CS466 Lab 1 -- Hardware, Development Tools and Blinking the LED (First Draft, Pre-Hardware)

Due by Midnight Friday 1-21-2022.

!!! Must use provided lab format on Canvas !!!

Note: This is an individual lab, you are free to collaborate but every student must perform the lab and hand in a lab report. It will be critical that each student is able to develop to the target platform.

I will be checking out a development board to each student. If you lose it, soak it, step on it, or let the smoke out of it during class you will be expected to replace it. They cost around $4 and I will collect one per student at the end of the semester.

Overview:

We will be using a Raspberry Pi Pico board. It's essentially a minimal development board to showcase the RP2040 microcontroller. The ASIC has a processor and a bevy of integrated peripherals that the developer can combine and use to control an arbitrary device.

Benefits of the hardware:

- Very Affordable, Currently around $4.00
- Dual-core Arm Cortex M0+ processor, flexible clock running up to 133 MHz
- 264KB of SRAM, and 2MB of on-board Flash memory
- Castellated module allows soldering direct to carrier boards
- USB 1.1 with device and host support
- Low-power sleep and dormant modes
- Drag-and-drop programming using mass storage over USB
- 26 × multi-function GPIO pins
- 2 × SPI, 2 × I2C, 2 × UART, 3 × 12-bit ADC, 16 × controllable PWM channels
- Accurate clock and timer on-chip
- Temperature sensor
- Accelerated floating-point libraries on-chip
- 8 × Programmable I/O (PIO) state machines for custom peripheral support
- Open source Gnu tool chain, for Windows, Linux, OS-X
- Small and versatile.

Cons of the new hardware:

- No Snazzy IDE, command line tools and debugger.
- One Single-Color (green) LED
- Very little electrical protection on I/O pins (We need to be careful)
    - Can damage dev-board (we've roasted a couple)
    - Potentially could damage host computer USB hardware (we've not done this yet)

Resources:

- Raspberry Pi PICO DevBoard Documentation (https:/www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html)
- Raspberry RP2040 Microcontroller Documentation (https://www.raspberrypi.com/documentation/microcontrollers/rp2040.html#welcome-to-rp2040)
- RP2040 Datasheet (https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf)
- Raspberry Pi Pico SDK documentation (https://raspberrypi.github.io/pico-sdk-doxygen/)

Lab Preparation:

- Take some time to browse the above documentation. Don’t install the development tools yet as I’d like everyone to have the same directory structure fir convince. We will be using a command line ARM GCC toolchain
- Locate the board schematic. One of the lessons I learned in drafting this lab is that the button on the poco board is not directly available   There is an example in the pico devkit but it has to jump though hoops on failing to read the QSPI memory device on the board and assuming that is button press… We’ll setup two GPIO pins to serve as two operator inputs.

- Take a slightly longer look at the microcontroller datasheet. This is a large pdf (654 Pages).. While we will be looking into specific sections in detail for now I want you to read sections 1 (16 pages), 2.1.9.1 and 2.1.9.2

    - If you are not familiar with data sheets this can be a daunting document. Part of what we will be discussing in lab is how to not freak out when presented with all the data. The EE students have an early edge here having been exposed to data sheets before but the CS guys get it back later in the class.

Objective:

This lab is mostly to familiarize students with the Pico development board and development tools. Making use of some GPIO pins will require some basic understanding of the schematic and understanding some of the GPIO section of the microcontroller reference manual.

Note that a popular convention is used on the blinky.c code to write to a fixed address (where controlling registers in the ASIC are located.) The code has the line

```
GPIO_PORTF_DIR_R = (1<<3);
```

And if you follow the header files back we see the definition

```
#define GPIO_PORTF_DIR_R        (*((volatile uint32_t *)0x40025400))
```

In short `GPIO_PORTF_DIR_R` is a literal that is forced to resolve as a pointer to an address. The assignment of (1<<3) to the token results in the value `0x00000008` being written to address `0x40025400`. I will use this notation a lot during class so be sure that you understand it.

The 'volatile' storage type qualifier informs the compiler not to optimize or otherwise play with the storage location or method.

Since all 32 bits of the register may have some behavior this instruction would set our interesting bit but clear the other 31 bits. The quick and dirty way for us to prevent stepping on the register contents is to perform a read-modify-write operation which in C can be coded as

```
GPIO_PORTF_DATA_R |= (1<<3);
```

Which would read all 32 bits, set the `0x00000008` bit, and then write the whole 32 bit word. The other 31 bits retain their pervious state. We have to perform some logic tricks to clear the same bit yet remain readable in the code.

```
GPIO_PORTF_DATA_R &= ~(1<<3);
```

Lab Work

1. ☐ For LAB 1 I would like everyone to complete their work on a Linux system, their own or the lab computers

2. ☐ All of the lab material will be available on a my git repo. Choose a directory to use to locate your cs466 project in (I recommend ~/src). From that directory run the command:

   ```
   $ git clone https://github.com/milhead2/cs466_s22
   ```

   Using my standard this should create a directory ~/src/cs466_s22 with the first set of class files. Keep the directory tree in this format and support will be easier for your labs.
   The README.md file on the class repo gives instruction on how to setup the directory.

