Building for the ARM Cortex-M0 with GNU tools

James Gowans

June 18, 2015

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 **5**
- 1.1 gcc-arm-none-eabi . . . 5
- 1.2 OpenOCD . . . 7
  - 1.2.1 Note on multiple instances of OpenOCD . . . 8

2 The Terminal **9**

3 Installing the Toolchain **10**
- 3.1 Linux Install Guide . . . 10
  - 3.1.1 Text Editor . . . 10
  - 3.1.2 gcc-arm-none-eabi . . . 10
  - 3.1.3 OpenOCD . . . 10
- 3.2 Windows . . . 11
  - 3.2.1 Text Editor . . . 11
  - 3.2.2 Driver . . . 11
  - 3.2.3 gcc-arm-none-eabi . . . 11
  - 3.2.4 OpenOCD . . . 12

4 Loading **13**

5 Assembling **15**
- 5.1 Source Code File . . . 15
  - 5.1.1 Instruction . . . 15
  - 5.1.2 Assembler Directives . . . 15
  - 5.1.3 Labels . . . 16
- 5.2 Command Line Arguments . . . 17

6 Linking **19**

7 Debugging **20**
- 7.1 GDB Commands . . . 20

8 C Debugging **21**
- 8.1 Querying Defined Variables . . . 21
  - 8.1.1 Global Variables . . . 21
  - 8.1.2 Local Variables . . . 21
  - 8.1.3 Arguments . . . 21

8.2 Getting More Info On Or Modifying Variables 21
8.2.1 Types 21
8.2.2 Dereferencing Pointers 22
8.2.3 Modifying Variables 22
8.3 Flow Control 22
8.3.1 Next 22
8.3.2 Finish 22

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 shown graphically in Figure 1.1 and will now be discussed.

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 relocatable (no final address) machine code files. These files are known as object files and are not directly executable.

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.

4. *OpenOCD* (On Chip Debugger). This is an interface to the hardware debugger. The hardware 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. When running in Windows, the ST-Link driver is required. When running in Linux the libusb-1.0 package is required.

5. *GDB* (GNU debugger) This is a debugger client. It is software which connects to the hardware debugger through the interface and manages what the hardware debugger 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 is a software package called gcc-arm-none-eabi which provides the assembler (arm-none-eabi-as), the linker (arm-none-eabi-ld) as well as the debugger (arm-none-eabi-gdb). We will now proceed to review gcc-arm-none-eabi and OpenOCD in some detail.

1.1 gcc-arm-none-eabi

gcc-arm-none-eabi is a fork of the very popular GCC, developed by GNU. This particular fork is maintained by the guys at ARM so it's good software which will probably be

