Lab: write your own code to control the r/pi; throw ours out.

Important: as always, read and complete the PRELAB before lab!

The first lab was just setup. Now we get to the fun part: you'll use the Broadcom document (`../../docs/BCM2835-ARM-Peripherals.annot.PDF`) to figure out how to write the code to turn the GPIO pins on/off yourself as well as reading them to get values produced by a digital device. You'll use this code to blink an LED and to detect when a capacitive touch sensor is touched.

Make sure you read the GPIO and device memory crash course in the guides directory.

Sign off

If you've never used a pi before, to get credit for the lab show the following:

1. That `code/2-blink.c` correctly blinks two LEDs on pin 20 and 21 in opposite orders (i.e., if 20 is on, 21 should be off and vice versa). (This will point out a subtle mistake people make reading the docs).
2. That `code/3-input.c` turns on an LED connected to pin 20 when pin 21 is connected to 3v (either directly, via an LED, or via a touch sensor or whatever other device you might want to try).

If you have used a pi before, implement `gpio_pullup` and `gpio_pulldown`. Ideally you also do some extensions.

- Extension: That you can forward the signal from one pi to another. This requires the ability to run two pi's at once (as described in 0-blink).

  Not everyone will be able to do this if you need to buy additional adaptors for your laptop --- make sure you can do this part very soon, since we need two pi's at once for networking.

Part 1. Make blink work. (30 minutes)

You'll implement the following routines in `code/gpio.c`:

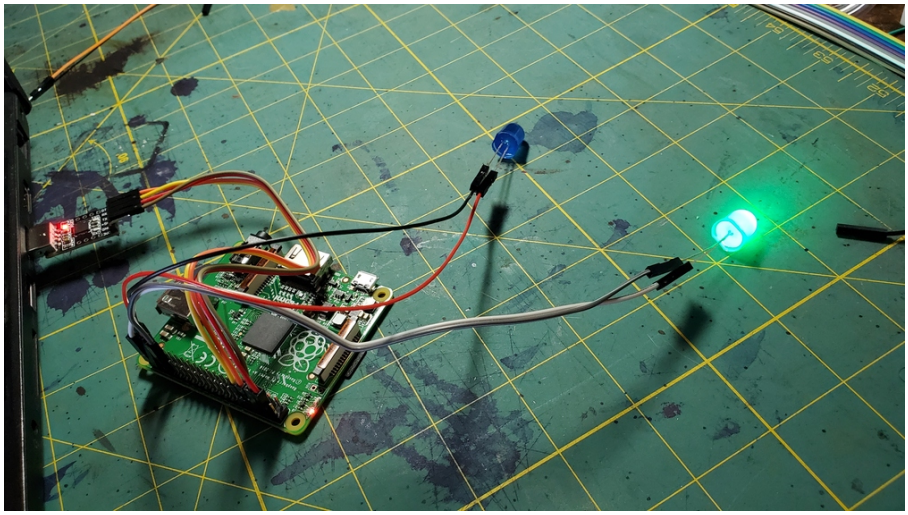
1. `gpio_set_output(pin)` which will set `pin` to an output pin. This should take only a few lines of code.
2. `gpio_set_on(pin)` which will turn `pin` on. This should take one line of code.
3. `gpio_set_off(pin)` which will turn `pin` off. This should take one line of code.
4. After doing so, wire up an LED pin to pin 20, power-cycle the pi, and use the bootloader to load the code:

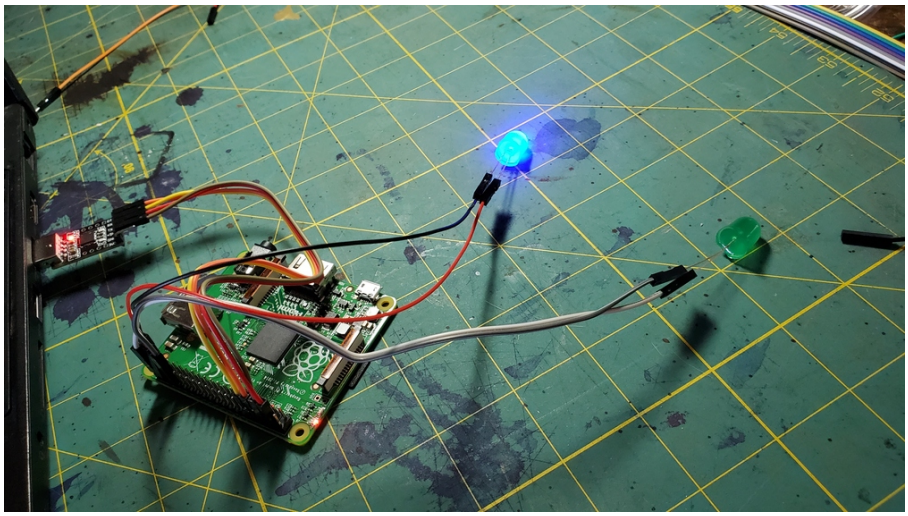
```
# unplug the pi.
# connect an LED to pin 20
% cd code
# do your edits
% make
% pi-install 1-blink.bin
# the LED on pin 20 should be blinking.
```

5. Make sure that `code/2-blink.c` also works:

```
# unplug the pi
# connect an LED to pin 21
# plug back in to reset.
% pi-install 2-blink.bin
# the LEDs on pin 20 and pin 21 should be in opposite orders.
```

Success looks like:





Hints:

1. You write `GPFSELn` register (pages 91 and 92) to set up a pin as an output or input. You'll have to set GPIO 20 in GPFSEL2 to output.
2. You'll turn it off and on by writing to the `GPSET0` and `GPCLR0` registers on page 95. We write to `GPSET0` to set a pin (turn it on) and write to `GPCLR0` to clear a pin (turn it off).
3. The different `GPFSELn` registers handle group of 10, so you can divide the pin number to compute the right `GPFSEL` register.
4. Be very careful to read the descriptions in the broadcom document to see when you are supposed to preserve old values or ignore them. If you don't ignore them when you should, you can write back indeterminate values, causing weird behavior. If you overwrite old values when you should not, the code in this assignment may work, but later when you use other pins, your code will reset them.

```
// assume: we want to set the bits 7,8 in <x> to v and
// leave everything else undisturbed.
x &=  ~(0b11 << 7);   // clear the bits 7, 8  in x
x |=   (v << 7);   // or in the new bits
```

5. You will be using this code later! Make sure you test the code by rewiring your pi to use pins in each group.

Part 2. Make input work.

Part 1 used GPIO for output, you'll extend your code to handle input and use this to read input. At this point you have the tools to control a surprising number of digital devices you can buy on eBay, adafruit, sparkfun, alibaba, etc.

What you will do below:

1. Implement `gpio_set_input` --- it should just be a few lines of code, which will look very similar to `gpio_set_output`.

   Make sure you do not overwrite a previous configuration in `fsel` for other pins! You code will likely still work today, but later if you have multiple devices it will not.

