

Tutorial 0: Everything You Need to Know about the nRF24L01 and MiRF-v2

Written by Brennen Ball, © 2007

Introduction

So you think that getting a wireless connection between two or more microcontrollers is hard, eh? Well, it definitely can be if you're using certain hardware. I'm going to show you that it's not hard or expensive to set up a quality wireless link. You'll wonder why you ever used cables!

Enter the nRF24L01 2.4 GHz wireless chip from our friends at Nordic Semiconductor. This chip was released a few years ago and is the successor to the popular nRF2401 series of chips. The 24L01 takes all of the good features of the 2401 and adds a true SPI interface, hardware link layer, multiple pipelines, and more. Plus, the chip is cheaper than your average microcontroller, and it doesn't require too many external components.

So what is my motivation for doing this? I don't work for Nordic or Sparkfun or any other company mentioned in this document. I really just do this because I like these transceivers and I would like to see more people use them. I hope that these tutorials help you in your quest for wirelessness!

Background Information

This document is essentially background information for the chip itself. I have included explanations of as many things as I could think of, as well as some suggestions for settings and operations. If you would like to skip directly to the tutorial-related information, please skip this tutorial and move on to tutorial 2. However, if you are a beginner with this device, it would almost certainly help you to at least skim through the background sections of this tutorial. If anything, keep it around for future reference when you need suggestions or help on certain parts of your quest to the world's coolest new widget!

Required Reading

Yes, I know it sounds like a homework assignment, but you have to do it. If you're going to be successful with this little venture, you absolutely have to know your microcontroller, and have a copy of the datasheet/user manual either on paper or in digital format on your machine. Also, you should have the schematic for the PCB/breakout board that you have your micro on.

For the 24L01, you absolutely have to download a copy of the datasheet (www.sparkfun.com/datasheets/Components/nRF24L01_prelim_prod_spec_1_2.pdf)! While I will attempt to instruct you to the point that you don't have to have it, there are

still some sections that I don't describe in great detail because it is blatantly spelled out in the datasheet (register definitions in particular). Finally, if you have a breakout of the 24L01, such as the MiRF-v2, you also need to obtain the schematic and the datasheet/user manual for it.

NRF24L01 Hardware Interface

More than likely, you are using a breakout for the 24L01 of some sort. If so, you will likely have eight pins to interface with, and these are Vcc, GND, IRQ, CE, and the four SPI-related pins (CSN, SCK, MISO, and MOSI). If you are dealing with the bare chip, then I would suggest looking at the example circuit which can be found in Figure 13 of the datasheet, as I will not be discussing the use of any pins besides the eight just listed. Note: some of this information may be covered again as necessary in different sections.

Thankfully for us, the guys at Nordic made the IO pins 5V-tolerant, which means you can run your microcontroller at 5V and it won't fry this chip. Remember, though, that **you can't run the 24L01 at 5V** – the data sheet gives an operating range of 1.9 to 3.6V for Vcc (the chip will be run at 3.3V in most circumstances).

The SPI interface uses four pins, CSN, SCK, MISO, and MOSI for data transmission and reception. The CSN (chip select not) pin is active-low, and is normally kept high. When this pin goes low, the 24L01 begins listening on its SPI port for data and processes it accordingly. The remaining three pins should be tied to the user's hardware SPI interface, to the same pins as their name suggests (SCK to SCK, MISO to MISO, and MOSI to MOSI).

The remaining two pins are CE and IRQ. CE is used to control data transmission and reception when in TX and RX modes, respectively. IRQ is the interrupt pin, and is active-low. There are three internal interrupts that can cause this pin to go low when they are active. Each of these bits can be masked out such that when the bit's respective interrupt becomes active, the status of the IRQ pin is not changed.

A Closer Look at the MiRF-v2 Breakout of the nRF24L01

For this tutorial, I will be describing the MiRF-v2 breakout of the nRF24L01 that is available from Sparkfun Electronics (<http://www.sparkfun.com>). These boards essentially give you all the supporting circuitry to get the link operating (matching network, clock, voltage regulator, and antenna/antenna jack) so that you don't have to lay out your own board.

There is another flavor of this board (the MiRF-v2 RP-SMA) from Sparkfun that gives you a coax jack to connect an antenna to (both units pictured in Figure 1 on the following page). I bought the plain MiRF-v2 board before the RP-SMA version came out, and it is considerably more expensive (\$30 versus \$15), but it does come with an on-board antenna. Other than the price, the other disadvantage of the plain MiRF-v2 is that it has considerably shorter range because the antenna runs over a ground trace. Either version will work, but don't forget that if you buy the RP-SMA version, you also have to buy an antenna for it. Not surprisingly, the guys at Sparkfun sell those, too, for around \$7.



