Project Zero

ENEE 447: Operating Systems – Spring 2020
Assigned: Week of Feb 3; Due: Week of Feb 10

Purpose

This project simply gets you up to speed, but it will nonetheless take quite a bit of time. There are several fundamentals you need to have under your belt before you can even begin any of the projects, and this project gets you to that point. Fundamentals include:

- Ownership of one (1) Raspberry Pi 3 B+ with multicore processor [Broadcom BCM2837, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz]
- Ownership of a Class-10, formatted micro-SD card, SD adapter, and SD card reader
- Ability to cross-compile from whatever operating system you run on your laptop to the ARM platform (there is an extremely small likelihood that your laptop runs ARM code natively)\
- [Not needed until Project 1] a "console cable" to interact with the RPi3 board over a serial port (yay character-based I/O), see https://www.adafruit.com/product/954

Those are the fundamentals. In this project you will start with them and then put them together to build a kind of "hello, world!" example for the RPi that blinks the activity LED. Once you have gotten that far, you will be ready to start building an operating system.

Get a Raspberry Pi

Go to https://www.raspberrypi.org for information on vendors. The board should cost $35, though I have seen it for sale at some vendors for $39 and up. Here is where I found links to purchase mine:

https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus

Very Important Note: Get the version 3B+, which has the 64-bit quad-core Broadcom BCM2837 (ARM Cortex-A53) chip in it. Don't get a version 1 or a version 0. Those boards are for wimps and sissies, and you are definitely not wimps and sissies, are you? No, I thought not. Don't get a version 2 because it uses a 32-bit chip, and its peripherals behave differently than the BCM2837/Cortex-A53. Using non-standard hardware or toolchain will cause you nothing but pain, as nobody in the class will be able to help you if you have any problems or questions. Don't get earlier versions of the 3, because of LED issues (see later). You might also want to purchase a full kit with a power supply, or purchase the correct type of micro-USB cable (note there are like 37,000 different versions of micro-USB out there, so make sure you get the right kind) if you want to power the Raspberry Pi directly off your laptop. This last part is a consideration because it would allow you to develop & debug your code anywhere, even if no wall socket is available. You will also want a NOOBS micro-SD card with adapter (comes pre-formatted, so it will save you time and hassle), and if your laptop doesn't have an SD-card slot, you will want a USB SD-card reader. Lastly, because you don't want to have to go out and buy a *second* Raspberry Pi board because you blew out one of the chips on your *first* Raspberry Pi board (the way I did), you might want to consider purchasing one of the many, attractive cases they have for them.

Read the Baremetal Introduction

At this link (https://github.com/dwelch67/raspberrypi/tree/master/baremetal) is an introduction to "bare metal" programming, written by David Welch, a professional software developer who happens to do a lot of enthusiast programming on the Raspberry Pi and then shares his code and his thoughts with the world. He has written a ton of code for the RPi1, RPi2, an RPi3, and a number of his examples are multicore-specific.

His introduction is eye-opening in that it shows you how different life is when you decide to write code that runs directly on the CPU, as opposed to running within the context of a particular operating system. For starters, none of your library routines work. That means *read, write, open, close* no longer work, but perhaps more importantly it means that you no longer have *printf* — or any other I/O, for that matter. You also have no *main()* function, because *main()* expects a whole lot of stuff to have happened before it ever gets to square one. How do you think the various "hello, world!" examples can do so much with so little code? The answer: there is tons of work going on behind the scenes, and it all runs before the first line of *your* code ever comes to life.

Because Welch uses the RPi and ARM as his example, one of the things that his introduction will show you is how to understand what is inside of the binary you create. He will show you how to disassemble files to make sure that everything is where it should be, and he will make you realize how easy it is to screw little things up. He will show you how to get around the lack of *main()* and libraries, and he will show you how little assembly code you really need to get the job done. The good news: very, very little assembly code is needed. You should all be cheering at this point.

This will also help you understand why it is important that you use the same toolchain as me, the TA, and all other students. Given the same C and assembly files as input, two different toolchains can and will generate radically different binary files … thus, if you are running a non-standard toolchain and run into problems, nobody will be able to assist you. You will get weird errors and behaviors that nobody else experiences. Your life will be sad. Don't choose a sad life.

Install the GNU Cross-Compilation Development Environment

The reason that the Raspberry Pi boards have captured the world's attention and imagination is because they started out at $35. That is a fully functional computer — minus only a screen and some form of data entry (keyboard, touchscreen, whatever). This means that you could build a computer in which the computer portion is actually the cheapest component (it's hard to buy a monitor or keyboard or mouse for less than $35). How could they possibly be that cheap? The answer: they are using cellphone chips as the main processor, and whenever a chip sells millions and millions of devices, the cost is low. Cellphones sell in the gazillions, so the components are cheap.

