CS433, Spring 2026
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 and Networking 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. Depending on your background, some of this might be new to you, and that's okay! This is your call to learn how it works now before it matters later.

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 Embedded Software, specifically the Memory-Mapped I/O section.

Q1: Basic Operation [1 point]

Assume we are working with a 32-bit little-endian 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:

- `0x40001000` configures whether the pin is an input (0) or output (1)
- `0x40001004` 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 ONLY affecting the proper pin(s) and leaving other pins unaffected:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
int main(void) {
  // Declare pointers that will allow you to access the GPIO registers:
  volatile uint32_t* gpio_config = 0x40001000;
  volatile uint32_t* gpio_value  = 0x40001004;

  // 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 [2 points]

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

```
// Pointers that will allow you to access the GPIO registers
volatile uint32_t* gpio_config = 0x40001000;
volatile uint32_t* gpio_value  = 0x40001004;


// Determine if the pin is currently configured as an output
// pin is the number of the pin, from 0 to 31
bool is_output(uint8_t pin) {}


// Set the level of only the specified pin
// pin is the number of the pin, from 0 to 31
// level is true for high (1) or false for low (0)
void set_level(uint8_t pin, bool level) {}
```

Q3: Many Views of Memory [1 point]

Lecture describes (on Tuesday) 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.** Here's a helpful explanation of the IPv4 packet structure. That will help you get one of the answers and you'll need to do your own research to find the other. **Beware**: the addresses are in hexadecimal!!

```
 buffer begins at: 0x20004000
    eth begins at: 0x20004008
   ipv4 begins at: 0x20004016
    udp begins at:
payload begins at:
```

Q4: Data Transmission [1 point]

Data is transmitted in packets, which have some overhead and some maximum amount of payload length. Two measures of data transmission are throughput and goodput. Throughput measures the total number of bits transmitted per second, while goodput only measures payload bits transmitted per second.

Consider the following communication system. Packets are up to 146 bytes in total size. 18 bytes of that are metadata (headers, checksums, etc.) while up to 128 bytes are payload data. One maximum-sized packet can be sent every 200 ms (assume a TDMA method is used for ensuring this dedicated transmission slot). Smaller packets take proportionally smaller amounts of time.

1. What is the total throughput of this system in bits per second?

2. What is the total goodput of this system in bits per second?

3. How long would it take to transmit a payload that is 1500 bytes in size in seconds?