```c
/* lab2.c
 * Christopher Brant
 * cbrant
 * ECE 222, Fall 2016
 * MP2
 *
 * Purpose:  To learn bitwise operations and how to implement them efficiently to
 *           then accomplish a different goal. Specifically to encode and decode
 *           sets of ascii text.
 *
 * 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: None left cause I fixed them all.
 *
 * 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);
// The next function creates the info word
unsigned int createinfword(unsigned char letter1, unsigned char letter2, unsigned c
har letter3);
void bininfprint(unsigned int info_word);       // Prints info word in binary
void bincodprint(unsigned int code_word);       // Prints code word in binary
unsigned int inftocode(unsigned int info_word); // Modifies info word to code word
int parityone(unsigned int code_word);          // Decides parity bit one
int paritytwo(unsigned int code_word);          // Decides parity bit two
int parityfour(unsigned int code_word);         // Decides parity bit four
int parityeight(unsigned int code_word);        // Decides parity bit eight
int paritysixteen(unsigned int code_word);      // Decides parity bit sixteen
int extp1(unsigned int code_word);              // Extracts parity bit one
int extp2(unsigned int code_word);              // Extracts parity bit two
int extp4(unsigned int code_word);              // Extracts parity bit four
int extp8(unsigned int code_word);              // Extracts parity bit eight
int extp16(unsigned int code_word);             // Extracts parity bit sixteen
unsigned int codetoinf(unsigned int code_word); // Modifies code word to info word
unsigned int char1(unsigned int info_word);     // Extracts letter one from info wo
rd
unsigned int char2(unsigned int info_word);     // Extracts letter two from info wo
rd
unsigned int char3(unsigned int info_word);     // Extracts letter three from info
word
unsigned int newcode(unsigned int code_word, int errbit);   // Modifies code word b
ased on error bit
unsigned int blankcode(unsigned int code_word); // Zeroes out parity bits for calcu
lations

// Add functions as necessary, Christopher.

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;
    unsigned int info_word;
    int P1, P2, P4, P8, P16;

    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);

    info_word = createinfword(first_letter, second_letter, third_letter);

    bininfprint(info_word);
    // print the information word in binary form with spaces

    codeword = inftocode(info_word);

    // Create parity bits
    P1 = parityone(codeword);
    codeword |= (P1 << 0);
    P2 = paritytwo(codeword);
    codeword |= (P2 << 1);
    P4 = parityfour(codeword);
    codeword |= (P4 << 3);
    P8 = parityeight(codeword);
    codeword |= (P8 << 7);
    P16 = paritysixteen(codeword);
    codeword |= (P16 << 15);

    // print the parity bits, one bit per line.  Do not change
    // the format, but you must change the dummy variable i
    // to match your design
    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
    bincodprint(codeword);
    // 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)
{
    unsigned int blank = blankcode(codeword);
    int p1, p2, p4, p8, p16;
    int errp1, errp2, errp4, errp8, errp16;
    unsigned int info_word = codetoinf(codeword);

    // Calculate correct parity bits
    p1 = parityone(blank);
    p2 = paritytwo(blank);
    p4 = parityfour(blank);
    p8 = parityeight(blank);
    p16 = paritysixteen(blank);

    // Check for parity bit errors
    errp1 = extp1(codeword) ^ p1;
    errp2 = extp2(codeword) ^ p2;
    errp4 = extp4(codeword) ^ p4;
    errp8 = extp8(codeword) ^ p8;
    errp16 = extp16(codeword) ^ p16;

    // you must determine these values:
    unsigned char first_letter;
    unsigned char second_letter;
    unsigned char third_letter;

    // Determine error bit location if error is present
    int bit_error_location = (16 * errp16) + (8 * errp8) + (4 * errp4) + (2 * errp2
) + (1 * errp1);
    printf("Decoding: 0x%.8X\n", codeword);

    codeword = newcode(codeword, bit_error_location);
    info_word = codetoinf(codeword);
    first_letter = char1(info_word);
    second_letter = char2(info_word);
    third_letter = char3(info_word);

    // print the error location bits, one bit per line.  Do not change
    // the format, but you must change the dummy variable i
    // to match your design
    printf("E1 : %d\n", errp1);
    printf("E2 : %d\n", errp2);
    printf("E4 : %d\n", errp4);
    printf("E8 : %d\n", errp8);
    printf("E16: %d\n", errp16);

    // 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);

    // you must print the info_word in binary format
    bininfprint(info_word);
    // print 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");
}

/* All of the following functions have been commented for their overall
   functionality at their intialization at the beginning of the file */

unsigned int createinfword(unsigned char letter1, unsigned char letter2, unsigned c
har letter3)
{
    /* Basic idea here was to create an intermediate variable with the correct bits
       moved into the correct places and then just OR it with a blank variable */

    /* Technically could be done without the "holder" variable and just use
       info_word in its place the whole time, but the idea for the algorithm
       is clearer in the current layout */
    unsigned int info_word = 0x0;
    unsigned int holder = letter3 << 16;
    holder |= (letter2 << 8);
    holder |= letter1;

    info_word |= holder;

    return info_word;
}

void bininfprint(unsigned int info_word)
{
    // Super basic, this just reads each bit and prints it one at a time.