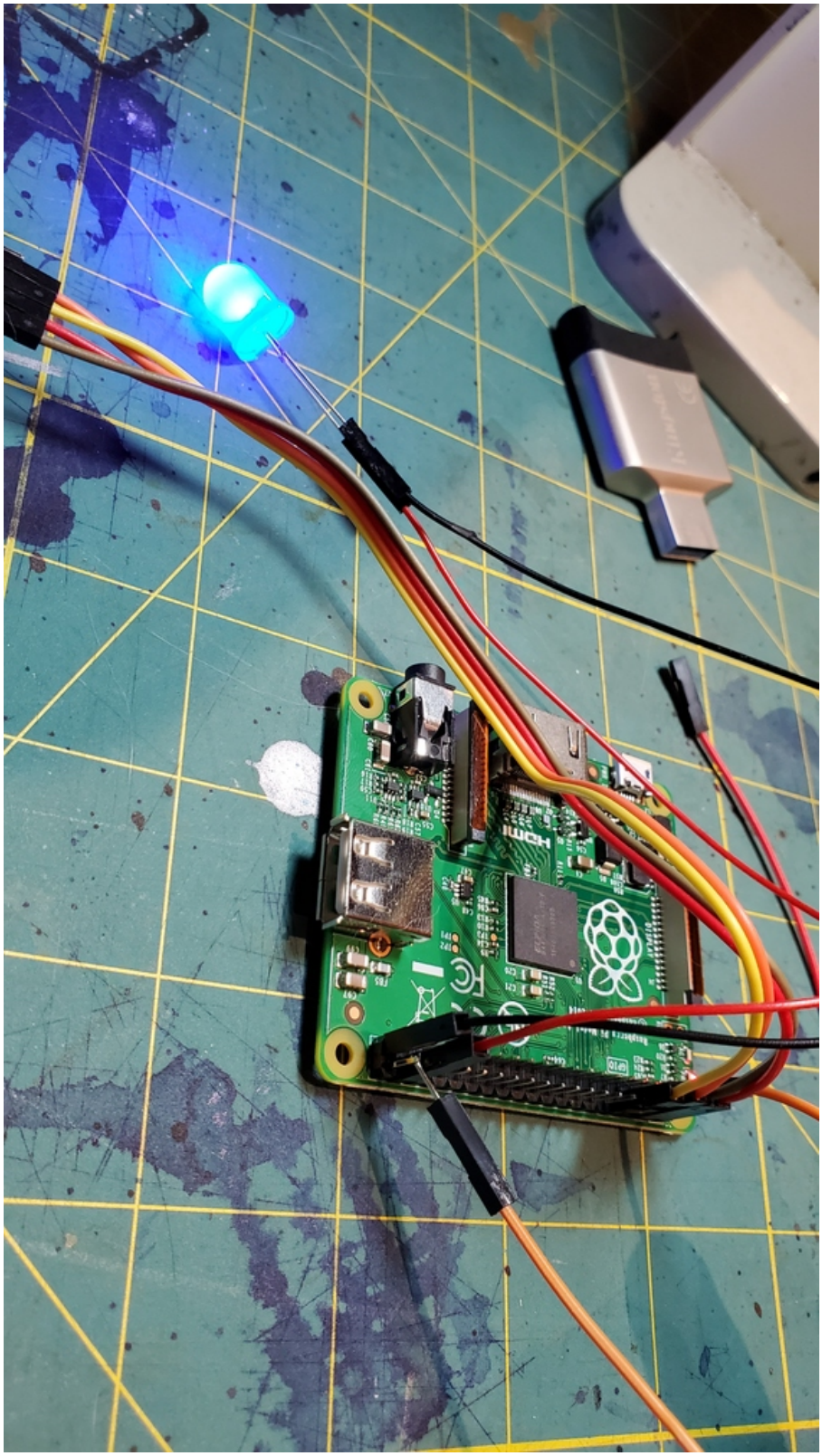
2. Implement `gpio_read` --- make sure you do not return bits that are spurious! This can lead to garbage results.