Figure 1. Left to right: MiRF-v2 top, MiRF-v2 RP-SMA top and bottom (pics from www.sparkfun.com)

Let's start out by looking at the pins we have on the MiRF-v2 board. If you don't have a MiRF-v2, this will still be helpful because the pins I describe are named the same as they are on the 24L01 chip itself. Starting at the top, you have Vcc. This pin is connected to the input of a voltage regulator, and can handle 3.3 to 7 Vdc per Sparkfun's website. I personally connect it to 3.3V because that's the main power rail that I use with my microcontroller. Note: if you aren't using the MiRF-v2, then you should run 3.3V to the Vcc pin, unless otherwise noted.

The second pin down is the CE pin. This pin is always an input with respect to the 24L01. This pin has different meanings depending on what mode the 24L01 is in at a given time. First we assume that the device is powered-up internally (by setting the PWR_UP bit in the CONFIG register...details on how to do that later). If the device is a **receiver**, having CE high allows the 24L01 to monitor the air and receive packets. CE being low puts the chip in standby and it no longer monitors the air. If the device is a **transmitter**, CE is **always held low** except when the user wants to transmit a packet. This is done by loading the TX FIFO and then toggling the CE pin (low-to-high transition, leave high for at least 10 uS, then back to low).

The third pin is CSN, which stands for chip select not. This is the enable pin for the SPI bus, and it is active low (hence the "not" in the name). You always want to keep this pin high except when you are sending the device an SPI command or getting data on the SPI bus from the chip.

The fourth pin from the top is SCK, which is the serial clock for the SPI bus. When you configure your SPI bus, SCK should stay low normally (rising edges are active), and the clock samples data in the middle of data bits. On my hardware, this configuration is represented by the values of the bits CPOL and CPHA in my microcontroller's SPI configuration register both being 0.

Coming in fifth from the top is the MOSI pin. MOSI stands for "master out, slave in," and from both the microcontroller's and the 24L01's perspectives, the master is the microcontroller and the slave is the 24L01. This is because the 24L01 never sends data without first being requested by the microcontroller. Essentially, this pin is the side of the bus on which the master (the microcontroller) sends data to the slave (the 24L01). It is also connected to the MOSI pin on your microcontroller's SPI interface.

Up sixth is the MISO pin. This pin is like the MOSI pin, but backwards. MOSI stands for, you guessed it, "master out, slave in." This pin is the side of the bus on which

the slave (the 24L01) sends data to the master (the microcontroller). As you can see, SPI is a full-duplex bus, meaning that you can send data in both ways at once, but on separate lines.

The SPI pins are all done, so now let's get to the seventh pin from the top, the IRQ pin. This guy is basically the interrupt pin to signal your microcontroller that something interesting has happened. You can set interrupts for any combination of the following events: data received, data transmitted, and maximum number of transmit retries reached (this particular one is related to the 24L01's hardware link-layer, more on that later). If you're not using interrupts, this pin isn't required because you can poll the 24L01's STATUS register over SPI to see if any interrupt has occurred, but it's still faster (in general) to check the status of an IO pin than to send an SPI command and then wait for the response. NOTE: the IRQ pin is ACTIVE-LOW. It is normally high, but when an interrupt is asserted, the pin will go low.

The bottom-most pin is GND, aka ground. I think we all know what goes here, so just connect it to the same ground as your microcontroller is hooked to.

Interfacing the nRF24L01 via SPI

Now that you know about the SPI interface pins and their purposes and how to connect them to your microcontroller's hardware SPI port, we can discuss the 24L01's SPI interface a little more in-depth. The SPI interface allows you to read/write registers, transmit data, receive data, and do all sorts of other things internal to the 24L01. The 24L01's predecessor, the 2401, had a one-wire interface that was rather cumbersome to interface. Thankfully for us, the guys at Nordic stepped it up on the 24L01 to a true SPI interface that is very simple to communicate with by almost any microcontroller.

The datasheet is contradictory in terms of the maximum SPI data rate. On the p. 1, it states that the max is 8 Mbps. However, on p. 19, a max of 10 Mbps is stated. I haven't ever pushed the max to see what it really is, so I can't really state for sure which of these numbers is correct. I normally just run my SPI at 2 Mbps because it was a simple thing to start out at and I never got around to upping my SPI data rate. Generally, the only reason you would need to push the max is if you needed really huge on-air data rates.