    int check, i;

    printf(" -------- ");

    for (i = 23; i >= 0; i--)
    {
        if (i == 7 || i == 15)
            printf(" ");

        check = (info_word & (1 << i)) >> i;
        printf("%d", check);
    }

    printf("\n");
}

void bincodprint(unsigned int code_word)
{
    // Also super basic, this also just reads each bit and prints them one at a tim
e.
    int check, i;

    printf(" ---");

    for (i = 28; i >= 0; i--)
    {
        if (i == 7 || i == 15 || i == 23)
            printf(" ");

        check = (code_word & (1 << i)) >> i;
        printf("%d", check);
    }

    printf("\n");
}

unsigned int inftocode(unsigned int info_word)
{
    /*Creates intermediate variables and uses them to copy and pad each grouping of
 bits
       to their correct bit positions */
    unsigned int hold = 0xFFF800;
    unsigned int check = (info_word & hold) << 5;
    unsigned int code_word = 0x0;

    code_word |= check;

    hold = 0x7F0;
    check = (info_word & hold) << 4;
    code_word |= check;

    hold = 0xE;
    check = (info_word & hold) << 3;
    code_word |= check;

    hold = 0x1;
    check = (info_word & hold) << 2;
    code_word |= check;

    return code_word;
}

int parityone(unsigned int code_word)
```

```
{
    // Checks every other bit from position 3 to position 29 and sums that

    // Then if the sum isn’t even it returns a 1 for the parity bit for even parity
    int check = 0;
    int p1 = 0;
    int i;

    for (i = 28; i >= 0; i -= 2)
    {
        check += ((code_word & (1 << i)) >> i);
    }

    check %= 2;

    if (check == 1)
        p1 = 1;

    return p1;
}

int paritytwo(unsigned int code_word)
{
    // Checks every other pair of bits starting from positions 2&3 to positions 26&
27

    // If the sum isn’t even it returns a 1 for the parity bit for even parity
    int check = 0;
    int p2 = 0;
    int i;

    for (i = 26; i >= 1; i -= 4)
    {
        check += ((code_word & (1 << i)) >> i);
        check += ((code_word & (1 << (i-1))) >> (i - 1));
    }

    check %= 2;

    if (check == 1)
        p2 = 1;

    return p2;
}

int parityfour(unsigned int code_word)
{
    /* Checks every other group of four bits until you run out of bits.
       Starts with positions 4-7 and stops when it runs out of bits trying to do 28
-31
       and only gets 28-29. */

    // If the sum isn’t even it returns a 1 for the parity bit for even parity
    int check = 0;
    int p4 = 0;
    int i, j;

    for (i = 28; i >= 27; i--)
        check += ((code_word & (1 << i)) >> i);

    for (i = 22; i >= 3; i -= 4)
    {
        for (j = 0; j < 4; j++, i--)
        {
            check += ((code_word & (1 << i)) >> i);
        }
    }

    check %= 2;

    if (check == 1)
        p4 = 1;

    return p4;
}

int parityeight(unsigned int code_word)
{
    /* Checks every other group of 8 bits until it runs out of bits.
