Lab 0: Getting Started

14-642 Fundamentals of Embedded Systems

Due: September 20th, 2016

Contents

1 Introduction

1.1 Overview

The goal of this lab is to set up the environment for the class and ensure that you can compile, load and debug code for the Raspberry Pi. You will learn about the Raspberry Pi boot sequence, runtime and how to cross compile and develop with a standard GNU GCC toolchain.

1.2 Grading and Hand-in

You will need to come in to office hours to pick up the raspberry pi and accompanying equipment. For this lab, you will demo your ability to use the hardware and GDB to a TA during office hours before midnight on the due date in order to get checked off.

We have provided a GitLab Repository that each member of the class can request access to via Shibboleth signin at `https://embedded.andrew.cmu.edu:5443`. Use this repository for getting starter code and source control during the labs and submission. More instructions for Gitlab can be found in the accompanying document Gitlab.pdf.

The points breakdown for your responsibilities in this lab are as follows: 10 points for demoing GDB, 5 points for showing boardcheck, 35 points for the assembly challenge. Use this as an opportunity to refresh your gdb debugging skills.

2 Development Board

For the course, you will be using a lab kit containing a Raspberry Pi 2 and two custom add-on boards. The programming board provides a JTAG debugging interface and a serial terminal interface. The I/O expansion board provides an ADC, light sensors, status LEDs (x10), speaker, microphone, servo outputs (x4), an Audio Codec (192 Khz), keypad input buttons, audio input/headphone output, a two channel H-bridge, encoder input (x2), an onboard rotary encoder, EEPROM, camera interface, 9D0F IMU with an Accelerometer, Rate Gyro, Magnetometer, and Air pressure sensor. The schematic for this board is available docs/Raspberry-Pi-B-Plus-V1.2-Schematics.pdf. You will use this hardware platform for the rest of the labs in this course, so be careful during the assembly process.

2.1 Raspberry Pi Hardware Overview

Before you run the code baremetal, you should understand the nuances of your hardware. The chip is a Broadcom BCM2836 system on-chip (SoC), which contains the following:

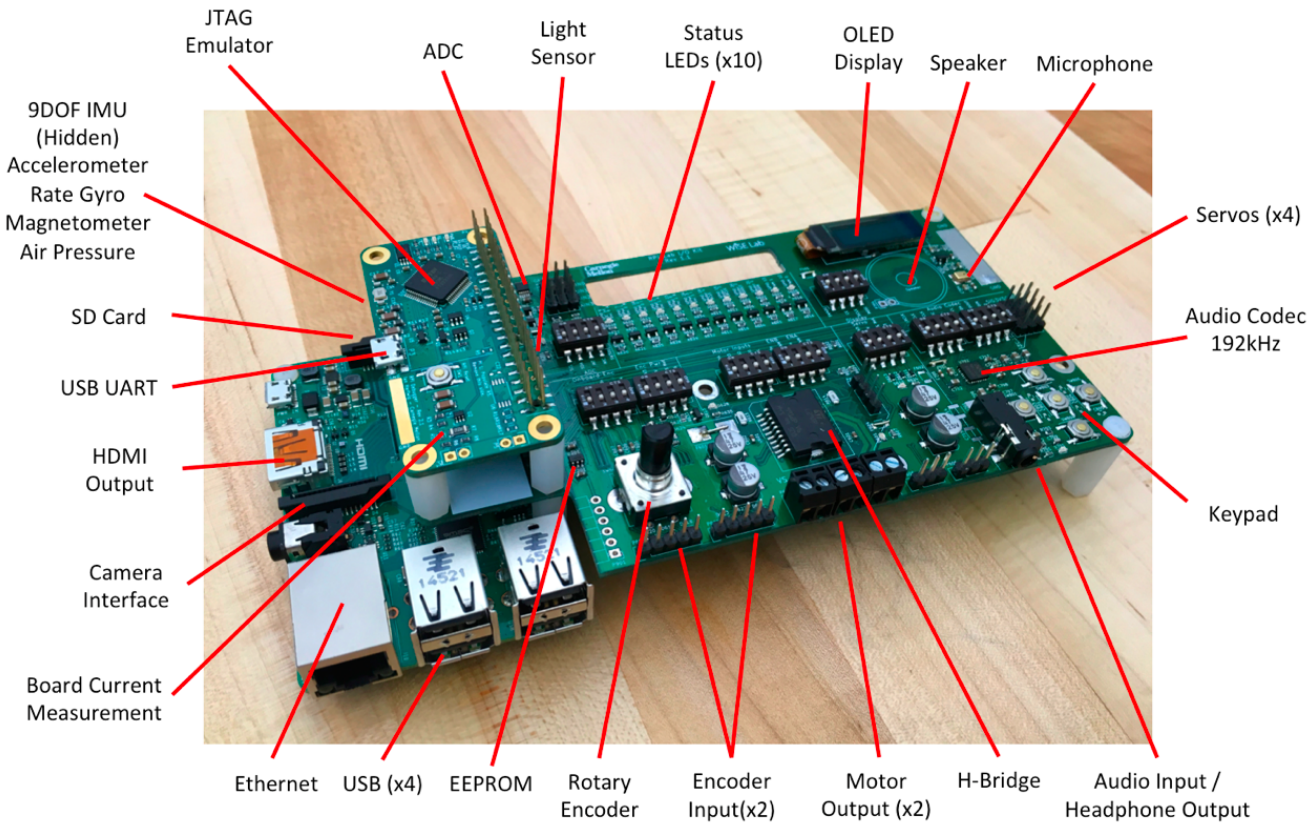
- CPU: 700 MHz quad-core ARM Cortex A7 *(ARMv7 instruction set)*
- GPU: 250 MHz Broadcom VideoCore IV
- Memory: 1 GB (shared with GPU)

