Tutorial 1: Getting a Simple Link Going with the nRF24L01 and the nRF24L01 C library

Written by Brennen Ball, © 2007

Introduction

Hopefully you've made it through Tutorial 0 and now you have a pretty good grasp on the basic operation of the nRF24L01. In this tutorial, we'll actually get into doing some hardware and software instead of a bunch of boring old reading so you can get a nifty little RF link up and running.

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 Hyperterminal, 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.

In case you didn't catch it from the description, you will need two microcontrollers and two 24L01's. This is pretty common sense, though, because to get a wireless link you are obviously going to have to have two 24L01's, and each of them requires a microcontroller to do anything useful with. You will also need a level shifter circuit (like the MAX232 for 5V or the MAX3232 for 3.3V) to allow you to communicate with your computer's serial port for one of the microcontrollers.

The Hardware I Used for this Tutorial

First off, I'll give you a run down of the hardware I'm using. The wireless boards that I have are the MiRF-v2 from Sparkfun Electronics (described extensively in the section titled "A Closer Look at the MiRF-v2 Breakout of the nRF24L01" in Tutorial 0).

The microcontroller that I'm using is the NXP (formerly Philips) LPC2148. It's an ARM7TDMI-based uC with integrated USB, 2 UARTs, 2 SPIs, 512 kB of program memory, and a ton of other features. The breakout board for the LPC2148 that I will be demonstrating with is the LPC-H2148 (see Figure 1 on the next page for a picture), which is also available from Sparkfun and is made by Olimex (http://www.olimex.com). At $40, it's not a bad price to have a 32-bit processor on a development board with all of

those features. The only pain with it is that you have to create a board to mount it to to break out the peripherals, but this isn't too difficult.

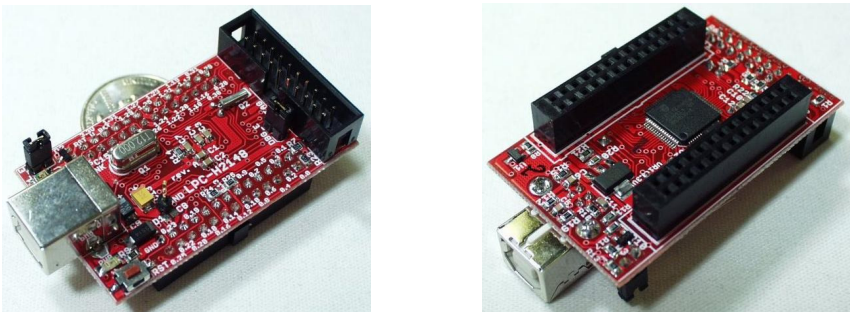

Figure 1. Top (left) and bottom (right) view of the LPC-H2148 board (pics from www.sparkfun.com)

As I mentioned, you must connect one of your microcontrollers to your computer over a serial port (RS232). You will need some type of level conversion for this to work, and I personally use MAX3232 chip, which allows the user to run at 3.3V and still get a working RS232 link. I have connected this chip up per the datasheet and it works flawlessly. If you are using an LPC chip, make sure that the RTS and DTR lines used by the programmer are disconnected when you are communicating to the PC with Hyperterminal. If they are left connected, your chip may remain in a state of reset and it will seem as if the chip is doing nothing.

Hardware Requirements for Non-LPC213x/LPC214x Users

Remember that you don't have to have the same micro as I do to get this code to work. Most any micro that has built-in SPI will work just fine, even if it works at 5V because of the 5V-tolerant inputs on the 24L01. You will also need at least two, but up to three free GPIO ports that (definitely one pin each for CE and CSN, but optionally one additional pin for IRQ). The only issue with using other micros is that you will have to write your own delay, SPI, and UART functions. This isn't terribly difficult, but it will set you back some time getting everything working if your IDE/compilation suite doesn't have them built-in.