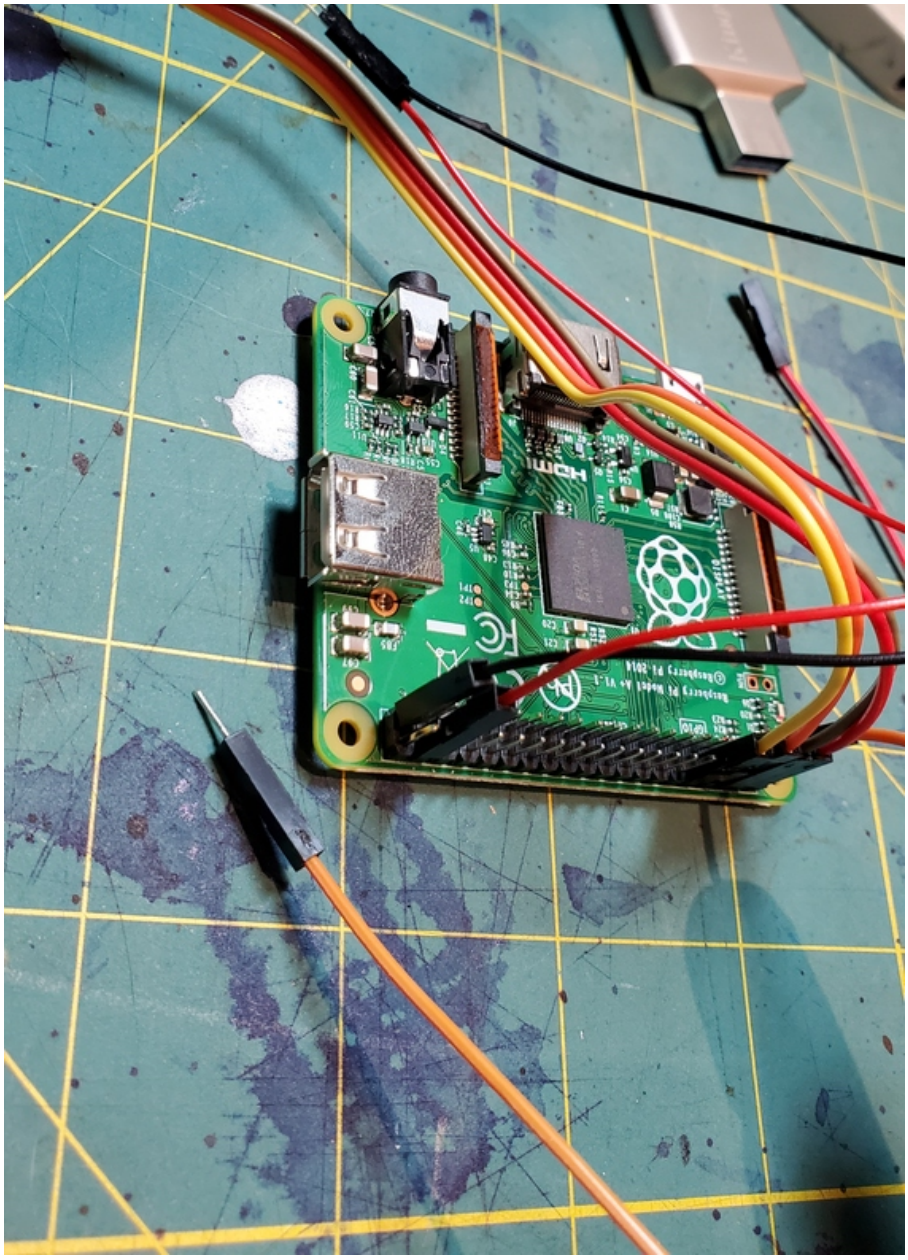
3. Connect the positive leg of an LED to one of the pi's 3v outputs: DO NOT CONNECT TO 5V! Test it by touching the other LED leg to ground: it should turn on.

4. Run the code:

```
% make
% pi-install 3-input.bin
# touch the LED leg to pin 21: the second LED (connected to pin 20) goes on.
# remove the LEG touching pin 21: LED (connected to pin 20) goes off.
```

5. Success looks like the following (note: we used a bare jumper, but that is likely asking for trouble):





Extension: Forward one pi signal to another.

We now do a cool trick: transparently forward signals from one pi to another. While mechanically trivial, this is a "hello world" version of some deep topics.

What to do:

1. Hook up pin 20 from one pi (call this pi-1) to pin 21 of the other (pi-2).

```
Note: strictly speaking, when connecting two devices we must
connect (share) ground as well.  However, since these are both
powered from your laptop they already do.
```