   I only 'add' files to this repo so you are free to work in your copy of the repo. It would be 'more proper' to fork the repo. As I only add files we should not see conflicts if you pull to get added files.

   The repo README.md has data that should help you setup your dev environment

3. ☐ Familiarize yourself with the pico development board.

4. ☐ Check that the complier is available on the system by running the commands that all should give a reasonable response. If your system does not have an `arm-none-eabi-gcc` application you will need to download and install it from https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads

```
$ arm-none-eabi-gcc –version
```

5. ☐ The Pico board does not have any default program loaded so when you plug it in no indication that the board is doing anything except for the occurrence of the directory above. In order to blink the LED we need to install the `blink` example program. So far I have had to reboot the pico with the button pressed to enter usb-drive mode.

a) This is cumbersome and will cause a lot of wear on the Micro-USB connector, I'll keep looking for another wat to reload but that's it for now.

6. ☐ Get your development environment setup so that you can build the oversimplified Blinky program that will be provided in lab.

a) When you look at the class repo online the readme outlines the steps that I took to get everything compiled. Wis a bit tricky and took me the lions share of this afternoon to get it worked out to your lab1 starting point.

7. ☐ Locate the schematic for the poco board on the board website. When compared to the lab1.c code is it clear what is occurring?

a) Question: Describe the expected operation of the lab1.c provided program

--- You can proceed to this point without hardware
--- I will update the following steps before lab on Thursday

8. ☐ Verify that when you plug in the development board on Linux it is recognized by the OS. After you have the board plugged in and running, type a dmesg command to display the latest system stuff that occurred on the workstation;

