x01

//RX_PW_P0 register bitwise definitions
#define nrf24l01_RX_PW_P0_RESERVED      0xC0

//RX_PW_P0 register bitwise definitions
#define nrf24l01_RX_PW_P0_RESERVED      0xC0

//RX_PW_P1 register bitwise definitions
#define nrf24l01_RX_PW_P1_RESERVED      0xC0

//RX_PW_P2 register bitwise definitions
#define nrf24l01_RX_PW_P2_RESERVED      0xC0

//RX_PW_P3 register bitwise definitions
#define nrf24l01_RX_PW_P3_RESERVED      0xC0

//RX_PW_P4 register bitwise definitions
```

```
#define nrf24l01_RX_PW_P4_RESERVED      0xC0

//RX_PW_P5 register bitwise definitions
#define nrf24l01_RX_PW_P5_RESERVED      0xC0

//FIFO_STATUS register bitwise definitions
#define nrf24l01_FIFO_STATUS_RESERVED   0x8C
#define nrf24l01_FIFO_STATUS_TX_REUSE   0x40
#define nrf24l01_FIFO_STATUS_TX_FULL    0x20
#define nrf24l01_FIFO_STATUS_TX_EMPTY   0x10
#define nrf24l01_FIFO_STATUS_RX_FULL    0x02
#define nrf24l01_FIFO_STATUS_RX_EMPTY   0x01

// RF power definitions
#define nrf24l01_DR_LOW 0x20
#define nrf24l01_DR_MED 0x00
#define nrf24l01_DR_HIGH 0x08

// Send no data for when you read data from the radio
#define nrf24l01_SEND_CLOCK             0x00

// State Definitions
#define PWR_DOWN 0
#define STANDBY_1 1
#define RX_MODE 2
#define TX_MODE 3

#define PBCLK 40000000 // peripheral bus clock

#define dTime_ms PBCLK/2000
#define dTime_us PBCLK/2000000


static char status;/**< The value of the status register, updated after each command
is sent*/
static volatile char RX_payload[32]; /**< Payloads will be stored here as they are
received*/
static char payload_size; /**< static payload size in bytes*/
static int pipe_no; /**< pipe most recent payload was received from*/
static int width; /**< width of most recent dynamic length payload in bytes*/

static int state; /**< The current state of the radio, NOTE: does not include TX
state*/

static volatile int received; /**< goes high when message is received*/
static volatile int sent; /**< goes high after radio finishes sending payload
correctly*/
static volatile int error; /**< goes high when no acknowledge is received*/
```

```
/**@defgroup UserFunc User Functions
 *
 * @brief Functions for simple radio use.
 *
 * This group contains functions intended for simple radio use. There are more
 * advanced functions available for developers and for more advanced radio
 * control in the nrf24l01.h.
 */

/**
 * \cond
 * @brief Transfer and receive a byte over SPI
 *
 * @param c The byte to transfer over SPI.
 * @return The byte received over SPI.
 */
char rf_spiwrite(unsigned char c);
/**\endcond*/

/**
 * @brief Set up SPI for the radio.
 */
void init_SPI();

/**
 * @ingroup UserFunc
 *
 * @brief Sets up the radio, SPI, and interrupts.  Also resets all radio
 * registers to their default values.
 */
void nrf_setup();

/**
 * @brief Read a register and store the data in an array. Can be multiple bytes
 * of data.
 *
 * @param reg The register to read from.  Use constants in nrf24l01.h
 * @param buff Pointer to the array the data will be stored in. LSB first.
 * @param len How many bytes of data need to be read. (1-5 bytes)
 */
void nrf_read_reg(char reg, char * buff, int len);

/**
 * @brief Write to a register from an array. Can be multiple bytes of data.
 *
 * @param reg The register to read from.  Use constants in nrf24l01.h
 * @param buff Pointer to the array data will be read from. LSB first.
 * @param len How many bytes of data to be written. (1-5 bytes)
 */
void nrf_write_reg(char reg, char * data, char len);
```

```
/**
 * @brief Flush the TX FIFO.
 */
void nrf_flush_tx();

/**
 * @brief Flush the RX FIFO.
 */
void nrf_flush_rx();

/**
 * @ingroup UserFunc
 *
 * @brief Get the width of the top payload in the RX FIFO.
 *
 * This function is only meant for use with dynamic payload lengths.
 *
 * @return The width of the payload in bytes.
 */
int nrf_get_payload_width();

/**
 * @brief Write a payload to the TX FIFO.
 *
 * @param data Pointer to data to be written.
 * @param len How many bytes of data will be written. (1-32 bytes)
 */
void nrf_write_payload(char * data, char len);

/**
 * @brief Read a payload from the RX FIFO.
 *
 * @param buff Pointer to array where data will be written.
 */
void nrf_read_payload(char * buff);

/** @addtogroup UserFunc */
/*@{*/
/**
 * @brief Read a received payload.
 *
 * Read a payload into an array if one is available.  If dynamic payload length
 * is enabled the length of the payload can be found using ::nrf_get_width. If a
 * payload is read, it is no longer available.  This function should only be
 * called if ::nrf_payload_available() returns 1.
 *
 * @param buff The array the payload will be read into.
 * @param len Length of the payload that will be read (1-32 bytes).
 *
```

```
 * @return 1 if a payload is read, 0 if no payload is available to be read.
 */
int nrf_get_payload(char * buff, char len);
/*@}*/

/**
 * @ingroup UserFunc
 *
 * @brief Check if a payload is available to be read.
 *
 * This function should be used to poll for received payloads.
 *
 * @return 1 if a payload was received and is available to be read, 0 if not.
 */
int nrf_payload_available();

/**
 * @ingroup UserFunc
 *
 * @brief Get the number of the pipe the most recent payload was received on.
 *
 * @return The number of the pipe the most recent payload was received on.
 */
int nrf_get_pipe();

/**
 * @ingroup UserFunc
 *
 * @brief Get the width of the most recently received payload.
 *
 * @return Width of the received payload in bytes.
 */
int nrf_get_width();


/**
 * @brief Sets the power up bit in the status register in order to leave the
 * power down state.
 */
void nrf_pwrup();

/**
 * @brief Clears the power up bit in the status register in order to enter the
 * power down state.
 */
void nrf_pwrdown();

/**
 * @ingroup UserFunc
 *
```

```
 * @brief Put the radio in the power down state.
 */
void nrf_state_pwr_down();

