

## Problem Set 7

Due before midnight on Tuesday, April 3, 2007.

### 1 Forward Error Correction

Information is subject to random bit errors in storage and transmission. Forward error correction (FEC) adds redundant data to a message to allow for the detection and correction of limited numbers of such errors. Here's how it works: An *FEC encoding* function  $E$  maps a  $k$ -bit *data block*  $x$  to an  $n$ -bit *code word*  $y = E(x)$ . A *noisy* channel or storage device changes  $y$  into a corrupted word  $y'$  of the same length.<sup>1</sup> An *FEC decoding* function  $D$  maps  $y'$  back to a  $k$ -bit word  $x' = D(y')$ , which becomes the output of the process. The transmission is *successful* if  $x' = x$ .

There are many different FEC codes, some of which involve rather sophisticated mathematics. We present a particularly simple one here. For the purposes of this problem set, we take  $k = 256$  and  $n = 288$ . In terms of bytes,  $x$  is 32 bytes long and  $y = E(x)$  is 36 bytes long. The first 32 bytes of  $E(x)$  are simply  $x$  itself. The last four bytes are the redundant *error control* bytes that allow for the detection and correction of errors. In our code, each bit of each error control byte is the XOR (exclusive-or) of a different subset of bits of the data block.

The XOR ( $\oplus$ ) operation is defined by Table 1 and is computed by the C bitwise operator '^'. Note that XOR is the same as the arithmetic sum modulo 2 when the arguments are restricted to values in  $\{0, 1\}$ . This follows since  $1 + 1 = 2 \equiv 0 \pmod{2}$ .

Table 1: Truth table for XOR.

$x$	$y$	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

#### 1.1 Encoding function

To compute the error control bytes, we arrange the 256 bits of the data block into a matrix  $M$  of 16 rows of 16 bits each. The first row of  $M$  consists of the the first two bytes of the data block, the second row the next two bytes, and so forth. Next, we compute the XOR of the bits in each row to yield 16 *row sums*, and we compute the XOR of the bits in each column to yield 16 *column sums*. The row sums are packed into two bytes  $r$  and the column sums are packed into two bytes  $c$ . The four error control bytes are  $rc$ , so  $E(x) = xrc$ .

An example may help make this process clear. Suppose  $x$  is the 32-character string

$x = \text{"Error-correcting\_codes\_are\_fun.\_"}'$

<sup>1</sup>We do not consider *insertion errors* and *deletion errors* here, although they can be important in practice.

Here I have printed the symbol ‘ $\_$ ’ to indicate a space in the string.<sup>2</sup>

Each character of this string is represented by a unique byte according to the ASCII code. For example, the code for the letter ‘E’ is the byte ‘01000101’, which is the integer 69 when regarded as a binary number. Table 2 shows the decimal and binary values for each of the 32 characters of our message. Note that the string has been placed in the table in *row major order*, meaning that the first two characters “Er” appear in the first row of the table, the next two characters “ro” appear in the second row, and so forth.

Table 2: 32 message bytes printed in various ways.

Chr	Dec	Binary	Chr	Dec	Binary
E	69	01000101	r	114	01110010
r	114	01110010	o	111	01101111
r	114	01110010	-	45	00101101
c	99	01100011	o	111	01101111
r	114	01110010	r	114	01110010
e	101	01100101	c	99	01100011
t	116	01110100	i	105	01101001
n	110	01101110	g	103	01100111
$\_$	32	00100000	c	99	01100011
o	111	01101111	d	100	01100100
e	101	01100101	s	115	01110011
$\_$	32	00100000	a	97	01100001
r	114	01110010	e	101	01100101
$\_$	32	00100000	f	102	01100110
u	117	01110101	n	110	01101110
.	46	00101110	$\_$	32	00100000

Table 3 gives the bit matrix  $M$  that corresponds to the message  $x$ . Table 4 gives the augmented matrix  $\hat{M}$ , which is just  $M$  with the row and column sums added. Notice that the lower right corner of  $\hat{M}$  is blank since we don’t use the value that would naturally go there. From  $\hat{M}$ , we can read off the row sum bytes  $r = (10000000)(11100101)$  and the column sum bytes  $c = (00101000)(00011000)$ .

