University of New Hampshire
Department of Electrical and Computer Engineering
ECE 562 – Computer Organization
Fall 2019
Lab #2 Notes

Raspberry Pi 3 Model A+

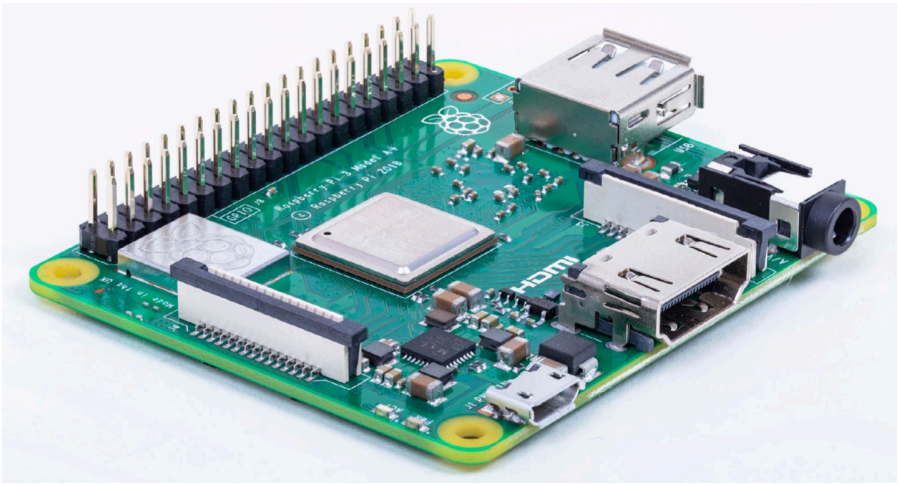

Figure 1. Raspberry Pi 3 Model A+
(https://www.raspberrypi.org/app/uploads/2018/11/Raspberry_Pi_3A_product_brief.pdf)

In this lab you will do bare-metal programming on a Raspberry Pi 3 Model A+, which is a small, affordable, and versatile single-board computer. It features the following:

- Broadcom BCM2837B0 SoC (system-on-chip)
- 512MB LPDDR2 SDRAM
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2/BLE

The SoC contains a multimedia processor, VideoCore IV, and an ARM Cortex-A53 processor with four cores that can operate at 1.4GHz clock frequency.

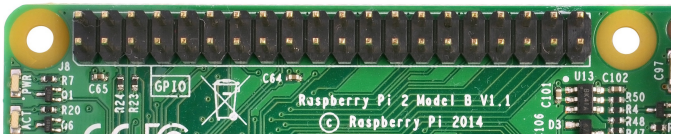

Figure 2. 40-pin male header that can be used for a wide range of purposes.
(https://www.raspberrypi.org/documentation/usage/gpio/)

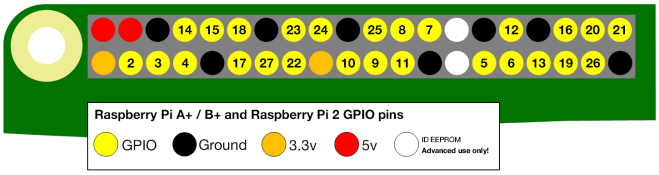

Figure 3. GPIO pin assignments. (https://www.raspberrypi.org/documentation/usage/gpio/)

Raspberry Pi 3 has 40 pins exposed as male headers on its side. These are called GPIO (general-purpose input/output) pins. While some of them are for power (two 5V, two 3.3V,

and eight ground pins) 28 of the pins are connected to the SoC and can be programmed as either inputs or outputs. See course website for reference documents on GPIOs.

