Learning Objectives

1. Leaning bare metal development
2. Understand memory mapped IO in a real system
3. Interact with external hardware (GPIO) to perform a useful task
4. Learn to use interrupts

1 Introduction

So far you have written, compiled and executed your programs using an interface provided by the operating system. You were able to use a keyboard and display to give inputs to the programs and receive outputs. You have used printf and scanf functions for this purpose. These functions rely on the operating system’s management of the hardware.

In this lab, you will learn how you can interact with the hardware at the lowest level. This kind of programming is important for writing drivers for new hardware or porting an operating system to a new processor. Working directly on the hardware also allows you to create extremely energy efficient systems. This will also expose you to the details of the processor and peripherals that are usually hidden from you when you are using an operating system (our main objective for making you program at this level). You will be using bare metal assembly (with some C libraries for complex parts) for your projects as well.

2 Lab Setup

1. Connect your memory card into the computer and delete everything from the boot partition. Now you do not have an operating system.

2. You need an environment where you can now compile programs for Raspberry Pi. Since Raspberry Pi does not have anything right now so it can not be used for compiling the programs. Download the AArch64 ELF bare-metal target (aarch64-none-elf) for your operating system (Windows, Linux on x86 or Linux on ARM) from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads. This will allow you to compile your programs on your computer but produce a binary that is compatible with Raspberry Pi. This is called cross compiling.

3. Linux users: Unzip and add the path to the bin folder to your PATH variable. Windows users: (We will figure it out if you need it)

4. Run (all in one line) to add our repository as a remote on your computer.

```bash
$ git remote add honors_release \\
https://github-dev.cs.illinois.edu/cs233-fa20/_honors_release.git
```
Run the following (separate commands) to get the files for this lab

```bash
git fetch honors_release
git merge --allow-unrelated-histories honors_release/RaspberryPiLab3
```

You only need to do this step once. Once you make a commit with merged lab, it will be available in your repository and simply cloning it would be enough.

5. Run `make all`

6. You will be adding your code to *.s files for this lab. `make all` will produce a new `kernel18.img`

7. Copy the files `bootcode.bin`, `config.txt`, `fixup.dat`, `start.elf` and `kernel18.img` to your memory card. The basic lab file flashes an led connected to GPIO (More on how that works later)

8. Make sure you attend the tutorial session or watch the video. Contact the TA’s if you have any questions.

9. Commit and push your work frequently.

10. Once you complete your lab, commit and push your final version.

11. You have the option of either working in pairs or individually but we strongly encourage pairs. If you work in pairs, please make sure to fill the partners.txt file.

12. You will need a few wires and an LED for this lab.
3 Memory Mapped IO

Any useful computer system consists of more than just a CPU. Generally, you might find SRAM, DRAM, GPU, Peripherals, and busses like PCIe, SPI, I2C, UART, RS485, etc... When running typical userspace programs, the Linux kernel will provide abstractions to all of this hardware for you. But sometimes it can be helpful or more performant to bypass these abstractions and utilize the hardware yourself.

The Raspberry Pi model 3 that we use in this course, like many similar boards, comes with a datasheet, a special manual detailing everything important to a programmer about the hardware specifications and configuration. A link to this document for Raspberry Pi 3 documentation is http://classweb.ece.umd.edu/enee447.S2019/ARMv8-Documentation/BCM2837-ARM-Peripherals.pdf. The portion of the datasheet that we are concerned about are related to the Memory Management Units (MMUs) and the Memory Mapped Input/Output addresses (MMIO addresses). Page 5 gives a high-level schematic of the addresses, where what we are concerned with are the “I/O Peripherals”. On Page 6, we see:

Peripherals (at physical address 0x3F000000 on) are mapped into the kernel virtual address space starting at address 0xF2000000. Thus a peripheral advertised here at bus address 0x7Ennnnnn is available in the ARM kernel at virtual address 0xF2nnnnnn.