## 1.2 Decoding function

The goal of the decoding function is to recover the original data block  $x$  from the corrupted code word  $y'$ . This won’t always be possible. Indeed, if enough bits of  $y$  are corrupted, any message could be changed into any other. However, we can guarantee to recover the original message if only one bit is corrupted. If more than one bit is corrupted, we might be able to detect that an error occurred but not know how to correct the error, in which case we should report that uncorrectable errors were detected. However, it is also possible that multiple errors will change the original code block into a valid code block for a different message, so the errors won’t be detected and a bad output will be produced. It can also happen that multiple errors will change the original code block into one that looks like one with a correctable one-bit error. In this case, the process of “correcting” the apparently bad bit might actually change a correct data block into an incorrect one.

<sup>2</sup>We ignore the null byte that C uses to terminate each string and consider only the 32 actual data characters.

Table 3: Example message as a bit matrix.

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Table 4: Bit matrix with row and column sums.

$$\hat{M} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The usefulness of FEC error control depends on the probability  $p$  of a bit error and the severity of the consequences of corrupted data. When  $p$  is small, the probability of a single bit error in an  $n$ -bit block is approximately  $np$ , whereas the probability of two errors is roughly  $(np)^2/2$ . In this case, among blocks with any errors at all, most will have only one error and hence be successfully corrected by the decoder.

Here's how the decoding function  $D(y')$  is defined. Let  $y' = x'r'c'$ , where  $x'$  is 32-bytes long and  $r'$  and  $c'$  are two bytes each. Let  $r$  and  $c$  be the correct row and column sums for  $x'$ , i.e.,  $E(x') = x'rc$ . Let  $n_r$  be the number of bit positions in which  $r'$  and  $r$  differ, and let  $n_c$  be the number of bit positions in which  $c'$  and  $c$  differ. We consider three cases:

**Case 1:** If  $n_r + n_c \leq 1$ , we define  $D(y') = x'$ .

**Case 2:** If  $n_r = n_c = 1$ , then let  $i_r$  be the position of the differing bit in  $r$ , and let  $i_c$  be the position of the differing bit in  $c$ . We assume that these two discrepancies were caused by an error in bit number  $(i_r, i_c)$  of the data block  $x'$  (where we're numbering the bits according to their

position in the bit matrix associated with  $x'$ ). Let  $x''$  be the same as  $x'$  in all bit positions except for position  $(i_r, i_c)$ , and let the bit in that position of  $x''$  have the opposite value as in  $x'$ . We define  $D(y') = x''$ .

**Case 3:** In all other cases, we define  $D(y') = x'$  and report that uncorrectable errors have occurred.

We claim that if  $y = D(x)$  and  $y'$  differs from  $y$  in at most one bit, then  $D(y') = x$ . Hence, this scheme successfully corrects a single bit error. To see this, first note that if  $y' = y$ , then  $x' = x$  and case 1 applies to give  $D(y') = x' = x$ . Now if  $y'$  differs from  $y$  in exactly one bit position, that position occurs either in  $x'$ , in  $r'$ , or in  $c'$ . If it occurs in  $x'$ , then case 2 applies, and  $D(y') = x'' = x$ . If it occurs in  $r'$  or in  $c'$ , then case 1 applies and  $D(y') = x' = x$ .

Table 5: A one-bit error in row 4, column 5.

$$\hat{M}' = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & \boxed{1} & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & \boxed{0} \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For example, Table 5 shows the effect of changing the bit in row 4 column 5 of  $\hat{M}$  in Table 4 from 0 to 1, yielding the new matrix  $\hat{M}'$  and corresponding corrupted code word  $y'$ . As a result of this change, the boxed row and column sums are both wrong. Hence, the decoding algorithm will find that  $n_r = n_c = 1$ ,  $i_r = 4$  and  $i_c = 5$ . By case 2 above, the error will be corrected by flipping bit (4, 5) back from 1 to 0, yielding correct output.

Many but not all multiple-bit errors will lead to case 3 and the information that uncorrectable errors occurred. However, as few as 2 bit errors can lead to undetected erroneous output. For example, if there is an error in both  $r'$  and  $c'$  but not in  $x'$ , rule 2 will apply and produce  $D(y') = x'' \neq x' = x$ .

## 2 Assignment

You are to write two commands `encode` and `decode` to implement a forward error correction scheme based on the above method.

The `encode` command takes a single file name argument, opens that file for *input*, encodes the file, and writes the result to *standard output*.