From what I've found, most microcontrollers that have hardware SPI ports have very similar configuration registers, as well. I have the bits CPOL and CPHA both at 0 in my microcontroller's SPI configuration register to interface to the 24L01. CSN was connected to a GPIO pin on my microcontroller since the microcontroller was acting as a master device (SSEL only works in slave mode). Also, you will want to set up your SPI for 8-bit data transfer. Finally, your SPI port should be set up as a master. Figure 2 on the following page shows the bit-wise operation for read and writes operations over SPI.

In terms of timing, almost none of the SPI-related timing will apply unless you have an absolutely blazing fast microcontroller (as long as you don't exceed the maximum SPI data rate). Tables 9 and 10 on p. 21 of the datasheet outline the timing requirements, and most of them are below 50 ns, which is less than one instruction cycle for most of the microcontrollers hobbyists are going to use. If you are indeed using a really quick micro, you will need to consult these tables to insure that you are keeping the proper timing.

Cn – SPI Instruction Bit
 Sn – Status Register Bit
 Dn – Data Bit (note: LSByte to MSByte, MSBit in each byte first)

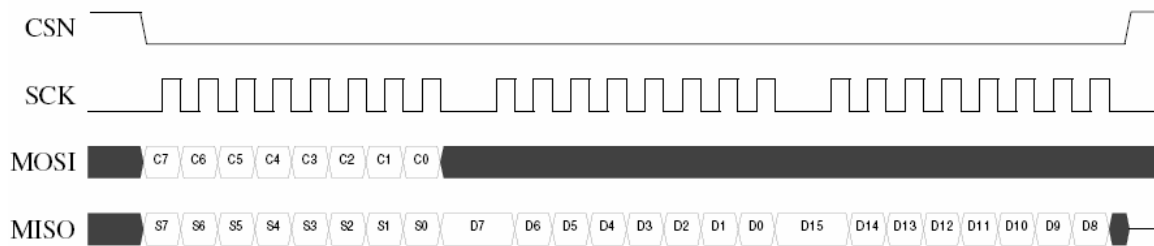


Figure 8 SPI read operation.

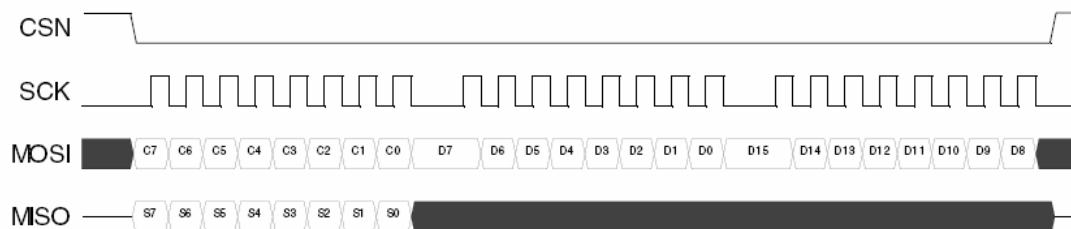


Figure 9 SPI write operation.

Figure 2. SPI timing for read (top) and write (bottom) SPI operations from Figures 8 & 9 in the datasheet.

If you're not using my SPI code in the tutorials to follow (basically if you're not using an LPC chip with SPI1 on it), then you will have to write your own SPI routines. You should become very intimate with your datasheet here, and I highly recommend using loopback connections to test your port. Many micros have built-in loopback functionality such that you can test the SPI internally to make sure you are sending and receiving data properly. If you get this working, try connecting your MOSI pin to your MISO pin to see if you are reading the same data you are sending (make sure nothing else is connected to either the MOSI or MISO pins during this test). If you can do this, you are well on your way to getting a working SPI link to the 24L01.

SPI Instruction Set Summary

Talking to the 24L01 is very simple once you have things set up and working on your micro's SPI port. Once you get hooked up and ready to go, you're going to want to start talking to the 24L01, and I'm going to show you how to do it with the SPI instruction set. This information can be found in Table 8 on p. 19 of the datasheet, as well as in Figure 3 on the following page.

Instruction Name	Instruction Format [binary]	# Data Bytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read registers. AAAAA = 5 bit Memory Map Address
W_REGISTER	001A AAAA	1 to 5 LSByte first	Write registers. AAAAA = 5 bit Memory Map Address <i>Executable in power down or standby modes only.</i>
R_RX_PAYLOAD	0110 0001	1 to 32 LSByte first	Read RX-payload: 1 – 32 bytes. A read operation will always start at byte 0. Payload will be deleted from FIFO after it is read. Used in RX mode.
W_TX_PAYLOAD	1010 0000	1 to 32 LSByte first	Used in TX mode. Write TX-payload: 1 – 32 bytes. A write operation will always start at byte 0.
FLUSH_TX	1110 0001	0	Flush TX FIFO, used in TX mode
FLUSH_RX	1110 0010	0	Flush RX FIFO, used in RX mode Should not be executed during transmission of acknowledge, i.e. acknowledge package will not be completed.
REUSE_TX_PL	1110 0011	0	Used for a PTX device Reuse last sent payload. Packets will be repeatedly resent as long as CE is high. TX payload reuse is active until W_TX_PAYLOAD or FLUSH TX is executed. TX payload reuse must not be activated or deactivated during package transmission
NOP	1111 1111	0	No Operation. Might be used to read the STATUS register