Physical addresses range from 0x3F000000 to 0x3FFFFFFF for peripherals. The bus addresses for peripherals are set up to map on to the peripheral bus address range starting at 0x7E000000. Thus a peripheral advertised here at bus address 0x7Ennnnnn is available at physical address 0x3Fnnnnnn.

If you are not familiar with these terms (particularly "virtual memory"), don't worry! You will encounter this concept in CS 241 and again in other optional hardware/operating systems classes (CS 433 and CS 423 are ones we highly recommend!). For now, just understand that in your code, writing/reading to address starting from 0x3F000000 will interact with the I/O peripherals of the board. The document specifies addresses as bus addresses throughout the document but you should convert them to physical addresses by replacing 7E with 3F at the star of the addresses.

Skipping ahead to page 90, we find information on the “GPIO Register View”. GPIO stands for General Purpose Input/Output. These are pins on the Raspberry Pi that can be used for general purposes. They can be configured to either receive input, send output, or do nothing. On the next page is the pinout (labels) of the GPIO pins on the Pi. Note that in this lab we will use the pin numbers (in the circles) as our numbering system.

Please note that you should always double check your work when connecting any pins on any hardware. In particular, don’t connect a GPIO pin to any power source unless you know what you are doing.

Back to page 90: here is the list of registers available that allow you to configure the behavior of the GPIO pins. On the left column are the physical addresses of the registers, but remembering page 6, we know that in the kernel, these addresses are shifted up by 0x3F000000.

The following pages of the datasheet go into more detail about the various registers available, which we recommend you read to have a better understanding of what the code we provide is doing for you (as well as a better grasp of capabilities for a final project).
4 Interrupts

See the Lab 3 tutorial session and contact the TA’s
5 Assignment

Remember if at any point you feel uncomfortable or unsure about connecting hardware, please ask us!

To run your code, run `make all` and then copy `kernel8.img` to memory card.

5.1 Part 1: Output

The first part will have you configure a pin as an output and flash an led to show output.

1. Configure a pin of your own choice as an OUTPUT. We recommend pin 32 (GPIO 12).
2. Set initial value on the pin to be 0.
3. Use the system timer to flip the output every 0.5s.
4. Connect an LED (with a resister in series) between pin 32 and ground (pin 30, 34 or 39).
5. This should flash the LED every 0.5s.

5.2 Part 2: Input

This part will have you learn how to set up a GPIO pin as INPUT and how to configure GPIO interrupts.

1. Start with your code from part 1.
2. Configure a pin exposed on Raspberry Pi pinout shown earlier as an INPUT. We suggest pin 40 (GPIO 21) In the absence of external pull ups you will have to configure pull ups as well. The procedure is documented towards the end of GPIO documentation. Configuring pull up resistor will causes the default value to be 1
3. Configure a new pin (not the one used in part 1) as output.
4. Configure a falling edge interrupt on the selected pin.
5. The interrupt handler should flip the new output pin.
6. Connect a wire between the part 1 output and the input pin. Connect the LED to the new output pin.
7. The LED should now flash once every second.

If you do not have wires then use the following pins: GPIO 06 for part 1 output, GPIO 5 for part 2 input and GPIO 19 for part 2 output. Commit your code and come to office hours we will check your code on our hardware.

GL;HF
6 Additional Resources

1. GNU ARM Assembler Quick Reference

2. The gnu Assembler
   (https://web.eecs.umich.edu/ prabal/teaching/resources/eecs373/Assembler.pdf)

3. Debugging Assembly Code with gdb
   (http://web.cecs.pdx.edu/ apt/cs491/gdb.pdf)

4. Introduction to ARM Assembly Basics
   (https://azeria-labs.com/writing-arm-assembly-part-1/)

5. ARM Quick Reference Sheet
   (http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf)

6. Procedure Call Standard for the ARM® Architecture

   (https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf).