C Programming SEC-DED Data Encoding

For this assignment, you will implement a collection of C functions to support a classic data encoding scheme.

Data transmission and data storage both raise the risk that one or more bits will be corrupted (received or stored incorrectly). Various schemes have been devised for detecting, and even correcting, such errors. It should be noted that any such scheme can be overcome if enough bits are corrupted.

Richard Hamming devised an ingenious scheme that provides simultaneously for the correction of corruption involving a single bit (SEC), and detection, but not correction, of corruption involving two bits (DED). The system will, however, fail if more than two bits are corrupted.

Given a sequence of data bits, we will compute a corresponding set of parity-check bits (or simply parity bits). The general idea is that the parity bits are intermingled with the data bits as follows:

- Parity bits are stored at index 0, and at indices that are powers of 2 (1, 2, 4, etc.).
- Data bits are stored (in their original order) at the indices not occupied by parity bits.

Each parity bit indicates whether a particular set of data bits contains an even or an odd number of 1's; a parity bit of 0 indicates an even number of 1's and a parity bit of 1 indicates an odd number of 1's (in the corresponding set of data bits).

The sets for the parity bits are defined as follows (think of the bit positions being represented in binary):

- The parity bit at index 0 corresponds to all the other bits, including parity bits.
- The parity bit at index 1 corresponds to the bits whose positions have their 1's bit set.
- The parity bit at index 2 corresponds to the bits whose positions have their 2's bit set.
  . . .
- The parity bit at index 2^k corresponds to the bits whose positions have their 2^k's bit set.

Put differently:

The parity bit at index 1 corresponds to bits at the positions: 3 5 7 9 11 13 15 17 19 21 . . .
The parity bit at index 2 corresponds to bits at the positions: 3 6 7 10 11 14 15 18 19 22 . . .
The parity bit at index 4 corresponds to bits at the positions: 5 6 7 12 13 14 15 20 21 22 . . .

Again, consider the bit positions represented in binary, and you will find a simple way to use bitwise operations to determine if a data bit at a given position corresponds to a parity bit at a given position.

As far as error detection goes, the ingenuity of Hamming's scheme lies in several facts (we ignore the "master" parity bit, and assume no more than one bit is corrupted, in the following statements):

- Each data bit affects at least two parity bits, and no parity bit affects another parity bit. Therefore, if a single bit is corrupted, we can tell whether it is a parity bit or a data bit. If one parity bit disagrees with its data bits, and all the other parity bits agree with their data bits, then the error lies in that single parity bit.
- If two or more parity bits disagree with their data bits (and no more than one bit is in error), there will be a single data bit that affects all of the disagreeable parity bits, and that must be the erroneous bit.
- Even better, if we know which parity bits are disagreeable, the sum of their indices equals the index of the erroneous data bit. We will call the sum of the disagreeable parity bits the *syndrome*.
- This is a *distance-3* coding scheme; in other words, if we start with a valid set of data and parity bits (i.e., one in which all the bits are correct), at least three bits must flip in order to yield a different, valid set of data and parity bits.

However, the facts above are not enough to provide a satisfactory system; it supports detection of single-bit errors (and even double-bit errors), but it cannot distinguish a single-bit error from a double-bit error.

Therefore, it leaves us with two alternatives: use the scheme to detect errors (in up to two bits) but not to attempt any corrections, or assume that no more than a single bit will ever be wrong, and use the scheme to correct single-bit errors. The first alternative seems weak, the second is dangerous. In other words, this scheme gives us DED, but not reliable SEC.

The next step is to add the aforementioned master parity bit, which measures the parity of all the other bits. This is sometimes referred to as an *extended Hamming code*. The additional parity bit provides a surprising benefit: we can now distinguish between single-bit and double-bit errors. The underlying theory is beyond this specification, but the following facts hold:

Possible Scenarios			Conclusion
Errors	Overall Parity	Syndrome	
0	even	0	no error
1	odd	0 not 0	master parity bit is erroneous syndrome indicates erroneous bit
2	even	not 0	double-bit error (not correctable)

For the following exercise, we will assume that bit corruption is sufficiently unlikely that we may disregard the possibility of more than two bits being corrupted.

In principle, Hamming's scheme can be used to encode an arbitrarily long sequence of data bits. Pragmatically, we need to fit things into an integer number of bytes. For this assignment, you will implement the necessary functions for encoding a sequence of 64 data bits, which will require 8 parity bits, for a total of 72 bits. This is called a *Hamming (72, 64) code*; the convention is that the first value is the total length of the encoded data and the second is the number of data bits.

Since this does not fit exactly into any of the standard C types, we will employ the following **`struct`** type:

```
#define NBYTES 9                // number of bytes (data + parity)

struct _Hamming72_64 {

   uint8_t bits[NBYTES];        // 72-bit data/parity chunk
};
typedef struct _Hamming72_64 Hamming72_64;
```

You might wonder why we wrap the array in a **`struct`** variable with no other members. The advantages are that we now have a specific data type for representing Hamming encodings, and the convenience that we can assign **`struct`** variables and use them as return values (neither of which is possible with a naked array in C).

The first function we need is one that can take a sequence of data bits, compute the corresponding parity bits, and assemble a proper object to represent the encoded data:

```
/**  Create the Hamming encoding for a sequence of 64 data bits.
 *
 *   Pre:  *pH is a Hamming72_64 object
 *         *pBits is an array of 64 bits
 *   Post: pH->bits has been initialized to store the given data bits and
 *            corresponding parity bits, as described in the specification
 *            of the Hamming (72, 64) scheme.
 */
void set_Hamming72_64(Hamming72_64* const pH, const uint8_t* const pBits);
```

We will test this function by passing it various data bit arrays and then examining the contents of `pH->bits` (both the data and parity bits). This function should be relatively easy for you to test, because it's easy to vary the data bits that correspond to each parity bit, and make sure that the parity bits vary correctly. That said, it does help if you can view the bits in a human-friendly manner.

The second function we need is one that can take a `Hamming72_64` object, determine if any errors can be detected, whether any such errors can be corrected, and correct them, if possible. We will also need a way for the function to report its findings in detail to the caller; so we will employ the following **`enum`** and **`struct`** types:

```
enum _Error {MASTERPARITY, SINGLEPARITY, SINGLEDATA, MULTIBIT, NOERROR};
typedef enum _Error Error;

struct _Report_Hamming72_64 {
   Error   status;
   uint8_t syndrome;
};
typedef struct _Report_Hamming72_64 Report_Hamming72_64;
```

The analysis and correction function will have the following specification:

```
/**  Checks the validity of the given Hamming encoding, and corrects errors
 *   when possible.
 *
 *   Pre:  *pH is initialized
 *   Post: If correctable errors are found, they are corrected in pH->bits.
 *   Returns:  A Report_Hamming72_64 object holding the computed syndrome
 *             and flag indicating what type of error, if any, was found.
 */
Report_Hamming72_64 correct_Hamming72_64(Hamming72_64* const pH);
```

A C header file containing these declarations is supplied on the course website; do not modify the contents of that file for any reason, since your submission will be compiled using the posted version of the header file.

You should consider making your solution modular by writing helper functions; my solution uses three for computing/checking parity bits, in addition to the print function mentioned below.

Testing

A tar file is posted on the course website with the following contents:

`driver.c`	calls test functions from `HammingGrader.o`
`Hamming72_64.c`	shell file for implementing `Hamming72_64` type
`Hamming72_64.h`	public declarations for `Hamming72_64` type
`HammingGrader.h`	public declarations of test functions; pay attention to this one!
`HammingGrader.o`	implementations of test functions

You should edit the file `Hamming72_64.c` and complete the two specified functions there (as well as any static helper functions or include directives you want to add to that file). You may edit the file `driver.c` if you want more control over the testing, but be sure to test your final solution with the original version of `driver.c` since that is how we will test your solution.

You must not change either of the posted header files (`Hamming72_64.h` and `HammingGrader.h`).

You can compile the code using the command

```
gcc -o driver -std=c99 -Wall -ggdb3 driver.c Hamming72_64.c HammingGrader.o
```

and execute it using the command

```
driver [-rand]
```

The switch `-rand` is optional (but must be used the first time you run the code). Using the switch causes the driver to randomize the test data; omitting it causes the test driver to reuse the previous test data.

The driver writes an output file, `HammingLog.txt`, which contains results and score information.

Some sample files will be posted showing the correct parity bits for selected data bit sets. You may use those samples as the basis for your analysis; however, they are not guaranteed to be comprehensive.

You will find that implementation and testing are both much easier if you write some specialized helper functions. My implementation of the `Hamming72_64` type contains 8 `static` functions, including:

```
static uint8_t getBit(const uint8_t* const pB, uint8_t pos);
static void    flipBit(uint8_t* const pB, uint8_t pos);
static bool    isPowerOf2(uint8_t N);
static void    printBits(FILE* fp, const Hamming72_64* const pH);
```

There are no suitable functions in the C math library, and you should not call any from your solution. Be aware that we will not compile your solution with the `-lm` switch.

Suggested resources

From the CS 2505 course website at:

http://courses.cs.vt.edu/~cs2505/summer2015/

you should consider the following notes:

C Bitwise Operations	http://courses.cs.vt.edu/cs2505/summer2015/Notes/ T13_CBitwiseOperators.pdf
Intro to Pointers	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T14_IntroPointers.pdf
C Pointer Finale	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T17_CPointerFinale.pdf
C struct Types	http://courses.cs.vt.edu/cs2505/summer2015/Notes/T24_CstructTypes.pdf

Some of the other notes on the basics of C and separate compilation may also be useful.

What to Submit

You will submit a single C implementation file, `Hamming72_64.c`, containing your implementations of the two specified functions, and any **`static`** helper functions your solution needs, and nothing else. Note that any changes you might have made to the posted header file, `Hamming72_64.h`, will be lost since we will use the original version in our testing.

Your submission will be compiled with the testing code using the command:

```
gcc -std=c99 -ggdb3 -Wall ...
```

Test your solution thoroughly before submitting it.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//    On my honor:
//
//    - I have not discussed the C language code in my program with
//      anyone other than my instructor or the teaching assistants
//      assigned to this course.
//
//    - I have not used C language code obtained from another student,
//      or any other unauthorized source, either modified or unmodified.
//
//    - If any C language code or documentation used in my program
//      was obtained from another source, such as a text book or course
//      notes, that has been clearly noted with a proper citation in
//      the comments of my program.
//
//    <Student Name>
```

Change Log

Version	Posted	Pg	Change
2.00	Jan 19		Base document.
2.01	Apr 17	3	Added some explicit instructions for editing/compiling the posted code files.