Figure 3. SPI instruction set from Table 8 in the nRF24L01 datasheet

In order to send data to or receive data from the SPI port on the 24L01, you have to do a few things. The CSN pin on the 24L01 must be **high** to start out with. Then, you bring the CSN pin **low** to alert the 24L01 that it is about to receive SPI data. (Note: this pin will stay low throughout the entire transaction.) Then, you will transmit the command byte of the instruction you wish to send. If you are receiving data bytes for this instruction, you must then send one byte to the 24L01 for every one byte that you wish to get out of the 24L01. If you are just sending the 24L01 data, you simply send your data bytes and generally don't worry about what it sends back to you. When receiving data from the 24L01, it makes absolutely no difference what is contained in the data bytes you send after the command byte, just so long as you send the correct number of them. Once you have transmitted and/or read all of the bytes that you need, you bring CSN back high. Finally, you can process this data in your micro at will.

As an example, let's say you are going to execute the R_REGISTER instruction on TX_ADDR register, which will read the contents of the TX address register out of the 24L01 and into your micro (more on the instruction set next). The TX_ADDR register is 5 bytes wide, and we'll assume that you are using 5-byte addresses.

First, you would bring CSN low and then send the command byte '00010000' to the 24L01. This instructs the 24L01 that you want to read register 0x10, which is the TX_ADDR register. Then you would send five dummy data bytes (it makes absolutely no difference what the data bytes contain), and the 24L01 will send back to you the contents of the TX_ADDR register. Finally, you would bring the CSN pin back high.

All totaled, you will receive six bytes. When you send any command byte, the 24L01 **always** returns to you the **STATUS register**. After that, you will have received the

five bytes that are contained in the TX_ADDR register. Note: I will be describing the registers in the next section, so don't worry if you don't know them that well yet.

Now that you know the format of sending instructions to the 24L01, I will describe the actual instruction set. First off is the R_REGISTER instruction. This instruction allows you to read any of the registers within the 24L01. It has the binary format of '000AAAAA', where 'AAAAA' is the **address** of the register that you want to read. This can be in the range of 0x00 to 0x17, or '00000' to '10111' in binary (these are given in Table 11 of the datasheet). Conveniently for you, I have included defines for both the register addresses and the instruction set binary formats in the header file for my library, so you don't have to worry about the hex values. You will be sending one to five data bytes for this command, depending on what register you're reading (the address registers can up to five bytes, but all the other registers are one byte).

Next is the W_REGISTER instruction, which allows you to write most any of the registers in the 24L01. It is very similar in format to the R_REGISTER instruction, with a binary format of '001AAAAA', where 'AAAAA' is the address of the register you want to write. Once you send the instruction byte, you will once again send one to five data bytes, depending on the register you're writing to, with the same rules for the registers as in the R_REGISTER.

Now for the first of the two "big-money" operations – R_RX_PAYLOAD. This operation allows you to read the contents of the RX FIFO if you have received a packet (when you are in RX mode), which is generally signaled by the RX_DR interrupt. Using this operation is a little more involved than the others, because it requires you to also use CE.

When you are receiving packets, CE is held high. Once you have received a packet you **MUST** bring CE low to disable the receiver, and then you execute the R_RX_PAYLOAD operation. You send the command byte (0x61), and then send the same amount of dummy bytes as your payload width is for the pipe you received data on. The 24L01 will automatically delete the payload that you read out from the top of the FIFO. If there are more payloads in the FIFO (the device can hold **3 payloads** in the RX FIFO – see the "FIFO Info" section for more details), then you should continue reading them out until they are all read. At this point, you would clear the RX_DR interrupt, bring CE high again to monitor for packets, and then process the data you received.

The second of the "big-money" operations is W_TX_PAYLOAD. This guy is used when you are in TX mode and wish to send a packet. In normal TX operation, the CE pin is held low. You first send the command byte (0xA0), and then the payload. The number of payload bytes you send **MUST** match the payload length of the receiver you are sending the payload to.