/**
 * @ingroup UserFunc
 *
 * @brief Put the radio in the standby 1 state.
 */
void nrf_state_standby_1();

/**
 * @ingroup UserFunc
 *
 * @brief Put the radio in the rx mode state.
 */
void nrf_state_rx_mode();

/**
 * @brief Set the PRIM_RX bit in the CONFIG register.
 */
void nrf_set_prim_rx();

/**
 * @brief Clear the PRIM_RX bit in the CONFIG register.
 */
void nrf_clear_prim_rx();

/**
 * @ingroup UserFunc
 *
 * @brief Set power of transmitter.
 *
 * The possible power settings are: \n
 *  0dBm: nrf24l01_RF_SETUP_RF_PWR_0 \n
 * -6dBm: nrf24l01_RF_SETUP_RF_PWR_6 \n
 * -12dBm: nrf24l01_RF_SETUP_RF_PWR_12 \n
 * -18dBm: nrf24l01_RF_SETUP_RF_PWR_18
 *
 * @param power Power level the transmitter will be set to.
 */
void nrf_set_transmit_pwr(char power);

/**
 * @ingroup UserFunc
 *
 * @brief Set data rate.
 *
 * The possible rate settings are: \n
 * 250 kbps: nrf24l01_DR_LOW \n
```

```
 * 1 Mbps: nrf24l01_DR_MED \n
 * 2 Mbps: nrf24l01_DR_HIGH
 *
 * @param rate Rate the transmitter will be set to.
 */
void nrf_set_transmit_rate(char rate);

/**
 * @brief Set the auto retransmit delay.
 *
 * Set how long the nrf24l01 should wait between retransmitting packets after
 * not receiving an acknowledgement packet. Delay is defined as the end of one
 * transmission to the start of the next. The delay is set according to the
 * equation, delay = 250 + ard * 250 (us).
 *
 * @param ard The length of auto retransmit delay to be set.
 */
void nrf_set_ard(char ard);

/**
 * @ingroup UserFunc
 *
 * @brief Set the auto retransmit count.
 *
 * Set how many times the nrf24l01 should attempt to retransmit the packet after
 * not receiving an acknowledgment packet.
 *
 * @param arc The amount of times the radio should attempt to retransmit.
 */
void nrf_set_arc(char arc);

/**
 * @ingroup UserFunc
 *
 * @brief Set the RF frequency the radio will operate at.
 *
 * The nrf24l01 can operate at frequencies ranging from 2.400GHz to 2.525 GHz.
 * At 250 kbps or 1 Mbps the radio occupies less than a 1MHz bandwidth. At
 * 2 Mbps the radio occupies less than a 2 MHz bandwidth. The frequency is set
 * according to the equation, frequency = 2400 + ch (MHz). The transmitter and
 * receiver must be set operate at the same frequency to communicate. This
 * function will set the center frequency of the radio.
 *
 * @param ch The frequency the radio should transmit at.
 */
void nrf_set_rf_ch(char ch);

/**
 * @ingroup UserFunc
 *
```

```
 * @brief Returns the pipe data is available in.
 *
 * If 8 is returned the FIFO is empty and no data was received.
 *
 * @return The pipe data is available in.
 */
char nrf_received_pipe_num();

/**
 * @ingroup UserFunc
 *
 * @brief Set the address width of RX and TX pipes
 *
 * Sets the address width of all receiving pipes and the transmitting address
 * width.
 *
 * @param width The address width to be set in bytes (3-5 bytes)
 */
void nrf_set_address_width(char width);

/**
 * @ingroup UserFunc
 *
 * @brief Send a constant carrier wave out at specified power.
 *
 * Send a continuous carrier wave for testing purposes. Wave can be stopped by
 * calling nrf_stop_cont_wave.
 *
 * @param pwr Power level to set transmitter to.  Possible values are:
 *  0dBm: nrf24l01_RF_SETUP_RF_PWR_0\n
 *  -6dBm: nrf24l01_RF_SETUP_RF_PWR_6\n
 *  -12dBm: nrf24l01_RF_SETUP_RF_PWR_12\n
 *  -18dBm: nrf24l01_RF_SETUP_RF_PWR_18
 */
void nrf_start_cont_wave(char pwr);

/**
 * @ingroup UserFunc
 *
 * @brief Stop sending the carrier wave.
 *
 * This function should only be called some time after ::nrf_start_cont_wave has
 * been called.
 */
void nrf_stop_cont_wave();

/**
 * @ingroup UserFunc
 *
 * @brief Check the power of the signal the nrf42l01 is receiving
```

```
 *
 * RX mode must be enabled for at least 40 us before measurements will be
 * accurate. 0 will be returned if power level is below -64dB and 1 will be
 * returned if power level is above -64dB.
 *
 * @return 1 if power level is above -64dB, 0 if the power level is not above
 * -64dB.
 */
char nrf_received_pwr();

/**
 * @ingroup UserFunc
 *
 * @brief Enable auto-acknowledge for a pipe.
 *
 * This function will enable receiving on a pipe automatically in addition to
 * enabling auto-acknowledge as if ::nrf_en_rxaddr was called.
 *
 * @param pipe The pipe autoack will be enabled on. Pipes range from 0 to 5.
 */
void nrf_en_aa(int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Disable auto-acknowledge for a pipe.
 *
 * @param pipe The pipe to autoack will be disabled on. Pipes range from 0 to 5.
 */
void nrf_dis_aa(int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Enable a pipe to receive packets.
 *
 * The nrf24l01+ contains six parallel pipes that can receive packets from six
 * different transmitters.  Each pipe must have a unique address.
 *
 * @param pipe The pipe that will be enabled. Pipes range from 0 to 5.
 */
void nrf_en_rxaddr(int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Disable a pipe from receiving packets.
 *
 * @param pipe The pipe that will be disabled. Pipes range from 0 to 5.
 */
```

