Wireless & Communication in the IoT, Fall 2022: Homework #1

Due: Friday, September 30th at 8:00pm on Gradescope

- Please either type in or if handwritten, make your answers as neat as possible.
- Enter your answers completely in the text box right below the question. Answers that appear outside of the box might not be graded.

The goal of this homework is largely to act as a quick refresher on low-level C concepts which you will need to use in lab and to understand and follow along in some of the lecture material. A lot of understanding various networking designs comes down to understanding how bits and bytes are arranged at various stages.

Background: Pointers & Peripherals

Most peripherals in embedded systems (and modern computing more generally) are interfaced with via *Memory-Mapped I/O (MMIO).* Basically, there are addresses that don't point to RAM or other traditional memory, but instead point to pieces of hardware. We call these *hardware registers* or often just *registers*.

One simple peripheral is *General-Purpose I/O (GPIO)*. A GPIO pin is a real, physical pin that comes out of the processor, which the processor can set to logic low (aka, 0 V), logic high (aka, 3.3 V; or whatever the system supply voltage level is), or don't care (commonly called *tristate* or sometimes *floating*, this is when the processor does not drive the pin high or low, so it will just float around randomly).

GPIOs can be controlled with two registers, an *OutputControl* register, which indicates "(1): the processor should drive a value on this pin, or (0): let it float" and a *GPIOLevel* register, which indicates the current value (0 or 1) of a GPIO pin.

Q1: Basic Operation [5pts]

Assume we are working with a 32-bit microcontroller that has 32 GPIO pins. There is one output control register located at address **`0x4000_1000`** and one level register at address **`0x4000_1004`**. Both registers are initialized to all 0's at startup. Each GPIO is mapped to one bit in each register; i.e., pin 0 is controlled by bit 0. Complete the following code snippets:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
int main() {
  // Declare pointers that will allow you to access the GPIO registers:
          uint32_t* gpio_control =               ;
  volatile uint32_t* gpio_level   =               ;

  // Set GPIO Pin 0 to output mode, and set the value of Pin 0 to high
                                                  ;
                                                  ;

  // Read the value of Pin 1, and print it
  bool pin1 =                            ;
  printf("Pin 1 is %s\n", pin1 ? "high" : "low");
}
```

Q2: Helper Functions [4pts]

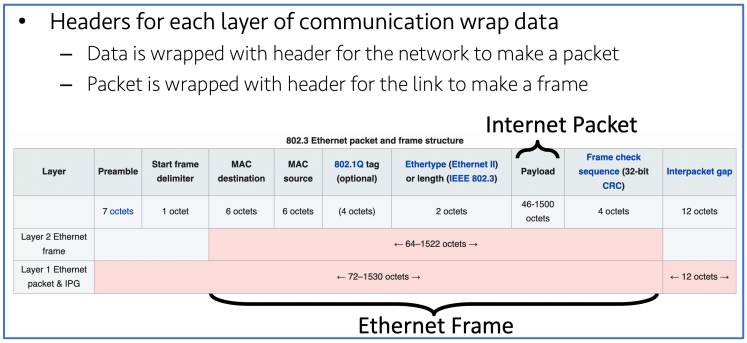
Working with the same device as Q1, fill in the following helper functions. Be careful that when your helper function changes one pin that it does not accidentally change others as well…

```
// Return whether the specified pin is currently configured as an output or not
bool is_output(unsigned pin) {}


// Set the level of (only) the specified pin
void set_level(unsigned pin, bool level) {}
```

Q3: Many Views of Memory [2pts]

Lecture described how networks are layered. A key idea of this layered model is that **many different pieces of code will look at the same bytes in memory in different ways**.



In practice, when software talks to hardware, it generally sends over one, big giant buffer. For example, if I wanted have a radio receive a packet, I have to tell that radio *where* it should put the packet. The radio does not know or care whether it's a TCP or UDP packet, whether it is HTTP or CoAP, etc, it just knows how big the whole packet is. As the packet flows up the reception chain, each layer just looks at a different part of the same buffer. A simplified view of some receive code then might look like this:

```
// Note: In embedded systems, we generally use static allocation
// for everything. i.e., you do not use malloc() or new
//
// This helps ensure at *compile-time* that you have enough space
// for everything.
uint8_t buffer[MAX_PACKET_LENGTH];
radio_receive(buffer, MAX_PACKET_LENGTH);

struct eth*   eth_frame = parse_raw_packet(buffer);
struct ipv4* ipv4_frame = parse_eth(eth_frame);
struct udp*   udp_frame = parse_ipv4(ipv4_frame);
uint8_t*        payload = udp_frame->payload;

printf(" buffer begins at: %p\n", buffer);
printf("    eth begins at: %p\n", eth_frame);
printf("   ipv4 begins at: %p\n", ipv4_frame);
printf("    udp begins at: %p\n", udp_frame);
printf("payload begins at: %p\n", payload);
```

Given the partial output of this code, complete the rest. **You may assume no optional features or anything complicated is happening here.** You will find definitions of the packet structure at each layer very helpful.

```
 buffer begins at: 0x20004000

    eth begins at: 0x20004008

   ipv4 begins at: 0x20004016

    udp begins at:

payload begins at:
```

Q4: Read the syllabus [2pts]. What is one piece of information (not counting obvious things such as the topics, grading, etc) on the syllabus for this course that you have not seen in the syllabi from your other classes?

Q5: Read all your syllabi [2pts]. What is one piece of (useful, ideally) information that is on the syllabus of a different course you are taking this term that is not on the syllabus for this course? (Again, excluding trivial/obvious differences such as course topics.) What class is it? Provide a link to the syllabus.[1]

[1] It is the policy of the CSE department that all courses must have publicly available syllabi. If you have having trouble getting a public link for a CSE syllabus, please ask your instructor to post one. If you are using an example from a non-CSE class and there is no public syllabus available, I would encourage you to ask your instructor to make their syllabus public, but is okay to simply indicate that there is no public copy of the syllabus available.

Extra Credit [1 pt]: 3C Survey

Researchers at Duke University are interested in understanding more about the cultural competence of computer science students. To help us with this research, we ask that you please complete the Cultural Competence in Computing Survey. Please review the following consent form (which details more about the survey) and begin the survey. It should take ~5-7 min to complete.

You may possibly be asked to do this survey by multiple classes. If so, you only need to do the survey once, and you may copy the survey completion information to each class.

If you finish the survey, please provide screenshot of completion / other evidence here: