Lab 3 - Hamming & CRC codes

Implementation:

You may use either device-to-device communication programs from Labs 1 or 2. The data is a static string composed of your names. E.g., `char *data2send = "Marconi & Shannon".`

The following functions are Layer 2 functions and are ***not*** to be implemented as part of the Layer 1 functions. Your "`loop()`" function ought to be of the form (function signatures and order can be different):

```
loop(){
      layer2_tx(); // Calls Hamming47_tx() or CRC4_tx()
      layer2_rx(); // Calls the Rx version of the above
      layer1_tx(); // Either uart_tx() or usart_tx() from Labs 1/2
      layer1_rx(); // Rx version of the above
}
```

1. Inter-Layer Communication

The layers "communicate" via flags (in real life, via API). I.e., these functions check if the data is ready to be sent and ready to be tested. The functions from Lab 1 should work (almost) the same, except for the data generator, printer, and random waiting times (after transmission), which should be moved to the Layer 2 functions.

Functions from Lab 1 should raise "busy" flags - so Layer 2 functions won't try to spam Layer 1 transmitter with requests OR read incomplete information from the receiver.

Examples:

- The Layer 2 transmitter has to poll (not busy waiting!) on "`layer_1_tx_busy`" before raising a flag "`layer_2_tx_request`". When the "`layer_2_tx_request`" is up, the Layer 1 transmitter raises the "`layer_1_tx_busy`" flag and resets the "`layer_2_tx_request`".
  Don't forget to copy the data from Layer 2's buffer to Layer 1's buffer.
- When Layer 1's receiver starts reading data, it raises "`layer_1_rx_busy`". This flag is reset after the Layer 1 receiver gets all the bits. Layer 2 reads "`layer_1_rx_busy`"'s value and looks for a falling edge before reading the data from a shared buffer.
  After reading the word, it tests the CRC/Hamming Parity bits (according to the instructions below).

You need to implement **both** Hamming & CRC (in separate files).

2. Hamming code (7,4,3):

You are required to create the following functions (you may change function signatures):

1. `Hamming47_tx()` - *splits an 8-bit character into two 4-bit nibbles.* For each nibble, calculate the parity bits. Print the result (7 bits, in binary) after the calculation.
2. `Hamming47_rx()` - checks each nibble for errors, issuing a fix if the syndrome is non-zero. Every received word (7 bits) should be printed (in binary), and if you issued a fix, the result word should be printed too.
   The data should be reassembled into the original ASCII character (print it too).

Do the following tests and **attach screenshots of your work to the report**:

1. Change **no bits**. Add a screenshot from the transmitter (the original sent word) and receiver (what it got). Show us you have successfully reassembled the character (*up to 3 images*).
2. Change **one bit** out of the 7 bits *after printing the word to send* (in `Hamming47_tx()`). Show us the original transmitted word and the received word. Let the receiver check for errors. Show a screenshot before and after the corrective response. Show us you have successfully reassembled the character (*up to 4 images*).
3. Repeat 2, but with **two bits** changed.
4. Repeat 2, but with **three bits** changed. What is the test result? Is this desirable?

Your code should allow us to quickly change the number of corrupt bits (by using a define, for example). **We will deduce points if your code doesn't allow this.**
Note that you ought to send 14 bits to send a single letter. You cannot send the 14 bits in "one-shot"; you should send it 7 bits at a time.

3. CRC-4:

Create the following functions (you may change function signatures):

1. `CRC4_tx()` - reads a character (8-bit) from your name string and calculates CRC-4. Print the result (12 bits, in binary) after the calculation.
   NOTE: Your Layer 1 transmitter may send all 12 bits together rather than splitting the data.
2. `CRC4_rx()` - checks errors. Every received character (8 bits) should be printed (as a character), along with the CRC status (fail/success).

Use the following generator polynomial for CRC-4: $x^4 + x + 1$.

Repeat the first two tests (zero and one-bit errors) in Hamming (7,4,3) and **attach the result to your report.**

Your submission should contain both your codes and your reports! Your report must have up to 15 images of Hamming's experiments and up to 6 from the CRC experiments.

General Tips:

1. **Read all general tips; they might help you with your code, questions, and defenses.**
2. You can print numbers in binary using "`Serial.print(<number>, BIN);`". You may use "`sprintf`" to create the strings to print. However, note that C has no support "out of the box" to print numbers in binary.
3. We have discussed all operations over binary numbers. Saving all polynomials and data as numbers allows faster CRC calculations (using XOR). E.g., "`uint16_t a=0x13, b=0x7; c=a^b;`" assigns `c=0x14` (the XOR result of `0b10011` and `0b00111`). Hence, the CRC calculations can be implemented using a single for loop.
4. Instead of padding the numbers with zeros, it is possible to mirror the polynomial and data, calculate CRC for N iterations (data length), and shift right the mirrored result four times.
5. **Your code should be well documented!**

Tools That May Help:

Hamming code simulator:
http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/Hamming/HammingCodes.html

Hamming(7,4) Code in Wikipedia:
https://en.wikipedia.org/wiki/Hamming(7,4)

CRC in Wikipedia:
https://en.wikipedia.org/wiki/Cyclic_redundancy_check

CRC Online Calculator:
asecuritysite.com/comms/crc_div