The `decode` command takes a single file name argument, opens that file for *output*, reads an encoded file from *standard input*, and writes the decoded error-corrected data to that file. Any messages about detected errors that cannot be corrected should be written to standard error (`stderr`).

I will supply a program `noise` that takes a command line argument giving the probability  $p$  of error in a single bit. It will read bytes from standard input, flip each bit randomly with probability  $p$ , and write the corrupted bytes to standard output.

For example, the pipeline

```
encode file.in | noise .001 | decode file.out
```

reads `file.in` and writes `file.out`. If no block of data has more than one error, then `file.out` will be identical to `file.in`. (Use the `cmp` command to test them for equality.) If more than one error occurs, then `decode` will report in most cases that there were uncorrectable errors, but as discussed above, certain multiple error combinations cannot be detected.

With a 0.001 error probability, there is 1 chance in 1000 of each *bit* being flipped, so the probability of a bit being correct is 0.999. The probability of no errors in a 288-bit block is thus  $0.999^{288} \approx 0.750$ . The probability of exactly one error is  $288 \times .001 \times 0.999^{287} \approx .216$ . Thus, the probability of 2 or more errors is approximately 0.034. Without FEC, 25% of the blocks would be corrupted; with FEC, only 3.4% are corrupted after decoding.

### 3 Detailed specifications

`encode` should read its input file 32 bytes at a time. For each 32-byte block  $x$ , it should write out the FEC-protected 36-byte block  $E(x)$ . Generally the length of the input file will not be an exact multiple of 32 bytes. Let  $k$  be the length of the last block  $x$  of the file. If  $k < 32$ , then one should pretend that the last block is filled with 0's for the purpose of computing the row and column sums  $r$  and  $c$ . However, all that should be written to standard output for that block are the  $k + 4$  bytes  $xrc$ .

`decode` will read its input stream in 36-byte blocks. If the last block has length  $k + 4$ , it means that the last data block has length  $k$ , so only the first  $k$  bytes of the decoded result should be written to the output file.

The matrix  $M$  should be stored as a  $16 \times 2$  array of bytes. In C, it should be declared as

```
unsigned char M[16][2];
```

Different strategies are appropriate for computing the row and column sums. The column sums can be computed in parallel, 8 columns at a time, by using the XOR bitstring operator applied to the bytes in the columns of  $M$ . This is much more efficient than doing them a bit at a time.

Computing the row sums efficiently is more complicated. The straightforward approach is to write a loop that extracts the bits one at a time and XOR's them together. This is done for each of the two bytes comprising the row, and the results from the two bytes are XOR'ed together to form the final answer. This method can be improved by first XOR'ing together the two bytes comprising a row of  $M$  and then using a loop to XOR together the bits of the result.

Still further improvements are possible (but not required for this problem set). One is to use the observation that the statement

```
x = x & (x-1);
```

has the effect of changing the rightmost bit of  $x$  from 1 to 0 and leaving the other bits of  $x$  alone. For example, if  $x$  is 00101100, then  $x-1$  is 00101011, so the new value of  $x$  is 00101000. Hence, the number of 1 bits in  $x$  gets reduced by one. The number of times this operation needs to be performed in order for  $x$  to become 0 is the number of 1 bits originally present in  $x$ .

A further improvement uses a recursive tree-like algorithm to count the bits in a word. See the [http://en.wikipedia.org/wiki/Hamming\\_weight](http://en.wikipedia.org/wiki/Hamming_weight) for details.

As usual, your code should show good modularity. You should write three code files. `encode.c` should contain the main program for the `encode` command. `decode.c` should contain the main program for the `decode` command. The code to implement the block encoding and decoding functions should go into a module `block.c` with header file `block.h`. The block module should define a type `block` which, as usual, will be a pointer to a `struct block`. The fields of that struct will be the ones that you need to carry out the encoding and decoding functions. Unlike previous assignments, I will not design the interface for you but will leave it to you to figure out for yourself. As usual, the goal is to make as clean a separation between the various parts of the program as possible.

## 4 Deliverables

You should submit source code for your modules and a `Makefile` for building the two commands: `encode` and `decode`. You should also submit unit test data to exercise as much of your code as you can. In particular, your tests should include both legal and illegal files, files whose lengths are and are not exact or near multiples of the block lengths, and so forth.