

# Lab 1: Bootstrapping

*18-349 Students..... Assemble!*

---

*18-349 Introduction to Embedded Systems*

Code Due: 5:00PM EST Tuesday, 19 September 2023

All Demos Due: Tuesday, 26 September 2023

# Contents

1	Introduction and Overview . . . . .	3
1.1	Goal . . . . .	3
1.2	Task List . . . . .	3
1.3	Grading . . . . .	3
2	Getting your hardware . . . . .	4
3	Setting up Development . . . . .	4
3.1	Virtual Machine (Recommended) . . . . .	4
3.2	Installing the Toolchain (Optional) . . . . .	5
4	STM32 . . . . .	5
4.1	STM32 Boot Process . . . . .	5
5	Testing your board . . . . .	7
5.1	Note: tmux/terminator . . . . .	7
5.2	How to Start GDB Debugging . . . . .	7
5.3	Checking Your Board . . . . .	8
6	Bootloader . . . . .	9
6.1	Linker Layout . . . . .	10
7	ARM Optimization . . . . .	12
7.1	Tips For Optimization . . . . .	13
8	In Lab Demo . . . . .	13
9	Submit to GitHub and Canvas . . . . .	14
10	Minicom Setup . . . . .	14
11	GDB Tips & Tricks . . . . .	15

# 1 Introduction and Overview

Welcome to the amazing world of 18-349! Over the next few weeks, you are going to bring up an embedded system from scratch, bootstrap it, write device drivers, interrupt handlers, and then use a real time operating system (FreeRTOS). Excited yet? Let's dive in!

## 1.1 Goal

The goal of this lab is to introduce the development tools you will use all semester, as well as to help you understand the initial stages of operations on your embedded CPU.

You will set up the environment and ensure that you can compile, load, and debug code. You will learn how to write a simple boot loader for the STM32. You will then use the provided timer to measure performance and see the results of optimizing ARM assembly code.

## 1.2 Task List

1. Get the equipment from the ECE Course Hub
2. Submit a link to your repo in Canvas
3. Download and run the VM
4. Run the boardcheck
5. Implement the bootloader
6. Do the ARM optimization challenge.
7. Create an issue on Github with your name, andrew id, and link to the commit you wish to submit.

## 1.3 Grading

The point breakdown for this lab is:

Task	Points
Demo Bootloader (Section 6)	10
assembly optimization	35
2x faster	5
5x faster	15
10x faster	30
15x faster	35
30x faster(bonus)	40
Submission Protocol (Section 9)	5
TOTAL	50

## 2 Getting your hardware

You will need to go to the ECE Course Hub to pick up the STM32 and accompanying equipment.

You should receive:

- an STM32 and a USB b-type cable
- a PCB (maybe, not needed for Lab 1)
- a Motor and an adapter (possibly, also not needed)
- a servo (perhaps. Again, not needed)
- possibly some other stuff (ditto)

## 3 Setting up Development

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 another 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
Minicom	For access to UART

To simplify setup, we have prepared a Linux Guest VM with the tools you'll need that can be run in VMWare Workstation 16 (Windows) or Fusion 12 (Mac, x86 ONLY). This software is available through the CMU license program.

### 3.1 Virtual Machine (Recommended)

Download and install VMWare Workstation 15 (Windows/Linux) or Fusion 12 (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: <https://drive.google.com/file/d/1UcmPGuLep2L4x9MBAv4fmFUflFeL637/view?usp=sharing>

Open the VM with Workstation or Fusion. It is Ubuntu 16.04.06 (LTS) The default user below has sudo privileges, but the root user is not set up. If you need root, you can use `sudo` to run a single command as root, or `sudo -s` to get a root shell.

- Username: `vm349`
- 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.

## 3.2 Installing the Toolchain (Optional)

It is possible to install the toolchain on Macs with Apple Silicon and Linux machines. The course staff has provided some hints on this (below), but are not bound to support any setup other than the provided VM. If you have any issues with the steps below, please see a TA.

