**The Ins and Outs**

Let me describe the tutorial to you first. Essentially, one microcontroller is connected to your PC via the serial port. Using “Hyper terminal”, you set up a connection that is 9600 baud, 8-N-1, with no flow control. You type a character, and this character is received by the microcontroller. Next, the microcontroller sends this character to the 24L01 which is attached to it via SPI, which then transmits the data over RF to another waiting 24L01. This other 24L01 is connected to another microcontroller, also via SPI.

Upon reception of a character, this remote microcontroller just sends the character right back to the first 24L01 chip. When the microcontroller receives this data, it transmits it back up the 232 link to your PC’s serial port and it is displayed on your screen. If a character isn’t received, it transmits a “?” to the computer.

**Pin Used**

In this example we use CE, CNS, SCK, MOSI, MISO, IRQ and Power Pin

Pin Connected Diagram:

CE 🡺 P1.4

CSN 🡺 P1.3

SCK 🡺 P1.5

MOSI 🡺 P1.7

MISO 🡺 P1.6

IRQ 🡺 P2.1 (active low – inactive high)

(these pin are defines in nrf24l01.h)

**NRF24L01.h and .c**

There are a few prerequisites that you need for this library to work. First you need a function that will delay a specified number of microseconds. Also, you need a function to operate the SPI bus. This function takes a byte as an argument, sends thatbyte over SPI, receives a byte from the 24L01 over SPI, and then returns this received byte. (function SPI\_Xfer\_byte)

Also, there is a section in the header file where you must make defines for the IO control registers and pin masks for each of the IO pins that go to the 24L01 (CSN, CE, and IRQ).

When you get the support code working and move onto your actual useful code, you will start off by initializing the nRF24L01. I have included three initialize functions, and they serve different purposes and are targeted for different users.

1. The first and most useful is nrf24l01\_initialize(). It allows you to initialize the values to every writeable register in the 24L01. It also allows you to set whether or not you set the CE pin when the 24L01 is set up as a receiver.
2. To get a simple connection up and running, use the nrf24l01\_initialize\_debug() function. It limits you to using pipe 0, but gives you the options of selecting RX/TX, payload width, and Enhanced Shockburst.
3. The third init function is nrf24l01\_initialize\_debug\_lite(). It doesn’t initialize all of the registers, but instead sets up only the CONFIG and RX\_PW\_P0 registers and leaves all other registers in whatever state they’re already in (Enhanced Shockburst is left on).

Both of the debug versions of the init functions set CE if the device is going to be a receiver.

After the init function has been called, the device will either be a receiver or a transmitter. If the device is a receiver and CE is high, then you will be monitoring packets 130 uS after the CE pin was set (this is important to remember for a transmitter that is sending packets to it). At this point, you have two options in your microcontroller: interrupts and polling.

1. If you are using interrupts, then you will more than likely put a call to nrf24l01\_read\_rx\_payload() in your interrupt service routine.
2. If you are polling, you should first check the function nrf24l01\_irq\_pin\_active() and then check the function irq\_rx\_dr\_active() to see if you have received a packet. Then you would call nrf24l01\_read\_rx\_paylaod() to get the packet. Once you read the packet in either mode, you should call nrf24l01\_irq\_clear\_rx\_ds() or nrf24l01\_irq\_clear\_all() to clear the RX\_DS interrupt.

In the case that you are using more than one pipe, once you receive a packet, you can check the status register to see what pipe it was received on. The easiest way to do this is to set a variable equal to the nrf24l01\_read\_rx\_payload() function because this function returns the status register at the time of the payload read. Then takeyou take this variable and give it to the nrf24l01\_get\_pipe\_from\_status() function as an argument to find what pipe you received your packet on. Remember that once you read the packet out of the FIFO, you won’t be able to get the pipe it came on. It is important to do this either before you call nrf24l01\_read\_rx\_payload() or use the return value of the function as was just described.

If the 24L01 was made a TX after initialization, then you have a slightly different road to follow. In normal operation, when you decide you want to send a packet, you call the nrf24l01\_write\_tx\_payload() function. If you are using a very high data rate, you should check the to make sure the TX FIFO isn’t full before you send the payload over SPI (this is available in both the STATUS and FIFO\_STATUS registers). Once you send the payload, you wait for the TX\_DS interrupt to go active before you send another packet, which is similar to waiting for RX\_DR with an RX. If you’re using the Enhanced Shockburst feature, you also need to watch for the MAX\_RT interrupt in case your packet never makes it to the receiver. You also need to clear the interrupts when you get them, just as with an RX.

Just as a hint, remember that anytime a receiver calls the nrf24l01\_read\_rx\_payload() function, it needs up to 130 uS to come back to active mode before it starts monitoring for packets again. You need to take this into account with your transmitter and allow for a 130 uS delay when appropriate or else you \*may\* lose packets. I have written code before that didn’t have the delay and still worked. In my experience I have found that this delay seems to be more important to include when you are using Enhanced Shockburst than regular Shockburst.

If in some time during your wireless transceiving you need to change a device from a TX to an RX or vice versa, there are also functions for this. These are, surprisingly enough, nrf24l01\_set\_as\_tx() and nrf24l01\_set\_as\_rx(). There are also parameterized versions of these functions that allow you to change the contents of the whole CONFIG register.

Many devices will want to conserve battery power and keep the 24L01 powered down. The functions nrf24l01\_power\_up() and nrf24l01\_power\_down() make easy work of this task by keeping all of the registers in their current state exceptfor the PWR\_UP bit in the CONFIG register. They also do all of the timing and other operations for you.

As with the mode change functions, these functions also have parameterized versions that allow you to change the entire CONFIG register.

