insideGadgets

I'm learning about electronics bit by bit, making projects and tearing things apart

« Standalone Temperature/Voltage Logger updated to v1.1
ATtiny Programmer Adapter v1.1 PCBs arrived »

Using the nRF24L01 wireless module

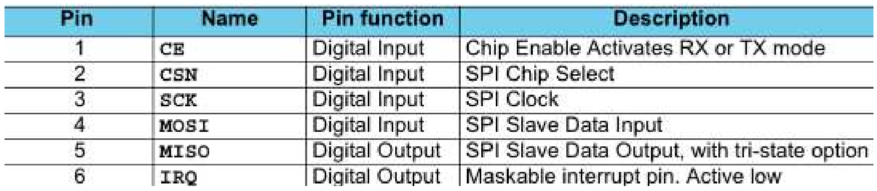
Aug 22nd, 2012 by Alex

I decided it was time for me to play around with wireless communication as I recently purchased a wireless alarm system. I was able to pick up two nRF24L01 wireless modules very cheaply at about $2 each from Ebay.

Some features of this chip are:

- 126 channels
- Up to 32 bytes payload data
- Up to 2Mbps
- About 12mA current consumption when receiving or transmitting

I found there is a RF24 library for the Arduino and a similar library for the AVR. However after looking through nRF24L01 datasheet it appears that the AVR library version appeared a tiny bit incomplete so I’ve made some changes to the code.



Pin	Name	Pin function	Description
1	CE	Digital Input	Chip Enable Activates RX or TX mode
2	CSN	Digital Input	SPI Chip Select
3	SCK	Digital Input	SPI Clock
4	MOSI	Digital Input	SPI Slave Data Input
5	MISO	Digital Output	SPI Slave Data Output, with tri-state option
6	IRQ	Digital Output	Maskable interrupt pin. Active low

The nRF24L01 modules uses 6 data pins but you can get by with only using 5. There are 4 for the SPI, 1 for the chip enable (CE) which sets the nRF24L01 in listen or transmit mode and an interrupt pin (IRQ) which can alert us when a packet arrives, packet is sent or maximum retries reached. I don’t use the interrupt pin in my example.

In my example, I have the transmitting code running on the ATtiny and the receiving code running on the Arduino so I could easily print the results. We run the nRF24L01 from the 3.3V pin on the Arduino.

ATtiny code

```
setup();
mirf_init();
_delay_ms(50);
mirf_config();
uint8_t buffer[5] = {14, 123, 63, 23, 54};
uint8_t buffersize = 5;

while(1) {
  mirf_send(buffer,buffersize);
  _delay_ms(1000);

  // If maximum retries were reached, reset MAX_RT
  if (mirf_max_rt_reached()) {
    mirf_config_register(STATUS, 1<<MAX_RT);
  }
}
```

Without getting into the specifics just yet, we run the initialise function, delay a few milliseconds for the nRF24L01 chip to start up, run the configuration function to set the channel, payload, addresses, etc and then we have the data to send in an array with the array size (buffer).

After the initialisations are complete, we just keep re-sending the data and pause for 1 second. If the transmission we sent didn’t make it after a few retries, we reset the maximum retries otherwise the nRF24L01 won’t try transmitting any more data.

Arduino code

```
void loop() {
  uint8_t buffer[5] = {0, 0, 0, 0, 0};
  uint8_t buffersize = 5;
  mirf_CE_hi; // Start listening

  while(1) {
    while (!mirf_data_ready()) {
      _delay_ms(50);
      Serial.print(".");
    }
    mirf_get_data(buffer);

    for (int x = 0; x < 5; x++) {
      Serial.println(buffer[x], DEC);
      buffer[x] = 0;
    }
  }
}
```

I've skipped the initialising code as it's the same as in the ATtiny. All we do is check if there is data that has arrived, if not then keep waiting. If there is data waiting for us, then we read the data and print out the results – it all sounds very simple.

Now it's time to view the code in detail, I'll be discussing the AVR version.