So the Raspberry Pi guys started out with a single-core Broadcom chip, the BCM2835. The first versions of the Raspberry Pi, based on that chip, took the world by storm. But, of course, that was several years ago, and Broadcom has not stood still … they now have a quad-core chip that they made especially for the Raspberry Pi 2, the BCM2836 chip that has four (4) ARM cores in it, not just one (1). And now they have a quad-core 64-bit chip, the BCM2837. Egads.

What is the new price? The same as before: $35. So what, you might ask, astutely, are they doing with the old single-core BCM2835 chip, now that nobody wants to buy the old RPi board design based on it? Well, that chip is now cheap enough that they can build a board based on it, and sell it for $5. Yes, that is a single-digit dollar figure. You can buy a completely functional computer for less than the cost of the SD card that holds its operating system. It is called the Raspberry Pi 0.

But I digress. The RPi is cheap because it is based off cellphone chips, and cellphone chips have to be cheap enough to put into, you guessed it, cellphones. Therefore they cannot use Intel chips because Intel chips aren't cheap. So the cellphone manufacturers use ARM chips instead.

The problem is that your laptop uses an Intel chip. Intel chips and ARM chips are not compatible. "Not compatible" means that the code you write and compile on your laptop will run fine on your laptop but will be garbage on any other platform, including an ARM chip.

So what do you do? Option 1: you buy a second Raspberry Pi, run Linux on it, using a mouse, keyboard, and HDMI monitor, and you develop all your code there, running the developed code on the first board. Option 2: you use your laptop and a cross-compiler. If you go with option #2, get ARM's development kit. Look it up, get it, download it, and install it. You will need the kit to build code for the "A" series of cores (we are running the Cortex-A53). The ARM GNU-A tool kit can be downloaded from here:

https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads

That site lists several cross-compiler downloads. For these projects, we will be using the RPi in 32-bit mode, so look for "**AArch32 bare-metal target (arm-none-eabi)**". Once it's downloaded, unzip the file, and move the folder somewhere safe. Then add the "bin" folder within to your system PATH. You can test whether or not the cross-compiler is installed successfully by typing "*arm-none-eabi-gcc*" on the command line.

General details on how to set up a GCC cross-compiler for those who are interested:
https://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/

macOS

Try this first: Get the MacOS tarball from this site, https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads, move it to root opt, and add it to your path. If that doesn't work for some reason (it should), try the following.

If you are running a Mac laptop, you will need to get Xcode installed. If you are running Mojave, you will also need to do the following at the command-line:

sudo installer -pkg /Library/Developer/CommandLineTools/Packages/ macOS_SDK_headers_for_macOS_10.14.pkg -target / (one line)

[see https://apple.stackexchange.com/questions/337940/why-is-usr-include-missing-i-have-xcode-and-command-line-tools-installed-moja]

This command creates and populates the /usr/include directory, which is the standard C-language directory that contains all system header files required for building anything. And, should you fail to run that rather obscure command on the command line, that directory will otherwise not exist. Yes, I know exactly how stupid that sounds. Apple has gone WAYYYYYYY downhill since Steve Jobs died.

Get the source code at https://developer.arm.com/open-source/gnu-toolchain/gnu-a/downloads. Currently this is the file "gcc-arm-src-snapshot-9.2-2019.12.tar.xz" at the bottom of the page, but as they create new versions they update the page, so just scroll to the bottom and get what's there.

Run the following commands:

brew install gmp
brew install mpfr
brew install libmpc

Inside the main directory of the source code and create a directory called "build" or whatever you want. Go to that directory and type the following:

../gcc-arm-src-snapshot-8.2-2019.01/configure --target=aarch64-elf --disable-nls --enable-languages=c --without-headers
make -j4 all-gcc
make install-gcc

Download binutils from here: http://ftp.gnu.org/gnu/binutils/binutils-2.32.tar.xz

Treat this just like GCC. Outside of the binutils directory, create a directory called "build". cd into it, then type the following:

../binutils-2.32/configure --target=aarch64-elf --disable-nls --enable-languages=c --without-headers
make -j4 all
make install

You will probably have to run the installs as root (sudo make install-gcc, sudo make install) so that the make processes can put the files into the /usr/local/bin space.

Windows

The arm-none-eabi cross-compiler is *supposed* to work on Windows. I'm sure that it does if you configure everything correctly, but after significant effort I was unable to get it right. Therefore, if you are on a Windows machine, I recommend you use an Ubuntu virtual machine. You can install VMware from TERPware or get VirtualBox online for free. Ubuntu ISOs can be found here: https://ubuntu.com/download/desktop.