The Raspberry Pi also has the following hardware peripherals:

- 4 USB 2.0 ports (via the on-board 5-port USB hub)
- 1 15-pin MIPI camera interface (CSI) connector
- 10/100 Mbit/s Ethernet USB adapter on the third/fifth port of the USB hub
- 3.5mm phone jack for analog audio output
- 1 HDMI port with resolutions from 640350 to 19201200 and digital audio out

- 1 MicroSD slot

The Raspberry Pi also contains a General Purpose IO (GPIO) header which our breakout board exposes. This breakout deals with ensuring that the communication peripherals from the ARM cores (SPI and UART. *you will learn about these later!*) are broken out easily and connected to the correct sensors/pins for you.



3 Toolchain

To build programs for this class you will need an ARM cross-compiling toolchain. This will allow you to build machine code for ARM from an x86 machine. You will need to install GCC, GDB, OpenOCD, and a serial terminal emulator.

Tool	Purpose
GCC	Compiler, assembler, & linker
GDB	Interactive debugger
OpenOCD	GDB server & JTAG on FTDI interface A
FTDITerm	Access to RPi UART on FTDI interface B

To simplify setup, we have prepared a Linux VM with the tools you'll need that can be run in VMWare Workstation 11 (Windows) or Fusion 7 (Mac). This software is available through the CMU license program to you for free. We suggest using VMware because we have seen issues with the JTAG interface on VirtualBox. We have also verified that Parallels works.

If you are using a Linux host or interested in installing the toolchain natively on your machine, it is available for Mac, and Linux. Please see Appendix A.

3.1 Virtual Machine

Download and install VMWare Workstation 11 (Windows) or Fusion 7 (Mac) from the VMWare Campus Webstore: `https://www.cmu.edu/computing/software/all/vmware/index.html`. From the webstore homepage find the **Software** tab to find Workstation or Fusion.

Download the VM image from: `http://embedded.andrew.cmu.edu/ECE%20349%20VM.vmwarevm.zip`

This link can also be found on the course website.

Open the VM with Workstation or Fusion. It is Ubuntu 14.04 (LTS) 32-bit with the guest additions installed. It has the Linux setup described below complete. The default user below has sudo privileges, but the root user is not set up. If you need root, you can use `sudo su`.

Username: `ece349`

Password: `password`

When you boot the VM, we recommend not upgrading since it may change toolchain component versions.

Your VM should be configured to use your host machine's network connection. This means you can ssh, scp and run applications like a web browser to download and install content. If you are interested in installing the toolchain on a bare operating system (not within a VM), we have instructions below.

4 Startercode

At this point, you have the toolchain setup and the hardware breakout completed. Now we can compile code for the Raspberry Pi.

It is now time to clone a copy of the code from the course Gitlab Repository. You can request access via Shibboleth signin at `https://embedded.andrew.cmu.edu:5443/`. More instructions for Gitlab can be found in the accompanying document `Gitlab.pdf`. You should clone the 642-TAs/642-f16.

The startercode you received for this lab should look as follows (**Note**: you don't need to know what every file does at the moment):

```
lab0_handout/
  lab0.pdf           -> this file
  sdcard/
    kernel.img       -> JTAG kernel
    start.elf        -> third-stage bootloader
    bootcode.bin     -> second-stage bootloader
  code/
    Makefile         -> Makefile for the kernel
    349libk/
      ...            -> (TA written source used by kernel.c)
    349util/
      doxygen.conf   -> Doxygen config file (not necessary for this lab)
      kernel.ld      -> linker script
      rpi2.cfg       -> openocd JTAG configuration file
      init.gdb       -> GDB startup script commands
    kernel_blink/
      kernel.c       -> simple kernel that blinks on-chip leds
      config.mk      -> config file used by Makefile
    boardcheck/
      kernel.elf     -> TA written kernel used to check board connections
  docs/
    arm_arch_ref.pdf -> ARM v7 architecture reference
    arm_isa_ref.pdf  -> ARM assembly reference
    BCM2835.pdf      -> documentation of Broadcom SoC
    BCM2836.pdf      -> documentation of changes between BCM2835 and BCM2836
    gdb_cmd_ref.pdf  -> quick reference for GDB commands
    gnu_arm_ref.pdf  -> overview of using GNU ARM assembler
    rpi_breakout.pdf -> EAGLE schematic of the breakout board you made
```

Now that you have the toolchain installed, compile the code yourself. To compile the code run `make` inside the `code/` directory. To delete generated files run `make clean`. The makefile should have generated files called `kernel.asm`, `kernel.img`, and `kernel.elf`.

The startercode is an infinite loop that blinks the 2 on-chip LEDs. This code is in `kernel_blink/kernel.c`. `kernel.asm` is the assembly dump produced by running objdump on `kernel.elf`. Feel free to change the blinking pattern in `kernel.c`.

5 Raspberry Pi Boot Process

Now that you can compile the code, lets figure out how we get to your code when the Pi boots up. This is called the *boot process* of the Raspberry Pi. Here is a step-by-step of how your code is loaded after turning on the Pi.

1. When you power on your Raspberry Pi, the first bits of code run are stored in a small ROM chip inside the SoC (System on a Chip). This code is therefore unmodifiable and unreadable. This is the called the first-stage bootloader. At this point, the ARM cores are in RESET mode since they have not been initialized.

2. This first-stage bootloader can read the FAT32 filesystem on the MicroSD card that you attach to the Pi. The code looks for a file called `bootcode.bin` on the SD card which contains the second-stage bootloader.

3. This second-stage bootloader is provided by Broadcom as well and is loaded into the L2 cache of the GPU and then executed by the GPU. This code enables the on-chip RAM. Lastly, this code looks in the SD card for `start.elf`, which contains the firmware for the GPU. `start.elf` is also written by Broadcom.

4. `start.elf` is the third-stage bootloader. It sets up the GPU and splits up RAM such that it is shared between the GPU and the ARM cores (more about this in a later lab). Lastly, it looks on the SD card for `kernel.img`. This file is what contains the code you wish to run on the Raspberry Pi. It is loaded at 0x8000 by `start.elf` as defined by the Broadcom datasheet. Once `kernel.img` is loaded into RAM at 0x8000, the GPU releases the RESET mode on **only one** of the ARM cores and allows that core to run the code loaded at 0x8000.

On another note, a good rule to remember for the Raspberry Pi is ***the GPU is king***. Since the GPU is what boots up first, the Pi is essentially unbrickable (No, this is not a challenge). As long as you are not putting unexpected voltages at various IO pins, the Pi can always come back from some weird state by just turning it off and back on again.

NOTE: This process is not how you will be loading your kernel usually. Instead we will use JTAG and GDB to flash over the kernel you compile. We will walk through this in the next section.

6 Running Code and Debugging with JTAG and GDB

Now we will see how to debug a running CPU using JTAG.

6.1 What is JTAG

To debug on the Pi, we will use JTAG (Joint Test Action Group). This is a debugging method developed in 1985 to test PCBs (Printed Circuit Boards) after they were manufactured. This is the most common interface for debugging embedded processors and is supported by most modern systems. JTAG is also commonly used to load firmware onto new devices. To interpret the JTAG commands, we will use openocd. Openocd is an open-source on-chip debugger. You should not need to interact with openocd directly in this course so don't worry about its internal commands.

6.2 Setting Up Your SD Card

To make sure your SD card is in a clean state, you should reformat it before loading any code. You only need to do this once.

