Lab Environment Setup

04-633 Students.......... Assemble!

04-633 Embedded Systems Development

Contents

Contents

1 Introduction and Overview

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!

2 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.

2.1 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/10hs7xkNUtgknOAvOlL5ZwoQfy2_xsQQR/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.

2.2 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 the 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_ocd or ./linux_ocd throws no errors.

3 STM32

In your VM, you will git clone your repo.

3.1 STM32 Boot Process

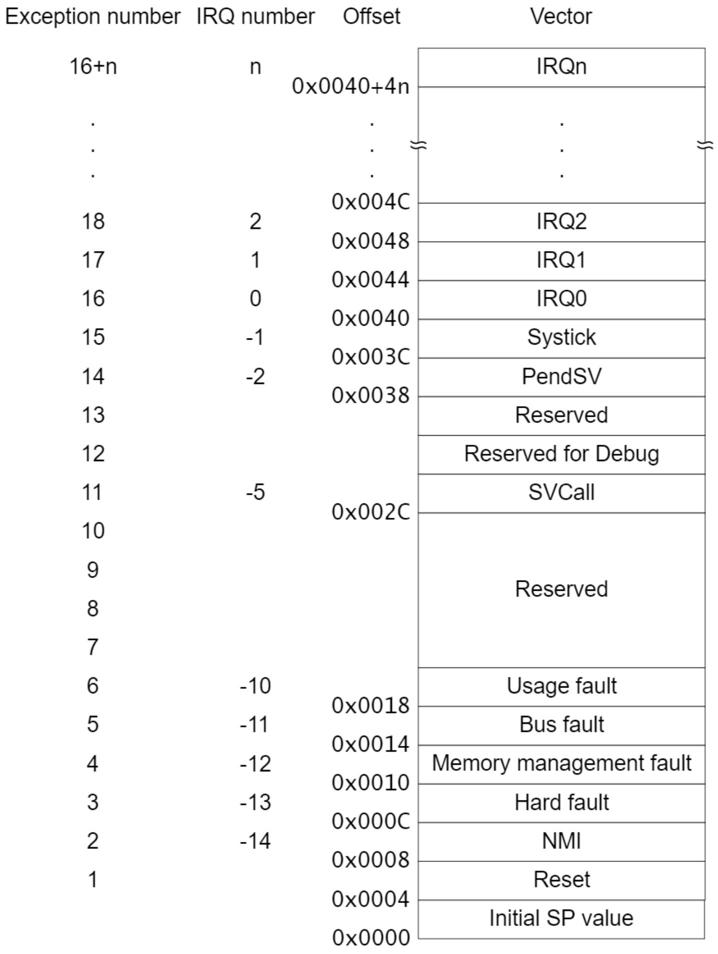
To debug on the STM32, we will use JTAG (a testing interface defined by the Joint Test Action Group). This is a debugging method developed in 1985 to test PCBs (Printed Circuit Boards) after they are manufactured. This is the most common interface for debugging embedded processors and is supported by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



4 Testing your board

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

4.1 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

4.2 How to Start GDB Debugging

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

1. Clone your repo.
2. 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).

3. 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 ommited
(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.

4.3 Checking Your Board

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

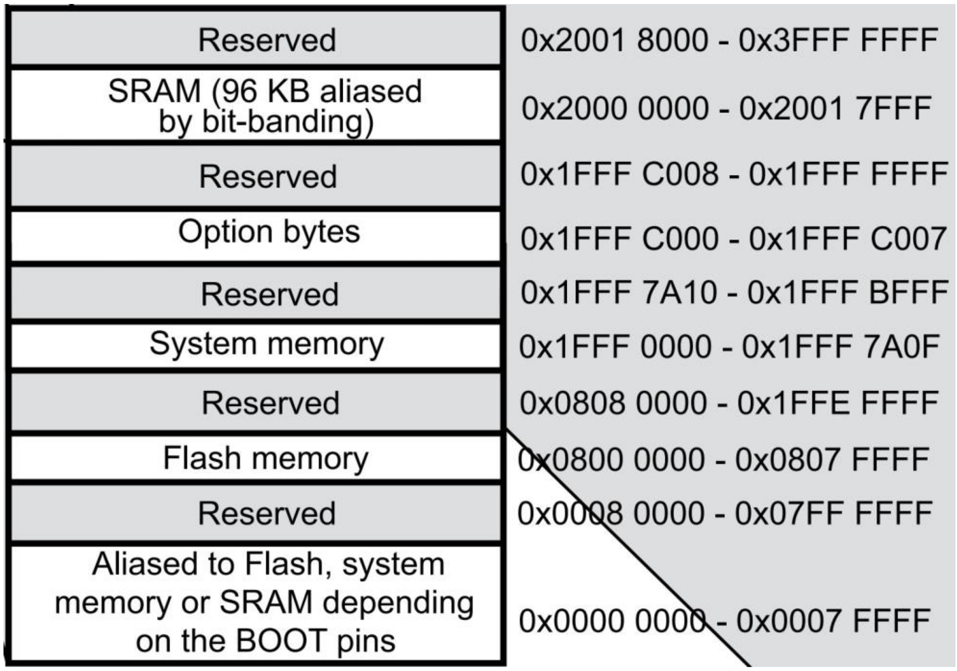
1. Run $ sudo ./linux_ocd or windows_ocd.bat or ./osx_arm_ocd
2. In a new terminal window, run $ make flash
3. In an another new terminal window, start mincom (see section 6). Or you can run $ cat /dev/serial/by-id/[really long name] (you don't have to type the whole thing, just get to by-id and press tab). In Windows, you can double click windows_serial.
4. In the GDB window, run (gdb) continue
5. 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.
6. 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
```

5 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 to 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).

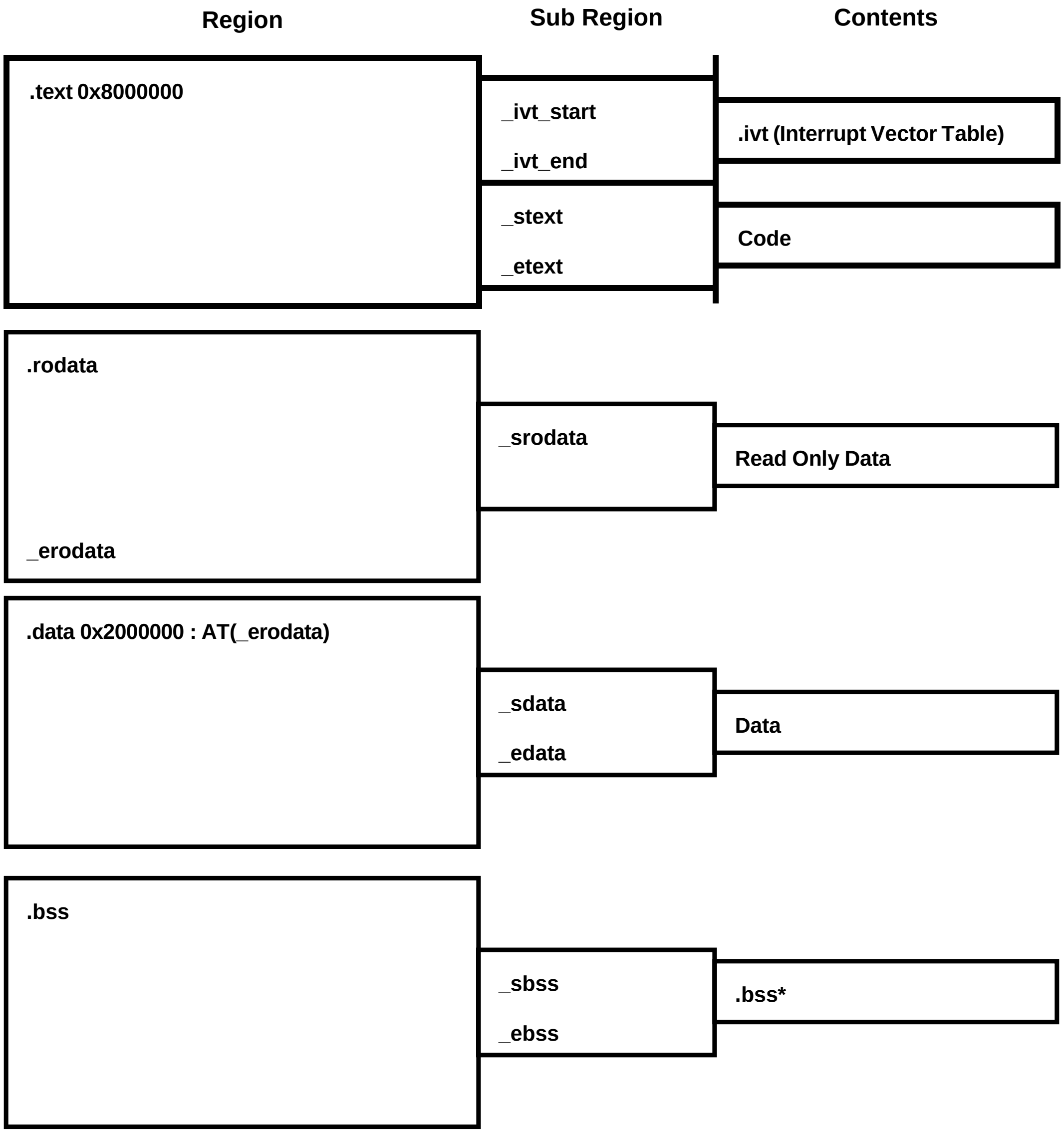
The way around this is to reserve space in SRAM for these variables, but place their values in flash. However, for this to work, the initialized values must be copied from flash to their respective locations in SRAM. Additionally, the uninitialized global variables must be set to 0.

5.1 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.

All of this has been done for you, so no need to stress. Cool, right? Then, 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
```

Please refer to section 6 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 4.3

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
```

6 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)
3. 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.

7 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>
break <location> if <condition>
	b
	b <loc> if <cond>
		sets a breakpoint at <location>
		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 stepi	s si	steps by one line of C, entering function calls steps by one assembly instruction
next nexti	n ni	steps by one line of C, ignoring function calls 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> print/<format> <arg>	p p/<fmt>	interpret and print the argument (analogous to printf) 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.