# Lab 3 –Bitwise Ops – Error Control Coding

**Objectives**

In this lab, each student is to write a prompt-driven program called **prog3.c** that performs bitwise operations on data in order to implement an error correcting code.

The student should gain an understanding of:

* **Bitwise Operators**
* **Formatted input with keyphrases and formatted output in hex and binary**
* **Storage types**

**Background**

With higher and higher bit rates and more and more interference (noise), error control coding is necessary in today’s communication. Error *detection* coding is used to detect whether or not bits are (probably) in error, and error *correction* coding is used to correct bits which are assumed to be in error.

**Input**

Your program should prompt the user to enter either *two* printable **ASCII** characters or an exit *keyphrase* of your choice.

**Operation**

Upon execution, the program should tell the user what it does and the syntax of the commands to be entered. The program should loop continuously showing a prompt until the exit keyphrase command is given. The exit keyphrase can be of your choosing, but should be logical and practical.

When the **ASCII** characters are entered, your program should construct one or two binary code words from the characters. The codeword will be built using a (21, 16) Hamming code. That is, this code takes 16 bits (two bytes) of information and adds in five more *parity* bits to give a 21-bit codeword. When transmitted, this code word will allow a user to determine if the information transmitted arrived correctly—or not—as long as no more than one bit gets corrupted. Thus it is called a single bit-error correcting code.

The parity bits will be in bit positions 1, 2, 4, 8, and 16 of the code word, given that these bits make the following sums of bits equal to zero (mod 2):

**Bit 1:** 1 3 5 7 9 11 13 15 17 19 21

**Bit 2:** 2 3 6 7 10 11 14 15 18 19

**Bit 4:** 4 5 6 7 12 13 14 15 20 21

**Bit 8:** 8 9 10 11 12 13 14 15

**Bit 16:** 16 17 18 19 20 21

That is, **Bit 1** will be set to **0** if bits 3, 5, 7, 9, 11, 13, 15, 17, 19 and 21 sum to an even number, and will be set to a **1** if they sum to odd number. (*Note: There is a reason that I have numbered the bits 1 – 21 and not 0 – 20. The reason will be evident later.*)

The non-parity (*information*) bits will simply be the values they were in the original 16 bits, only shifted to their new position.

For example, if the user inputs the two characters, a **C** first and **U** second, you will first construct the following 32 bit integer as follows. Since **C** = **0x43** and **U** = **0x55**, your information word (integer) should be **0x00005543** or

**Bit 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17**

**0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**Bit 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01**

**0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 1**

in binary where the first character entered is the lowest significant byte and the upper two bytes are always zero.

You must then take these lower 16 bits (the information word) and shift them into a new integer (the code word) at their proper positions. Since bits 1 and 2 of the code word are parity bits, bit 1 of the information word will be bit 3 of the code word. Bit 4 of the code word is a parity bit, so bits 2 – 4 of the information word will be bits 5 – 7 of the codeword, and so on as shown below:



Once the information bits have been copied to their appropriate location in the code word, you should calculate the value of the parity bits and set them accordingly.

**Output**

Your program should echo the two characters entered, and then print the information word in both hexadecimal and binary with the most significant bits and bytes shown first. It should then print the value of each of the five parity bits. Lastly, it should print the hex and binary values of the new code word. (*Note: this output should help you debug and validate your program.*)

For example, if the user were to input ‘**CU**’ as mentioned above, your program should print something like the following:

**U C**

**0x 00 00 55 43**

**-------- -------- 01010101 01000011**

Then the partially-filled code word would be:

**-------- ---01010 P1010100 P001P1PP**

where the **P**s are parity bits. The value of each parity bit would then be whatever it takes to make the corresponding sums zero. That is:

**CW: P P 1 P 1 0 0 P 0 0 1 0 1 0 1 P 0 1 0 1 0**

**P1: 1 3 5 7 9 11 13 15 17 19 21**

**P 1 1 0 0 1 1 1 0 0 0 = P+5**

So **P1 = 1** so that **P1+5 = 1+5 = 6 = 0 (mod 2).**

Now for **P2,**

**CW: P P 1 P 1 0 0 P 0 0 1 0 1 0 1 P 0 1 0 1 0**

**P2: 2 3 6 7 10 11 14 15 18 19**

**P 1 0 0 0 1 0 1 1 0 = P+4**

So **P2 = 0** so that **P2+4 = 0+4 = 4 = 0 (mod 2).**

Now for **P4,**

**CW: P P 1 P 1 0 0 P 0 0 1 0 1 0 1 P 0 1 0 1 08**

**P4: 4 5 6 7 12 13 14 15 20 21**

**P 1 0 0 0 1 0 1 1 0 = P+4**

So **P4 = 0.**

For **P8,**

**CW: P P 1 P 1 0 0 P 0 0 1 0 1 0 1 P 0 1 0 1 0**

**P8: 8 9 10 11 12 13 14 15**

**P 0 0 1 0 1 0 1 = P+3**

So **P8 = 1.**

And for **P16,**

**CW: P P 1 P 1 0 0 P 0 0 1 0 1 0 1 P 0 1 0 1 0**

**P16: 16 17 18 19 20 21**

**P 0 1 0 1 0 = P+2**

Then **P16 = 0.**

So the five parity bits **P1**, **P2**, **P4**, **P8**, and **P16** are: **1**, **0**, **0**, **1**, and **0**.

Therefore, the final code word with the parity pits inserted would be:

**-------- ---01010 01010100 10010101**

or in hex:

**0x 0 0 0 A 5 4 9 5 = 0x000A5495**

**Further Considerations**

Your program must not store individual bits, an array of bits (or a string of ‘1’s and ‘0’s), or even an array three bytes. Your program should use a single 32-bit integer to store the 21 code bits and should employ bitwise operators to manipulate these bits.

Your program should be structured neatly, easily readable, and well commented. Furthermore, variable and function names should be such that the software is as “self-commenting” as possible.

**Creation and Submission**

***Each individual student must complete their own program. Copying other students’ code will be tested for and will not be tolerated.***

Use the following line to compile your program

**gcc -Wall -g prog3.c -o prog3**

The code you submit must compile using the **–Wall** flag and no compiler errors or warnings should be printed. To receive credit for this assignment your code must compile and at a minimum perform some required function.

Code that does not compile or crashes before performing some required function will not be accepted or graded. **All students must do a final check on one of the CES Ubuntu machines to verify that gcc using Ubuntu shows no warning messages before submitting your project.**

Submit your program on Blackboard before midnight on **Monday, February 18th**.