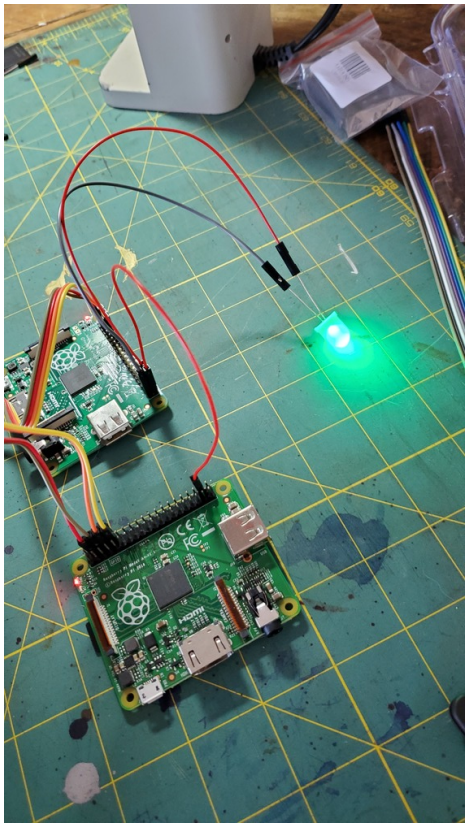
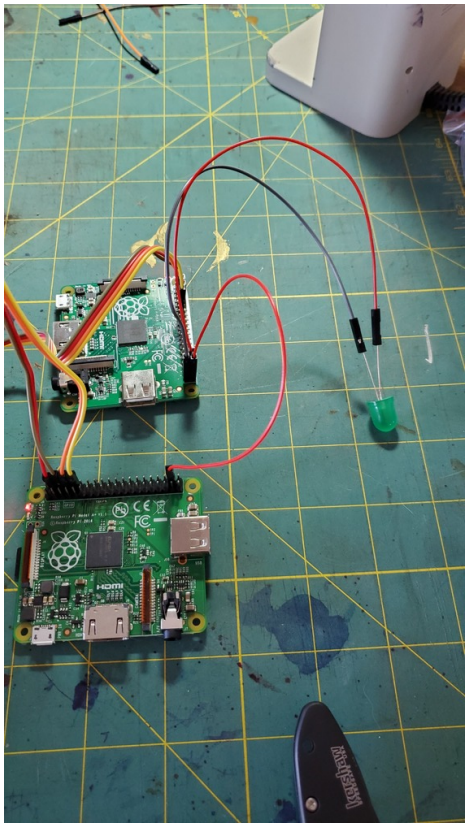
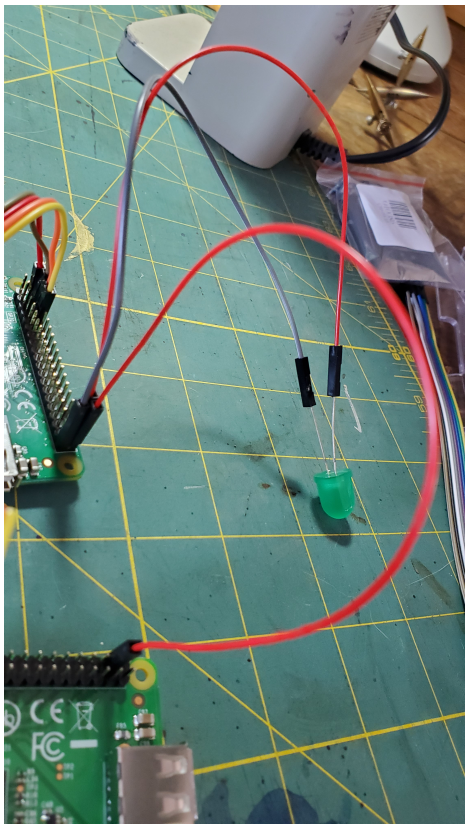
2. Plug pi-2 in and run `3-input.bin` input program.

   % pi-install /dev/ttyUSB0 code/3-input.bin

3. Plug pi-1 in and run `1-blink.bin` program.

```
% pi-install /dev/ttyUSB1 code/1-blink.bin
```

3. Success looks like:

This doesn't seem like much, but is shows the glimmer of several deeper tricks:

1. It shows how you can take code written with very strong assumptions and --- without any software changes --- dramatically change its behavior, correctly. In this case we take a system that assumed it running locally, controlling its own device and made it transparently control a remote device (or even many).

Among other examples this trick (on a much grander scale) is similar in spirit to how virtual machines such as VMWare work.

2. You can view this single wire as a very primitive network or bus. On = sending a 1 bit. Off = sending a 0. And, if we can send 0 and 1 we can send anything.

