The Nordic nRF2401 Single Chip Data Transceiver: High Speed, Short Range Data Communication At An Extremely Low Cost

By John A. Hansen, W2FS
Department of Computer and Information Sciences
State University of New York at Fredonia

1. Introduction

The Nordic nRF2401 is a single chip data transceiver that works in the 2.4 GHz ISM band. It provides data speeds of up to 1 Mbit/sec. The frequency can be set in software anywhere from 2400 MHz to 2524 MHz. So, for example, it is possible to place the device at 2414 MHz where the ARRL band plan has allocated a channel for high speed data communication. The chip is available in single quantities for $3.50[1] and can also be purchased as an easy to interface module (including antenna) for $22.[2]

My interest in this transceiver stems from a desire to create a “virtual” serial cable. I have a number of radios in various automobiles that are all programmable via their serial ports. Using a PC and software to program memory channels and configuration information in these radio is vastly easier than entering the data via the radio’s front panel. However, programming with a PC means that I either have to bring my radio in the house to where the PC is or I have to take a laptop PC to the car where the radio is. As a result, I almost never update the programming of any of these radios. If it were possible to create a wireless link that would act just like a serial cable between my computer’s USB port and the radio’s serial port, editing the radio’s frequency and configuration would be a snap. The range involved would be quite short… perhaps as little as 30 feet, so a low power solution like the Nordic nRF2401 looks as if it might be just the thing for this job.

At this writing, this project is not complete. I do have data flowing back and forth between two serial ports on my computer at 9600 baud using a pair of Nordic chips, but a number of issues still need to be resolved. The purpose of this paper is to provide a simple guide to using the nRF2401 and to provide some sample programming code to show how to configure and use the device.

2. Hardware Configuration

To simplify experimentation, I purchased 2 WRL-00151 transceiver modules (also known as TRW-24G) from Spark Fun electronics. These modules require 3 volts to operate (drawing a maximum of 18 ma) and have an integrated antenna. The only external connections that are required (other than power) are five control lines that are connected to a microcontroller. The microcontroller clocks data in and out of the transceiver using a synchronous serial data transmission protocol. Figure 1 shows one of the modules mounted on a piece of perfboard with a 16F628A PIC microcontroller, a MAX232 serial driver chip (to interface with the PC serial port) and two voltage regulators (an LM317 to provide 3 volts for the transceiver and the PIC and a 78L05 to provide 5 volts for the MAX232). The two jumpers on the board allow me to switch the serial connector from DCE to DTE and vice versa so I can move the unit between a PC and a radio’s serial port without using a null modem adapter. I run the system from a

[1] See www.semiconductorstore.com/pages/asp/item.asp?ItemNumber=NRF2401AG-REEL
[2] See www.sparkfun.com/commerce/product_info.php?products_id=151

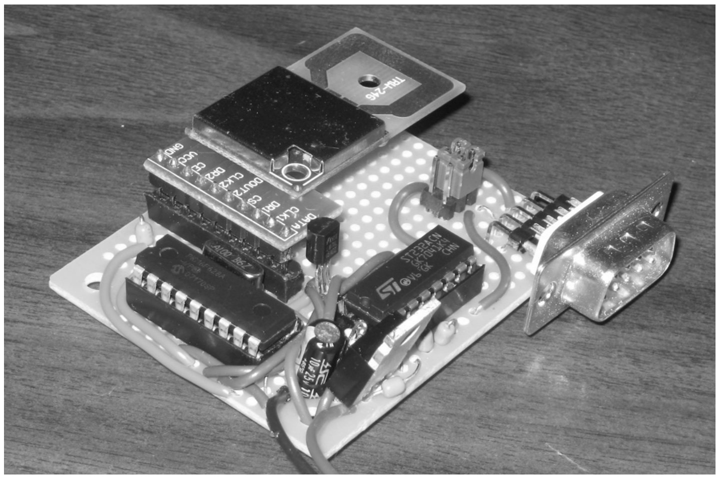
Figure 1

9 volt battery. I created to of these units for initial experimentation transmitting data between two serial ports running on the same computer.

