CS397/497, Winter 2023
Homework: Background

This is an **individual** assignment. Submission is through Gradescope. You can submit a scanned version of this document, a computer edited version, or simply paper with the answers. Please be careful to ensure that answers are labeled and legible.

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.

If you run into issues on this homework, please reach out to the professor! We're happy to help review background material with you.

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.

For more background here, take a look at this lecture on Microcontroller I/O.

Q1: Basic Operation [30pts]

Assume we are working with a 32-bit microcontroller that has 32 GPIO pins. Each GPIO is mapped to one bit in each register; i.e., pin 0 is controlled by bit 0. There are two relevant GPIO registers:

- **`0x4000_1000`** configures whether the pin is an input (0) or output (1)
- **`0x4000_1004`** controls or reads the current level of the pin (0 or 1)

Both registers are initialized to all 0's at startup.

Complete the following code snippet:

```
#include <stdbool.h>

#include <stdint.h>

#include <stdio.h>

int main(void) {

  // Declare pointers that will allow you to access the GPIO registers:

           uint32_t* gpio_config =                         ;

  volatile uint32_t* gpio_value  =                         ;


  // 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");

  return 0;

}
```

Q2: Helper Functions [40pts]

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

bool is_output(uint32_t pin) {}


// Set the level of (only) the specified pin

void set_level(uint32_t pin, bool level) {}
```

Q3: Many Views of Memory [20pts]

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 to have a radio receive a packet, I must 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. **Beware**: the addresses are in hexadecimal!!

```
 buffer begins at: 0x20004000

    eth begins at: 0x20004008

   ipv4 begins at: 0x20004016

    udp begins at:

payload begins at:
```