   With some simple changes (we will do these in later labs) you can use this method to send general-pupose messages between pi's. And if you add a second wire, you easily can do so with incredibly low-latency: almost certainly lower than between two full-fledged Unix servers. (One of many examples where we will be able to write custom, clean, simple code that is far faster or more powerful than a full-fledged "real" system.)

Extension: Implement `gpio_set_pullup` and `gpio_set_pulldown`

These are described, with much ambiguity, in the Broadcom document on page 101 (using my pdf reader page numbering). Some issues:

1. They say to use clock delays, but do not way which clock (the pi clock? The GPU clock? Some other clock?)
2. A straight-forward reading of steps (5) and (6) imply you have to write to the `GPPUD` to and `GPPUDCLK` after setup to signal you are done setting up. Two problems: (1) write what value? (2) the only values you could write to `GPPUD`, will either disable the pull-up/pull-down or either repeat what you did, or flip it.

In other domains, you don't use other people's implementation to make legal decisions about what acts are correct, but when dealing with devices, we may not have a choice (though, in this case: what could we do in terms of measurements?).

Two places you can often look for the pi:

1. Linux source. On one hand: the code may be battle-tested, or written with back-channel knowledge. On the other hand: it will have lots of extra Linux puke everywhere, and linux is far from not-buggy.
2. `dwelch76` code, which tends to be simple, but does things (such as eliding memory barriers) that the documents explicitly say are wrong (and has burned me in the past).

For delays: [Linux] (https://elixir.bootlin.com/linux/v4.8/source/drivers/pinctrl/bcm/pinctrl-bcm2835.c#L898) uses 150 usec. [dwelch76] (https://github.com/dwelch67/raspberrypi/blob/master/uart01/uart01.c) uses something that is 2-3x 150 pi system clock cycles. The the general view is that we are simply giving the hardware "enough" time to settling into a new state rather than meeting some kind of deadline and, as a result, that too-much is much better than too-little, so I'd go with 150usec.

For what to write: from looking at both Linux and dwelch67 it *seems* that after steps (1)-(4) at set up, you then do step (6), disabling the clock, but do not do step (5) since its simply hygenic.

Since we are setting "sticky" state in the hardware and mistakes lead to non-determinant results, this is one case where I think it makes sense to be pedantic with memory barriers: do them at the start and end of the routine. (NOTE, strictly speaking: if we are using the timer peripheral I think we need to use them there too, though I did not. This is a place where we should perhaps switch to using the cycle counter.)

If you get confused, this page gives an easier-to-follow example than the broadcom.

Extension: Break and tweak stuff.

If you finish, the are a bunch of ways you can "kick the tires" on your system to understand better what is going on.

1. Change the delay in the blink code to increasingly smaller amounts. What is going on?
2. Add the reboot code below (we'll go into what different things mean) so that you don't have to unplug, plug your rpi each time:

```
// define: dummy to immediately return and PUT32 as above.
void reboot(void) {
     const int PM_RSTC = 0x2010001c;
     const int PM_WDOG = 0x20100024;
     const int PM_PASSWORD = 0x5a000000;
     const int PM_RSTC_WRCFG_FULL_RESET = 0x00000020;
     int i;
     for(i = 0; i < 100000; i++)
          nop();
     PUT32(PM_WDOG, PM_PASSWORD | 1);
     PUT32(PM_RSTC, PM_PASSWORD | PM_RSTC_WRCFG_FULL_RESET);
     while(1);
}
```

Change your code to just loop for a small fixed number of times and make sure reboot() works.

3. Force the blink loop to be at different code alignments mod 64. If you look at `1-blink.list` you can see the executable code and the addresses it is at. Do you notice any difference in timing? (You may have to make your delay longer.) What is going on?

Additional information

More links:

1. Useful baremetal information: (http://www.raspberrypi.org/forums/viewtopic.php?t=16851)
2. More baremetalpi: (https://github.com/brianwiddas/pi-baremetal)
3. And even more bare metal pi: (http://www.valvers.com/embedded-linux/raspberry-pi/step01-bare-metal-programming-in-cpt1)
4. Finally: it's worth running through all of dwelch's examples: (https://github.com/dwelch67/raspberrypi).