1. Insert the SD card into the USB to SD card reader and plug the reader into your computer. (or if your computer has an SD card reader, use that directly).

2. Next, you need to reformat the SD card to FAT32. On the VM you can use GParted to do this. Otherwise, Google for instructions on how to do this with your respective OS.

3. Once formatted, your SD card should be completely empty. Copy over all files from `sdcard/` onto the SDcard. These should include:

   - `bootcode.bin`
   - `start.elf`

- `kernel.img`

The `kernel.img` on the SD card is a TA written kernel that enables the JTAG interface on the Pi. You should never have to change the `kernel.img` on the SD card unless you want your own kernel to run baremetal without debugging capabilities (not recommended).

6.3 How to Start GDB Debugging

Now plug the Pi in with the formatted SD card inserted. When you plug the Pi in, the TA JTAG kernel enables JTAG and hangs waiting for you to flash over your kernel using GDB. To do this, follow these steps:

1. From the `code/` directory run `make openocd` (if you are on Linux or the VM, you'll need to run `sudo make openocd` instead). You should see output like the following:

```
... (some lines omitted)
Info : clock speed 1000 kHz
Info : JTAG tap: rpi2.dap tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : rpi2.cpu0: hardware has 6 breakpoints, 4 watchpoints
Info : ttbcr 0ttbr0 3f377b7bttbr1 dbfbb45b
Info : rpi2.cpu0 rev 5, partnum c07, arch f, variant 0, implementor 41
Info : number of cache level 2
Error: cache l2 present :not supported
Info : rpi2.cpu0 cluster f core 0 multi core
Info : rpi2.cpu1: hardware has 6 breakpoints, 4 watchpoints
Info : ttbcr 0ttbr0 14b76369ttbr1 2ab99d79
Info : rpi2.cpu1 rev 5, partnum c07, arch f, variant 0, implementor 41
Info : number of cache level 2
Error: cache l2 present :not supported
Info : rpi2.cpu1 cluster f core 1 multi core
Info : rpi2.cpu2: hardware has 6 breakpoints, 4 watchpoints
Info : ttbcr 0ttbr0 b536db5bttbr1 467e7ccb
Info : rpi2.cpu2 rev 5, partnum c07, arch f, variant 0, implementor 41
Info : number of cache level 2
Error: cache l2 present :not supported
Info : rpi2.cpu2 cluster f core 2 multi core
Info : rpi2.cpu3: hardware has 6 breakpoints, 4 watchpoints
Info : ttbcr 0ttbr0 7f9f535bttbr1 9eeebc3
Info : rpi2.cpu3 rev 5, partnum c07, arch f, variant 0, implementor 41
Info : number of cache level 2
Error: cache l2 present :not supported
Info : rpi2.cpu3 cluster f core 3 multi core
```

   Important: Any time you reset or unplug the Raspberry pi, you'll need to terminate openocd and restart it or else gdb will not work.

2. Then, in a new terminal window, run `make PROJECT=kernel_blink gdb` inside the `code/` directory to begin a GDB debugging session. You should see a standard gdb terminal with output like the following:

```
... (some lines omitted)
0x00008238 in printnumk ()
Loading section .text, size 0xb58 lma 0x8000
Loading section .rodata, size 0x11 lma 0x9000
Loading section .rodata.str1.4, size 0x18 lma 0x9014
```

```
Loading section .ARM.exidx, size 0x8 lma 0x902c
Start address 0x8000, load size 2953
Transfer rate: 15 KB/sec, 738 bytes/write.
0x00008004 in _start ()
(gdb)
```

This just loaded a GDB initialization script called `349util/init.gdb`. This script connects to the openocd JTAG session and loads the `kernel.img` (that **you** compiled) over to the board using JTAG. Then we step the processor once so you can see where you are in the kernel.

Thats it!

7 Demoing GDB to TAs

Note that due to an openocd+vm bug (detail below) we have disabled SIGINT (ctrl+c) in the VM.

You are now inside the startercode and should be at `_start` in `code/349libk/src/boot.S`. When you demo this lab, you will be required to perform the following steps using GDB:

1. Show us you can compile, load, and launch your own code through GDB with JTAG
2. Run `(gdb) layout split`
3. Run `(gdb) layout regs`
4. Now use `(gdb) si` to single step the program until you reach `kernel_main`
5. Set a breakpoint at `led_set_red` with `(gdb) b led_set_red`
6. Now continue until you hit the breakpoint with `(gdb) c`
7. Now print the value of `cnt` in `kernel/kernel.c` with `(gdb) print cnt`
8. Now to see all registers (including banked ones) run `(gdb) info all-reg`
9. Now set a breakpoint on the `cnt` variable that allows the leds to flash 10 times before it halts. Search the gdb documentation to learn how to break with a conditional on a particular file.