Once you load the packet, you toggle the CE pin to send the packet on its way (keeping it high for at least 10 us). If the packet was sent, the TX_DS interrupt will occur. This is to signal to you that the 24L01 has successfully sent your packet. If you had auto-ack enabled on this pipe, the TX_DS flag will only be set if the packet actually gets through. If the maximum amount of retries is hit, then the MAX_RT interrupt will become active. At this point, you should clear the interrupts and continue based on which interrupt was asserted. Also remember that, like the RX FIFO, the TX FIFO is **three levels** deep. This means that you can load up to three packets into the 24L01's TX FIFO before you do the CE toggle to send them on their way.

The FLUSH_TX and FLUSH_RX operations (opcodes 0xE1 and 0xE2, respectively) are used to clear out any data in the TX and RX FIFOs, respectively. This can be useful if you should happen to receive a packet that you know is not valid without even checking it (like during setup, etc.). These operations have no data bytes.

An operation that I have never found a real use for is REUSE_TX_PL. This allows you to constantly send a packet as long as CE is held high for a TX device (the status of this operation is reflected in the TX_REUSE field of the FIFO_STATUS register). The opcodes for this guy is 0xE3 and it has no data bytes. If you should decide to execute this operation, you can stop it temporarily by bringing CE low. To stop it permanently, you can stop it by issuing either the W_TX_PAYLOAD operation or the FLUSH_TX operation. You should be sure to not execute this instruction while the 24L01 is sending a packet though, because it could cause strange stuff to happen.

The final instruction the 24L01 knows is the NOP. It has an opcode of 0xFF and no data bytes. The only real use for this operation is to read the STATUS register quickly, but this does indeed come in handy, especially if you're not using the IRQ pin.

A Quick Overview of the nRF24L01's Internal Registers

Below you can find descriptions of each register, as well as the information from the datasheet for the register below the description. I will point out some important points to each register and potential pitfalls that I have run into. Oh, and also remember that the addresses of the registers are in HEX, not in decimal

The first register (well, the zeroth technically, but I digress) is the CONFIG register. It is the **most important** register as far as getting the 24L01 to do something goes. This register contains the PWR_UP bit, which **MUST** be set in order to make the device do anything. Therefore, if you want to get the 24L01 talking, you have to write to this register at least once. Also important in this register is PRIM_RX bit, which, if set, makes the device a receiver, and if cleared, makes the device a transmitter. You configure your interrupts here as well, and you also set up your CRC scheme. I personally recommend a 2-byte CRC, but it depends a lot on how much data rate you need. The register description can be found on the following page in Figure 4.

The next register is EN_AA, or the enable auto-ack register. This register allows you to enable or disable auto-acknowledgements on a per-pipe basis. If you are at all familiar with TCP, you know what acknowledgements are. Essentially, the transmitting 24L01 will send a packet and then **switch momentarily for a receiver**. If the receiving 24L01 gets the packet, it will send back an acknowledgement. If the transmitting 24L01 gets the acknowledgement, it **changes back** to a transmitter and all is happy. If the transmitting 24L01 doesn't get the acknowledgement back in the specified window, it sends the packet again and waits for an ack. It will do this the number of times allowed in the SETUP_RETR register for the pipe being used, and then if no acknowledgement is received it will assert the MAX_RT interrupt. This is an extremely useful feature and I would recommend using it if possible. One caveat is that you must have the receive address of pipe 0 be the **same as** the transmit address if you have auto-ack enabled on the pipe you're sending data to/from (see p.14 of the datasheet). The register description can be found on the following page in Figure 5.

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the IRQ pin 0: Reflect RX_DR as active low interrupt on the IRQ pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the IRQ pin 0: Reflect TX_DS as active low interrupt on the IRQ pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the IRQ pin 0: Reflect MAX_RT as active low interrupt on the IRQ pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' - 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0: POWER DOWN
	PRIM_RX	0	0	R/W	1: PRX, 0: PTX

Figure 4. CONFIG register

01	EN_AA Enhanced ShockBurst™				Enable 'Auto Acknowledgment' Function Disable this functionality to be compatible with nRF2401, see page 26
	Reserved	7:6	00	R/W	Only '00' allowed
	ENAA_P5	5	1	R/W	Enable auto ack. data pipe 5
	ENAA_P4	4	1	R/W	Enable auto ack. data pipe 4
	ENAA_P3	3	1	R/W	Enable auto ack. data pipe 3
	ENAA_P2	2	1	R/W	Enable auto ack. data pipe 2
	ENAA_P1	1	1	R/W	Enable auto ack. data pipe 1
	ENAA_P0	0	1	R/W	Enable auto ack. data pipe 0

Figure 5. EN_AA register

Register 0x02 is EN_RXADDR. This register is very cut-and-dried: write a 1 to the bit for any pipe you want to enable, and a 0 to disable the pipe.

