```c
/* lab2.c
 * Basil Lin
 * basill
 * ECE 222, Fall 2016
 * MP2
 *
 * Purpose: Use bitwise operators to encode and decode ASCII characters and fix
 *          erroneous bits
 *
 * Assumptions:
 *  #1:  The menu driven input was provided and must be used exactly
 *       as written.  A user can enter commands:
 *            enc CUt
 *            dec 0E8A549C
 *            quit
 *       Encoding takes three printable ASCII letters
 *       Decoding takes up to eight HEX digits. If exactly eight digits are
 *            entered, the first digit must be 0 or 1.
 *            Leading zeros can be dropped.
 *
 *  #2:  The string and character type libraries cannot be used except as
 *       already provided.  These libraries are for checking inputs in main
 *       and in printing after decoding is complete.  They cannot be used
 *       for anyother purpose.
 *
 *  #3:  No arrays can be used (excpet as already provided for collecting
 *       keyboard input).  You must use bitwise operators for all encoding
 *       and decoding.  If you want to use an array as a lookup table you
 *       must first propose your design and get it approved.  Designs that
 *       use tables to avoid bitwise operators will not be approved.  There
 *       are many good and simple designs that do not require tables.
 *
 *  #4   No changes to the code in main.  Your code must be placed in
 *       functions.  Additional functions are encouraged.
 *
 * Bugs:
 *
 * See the ECE 223 programming guide
 *
 * If your formatting is not consistent you must fix it.  You can easily
 * reformat (and automatically indent) your code using the astyle
 * command.  If it is not installed use the Ubuntu Software Center to
 * install astyle.  Then in a terminal on the command line do
 *
 *     astyle --style=kr lab1.c
 *
 * See "man astyle" for different styles.  Replace "kr" with one of
 * ansi, java, gnu, linux, or google to see different options.  Or, set up
 * your own style.
 *
 * To create a nicely formated PDF file for printing install the enscript
 * command.  To create a PDF for "file.c" in landscape with 2 columns do:
 *     enscript file.c -G2rE -o - | ps2pdf - file.pdf
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXLINE 100

// function prototypes
void encode(unsigned char first_letter, unsigned char second_letter,
        unsigned char third_letter);
void decode(unsigned int codeword);

int main()
{
    char line[MAXLINE];
    char command[MAXLINE];
    char inputcs[MAXLINE];
    int  items;
    int i, invalid;
    unsigned int codeword;

    printf("\nMP2: encoding and decoding (29, 24) Hamming code.\n");
    printf("Commands:\n\tenc 3-letters\n\tdec 8-hex-digits\n\tquit\n");

    // each call to fgets, collects one line of input and stores in line
    while (fgets(line, MAXLINE, stdin) != NULL) {
        items = sscanf(line, "%s%s", command, inputcs);
        if (items == 1 && strcmp(command, "quit") == 0) {
            break;
        } else if (items == 2 && strcmp(command, "enc") == 0) {
            // encoding
            if (strlen(inputcs) != 3 || !isprint(inputcs[0]) ||
                    !isprint(inputcs[1]) || !isprint(inputcs[2])) {
                printf("Invalid input to encoder: %s\n", inputcs);
            } else {
                encode(inputcs[0], inputcs[1], inputcs[2]);
            }
        } else if (items == 2 && strcmp(command, "dec") == 0) {
            // decoding: convert hex digits to integer
            items = sscanf(inputcs, "%x", &codeword);
            if (items != 1 || strlen(inputcs) > 8) {
                printf("Invalid input to decoder: %s\n", inputcs);
            } else {
                // verify all digits are hex characters because
                // scanf does not reject invalid letters
                for (i=0, invalid=0; i < strlen(inputcs) && !invalid; i++) {
                    if (!isxdigit(inputcs[i]))
                        invalid = 1;
                }
                // if 8 digits, leading digit must be 1 or 0
                if (invalid) {
                    printf("Invalid decoder digits: %s\n", inputcs);
                } else if (strlen(inputcs) == 8 && inputcs[0] != '1'
                        && inputcs[0] != '0') {
                    printf("Invalid decoder leading digit: %s\n", inputcs);
                } else {