If any of these commands look unfamiliar to you then look at `docs/gdb_cmd_ref.pdf`. When you demo this lab, you will be asked the following questions:

1. What is the value of the CPSR at the first breakpoint?
2. At the first breakpoint, what mode is the processor in?

8 Checking Your Breakout Board and Using FTDITerm

Now that you can load kernels using GDB and debug them, you can test the UART on your board. Plug in the Pi and start your GDB debugging session as mentioned in section 6.3, instead of `make gdb` you need to run `make PROJECT=boardcheck gdb`.

1. `$ make openocd`
2. In a new terminal window `$ make PROJECT=boardcheck gdb`

3. In a new terminal window `$ sudo ftditerm.py -b 115200`
4. In the GDB window, run `(gdb) continue`
5. Look back at the window running `ftditerm.py`
6. You should see the following output:

```
opening first available ftdi device...
--- Miniterm on None: 115200,8,N,1 ---
--- Quit: Ctrl+]  |  Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Testing UART... Success
```

   If in doubt, ask a TA.

You will be required to show us this output during your lab check off.

9 GDB and Assembly Challenge

We now need you to demonstrate your GDB skills and your ability to understand ARM assembly.

9.1 Starting the challenge

Start your GDB debugging session as mentioned in section 6.3, and instead of running `make gdb` run the command `make PROJECT=gdbchallenge gdb`.

!!!Your main objective is to turn the LED green!!!

You would need to change the memory contents at 2 locations - `0xa000` and `0xa028`.

- At `0xa028` you would need to insert your andrew id. Your id needs to be exactly 8 characters, if it is less than 8 characters please append letters starting with 'a', then 'b', etc. For example, if your andrew id is "doc", you will need to insert your andrew id as "docabcde".
- `0xa000` you need to figure out this one on your own.

Note, that if there is no C source data, `stepi` will advance a single instruction.

10 Handin

Please write a step-by-step list of GDB commands required to complete the GDB Challenge into a textfile called `lab0_solution.txt` in your base directory (with `code`, `docs`, etc.). In order to learn the submission process, we want you to add this file to your repository, commit it and then tag that commit as your handin. Instructions for this can be found in the `gitlab.pdf` handout.

You should tag your final commit with `lab0-submit`.

10.1 Hints and Suggestions

1. Look out for functions `decode` and `compare`
2. You will be able to turn the red led to green, once you are able to figure out the correct content at `0xa000` corresponding to your andrew id.

3. Try and understand the logic of the problem challenge. Brute force will be frustrating!
4. Be patient! Debugging assembly is painful, especially when you do not have the corresponding C code.

11 FAQ and Errors

11.1 Warn : Invalid ACK 0x7 in JTAG-DP transaction

If you see this error, then the kernel you have on the SD card is most likely not the JTAG kernel we released. Re-copy the files from `sdcard/` to the SD card and see if the error continues.

11.2 Program received signal SIGINT, Interrupt. 0x000000c4 in ?? ()

This is an issue we have seen in the VM using openocd + gdb. We fixed the issue by ignoring all SIGINTs in gdb (in the `349util/init.gdb` script). This avoids interrupting the debugging session from extra signals sent by openocd, but it also means you can't stop the processor if you continue without setting a breakpoint.

The line we added is `handle SIGINT nostop noprint ignore`. If you see this problem not in the VM or with this line in your init script, please let us know!

Appendix A

Running the toolchain directly on a native host machine is usually faster then running a VM. Below you will find instructions for directly installing the toolchain.

