nRFLoader

An Over-the-Air bootloader for PIC microcontrollers using nRF24L01 radio

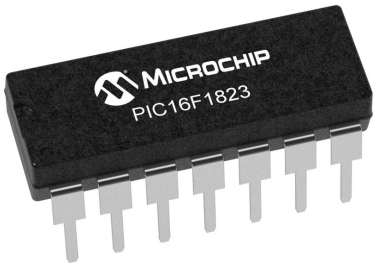
Introduction

This paper introduces a bootloader (yet another one) for the PIC16F182x family of devices. What makes this code different from other minimalist bootloaders, is that the boot loader is designed to be used in conjunction with the popular NRF24L01 radio modules, and thus providing an *Over The Air (OTA)* ability to re-program the firmware without the need for any further physical interaction with the deployed client devices.

As the name implies, a *boot loader*, is a piece of code that executes during the booting process of a computer (or micro-processor) and provides functionality to *load* new software (firmware) images to the device. One of the most popular examples of this would be the *Arduino Uno* development board. Without the *Arduino boot loader* running, the ATMEGA328 micro-processor would have no ability to communicate with the *Arduino IDE,* nor to use the standard procedure of re-flashing the code space over the serial connection.

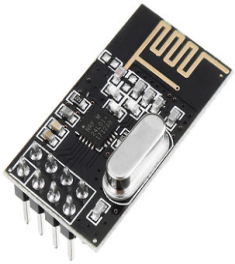
Starting Point

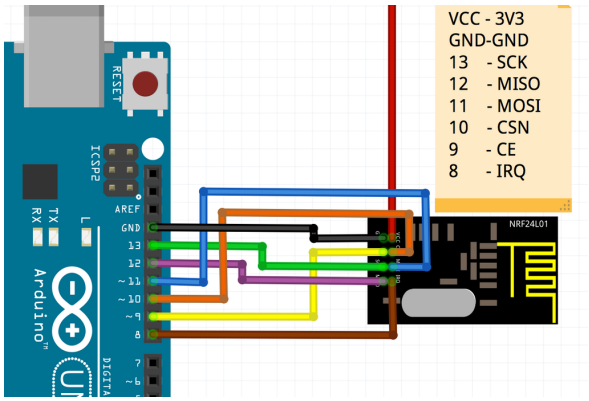
There are multiple logical components that are involved in the process of performing an *OTA* upgrade of the target device. Starting with the firmware image to be flashed, a PC application is required to read the file (.hex) and to communicate the erase/re-write commands to the nRFEncoder, a device that that can send those erase/re-write commands to the client device (*boot loader)* via the nRF24L01 radios.



The processor selected for this project was the *Microchip PIC 16F1823* microprocessor. As with most projects of this nature, it was selected due to existing familiarity with the device. This micro-processor is a 14-pin device, sporting 2K of Flash Code space, 256 Bytes of EEPROM, 128 bytes of RAM, a 32MHz internal oscillator, and the ability for an application running on the device, to erase and re-program the code space of the device itself. The 16F182x series also includes *code protection* capabilities that enable us to prevent self-erasing (and re-programming) portions of the flash code space, and thus making our *boot loader* safe from accidental self-destruction.

The radio modules are based on the nRF24L01+ transceiver from Nordic Semiconductors. These modules are extremely popular amongst engineering hobbyists, inexpensive, use a SPI interface for communications, and require minimal additional parts/connections to get them operational.





The nrfEncoder, the device that translates the erase/write requests from the PC into commands that can be sent wirelessly to the client device, is an *Arduino Uno* that that has been connected to zn additional nRF24L01 radios, according to the FRITZING diagram[1].

NOTE: There are plenty of online comments lamenting the *Uno's* under powered 3.3v regulator, and I found the need to modify my nRF24L01 modules, by adding a 10uf Electrolytic Capacitor to them (Vcc to GND) to maintain a more reliable wireless connection.

The project for the nRFLoader is based on the existing 16F1-USB-Bootloader from Matt Sarnoff. This project leverages the PC loader application itself, whereas the PIC code has been completely replaced. As with the original project, I opted for hand coded MPASM code[2], and managed to keep it within the same 512-byte size target.

The NRF24 radio modules

While appearing simplistic at the outset, the nRF24L01 radios turned out to be quite challenging to operate. A plethora a simple transmitter/receiver examples exist in the wild (including the examples from the Arduino RF libraries), yet there are very few examples of code toggling the device to/from transmit/receive functions. This code goes even further, as it also attempts to migrate between broadcast operating mode, and the point-to-point mode using the intrinsic Auto Acknowledge feature of the radio. To that end the reader may find some explicit delays introduced into the code, most of

[1] FRITZING image source: https://www.instructables.com/Arduino-NRF24-Two-Way-Radio-for-Telemetry-and-Rc

[2] I have just (2022-03-28) learned that Microchip has retired the MPASM assembler and thus this project may have less appeal than I had hoped. But in for a penny, in for a pound (dollar, lira, rupee)

which have been created as a result of brute force trial and error exercises. The author is open to constructive feedback in this area. Here is a list of some of the configuration settings selected for the server and receiver:

Radio Channel: A radio channel setting of 0x52 (82) was selected, as on-site experiments indicated that I should not experience much interference in this range.