The connections between the transceiver and the microcontroller consist of five lines. Two of the lines (CS and CE) determine the mode that the transceiver is in. These are as follows:

Mode	CE	CS
Active (RF on)	high	low
Configuration	low	high
Stand by (RF off)	low	low

Another line, labeled DR1, allows the transceiver to tell the microcontroller when data has been received and is ready to be clocked out of the radio. I connected this line to the external interrupt line of the 16F628A so that I would have the option of using this interrupt to trigger the processing of received data. Two other lines (clock and data) are used to communicate received or transmitted data from the transceiver to the microcontroller. The clock is always controlled by the microcontroller, thus it is always set as an output. The data line must be set as an input when data is being received and an output when data is being transmitted.

I connected all five lines to PORTB of the 16F628A. The pin assignments are as follows:

Port B Pin	I/O	Description
0	In	DR1: This is the external interrupt on this PIC.
1	In	RXD: Receives data from PC serial port (connects to PIC USART.)
2	Out	TXD: Sends data to PC serial port (connects to PIC USART.)
3	Out	CE: Sets transceiver mode
4	---	data: Sends/Receives data to/from transceiver.
5	Out	clock: Clocks data to/from transceiver.
6	Out	CS: Sets transceiver mode

When the PIC is sending data to the transceiver the value in the TRISB registers should be: 0b00000011. When it's receiving data from the transceiver the value should be: 0b00010011.

3. Shockburst Mode

The nRF2401 is designed to transmit data at either 250 kbits/sec or 1 Mbit/sec. The data rate is selected during the transceiver configuration process (see below). However, performance is supposed to improve by 10 dB when using the lower data rate so I decided to begin my experimentation at the lower speed. The transceiver has two modes of operation: "Direct" and "Shockburst" (where do they come up with these names?!). In direct mode you must feed the data to the transceiver at the same rate it will be transmitted over the air (that is, either 250 kbits/sec or 1 Mbit/sec). Any error checking that you might want to have done is your responsibility. You also must take care of adding any flags or other preamble bytes that you might want to employ. Using direct mode, for example, it should be possible to send standard AX.25 packets over this link. You could think of the device as being similar to a Bell 202 modem chip where the chip takes care of transmitting the data, but you are responsible for everything else including timing the bits and constructing the frames.

Shockburst mode causes the transceiver to take responsibility for many of these details itself. To send data, for example, you clock the data into the chip at whatever speed you want (within limits) and the chip gathers the data into packets and calculates a 1 or 2 byte CRC and appends it to the end. It also generates a "preamble" which can be thought of as the equivalent of "flags" used in AX.25. Data can be sent in 1 to approximately 29 byte packets. On the receive side the transceiver checks the CRC to ensure that the data was received without error, checks the address to see if the data is intended for this station, and then signals the microcontroller that data has been received and is ready for processing. Nordic claims that a key advantage of the "shockburst" mode is that the RF section of the transceiver can be powered down while processing of the data is being done, thereby consuming relatively little power. However, the advantages that I found most appealing were its ability to construct/error check the packets and send them at a data rate that was independent of the rate that the data was arriving at the chip. As a result, I have thus far limited my experimentation with the device to shockburst mode.

One other interesting feature of the nRF2401 is that it is capable of receiving on two separate channels at the same time through a single antenna (the second channel is 8 MHz higher than the first). The data from the second receiver is clocked into the microcontroller via a separate pair of data/clock lines. As yet, I have not had an opportunity to experiment with this mode.