SparkFun Pi Wedge

Since the arrangement of the pins is not very convenient, and it is difficult to build a semi-permanent circuit with stray jumper cables, we connect the entire dual row array of pins to a breadboard using the following breakout board and a 40-pin flat ribbon IDC (insulation-displacement connector) cable:

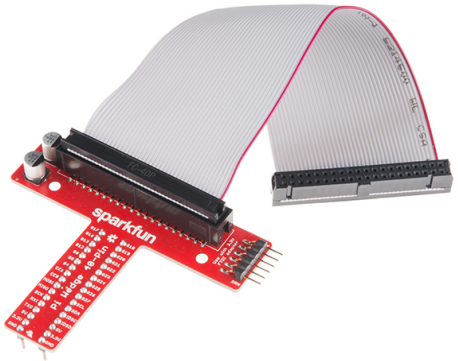

Figure 4. SparkFun Pi Wedge for connecting GPIO pins to a breadboard. (https://www.sparkfun.com/products/13717)

Adafruit FT232H Breakout Board

Raspberry Pi is commonly used as a complete computer rather than a development board. So, one of the features missing for easy experimentation with bare-metal programming is a USB connection to a host computer for loading and debugging programs.

We solve this problem by connecting a USB debugger chip to the GPIO pins. There are many ways to achieve this, but for cost and simplicity, the following Adafruit FT232H Breakout board is used. It packages an FTDI USB to JTAG chip and makes the important pins available for connection on a breadboard.

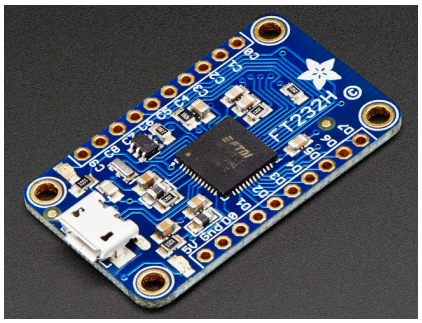

Figure 5 Adafruit FT232H Breakout board. (https://www.adafruit.com/product/2264)

See the course website for relevant datasheets.

USB Cables

Finally, we have a 5V 10 W wall mount AC/DC adapter with a USB outlet and a USB Male A to USB Male Micro-B switched power cable (shown below) for powering the Raspberry Pi and for easily turning it on and off. We also have a commonly used USB Male A to USB Male Micro-B to connect the debugging chip to our host computer.

Figure 6 Adafruit Power Cable with Switch (https://www.adafruit.com/product/2379)

LEDs

The “Hello World!” equivalent of bare-metal programming would probably be turning an LED on and off. So, we are connecting one green and one red LED to the GPIO 6 and GPIO 5 pins (3.3 V), each with 150 Ω resistor for 1.8 V forward voltage and 10 mA current.

microSD Card

As the Raspberry Pi is turned on, some booting code from a read-only memory (ROM) is executed, which initializes some of the peripherals including a microSD card reader. If there is one installed, the bootloader expects to find certain files for further initialization, including `bootcode.bin`, `fixup.dat`, `start.elf`, and finally `kernel8.img` which is the first program we can write to be executed on Raspberry Pi. It is named that way, since it is usually the “kernel” of an operating system. The number 8 comes from ARMv8, which can execute in AArch64 execution mode.

For providing these files, we install a regular microSD card. Every time we want to load a program, we could turn the Raspberry Pi off, remove the SD card, plug into our development machine, create an ELF file named `kernel8.img`, then write it into the SD card, remove and install it again into the Raspberry Pi, and power it back on. But that takes too many steps, let alone increasing the probability of failure of the SD card and/or the card readers.

Instead, we put a minimal infinite loop program on the SD card, and configure the boot process with a `config.txt` file with the line `enable_jtag_gpio=1`, which enables the JTAG functionality. That way, we can use the debugger to load a new application without removing the SD card.