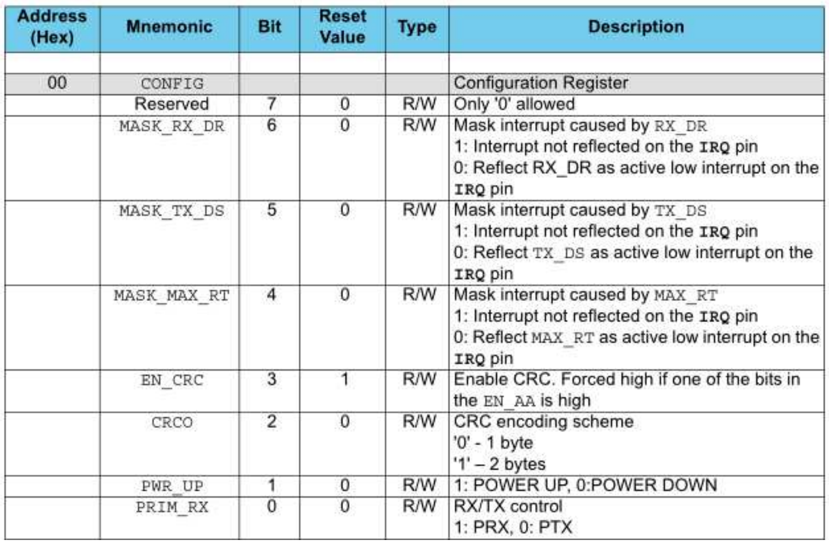
```
void setup(void) {
  spi_init(); // Initialise SPI
}
```

We just initialise the SPI.

```
// Read or write 1 byte
uint8_t spi_transfer(uint8_t data)
// Write data using SPI
void spi_write_data(uint8_t * dataout, uint8_t len)
// Read data using SPI
void spi_read_data(uint8_t * datain, uint8_t len)
```

We use the above SPI functions, one that transfers 1 byte and 2 others that write multiple bytes and read multiple bytes.

By looking at the register map we can configure the nRF24L01.



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	RX/TX control 1: PRX, 0: PTX

Defines / Settings

```
// Mirf settings
#define RXMODE 1
#define TXMODE 0
#define mirf_CH          2
#define mirf_PAYLOAD     5
#define mirf_CONFIG      ( (1<<EN_CRC) | (0<<CRCO) )
#define RADDR             (byte *)"clnt2"
#define TADDR             (byte *)"clnt1"

// Flag which denotes transmitting or receiving mode
volatile uint8_t PMODE;

// Defines for setting the MiRF registers for transmitting or receiving mode
#define TX_POWERUP mirf_config_register(CONFIG, mirf_CONFIG | ( (1<<PWR_UP) | (0<<PRI
#define RX_POWERUP mirf_config_register(CONFIG, mirf_CONFIG | ( (1<<PWR_UP) | (1<<PRI
```

We setup RXMODE and TXMODE so we can determine which mode we are in (but it’s not used in my example), define the channel (mirc_CH), the number of bytes to transmit (mirf_PAYLOAD), the configuration to apply and the receiving and transmitting addresses (RADDR and TADDR, 5 bytes maximum). Also we have 2 power up modes we can call to switch between transmit (TX_POWERUP) and receive (RX_POWERUP) modes.

```
// Pin definitions for chip select and chip enabled of the MiRF module
#define CE  PB3
#define CSN PB4

// Definitions for selecting and enabling MiRF module
#define mirf_CSN_hi     PORTB |=  (1<<CSN);
#define mirf_CSN_lo     PORTB &= ~(1<<CSN);
#define mirf_CE_hi      PORTB |=  (1<<CE);
#define mirf_CE_lo      PORTB &= ~(1<<CE);
```

We define the CE and CSN pins plus some defines to make switching between a high and low output easier.

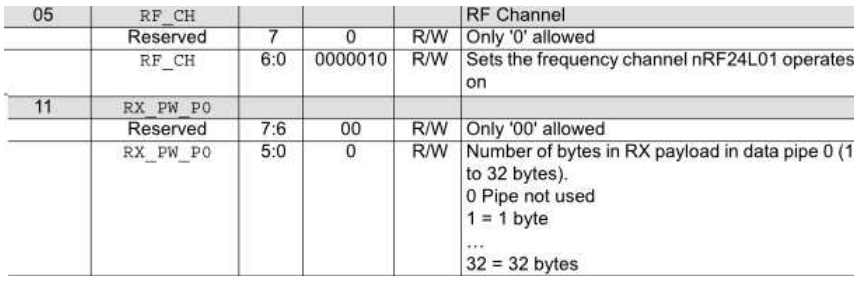
```
void mirf_init(void) {
  DDRB |= ((1<<CSN)|(1<<CE));
  mirf_CE_lo;
  mirf_CSN_hi;
}
```

We set up our outputs and turn CE to low and CSN to high. When CE is low it means that the chip isn’t listening for or sending data.