4. Configuring the Transceiver

When the transceiver is first powered up it must be configured before data can be either sent or received. In shockburst mode with one receiver, it is necessary to send 14 bytes to configure the device.

```
bit  7 ______bit 0    bit 7_____bit 0   _bit 7___bit 0   bit 7____bit 0         bit 7______bit 0
     |  Byte 13  |    | Byte 12    |    |  Byte 11   |   | Byte 10   | . . . .  |   Byte 0      |
     |__________|     |___________|     |____________|   | _________ |          | ____________|
```

The data is transmitted left to right. That is, the most significant byte (Byte 14) is sent first and within this byte the most significant bit (bit 7) is sent first. Byte 0, bit 0 is sent last. Byte 0, bit 0 determines whether the device is in transmit or receive mode (it is not full duplex). Thus you must reconfigure the device every time you switch between sending and receiving. According to the Nordic documentation, it is not necessary to rewrite the entire configuration to do this. However many bits you send while in configuration mode is the number that will be written into the configuration registers… the other bits and the other registers will be unchanged. Thus you should be able to put the device into configure mode and then send a single bit to switch between transmit and receive. As yet I have not had any success doing this. So far I've been rewriting the first 14 bytes of the configuration each time. However, because I'm writing data from the microcontroller to the transceiver at a very high speed, it doesn't take long to do this.

The configuration bits are documented in detail in the Nordic datasheet. Here is an overview of what each byte does:

Byte 0:	Bit 0 controls transmit/receive, the other 7 bits select the operating frequency.
Byte 1:	Bits 0 and 1 set the output power, bits 2-4 specify the frequency of the crystal attached to the transceiver (leave it at 16 MHz), bit 5 specifies the data rate, bit 6 specifies whether it will be used in direct or shockburst mode, and bit 7 enables dual channel receive
Byte 2:	Bit 0 turns CRC checking on or off, bit 1 specifies an 8 bit or 16 bit CRC (leave it on 16), the other 6 bits specify how many bits will be used for the address of the transceiver (in shockburst mode).
Bytes 3-7:	Contains the address of the transceiver for the first receive channel
Bytes 8-12:	Contains the address of the transceiver for the second receive channel
Byte 13:	Specifies the length of the data blocks being received (just the payload, not including the address, CRC, etc.) for the first receiver
Byte 14:	Specifies the length of the data blocks being received (just the payload, not including the address, CRC, etc.) for the second receiver (you need not send this byte if you are only using one receiver

I set the operating frequency to 2402 MHz and the power output to 0 dBm (that's 1 mw… the maximum possible). I used the slower data transmission speed, selected shockburst mode and disabled the second receiver. I specified only 8 address bits (the minimum) and a two byte CRC. The folks at Spark Fun have found a lot of errors slipped through when using the one byte CRC. This should not be too surprising given that there is a 1 in 256 chance of getting the "right" value even if the CRC was randomly chosen.

Testing at Spark Fun found some interesting results related to the length of the data payload in each packet. They tested payloads that were 4 bytes long (making a 7 byte packet) and payloads that were 29 bytes long (making a 32 byte packet). The tests were performed with the two transceivers only 4 inches from each other so signal strength should not have been an issue. What they found was that when they used 29 byte payloads, approximately 40% were lost (CRC did not check). When they used 4 byte payloads, only 10% were lost. Both of these values are consistent with a failure rate of 1.5% for each byte in the packet. If I apply that figure to a 1 byte payload (making a 4 byte packet with the address and CRC) I should get about a 6% failure rate. In fact when I tried this experiment the failure rate was much lower than this…literally hundreds and hundreds of characters were transmitted without any errors. It seemed to me that there must be something else at work here besides a constant probability of error for each byte. As a result, it seemed to me that my experiments were likely to be much more successful initially if I used a one byte payload. A couple of other advantages of this were that since all payloads must be a fixed length, if I used a one byte payload I wouldn't have to worry about padding payloads where there were only a fraction of the total needed bytes available. In addition when people usually experience a serial data link they expect to have each character transmitted as they type it, so a single byte payload would more closely conform to user's expectations.