```
void nrf_dis_rxaddr(int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Set the width received static payloads should be.
 *
 * When using static payload widths the width of packets must be explicitly set.
 * The receiver's width must be set to the size of the payloads being
 * transmitted. Use this function to set the width on the receiver. This
 * function is unnecessary for a pipe if dynamic payload length is enabled on
 * that pipe.
 *
 * @param width The number of bytes the pipe will be set to receive in a static
 * payload. (1-32 bytes)
 *
 * @param pipe The pipe whose payload width will be set.
 */
void nrf_set_pw(char width, int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Enable dynamic payload length for a pipe.
 *
 * If dynamic payload length is enabled the number of bytes in a packet does not
 * need to be specified.  For this feature to work the transmitter must
 * have dynamic payload length enabled for pipe 0 and the receiver must have
 * dynamic payload length enabled for the pipe it will receive from this
 * transmitter on.
 *
 * @param pipe The pipe dynamic payload length will be enabled on.
 */
void nrf_en_dpl(int pipe);

/**
 * @ingroup UserFunc
 *
 * @brief Disable dynamic payload length for a pipe.
 *
 * If dynamic payload length is disabled the length of a packet must be set
 * using ::nrf_set_pw.
 *
 * @param pipe Which pipe to disable dynamic payload length on.
 */
void nrf_dis_dpl(int pipe);

/**
 * @brief Enable dynamic auto-acknowledgements.
 *
```

```
 * Enables sending payloads without using auto-acknowlegment without disabling
 * the auto-acknowledge setting on the transmitter or receiver.  The SENDNOACK
 * function can then be used to do this.
 */
void nrf_en_dyn_ack();

/**
 * @brief Disable dynamic auto-acknowledgements.
 *
 * Disables sending payloads without using auto-acknowlegment without disabling
 * the auto-acknowledge setting on the transmitter or receiver.
 */
void nrf_dis_dyn_ack();

/**
 * @ingroup UserFunc
 *
 * @brief Set the address of a pipe.
 *
 * The nrf24l01 has six pipes to receive on.  Pipes have addresses
 * that are 3-5 bytes long.  Pipes 0 and 1 can have all 5 bytes of their
 * addresses changed.  The other four pipes can only have the LSB of their
 * address changed.  The four MSBs of the other four pipes are the MSBs of the
 * address of pipe 1.  The address width must be set using nrf_set_address_width
 * before calling this function.  If the address width set does not match the
 * length specified in this function 0 will be returned and the address will not
 * be written.  0 will also be returned if length is not set to 1 when writing
 * to pipes 2-5.
 *
 * @param pipe The pipe whose address will be set. Pipes range from 0-5.
 * @param address Address pipe will be set to.
 * @param len The length of the address being set in bytes. (1-5 bytes)
 * @return Returns 1 if address was written correctly.  Returns 0 if address was
 * written incorrectly.  If 0 is returned address was not written.
 */
int nrf_set_rx_addr(int pipe, uint64_t address, int len);

/**
 * @ingroup UserFunc
 *
 * @brief Set the address for transmitting.
 *
 * The address of the transmitter should match the address of the receiver or
 * packets will not be received.  If auto acknowledge is enabled rx pipe 0 on
 * the transmitter must be set to the same address as the tx address set in this
 * function.
 *
 * @param address The address that will be set for transmitting. (5 bytes)
 */
void nrf_set_tx_addr(uint64_t address);
```

```
//\cond
// Helper function for parsing addresses into a buffer
char * parse_addr(uint64_t address);


/**
 * @brief Enables acknowledge packets to carry payloads.
 */
void nrf_en_ack_pay();

/**
 * @brief Disables acknowledge packets to carry payloads.
 */
void nrf_dis_ack_pay();
//\endcond

/**
 * @ingroup UserFunc
 *
 * @brief Resets all registers to their default values as listed on the
 * datasheet.
 */
void nrf_reset();

//\cond
/**
 * @brief Send a payload without using auto-acknowledge.
 *
 * Auto-acknowledge mode should not be enabled when calling this function. This
 * function will not check if a payload is received successfully.
 *
 * @param data The payload to be sent
 * @param len The length of the payload in bytes.
 * @return 1 if payload was successfully sent 0 if not.
 */
int nrf_send_payload_nonblock(char * data, char len);
//\endcond

/**
 * @ingroup UserFunc
 *
 * @brief Send a payload over the radio.
 *
 * If auto-acknowledge is disabled this function will return 1 when the packet
 * is successfully sent and does not reflect the packet being acknowledged.
 * This function will never return 0 if auto-acknowledge is disabled. If auto-
 * acknowledge is enabled this function returns 1 when a packet is acknowledged
 * after being sent and 0 if the packet is not acknowledged.
 *
```

```
 * @param data The payload to be sent.
 * @param len The length of the payload in bytes.
 * @return 1 if payload was successfully sent and acknowledged 0 if payload was
 * not acknowledged.
 */
int nrf_send_payload(char * data, char len);

//\cond
//From tft_master.h, by Syed Tahmid Mahbub
void nrf_delay_ms(unsigned long);

//From tft_master.h, by Syed Tahmid Mahbub
void nrf_delay_us(unsigned long);
//\endcond
```

***nrf24l01.c***

```
// serial stuff
#include <stdio.h>
#include <stdlib.h>

#include "nrf24l01.h"

// frequency we're running at
#define      SYS_FREQ 64000000

//Taken from tft_master.h, by Syed Tahmid Mahbub
void nrf_delay_ms(unsigned long i){
/* Create a software delay about i ms long
 * Parameters:
 *      i:  equal to number of milliseconds for delay
 * Returns: Nothing
 * Note: Uses Core Timer. Core Timer is cleared at the initialiazion of
 *      this function. So, applications sensitive to the Core Timer are going
 *      to be affected
 */
    unsigned int j;
    j = dTime_ms * i;
    WriteCoreTimer(0);
    while (ReadCoreTimer() < j);
}

//Taken from tft_master.h, by Syed Tahmid Mahbub
void nrf_delay_us(unsigned long i){
/* Create a software delay about i us long
 * Parameters:
 *      i:  equal to number of microseconds for delay
 * Returns: Nothing
 * Note: Uses Core Timer. Core Timer is cleared at the initialiazion of
 *      this function. So, applications sensitive to the Core Timer are going
 *      to be affected
 */
    unsigned int j;
    j = dTime_us * i;
    WriteCoreTimer(0);
    while (ReadCoreTimer() < j);
}

char rf_spiwrite(unsigned char c){ // Transfer to SPI
    while (TxBufFullSPI2());
    WriteSPI2(c);
    while( !SPI2STATbits.SPIRBF); // check for complete transmit
    return ReadSPI2();
}
```

