Building for the ARM Cortex-M0 with GNU tools

James Gowans

July 13, 2014

Licence

Preface

This guide is intended for a range of audiences from those who have never compiled code with GCC before, to those who are experienced with GNU and want to start writing code for ARM. This guide does not intend to teach you programming, only building and debugging. You should have some familiarity with C and assembly before attempting to develop code.

Contents

1 Toolchain Overview

A toolchain is a collection of software tools that facilitates the process of getting your source code executing on a target platform. This guide will start with the simplest toolchain and introduce more tools and complexity later as required. The simplest set of tools is as follows:

1. *Text editor:* provides the capability to modify source code files. Typically something like notepad (Windows) or gedit (Linux) will work fine. However, it's recommended for Windows users to install a more customisable editor: Notepad++

2. *Assembler:* converts human-readable assembly code into machine code, known as object files

3. *Linker:* takes one or more object files and turns them into a single executable which can run directly on the target. The process of linking involves ascertaining the final (absolute) memory address for each section of the executable. For both the assembler and linker, we will be using a fork of GCC called gcc-arm-none-eabi which provides these two pieces of software.

4. *Interface to hardware debugger:* the debugger microcontroller will typically be connected to the computer via a USB connection. There needs to be a way to send data down the USB cable to the debugger to get it to load code onto the micro or debug running code. The program we will be using to provide this interface is OpenOCD.

5. *Debugger client:* A debugger client is software which connects to the hardware debugger through the interface and manages what it does by sending it instructions. This can involve sending it executable machine code to load onto the target microcontroller, or causing the target to stop/start/pause execution of the code, or even reading/writing of data in the memory of the target. The debugger client we will use is GDB, also provided in the gcc-arm-none-eabi package.

We will now proceed to review gcc-arm-none-eabi and OpenOCD in some details

1.1 gcc-arm-none-eabi

gcc-arm-none-eabi is a set of utilities (a package) aimed at facilitating the building and debugging of code for ARM processors. As discussed earlier, we will be using 3 utilities from this package when building our code, namely the assembler (arm-none-eabi-as),

the linker (arm-none-eabi-ld) and the debugger (arm-none-eabi-gdb). There are about 25 utilities in total in this package, some more of which we will introduce in the later chapters of this guide.

A note on the naming convention of *gcc-arm-none-eabi*:

- *gcc:* This software package is a modification of GCC (GNU Compiler Collection). GNU is a project launched in the late 1980's aiming to promote software freedom. A major part of GNU is the compiler, GCC.
- *arm:* This toolchain is designed specifically to operate on code for the ARM family of CPUs.
- *none:* This referrs to the operating system on which the code that is built will run on. "None" mean it will not run on an operating system. Rather it will run "bare-metal": directly on the hardware. It's worth noting that when we work on a Linux or Windows system and compile code to run on a different platform (in this case: bare-metal), we are doing something known as cross-compiling: compiling code for a different architecture than the compiler is running on.
- *eabi:* Embedded Application Binary Interface. As far as I can tell, this is some sort of standard or specification which defines how the machine code files which the toolchain produces are structured. By having a standard, it allows our files to be portable and allows files produced by different toolchains to work together.

1.2 OpenOCD

This is a program which facilitites communication with hardware over a USB connection. The way it works is that it opens up a port[1] to accept user commands, interprets those commands and passes them to the hardware. It also listens for responses from the hardware and presents that to the user by sending data out of the port. When you launch OpenOCD, you specify two things: the type of hardware you're connecting to (target microcontroller) and the type of hardware debugger which you're connecting to it through. In our case, out target is a STM32F0 family microcontroller and our hardware debugger type is a ST-Link-v2. ST-Link-v2 is the type of firmware which is running on the debugger micro.

Typically, when OpenOCD runs it make a number of ports available to us, each with different purposes. The one which is most important to us is port 3333; the port which deals in GDB traffic. It's designed to have a GDB client connect to it, and it knows how

[1] If you've never worked with ports before, a port is basically an address which network traffic can go to. Much like a numbered mailbox, different applications can communicate with each other through ports. One application can place data into the mailbox (send to a port) and another application and receive data from the mailbox (listen on a port). Typically ports are used to communicate between applications on different computers (ie: your computer's browser communicating with a web server on another computer) but nothing stops applications on the same computer also communicating via ports.

to deal in GDB commands. When we want to use GDB we'll need to connect it to port 3333, which is where OpenOCD is listening for connections and instructions.

1.2.1 Note on multiple instances of OpenOCD

Note that only one application can be listening on a port at one time. This means that if another application is listening on port 3333 when OpenOCD launches, it will crash. The further implication of this is that only one instance of OpenOCD can run at a time! If OpenOCD is refusing to run, it may be because you have another instance of it running. Kill all running instances of OpenOCD (or to be really sure: reboot your computer) if OpenOCD won't launch.

2 The Terminal

If you're familair with the terminal, skip this chapter.

When we run these programs which have just been discussed, we will use the terminal to run them and view their output. The terminal (aka: "command prompt" in Windows) is a program which allows you to type commands and then execute the commands.

2.1 The Linux Terminal

Assuming you're in Ubuntu, a terminal is launched by pressing Ctrl+Shift+T. Alternatively, you can press your Super key and then type "terminal" and you should be presented with a launcher for it which you can click on.

Once in the terminal, you can print out your current working directory with the command `pwd`. You can change directory with the `cd` command. For example:

```
$ cd /home/$USER/Documents
```

to change to the documents directory, or simply The terminal

A terminal always operates in a specific directory. When you launch a terminal, the default directory is usually your home directory. Whenever you try to run a program or modify a file from the terminal, it attempts to modify the file which is inside the working directory which the terminal is set to. In order to see what working directory your terminal is in, you can use the command *pwd* (print working directory). In order to change the working directory to a different one, you can use the comma

```
cdqe 1, r8
push 1
add rsp, 4
push 1
```