Configuration code

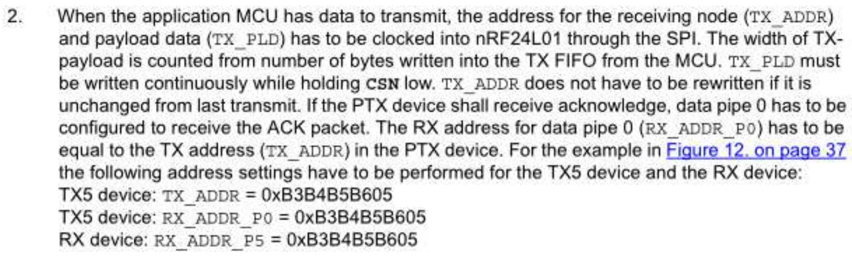
```
void mirf_config(void) {
  // Set RF channel
  mirf_config_register(RF_CH, mirf_CH);

  // Set length of incoming payload
  mirf_config_register(RX_PW_P0, mirf_PAYLOAD);

  // Set RADDR and TADDR as the transmit address since we also enable auto acknowledg
  mirf_write_register(RX_ADDR_P0, TADDR, 5);
  mirf_write_register(TX_ADDR, TADDR, 5);

  // Enable RX_ADDR_P0 address matching
  mirf_config_register(EN_RXADDR, 1<<ERX_P0);

  // Start transmitter
  PMODE = TXMODE; // Start in transmitting mode
  TX_POWERUP;     // Power up in transmitting mode
}
```

Now we write to the chips registers by using the mirf_config_register function for single byte operations and the mirf_write_register function for multi-byte writes.

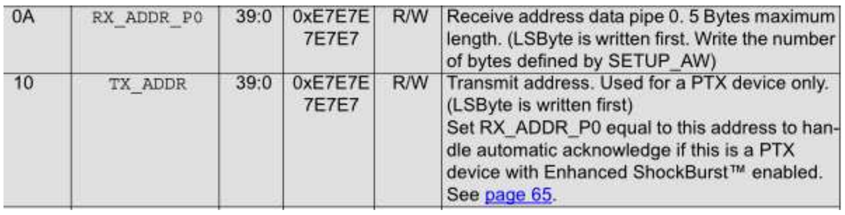


05	RF_CH				RF Channel
	Reserved	7	0	R/W	Only '0' allowed
	RF_CH	6:0	0000010	R/W	Sets the frequency channel nRF24L01 operates on
11	RX_PW_P0				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P0	5:0	0	R/W	Number of bytes in RX payload in data pipe 0 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes

We set the RF channel and set the incoming payload length (not really needed for the transmitter).

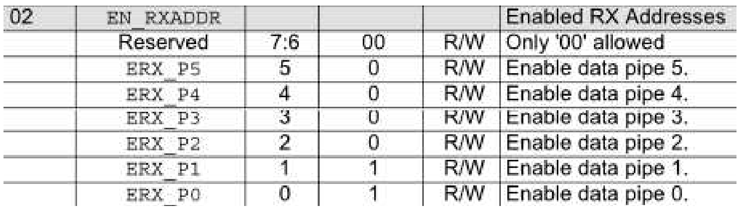


2. When the application MCU has data to transmit, the address for the receiving node (TX_ADDR) and payload data (TX_PLD) has to be clocked into nRF24L01 through the SPI. The width of TX-payload is counted from number of bytes written into the TX FIFO from the MCU. TX_PLD must be written continuously while holding CSN low. TX_ADDR does not have to be rewritten if it is unchanged from last transmit. If the PTX device shall receive acknowledge, data pipe 0 has to be configured to receive the ACK packet. The RX address for data pipe 0 (RX_ADDR_P0) has to be equal to the TX address (TX_ADDR) in the PTX device. For the example in Figure 12. on page 37 the following address settings have to be performed for the TX5 device and the RX device:
TX5 device: TX_ADDR = 0xB3B4B5B605
TX5 device: RX_ADDR_P0 = 0xB3B4B5B605
RX device: RX_ADDR_P5 = 0xB3B4B5B605



0A	RX_ADDR_P0	39:0	0xE7E7E 7E7E7	R/W	Receive address data pipe 0. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
10	TX_ADDR	39:0	0xE7E7E 7E7E7	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 65.

On page 65 of the PDF it shows what we need to do when receiving and transmitting. Whenever we transmit data, we can set the receiver to auto acknowledge the payload we've sent (it's set on by default) and in order to receive the acknowledgement we need to set the receive address to be the same as the address we transmit to as we'll receive a packet back.



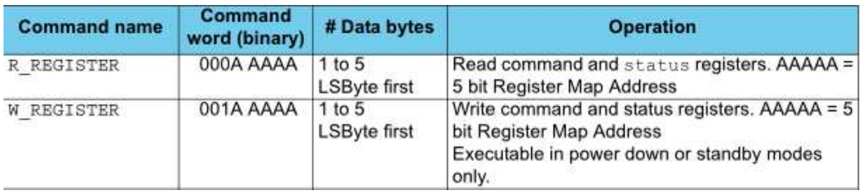
02	EN_RXADDR				Enabled RX Addresses
	Reserved	7:6	00	R/W	Only '00' allowed
	ERX_P5	5	0	R/W	Enable data pipe 5.
	ERX_P4	4	0	R/W	Enable data pipe 4.
	ERX_P3	3	0	R/W	Enable data pipe 3.
	ERX_P2	2	0	R/W	Enable data pipe 2.
	ERX_P1	1	1	R/W	Enable data pipe 1.
	ERX_P0	0	1	R/W	Enable data pipe 0.

We've set the receive address P0 (pipe 0) and now we just need enable that pipe by setting EN_RXADDR to 1 which we do next. As you can see there are multiple pipes, so it's possible to have one receiver listen for 6 different addresses.

Lastly we set the chip into transmit mode and we don't set CE to high just yet.

Config Register / Read & Write Register functions

```
// Write one byte into the MiRF register
void mirf_config_register(uint8_t reg, uint8_t value) {
  mirf_CSN_lo;
  spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
  spi_transfer(value);
  mirf_CSN_hi;
}

// Reads an array of bytes from the MiRF register
void mirf_read_register(uint8_t reg, uint8_t * value, uint8_t len) {
mirf_CSN_lo;
spi_transfer(R_REGISTER | (REGISTER_MASK & reg));
spi_read_data(value,len);
mirf_CSN_hi;
}

// Writes an array of bytes into the MiRF register
void mirf_write_register(uint8_t reg, uint8_t * value, uint8_t len) {
  mirf_CSN_lo;
  spi_transfer(W_REGISTER | (REGISTER_MASK & reg));
  spi_write_data(value,len);
  mirf_CSN_hi;
}
```

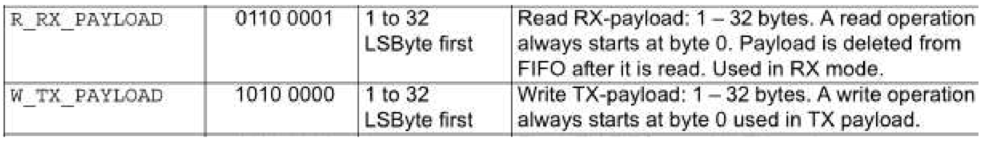


Command name	Command word (binary)	# Data bytes	Operation
R_REGISTER	000A AAAA	1 to 5 LSByte first	Read command and status registers. AAAAA = 5 bit Register Map Address
W_REGISTER	001A AAAA	1 to 5 LSByte first	Write command and status registers. AAAAA = 5 bit Register Map Address Executable in power down or standby modes only.

The config register and read & write register functions are quite easy as it just uses SPI so we just send the above commands and replace AAAAA with the register map address we wish to access. For example, to access the RF_CH register, you would send the address 00000101 and then the byte to write.

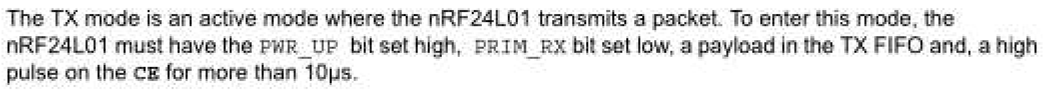
Transmitting data

```
void mirf_send(uint8_t * value, uint8_t len) {
  PMODE = TXMODE;                  // Set to transmitter mode
  TX_POWERUP;                      // Power up

  mirf_CSN_lo;                     // Pull down chip select
  spi_transfer( FLUSH_TX );      // Write cmd to flush tx fifo
  mirf_CSN_hi;                     // Pull up chip select

  mirf_CSN_lo;                     // Pull down chip select
  spi_transfer( W_TX_PAYLOAD ); // Write cmd to write payload
  spi_write_data(value,len);   // Write payload
  mirf_CSN_hi;                     // Pull up chip select

  mirf_CE_hi;                      // Start transmission
  _delay_us(15);
  mirf_CE_lo;
}
```



R_RX_PAYLOAD	0110 0001	1 to 32 LSByte first	Read RX-payload: 1 – 32 bytes. A read operation always starts at byte 0. Payload is deleted from FIFO after it is read. Used in RX mode.
W_TX_PAYLOAD	1010 0000	1 to 32 LSByte first	Write TX-payload: 1 – 32 bytes. A write operation always starts at byte 0 used in TX payload.

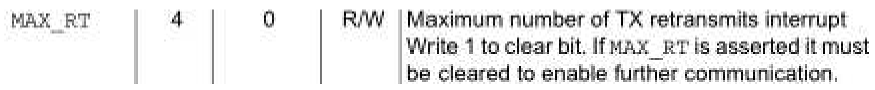
First thing we do is change to transmitter mode and flush any data that may be in the TX buffer from the last time we sent data (this is useful because we may have reached the maximum retry limit which means the data in the TX buffer won't get cleared).



The TX mode is an active mode where the nRF24L01 transmits a packet. To enter this mode, the nRF24L01 must have the PWR_UP bit set high, PRIM_RX bit set low, a payload in the TX FIFO and, a high pulse on the CE for more than 10µs.

Next we load our payload and then set the CE pin high for 15us and we are all done, the chip will do the rest. The minimum we need to keep the CE pin high for is 10us.

```
// Checks if MAX_RT has been reached
uint8_t mirf_max_rt_reached(void) {
  mirf_CSN_lo; // Pull down chip select
  spi_transfer(R_REGISTER | (REGISTER_MASK & STATUS));
  uint8_t status = spi_transfer(NOP); // Read status register
  mirf_CSN_hi; // Pull up chip select
  return status & (1<<MAX_RT);
}
```



MAX_RT	4	0	R/W	Maximum number of TX retransmits interrupt Write 1 to clear bit. If MAX_RT is asserted it must be cleared to enable further communication.

If we send some data and we don't receive any acknowledgements back, the MAX_RT is set and we won't send any more data which is good in a way so we know what's happened but bad because if the receiver comes back online we try to re-communicate to it. This function will check if MAX_RT is set and early in our code if it was set, we clear it so we can keep trying to send data.

Receiving data

```
uint8_t mirf_data_ready(void) {
  mirf_CSN_lo;                                // Pull down chip select
  spi_transfer(R_REGISTER | (REGISTER_MASK & STATUS));
  uint8_t status = spi_transfer(0x00);        // Read status register
  mirf_CSN_hi;                                // Pull up chip select
  return status & (1<<RX_DR);
}
```

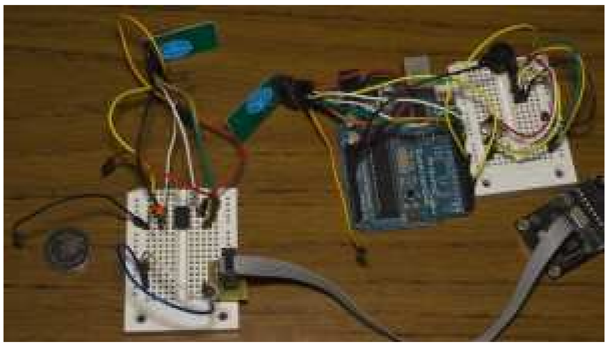
The function above just checks the register to see if there is any data waiting to be read. If there is then we return 1.

```
// Reads mirf_PAYLOAD bytes into data array
void mirf_get_data(uint8_t * data) {
  mirf_CSN_lo;                               // Pull down chip select
  spi_transfer( R_RX_PAYLOAD );            // Send cmd to read rx payload
  spi_read_data(data, mirf_PAYLOAD); // Read payload
  mirf_CSN_hi;                               // Pull up chip select
  mirf_config_register(STATUS,(1<<RX_DR));   // Reset status register
}
```

When there is data waiting to be read, we can request to read the payload, it will automatically remove the payload from the RX and then we clear the RX_DR bit which alerted us that there was data waiting for us.

Results and Conclusion

Download the code here: nRF24L01_TX-RX_Test



I have the ATtiny running from a 3V coin cell and the Arduino with a small speaker to beep once some data is received. When transmitting the packet on the ATtiny's side, it was performed so fast that my cheap multimeter didn't pick up any current or voltage changes to the coin cell battery I was using. On the Arduino reading side, as it's set to always be listening it was taking up about 12mA.

Here are the results I received from the serial monitor on the Arduino. As you can see the packet is being received successfully.

```
..........................................................................
123
63
23
```

```
54
........14
123
63
23
54
........14
123
63
23
54
```

We have shown the basic configuration of a one way communication from the ATtiny to the Arduino which you should be able to use for a two way communication. I would recommend reading the datasheet as it provides much more information on how you should interact with the chip as well as flowcharts, examples of the interaction between 2 wireless modules and some tips.