```
void init_SPI(){
    // Set up SPI2 to be active high, master, 8 bit mode, and ~4 Mhz CLK
    SpiChnOpen(2, SPI_OPEN_MSTEN | SPI_OPEN_MODE8 | SPI_OPEN_ON | SPI_OPEN_CKE_REV,
16);
    // Set SDI2 to pin 12
    PPSInput(3, SDI2, RPA4);
    // Set SDO2 to pin 9
    PPSOutput(3, RPA2, SDO2);
}

void nrf_setup(){
    nrf_delay_ms(200);
    init_SPI();
    // Set external interrupt 1 to pin 21
    PPSInput(4, INT1, RPB10);

    ConfigINT1(EXT_INT_PRI_2 | FALLING_EDGE_INT | EXT_INT_ENABLE);
    EnableINT1;
    mINT1ClearIntFlag();

    TRIS_csn = 0;
    TRIS_ce = 0;

    nrf_reset();
}

// Read a register from the nrf24l01
// reg is the array to read, len is the length of data expected to be received (1-5
bytes)
// NOTE: only address 0 and 1 registers use 5 bytes all others use 1 byte
// NOTE: writing or reading payload is done using a specific command
void nrf_read_reg(char reg, char * buff, int len){
    //char reg_read[5]; // register used for reading a single register
    int i = 0;
    _csn = 0; // begin transmission
    status = rf_spiwrite(nrf24l01_R_REGISTER | reg); // send command to read register

    for(i=0;i<len;i++){
        buff[i] = rf_spiwrite(nrf24l01_SEND_CLOCK); // send clock pulse to continue
receiving data
    }
    _csn = 1; // end transmission
}


void nrf_write_reg(char reg, char * data, char len){
    int i = 0;
    _csn = 0; // begin transmission
    status = rf_spiwrite(nrf24l01_W_REGISTER | reg); // send command to write reg
```

```
    for(i=0;i<len;i++){
        rf_spiwrite(data[i]); // write each char/byte to address reg
    }
    _csn = 1; // end transmission
}

// flushes the tx FIFO
void nrf_flush_tx(){
    _csn = 0;
    rf_spiwrite(nrf24l01_FLUSH_TX);
    _csn = 1;

}

// flushes the rx FIFO
// NOTE: do not use while sending acknowledgement
void nrf_flush_rx(){
    _csn = 0;
    rf_spiwrite(nrf24l01_FLUSH_RX);
    _csn = 1;
}


int nrf_empty_tx_fifo(){
    char reg;
    nrf_read_reg(nrf24l01_FIFO_STATUS, &reg, 1);
    char empty = reg & nrf24l01_FIFO_STATUS_TX_EMPTY;
    if(empty == nrf24l01_FIFO_STATUS_TX_EMPTY){
        return 1;
    }else{
        return 0;
    }
}

int nrf_full_tx_fifo(){
    char reg;
    nrf_read_reg(nrf24l01_FIFO_STATUS, &reg, 1);
    char full = reg & nrf24l01_FIFO_STATUS_TX_FULL;
    if(full == nrf24l01_FIFO_STATUS_TX_FULL){
        return 1;
    }else{
        return 0;
    }
}

// This is meant for library use
int nrf_get_payload_width(){
    char width;
    _csn = 0;
    status = rf_spiwrite(nrf24l01_R_RX_PL_WID);
```

```
    width = rf_spiwrite(nrf24l01_SEND_CLOCK);
    _csn = 1;
    return width;
}

// Write a payload to be sent over the radio
// data: array of chars to be sent (1-32 chars/bytes)
// len: amount of chars in array/bytes to be sent
void nrf_write_payload(char * data, char len){
    int i = 0;
    //Write packet to TX FIFO before pulsing
    _csn = 0; // begin transmission
    status = rf_spiwrite(nrf24l01_W_TX_PAYLOAD); // send the command to write the
payload
    for(i=0;i<len;i++){
        rf_spiwrite(data[i]); // write each char/byte to tx payload one at a time
    }
    _csn = 1; // end transmission

}

// should read the payload into a buffer
void nrf_read_payload(char * buff){
    char dpl;
    // get the pipe the payload was received on
    nrf_read_reg(nrf24l01_STATUS, &status, 1);
    pipe_no = (status & 0x0E) >> 1;

    nrf_read_reg(nrf24l01_FEATURE, &dpl, 1);
    // check if dynamic payload lengths are enabled
    if(dpl & 0x04){
        width = nrf_get_payload_width();
        _csn = 0; // begin transmission
        status = rf_spiwrite(nrf24l01_R_RX_PAYLOAD); // send command to read payload
        int i;
        for(i=0;i<width;i++){
            buff[i] = rf_spiwrite(nrf24l01_SEND_CLOCK);
        }
        _csn = 1; // end transmission
    }else{
        _csn = 0; // begin transmission
        status = rf_spiwrite(nrf24l01_R_RX_PAYLOAD); // send command to read payload
        int i;
        for(i=0;i<payload_size;i++){
            buff[i] = rf_spiwrite(nrf24l01_SEND_CLOCK);
        }
        _csn = 1; // end transmission
    }

}
```

```
int nrf_get_payload(char * buff, char len){
    //A payload is available
    if(received){
        int i;
        for(i=0;i<len;i++){
            buff[i]=RX_payload[i];
        }
        received = 0;
        return 1;
    }else{//No payload is available
        return 0;
    }
}

int nrf_payload_available(){
    return received;
}

int nrf_get_pipe(){
    return pipe_no;
}

// This is meant for user use
int nrf_get_width(){
    return width;
}

//Sets the power up bit and waits for the startup time, putting the radio in Standby-I
mode
void nrf_pwrup(){
    char config;
    nrf_read_reg(nrf24l01_CONFIG, &config, 1);
    config |= nrf24l01_CONFIG_PWR_UP;
    nrf_write_reg(nrf24l01_CONFIG, &config, 1);
    nrf_delay_ms(2);//Delay for power up time
}