For Macs with Apple Silicon and Ubuntu, here's what worked for me:

1. **For Apple Silicon:** Install homebrew (<https://brew.sh/>) and xcode commandline tools (run `xcode-select --install`). Then run

```
brew tap osx-cross/arm
brew install md5sha1sum minicom arm-gcc-bin openocd doxygen
```

**For Ubuntu:** Try <https://askubuntu.com/questions/1243252/how-to-install-arm-none-eabi-gdb-on-ubuntu-20-04-lts-focal-fossa> and `sudo apt-get install -y openocd minicom doxygen`

2. Make sure

```
arm-none-eabi-gcc --version
arm-none-eabi-gdb --version
arm-none-eabi-objcopy --version
arm-none-eabi-objdump --version
```

throws no errors.

3. Make sure `./osx_arm.o cd` or `./linux.o cd` throws no errors.

## 4 STM32

In your VM, you will `git clone` your repo. For the purpose of this lab, the only files you will be editing are:

```
kernel/asm/boot.S
kernel/asm/optimize_me.S
```

### 4.1 STM32 Boot Process

To debug on the STM32, we will use JTAG (a testing interface defined by the Joint Test Action Group). This is a debugging method developed in 1985 to test PCBs (Printed Circuit Boards) after they are 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.

The OpenOCD client runs a telnet server on your localhost that allows various applications to communicate with the debug target via standard network protocols.

1. For this class, OpenOCD is setup to run servers at ports 3333 and 4444 for the flash programmer and gdb respectively.
2. The flash programmer is able to connect to the port and write the raw program binary image straight into flash over serial.

3. Upon coming out of reset, the board looks at the first two entries in the exception vector table, the first value is loaded into sp and the second value is loaded into pc. Thus on reset, it effectively jumps to whatever is in the second entry this special piece of code is the reset handler, otherwise known as the boot loader and is your first task.

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

## 5 Testing your board

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

### 5.1 Note: tmux/terminator

Since you will be using 3 terminal windows to run all of the code in this class, it will make your life much easier to use something like tmux, which is a terminal multiplexer that allows you to split a single terminal window into multiple panes, allowing you to have all 3 of the terminal windows (GDB, OpenOCD, and minicom/cat) open at once. You can install this with `$ sudo apt-get install tmux` on Linux. See this tmux reference for some helpful commands: <http://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/>.

If you'd prefer an alternative that's less heavy on remembering magic button combinations, you can try terminator, it has the same functionalities as tmux but a clean interface that just requires you to right-click to split panes. You can install it via - `$ sudo apt-get install terminator`

### 5.2 How to Start GDB Debugging

Connect the STM32 to your laptop via the provided cable. then follow the following steps:

1. `cd` to the root directory of the repo. If you are using Windows (and thus, aren't using the VM), just double click the file windows\_ocd.bat. If you are in linux (VM or otherwise), use `sudo ./linux_ocd`:

```
Open On-Chip Debugger 0.10.0 (2018-11-30) [https://github.com/sysprogs/openocd]
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control.
The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v28 API v2 SWIM v17 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.237795
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
```

**Important:** To end a debugging session, just send SIGINT (ctrl+c), that will halt the STM32. If you wish to start debugging from the beginning, just type **reset** and hit enter. If you wish to exit gdb type **exit** or press EOF (ctrl + d).

- Then, in a new terminal window, run `make flash` to begin a GDB debugging session. You should see a standard gdb terminal with output like the following:

```
*****
Flashing kernel_default_642fb6bd6154557633e19751ec3cddaf to board...
*****
Open On-Chip Debugger
> reset halt
  Unable to match requested speed 2000 kHz, using 1800 kHz
  Unable to match requested speed 2000 kHz, using 1800 kHz
  adapter speed: 1800 kHz
  target halted due to debug-request, current mode: Thread
  xPSR: 0x01000000 pc: 0x08000198 msp: 0x20005000
> flash write_image erase build/bin/kernel_default_642fb6bd6154557633e19751ec3cddaf.bin 0x08000000
  auto erase enabled
  device id = 0x10016433
  flash size = 512kbytes
  target halted due to breakpoint, current mode: Thread
  xPSR: 0x61000000 pc: 0x20000044 msp: 0x20005000
  wrote 16384 bytes from file
  build/bin/kernel_default_642fb6bd6154557633e19751ec3cddaf.bin in 0.599541s (26.687 KiB/s)
  (...) some lines omitted
(gdb)
```

This just loaded a GDB initialization script called `util/init.gdb`. This script connects to the OpenOCD JTAG session and loads the binary image you compiled over to the board using JTAG. Then we step the processor once so you can see where you are in the kernel.