Here is the C code for the function that sets up the configuration of the chip:

```
set_tris_b(0b00000011);              // sets the TRIS register.  The lines are connected to clock,
                                      // data, CE, CS, and DR1 must all be outputs
output_low(CE);
output_high(CS);                     // puts the device in configuration mode
for (i = 0; i<14; i++){              // for each of the 13 configuration bytes
        for (j=0; j<8; j++){         // for each of the bits in the byte
                output_low(data);    // if configuration bit is a 1 set RXdata high, else low
                if (conf[i] & 0b10000000)
                        output_high(data);
                output_high(clock);  // we need to raise and lower the clock line in order to send the data
                delay_cycles(1);     //  The PIC is running at 10 MHz.  Without a delay here we'd be
                output_low(clock);   //  sending the data too fast
                conf[i] = conf[i]<<1; //  Move to next bit
        }
}
output_low(CS);                      //  This locks the configuration data into the transceiver's registers
output_high(CE);                     //  Make the transceiver active (turn on the RF section)
```

The idea here is that the leftmost bit of the most significant byte is sent first. It is stripped off by the line of code: **if (conf[i] & 0b10000000) output_high(data**); Then all the bits are shifted to the left one place (**conf[i] = conf[i]<<1;**) so that the next bit can be sent.

One note…. You may find it odd that the following code was used:

```
output_low(data);
if (conf[i] & 0b10000000)  output_high(data);
```

Instead of:

```
if (conf[i] & 0b10000000)  output_high(data);
        else output_low(data);
```

The reason the former is used is that I've found it always results in a slightly smaller hex file (at least using the CCS C compiler).

5. Transmitting Data.

Transmitting data involves taking characters as they come in the serial port (from the PC) and sending them out to the transceiver. I used an 80 character ring buffer (called **buf[]**) to hold the characters as they come in and wait for transmission. The variable **inpointer** keeps track of where in the buffer the next character should be inserted and the variable **backlog** keeps track of the number of characters that have yet to be transmitted. The code for receiving the characters from PC looks like this:

```
void addchar(){
        buf[inpointer] = getc();          //get character from USART
        inpointer++;                      //increment pointer
        if (inpointer >79) inpointer = 0; //wrap if end of buffer
        backlog++;                        //one more byte in the buffer
}
```

A third variable, **outpointer**, keeps track of the location in the buffer that the next character to be transmitted is. If **backlog** is greater than zero, it means characters are available to be transmitted. Each time a character is transmitted, the **backlog** variable must be decremented. The code to transmit looks like this:

```
config(XMIT);  //switch to transmit mode
while (backlog > 0){
        output_high(CE);                      //  get ready to put in address and data
        if (bit_test(pir1,5)) addchar();      //  add bytes to buffer from serial port if necessary
        sendbyte(address);                    //  load the address byte into the transceiver
        sendbyte(buf[outpointer]);            //  load the data byte
        output_low(CE);                       //  this actually causes the transceiver to send
        TMR0=0;                               //  you have to wait while it's doing the sending
        while(TMR0 <10) {                     //  we use timer0 to measure this time.
                if (bit_test(pir1,5)) addchar();   //add bytes to buffer from serial port if necessary
        }
        outpointer++;                         //increment sending pointer
        if (outpointer > 79) outpointer = 0;  //wrap if at end of buffer
        backlog--;                            //decrement characters to be sent
}//end of while
config(RCV);  //switch back to receive mode
```

Calls to the config function simply change the transceiver from receive to transmit and back again. If there are multiple characters awaiting transmission, the code does not switch back and forth between transmit and receive in between characters. Raising the CE line makes the transceiver active. Note: the CE line must be high while characters are being loaded into the radio even though the RF section is not being used at this point. This is because lowering the CE line is the signal to the transceiver to actually send the data. The fourth line of code checks to see if another character has come in from the serial port (the PC). If so, the addchar function is called to add it to the buffer. The address and data bytes are then clocked in using the sendbyte function. Sendbyte takes bits off the left end of the byte to be loaded and clocks them into the transceiver. This is exactly the same way the configuration bytes are loaded in the config function.