                    decode(codeword);
                }
            }
        } else {
            printf("# :%s", line);
        }
    }
    printf("Goodbye\n");
    return 0;
}

/* encode: calculates parity bits and prints codeword
 *
 * input: three ASCII characters
 * assumptions: input is valid
 *
 * Example: if input letters are is 'C', 'U', and 't'
 * the final print must be:
 *  ---01110 10001010 01010100 10011100
 *  Codeword: 0x0E8A549C
 */
void encode(unsigned char first_letter, unsigned char second_letter,
        unsigned char third_letter)
```

```
{
    // you must construct the codeword
    unsigned int codeword = 0;

    printf("%9s%9c%9c%9c\n", "Encoding:", third_letter, second_letter, first_letter
);
    printf(" 0x    00%9x%9x%9x\n", third_letter, second_letter, first_letter);

    int third_let = third_letter << 16;
    int second_let = second_letter << 8;
    int total = third_let + second_let + first_letter; //finds binary combination o
f characters
    int temptotal = total; //variable used later for printing binary combination of
 characters
    int shift1, shift2, shift3; //temporarily stores bits from total
    int P1, P2, P4, P8, P16; //parity bits
    int p1, p2, p4, p8, p16;
    int count1, count2, count4, count8, count16;
    int P_1, P_2, P_4, P_8, P_16;
    int i;

    //saves more digits, clears bits, and adds back saved values
    shift1 = 0x7FF & total;
    total <<= 5;
    total &= ~0xFFFF;

    shift2 = 0xF & shift1;
    shift1 <<= 4;
    shift1 &= ~0xFF;
    total = total + shift1;

    shift3 = 0x1 & shift2;
    shift2 <<= 3;
    shift2 &= ~0xF;
    total = total + shift2;

    shift3 <<= 2;
    shift3 &= ~0x3;
    total += shift3;

    //finds parity bits
    p1 = 0x15555555 & total;
    count1 = 0;
    while(p1 > 0) {
        if((p1 & 1) == 1)
            count1++;
        p1 >>= 1;
    }
    if((count1 % 2) == 1)
        P1 = 1;
    else
        P1 = 0;

    p2 = 0x6666666 & total;
    count2 = 0;
    while(p2 > 0) {
        if((p2 & 1) == 1)
            count2++;
        p2 >>= 1;
    }
    if((count2 % 2) == 1)
        P2 = 1;
    else
        P2 = 0;

    p4 = 0x18787878 & total;
    count4 = 0;
    while(p4 > 0) {
        if((p4 & 1) == 1)
            count4++;
        p4 >>= 1;
    }
    if((count4 % 2) == 1)
        P4 = 1;
    else
        P4 = 0;

    p8 = 0x1F807F80 & total;
    count8 = 0;
    while(p8 > 0) {
        if((p8 & 1) == 1)
            count8++;
        p8 >>= 1;
    }
    if((count8 % 2) == 1)
        P8 = 1;
    else
        P8 = 0;

    p16 = 0x1FFF8000 & total;
    count16 = 0;
    while(p16 > 0) {
        if((p16 & 1) == 1)
            count16++;
        p16 >>= 1;
    }
    if((count16 % 2) == 1)
        P16 = 1;
    else
        P16 = 0;

    //finds actual values of parity bits for adding
    P_16 = P16 * 0x8000;
    P_8 = P8 * 0x80;
    P_4 = P4 * 0x8;
    P_2 = P2 * 0x2;
    P_1 = P1 * 0x1;

    codeword = total + P_16 + P_8 + P_4 + P_2 + P_1;

    //prints original information word in binary
    printf(" -------- ");
    for(i=23; i>=0; i--) {
        printf("%d", (temptotal & (1 << i)) >> i);
        if((i == 8) || (i == 16))
            printf(" ");
    }
    printf("\n");

    // prints the parity bits, one bit per line.  Do not change
    printf("P1 : %d\n", P1);
    printf("P2 : %d\n", P2);
    printf("P4 : %d\n", P4);
    printf("P8 : %d\n", P8);
    printf("P16: %d\n", P16);

    // print the codeword bits in binary form with spaces
    printf(" ---");
    for(i=28; i>=0; i--) {
        printf("%d", (codeword & (1 << i)) >> i);
        if((i == 8) || (i == 16) || (i == 24))
            printf(" ");
    }
    printf("\n");
```

```
    // print the codeword in hex format
    printf(" Codeword: 0x%.8X\n", codeword);
    printf("\n");
}

/* decode: checks parity bits and prints information characters
 *
 * input: A 29-bit codeword
 * assumptions: the codeword has either no or only one error.
 *
 *              the information characters may not be printable
 *
 * FYI: when a codeword has more than one error the decoding algorithm
 * may generate incorrect information bits.  In a practical system
 * inforamtion is grouped into multiple codewords and a final CRC
 * code verifies if all codewords are correct.  We will not
 * implement all of the details of the system in this project.
