COMP 222
C Project: Hamming Code Calculator (second part)

In **Part 1** of this project, you wrote and tested some basic C functions to allow you to manipulate and count bits with respect to an arbitrary bit pattern. In this part of the project, you will complete the functions to calculate Hamming codes for 8 bit data and 4 bit checks. There are 3 remaining sets of functions to implement:

- Functions for check bit calculations
- Functions for code word building and extraction
- Functions for error detection

Having the basic functions from the previous work will simplify and streamline the implementation of the new functions.

Don't begin this part of the project until the functions from the first part are completed and pass the test cases. Use the same strategy for each of the three parts below.

Program Development Strategy: Using Stub Functions During Development

A stub is a skeleton version of a function that provides the same info as the prototype – return type, function name, and argument list – and a minimal function body that results in source code that compiles and links with no errors. Note that there is no requirement that it execute, but the "compiles and links" requirement is important. A set of stub methods may not seem useful at first glance, but using stubs greatly enhances the program developer's ability to incrementally code and complete a potentially large source file. In a team environment with configuration management of source code, it is a common requirement that all checked in code must compile.

Given a function prototype, what must be done to convert it to a compilable stub?

- The argument list in a stub contains datatypes of each argument. To convert to a stub, argument names must be added for each argument.
- The semicolon that terminates the prototype must be replaced by an empty function body – { }
    - If the return type of the stub is void, the function body can be empty, and stub is complete.
    - If the return type of the stub is not void, then the function body must contain a return statement that returns a value that matches the return type.

Stubs allow the program developer to create a skeleton version of the application source code that compiles even though it is largely incomplete. The developer can then select which stubs to work on while leaving other stub implementations for later.

In the examples for this class, you are being provided the prototypes for the functions needed for the application being developed. In other words, the instructor has already completed the requirements and design stages of the software development process, you only have to complete the implementation and testing stages. In a more advanced class and in the workplace, you may be called on to do application design, not just code development. In that case you will have to define the prototypes before implementing them. This is a more advanced skill, you will get more practice at it in later courses.

Example

```
double sum(double a, double b, double c) {            // Definition
      double result = a + b + c;
      return result;
}

double sum(double, double, double);                   // Prototype

double sum(double a, double b, double c) {            // Stub
      return 0.0;
}
```

The definition and prototype are valid elements of a completed application. We've discussed how and where to position them in a collection of source and header files. A stub is a temporary version or place holder for the to-be-completed-later definition but which in the meantime allows the source code to compile. Stubs won't be seen in a completed application since they are by definition an incomplete skeleton version of a complete definition.

Functions for Part 2
A) Check Bit Calculation Functions

```
unsigned calculateCheckValue(unsigned dataword, int index);
unsigned calculateCheckWord(unsigned dataword);
```

The first function calculates a single check bit based on the inputs of 8 bit data, and 1-based check index. The function call

```
unsigned c = calculateCheckValue(0xFF,3);
```

calculates the 3rd check bit (code word position 4) for the 8 bit data pattern in the first input. The return value c will be either 0x0 or 0x1. The second function repeatedly calls the first function to calculate all 4 check bits and return them as a 4 bit hex value. The function call

```
unsigned d = calculateCheckWord(0xFF);
```

internally calls **`calculateCheckValue()`** for all 4 bits and assembles the results into a single 4-bit hex value and returns it. Note that these functions keep data and check bits separate, they are not combined into a 12 bit code word. The next set of functions below take care of that.

