Wireless for the IoT — Homework 1

- 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.

Name: **Robert Owens**

Computing ID: **rao7utn**

Q1: Pointers and Buffers [5pts]

Implement the following function in valid, compileable C code according to the specification in the comment. You may use standard C function such as memcpy. Your code should be robust to buffer overflows.

```
// This function adds a header at the beginning of the `dst` buffer, then
// copies up to `pkt_len` bytes of `pkt` into the buffer immediately after
// the header.
//
// If the header will not fit in `dst`, `ret` is set to -1. Otherwise, `ret`
// contains the number of bytes from `pkt` copied into the `dst` buffer.
//
// The header format is:
// - 2 bytes: the length of the entire output buffer, including the header.
// - 4 bytes: the `id`.
// The header is in network byte order (big endian).
void insert_header(uint8_t* dst, uint16_t dst_len,
                   uint8_t* pkt, uint16_t pkt_len,
                   uint32_t id,
                   int*     ret) {

    // Buffer overflow check
    if (dst_len < pkt_len + 6) {
        *ret = -1;
        return;
    }

    // Create header
    unsigned short low = (dst_len & 0x00FF) << 8;
    unsigned short high = (dst_len >> 8) & 0x00FF;
    *dst = low | high;
    dst += 2;

    //id - switch endianness
    uint32_t low = (dst_len & 0x000000FF) << 24;
    uint32_t low_mid = (dst_len & 0x0000FF00) << 8;
    uint32_t high_mid = (dst_len & 0x00FF0000) >> 8;
    uint32_t high = (dst_len >> 24) & 0x000000FF;//prevent right-shift from filling
with 1

    *dst = low | low_mid | high_mid | high;
    dst += 4;

    //copy over the packet
    memcpy(dst, pkt, pkt_len);
    *ret = pkt_len;

}
```

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.

embedded.com has a decent tutorial on programming with MMIO.

Q2a: Basic Operation [4pts]

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—your code should compile without any warnings or errors using a standard C compiler:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

// Declare pointers that will allow you to access the GPIO registers:
uint32_t* gpio_control = 0x40001000;
volatile uint32_t* gpio_level = 0x40001004;

int main() {
  // Read the value of Pin 0, and print it
    while ((*gpio_control & 0x1) == 0)
        wait();

    bool pin0 = ((*gpio_control) & 1) > 0;
    printf("Pin 0 is %s\n", pin0 ? "high" : "low");
}
```

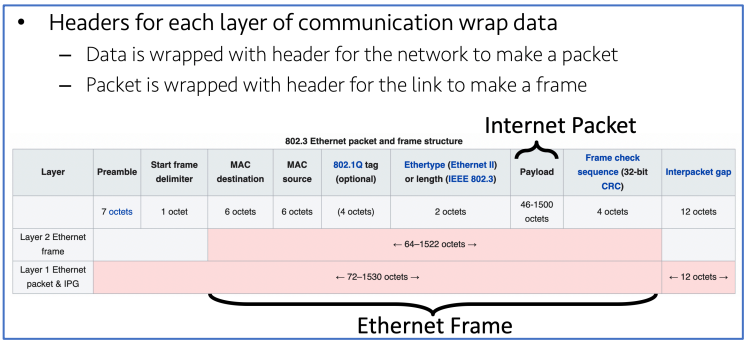
Q2b: Helper Functions [5pts]

Working with the same device as Q2a, fill in the following helper functions. You may assume all inputs to the functions are valid and can omit error-checking. Be careful that when your helper function changes one pin that it does not accidentally change others as well…

```
// Think of this file as a continuation of Q1a.
// i.e. gpio_control and gpio_level are declared correctly and are available to use.

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

       return (*gpio_control & (0x1 << pin_index)) != 0;

}

// Set the level of (only) the specified pin.
// This function assumes the target pin is already configured in output mode.
void set_level(unsigned pin_index, bool level) {
    if (level) {
        // set pin at pin_index to 1
        uint32_t mask = 0x1 << pin_index;
        *gpio_level = *gpio_level | mask;
    } else {
        // set pin at pin_index to 0
        uint32_t mask = ~(0x1 << pin_index);
        *gpio_level = *gpio_level & mask;
    }

}
```

Q3: Many Views of Memory [4pts]

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: 0x20004036 [assuming ipv4_frame.IHL < 5, otherwise 0x20004016 +
8*assuming ipv4_frame.IHL]

payload begins at: 0x20004044
```