Linux

First we need to install a few manual dependencies:

```
$ sudo apt-get install build-essential autoconf automake libtool libusb-dev libusb-1.0-0-dev
  libhidapi-dev git vim
```

The ARM GCC Embedded project supported by ARM provides precompiled binaries for linux. At the time of writing, 4.9-2015-q2 was the latest release. We can download the binaries as a tar, extract to a known directory (like `/`), fix the permissions, and add them to our path:

```
$ cd ~/
$ wget https://launchpad.net/gcc-arm-embedded/4.9/4.9-2015-q2-update/+download/gcc-arm-none-eabi-
  4_9-2015q2-20150609-linux.tar.bz2
$ tar -xf gcc-arm-none-eabi-4_9-2015q2-20150609-linux.tar.bz2
$ mv gcc-arm-none-eabi-4_9-2015q2 gcc-arm-none-eabi
$ chmod -R 755 ./gcc-arm-none-eabi
$ echo "export PATH=\$PATH:~/gcc-arm-none-eabi/bin" >> ~/.bashrc
$ source ~/.bashrc
```

(64-bit only) Note that these are 32-bit binaries. If you are on a 64-bit version of linux, you'll need to install the 32-bit runtime libraries:

```
$ sudo apt-get install lib32bz2-1.0 lib32ncurses5 lib32tinfo5 lib32z1 libc6-i386
```

To check that the tools are installed correctly, try to run `arm-none-eabi-gcc` and `arm-none-eabi-gdb`.

Next we will need to download and compile OpenOCD. We need to clone the repo and run a standard linux compile/install. For convenience, we have created a `/repos` folder in the home directory.

```
$ cd ~/
$ mkdir repos
$ cd repos
$ git clone git://git.code.sf.net/p/openocd/code openocd
$ cd openocd
$ ./bootstrap
$ ./configure
$ make
$ sudo make install
```

To check that OpenOCD is installed correctly, try to run `openocd --version`. You should see the compilation date as just a few minutes ago.

To access the UART on the FT2232's channel B we'll need a special serial console. It depends on libftdi and pylibftdi which can be found in apt/pip. We'll also add it to our path for convenience.

```
$ sudo apt-get install python python-pip libftdi1
```

```
$ sudo pip install pylibftdi
$ cd ~/repos
$ git clone https://github.com/ihartwig/ftditerm.git
$ echo "export PATH=\$PATH:~/repos/ftditerm" >> ~/.bashrc
$ source ~/.bashrc
```

Finally, in some cases we want the programs we added to the path above to be available when we use `sudo`. Normally, these get overridden by a `secure_path` variable. To disable this behavior, run `sudo visudo` and comment out the line that includes `Defaults secure_path=...`. You can test that the `secure_path` is disabled by trying to run FTDITerm with your development board plugged in:

```
$ sudo ftditerm.py -b 115200
opening first available ftdi device...
--- Miniterm on None: 115200,8,N,1 ---
--- Quit: Ctrl+]  |  Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---

--- exit ---
```

Mac

Although there are a couple different package managers, we found the packages we needed in Homebrew. You can find installation instructions at `http://brew.sh/`. Homebrew will also require that you install the Xcode Command Line Tools, which we will need later.

The ARM GCC Embedded (`https://launchpad.net/gcc-arm-embedded`) project supported by ARM provides precompiled binaries for Mac. At the time of writing, 4.9-2015-q2 was the latest release. Download the latest Mac .tar.bz2 package. We can then extract the binaries and add them to our path:

```
$ cd ~/Downloads
$ tar -xf gcc-arm-none-eabi-4_9-2015q2-20150609-mac.tar.bz2
$ mv ./gcc-arm-none-eabi-4_9-2015q2 ~/gcc-arm-none-eabi
$ echo "export PATH=$PATH:~/gcc-arm-none-eabi/bin" >> ~/.bashrc
$ source ~/.bashrc
```

To check that the tools are installed correctly, try to run `arm-none-eabi-gcc` and `arm-none-eabi-gdb`.

We also need to install openocd, the glue between our debugger hardware and gdb. It is available in homebrew:

```
$ brew install openocd
```

To access the UART on the FT2232's channel B we'll need a special serial console. It depends on libftdi and pylibftdi which can be found in apt/pip. We'll also add it to our path for convenience.

```
$ brew install python libftdi
$ pip install pylibftdi
$ cd ~/
$ mkdir repos
$ cd repos
$ git clone https://github.com/ihartwig/ftditerm.git
$ echo "export PATH=\$PATH:~/repos/ftditerm" >> ~/.bashrc
$ source ~/.bashrc
```

Finally, we can try to run FTDITerm with your development board plugged in:

```
$ ftditerm.py -b 115200
opening first available ftdi device...
--- Miniterm on None: 115200,8,N,1 ---
--- Quit: Ctrl+]  |  Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---

--- exit ---
```

Windows

Not supported. Use the VM. We had issues running the UART and JTAG simultaneously.