Test It

The first thing that you should do once you have your cross-compiler set up is to verify that it outputs the same thing that ours does. On the course website is the source code and output of the compiler, assembler, linker, etc. Download all the files and ensure that when you type "make" you get the exact same output that we do. If not, if any of the files have different content, then there is something amiss, and you might have significant difficulties getting anything to work. If this is the case, debug what you have, and/or ask questions of the TA and professor.

General Notes

What is the difference between "arm-none-eabi" and "arm-linux-gnueabihf "? Can I use the "arm-linux-gnueabihf " tool chain in a bare-metal environment? How do you know which toolchain binary to use where?

The general form of compiler/linker prefix is as follows: A-B-C

Where the strings between the dashes mean the following:

- A indicates the target (*arm* for AArch32 [32-bit] little-endian, and *aarch64* for AArch64 [64-bit] little-endian).
- B indicates the vendor (*none* or *unknown* for generic). Note that this is optional (Eg: it is not present in aarch64-elf).
- C indicates the ABI in use (linux-gnu* for Linux, linux-android* for Android, *elf* or *eabi* for ELF based bare-metal).

We are running bare metal on the ARM Cortex-A53, which is of type AArch32, and so we will want to use the *arm* target, and the *eabi* ABI.

How the RPi boards boot:

- The GPU starts first @0x00000000
- The GPU runs *bootcode.bin*, configures the system
- Then the GPU runs *start.elf* to load the *kernel7.img* image at location 0x00008000
- Once the kernel is loaded into memory, the GPU hands off control by jumping to 0x00008000
- 0x00008000 is the location for the first ARM instruction to be executed
- 0x00008000 is where *kernel7.img* will be located (your code)

- config.txt and its contents can change the way the GPU boots the ARM, but we will try to rely upon that as little as possible

What a compiler does: source code -> compiler -> object code
What an assembler does: assembly code -> assembler -> object code (e.g., in ELF format)
Object code is a binary file, but it is not runnable yet; we need to *link* it with libraries/functions/variables that are found in other objects.
What a linker does: resolve and link all necessary object files to generate the final binary to run. The linker connects the object code of the program with the object code from libraries, etc. In the GNU toolchain, the linker is the "ld" program.

Additional tools in the GNU toolchain:

- objdump - generate human readable output from a.o binary (ELF, etc)
- objcopy - copy.o file from one format (e.g., ELF) to another format (e.g., hex, bin)

The kernel7.img file needs to be of the "bin" type, which is why the Makefile we give you uses *objcopy* to create the final executable kernel image to be put on the SD card.

Blink the Activity LED via General-Purpose I/O (GPIO)

An LED is *extremely* useful as a rudimentary debugging tool and a simple indicator of progress/activity. Because they are also so simple to use, they are typically the first thing you will want to get working whenever dealing with bare metal programming on a hardware device. We have given you code that interacts with the RPi3's activity LED. This will let you know whether your code boots, and it will serve as a useful sanity checker in the future. It is always a good idea to have a failsafe backup just in case other, more sophisticated forms of I/O are not working, for whatever reason.

In the Raspberry Pi 3B+, the activity LED is at GPIO 29 … I believe that in earlier versions of the RPi3, the LEDs are unavailable, or are only available through mailbox-based IPC calls to the graphics processing unit, so this is yet another reason to get version 3B+ of the board. The following is a quick overview of how GPIO works, in case you want to mess with it yourself. For instance, it is how we will get a serial port up and running in Project 1.

GPIO Device Programming

The GPIO base address is at `0x3f200000`.

Addresses `0x3f200000,0x3f200004,0x3f200008,0x3f20000c,0x3f200010` are called "GPFSEL" registers (GPFSEL0, GPFSEL1, etc.). These are the control registers that determine whether or not a particular GPIO pin is enabled, and, if so, whether it is configured to be input, output, both, etc. Each GPIO pin uses 3 bits for its control; a value of '0' enables the GPIO for input; a value of '1' enables it for output, and so forth. Moreover, each register address listed above is 32 bits, so we have the following:

```
GPIO0 is controlled by GPFSEL0, bits 0-2
GPIO6 is controlled by GPFSEL0, bits 18-20
GPIO16 is controlled by GPFSEL1, bits 18-20
GPIO18 is controlled by GPFSEL1, bits 24-27
GPIO47 is controlled by GPFSEL4, bits 21-23
etc.
```

To set a value, clear the 3-bit range to zero, write the bit/s you want, and then write the 32-bit value to the register, retaining previous settings. For instance, to initialize GPIO 29 for output, do the following:

```
unsigned int ra;

ra=GET32(GPFSEL2);
ra &= ~(7<<27);
ra |= 1<<27;
PUT32(GPFSEL2,ra);
```

