ECE 222: System Programming Concepts **Fall 2016**
Machine Problem 2 **Due: 11:59 pm, Thursday, September 8**

The goal of this machine problem is to learn to program with bitwise operations. In this lab, you write functions that perform bitwise operations on data in order to implement an error correcting code. With faster bit rates and greater interference, error control coding is becoming increasingly important in communications. Error detection coding can be used to detect whether or not bits are (probably) in error, and error correction coding can be implemented to correct bits which are (probably) in error.

You are provided with a template program that prompts the user to enter either a command to encode three ASCII characters, decode up to 8 hexadecimal digits, or exit. Using the template, add functions to encode or decode the input. A template is provided and you must use the organization provided.

Encoding

Three characters are entered as input, and your program constructs a binary code word from the three characters. The codeword will be built using a (29, 24) Hamming code. That is, this code takes 24 bits of information and adds five additional parity bits to give a 29-bit code word. If we include our encoding algorithm in a communications system, this code word allows the system to determine if the information that is transmitted is received correctly 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. A parity bit is set such that the sum of the bits selected for a parity equation is equal to zero (mod 2): The bits in the codeword that are included in a particular parity bit as shown in the table to the right.

	Parity				
	Bit 16	**Bit 8**	**Bit 4**	**Bit 2**	**Bit 1**
1					1
2				2	
3				3	3
4			4		
5			5		5
6			6	6	
7			7	7	7
8		8			
9		9			9
10		10		10	
11		11		11	11
12		12	12		
13		13	13		13
14		14	14	14	
15		15	15	15	15
16	16				
17	17				17
18	18			18	
19	19			19	19
20	20		20		
21	21		21		21
22	22		22	22	
23	23		23	23	23
24	24	24			
25	25	25			25
26	26	26		26	
27	27	27		27	27
28	28	28	28		
29	29	29	29		29

That is, `Bit 1` will be set to `0` if bits `3`, `5`, `7`, `9`, `11`, `13`, `15`, `17`, `19`, `21`, `23`, `25`, `27`, and `29` sum to an even number, and `Bit 1` will be set to a `1` if they sum to odd number. The non-parity (i.e., information) bits will simply be the values they were in the original 24 bits, only shifted to their new position.

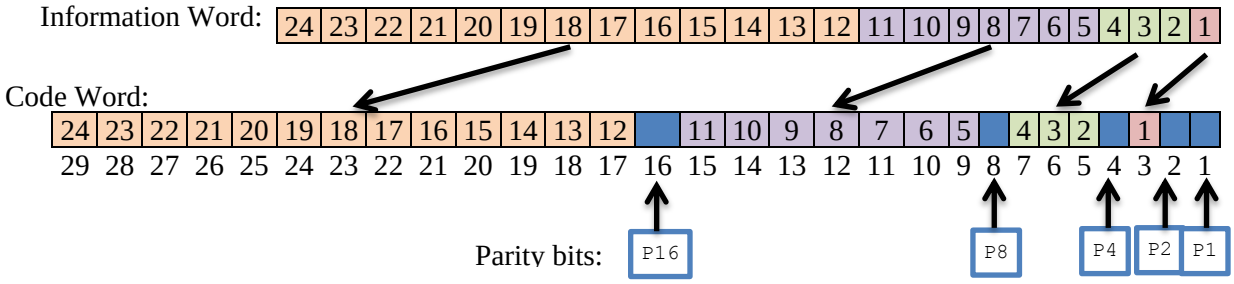
To generate the information bits, the user inputs three characters. For example, if the input is a **`C`** as the first character, **`U`** as the second, and **`t`** as the third, you will construct the following 32 bit integer as follows. Since **`C=0x43`**, **`U=0x55`**, and **`t=0x74`** your information word (integer) should be **`0x00745543`** 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  1  1  1  0  1  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 byte is always zero.

You must then take these lower 24 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 is bit `3` of the code word. `Bit 4` of the code word is a parity bit, so bits `2 – 4` of the information word are 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 calculate the value of the parity bits and set them accordingly. Continuing the same example, if the user input is '**`CUt`**', the template program has code to print the following two lines. You must write the code to print the third line, which prints the information word as binary digits. Note that bit 24 is the first digit that is printed and bit 1 is the last bit.

```
                     t        U        C
0x     00           74       55       43
-------- 01110100 01010101 01000011
```

Next you calculate the code word. The first step is to shift the information bits into their final locations. The partially-filled code word is:

```
---01110 10001010 P1010100 P001P1PP
```

where the **`P`**s are parity bits. The value of each parity bit is calculated so that the corresponding parity equation sums to zero. For example, here is the calculation for parity bit 1. Using the table on the previous page, make the sum of all the odd bits equal to zero (mod 2).

```
          22222 22222111 1111111
          98765 43210987 65432109 87654321
---01110 10001010 P1010100 P001P1PP

          0+1+0 +0+0+0+0 +1+1+1+0 +0+1+1+P = 6+P1
```

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

Now for parity bit 2, **P2,**

```
         22222 22222111 1111111
         98765 43210987 65432109 87654321
      ---01110 10001010 P1010100 P001P1PP
```

Count the number of red bits that are equal to one. There are 6, so parity bit 2 is zero. That is, **P2 = 0**.

Now for **P4,**

```
         22222 22222111 1111111
         98765 43210987 65432109 87654321
      ---01110 10001010 P1010100 P001P1PP
```

For parity bit 4, we see that among the red bits, 5 of them are ones. So **P4 = 1** to make the parity even.

Now for **P8,**

```
         22222 22222111 1111111
         98765 43210987 65432109 87654321
      ---01110 10001010 P1010100 P001P1PP
```

So **P8 = 1.**

And for **P16,**

```
         22222 22222111 1111111
         98765 43210987 65432109 87654321
      ---01110 10001010 P1010100 P001P1PP
```

Then **P16 = 0.**

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

The final code word with the parity pits inserted is:

```
      ---01110 10001010 01010100 10011100
in hex: 0x 0   E    8   A    5   4    9   C = 0x0E8A549C
```

The template program has code to print the ASCII letters and the information word as hexadecimal digits. You must add code to print the information word in binary with the most significant bits and bytes shown first. Next you must print the value of each of the five parity bits. Lastly, you must print the binary values of the code word. The program prints the hex values of the code word.

Decoding

The decoding operation inverts the above steps. The decoding function accepts the 29-bit code word that is input by the keyboard. You write code to re-calculate the value of the parity bits. Compare your calculation of your expected parity bit value to the value in the code word, and if the values do not match then set an error bit. Continuing the example above, if bit 29 of the code word is incorrect (i.e., 1 instead of 0), then the received code word is **0x1E8A549C**. When you re-calculate the parity bits you find that

the expected value for **P1** is **1**, but bit 1 in the code word (which holds the first parity bit) is **0**. Since the parity bit in the code word does not match the expected value, set **E1 = 1**. Note **P2** has the expected value (since the equation for parity bit 2 does not include the bit in position 29). Thus, **E2** equals **0**. Likewise, you find that **E4**, **E8** and **E16** are **1** because the expected value of these parity bits does not match the bit found in the code word (or equivalently, bit number 29 is included in all of these parity equations so the sum is odd instead of even). Now form the bit error location by forming the binary number using the error location bits:

E16 E8 E4 E2 E1

For this example we have the binary number **11101** or decimal **29**. This confirms that the error was in bit location 29, and you correct the value of bit 29. The template program prints the code word in hex. You add code to find the error location bits. Print the five error location bits and convert them to an integer. The template program has code to print the integer error location. You correct the code word, if necessary, and then print the information word in binary. Code is provided that will print the information word in hex and display the information as characters if possible.

Note that if the bit error location is greater than 29, then there was a decoding failure, and the code word cannot be corrected. This can only happen if there is more than one bit in error. Our program assumes that no more than one bit error occurs.

Notes

1. This program can be written many different ways, including using string functions to create an array of bits instead of an integer of bits. Your program, however, **must** use bit manipulations (bitwise operations; no string functions or arrays allowed!) on the given integers to receive credit. In addition, you cannot use arrays to store the value of the bits. Code that does not use bitwise operations will not be accepted.

2. You compile your code using:

```
gcc -Wall -g lab2.c -o lab2
```

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. A few (and incomplete) set of tests are given in the `testinput` file, and the results in the `correctoutput` file. To run these checks do

```
./lab2 < testinput > myoutput
diff myoutput correctoutput
```

The files `myoutput` and `correctoutput` must be identical. Code that fails to produce this output will not be accepted. An alternative tool to `diff` is `meld`, and it provides a graphical display of the differences between two files. To use it do

```
meld myoutput correctoutput
```

3. Submit your file `lab2.c` to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE222-1,#2. The 222-1 identifies you as a member of Section 1 (the only section this semester). The #2 identifies this as the second assignment. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes. If you don't get a confirmation email, your submission was not successful. You must include your file as an attachment. You can make more than one submission but we will only grade the final submission.

4. Turn in a paper copy of your code file at the start of the first class meeting following your submission. Do not print your input or output files.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments. Work must be completed by each individual student. See the course syllabus for additional policies.