//Clear the pwr_up bit, transitioning to power down mode
void nrf_pwrdown(){
    char config;
    nrf_read_reg(nrf24l01_CONFIG, &config, 1);
    config &= ~(nrf24l01_CONFIG_PWR_UP);
    nrf_write_reg(nrf24l01_CONFIG, &config, 1);
    _ce = 0;
}

void nrf_state_pwr_down(){
    switch(state){
        case PWR_DOWN :
```

```
            break;
        case STANDBY_1 :
            nrf_pwrdown();
            break;
        case RX_MODE :
            _ce = 0;
            nrf_pwrdown();
            break;
        default :
            _ce = 0;
            nrf_pwrdown();
    }
    state = PWR_DOWN;
}

void nrf_state_standby_1(){
    switch(state){
        case PWR_DOWN :
            nrf_pwrup();
            break;
        case STANDBY_1 :
            break;
        case RX_MODE :
            _ce = 0;
            break;
        default :
            _ce = 0;
            nrf_pwrup();
    }
    state = STANDBY_1;
}

void nrf_state_rx_mode(){
    switch(state){
        case PWR_DOWN :
            nrf_state_standby_1();
            nrf_set_prim_rx();
            _ce = 1;
            nrf_delay_us(130);
            break;
        case STANDBY_1 :
            nrf_set_prim_rx();
            _ce = 1;
            nrf_delay_us(130);
            break;
        case RX_MODE :
            break;
        default :
            nrf_state_standby_1();
            nrf_set_prim_rx();
```

```
            _ce = 1;
            nrf_delay_us(130);
    }
    state = RX_MODE;
}

void nrf_set_prim_rx(){
    char config;
    nrf_read_reg(nrf24l01_CONFIG, &config, 1);
    config |= nrf24l01_CONFIG_PRIM_RX;
    nrf_write_reg(nrf24l01_CONFIG, &config, 1);
}

void nrf_clear_prim_rx(){
    char config;
    nrf_read_reg(nrf24l01_CONFIG, &config, 1);
    config &= ~(nrf24l01_CONFIG_PRIM_RX);
    nrf_write_reg(nrf24l01_CONFIG, &config, 1);
}

// sets power of transmitter, possible values and definitions for them are
//  0dBm: nrf24l01_RF_SETUP_RF_PWR_0
// -6dBm: nrf24l01_RF_SETUP_RF_PWR_6
// -12dBm: nrf24l01_RF_SETUP_RF_PWR_12
// -18dBm: nrf24l01_RF_SETUP_RF_PWR_18
void nrf_set_transmit_pwr(char power){
    char setup;
    nrf_read_reg(nrf24l01_RF_SETUP, &setup, 1); // check value of setup register
    setup &= ~(nrf24l01_RF_SETUP_RF_PWR); // clear power bits in register
    setup |= power; // set power bits in register
    nrf_write_reg(nrf24l01_RF_SETUP, &setup, 1);
}

// sets the rf data rate, possible values and definitoins for them are
// 250 kbps: nrf24l01_DR_LOW
// 1 Mbps: nrf24l01_DR_MED
// 2 Mbps: nrf24l01_DR_HIGH
void nrf_set_transmit_rate(char rate){
    char setup; // check value of setup register
    nrf_read_reg(nrf24l01_RF_SETUP, &setup, 1);
    setup &= 0xd7;  // clear data rate bits in register
    setup |= rate; // set data rate bits in register
    nrf_write_reg(nrf24l01_RF_SETUP, &setup, 1);
}

void nrf_set_ard(char ard){
    char setup; // check value of setup register
    nrf_read_reg(nrf24l01_SETUP_RETR, &setup, 1);
    setup &= 0x0F;  // clear data rate bits in register
    setup |= ard; // set data rate bits in register
```

```
    nrf_write_reg(nrf24l01_SETUP_RETR, &setup, 1);
}

void nrf_set_arc(char arc){
    char setup; // check value of setup register
    nrf_read_reg(nrf24l01_SETUP_RETR, &setup, 1);
    setup &= 0xF0;  // clear data rate bits in register
    setup |= arc; // set data rate bits in register
    nrf_write_reg(nrf24l01_SETUP_RETR, &setup, 1);
}

void nrf_set_rf_ch(char ch){
    nrf_write_reg(nrf24l01_RF_CH, &ch, 1);
}

char nrf_received_pipe_num(){
    nrf_read_reg(nrf24l01_STATUS, &status, 1);
    return status & 0x0E;
}

void nrf_set_address_width(char width){
    char setting = width - 2;
    nrf_write_reg(nrf24l01_SETUP_AW, &setting, 1);
}

char * parse_addr(uint64_t address){
    static char addr[5];
    int i = 0;
    for(i = 0; i < 5;i++){
        addr[i] = (char)((address >> i*8) & 0xFF);
    }
    return &addr;
}

int nrf_set_rx_addr(int pipe, uint64_t address, int len){
    char width;
    nrf_read_reg(nrf24l01_SETUP_AW, &width, 1);
    width = width + 2;
    char * addr = parse_addr(address);

    if(pipe == 0 || pipe == 1){
        if(len != width){//Must equal preset address width
            return 0;
        }

    }else{
        if(len != 1){
            return 0;//For pipes 2,3,4,5 can only set last byte
        }
    }
```

```
    nrf_write_reg(nrf24l01_RX_ADDR_P0+pipe, addr, len);
    return 1;
}

void nrf_set_tx_addr(uint64_t address){
    char * addr = parse_addr(address);
    nrf_write_reg(nrf24l01_TX_ADDR, addr, 5);
}

void nrf_start_cont_wave(char pwr){
    nrf_state_standby_1();
    char setting;
    nrf_read_reg(nrf24l01_RF_SETUP, &setting, 1);
    setting |= nrf24l01_RF_SETUP_PLL_LOCK | nrf24l01_RD_SETUP_CONT_WAVE;
    nrf_write_reg(nrf24l01_RF_SETUP, &setting, 1);
    nrf_set_transmit_pwr(pwr);
    _ce = 1;
}

void nrf_stop_cont_wave(){
    _ce = 0;
    char reg;
    nrf_read_reg(nrf24l01_RF_SETUP, &reg, 1);
    reg &= ~(nrf24l01_RF_SETUP_PLL_LOCK | nrf24l01_RD_SETUP_CONT_WAVE);
    nrf_write_reg(nrf24l01_RF_SETUP, &reg, 1);
    nrf_pwrdown();
}

char nrf_received_pwr(){
    char pwr;
    nrf_read_reg(nrf24l01_RPD, &pwr, 1);
    pwr &= 0x01;
    return pwr;
}

void nrf_en_aa(int pipe){
    char num = 0x01;
    num = num << pipe;
    char reg;
    nrf_read_reg(nrf24l01_EN_AA, &reg, 1);
    reg |= num;
    // enable the pipe
    nrf_en_rxaddr(pipe);
    nrf_write_reg(nrf24l01_EN_AA, &reg, 1);
}

