`nrftool`
A Python Script to Program a Nordic nRF51 Using a Bus Pirate

Ted Herman

3 September 2016

Abstract

Nordic Semiconductor (`nordicsemi.com`) has a product line combining an ARM processor with a low-energy Bluetooth radio (BLE) on a chip. This document explains the rationale and sources for a linux command, written in Python, to program an nRF51 device. At present, the tools has only been tested on one particular device.

Introduction

Nordic's nRF5x series of products offers an inexpensive yet powerful entr'ee to the Internet of Things. One small package combines a processor, memory, Bluetooth radio, and pins for common bus protocols and general-purpose input/output operations. For example, SeeedStudio sells (as of this writing) an nRF51822 module, including on-board antenna, that is 13.0mm × 18.5mm × 2.3mm. By virtue of low-power modes of the ARM processor, a device based on this module could run for years on a coin cell battery by sparingly using the radio. The Nordic nRF5x products are flexible: they can be programmed and reprogrammed (even over-the-air programming by radio is possible), and a device can operate either as an endpoint, such as a sensor, or a collection point, which communicates with multiple sensors using Bluetooth. Many experimental projects and new devices are now being developed using an nRF5x as a core processor.

A motivation for the work described here is Nordic's willingness to allow development based on an open-source toolchain. Programs can be written in C, compiled with GCC (the ARM version), and standard "makefile" commands compile the programs. Less open is the process of uploading a compiled program to the flash ROM on an nRF5x processor. Nordic does sell a development kit that includes a third-party solution, the Segger J-Link, which is an excellent tool to program and debug an nRF5x processor.

The Nordic SDK can be freely downloaded (there are several versions), which is essential to use Bluetooth. Like Texas Instruments also does, the Bluetooth

stack is provided only in compiled form; the source code is proprietary. In practice, development consists of first putting the BLE core stack onto the nRF5x, which occupies around 90-120KiB of the read-only program flash memory. The custom part of development is to upload an application that sits alongside the BLE stack in ROM. When the nRF5x boots, it loads both parts.

For those interested in using open-source tools, the Nordic SDK is not a well-crafted library of modules, header files, and system calls: components tend to be scattered around several locations, and different levels of abstractions in how the files and calls are packaged can make for tough going. For instance, unlike standard C programming in Linux, where with one parameter on a compile, a rich library of headers and shared libraries is accessible, the Makefile for a Nordic SDK program has to manually put in specific includes for header file, additional code to be co-compiled, and so on.

In keeping with the open-source direction of the effort described here, the device for programming an nRF5x is the Bus Pirate[1]. A Bus Pirate is useful for experimenting with many common sensors (temperature and humidity, color sensor, accelerometer), so it has broad appeal for developer experiments on multiple projects. Software for a Bus Pirate can be in Python, which was used to develop the `nrftool` described here.

Prerequisites

One goal in constructing `nrftool` was to keep prerequisite software to a minimum. To use `nrftool`, three components are needed (beyond linux):

- `Python 2.7` (though probably it's not difficult to adapt it to Python 3).
- `pySerial`, which is needed by the Bus Pirate software, to communicate using the FTDI driver across USB to the Bus Pirate hardware. After `pySerial` has been installed, a Python program can `import serial` and work with many serial-to-USB converters. There is at least one place in `nrftool` that may be version-dependent for `pySerial`, which is the `inWaiting()` call; this has been changed into an `inWaiting` attribute in newer versions of `pySerial`, so the code in `nrftool` might need to change a bit.
- `pyBusPirateLite`, which can be copied from software found via the main Bus Pirate home page. I'm not sure about version numbers or whether `pyBusPirateLite` should be installed into a standard Python library.

For the remainder of this document, I'll suppose that `/dev/ttyUSB0` is the linux device which refers to the connected Bus Pirate, and that this device can be read from and written to by a Python program using the `serial` module without special permission (likely meaning one has to set up something like `udev` permissions to allow user programs such read/write access).

[1] `en.wikipedia.org/wiki/Bus_Pirate`

Bus Pirate Wiring. In my tests, I've connected four wires from the Bus Pirate to an nRF51:

- GND (ground to nRF5x ground, the one that's right next to the VIN)
- 3.3v (power to nRF5x VIN, Voltage In)
- CLK (to nRF5x SWD, where "Single Wire Debug" or something like that; this is the clock wire)
- MOSI (to nRF5x SWIO, which is the data wire)

The terminology for these pins may vary depending on specific nRF5x platform.

Command Examples

The command syntax for `nrftool` isn't too pretty. Rather than describe a comprehensive syntax, here are the common examples.

```
nrftool info /dev/ttyUSB0
```

Use this command to test connectivity to the Bus Pirate, which in turn is wired up to an nRF5x. It will read a few registers from the nRF5x and print their value. It may also launch the current program on the nRF5x ROM (no guarantee, though it has worked reliably for me).

```
nrftool masserase /dev/ttyUSB0
```

Clears all the nRF5x ROM memory, setting it to `0xff` values everywhere, which is standard for NAND-type flash memory (later, when programmed, some of the bits will be rewritten to zero).

```
nrftool --address 0 --downloadsize 0xC00 \
     --downloadfile image.bin download /dev/ttyUSB0
```

