Programming Assignment #4 - HW #8

COMS W4119 - Computer Networks — Due April 19, 2006
Spring 2006 — Prof. Rubenstein

Programming Assignments must be e-mailed to the TA. The e-mail should contain a single compressed file containing all files to be turned in. If desired, answers to questions can be turned in at the beginning of class on the day that the homework is due. CVN students have one additional day. Late assignments will not be accepted.

In this programming assignment, you will build bit error detecting and correcting codes. You shall implement two versions: one that implements a 2-D parity check, and one that implements a (7,4) Linear Hamming Code. The remaining text will refer to these respectively as `2D` and `LH`.

You should produce two executables, one called `encoder` and one called `decoder`. Both encoder and decoder will take 2 parameters:

- the first parameter will be the code type (0 for `LH`, 1 for `2D`).
- For the encoder, the second parameter is the data. Data will always be input as a sequence of 0's and 1's.
- For the decoder, the second parameter is the received codeword, input as a sequence of 0's and 1's.

Your output

Both your encoder and decoder should output a sequence of 0's and 1's. The encoder should output the valid codeword. The decoder's output depends on whether the codeword was (believed to have been) corrected.

- If the codeword was corrected, the decoder should output the correct data, then a colon, then the checkbits. For example, if the codeword is 1001101 where the first four bits are data, then 1001:101 should be returned
- If the codeword was not corrected, but errors are known to exist, the decoder should output `UNCORRECTED`.

Testing/Verification

You should be able to test your code on your own by taking the output of your encoder, changing some bits, and feeding that into your decoder.

When using `LH`, we will always use a 4-bit data sequence. When using `2D`, we will use a k^2-bit sequence for $2 \leq k \leq 10$. Your code should properly handle this entire range.

What to turn in

The assignment should be turned in electronically, by e-mail to 4119-submit@cs.columbia.edu. You must turn in **as a single attachment** all work wrapped up together as a single gzipped tarfile with filename `pa2-CUID.tgz` (where `CUID` is your Columbia UNI). The tarfile should decompress to create a directory `pa2-CUID/` with the following files (example given in C):

- `encoder.h`: header information for the encoder
- `encoder.c`: encoder main code
- `decoder.h`: header information for the decoder
- `decoder.c`: decoder main code
- If you have additional c code that you wrote that is included by these files, include that code as well.

- `encoder`: your encoder executable
- `decoder`: your decoder executable
- `Makefile`: your makefile or `make.txt` that explains the procedures you followed to compile your file.
- A file named `test.txt` that includes some tests on which you ran your code.
- **IMPORTANT:** A file, `describe.txt` that explains the specifics of how your encoder does its encoding. In particular, if your encoder outputs $b_1 b_2 \cdots ... b_n$, in this file, you should describe what each bit b_i is: whether it is data or checkbit, and if it is a checkbit, how it is formed (what data bits are XOR'd together to create it.)

Additional Comments

We recommend having the encoder place all data bits, in order, before the checkbits. For grading, we will flip specific sets of bits and observe the outcome.

Grading

When your decoder receives as input a modified bitstream, it should correctly repair any 1-bit errors, and correctly detect any 2-bit errors. Your decoder should **never** return an invalid codeword, i.e., if it incorrectly repairs a bit-sequence with more than 2 errors, the incorrect codeword returned should always be valid.

For 2-bit errors, `2D` should always return `UNCORRECTED`, while `LH` should return the wrong valid codeword (i.e., the one that was Hamming Distance 1 from the error).

Your grade will be based on *our* execution of your code against some pre-determined test cases (that may be slightly different) from the tests we provide above. So make sure that your code works, or we will not be able to test. We will also randomly spot-check the source code turned in to verify that it matches the object code, and will also compare object code across student assignments.

Again, *the code must execute on Columbia CS Linux boxes to receive credit.*