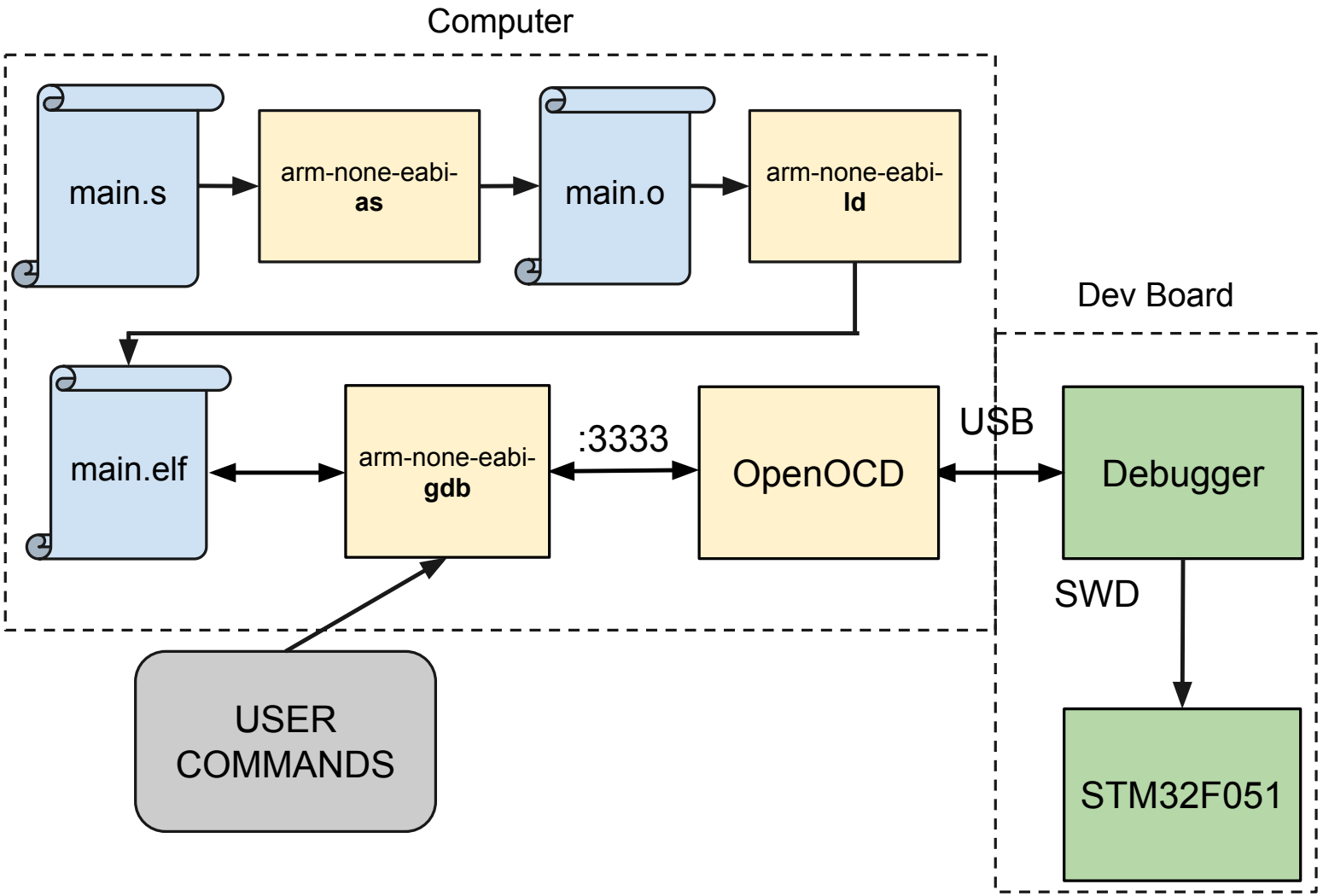

Figure 1.1: Process which source files takes to be loaded onto micro.

around for quite some time. The package contains a set of utilities 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 `as`, `ld` and `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 refers 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 facilitates 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 to whatever application is connected to it. 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

[1] If you've never worked with ports before, a port is basically an address which IP 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.

deals in GDB traffic. It's designed to have a GDB client connect to it, and it knows how 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 familiar with the terminal, skip this section.

When we run these programs which have just been discussed, we will use the terminal to run them and view their output. The terminal is known as command prompt in Windows and as many things including the command line, shell or bash or terminal in Linux. The Linux terminal is much better and nicer to use than the Windows one. This is mainly because a lot of Linux development work is done in the terminal and as such it needs to be really good for productivity reasons.

In light of this, if you've never used Linux before and are keen to give it a try, I'd strongly recommend it. For the sort of work we will be doing, it is easier and more enjoyable to use.

I'm not going to cover how to use the terminal in this guide as that has already been documented very well in other places.

For Linux users, a brief introduction to the terminal can be found here: http://linuxcommand.org/. Additionally, there is a free book covering the contents of this website called The Linux Command Line which has been uploaded to `Resources\Further_Reading`. I advise you be familiar with at least the first two chapters, preferable the first four.

For Windows users, there is a short introduction to command prompt found at http://dosprompt.info/. You should read through and practice everything discussed in that web page.

Henceforth, you are assumed to have adequate knowledge of the terminal.

3 Installing the Toolchain

Now that we have some idea of the software components which we need to install in order to build and debug code, let's install them.

3.1 Linux Install Guide

This section is written for Ubuntu. If you run a different distro, you may need to modify the commands slightly at best.

3.1.1 Text Editor

Either Vim or Gedit will be good enough for our needs. No need to install an additional editor.

3.1.2 gcc-arm-none-eabi

. The following assumes you can use PPAs. If your distro does not support PPAs you'll need to compile gcc-arm-none-eabi from source. Run the following commands to add the PPA to your system and install gcc-arm-none-eabi:

```
$ sudo apt-add-repository ppa:terry.guo/gcc-arm-embedded
$ sudo apt-get update
$ sudo apt-get install gcc-arm-none-eabi
```

You can check that the tools have been successfully installed, open a new terminal and run the following.

```
$ arm-none-eabi-gdb --version
$ arm-none-eabi-ld --version
```

Each command should give you some info about the tool if it's working properly. If you get something like *Command not found* then it's not installed correctly.

3.1.3 OpenOCD

To get OpenOCD we will pull it from SourceForge, compile it and install it. First, the libusb-1.0 package must be installed.

```
$ sudo apt-get install pkg-config libusb-1.0*
$ cd /tmp/
$ wget http://downloads.sourceforge.net/project/\
    openocd/openocd/0.8.0/openocd-0.8.0.tar.bz2
```

```
$ tar xf ./openocd-0.8.0.tar.bz2
$ cd openocd-0.8.0
$ ./configure --enable-stlink
$ make # takes about 3 minutes
$ sudo make install
```

To test that it's successfully installed, connect the micro to the computer and execute the following in a fresh terminal:

```
$ sudo openocd -f interface/stlink-v2.cfg -f target/stm32f0x_stlink.cfg
```

If you get a message about 4 breakpoints and 2 watchpoints, you're all good.

3.2 Windows

3.2.1 Text Editor

While one could work in Notepad, it's not a great editor for real dev work. A much more popular one is Notepad++. Download and install Notepad++ (called npp.6.6.7.Installer.exe) from Vula. Once the install is complete, download the syntax highlighting file (userDefineLang.xml) from Vula and paste it into to the Notepad++ directory:
`%APPDATA%\Notepad++`
`%APPDATA%` is a special name which Windows understands. Simply type the above string into Explorer to be taken to the directory. Once installed, run Notepad++ and disable spell checking by going to `Plugins -> DSpellCheck -> Settings`. Under File Types:

- change to *Check only NOT those*
- replace *.* with *.s

The reason we do this is that the spell checker keeps underlining all of our assembly instructions because they are not English words. Annoying.

3.2.2 Driver

Download the correct driver for your version of Windows from Vula. Extract the .zip and install the contents of it either by executing the .exe for the Windows 7 driver .zip or by executing the stlink_winusb_install batch file for the Windows 8 .zip. You can now connect the micro to your computer. If the red LED on the debugger goes solid red then the driver is probably installed correctly. If it's flashing red then the driver's probably not installed properly.

3.2.3 gcc-arm-none-eabi

Download the executable from Vula and run it. When the install is complete you will be presented with some tick boxes. Tick the one called *Add path to environment variable* and leave the others as defaults. By adding it to the path you are able to execute the tools

from any directory rather than having to have your terminal in a specific directory. Close the terminal which appears and launch a fresh one. Check that the tools are accessible by running:

```
\$ arm-none-eabi-gdb --version
\$ arm-none-eabi-ld --version
```

If some words about the GNU debugger and linker appear, it's working. If it goes on about *not recognised as an internal or external command* it's broken. Re-install gcc-arm-none and tick the path box. Close the terminal by typing `exit`.

3.2.4 OpenOCD

Download the OpenOCD zip from Vula.

Extract to somewhere logical like `C:\Program Files\`.

Rename `openocd-0.8.0\bin-x64\openocd-x64-0.8.0.exe` to `openocd.exe`

Next, we want to add the OpenOCD path to the PATH environment variable so that we can run OpenOCD from any directory.

Go to `Control Panel -> System -> Advanced System Settings`. Under User Variables, click the Path variable and click Edit. You'll see a list of paths (possibly only 1). Append your OpenOCD binary path to the end. Something like:

```
;C:\Program Files\openocd-0.8.0\bin-x64\
```

Note: The semicolon separating the paths is critical Click OK a few times

To test that OpenOCD is accessible and working, connect your micro and run the following.

```
$ openocd.exe -f interface/stlink-v2.cfg -f target/stm32f0x_stlink.cfg
```

If you get a message about 4 breakpoints and 2 watchpoints, you're all good.

4 Loading

Now that the tools are installed, we will load an executable onto the micro using OpenOCD and GDB. The executable to be used is a pre-compiled version of the demonstration code called `demo.elf` which should be downloaded from Vula.

Kill any existing terminals and open a fresh one. Connect your micro and launch OpenOCD again as you did before. Again, ensure that you get the message about breakpoints and watchpoints. This means it's successfully connected to the micro.

Leave that terminal running in the background and open another one. `cd` to whatever directory you downloaded the .elf file to. Run:

```
$ arm-none-eabi-gdb demo.elf
(gdb) target remote localhost:3333
(gdb) monitor reset halt
(gdb) load
(gdb) continue
```

After running `continue` your micro should start running and the words `You did it!` should appear on the LCD screen. If they don't you didn't manage to do it.

The commands mean the following:

- `arm-none-eabi-gdb demo.elf`: This launches GDB and tells it that we want to use the demo.elf file for debugging. The file name which you specify must be in the terminal's working directory
- `target remote localhost:3333`: This tells GDB to connect to OpenOCD. Specifically, it says that the target which we want to debug is a remote target (as opposed to running the code on the computer) and that the target can be reached by connecting to port 3333 which is the port which OpenOCD listens for connections on.
- `monitor reset halt`: This causes the debugger to reset the target micro by pulling its NRST line low for a few milliseconds. When the line is released, the debugger prevents the target from running whatever code is already loaded onto it. Once it has been reset, the debugger can jump in and prepare the micro to have code loaded onto it.
- `load`: GDB pushes the machine code contained in the demo.elf file onto the target micro via OpenOCD and the debugger.
- `continue`: The target micro is instructed to start running the new code which is now on it.

You can stop the target from running with Ctrl+C. You can then quit GDB with the `quit` command. OpenOCD can be killed with Ctrl+C. Note that if you disconnect your dev board from the computer you will have to kill and relaunch OpenOCD to get it to reestablish comms with the debugger.

If you're running Linux, you can launch gdb with the flag `-tui` placed before the demo.elf filename. This allows GDB to run in a split screen mode where the top half of your screen shows you the source code you are debugging and the bottom half allows command entry. This is one of the coolest things about GDB - check it out.

5 Assembling

A note on the word *compiler:* A compiler is any piece of software which removes a level of abstraction in code. In other words: a compiler takes in high level code and produces low level code. Converting C code to assembly code is compiling. Converting assembly code to machine code is also compiling. The specific name for the compiler which converts assembly code to machine code is an *assembler*. Hence: an assembler is a form of compiler, so we are allowed to use the words fairly interchangeably unless we need to be specific.
Assembling is the process of taking human-readable assembly code and converting it into machine code instructions which the CPU can understand and carry out. The program we will use for assembly is `arm-none-eabi-as`. There are two things which we need to write in order for the assembler to run happily. The first is the source code file, typically called main.s. The second is the terminal command which launches the assembler. We will now deal with those individually.

5.1 Source Code File

There are two different sorts of lines of code which you will type: assembly instruction and compiler directives.

5.1.1 Instruction

The simplest to understand are the assembly instructions. These are instructions which we look up in the programming manual for the CPU, figure out the format we should write them in and then type them into the source file. The assembler then takes these instructions in our source file, interprets them and hence figures out what sequence of bits represent the instruction. The machine code produced by the assembler is laid out in memory with each instruction coming sequentially after the next.

5.1.2 Assembler Directives

These are also referred to as compiler directives.

Compiler directives are not CPU instructions. They are not converted to machine code, and they are not executed by the CPU. Rather, they are lines of code which tell the compiler to do something, or provide the compiler with information about how we want the output to be structured or formatted.

The general GCC assembler directives are documented at https://sourceware.org/binutils/docs/as/Pseudo-Ops.html#Pseudo-Ops. The ones which are interesting to

us are:

- `.global` *`symbol`*: makes the symbol visible to the linker. A symbol is usually a label. The linker expects the symbol *_start* to be made available to it, as that's what it uses to define the program entry point.
- `.word` *`<word>`*: places whatever 32-bit word we type after it directly into memory. A word which is placed in memory like this is called a *literal* - it is binary data which is not interpreted at all by the compiler.

The assembler directives specifically for ARM are documented https://sourceware.org/binutils/docs/as/ARM-Directives.html#ARM-Directives. The ones of interest to us are:

- `.2byte` or `.4byte`: Similar to the .word directive, except now we have the option to place a full word (4 bytes) or half a word (2 bytes).
- `.syntax` *`<type>`*: Allows us to specify the type of assembly syntax. The default is the old *divided* type where ARM and Thumb instructions have their own syntax. The new *unified* type is what we will be using.
- `.cpu` *`<cpu type>`*: specifies which CPU we should be compiling for. The assembler knows which instructions are supported by which CPU, and hence can throw errors if we try to use instructions which do not exist for our CPU. The CPU we're compiling for is the *cortex-m0*.
- `.thumb`: specified that the our instructions should be compiled to the Thumb instruction set. This directive is probably not strictly necessary as we have already specified the CPU type which only supports the Thumb instruction set, but it's probably best to keep it for clarity.

5.1.3 Labels

Labels are one of the most useful abstractions provided by the assembler. They give you the ability to gave a name to the address of some data, and then refer to that address by name in future. The documentation of labels is https://sourceware.org/binutils/docs/as/Labels.html#Labels. One can label either instructions or literals. If we wanted to use PC-relative addressing without labels, we would have to manually calculate the difference between the memory address of the instruction trying to access data and the data being accessed, and supply that difference value as an operand to the data. This would mean that every time you changed the order of instructions or added new instructions to a program you'd have to re-calculate offsets.

It's important to note that the labels to not in any way make it into the microcontroller memory. They are only used to by the assembler to calculate offsets or to name memory addresses. When the code as been assembled, the labels have been replaced with actual numerical values.

5.2 Command Line Arguments

The assembler is launched with the command

```
$ arm-none-eabi-as
```

After the command one is able to supply arguments to it. If you type `-help` as an argument, all possible arguments and formats are listed. You'll see from this that the general format is:

```
$ arm-none-eabi-as [option...] [asmfile...]
```

Where [asmfile] is the input file which must be assembled and [opttion...] is a list of options describing how the assembly must take place. The options which are interesting to us are:

- `-o <filename>`: specified the output filename. Something like main.o is generally good.
- `-a=<filename>`: this optional flag generates a listings file with the name <*filename*>. The listings file is useful for debugging as is shows source code, machine code and addresses all in one.
- `-g`: this option seems poorly documented. As far as I can see it causes additional debugging info to be placed into the object files, which in turn makes its way into the elf file. This information is necessary for GDB to provide us with useful feedback on where our program is in execution. The option seems to be enabled by default on Windows, but is disabled by default on Linux. I suggest you include it in your command line argument, in case the defaults change.
- `-mcpu=<cpu type>`: specifies the CPU which our instructions should be compiled for, just like the `.cpu` directive.
- `-mthumb`: instructions the compiler to use the Thumb instruction set, just like the `.thumb` directive.

The general options are documented here: https://sourceware.org/binutils/docs/as/Invoking.html#Invoking. The ARM specific options are documented here: https://sourceware.org/binutils/docs/as/ARM-Options.html#ARM-Options.

The `-g` Note that the CPU type and instruction set (Thumb) can either be specified as a command line argument to arm-none-eabi-as or as an assembler directive in the source file.

When you run the assembler, it will provide no output if it assembles cleanly. If there are errors or warnings during assembly they will be printed to the terminal. Examine them closely and fix them.

A suggested command for assembling is as follows, where `main.s` and `main.o` should be replaced with whatever source code file name you have and whatever object code file name you desire.

```
$ arm-none-eabi-as -g -a=main.lst -o main.o main.s
```

6 Linking

The process of linking take the object code produced by the assembler and turns it into executable code. Object code is machine code with *relocatable* addressing. Relocatable means that the actual memory addresses of instructions or literals have not yet been defined. After all, the assembler has no clue where the flash memory on our STM32F051C6 device starts, so does not know where the code should be placed.

Most (all?) of what we type in our assembly source code files goes into a *segment* called *text*. I'm not sure the history of how it got the name text, but it probably has something to do with how it contains instruction which are *read* by the CPU (ie: the CPU reads and understand text). No matter the history of the name of this segment, the fact remains that we have to define where it must go it memory. If you look at the `main.lst` file produced by the assembler, you will see that all of the relative addresses are relative to the start of text. By defining where the text segment should start, we define all of the addresses, and hence produce an elf file where each byte has a defined destination address in the microcontroller memory.

If you run the command below, it will print out all of the (very) many options which the linker can accept.

```
$ arm-none-eabi-ld --help
```

Much like the assembler, the format for calling the linker is to specify options and then the input file name. The options which are of interest to us are:

- `--verbose`: if we call the linker with this flag, it will print the entire default linker script which it uses to link the source file. Most of this is unnecessary for us to know about, but take a look at the 5th line of the script: you'll see `ENTRY(_start)`. This is where the linker defines the entry point, and explains why we have to make the symbol _start available to it.
- `-o <filename>`: defines the output file name. Something like `main.elf` is generally a good file name.
- `-Ttext <address>`: specifies the address where we want the *text* segment to be placed. This should be at the start of flash.

As with the assembler, if the linker prints nothing to the terminal it exited happily. If it prints errors, read them carefully and try to correct them.

With the above options in mind, a suggested command for linking is:

```
$ arm-none-eabi-ld -Ttext 0x08000000 -o main.elf main.o
```

7 Debugging

7.1 GDB Commands

This section will deal with the commands which we can supply to GDB to gain insight into our program, and also manipulate it.

1. `step`: runs a single line of code. Can be shortened to `s`.
2. `continue`: causes the debugger to releases control of the CPU and allows the CPU to execute the code freely. The debugger will regain control if the CPU hits a breakpoint, or if you press Ctrl+C. Can be shortened to `c`.
3. `monitor` *`command`*: passes *command* directly to openOCD. This allows non-GDB commands to be used. An example of this is the command `reset halt`. GDB has no concept of a reset, but openOCD does: it pulls the NRST line low for a few milliseconds.
4. `info registers`: prints the names and values of all CPU registers. Can be shortened to `i r`.
5. `info registers rX`: where X is a number 0 - 15. Prints just the value of rX. Can be shortened to `i r rX`.
6. `set $rX=`*`val`*: where X is a register number 0 - 15 and *val* is some numerical value. Sets the contents of RX to hold the value *val*.
7. `x/`*`format address`*: reads the contents of *address* and displays it according to *format*. There are many format options. See the detailed documentation here: https://sourceware.org/gdb/current/onlinedocs/gdb/Memory.html#Memory. For example: `x/1xw 0x08000000` will display one word at the start of flash as a hex number.

8 C Debugging

8.1 Querying Defined Variables

C has a concept of variables, which assembly does not. In assembly we debug by reading from or writing to arbitrary memory addresses or CPU registers. In C, we need to be able to read/write to the memory addresses which the compiler has allocated for our variables. GDB makes this very easy to do.

This section deals with how we can check which variables are in scope. This is a quick way to get an overview of what variables exist and what values they have.

There are essentially three classes of variables which we need to have the ability to inspect: global variables, local variables and variables passed as arguments.

8.1.1 Global Variables

The following GDB command prints out all variables which are in scope but defined outside of the function. For our purposes this is essentially the same as global variables. The global variables are listed in the "All defined variables:". Some other symbols defined in the startup code or the linker script in the "Non-debugging symbols".

```
info variables
```

8.1.2 Local Variables

There are variables which are defined inside the function. They can be shown with the following command:

```
info locals
```

8.1.3 Arguments

Arguments are values passed in as parameters when the function is called. They can be shown with:

```
info args
```

8.2 Getting More Info On Or Modifying Variables

8.2.1 Types

If you want to check what type a variables is, the following command will print out the type, where 'var' is some variable name.

```
whatis var
```

8.2.2 Dereferencing Pointers

Often you will have variables of type pointer and you may want to inspect the data which they are pointing to. You could of course just query the value which the pointer holds with one of the above commands and then use the `x` command to get the contents of that memory address, but that's a cumbersome.

A more elegant way to do it would be with C-style language. Assuming you had a pointer *foo_ptr* and wanted to see what it was pointing to, you could use:

```
print *foo_ptr
```

This is great as it's the same syntax as we'd use is C for dealing with pointers. Similarly, if you wanted to check the memory address of a variable, you could use the reference operator to get address of a variable:

```
print &foo
```

If you just wanted to see the value of the variable you could of course do:

```
print foo
```

8.2.3 Modifying Variables

There are occasions where you may want to alter the value of a variable

8.3 Flow Control

Much of the flow control which GDB offers in C is the same as in assembly. We can still step, break and continue.

Additional flow control which C has is now discussed.

8.3.1 Next

If the line of code about to be run is a function call and you do a `step`, execution then continues inside the function. This is often called a 'step into' instruction. This is what you want to do if you want to see what's going on inside the function. If, however, you are not interesting in the function you can 'step over' it with the `next` instruction. The function will still be executed, but you won't have to step through each line of it. It's equivalent to setting a breakpoint on the line after the function call and then allowing the code to free run.

8.3.2 Finish

Assuming you're in a function which returns at some point, GDB can be instructed to free run until that function returns with the `finish` command.

This is very useful if you just want to check something in a function. After checking the thing you're then done with the function and want to skip over the rest of the code until the function returns.