Payload Size: A fixed payload size of 32 was selected in order to accommodate the WRITE request, and to keep the receive payload checks to a minimum.

Transmission Rate: This project opts to use a 2Mbps data rate as it matched the existing data rate for the original project, and I didn't want to deal with dynamic "changing data rates" as the device moves from bootloader to application operating modes.

CRC: A 16-bit CRC was selected, because this will provide a more reliable communications pipe.

Auto Acknowledge: The AutoAck feature of the NRF24L01 is leveraged to provide a more reliable data communications pipe. This AutoAck feature is initially disabled during the boot sequence, but is later enabled if a BIND request is received from the server.

Address: For this project I have opted to use a 3-byte address for the nRF23L01 radio communications. This allows for up to 16 million unique device addresses, while also reducing the amount of memory required for storing and processing addresses to a manageable size.

Idle Server	**Idle Client**
Tx Address: 0xC0DEC1	Tx Address: 0xC0DEC1
Pipe 0: <unused>	Pipe 0: 0xC0DEC1
Pipe 1:0xC0DEC1	Pipe 1:<deviceID>

Bound Server	Bound Client
Tx Address: <deviceId> AutoAck Pipe 0: <deviceId> Pipe 1:0xC0DEC1 Pipe 2:0xC0DE01	Tx Address: 0xC0DE01 AutoAck Pipe 0: 0xC0DE01 AutoAck Pipe 1:<deviceID>

PIC to nRF communications

The nRF module is configured/controlled via the SPI interface (see the nRF24L01 data sheet for details on commands and registers). The PIC microcontroller has an integrated Master Synchronous Serial Port (MSSP) to handle the task of clocking data in/out of the SPI interface. The following code is required to configure the MSSP for communicating with the nRF module:

```
bcf     SSP1STAT,SMP          ; Sample in the END of data output time
bsf     SSP1STAT,CKE          ; Transmit when clock goes ACTIVE to IDLE

movlw   0x01                  ; Go for divide by 8 = 4Mhz clock
movwf   SSP1ADD
movlw   0x2A                  ; Mode 0,0 = CKE=1 CKP=0
movwf   SSP1CON1              ; Select SPI clock and enable SSP
```

The SPI interface also requires the use of a GPIO for chip select. This pin is pulled LOW at the start of the SPI command sequence, and then pulled HIGH upon completion. Pseudocode for writing a value to a register of the radio is presented below.

```
Pull SPI_CS low
Write the Register Address to the SPI interface
Wait until the register has been clocked out
Write the Value to the SPI interface
Wait until the value has been clocked out
Pull SPI_CS high
```

A second GPIO, NRF Chip Enable (NRF_CE), is required to manage the NRF214L0 states. Taking the device into and out of standby to either *Receive* or *Transmit* modes. The NRF_CE is also used to trigger the transmission of a payload in the transmission FIFO, via a minimal 10us

pulse. A 50Mhz oscilloscope was connected to the PIC as various values were used to determine that the following code sequence would generate a reliable 12us pulse.

```
bsf     NRF_CE  ; Enable Transmitter
movlw   32      ; Scoped at 12 us
decfsz  WREG,f
goto    $-1
 bcf     NRF_CE
```

Communication Protocol

PC to the Arduino

The *nRFLoader* PC application is used to parse the input firmware file and transmit commands to the Arduino. The *nRFLoader* is based on the *usb16f1prog* PC application from Matt Sarnoff's project, and it defined the following messages that can be sent from the PC to the target device:

1. [0x42] SYNC– used to synchronize the communications between the PC and the Arduino
2. [0x01] SET_PARAM– used to setup and erase the row of flash at the specified address. This message also includes a checksum calculation for the subsequent WRITE request.
3. [0x02] WRITE – sequence of bytes to write to the previously defined address.
4. [0x03] RESET – command to cause the target device restart

As part of the startup sequence, the pic bootloader performs a checksum validation prior to jumping to the application code space. To accommodate this, a new message from the *nrfLoader* PC application has been defined, which contains code size, and checksum values for the loaded application:

5. [0x04] AUDIT – command to perform a checksum calculation over the entire code space. if the checksum audit is successful, the checksum value is written to the EEPROM and will be used on boot to validate the application code space before use.

Arduino to Client

The Arduino, running the *nRFLoader.ino* image, converts the incoming serial commands into a series of commands, fitting in the 32-byte payload limit of the nRF24L01 radio. The first step in this process is for the client device to use the broadcast message capabilities of the nRF24, to share pipe addresses, allowing the device and server to establish a point-to-point communications link. This handshaking is referred to as "Binding" within the code state machines.