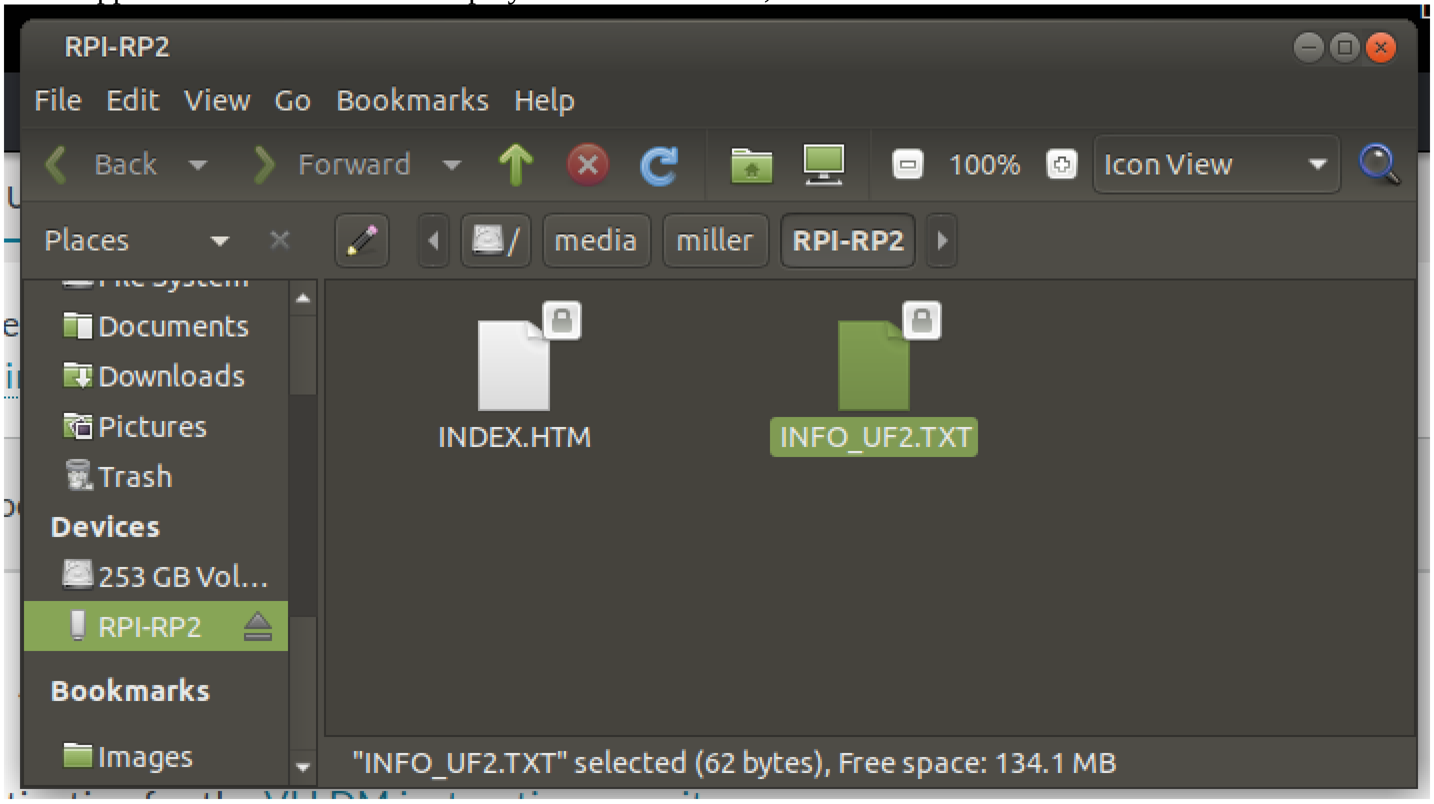
```
$ dmesg
```

'dmesg' will display a gob of stuff but the last few lines are most interesting. You should see something like…

```
[102582.178254] usb 2-1.6: new full-speed USB device number 15 using ehci-pci
[102582.287864] usb 2-1.6: New USB device found, idVendor=2e8a, idProduct=0003, bcdDevice= 1.00
[102582.287869] usb 2-1.6: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[102582.287871] usb 2-1.6: Product: RP2 Boot
[102582.287874] usb 2-1.6: Manufacturer: Raspberry Pi
[102582.287876] usb 2-1.6: SerialNumber: E0C912952D54
[102582.288441] usb-storage 2-1.6:1.0: USB Mass Storage device detected
[102582.289099] scsi host10: usb-storage 2-1.6:1.0
[102583.299694] scsi 10:0:0:0: Direct-Access     RPI      RP2              2    PQ: 0 ANSI: 2
[102583.300536] sd 10:0:0:0: Attached scsi generic sg5 type 0
[102583.301990] sd 10:0:0:0: [sde] 262144 512-byte logical blocks: (134 MB/128 MiB)
[102583.303177] sd 10:0:0:0: [sde] Write Protect is off
[102583.303182] sd 10:0:0:0: [sde] Mode Sense: 03 00 00 00
[102583.304309] sd 10:0:0:0: [sde] No Caching mode page found
[102583.304317] sd 10:0:0:0: [sde] Assuming drive cache: write through
[102583.351867]  sde: sde1
[102583.382682] sd 10:0:0:0: [sde] Attached SCSI removable disk
```

This shows that Linux has seen the device and understands what it is.

In the default configuration the board will load the last program you downloaded. If you don't see the board appear as a USB drive and display a file browser like;



"INFO_UF2.TXT" selected (62 bytes), Free space: 134.1 MB

Then unplug the board and hold down the BOOTSEL button while you plug it in.

- Raspberry claims that you cannot brick this board and holding the BOOTSEL button will 'always' return the board to default waiting-for-image program… We'll see if that's true.

9. ☐ In order to proceed with the remaining lab steps we will need three additional GPIO's in play. Setup GPIO's 21 and 22 as inputs.

10. ☐ Switch the blinking LED back to green and measure the frequency of the blinking LED. Modify the high speed delay routines to get as close as possible to a 1Hz blink frequency.

11. ☐ Add code to read the status of SW1 and SW2. You will need to enable new GPIO pins for input and internal pull-up resistors.

a) (#3) Do both switches follow the same rules (hint, hint, maybe the datasheet says something).

12. ☐ Modify the code so that:

a) The green LED always blinks at 1.0Hz
b) By default the red and blue LED, are off.
c) Pressing SW1 will cause the red led to flash 20 times at 15Hz
d) Pressing SW2 will cause the blue led to flash 10 times at 13Hz
e) This should work with any combination of SW1 and SW2 (ugly blinkin!)
f) Use only the button press as a trigger, the led should blink their required number of times no matter how long the button is pressed.

13. ☐ Verify the frequency (or as close as you can get) with the scope.

a) (#4) What is the purpose of the blinky.ld file?
b) (#5) What is the purpose of the startup_gcc.c file?

14. ☐ Without a friendly operating system to startup our code this environment is a bit more bare than you are 'probably' used to.

a) (#6): Describe the execution path from processor reset until main is called.

15. Write up your lab using the lab format provided on Blackboard. Include your program as a fixed spaced (`I recommend Lucida Console`) addendum to your lab. I will cut points for proportionally spaced code pasted in the end of the lab.

Submit your lab as a 'SINGLE' PDF file on Blackboard. Name the file
`CS466L01_Name(s)_Report.pdf`

…. Not a .zip File
…. Not a set of files
…. Not a set of .jpg files from your phone..

(I'm not too much a jerk.. Using a single PDF makes grading, feedback and organization of reports easier for everyone)

16. Due by Midnight Friday 1-24-2020