02	EN_RXADDR				Enabled RX Addresses
	Reserved	7:6	00	R/W	Only '00' allowed
	ERX_P5	5	0	R/W	Enable data pipe 5.
	ERX_P4	4	0	R/W	Enable data pipe 4.
	ERX_P3	3	0	R/W	Enable data pipe 3.
	ERX_P2	2	0	R/W	Enable data pipe 2.
	ERX_P1	1	1	R/W	Enable data pipe 1.
	ERX_P0	0	1	R/W	Enable data pipe 0.

Figure 6. EN_RXADDR register

Up next is SETUP_AW. Here is where you set your address width (this MUST match for both the transmitter and receiver). I personally recommend 5 address bytes unless you absolutely need the bandwidth, because the more address bytes (and CRC bytes) you have, the fewer miscellaneous packets you'll receive. This is especially good if you happen to have your 24L01's close to other devices that might operate in the 2.4 GHz range (wireless routers, etc.).

03	SETUP_AW				Setup of Address Widths (common for all data pipes)
	Reserved	7:2	000000	R/W	Only '000000' allowed
	AW	1:0	11	R/W	RX/TX Address field width '00' - Illegal '01' - 3 bytes '10' - 4 bytes '11' - 5 bytes LSByte will be used if address width below 5 bytes

Figure 7. SETUP_AW register

The aforementioned SETUP_RETR register is the next register. This register allows you to set how many retries you will use (common to all pipes) if you have EN_AA enabled on a pipe you're sending/receiving data from in the ARC field. The default value is 3 and this may or may not work depending on your application. The ARD field allows you to change the time delay that a retransmit waits. I would personally leave it at the default, but feel free to experiment based upon your situation.

04	SETUP_RETR				Setup of Automatic Retransmission
	ARD	7:4	0000	R/W	Auto Re-transmit Delay '0000' - Wait 250+86uS '0001' - Wait 500+86uS '0010' - Wait 750+86uS '1111' - Wait 4000+86uS (Delay defined from end of transmission to start of next transmission) ¹⁴
	ARC	3:0	0011	R/W	Auto Retransmit Count '0000' - Re-Transmit disabled '0001' - Up to 1 Re-Transmit on fail of AA '1111' - Up to 15 Re-Transmit on fail of AA

Figure 8. SETUP_RETR register

Register 0x05 is the RF_CH register, which allows you to set your RF channel (I'm sure you couldn't have figured that out without me telling). The register allows you to set the channel from 0 to 127, and each channel has a separation of 1 MHz (typically). One caveat is to watch out for the ISM band guidelines of the country in which you are operating your 24L01. The FCC in the US only allows you to go from 2.4 to 2.4835 GHz in the ISM band, which gives you 835 MHz or the first 84 channels to use (read channels

0 to 83). Any channel higher than 83 is off limits in the states, so stay away from that frequency range if you want to avoid potential trouble with Uncle Sam and the FCC.

05	RF_CH				RF Channel
	Reserved	7	0	R/W	Only '0' allowed
	RF_CH	6:0	0000010	R/W	Sets the frequency channel nRF24L01 operates on

Figure 9. RF_CH register

The next register is the RF_SETUP register. This guy contains all the necessary parameters (besides the channel number) to set up the RF section of the 24L01. Be sure to leave PLL_LOCK and LNA_HCURRE at their defaults at all times because changing them could cause stuff to go weird. The RF_DR field allows you to change your data rate, and I would recommend using 2 Mbps as long as you have a decent quality link. Going to 1 Mbps will allow you to get better range and a larger percentage of packets though, with the obvious tradeoff of lower bandwidth. You can also change the RF power level using the RF_PWR field. I would recommend leaving it at its highest value of binary 11, which is also the default. You can lower this if you need to save power and/or have devices really close together.

06	RF_SETUP				RF Setup Register
	Reserved	7:5	000	R/W	Only '000' allowed
	PLL_LOCK	4	0	R/W	Force PLL lock signal. Only used in test
	RF_DR	3	1	R/W	Data Rate '0' – 1 Mbps '1' – 2 Mbps
	RF_PWR	2:1	11	R/W	Set RF output power in TX mode '00' – -18 dBm '01' – -12 dBm '10' – -6 dBm '11' – 0 dBm
	LNA_HCURRE	0	1	R/W	Setup LNA gain

Figure 10. RF_SETUP register

Your new best friend is up next – the STATUS register. This is the guy you will interface with most when you're using the chip (other than the TX/RX FIFOs – more on them later). When the IRQ pin is asserted (goes low), this is the register you check to see what has happened. It tells you the interrupts that are currently asserted, the data pipe for a payload that has been received for a receiver (RX_P_NO field), and if the TX FIFO is full for a transmitter (TX_FULL field). If you aren't using the IRQ pin, this is the register you poll to see if anything has happened. When an interrupt is asserted, you **MUST write a 1** to the proper bit in this register to clear it. The register description can be found on the following page in Figure 11.