### 5.3 Checking Your Board

Now that you can load binaries onto the board debug them, you can test the UART on your board.

- Run `$ sudo ./linux_ocd` or `windows_ocd.bat` or `./osx_arm_ocd`
- In a new terminal window, run `$ make flash`
- In an another new terminal window, start minicom (see section 10). Or you can run `$ cat /dev/serial/by-id/[really long name]` (you don't have to type the whole thing, just get to `by-id` and press `tab`). In Windows, you can double click `windows_serial`.
- In the GDB window, run `(gdb) continue`
- You should get a message in gdb saying that a hard fault was detected. To fix this, in `boot.S` update the first entry in the vector table to `__stack_top` and the second to `__reset`. `make flash` and continue again.
- In the serial terminal you should see the following output:

```
Entered kernel_main, starting boot loader test

Failed to initialize global var
Make sure you copy all the global vars from flash to sram.
Boot Loader Failed
```



## 6 Bootloader

The STM32 nucleo board's memory map is as follows:

Reserved	0x2001 8000 - 0x3FFF FFFF
SRAM (96 KB aliased by bit-banding)	0x2000 0000 - 0x2001 7FFF
Reserved	0x1FFF C008 - 0x1FFF FFFF
Option bytes	0x1FFF C000 - 0x1FFF C007
Reserved	0x1FFF 7A10 - 0x1FFF BFFF
System memory	0x1FFF 0000 - 0x1FFF 7A0F
Reserved	0x0808 0000 - 0x1FFE FFFF
Flash memory	0x0800 0000 - 0x0807 FFFF
Reserved	0x0008 0000 - 0x07FF FFFF
Aliased to Flash, system memory or SRAM depending on the BOOT pins	0x0000 0000 - 0x0007 FFFF

For this exercise, we are interested in the Flash and SRAM sections. The compiler generates object(.o) files. However, these files cannot be uploaded directly to the board, they must be linked into a binary(.bin) file. During the linking step, you can specify a script which tells the linker where to place the object files and where to initialize stacks, heaps, etc. This linker file is located in util/linker\_template.lds and is discussed in the next section. The code and data is placed into the bin file and this file is uploaded to the board.

However, there are two caveats that we must consider:

1. The flash programmer can only write to the flash section and not to sram.
2. The processor cannot write to flash directly.

The problem that we must contend with is global variables and static variables, (variables that are stored in memory). They must be in SRAM for the processor to write to them, however, they cannot be initialized in SRAM via the flash programmer (which is the only way to get data onto the board).

