Implementation:

Download the "`.h`" file from moodle and put it in the same folder as the "`.ino`" file. You should import it using "`#include "EthernetLab.h"`". The payload is like in the previous lab, your cool names.

The functions:

- `calculateCRC(char* payload, int payload_size)` - calculate the CRC32 of "`payload`" up to position "`payload_size`" (inclusive).
  It returns a "`uint32_t`" type.
  An example of how to use this function will be provided soon.
- `setMode(int mod)` - sets the CRC type in the "`.h`" file. Use this once in "`setup()`."
  - 1 - sets it to receiver mode. The receiver should use its CRC function in this mode.
  - 0 - sets it to sender mode.

You should calculate the CRC32 bits of your data and attach them to the data before transmission.

Here is a small example of the smart use of casting to put the CRC bits (calculated using the function in the "`.h`" file). This example makes use of our CRC32 calculation:

```
char arr2d2[5] = {0};        // Data container
setMode(1);                  // Example is Tx. Should be in setup()
arr2d2[0] = 'c';

// calculate CRC32
uint32_t crc = calculateCRC(arr2d2, strlen(arr2d2));

// Save CRC32 at array's end
*((unsigned long*)(arr2d2 + strlen(arr2d2))) = crc;

// result array is ['c'|CRC32]
```

Reading 4 bytes as "`uint32_t`" (instead of char) is done similarly.

Setting the Serial connection (`BAUD_RATE`) at 115200 bps is recommended.

1. Ethernet-Wannabe Protocol:

You will implement an Ethernet-*like* frame in this lab. Ethernet is defined in a constant size range (64 to 1522 bytes), but we will implement a shorter frame (due to rate limitations). Your implementation should be in the Layer 2 functions (from lab 3, instead of CRC/Hamming). In other words, change `layer2_tx()` and `layer2_rx()` from the previous lab.
The frame has the following fields:

1. Destination Address - 1 byte. The addresses are (0x10+Y) or (0x00+Y), where Y is your pair number.
2. Source Address - 1 byte. This must be the address not picked in 1. 0x00+4
3. Type - 1 byte set to zero[1].
4. Length - 1 byte containing Payload's length (in bytes). Y=5
5. Payload - The data to send to the other device.
6. Frame Check Sequence (FCS) - 4 bytes of CRC. It is calculated via the `calculateCRC` function. **You should calculate the CRC bits for all of the above fields.**

Hence, our frame size varies from 8 bytes to *infinity* (trivia: the longest name recorded has 747 characters). You may use buffers of size 100 bytes.
The payload in this lab consists of one byte (or two, your choice) for the S&W protocol (next section). The rest of the payload is your cool names.

The time between frames should be 12 bytes long (calculatable using $BIT\ TIME\ \ *\ \#\ of\ Bits$). This time gap is called Inter-Frame Gap (IFG). The receiver waits $0.5\ *\ IFG$ before reassembling the data.
When it does, it checks the CRC of the frame (drops it immediately if it is bad) and checks if the destination is correct. If the destination is correct, print the payload (your names only, ignore the first byte).

We recommend using a struct for the frame, like this:

```
typedef struct Frame {
      uint8_t destination_address;
      uint8_t source_address;
      uint8_t frame_type;
      uint8_t length;
      uint8_t* payload;
      uint32_t crc;
} frame;
```

Now we can copy the frame's content into an array (and vice-versa) faster:

```
uint8_t array2send[100];
for(int i = 0; i < 4; i++){
      array2send[i] = *((unit8_t *)&frame + i);
}
for(int i = 0; i < strlen(payload); i++){
      array2send[i+4] = payload[i];
}
```

[1] Place holder for VLAN tag in the original Ethernet Frame. This is an optional field in the Ethernet frame

2. Stop and Wait (S&W):

You are required to create the S&W ARQ protocol. One Arduino is the sender; the other is the receiver (the sender sends data; the receiver sends ACKs).
The sender needs to hold a timeout "timer." Once timed out, the sender resends the frame to the receiver. The Arduino has only one general-purpose timer, so time measurements should use "`millis`."
The payload in the Ethernet-*like* frame consists of a serial number field (*what is its size?*) and an identifier (so the other side would know whether it's an ACK or a data frame). You may order them as you like and use up to two bytes for these fields.
Measure the following:

1. Average Round Trip Time (RTT) - Measured by taking time stamps from frames sent up to the corresponding ACK. RTT is measured *after* sending the frame (when you start the IFG timer). The RTT will be averaged and saved for timeout calculation.
   You may use an initial RTT of $2 * IFG$ (*why?*)
2. Error probability - The receiver should have two counters; The first one counts bad frames, and the other counts the total number of frames.
   This (empirical) error probability is calculated by $\frac{Number\ Of\ Bad\ Frames}{Total\ Number\ of\ Frames}$.
   <u>Print your measured error probability</u>

General Tips:

1. **Read all general tips; they might help you with your code, questions, and defenses.**
2. Waiting for an IFG of 12 bytes can be tedious. You can use defines to adjust this time to your liking. Your final version, however, must use 12 bytes.
3. It is very recommended to write your design for the S&W next to the lectures.
4. Notice that some types have different sizes. You should check this using "`sizeof`."
5. Notice that $RTT_n = \frac{n-1}{n} \cdot RTT_{n-1} + \frac{1}{n} \cdot rtt$, where rtt is your new measured RTT, RTT_{n-1} is your old RTT, and RTT_n is the new RTT.
   (proof: manipulate $b_n = \frac{1}{n}\sum_{i=1}^{n} a_i$ to be a function of b_{n-1} and a_n)
6. Do not eat the controller (בקר); we know it's in a tasty Kosher box.
7. **Your code should be well documented!**