Testing for A)

Use the following function to test:

```
void testcheckfunctions() {
      int datawordscount = 4;
      unsigned datawords[] = {0xFF, 0xA3, 0x00, 0x54};

      printf("test individual check values----------------------\n");
      for (int i=0; i<datawordscount; i++) {
            unsigned data = datawords[i];
            printf("\t data = %x; checkbits = ",data);
            for (int j=1; j<=CHECKBITCOUNT; j++) {
                  unsigned c = calculateCheckValue(data,j);
                  printf("%x ",c);
            }
            unsigned check = calculateCheckWord(data);
            printf("complete check word = %x\n",check);
      }
}
```

Output

```
test individual check values------------------------------
          data = ff; checkbits = 1 1 0 0 complete check word = 3
          data = a3; checkbits = 0 0 0 0 complete check word = 0
          data = 0; checkbits = 0 0 0 0 complete check word = 0
          data = 54; checkbits = 0 0 1 0 complete check word = 4
```

Note that the individual check bits are printed in order from bit 1 to bit 4, so they're reversed relative to the way we number the bits.

B) Code Word Calculation Functions

`unsigned createCodeWord(unsigned, unsigned);`

This function takes an 8-bit data word and the already computed 4-bit check word and combines it into a 12-bit code word with the bits in their proper positions.

`unsigned extractDataWordFromCodeWord(unsigned);`

This function takes a 12-bit code word and extracts the 8-bit data word and returns it, it doesn't do any check bit calculations, comparisons, or error correction.

`unsigned extractCheckWordFromCodeWord(unsigned);`

This function takes a 12-bit code word and extracts the 4-bit code word and returns it.

Testing for B) TBD

Code

```
void testcodewords() {
        int datawordscount = 4;
        unsigned datawords[] = {0xFF, 0xA3, 0x00, 0x54};

        printf("test code words------------------------------\n");
        for (int i=0; i<datawordscount; i++) {
                unsigned data = datawords[i];
                unsigned check = calculateCheckWord(data);
                unsigned code = createCodeWord(data,check);
                unsigned data2 = extractDataWordFromCodeWord(code);
                unsigned check2 = extractCheckWordFromCodeWord(code);
                printf("data %x, chk %x, code %x, extdata %x, extchk %x\n",
                        data,check,code,data2,check2);
        }
}
```

Output

```
test code words------------------------------
data ff, chk 3, code f77, extdata ff, extchk 3
data a3, chk 0, code a14, extdata a3, extchk 0
data 0, chk 0, code 0, extdata 0, extchk 0
data 54, chk 4, code 528, extdata 54, extchk 4
```

C) Error Correction Functions

```
unsigned calculateCheckWordXor(unsigned codeword);
```

This function takes a 12-bit code word and computes the 4-bit xor of the original check bits and the recalculated check bits and returns it.

```
unsigned extractCorrectedDataWord(unsigned codeword);
```

This function is the final function, it takes a 12-bit code word, extracts the 8-bit data word, corrects a single bit if necessary, and returns the corrected 8-bit data word.

Testing for C) TBD

Code

```
void testcorrecterror() {
	int datawordscount = 4;
	unsigned datawords[] = {0xEF, 0x11, 0x56, 0x9A};
	int databiterror[] = {1,5,6,2};

	printf("test code words------------------------------\n");
	for (int i=0; i<datawordscount; i++) {
		unsigned data = datawords[i];
		unsigned check = calculateCheckWord(data);
		unsigned code = createCodeWord(data,check);
		int errorpos = convertDataIndexToCodeWordIndex(databiterror[i]);
		unsigned codeerr = code;
		setBitValueAtIndex(&codeerr,errorpos,
			(getBitValueAtIndex(codeerr,errorpos)==0x0)?0x1:0x0);
		unsigned dataerr = extractDataWordFromCodeWord(codeerr);
		unsigned datacorrected = extractCorrectedDataWord(codeerr);
		printf("data=%x, code=%x, error code=%x, error data=%x, corr data=%x\n",
			data,code,codeerr,dataerr,datacorrected);
	}
}
```

Output

```
	test code words------------------------------
	data = ef, code = ef6, error code = ef2, error data = ee, corrected data = ef
	data = 11, code = 186, error code = 86, error data = 1, corrected data = 11
	data = 56, code = 531, error code = 731, error data = 76, corrected data = 56
	data = 9a, code = 95b, error code = 94b, error data = 98, corrected data = 9a
```