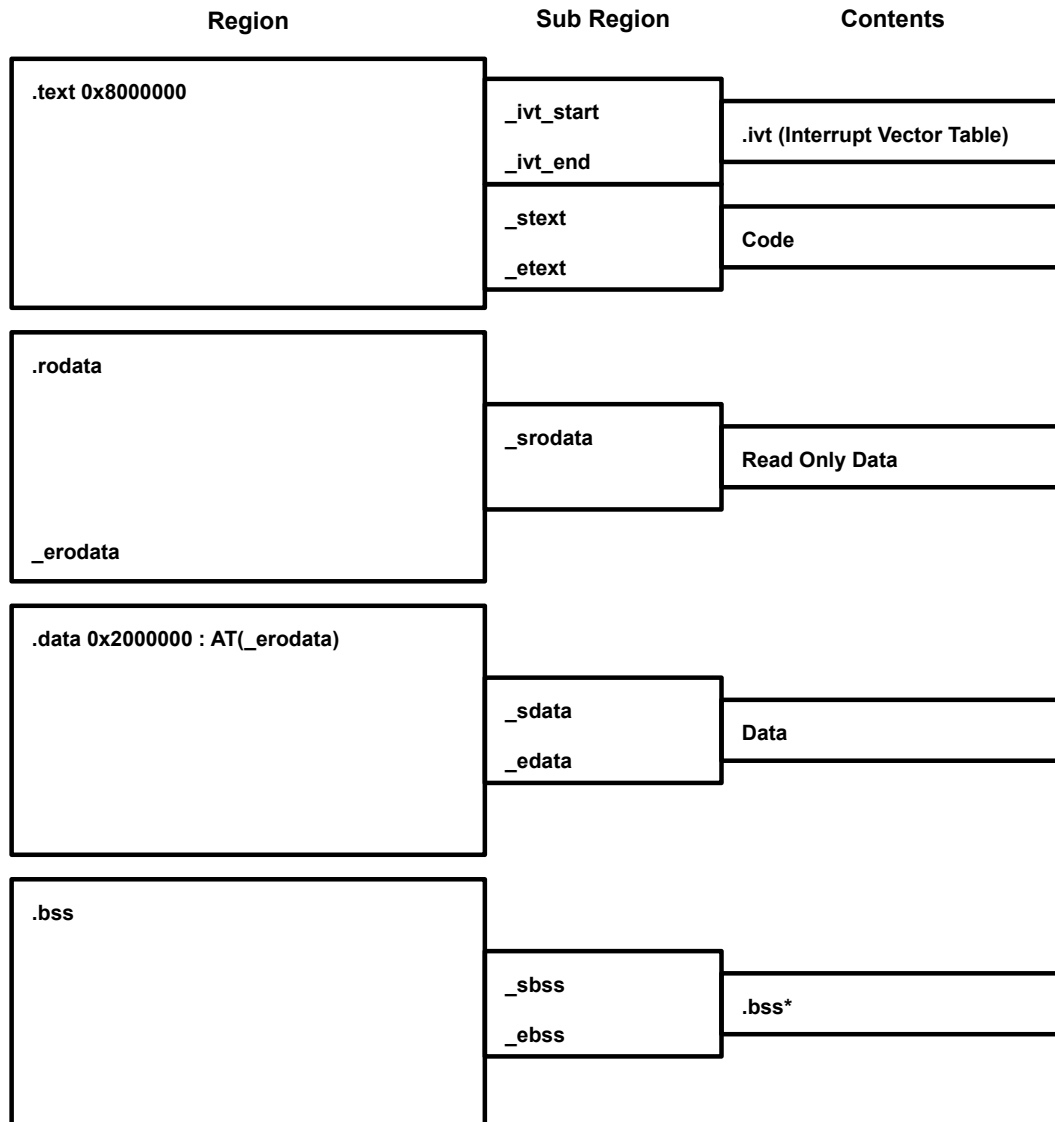
The way around this is to reserve space in SRAM for these variables, but place their values in flash. However, for this to work, the initialized values must be copied from flash to their respective locations in SRAM. Additionally, the uninitialized global variables must be set to 0.

## 6.1 Linker Layout

You will need to have basic understanding of linker scripts to do this lab. Be sure to read this short example before proceeding if you are not familiar with them. <http://www.bravegnu.org/gnu-eprog/lds.html>.

For this class, we will be using the linkerscript in `util/linker_template.lds`. The STM32 board has 96KB of SRAM and 512KB of flash and the start addresses for each of these are `0x2000000` and `0x8000000` respectively. The text and rodata sections are placed in flash. The data and bss sections are placed in SRAM.

The naming convention for variables are `_s(start)[section name]` and `_e(end)[section name]`. For instance, the bss section starts at `_sbss` and ends at `_ebss`.



The read-only global and static variables are placed in the `rodata` section in flash. These variables are not written to and thus can be kept in flash. The initialized global and static variables are placed after `erodata` and must be copied to the `data` section located in SRAM. The `bss` section is where the uninitialized variables are placed and must be set to 0.

Therefore you must do the following:

1. Familiarize yourself with the linker layout (You may be asked questions about it during checkoff).
2. Fill in the first two entries in the vector table (see the first `@ TODO`.)
3. Copy the initialized global variables from flash to SRAM. (The initialized values in flash are placed from `_erodata` onwards, as specified by the AT directive, and the SRAM begins at `.data`)
4. Set the entire `bss` section to 0.
5. Branch to `kernel_main`.

Note: You *must* make sure that you do not skip over any bytes when copying the `data` section or zeroing out the `bss` section. You *also* need to handle the cases when these sections are of size 0.