       Therefore it checks bits 8-15 and then bits 24-29 since it can’t check 24-31
. */

    // If the sum isn’t even it returns a 1 for the parity bit for even parity
    int check = 0;
    int p8 = 0;
    int i;

    for (i = 28; i >= 23; i--)
    {
        check += ((code_word & (1 << i)) >> i);
    }

    for (i = 14; i >= 7; i--)
    {
        check += ((code_word & (1 << i)) >> i);
    }


    check %= 2;

    if (check == 1)
        p8 = 1;

    return p8;
}

int paritysixteen(unsigned int code_word)
{
    /* Theoretically checks every other group of 16, but in practice it only
       checks the series of bits 15-29. */

    // If the sum isn’t even it returns a 1 for the parity bit for even parity
    int check = 0;
    int p16 = 0;
    int i;

    for (i = 28; i >= 15; i--)
    {
        check += ((code_word & (1 << i)) >> i);
    }

    check %= 2;

    if (check == 1)
        p16 = 1;

    return p16;
}

/* Each of the following extp# functions each read their respective parity bit and
then return
   that bit value to the variable that they are called by. */

int extp1(unsigned int code_word)
```

```c
{
    unsigned int exp1, mid, parcheck = 0x1;
    mid = code_word & parcheck;
    exp1 = (mid & (1 << 0)) >> 0;

    return exp1;
}

int extp2(unsigned int code_word)
{
    unsigned int exp2, mid, parcheck = 0x2;
    mid = code_word & parcheck;
    exp2 = (mid & (1 << 1)) >> 1;

    return exp2;
}

int extp4(unsigned int code_word)
{
    unsigned int exp4, mid, parcheck = 0x8;
    mid = code_word & parcheck;
    exp4 = (mid & (1 << 3)) >> 3;

    return exp4;
}

int extp8(unsigned int code_word)
{
    unsigned int exp8, mid, parcheck = 0x80;
    mid = code_word & parcheck;
    exp8 = (mid & (1 << 7)) >> 7;

    return exp8;
}

int extp16(unsigned int code_word)
{
    unsigned int exp16, mid, parcheck = 0x8000;
    mid = code_word & parcheck;
    exp16 = (mid & (1 << 15)) >> 15;

    return exp16;
}

unsigned int codetoinf(unsigned int code_word)
{
    // Modifies the code word to its information word counterpart.
    unsigned int info_word = 0;
    unsigned int mid = 0;
    unsigned int hold, grab = ˜0x808B;
    unsigned int intermed = code_word & grab;   // Zeroes out parity bits here.
    int i, j, k;

    // Loops through until every bit has been shifted into its correct spot.
    for (i = 2; i <= 28; i++)
    {
        grab = 1;

        if (i == 15 || i == 7 || i == 3)
            i++;        // Skips parity bit locations

        for (j = 1; j <= i; j++)
            grab *= 2;  // Places a 1 in the correct binary location for the desire
d bit check.

        // The next four "if" statements determine how far to shift the current bit
.
        if (i <= 28 && i >= 16)
            k = 5;

        if (i <= 14 && i >= 8)
            k = 4;

        if (i <= 6 && i >= 4)
            k = 3;

        if (i == 2)
            k = 2;

        hold = intermed & grab; // Stores intermediate value.
        mid = hold >> k;        // Shifts bit to correct location.
        info_word |= mid;       // Stores correct value into information word varia
ble.
    }

    return info_word;
}

/* The following three char# functions are identical except for the placeholders
   and the amount they shift their values. Otherwise, they all do the same thing,
   which is grab the correct bits, save them intermediately, shift them to the corr
ect positions,
   and then save them in the returning variable */

unsigned int char1(unsigned int info_word)
{
    unsigned int letter1, mid = 0xFF;
    mid &= info_word;
    letter1 = mid;

    return letter1;
}

unsigned int char2(unsigned int info_word)
{
    unsigned int letter2, mid = 0xFF00;
    mid &= info_word;
    letter2 = mid >> 8;

    return letter2;
}

unsigned int char3(unsigned int info_word)
{
    unsigned int letter3, mid = 0xFF0000;
    mid &= info_word;
    letter3 = mid >> 16;

    return letter3;
}

unsigned int newcode(unsigned int code_word, int errbit)
{
    int errbitpos = errbit - 1;     // Corrects error bit position to CS notation

    code_word ^= (1 << errbitpos);  // Flips the value of the error bit and then re
turns that value

    return code_word;
}

unsigned int blankcode(unsigned int code_word)
{
    // This function zeroes out the parity bits so that they can be recalculated du
ring decoding.
    unsigned int grab = ˜0x808B;
    code_word &= grab;

    return code_word;
}
```