The OBSERVE_TX register is a bit complicated to describe. The PLOS_CNT field describes how many packets have been lost since the last reset. This means that for all the channels that have or have had auto-ack enabled, if a packet has been retransmitted the maximum number of retries without success, this field increments. It is limited to 15 however. You reset this counter by making a write to RF_CH. I would presume they did it this way to tell you that you should change the RF channel because you are in an area that is not giving you good quality. The other field in this register is ARC_CNT, and this

is basically the number of resent packets for a given data payload. It resets itself when a new packet is transmitted.

07	STATUS				Status Register (In parallel to the SPI instruction word applied on the MOSI pin, the STATUS register is shifted serially out on the MISO pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Set high when new data arrives RX FIFO ¹⁵ . Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Set high when packet sent on TX. If AUTO_ACK is activated, this bit will be set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retries interrupt. Write 1 to clear bit. If MAX_RT is set it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.

Figure 11. STATUS register

08	OBSERVE_TX				Transmit observe register
	PLOS_CNT	7:4	0	R	Count lost packets. The counter is overflow protected to 15, and discontinue at max until reset. The counter is reset by writing to RF_CH. See page 14 and 17.
	ARC_CNT	3:0	0	R	Count resent packets. The counter is reset when transmission of a new packet starts. See page 14.

Figure 12. OBSERVE_TX register

Register 0x09 is the **CD register**. This register essentially monitors the air to see if there is anything broadcasting in your RF channel. Its value is 0 if nothing is detected and 1 if there is something going on. I can honestly say I've never used it, but it would probably be useful for situations where you have multiple 24L01s that are trying to **transmit at the same time** (mesh networks that aren't master/slave-based, for example). One thing to note with this function that is stated in the datasheet is that the 24L01 has to see an in-band signal for at least **128 uS** before it will set up the value in the CD register.

09	CD				
	Reserved	7:1	000000	R	
	CD	0	0	R	Carrier Detect. See page 17.

Figure 13. OBSERVE_TX register

Registers 0x0A to 0x0F are the receive addresses for each of the data pipes. There isn't really much I can say here since it's pretty self-explanatory

0A	RX_ADDR_P0	39:0	0xE7E7E7E7	R/W	Receive address data pipe 0. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
0B	RX_ADDR_P1	39:0	0xC2C2C2C2	R/W	Receive address data pipe 1. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
0C	RX_ADDR_P2	7:0	0xC3	R/W	Receive address data pipe 2. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0D	RX_ADDR_P3	7:0	0xC4	R/W	Receive address data pipe 3. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0E	RX_ADDR_P4	7:0	0xC5	R/W	Receive address data pipe 4. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]
0F	RX_ADDR_P5	7:0	0xC6	R/W	Receive address data pipe 5. Only LSB. MSBytes will be equal to RX_ADDR_P1[39:8]

Figure 14. RX_ADDR registers

The TX_ADDR register is next. Like the RX_ADDR registers, this is pretty common sense.

10	TX_ADDR	39:0	0xE7E7E7E7	R/W	Transmit address. Used for a PTX device only. (LSByte is written first) Set RX_ADDR_P0 equal to this address to handle automatic acknowledge if this is a PTX device with Enhanced ShockBurst™ enabled. See page 14.
----	---------	------	------------	-----	---

Figure 15. TX_ADDR register

Registers 0x11 through 0x16 are the data payload widths for each data pipe. Remember that you **MUST write** something other than 0 to any pipe that you want to enable. As you can see in the datasheet, all of the defaults are 0. Therefore, the link won't work for a particular pipe if you don't increase the size of the data payload for that pipe (assuming you also have the pipe enabled in the EN_RXADDR register). The register description for pipe 0 can be found in Figure 16 (all other pipes are identical).

11	RX_PW_P0				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P0	5:0	0	R/W	Number of bytes in RX payload in data pipe 0 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte ... 32 = 32 bytes

Figure 16. RX_PW_P0 register

The final register is the FIFO_STATUS register (all non-reserved bits are read-only). This guy has bits that tell you if the TX and the RX FIFOs are full or empty, which are very useful when you're sending data at high rates and want to max out the FIFO use. The TX_REUSE bit, as mentioned above, indicates when the REUSE_TX_PL instruction has been executed.