If you successfully performed the above tasks you should see the following output on the minicom terminal:

```
Entered kernel_main, starting boot loader test
Boot Loader Successful !
Baseline Time = 47375
Optimized Time = 47366
```

Note: Your optimized time should be significantly less than the baseline time.

Please refer to section 10 in this handout to configure your minicom settings. You might have to open a new terminal window for this purpose in addition to those two windows you already opened as per section 5.3

Note that the bootloader test only tests for the most basic functionality, and there are many things that are may not be caught (for example, when `bss` or `data` size is 0.)

**Hint:** For this lab and future labs you may need to load symbols declared in the linker script. In C, you can do this by declaring them as `extern`.

In assembly, ARM provides a pseudo-operation.

LDR Rd, =expr

## 7 ARM Optimization

In the next part of the lab, you will apply your knowledge of assembly programming towards optimizing a simple assembly program, found in `kernel/asm/optimize_me.S`.

Your grade will be determined by your speedup over the unoptimized code. The speedup will be calculated by the following formula:

$$\lfloor \frac{BaselineTime}{OptimizedTime} \rfloor \quad (1)$$

The point breakdowns are listed below. Note that we will NOT be rounding up.

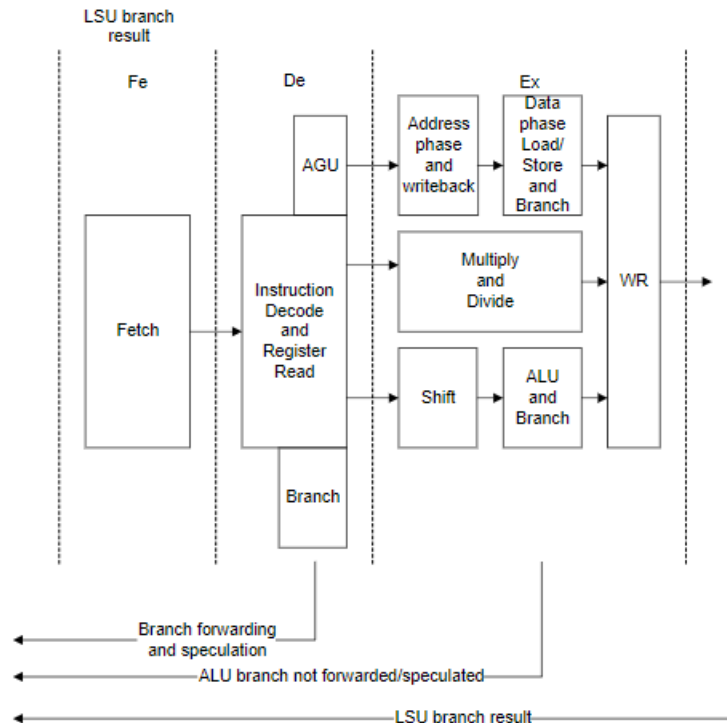
Fails correctness check	0
2x faster	5
5x faster	15
10x faster	30
15x faster	35
30x faster(bonus)	40

We measure actual wall clock time, so its not enough to minimize code size, you must also minimize time delays from branch and load stalls. For those of you that understand pipelines, below is the pipeline for your processor.

For this task, you may only use the DSP extensions once you have reached a 15x speedup to get the bonus points. **Using the DSP extensions under a 15x speedup will result in point deductions.** If you are worried if a certain instruction is a part of the DSP extension, you can consult [https://www.keil.com/pack/doc/cmsis/Core/html/group\\_\\_intrinsic\\_\\_SIMD\\_\\_gr.html#details](https://www.keil.com/pack/doc/cmsis/Core/html/group__intrinsic__SIMD__gr.html#details). However, a good rule of thumb is if the instruction acts on more than once piece of data at a time, it is probably part of the extension.

**There is one exception to this rule, the `smulbb` instruction. Its in the starter code. Feel free to use it.**

We have also included a copy of the instruction quick reference with the name "`m4_programming_manual.pdf`" in the `docs/` directory. If you have a question like "is there an instruction for x" be sure to consult the quickref and the slides before posting on Piazza.