 *
 * Example: if the codeword is 0x0E8A549C
 * the final print must be:
 *  No error
 *  -------- 01110100 01010101 01000011
 *  Information Word: 0x745543 (CUt)
 *
 * Example with one error in codeword bit 21: 0x0E9A549C
 * Notice the 8 in the previous example has been changed to a 9
 * the final print must be:
 *  Corrected bit: 21
 *  -------- 01110100 01010101 01000011
 *  Information Word: 0x745543 (CUt)
 */
void decode(unsigned int codeword)
{
    // you must determine these values:
    int codeword1 = codeword & ~0x808B; //codeword without parity bits
    int P1, P2, P4, P8, P16;
    int E1, E2, E4, E8, E16; //error values
    int E_1, E_2, E_4, E_8, E_16;
    int shift1, shift2, shift3;
    int p1, p2, p4, p8, p16;
    int count1, count2, count4, count8, count16;
    unsigned int info_word;
    int bit_error_location;
    int error_value;
    int i;
    unsigned char first_letter;
    unsigned char second_letter;
    unsigned char third_letter;

    //recalculates parity bits from codeword and assigns error values
    p1 = 0x15555555 & codeword1;
    count1 = 0;
    while(p1 > 0) {
        if((p1 & 1) == 1)
            count1++;
        p1 = p1 >> 1;
    }
    if((count1 % 2) == 1)
        P1 = 1;
    else
        P1 = 0;
    if(P1 == (0x1 & codeword)) {
        E1 = 0;
    }
    else
        E1 = 1;

    p2 = 0x6666666 & codeword1;
    count2 = 0;
    while(p2 > 0) {
        if((p2 & 1) == 1)
            count2++;
        p2 >>= 1;
    }
    if((count2 % 2) == 1)
        P2 = 1;
    else
        P2 = 0;
    if(P2*2 == (0x2 & codeword)) {
        E2 = 0;
    }
    else
        E2 = 1;

    p4 = 0x18787878 & codeword1;
    count4 = 0;
    while(p4 > 0) {
        if((p4 & 1) == 1)
            count4++;
        p4 >>= 1;
    }
    if((count4 % 2) == 1)
        P4 = 1;
    else
        P4 = 0;
    if(P4*8 == (0x8 & codeword)) {
        E4 = 0;
    }
    else
        E4 = 1;

    p8 = 0x1F807F80 & codeword1;
    count8 = 0;
    while(p8 > 0) {
        if((p8 & 1) == 1)
            count8++;
        p8 >>= 1;
    }
    if((count8 % 2) == 1)
        P8 = 1;
    else
        P8 = 0;
    if(P8*128 == (0x80 & codeword)) {
        E8 = 0;
    }
    else
        E8 = 1;

    p16 = 0x1FFF8000 & codeword1;
    count16 = 0;
    while(p16 > 0) {
        if((p16 & 1) == 1)
            count16++;
        p16 >>= 1;
    }
    if((count16 % 2) == 1)
        P16 = 1;
    else
        P16 = 0;
    if(P16*32768 == (0x8000 & codeword)) {
        E16 = 0;
    }
    else
```

```
    E16 = 1;

//finds int values of error to add for error location
E_16 = E16 * 0x10;
E_8 = E8 * 0x8;
E_4 = E4 * 0x4;
E_2 = E2 * 0x2;
E_1 = E1 * 0x1;

printf("Decoding: 0x%.8X\n", codeword);

bit_error_location = E_16 + E_8 + E_4 + E_2 + E_1;

//finds int values bit error location
error_value = 1;
for(i = 0; i < bit_error_location-1; i++) {
    error_value *= 2;
}

//corrects erroneous bit
if((error_value & codeword) == error_value)
    codeword -= error_value;
else
    codeword += error_value;

//creates info_word from corrected codeword
info_word = codeword >> 2;
shift1 = 0x1 & info_word;
info_word >>= 2;
shift2 = 0x7 & info_word;
shift2 <<= 1;
info_word >>= 4;
shift3 = 0x7F & info_word;
shift3 <<= 4;
info_word >>= 8;
info_word <<= 11;
info_word = info_word + shift1 + shift2 + shift3;

//converts info_word into three ASCII codes for printing
first_letter = info_word & 0xFF;
second_letter = (info_word & 0xFF00) >> 8;
third_letter = (info_word & 0xFF0000) >> 16;


// prints the error location bits, one bit per line.  Do not change
printf("E1 : %d\n", E1);
printf("E2 : %d\n", E2);
printf("E4 : %d\n", E4);
printf("E8 : %d\n", E8);
printf("E16: %d\n", E16);

// here is the required format for the prints.  Do not
// change the format but update the variables to match
// your design
if (bit_error_location == 0)
    printf(" No error\n");
else if (bit_error_location > 0 && bit_error_location <= 29) {
    printf(" Corrected bit: %d\n", bit_error_location);
} else
    printf(" Decoding failure: %d\n", bit_error_location);

//prints the info_word in binary format
info_word &= 0xFFFFFF;
printf(" -------- ");
for(i=23; i>=0; i--) {
    printf("%d", (info_word & (1 << i)) >> i);
    if((i == 8) || (i == 16))
        printf(" ");
    }
    printf("\n");

    // prints the information word in hex:
    printf(" Information Word: 0x%.6X", info_word);

    // You must convert the info_word into three characters for printing
    // only print information word as letters if 7-bit printable ASCII
    // otherwise print a space for non-printable information bits
    if ((first_letter & 0x80) == 0 && isprint(first_letter))
        printf(" (%c", first_letter);
    else
        printf(" ( ");
    if ((second_letter & 0x80) == 0 && isprint(second_letter))
        printf("%c", second_letter);
    else
        printf(" ");
    if ((third_letter & 0x80) == 0 && isprint(third_letter))
        printf("%c)\n", third_letter);
    else
        printf(" )\n");
    printf("\n");
}
```