Breadboard and Schematic

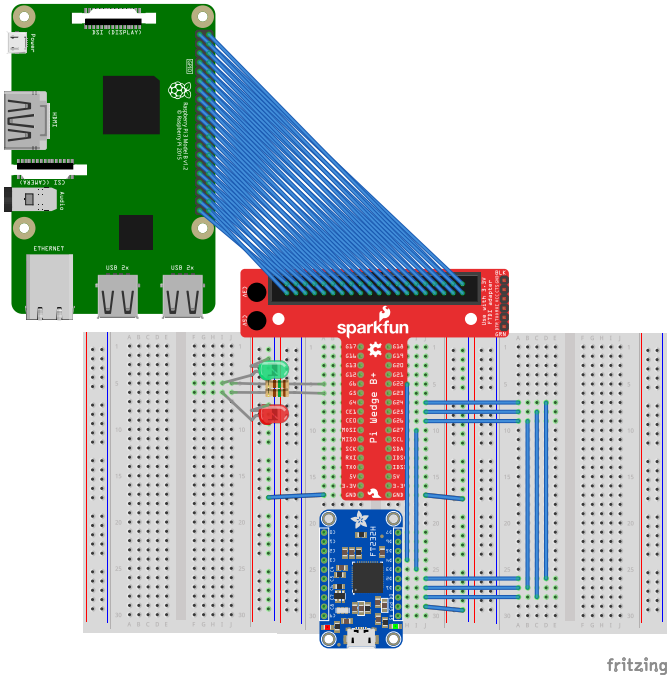

Figure 7 Breadboard layout.

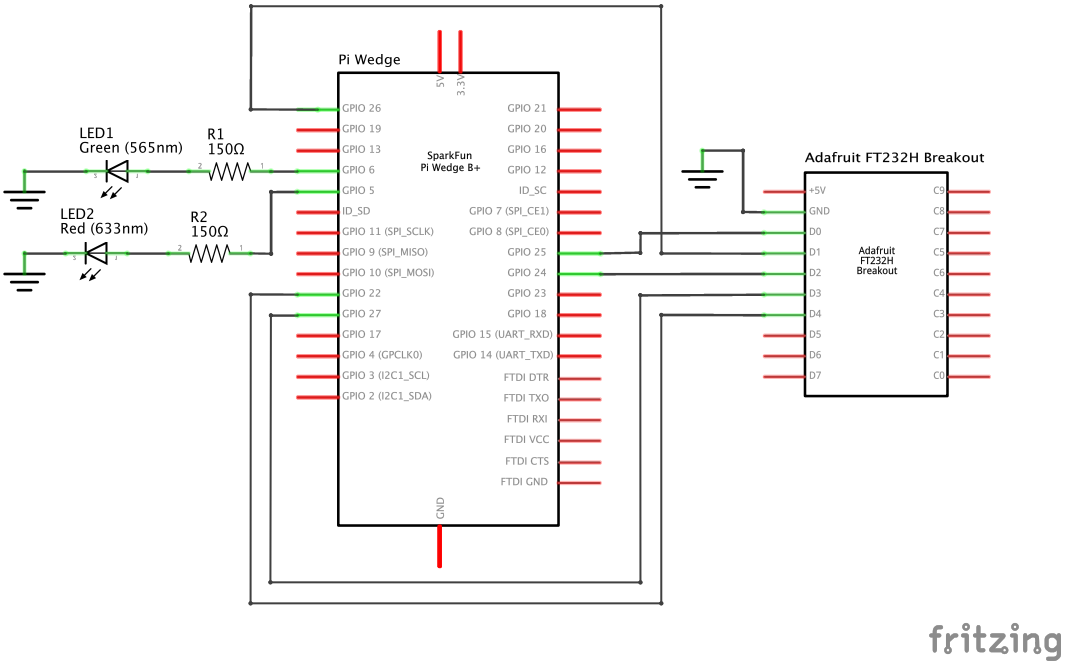

Figure 8 Circuit diagram.