## 7.1 Tips For Optimization

The STM32 processor fetches and executes code from flash. Flash is far slower than SRAM, thus under normal circumstances; execution would be extremely slow. Each Flash memory read operation provides 128 bits for either four instructions of 32 bits or 8 instructions of 16 bits according to the program launched. So, in case of sequential code, at least four CPU cycles are needed to execute the previous read instruction line. Thus the processor continually fetches ahead so that it can execute with 0 wait states. However, if there is a branch to a target outside this prefetched buffer, it must stall until the flash read is done. Thus to get good performance, you should try to minimize the number of branches in your code.

Review your lecture notes for more optimization techniques.

Start early and make small incremental changes, make sure to run the program periodically to make sure you pass correctness. For reference, the TA solution takes only 16 and 14 instructions for the baseline and bonus respectively. These values are before we performed loop unrolling.

## 8 In Lab Demo

During office hours, bring your board and demo your Boot Loader and your ARM optimization. Your output should look something like this.

```
Entered kernel_main, starting boot loader test
Boot Loader Successful !
Unoptimized Time = 47375
Optimized Time = 47366
```

## 9 Submit to GitHub and Canvas


Fill in any comments or suggestions you have about the lab in README.txt. For instance: was it too easy, too difficult, too short, too long, you get the idea.

To submit, create an issue on GitHub titled Lab1-Submission. In the comments section include your name, andrewid and the commit-hash you want to submit. Also be sure to submit the link to your repository to Canvas.

When you are done, it should look like below. Pictures of cute animals are not needed but are recommended.

**NOTE:** Please do not forget to make a git commit prior attempting bonus. For this lab, you must raise one git issue with a hash id for the usual submission with the title "Lab1-Submission" and another one for the bonus (if you have attempted) with a different hash with a title "Lab1-Submission-Bonus".

### Lab1-Submission #1

 Open ronit10z opened this issue 5 minutes ago · 0 comments



ronit10z commented 5 minutes ago • edited ▼

+ 🗨️ ...

Ronit Banerjee (ronitb), James Zhang(jameszha)

[c31d0a6](#)



## 10 Minicom Setup

1. To setup minicom, you must first find out the device id assigned to your STM32. This is usually `/dev/ttyACM0`. To find it unplug your STM32 and `cat /dev/`, then replug it and `cat /dev/` again. Find the device that shows up after you plug your STM32 in.
2. **For Linux:** run `sudo minicom -b 9600 -8 -D /dev/ttyACM0`  
**For Mac:** Run `minicom -b 9600 -8 -D /dev/cu.usbmodem103` (you may have a different number than 103)

- By default, minicom expects carriage returns with every newline char, to disable this press ctrl+A Z and then U. For Mac it is ctrl+Z U.

## 11 GDB Tips & Tricks

GDB is a very powerful program debugger. For those who have never used it or are just getting reacquainted with it, you may not know some of the more useful features of GDB that will make the debugger work for you, instead of the other way around. Listed below are a bunch of helpful shortcuts that can make your life easier.

Command	Shortcut	Description
run	r	runs the program until completion, fault, or breakpoint
quit	q	exits debugger
break <location>	b	sets a breakpoint at <location>
break <location> if <condition>	b <loc> if <cond>	stops program execution at <location> only if <condition> is met (can be very slow).
delete <bkpt/whpt num>	del	deletes a breakpoint/watchpoint
watch <var>	wa	sets a watchpoint that displays when <var> changes value
backtrace	bt	prints out the chain of function calls until currently running function
step	s	steps by one line of C, entering function calls
stepi	si	steps by one assembly instruction
next	n	steps by one line of C, ignoring function calls
nexti	ni	steps by one assembly instruction, ignoring function calls
continue	c	continue execution until completion, fault, or breakpoint
finish	fin	execute until return from currently running function
print <arg>	p	interpret and print the argument (analogous to printf)
print/<format> <arg>	p/<fmt>	interpret as <format> and print the argument (analogous to printf)
examine <arg>	x	examine memory at address <arg> (analogous to *(arg))

For details on what various arguments these commands can take, checkout the various cheatsheets and reference cards on Canvas. NOTE: some of the commands in the cheatsheet may have varying functionality.