If you want to output to a GPIO, you use the GPSET and GPCLR registers. Addresses `0x3F20001c,20` are the GPSET registers (GPSET0 and GPSET1); addresses `0x3F200028,2c` are the GPCLR registers (GPCLR0 and GPCLR1). To set a GPIO, write a '1' to its corresponding bit position in a GPSET register; to clear a GPIO, write a '1' to its corresponding bit position in a GPCLR register. GPIOs 0–31 correspond to GPSET/CLR0, and GPIOs 32–63 correspond to GPSET/CLR1. For instance, to turn off the LED at GPIO 29, do the following:

```
PUT32(GPCLR0,1<<29);
```

And to turn it on, do the following:

```
PUT32(GPCLR0,1<<29);
PUT32(GPSET0,1<<29);
```

I have had good luck initializing the LEDs on every access and clearing before setting. You will see this in the *led.c* code provided to you. I do not know whether doing these extra steps is necessary or not, but it works for me.

General Memory-Mapped I/O Information

The following gives the memory-mapped addresses at which you can find the control registers for a number of the devices in the Broadcom 2837 chip:

```
0x3F003000 - System Timer
0x3F00B000 - Interrupt controller
0x3F00B880 - VideoCore mailbox
0x3F100000 - Power management
0x3F104000 - Random Number Generator
0x3F200000 - General Purpose IO controller
0x3F201000 - UART0 (serial port, PL011)
0x3F215000 - UART1 (serial port, AUX mini UART)
0x3F300000 - External Mass Media Controller (SD card reader)
0x3F980000 - Universal Serial Bus controller
```

The term "memory-mapped I/O" means that the registers belonging to the various hardware devices that perform I/O functions on behalf of software are *mapped into* the address space, which means that, instead of having to use special I/O instructions to address them, you can use simple load/store instructions as you learned about in 350 and/or 446. This tends to simplify the development of driver code tremendously.

Write a Simple Timer Library

Once you have read about how to write code for the ARM architecture, and once you have an *arm-none-eabi* cross-compiler development environment up and running, you are ready to build something. On the course website are some documents on how the ARM's peripherals are set up, including timers. Timers are extremely useful, and you cannot really do much in the way of system-level development without

them. Your job is to create a function called *wait()* so that the following code example will work as expected:

```
while (1) {
      wait( ONE_SECOND );
      led_on();
      wait( ONE_SECOND );
      led_off();
}
```

Additional functions you need to create for this time-keeping facility are the *gettime()* and *timediff()* functions. These should work as one might expect from their names, for instance in the following example, where you would use them to measure the runtime of a particular function or operation:

```
unsigned int now, then, delta;

then = gettime();
some_long_function_or_operation_that_we_want_to_time();
now = gettime();
delta = timediff(now, then);
```

We will provide you with a Makefile, skeleton code, and the code for the *led_on()/off()* functions. The code should boot as-is, but without correct timing of the LED blinking. Your job is to write the *wait()*, *gettime()*, and *timediff()* functions such that *wait(ONE_SECOND)* exits approximately one second after entering. By "approximately" I mean that any time duration in the immediate vicinity of one second (as measured by a stopwatch) is fine. A quarter second, ten seconds, one millionth of a second, an infinite stall, etc. — those examples count as "non-functional." The functions *gettime()* and *timediff()* can be at any time granularity you want, millisecond or finer. In my own code, I call them *now_usec()* and *usec_diff()*.

You will find the ARM documentation on the course website especially useful here, because the "QA7: ARM Quad A7 Core" document describes, among many other things, the various clocks and timers at your disposal.

Build It, Load It, Run It

On the course website, you will find a folder named "SD Files." The files inside this folder must be added to your RPi SD card for the RPi to boot. You should also see that the Makefile includes a line for the command *make cp* that copies *kernel7.img* to your SD card. You may need to modify the path in this line to fit your environment – it should be the path to your SD card when it's plugged into your computer.

Once you have it working, show us.

Your Tasks

Your task in this project is to follow the steps detailed above:

- Get an RPi v3 B+ with a power adapter or the right type of USB cable
- Get a NOOBS SD card and an SD card reader (if your laptop doesn't have an SD card slot) **It Must Be A Class-10 Device!**
- Get a case for the RPi board [optional, but a good idea nonetheless]
- Install the ARM Aarch32 cross-compiler on your laptop
- Read the bare-metal programming introduction on the course website
- Write a small timer library

Get it working and demonstrate your working code to the TA during your weekly lab section.