All messages defined in the nRFLoader protocol have the same overall structure. The first byte of the payload contains the *command* identifier for the packet, with up to 31 bytes of additional data as part of the payload. The communication protocol includes the following commands, and their responses.

[0x88] – Device Announce [BROADCAST]

Message Source: DeviceID of the client
Message Target: The Server broadcast listener [0xC0DEC1]
Structure: [0x88] [$DeviceID_0$] [$DeviceID_1$] [$DeviceID_2$]
Response: none

Transmitted as part of the power-on sequence within the client device. Note that the command byte for this broadcast is 0x88, allowing it to be easily masked/ignored by the client device.

[0x87] – Bind Request [BROADCAST]

Message Source: 0xC0DE<pipeID> – the server point-point-address
Message Target: 0xC0DEC1 – the server broadcast address
Structure: [0x87] [$pipeID_0$] [$ServerID_1$] [$ServerD_2$]
Response:
ACK – [0x87][00]

The BIND request is part of the handshake messaging, used to establish a more robust point-to-point communication connection between the server and the client. The client device, upon receiving this message, will change from broadcasting behaviour, and attempt to establish a point-to-point connection to the address/pipe specified in the payload of Bind Request. This point-to-point enables the Auto Acknowledge feature of the NRF24L01 for all subsequent communications.

Note: The Arduino server uses a timeout on transmission of the BIND request. If the device fails to respond in a timely manner, the BIND is re-broadcast to the client.

Once the secure point-to-point handshake is completed, the Arduino will parse the serial commands from the *nRFLoader* application, into the following nRF command payloads.

SET_PARAM

The SET_PARAM message contains the flash row address, a checksum value for the new row, and a flag to identify if the row should be erased prior to writing. This command is converted in the NRF_START command:

[0x80] – Start Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x80] [AddressL] [AddressH] [Erase]`

Response:

ACK – [0x80][00]

NACK – [0x80][01]

WRITE

The WRITE message contains the bytes/words to be written to the flash, and the previously defined address. Note however, that as the PIC microcontroller requires that any erase of the flash be completed on a 'per ROW' (a ROW is equal to 16 words (32 bytes)), there is insufficient room to include all the data for a single row within the nRF payload packet. The WRITE command, therefore, is broken down into two nRF payload commands, to get all the necessary data to the target.

[0x81] – Write Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x81] [Byte`$_0$`] [Byte`$_1$`]… [Byte`$_{30}$`]`

Response:

ACK – [0x81][00]

NACK – [0x81][01]

Upon receipt of a WRITE request, the client device will copy all the bytes in the payload to the *write latches* for the row, maintaining an on-going 8-bit checksum calculation.

[0x82] – Commit Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x82] [#words] [DataL] [DataH] [CSUM]`

Response:

ACK – [0x82][00]

NACK – [0x82][01]

Upon receipt of a COMMIT request, the client device will copy all remaining words, as indicated in the *numWords* field, to the *write latches* for the row. If the calculated checksum for the row matches, then the appropriate register sequence to invoke the FLASH of the code is triggered.

AUDIT

Once all WRITE requests have been fulfilled an AUDIT message is sent to the client in order to verify the integrity of the entire image, and to update the boot time validation checksum that is stored in the EEPROM of the device.

[0x83] – Audit Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x83] [Address`$_L$`] [Address`$_H$`] [Size`$_L$`] [Size`$_H$`] [CSUM`$_L$`] [CSUM`$_H$`]`

Response:

ACK – [0x83][00]

NACK – [0x83][01]

Note: In order to save code space in the client, the start address of the Audit message is ignored, and instead all Audit requests are calculated using the SIZE bytes from the start of the application code space.

After a successful upgrade of the firmware image, the device is reset using the REBOOT command.

[0x86] – Reboot Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x86]`

Response: none

To maintain communications between the client and server, a heartbeat message (and response) was introduced. If the server fails to receive a heart beat reply within 7 seconds, the Arduino server will return to an IDLE state and abandon the communications.

[0x84] – Heart Beat Request [P2P]

Message Source: 0xC0DE<pipeID> – the server point-point-address

Message Target: <client Device ID>

Structure: `[0x84]`

Response:

ACK – [0x84][00]

Access to the Code

The Code

- The nRFLoader: https://github.com/LabRat3K/NrfLoader
- The USB16F1 loader: https://github.com/74hc595/PIC16F1-USB-Bootloader

Data Sheets

- PIC16F1823 http://ww1.microchip.com/downloads/en/devicedoc/40001413e.pdf
- nRF24L01 https://www.mouser.com/datasheet/2/297/nRF24L01_Product_Specification_v2_0-9199.pdf

Other Reading

- nrf24 Tutorials http://blog.diyembedded.com/search/label/tutorials

---

What's Next?

- WNRF – an ESP8266 to NRF24 bridge

- Writing PWM client to work with the Bootloader
- Review/Export common code in the bootloader for use by client.