Read 3KiB from the ROM of the nRF5x and write it to file `image.bin`. Instead of specifying the read amount as `0xC00`, one could also write 3072. The `address` parameter specifies the staring point, within ROM, to start reading.

```
nrftool --address 0 --progfile myprogram.bin \
      program /dev/ttyUSB0
```

Write the binary from the file `myprogram.bin` into the ROM, first erasing what was there before, and start at address zero in ROM. This command would be used either to program an nRF5x with some standalone program not using Bluetooth, or the command would be used to put Nordic's BLE stack onto the ROM. (For the endpoint Nordic BLE stack, which is around 90KiB in size, this `nrftool` operation takes about two minutes to complete.)

```
nrftool --address s110 --progfile myprogram.bin \
     program /dev/ttyUSB0
```

Writes `myprogram.bin` to ROM starting at location `0x18000`, which is what the endpoint BLE stack expects (one can equivalently specify the address as `0x18000`); this stack is known as "s110" throughout the Nordic SDK. For programming a collection point, use s120 instead. There is also an s130, but not yet known in `nrftool`'s code.

`nrftool` logic

This section has some documentation on how `nrftool` works. The Python code communicates with the Bus Pirate primarily as it would with a serial device; the method call `write` (and `flush` to wait until buffered transmission finishes) sends bytes to the Bus Pirate; the `response` method call returns bytes read from the Bus Pirate. There is a brief sequence of values written that initialize the Bus Pirate for raw-wire communication, which is used as the basis for the one-wire (SWD) protocol that the nRF5x expects. After this setup, the Bus Pirate has commands[2] to read a byte, read a bit, write bytes, write bits, and delay for some number of clock ticks.

Background

Most of what `nrftool` does depends on knowing facts about the `nRF5x` platform, the SWD protocol, and ARM over-the-wire debugging. There are primary sources for such facts:

- The *ARMv7-M Architecture Reference Manual*, particularly parts B1.2, B3.2, and C1.2. The tables and text in this manual give the memory layout, some crucial registers for the debug protocol (how `nrftool` reads and writes ROM), and the Non-Volatile Memory Controller (NVMC), which is the ARM terminology for ROM.
- The *nRF51 Series Reference Manual*, which explains more about how the debug operations can be used to program ROM and get useful information about the particulars of a Nordic nRF5x device (page size, memory size, features).
- There's a quite readable introduction in *AN0062 Application Note: EFM32 Programming Internal Flash Over the Serial Wire Debug Interface*, which is about a different device than used here, but nonetheless explains basic commmands and responses of the single-wire debug (SWD) protocol.

There is much practical wisdom in the MC HCK project (`http://mchck.org`). There is a github repository with Ruby programs for SWD operations (with Bus

[2] `http://dangerousprototypes.com/docs/Raw-wire_(binary)`

Pirate), which have been successfully translated to nRF5x programming. Without question, this Ruby program is far more complete than the modest `nrftool` described here. The only advantage of `nrftool` is for people not familiar with Ruby, and for those who want to learn more about the primary sources of information and the protocol techniques. For such an audience, `nrftool` is a mostly self-contained starting point for further development using SWD to inspect and control an ARM device.

Register Sets

Given the complexity of the ARM family of processors, it's not surprising to find many abstractions for components of memory and register organization. Only a few registers were used for `nrftool`:

DP Registers are used for a few simple "debug port" operations. Mostly they are just used to go up the next level, the "access port" registers.

AP Registers are the portal for reading and writing RAM on the processor. Though `nrftool` doesn't use many registers (because it doesn't do actual debugging, such as stepping through instructions in a program), it does read and write a few registers in a 256-byte address space of AP registers.

FICR Registers can be read, through AP registers, to discover facts about the particulars of an nRF5x device. These registers are address-aliased to places in RAM.

SCS Registers are in a memory-mapped 4KiB address space with words for configuration, status and control of the system, faults, timer, interrupts, and such.

`NVMC Registers` are the means to read and write ROM. Here again, the mechanism is memory-mapping through RAM. So, to read or write ROM, one has to write a RAM address in AP Register, then read or write another AP Register (some of which memory-map to RAM), and finally read a value through a DP register.

A typical command sequence transmitted to the Bus Pirate will start with a length descriptor, followed by that number of bytes (of the command, or of a register value to be written). In addition to such a command sequence, extra bytes sent to the Bus Pirate solicit acknowledgements or data values. The SWD protocol sometimes requires a "pause" (silent clock cycle) known as the *turnaround*, which is a change in direction of information flow on the single wire.

Turnarounds

The manuals cited previously explain the circumstances for injecting turnarounds into any protocol between host and ARM (or nRF5x device). My experience deviated a bit from these explanations. What I found were three circumstances where turnaround is needed:

1. A turnaround is needed before a write operation when the previous operation was a read.

2. A turnaround is needed before supplying the data of a write operation.

3. A turnaround is needed only for a read when the previous operation was a read.

In these three rules, the presumption is that the only thing a program does is read or write operations. That is certainly the case for `nrftool`.

Read and Write Amounts

Does it make a difference whether a program sends a byte at a time or sends multiple bytes to the serial port? In my experiments, it didn't seem to make a difference, with one exception. Reading a 4-byte (32 bit) register failed when done with single-byte reads; only a read of four bytes worked correctly. In the many experiments leading up to this discovery, the design of `nrftool` evolved. The philosophy of the program is now to defer actual read and transmit with the Bus Pirate until necessary, by buffering commands and delaying the processing of acknowledgements.