At this point a pause is necessary because we can't load the address and data registers with new characters until we're sure that the previous byte has been transmitted. However, we can't just write a line of code that causes a delay here because additional data may still be coming in via the serial port and we need to put that data in the buffer. So timer0 is used to cause a 1 millisecond delay and during this period we service the serial port if necessary. This is actually more time than should be required here; further experimentation is needed to determine the smallest possible value.

After all the outstanding characters have been sent, the config function is called again to place the transceiver back in receive mode.

6. Receiving Data

Receipt of a byte of packet of data is triggered by the DR1 line (hooked to PORTB, pin 0) going high. At that point the data is ready to be transferred to the PC via the microcontroller's serial port. Here is the code that causes this to happen:

```
void receive(){
int j, indata;
        set_tris_b(0b00010011);           //  data line must be an input
        indata = 0;                       //  initialize the variable indata
        for (j = 0; j<8; j++){            //  for each of 8 bits
                indata = indata <<1;      //  shift to next bit.  Note the first bit received will be MSB.
                if (input(data)) indata++; // the default is a zero, if the data line is high, make it a 1.
                output_high(clock);       // have the transceiver move to the next bit
                delay_cycles(1);
                output_low(clock);
        }
        putc(indata);
}
```

The data line on PORTB must be changed from an output to an input in order to be able to read the incoming data. The CCS C compiler statement set_tris_b accomplishes this. Then for each of the 8 bits the data line is read. Having the clock line rise and then fall causes the transceiver to move to the next bit in the byte. This process is essentially the reverse of the process that is used to transmit data.

6. Putting It All Together

The main part of the program must accomplish three things. First, it must make sure that bytes that come in via the serial port from the PC are put in the buffer. Second, when the DR1 line goes high it must receive the incoming characters from the transceiver. Third, when there are characters in the buffer to send it must call the transmit function. The first two of these functions could have been accomplished using interrupts, but since this program really doesn't do anything else, there didn't seem to be much point to it. As complexity grows (implementing error checking or forward error correction, for example) it may be useful to switch to an interrupt driven structure. For now the main program loop looks like this:

```
while(1){
        if (bit_test(pir1,5)) addchar();          // add bytes to buffer from serial port
        if (input(DR1)) receive();                //  if there are bytes to be received, get them
        if (backlog> 0)   transmit();             //  if there are bytes to send (via radio) send them
 }
```

7. Experimentation

Unfortunately at this writing I'm just getting started on the experimentation portion of this project. Currently I have data flowing back and forth between two serial ports on my PC using this virtual serial cable at 9600 baud. It should be possible to revise the code to shorten the delays to achieve higher data rates. However, since it is possible to reprogram most radios using a serial port speed of 9600 baud, increasing the speed is not my highest priority.

The biggest problem that I have currently is that the range of the device in my environment is nowhere near that which is advertised. The folks at Spark Fun used these modules in the open air for a range test and came up with a figure of 686 feet. They did not get anything like perfect copy at that distance, but they were able to push data through the pipe. At distances of 20 feet indoors I'm seeing virtually perfect copy. As I push this out to 30 to 40 feet, however, the number of characters received falls off. I've not had enough time to experiment carefully with this, but my impression is that copy declines very rapidly… that is, there is a point where nearly perfect copy falls off to nearly no copy. A significant part of the range issue is almost certainly related to the fact that I'm doing the testing indoors. I've also got a number of other 2.4 GHz devices in the house and it may be that changing frequency will help some. Since I'm planning to run the indoor version of this unit off of a USB cable, it may also help to mount the transceiver on a window and run a USB cable from there to the computer.

The Nordic nRF2410 is really quite an amazing device. Just a few years ago the idea that data could be sent even over short distances at these high speeds using a $3.50 device would have been considered absurd.