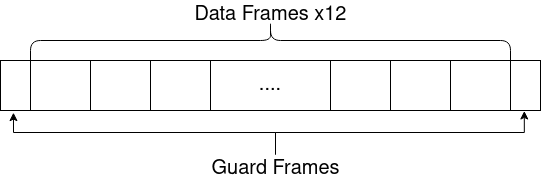
Barcode Scanning

Task

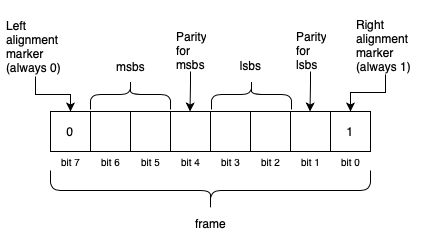
You are designing a barcode scanner for a new barcode format, used for electronics parts organisation in the Mechatronics Lab. The image sensor for the barcode scanner has already been developed, so now it's up to you to develop the decoder that will actually find out what each barcode means.

Barcodes

A barcode encodes a 12 digit number, where each digit is between 0 and 9. It can have any number of rows, but the number of columns is fixed at 102. These 102 columns are divided between guard frames and data frames as shown below. A frame is 8 of consecutive columns which encode a single piece of information. Every barcode contains a left guard frame and a right guard frame. These are used to locate the barcode within the image for easier processing. Each guard frame is 3 columns wide and is always of the form 101, where a 1 represents a black pixel and a 0 represents a white pixel.



In between the guard frames, there are 12 data frames, each of which is 8 columns wide. Each frame of the barcode encodes a single number, ranging from 0-9. Each column of the frame is encoded in the barcode as a pixel, so each frame is 8 pixels wide. From left to right, the purpose of the columns is as follows:



Note that there may be issues with how the barcode reader imaged the barcode and as such some numbers may not be correctly encoded in the barcode. In order to provide some basic error checking, a parity bit for each of the 2 MSB's (Most Significant Bits) and LSB's (Least Significant Bits) is included. For a brief introduction to parity errors, see [this website](https://www.computerhope.com/jargon/p/paritybi.htm). For the purposes of this task, you may assume that if the parity bits are correct, then the frame is correctly encoded. If at least one parity bit is incorrect for every row in a specific frame, then that frame is unable to be read and the barcode is invalid.

For example, if a certain frame were encoded in the image as 00111101, we can split it up into sections as follows (ignoring the alignment markers):

* MSBs = 01
* Parity bit 1 = 1
* LSBs = 11
* Parity bit 2 = 0

The parity bits are correct, and thus the encoded number is 0b0111 = 7.

It should be noted that while a 4 bit number (as encoded in each frame) can store a number from 0-15, a frame encodes a number from 0-9. Thus, only the least significant digit of the encoded number should be used as the decoded result from a specific frame. For example, if the bits in the above example represented 12, then the encoded number would be 2.

It is also possible that the camera system photographed some of the barcodes upside down. Your program should be able to identify if this has occurred and read the barcode correctly regardless. (Hint: Use the left and right alignment markers to check if the barcode is reversed).

NOTE: a frame is only valid both the parity bits are set correctly (if bit1 XOR bit2 == parity)

Input

Your program should expect at least one argument from the command line, with one optional argument, in the following format:

./barcode\_reader <filename.bmp> [-d]

The first argument (which must be given) is the filename for the barcode image. If this is not present upon program invocation the program should print:

No bmp image filename provided.

The second argument may optionally be provided, and will always take the form -d (no error checking is required for flag correctness).

Output

If the program is invoked with the -d flag, then **diagnostic** mode is enabled. In diagnostic mode, the program should read the given BMP file and simply output the height and width of the given image, in the following format:

Read file <filename.bmp>

Width: XX

Height: YY

The program should then immediately exit.

If the -d flag is not given, then the program should instead operate in **scanning** mode. In scanning mode, the program should read the BMP file and output the decoded number (via the calculation for each frame explained in the *Barcode Format* section) in space-separated format as according to the following format:

5 4 9 6 5 1 3 4 8 4 6 0

If there is a frame in the barcode where there is no row that encodes valid information, the program should output in the format:

Unable to read frame: X

If there are more than one such frames in the barcode, then the program should output each unidentified frame in ascending order:

Unable to read frames: X Y Z

where X, Y and Z are numbers in the range 0 to 12.

Note that if the barcode is reversed, frame 0 is still the leftmost frame in the image (reading from left to right).

Note: All testcases expect a newline at the end of the output string.

The Scaffold

Aware that students are short on time towards the end of semester, the tutors have included a basic makefile for you to use in this task. It currently only uses bitmap.c, bitmap.h and main.c. You are welcome to use more files but you will need to edit the makefile to use them.

We have given you six example barcodes to develop and test your code, however we will be using many more during the testing phase to ensure your code accounts for any and all edge cases which may come up (within the bounds of the specification).

NOTE: You do not need to edit bitmap.c or bitmap.h, any errors which are printed from those files are due to other errors and can be used as diagnostic tools.

PS: don't forget to free your memory!!

Background Information

BMP File Format

Bitmap files (\*.bmp) have a relatively simple file format. A bitmap file consists of a 14-byte general header followed immediately by a DIB (device-independant bitmap) header containing metadata about the pixel format and image size. The pixel data follows immediately after the DIB header. The size of the pixel data can be calculated from the information in the header. Some example images have been included in the scaffold.

Running the hex dump utility   
xxd <file\_name.bmp>   
in the terminal window is a useful way of displaying the encoded BMP information.

For more information see <https://en.wikipedia.org/wiki/Hex_dump>

Bitmap File Header

A bitmap file always begins with a 14-byte header that describes the contents of the file. The bitmap file header always starts with the **2 bytes** 0x42 0x4d (i.e. BM). The next **4 bytes** give the size of the file in bytes. The next **4 bytes** are reserved. The next **4 bytes** indicate the offset from the start of the file to where the pixel data start.

DIB Header

The bitmap file header is always immediately followed by the DIB header. The first **4 bytes** of the DIB header indicate the size of the DIB header. The next **4 bytes** indicate the number of pixels in the horizontal direction. The next **4 bytes** indicate the number of pixels in the vertical direction. The remainder of the DIB header contains additional information about the image, such as colour data and compression, that is not relevant to this assignment.

If you are interested in learning more about the BMP file format see <https://en.wikipedia.org/wiki/BMP_file_format>

Image Pixel Data

Each pixel in the image is represented by **3 bytes** that give the values of the red, green and blue (R, G, B) content of the pixel. Each of the 3 bytes can have a value between 0 and 255 that indicates the amount of R, G or B in a pixel. Although a pixel can therefore have many (256 x 256 x 256 = 16,777,216) different colours, all of the pixels that represent the maze will be either black or white.

* Black: (R, G, B) = (0, 0, 0)
* White: (R, G, B) = (255, 255, 255)

For this assignment, a Black pixel (0 values for RGB) is a 1.