Pin Connections to the MiRF-v2 Breakout Board

The hardware pin connections to the 24L01 are pretty simple, and here is how I have my stuff connected, from the perspective of the 24L01. For those of you also using the LPC2148, I have included the way that the pins are set up on my micro, including IO function select and input/output. Just as a caveat, I am using SPI1 rather than SPI0.

<u>24L01 pins</u>
(1) Vcc -> +3.3Vdc supply
(2) CE -> uC pin P0.21, configured as GPIO/input
(3) CSN -> uC pin P0.20, configured as GPIO/output
(4) SCK -> uC pin P0.17/SCK1, configured as SCK1(SPI1)/output

(5) MOSI -> uC pin P0.19/MOSI1, configured as MOSI1(SPI1)/output
(6) MISO -> uC pin P0.18/MISO1, configured as MISO1(SPI1)/input
(7) IRQ -> uC pin P0.15, configured as GPIO/input
(8) GND -> common digital ground

I apologize for the ASCII wiring hardware connections. I don't have a circuit CAD program handy, but since there aren't a terribly large amount of pins to connect, I think that you should probably be able to handle the task.

Pin Connections to the MAX3232 Level Converter

For this tutorial to work, you're also going to have to get some sort of level converter to hook up the local microcontroller to your serial port. In Figure 2, I have included the application circuit from the MAX3232 datasheet. To get this everything going, you have to hook pins up to the MAX3232 as follows:

MAX3232 pins
(16) Vcc -> +3.3Vdc supply
(11) T1IN -> uC pin P0.0/TX0, configured as TX0(UART0)/output
(14) T1OUT -> RS232 pin 2
(12) R1OUT -> uC pin P0.1/RX0, configured as RX0(UART0)/input
(13) R1IN -> RS232 pin 3
(15) GND -> common digital ground

Don't forget that you also have to connect pin 5 of your RS232 connector to the same ground as your level shifter is connected.

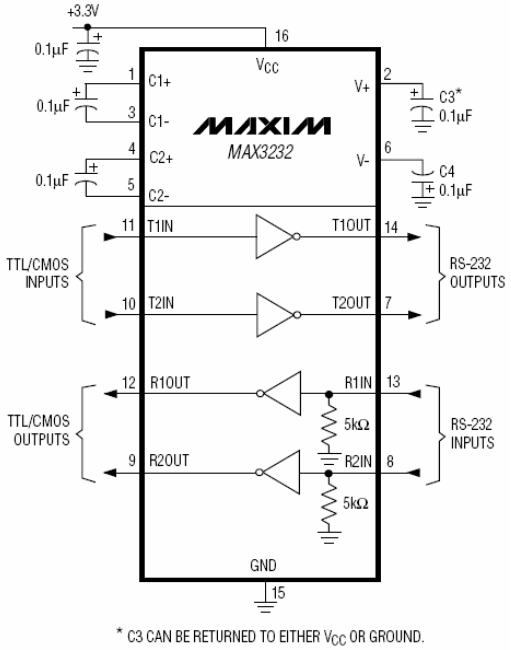

Figure 2. Application circuit of the MAX3232 from the datasheet

Tutorial Software Description/Requirements

The code for this tutorial was written in C and has been written to be compiled under the GNUARM suite. The library code for the 24L01 (nrf24l01.h and nrf24l01.c) *should* work with pretty much any C compiler because it doesn't have anything other than straight C code in it.

Some of the support files included in this tutorial (the makefile and linker script, as well as some of the routines in the main files) are based off Jim Lynch's examples in his ARM tutorial. If you would like more help on these files, please consult his excellent tutorial on getting started with ARM processors.

Within the nRF24L01 library, you will find defines for all of the registers, fields within the registers, register default values, opcodes, and functions that are necessary for getting your link up and running. There are also a ton of other functions that will make things easier for you to get things done, and I constantly am making additions to the library as I use it more and more and find the need for more functionality.

One issue that has been found is that since the library is rather large, users that have very small program memories in their micros sometimes don't have enough space to link the code properly. I am planning on working on a lite version of the library for these users, so stick around for that in the future.

Compiling the Tutorial Code

In order to compile the code as-is, you need to install Cygwin (if you're using a Windows-based computer) and GNUARM. This is described in great detail in Jim Lynch's ARM tutorial. Make sure that you have included the directories of all of the binary files in your PATH environment variable or else you may get errors when you try to compile.

The project comes pre-compiled for those of you who are using a uC that is compatible with code generated for the LPC2148. The .hex files that you can upload to your local and remote microcontrollers are contained in the "hex" folder. Simply use the LPC2000 Flash Utility to load them up and you should be off and running. As I mentioned before, make sure that you disconnect RTS and DTR before you run the code on the local uC.

If you would like make changes to the code and compile the tutorial yourself, simply point a command prompt or shell to the "src" directory. First, type "make clean" at the command prompt (without the quotation marks). Then, type "make" (also without the quotation marks). If you have all of the necessary tools working properly, you should have a new set of .hex files in the hex directory that you can upload.

For those of you who aren't using LPC2148-compatible uC's, feel free to change the makefile as you need to get things going. If you don't know how to do this, I will be of very limited assistance since I'm not that well-versed with complicated makefiles and other compilation suites besides GNUARM. There should more than likely be examples for whatever suite you're using on the web, so try Google to see if you can find some good stuff there.

LPC-specific Support Files

There are a few files included in the software that are specific to the processor that I'm using. These are lpc214x.h, crt.s, demo2148_blink_flash.cmd, delays.h, delays.c, spi1.h, spi1.c, uart0.h, and uart0.c. The file lpc214x.h is an include file that contains defines for all of the registers within the LPC2148. Please note that I have modified this file somewhat from the original version to include some additional register definitions, and the main code depends on these defines to be there. Crt.s and demo2148_blink_flash.cmd are files that were written by Jim Lynch, and are the assembly start-up file and the linker script for the LPC2148, respectively. The other .c and .h files mentioned are libraries to do specific features in the LPC2148, and will likely not work with your processor if it isn't an ARM-based LPC.

NRF24L01.h and .c

Since the source files in the nRF24L01 library are pretty well commented, I don't really want to use this space to write redundant information. Rather, what I do want to do here is to give you an outline of the normal procedures of using my library in your own projects.

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 that byte over SPI, receives a byte from the 24L01 over SPI, and then returns this received 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. 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. 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. 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. If you are using interrupts, then you will more than likely put a call to nrf24l01_read_rx_payload()in your interrupt service routine. 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 take you 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 except for 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 LPC2148-specific to 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 lpc214x defines, SPI1 library, delays library, nRF24L01 library, and UART0 library.

Some parts of the next section of code is specific to the LPC2148, and others are simply declarations for various initialization functions. First is the define PLOCK, which is used in the InitializePLL() function. Initialize() is simply a function that calls all of the other, more specific, initialization functions. InitializePLL() is there to set the PLL up to 5x the external oscillator and set up the peripheral clock to run at this same frequency InitializeIO() sets up all of the IO pin directions and functionalities for the device. The Feed() routine is used by the InitializePLL() routine to get the PLL going.

The following four declarations are for the various interrupt sources on the chip. These are not actually used, so I won't really describe them. The last of the declarations is the ToggleLED() routine, which is called when I want to change the status of the on-board LED attached to GPIO pin P1.24.

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 is used 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 get everything 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 wait until we receive a character at the UART. When we have a character ready in the UART, 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 have put 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 UART characters again.

The other function of interest is the Initialize() function. This function sets up the PLL and the IO pins in the LPC2148, as well as opens UART0 and SPI1 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, uart.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 a 130 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 uart0_open(), because we are not using UART0 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.

Actual Program Output

In Figure 3, you can see the window in AbsoluteTelnet that shows the program working (AbsoluteTelnet is similar to HyperTerminal). The 24L01s are in very close proximity to one another (around 6" apart), so there is an extremely high percentage of successful packets. Only one of the packets actually did not make it through. This packet can be seen on the 4th line and the 8th character as a question mark.

For those of you using HyperTerminal, you may have to alter the way your return key is set up within HyperTerminal's settings in order for it to send the proper characters (a carriage return and line feed combination). Make sure that you are disconnected in HyperTerminal by pressing the "Disconnect" button. Then, click File->Properties, and on the "Settings" tab, click the "ASCII Setup" button. Check the box beside "Send line feeds with line ends" (it's the very first check box). Take a look at Figure 4 to see what I'm talking about.

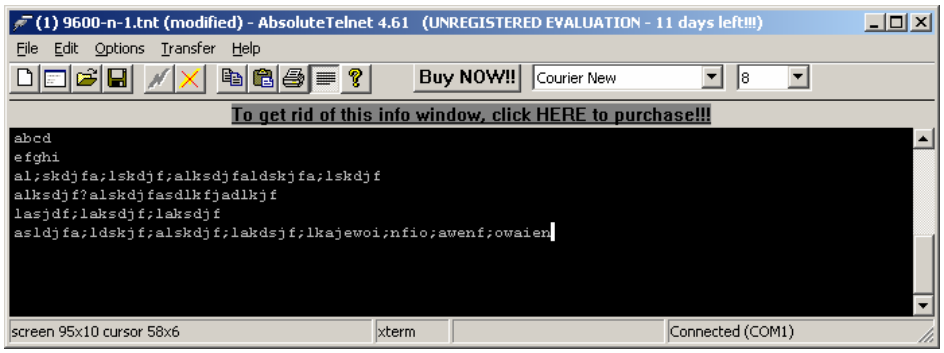

Figure 3. Output window from AbsoluteTelnet, a similar program to HyperTerminal

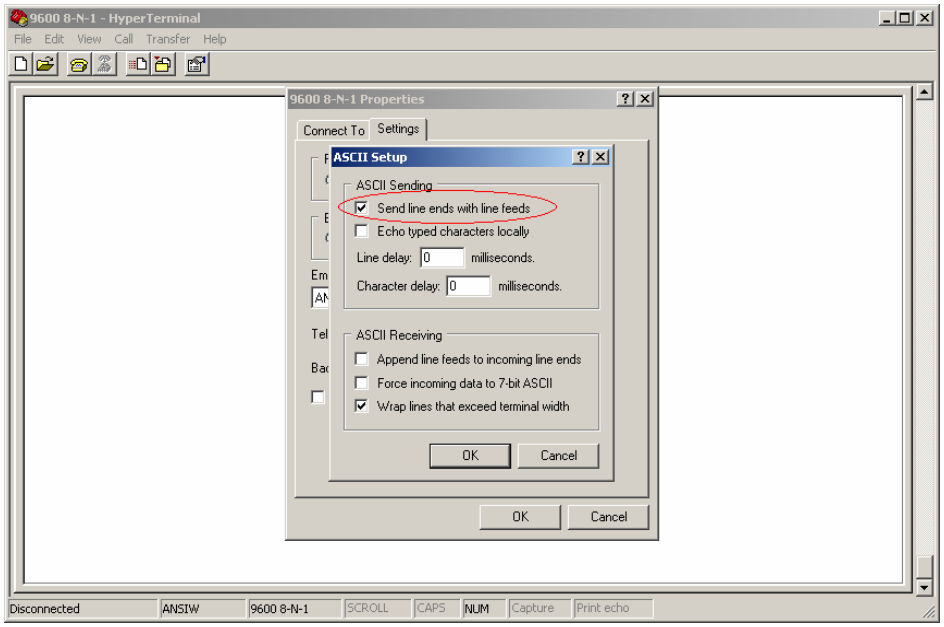

Figure 4. Setting up HyperTerminal to send CR/LF pairs when pressing the Enter key

Concluding Remarks

At this point, you should have a working link that you can use in future tutorials and in your own projects. You should also now have a basic understanding of the flow of the more important functions in the nRF24L01 library. If you have any questions, feel free to email me at brennen@diyembedded.com.