void nrf_dis_aa(int pipe){
    char num = 0x01;
    num = num << pipe;
```

```
    char reg;
    nrf_read_reg(nrf24l01_EN_AA, &reg, 1);
    num = ~num;
    reg &= num;
    nrf_write_reg(nrf24l01_EN_AA, &reg, 1);
}

void nrf_en_rxaddr(int pipe){
    char num = 0x01;
    num = num << pipe;
    char reg;
    nrf_read_reg(nrf24l01_EN_RXADDR, &reg, 1);
    reg |= num;
    nrf_write_reg(nrf24l01_EN_RXADDR, &reg, 1);
}

void nrf_dis_rxaddr(int pipe){
    char num = 0x01;
    num = num << pipe;
    char reg;
    nrf_read_reg(nrf24l01_EN_RXADDR, &reg, 1);
    num = ~num;
    reg &= num;
    nrf_write_reg(nrf24l01_EN_RXADDR, &reg, 1);
}

void nrf_set_pw(char width, int pipe){
   payload_size = width;
   nrf_write_reg(nrf24l01_RX_PW_P0 + pipe, &width, 1);
}

void nrf_en_dpl(int pipe){
    char num = 0x01;
    num = num << pipe;
    char reg;
    nrf_read_reg(nrf24l01_DYNPD, &reg, 1);
    // set bit corresponding to pipe
    reg |= num;
    nrf_write_reg(nrf24l01_DYNPD, &reg, 1);
    // set EN_DPL in FEATURE register
    nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
    reg |= 0x04;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
    // enable autoack for the pipe
    nrf_en_aa(pipe);
}

void nrf_dis_dpl(int pipe){
    char num = 0x01;
    num = num << pipe;
```

```
    char reg;
    nrf_read_reg(nrf24l01_DYNPD, &reg, 1);
    // clear bit corresponding to pipe
    num = ~num;
    reg &= num;
    nrf_write_reg(nrf24l01_DYNPD, &reg, 1);
    if(reg == 0x00){ // check if dpl has been disabled in all pipes
        // clear EN_DPL in FEATURE register
        nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
        reg &= ~0x04;
        nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
    }
}

void nrf_en_dyn_ack(){
    char reg;
    nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
    reg |= 0x01;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
}

void nrf_dis_dyn_ack(){
    char reg;
    nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
    reg &= ~0x01;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
}

void nrf_en_ack_pay(){
    char reg;
    nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
    reg |= 0x02;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
}

void nrf_dis_ack_pay(){
    char reg;
    nrf_read_reg(nrf24l01_FEATURE, &reg, 1);
    reg &= ~0x02;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
}

//Send a payload, with no checking for whether it was successfully received or not
//Does not provide reliable data transfer, but makes it possible to push out more
packets
//and to have greater application level control over packet timing.
int nrf_send_payload_nonblock(char * data, char len){
    nrf_flush_tx();
    nrf_write_payload(data, len);//Send payload to FIFO
    nrf_state_standby_1();//Transition to standby 1 state to operate in a known state
```

```
    nrf_clear_prim_rx();

    _ce = 1;//Pulse the line to begin the transition to TX
    nrf_delay_us(100);
    _ce = 0;

    nrf_delay_us(130);//RX Settling Time

    if(sent) return 1;
    else return 0;
}

// send a payload with auto ack. Returns 1 if packet was received correctly
int nrf_send_payload(char * data, char len){
    nrf_flush_tx();
    nrf_write_payload(data,len);
    nrf_state_standby_1();
    nrf_clear_prim_rx();
    _ce = 1;
    while(!sent && !error);
    _ce = 0;
    if(sent){
        sent = 0;
        return 1;
    }
    error = 0;
    return 0;
}


void nrf_reset(){
    char reg = 0x08;
    nrf_write_reg(nrf24l01_CONFIG, &reg, 1);
    reg = 0x3F;
    nrf_write_reg(nrf24l01_EN_AA, &reg, 1);
    reg = 0x03;
    nrf_write_reg(nrf24l01_SETUP_AW, &reg, 1);
    reg = 0x03;
    nrf_write_reg(nrf24l01_EN_RXADDR, &reg, 1);
    reg = 0x03;
    nrf_write_reg(nrf24l01_SETUP_RETR, &reg, 1);
    reg = 0x02;
    nrf_write_reg(nrf24l01_RF_CH, &reg, 1);
    reg = 0x0E;
    nrf_write_reg(nrf24l01_RF_SETUP, &reg, 1);
    reg = 0x70;
    nrf_write_reg(nrf24l01_STATUS, &reg, 1);
    reg = 0x00;
    nrf_write_reg(nrf24l01_OBSERVE_TX, &reg, 1);
    nrf_set_rx_addr(0, 0xE7E7E7E7E7, 5);
```

```
    nrf_set_rx_addr(1, 0xC2C2C2C2C2, 5);
    nrf_set_rx_addr(2, 0xC3, 1);
    nrf_set_rx_addr(3, 0xC4, 1);
    nrf_set_rx_addr(4, 0xC5, 1);
    nrf_set_rx_addr(5, 0xC6, 1);
    nrf_set_tx_addr(0xE7E7E7E7E7);
    nrf_set_pw(0, 0);
    nrf_set_pw(0, 1);
    nrf_set_pw(0, 2);
    nrf_set_pw(0, 3);
    nrf_set_pw(0, 4);
    nrf_set_pw(0, 5);
    reg = 0x00;
    nrf_write_reg(nrf24l01_DYNPD, &reg, 1);
    reg = 0x00;
    nrf_write_reg(nrf24l01_FEATURE, &reg, 1);
    nrf_flush_tx();
    nrf_flush_rx();
    state = PWR_DOWN;
}

void __ISR(_EXTERNAL_1_VECTOR, ipl2) INT1Handler(void){
    nrf_read_reg(nrf24l01_STATUS, &status, 1); // read the status register
    // check which type of interrupt occurred
    if (status & nrf24l01_STATUS_RX_DR) { // if data received
        nrf_read_payload(&RX_payload);
        received = 1; // signal main code that payload was received
        status |= nrf24l01_STATUS_RX_DR; // clear interrupt on radio
    }
        // if data sent or if acknowledge received when auto ack enabled
    else if (status & nrf24l01_STATUS_TX_DS) {
        sent = 1; // signal main code that payload was sent
        status |= nrf24l01_STATUS_TX_DS; // clear interrupt on radio
    } else { // maximum number of retransmit attempts occurred
        error = 1; // signal main code that the payload was not received
        status |= nrf24l01_STATUS_MAX_RT; // clear interrupt on radio
    }

    nrf_write_reg(nrf24l01_STATUS, &status, 1);
    mINT1ClearIntFlag();
}
```

### Counter.c

```
#include "config.h"
#include "tft_gfx.h"
#include "tft_master.h"
#include <stdio.h>
#define      SYS_FREQ 64000000 // change the frequency of the clock
#include "nrf24l01.h"
// These two defines are for the tft
#define spi_channel 1
#define spi_divider 10

/*
 * This code implements a synchronous counter using two PIC32s both connected to
 * tft displays and nrf24l01+ radios.  The ID field for the two PICs must be set
 * to 0 on the first PIC and 1 on the second.  Radio 0 will begin the program by
 * sending the initial value of counter which will be zero.  Once it gets an
 * acknowledgement from radio 1 it will go into receive mode and the counting
 * will start.  Radio 0 will increment the counter and send it to radio 1 which
 * will display the count and send it back to radio 0.  Radio 0 will receive the
 * count back and will then display it, increment it, and send the updated count
 * to radio 1.  Radio 1 will repeat what it did before and the loop will then
 * continue indefinitely.
 *
 * The counter will increment as fast as the radios can send and the screen can
 * update.
 */

// set up the radio
void radioSetup() {
    nrf_setup(); // initializing function

    nrf_set_arc(0x0A); // 10 retransmits

    nrf_set_ard(0x00); // 250 us between retransmission

    nrf_set_rf_ch(0x01); // freq = 2.401 GHz

    nrf_en_aa(0); // enable autoack on pipe 0

    nrf_set_pw(1, 0); //set the payload width to be 1 byte

    // set up addresses for autoack mode
    // The tx address and the pipe 0 rx address must be the same for autoack mode
    nrf_set_address_width(5);
    nrf_set_rx_addr(0, 0xAABBCCDDFF, 5);
    nrf_set_tx_addr(0xAABBCCDDFF);
}

void main(void) {
```

```
    INTEnableSystemMultiVectoredInt();
    radioSetup(); // setup the radio for this program
    tft_init_hw(); // setup for tft
    tft_begin(); // setup for tft
    tft_fillScreen(ILI9340_BLACK);
    //240x320 vertical display
    tft_setRotation(0); // Use tft_setRotation(1) for 320x240
    char buffer[120]; // a buffer for writing to the tft
    while (1) {
        //NOTE: Set one radio's ID to 0 and the other radio's ID to 1
        int ID = 0;
        volatile char counter = 0; // synchronous counter
        if (!ID) { // the first radio will keep transmitting until a packet is
received
            // while the packet isn't acknowledged/received continously resend
            while (!nrf_send_payload(&counter, 1)) {
                // blink a circle on screen while waiting
                tft_fillCircle(20, 20, 10, ILI9340_BLACK);
                nrf_delay_ms(300);
                tft_fillCircle(20, 20, 10, ILI9340_RED);
                nrf_delay_ms(300);
            }
        }
        // set radio to receive mode to start
        nrf_state_rx_mode();
        // clear the screen
        tft_fillScreen(ILI9340_BLACK);
        while (1) {
            // display "Current Count:" at top of screen
            tft_setTextColor(ILI9340_GREEN);
            tft_setTextSize(2);
            tft_setCursor(20, 0);
            sprintf(buffer, "%s", "Current Count:");
            tft_writeString(buffer);
            // wait until a payload is received before doing anything
            if (nrf_payload_available()) { // when a payload has been received
                tft_fillScreen(ILI9340_BLACK);
                // get new counter value from transmitter
                nrf_get_payload(&counter, 1);
                // display counter value
                tft_setCursor(20, 20);
                sprintf(buffer, "%d", counter);
                tft_writeString(buffer);
                if (!ID) { // increment counter on one radio
                    counter++;
                }
                // send incremented payload
                nrf_send_payload(&counter, 1); // send counter back to other radio
                nrf_state_rx_mode(); // put the radio in rx mode after the
transmission
```

```
            }
        }
    }
} // main

// === end  ======================================================
```

### *TX_carrier.c*

```
// graphics libraries
#include "config.h"
// serial stuff
#include <stdio.h>
// threading library
#define      SYS_FREQ 64000000 // change the frequency of the clock
// radio library
#include "nrf24l01.h"

/*
 * This code will send a continuous carrier wave that can be detected by another
 * radio.  It can be helpful for finding the maximum distance two radios can
 * receive from.  The power the radio transmits at can be changed if desired.
 */

// set up the radio
void radioSetup() {
    nrf_setup(); // initializing function
    nrf_set_rf_ch(0x01); // freq = 2.401 GHz
}

void main(void) {
    INTEnableSystemMultiVectoredInt();
    radioSetup(); // setup the radio for this program
    mINT1ClearIntFlag();
    /* Potential Power levels
     0dBm: nrf24l01_RF_SETUP_RF_PWR_0
    -6dBm: nrf24l01_RF_SETUP_RF_PWR_6
    -12dBm: nrf24l01_RF_SETUP_RF_PWR_12
    -18dBm: nrf24l01_RF_SETUP_RF_PWR_18
     */
    char pwr = nrf24l01_RF_SETUP_RF_PWR_0;
    nrf_start_cont_wave(pwr); // send the continuous wave
    while(1); // continuously send carrier wave.
} // main

// === end  ======================================================
```

### RX_carrier.c

```
#include "config.h"
#include <stdio.h>
#define      SYS_FREQ 64000000 // change the frequency of the clock
#include "nrf24l01.h"

// Set up LEDs for displaying power
// Green LED on pin 2
#define _LEDGREEN        LATAbits.LATA0
#define _TRIS_LEDGREEN   TRISAbits.TRISA0

// Yellow LED on pin 10
#define _LEDYELLOW        LATAbits.LATA3
#define _TRIS_LEDYELLOW   TRISAbits.TRISA3

// Red LED on pin 11
#define _LEDRED        LATBbits.LATB4
#define _TRIS_LEDRED   TRISBbits.TRISB4