17	FIFO_STATUS				FIFO Status Register
	Reserved	7	0	R/W	Only '0' allowed
	TX_REUSE	6	0	R	Reuse last sent data packet if set high. The packet will be repeatedly resent as long as CE is high. TX_REUSE is set by the SPI instruction REUSE_TX_PL, and is reset by the SPI instructions W_TX_PAYLOAD or FLUSH TX
	TX_FULL	5	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.
	TX_EMPTY	4	1	R	TX FIFO empty flag. 1: TX FIFO empty. 0: Data in TX FIFO.
	Reserved	3:2	00	R/W	Only '00' allowed
	RX_FULL	1	0	R	RX FIFO full flag. 1: RX FIFO full. 0: Available locations in RX FIFO.
	RX_EMPTY	0	1	R	RX FIFO empty flag. 1: RX FIFO empty. 0: Data in RX FIFO.

Figure 17. FIFO_STATUS register

FIFO Info

There are FIFOs for both TX and RX modes (as mentioned in the “SPI Instruction Set” section). If you don't know what a FIFO is, it means “first in, first out,” just like a line at the grocery store works. Both of the FIFOs are three levels deep, meaning that they will hold the three newest packets that have been put into them. Essentially, if you receive three packets in RX mode and you don't read the payloads, the first (oldest) packet received is pushed out by the newest packet received. The same goes for the TX payload – if you load three packets and don't transmit them (by executing the aforementioned CE toggle), then the fourth packet will push out the first packet you loaded.

Data Packet Format

You might wonder why you would even care about what goes into the air between the 24L01s. For those who have low data rates, this doesn't really matter too much. However, if you're trying to get a certain actual on-air data rate, this is very important to know because the amount of overhead bytes that are in the packet will affect greatly your achievable data rate.

To see a drawing of the message format for both Shockburst and Enhanced Shockburst modes, see page 27 of the datasheet, or Figures 18 and 19, respectively, on the following page. In both modes, the preamble is sent first, and it is one byte long. It is composed of alternating zero and one bits, which is used to allow the receiver to know

that what it is hearing is the beginning of a packet and not just on-air noise. Also in both modes, the next bytes sent are the address bytes. This is set by you, the user, and is between three and five bytes long.



Figure 18. Shocburst data packet



Figure 19. Enhanced Shockburst data packet

The next thing sent is different between the two modes. In Enhanced Shockburst only, a flag word of nine bits is sent to indicate the message status concerning retransmissions. Only two bits are currently used (a count of resent packets), and the other seven are reserved for future use.

The last half of the packet that is sent is the same in both modes. The first of the fields to be sent in both modes is the payload data. The length of the payload is also set by the user and can be from one to 32 bytes long. The final part of the packet to be sent is the CRC, which is user-settable to zero, one, or two bytes.

Link Integrity and On-Air Data Rate Calculations

Now that you know the format of the data packet, we can make some calculations on how much real payload data you can squeeze through your link. Some people will want to push as much data as possible through the link with less worry about error checking and missed packets (similar to streaming audio over the internet). Other users will want to have error-free packets, but will have to live with a slower data rate.

Most users who are reading this will likely be of the persuasion of fewer errors. The trade-off here is that you need more overhead (longer address, more CRC bytes, and/or acknowledgements/retransmissions) to ensure that you aren't receiving garbage data packets at the receiver. The problem is that with more overhead, you get a lower payload data rate (assuming the size of the payload stays constant).

Taking this into consideration, here's an example data rate calculation that shows the maximum possible data rate you can squeeze out of the 24L01. Assume that you can constantly send packets (no delay in between), 100% of packets reach the receiver, the data rate is 2 Mbps, and Shockburst mode is being used. With this configuration, it is possible to have between four and eight bytes of overhead (one byte of preamble, three to five bytes of address, and zero to two bytes of CRC). Therefore, at the absolute best

utilization, you can send 32 data bytes with 4 bytes of overhead. This gives 88.9% utilization, or 1.78 Mbps of actual data rate (32 bytes of data/36 bytes in total packet).

You can use this type of calculation for any combination of overhead and data bytes to see what you can actually push through your link (at the theoretical maximum, that is). Also remember that the previous calculation is very ideal – in the real world you won't get 100% packet success rate and there will be a finite delay between packets.

Concluding Remarks

At this point, you should have enough background information to know what's going on in the future tutorials. You can always come back to this document when you aren't sure of something, and if you find something that should be here that isn't, feel free to contact me at brennen@diyembedded.com.

Disclaimer: The author provides no guarantees, warranties, or promises, implied or otherwise. By using the software in this tutorial, you agree to indemnify the author of any damages incurred by using it. You also agree to indemnify the author against any personal injury that may come about using this tutorial. The plain English version – I'm doing you a favor by trying to help you out, so take some responsibility and don't sue me!