A very useful function for debugging is the nrf24l01\_get\_all\_registers() function. For this guy, you simply pass it an array of 36 bytes and it will read every register in the 24L01. I use it very often when I am writing new code for the library and trying to debug new functions to make sure they are actually doing what I want them to do. The only thing to remember with this function is that it gets all five of the three multi-byte registers (TX\_ADDR, RX\_ADDR\_P0, and RX\_ADDR\_P1).

Most of the other functions in the nRF24L01 library have somewhat more sparse uses, so I won’t be discussing them in this tutorial. All of the functions are well commented in the .c file, so feel free to look there to see what each function does and if it will solve a specific problem for you.

**Local Main File: maintutorial1local.c**

This is the place where the interesting stuff starts. This file is the main routine for the uC that is connected to the serial port of your computer. I am going to try to describe in detail the things that are going on in this file that aren’t PIC18F452-specificto give you a clear vision of the things that are going on.

Let’s get started on the code from the top down. The first few lines of code are the necessary defines and include files for main. The includes are pretty self-explanatory, and are there for the headers for the PIC18F452 defines, stdio, USART library, SPI library, delays library, and nRF24L01 library. Some parts of the next section of code is specific to the PIC18F452, and others are simply declarations for various initialization functions.

Initialize() is simply a function that calls all of the other, more specific, initialization functions. InitializeIO() sets up all of the IO pin directions and functionalities for the device.

The next declaration is the ToggleLED() routine, which is called when I want to change the status of theon-board LED attached to GPIO pin PORTA.0. The final declaration is PIC-specific, and is called to fix errors in the USART that will stop normal operation if you hold down a button on your keyboard.

Now we move on to the main function. This is our primary point of entry to the application and it’s where all the magic will happen, as you will soon see. First, we define a couple of variables to use in the code. These are unsigned char data, which isused to hold the data that we are sending and receiving, and unsigned int count, which is a loop counter.

First we call the Initialize() function, which was mentioned before, to geteverything setup. This includes a call to nrf24l01\_initialize\_debug(), which sets up the 24L01 to be a TX with payload width of 1 byte and no Enhanced Shockburst features. At this point, we move into the main while loop for the program code.

First in the main loop, we check to make sure there aren’t any errors in the USART. Then we wait until we receive a character from the computer. When we have a character ready in the USART, we read it into the variable “data”. Next, we write data as a TX payload with the nrf24l01\_write\_tx\_payload() function. The first argument is a pointer to the data being sent, so we send it the data variable’s address.

The next argument is the length of the payload, which we have set to 1 byte. The final argument for the function is whether to send or not, so we tell it to go ahead and do so.

Now we wait until the packet has been sent. This is done by first monitoring the IRQ pin itself with the nrf24l01\_irq\_pin\_active() function, and then when this function is true, we check to see if the TX\_DS interrupt is active. If both of these are true, we know that we have sent the packet and we can then clear the interrupts with the nrf24l01\_clear\_all() function.

Next, we want to get back the character that we just sent to the other 24L01 over RF. Therefore, we call the nrf24l01\_set\_to\_rx() function to change the local 24L01 to a receiver. This function has one argument, and it is true to set the device into RX mode (CE high) and false to set the device to standby mode (CE low). In this call, we will put the device into RX mode so it will monitor the air.

The next thing we must do is wait until the RX\_DR interrupt is active. However, we don’t want to wait forever if the packet doesn’t make it back to us. Therefore, I haveput in a for loop that will allow the unit to check 25,000 times to see if the interrupt is active (I don’t know how long this goes, but I am guessing on the order of milliseconds).

If we successfully read the IRQ pin and the RX\_DS interrupt both being active, we read the payload, put the character into the data variable, and then exit out of the for loop. If we reach the last iteration and we still haven’t received the character, we set data to a question mark to indicate that the character was not received.

Next up, we once again clear our interrupts and then print what we set the data variable to be in the previous for loop to the screen. Next, we need to wait on the remote unit to get back into active RX mode. This is the purpose of the 130 uS delay (see Table 13 in the 24L01 datasheet). Finally, we toggle the status of the on-board LED to show that all the data operations for this particular character session have been completed. We then go back to the beginning of the loop to check for USART characters again.

The other function of interest is the Initialize() function. This function setsup the IO pins in the PIC18F452, as well as opens the USART and SPI sections for communications. Finally, it sets up the 24L01 using the nrf24l01\_initialize\_debug() function.

**Remote Main File: maintutorial1remote.c**

This guy contains the main routine for the remote unit. The defines before main are identical to those found in maintutorial1local.c, except for that in this file, usart.h is not included since the remote unit doesn’t connect to a computer.

The main function in this file is a bit different than the other. Initially, we set up a variable unsigned char data to receive the data we got over RF and then send it back. Initialize() is also called, just as it is in maintutorial1local.c, but the actual function body is slightly different because we have to setup the 24L01 to be a receiver initially.

Once we get into the main program loop, we sit and wait until we receive a packet, in the same way as in maintutorial1local.c.

Then, we read the data that was received and clear interrupts. Next, we wait on the local unit to go into RX mode with a130 uS delay and set our attached 24L01 to be a transmitter. We then send whatever character we received back to the local unit over RF. When the character has been sent, we clear the interrupts and set up this 24L01 to be a receiver again. At this point, we go back to the top of the loop and wait for more characters.

As mentioned before, the Initialize() routine in this file is slightly different from that in maintutorial1local.c. There is no call to OpenUSART(), because we arenot using USART for this node. The other difference is the call to nrf24l01\_initialize\_debug().

Here, the first argument is false to tell the 24L01 to be a TX.