/*
 * This code will send a continuous carrier wave that can be detected by another
 * radio.  It can be helpful for finding the maximum distance two radios can
 * receive from.  The power the radio transmits at can be changed if desired.
 */

// set up the radio
void radioSetup() {
    nrf_setup(); // initializing function

    nrf_set_rf_ch(0x01); // freq = 2.401 GHz
}

void main(void) {
    INTEnableSystemMultiVectoredInt();
    radioSetup(); // setup the radio for this program
    mINT1ClearIntFlag();
    nrf_state_rx_mode(); // put the radio into receive mode
    nrf_delay_us(40); // need to wait 40us after entering receive mode before checking
power
    // set up LEDs
    _TRIS_LEDGREEN = 0;
    _TRIS_LEDRED = 0;
    // continuously check power
    while(1){
        // if power is received
        if(nrf_received_pwr()){
            _LEDGREEN = 1; // turn on green LED when power is received
            _LEDRED = 0; // turn off red LED when power is received
        }else{
```

```
            _LEDGREEN = 0; // turn on green LED when power is not received
            _LEDRED = 1; // turn on red LED when power is not received
        }
    }
} // main

// === end  ======================================================
```

***dynPldTX.c***

```
#include "config.h"
#include <stdio.h>
#define      SYS_FREQ 64000000 // change the frequency of the clock
#include "nrf24l01.h"
#define spi_divider 10

/*
 * This code sends data of different sizes using the dynamic payload length
 * (dpl) feature of the radio. This file is the transmitter and will only send
 * data. The other file is the receiver and will receive the data of each array
 * and display it. This code is made to demonstrate sending multiple bytes at
 * once and sending dynamic length payloads.
 */

// set up the radio
void radioSetup() {
    nrf_setup(); // initializing function

    nrf_set_arc(0x05); // 5 retransmits

    nrf_set_ard(0x00); // 250 us between retransmission

    nrf_set_rf_ch(0x01); // freq = 2.401 GHz

    nrf_en_dpl(0); // enable dynamic payloads for transmitting

    // set up addresses for autoack mode and dynamic payload length (dpl) mode
    // The tx address and the pipe 0 rx address must be the same for autoack mode
    // and dpl mode
    nrf_set_address_width(3);
    nrf_set_rx_addr(0, 0xAABBCC, 3);
    nrf_set_tx_addr(0xAABBCC);
}

// Fill an array with with consecutive numbers
// This is so there is something to look at on the receiving end.
void fillArray(char * array, int len){
    int i;
    for(i=0;i<len;i++){
        array[i] = i;
    }
}

void main(void) {
    INTEnableSystemMultiVectoredInt();
    radioSetup(); // setup the radio for this program
    //240x320 vertical display
    tft_setRotation(0); // Use tft_setRotation(1) for 320x240
```

```
    /* array of data that will be sent.
     * the maximum data length that can be sent in one packet is 32 bytes. */
    char array[32];
    // Fill the array with consecutive incrementing numbers
    fillArray(&array,32);
    while (1) { // send varying amounts of data
        while(!nrf_send_payload(&array, 1)); // send one byte of data
        nrf_delay_ms(1000); // wait a second between each transmission
        while(!nrf_send_payload(&array, 5)); // send five bytes of data
        nrf_delay_ms(1000);
        while(!nrf_send_payload(&array, 10)); // send ten bytes of data
        nrf_delay_ms(1000);
        while(!nrf_send_payload(&array, 32)); // send 32 bytes of data
        nrf_delay_ms(1000);
    }
} // main

// === end  ======================================================
```

***dynPldRX.c***

```
#include "config.h"
#include "tft_gfx.h"
#include "tft_master.h"
#include <stdio.h>
#define      SYS_FREQ 64000000 // change the frequency of the clock
#include "nrf24l01.h"
// These two defines are for the tft
#define spi_channel 1
#define spi_divider 10

/* This code sends four arrays of different sizes using the
 * dynamic payload length (dpl) feature of the radio. This file is the
 * receiver and will receive arrays of varying size and display them. The other
 * file will create and send these arrays.
 */

// set up the radio
void radioSetup() {
    nrf_setup(); // initializing function

    nrf_set_rf_ch(0x01); // freq = 2.401 GHz

    nrf_en_dpl(1); // enable dpl for pipe 1

    // set up addresses for autoack mode and dynamic payload length (dpl) mode
    // The radio will receive on pipe 1
    nrf_set_address_width(3);
    nrf_set_rx_addr(1, 0xAABBCC, 3);
}
void main(void) {
    INTEnableSystemMultiVectoredInt();
    radioSetup(); // setup the radio for this program
    tft_init_hw(); // setup for tft
    tft_begin(); // setup for tft
    tft_fillScreen(ILI9340_BLACK);
    //240x320 vertical display
    tft_setRotation(0); // Use tft_setRotation(1) for 320x240
    char buffer[120]; // a buffer for writing to the tft
    // arrays to read payloads into
    char array[32];
    int width; // the width of the received payload
    while (1) {
        tft_setTextColor(ILI9340_GREEN);
        tft_setTextSize(2);
        tft_setCursor(0, 0);
        sprintf(buffer, "%s", "Waiting for data...");
        tft_writeString(buffer);
        // enter receive mode to search for packets
```

```
        nrf_state_rx_mode();
        if(nrf_payload_available()){ // wait for a payload to be received
            width = nrf_get_width(); // get the width of the payload
            nrf_get_payload(&array, width); // read the payload into the array
            tft_fillScreen(ILI9340_BLACK); // clear the display

            // display the received data on the tft
            tft_setTextColor(ILI9340_BLUE);
            tft_setTextSize(2);
            tft_setCursor(0, 20);
            sprintf(buffer, "%s", "Received data:");
            tft_writeString(buffer);
            tft_setCursor(40, 60);
            tft_setTextColor(ILI9340_RED);
            sprintf(buffer, "%s", "Width:");
            tft_writeString(buffer);
            tft_setCursor(110, 60);
            sprintf(buffer, "%d", width);
            tft_writeString(buffer);
            tft_setTextColor(ILI9340_BLUE);
            int i;
            for(i=0;i<width;i++){
                tft_setCursor(0, 40+20*i);
                sprintf(buffer, "%d", array[i]);
                tft_writeString(buffer);
            }
        }
    }
} // main

// === end  ======================================================
```