Debugger – OpenOCD

In the first lab, we used an emulated system to host a debug server, to which we connected from a separate debugging program. In this lab, we are replacing the emulator with the real hardware. Certain GPIO pins that we connected to the FT232H board are used to interact with the debugging facilities of the processor using a standard called JTAG. The board can *speak JTAG* with Raspberry Pi and poses as a USB device to the host computer. We can then run a program on the computer that can *speak JTAG* to the USB device, which understands our debug commands. This program will be OpenOCD, the Open On-Chip Debugger. OpenOCD can operate with various boards and interfaces, but we need to configure it for our specific setup. We will do that by using two config files: `ft232h.cfg` and `rpi3.cfg`. See the course website for downloading these files.

When in lab, you need to plug in the USB cable for the debugger. Then turn the Raspberry Pi on from the switch on its cable, before running OpenOCD. When done, close the OpenOCD program (with `Ctrl+C`), turn off Raspberry Pi from the switch on the cable, and unplug the USB cable for the debugger.

The following command runs OpenOCD with our config files. Make sure you are in the same directory as your lab files before running this.

```
openocd -f ft232h.cfg -f rpi3.cfg
```

You could also run `make run-openocd`. You will see an output similar to the following:

```
Open On-Chip Debugger 0.10.0+dev-00921-g263deb38 (2019-07-22-11:55)
Licensed under GNU GPL v2
For bug reports, read
     http://openocd.org/doc/doxygen/bugs.html
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 1000 kHz
Info : JTAG tap: rpi3.tap tap/device found: 0x4ba00477 (mfg: 0x23b (ARM
Ltd.), part: 0xba00, ver: 0x4)
Info : rpi3.a53.0: hardware has 6 breakpoints, 4 watchpoints
Info : rpi3.a53.1: hardware has 6 breakpoints, 4 watchpoints
Info : rpi3.a53.2: hardware has 6 breakpoints, 4 watchpoints
Info : rpi3.a53.3: hardware has 6 breakpoints, 4 watchpoints
Info : Listening on port 3333 for gdb connections
Info : Listening on port 3334 for gdb connections
Info : Listening on port 3335 for gdb connections
Info : Listening on port 3336 for gdb connections
```

The output shows the four cores of the ARM Cortex A-53 processor have been detected, and a GDB server is listening on ports 3333 through 3336 for connections. We can connect our debugging program to these ports for debugging individual cores.

Debugger – GDB

Leaving the OpenOCD terminal open, we open another terminal and run `aarch64-elf-gdb` (or `aarch64-elf-gdb`), then we connect to core #0 by entering `target remote :3333` to the prompt. OpenOCD shows the following:

```
Info : accepting 'gdb' connection on tcp/3333
Info : rpi3.a53.0 cluster 0 core 0 multi core
target halted in AArch64 state due to debug-request, current mode: EL2H
cpsr: 0x000003c9 pc: 0x80004
MMU: disabled, D-Cache: disabled, I-Cache: disabled
Info : New GDB Connection: 1, Target rpi3.a53.0, state: halted
```

The GDB shows the following:

```
Remote debugging using :3333
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000000000080004 in ?? ()
```

We can use the usual GDB commands now. For example:

- `display /i $pc`: Show the next instruction each time the program stops.
- `info register x0`: Show the value of register x0.
- `stepi` (or `si`): Step one instruction exactly.
- `continue` (or `c`): Continue execution.
- `break *0x80000` (or `b *0x80000`): Set breakpoint at address 0x80000.
- `set $pc=0x80000`: Set the value of PC register (the program counter) as 0x80000.
- You can `type` help for more information.

Additionally, we can send commands to the OpenOCD through GDB using the `monitor` command. In order to load our program to the memory of the Raspberry Pi, we will use the following command: `monitor load_image lab2.ihex`. Whenever we load a new program, we have to change the program counter to point to the starting address of our code: `set $pc=0x80000`. Then we can either step through the instructions (with `si`), or just continue running the program (with `c`). When the program is running, we can interrupt and get back to the GDB prompt by pressing `Ctrl+C`.