Engineering Principles   Name________________________________

Electrical Engineering

Color Wheel 4.8.1 Software: Images

Date____________________ Team _________

Problem statement

Develop a persistence-of-vision color wheel utilizing 8 RGB diodes and an Arduino UNO R3 board.

Programming: Getting Started

1. Find your Arduino code file. If it has been modified by another member of your team since you last worked on it you may need to download it from Canvas.
2. Download any image.c files that your group has made to the same directory as your code file. Make sure that there are no special characters (space,~!@#$%^&*) in your image file name.
3. Open your program file with Arduino. You should see tabs for your image files as well.

Programming: #include compiler directive

It is sometimes useful to include other files in your project. We will use the #include statement to incorporate the graphics files that your team members have made.

```
#include "myGraphicsFile.c"
```

This is a complier directive starting with # so we don't use a semicolon at the end. The filename must be in quotes, and be exactly as it shows on the tab – case sensitive. If you don't see any tabs for your graphics files, make sure they are in the same directory as your .ino file and that there are no special characters in the file name. If you change the filename you will have to quit Arduino and then open the .ino file again.

4. Add an #include statement near the beginning of your file for the graphics files you downloaded.

Programming: Structures

Structures (struct) are compound variables containing multiple related pieces of information. The graphics images that your group created using GIMP defines a structure which creates a variable called "`gimp_image`". `gimp_image` contains everything you need to know to display the image. Inside the `struct` are the `width` (64), `height` (8) and `bytes_per_pixel` (3) along with an array of characters called `pixel_data` representing the individual pixel values.

```
static const struct {
  unsigned int    width;
  unsigned int    height;
  unsigned int    bytes_per_pixel; /* 2:RGB16, 3:RGB, 4:RGBA */
  unsigned char   pixel_data[64 * 8 * 3 + 1];
} gimp_image = {
  64, 8, 3,

"\377^\304\0\0\0\0\0\0\377^\304\0\0\0\0\0\0\377^\304\0\0\0\0\0\0\377^\
304"
```

Sidenote: Most of the pixel data is represented in octal format. Each byte or character begins with a backslash followed by one to three digits ranging from 0 to 7. The largest number is /377 which is the binary number 11 111 111. In decimal this is 255. In hex it would be written 0xFF. You may also see text characters in the data list – the pixel value is the ASCII value of the character (A = 65, a = 97).

To access elements of a struct, use the struct variable name followed by a dot (.) followed by the element name within the struct.

```
gimp_image.width = 64;  // sets the image width to 64
pixelValue = gimpImage.pixel_data[35];  // retrieves the 36th pixel
```

Pixels are stored in the array by column, row, and then color (R1C1 Red, R1C1 Green, R1C1 Blue, R1C2 Red, R1C2 Green, R1C2 Blue, ...). The pixel array index can be calculated by adding color (0, 1 or 2) to (row * width + column) * bytes_per_pixel.

5. Create a function to return the value of a pixel from `gimp_image` given a row, column and color. Use `width` and `bytes_per_pixel` as stored in the `struct` so the function will work even if these parameters change. This function should `return` an integer (`int`, not `void`).

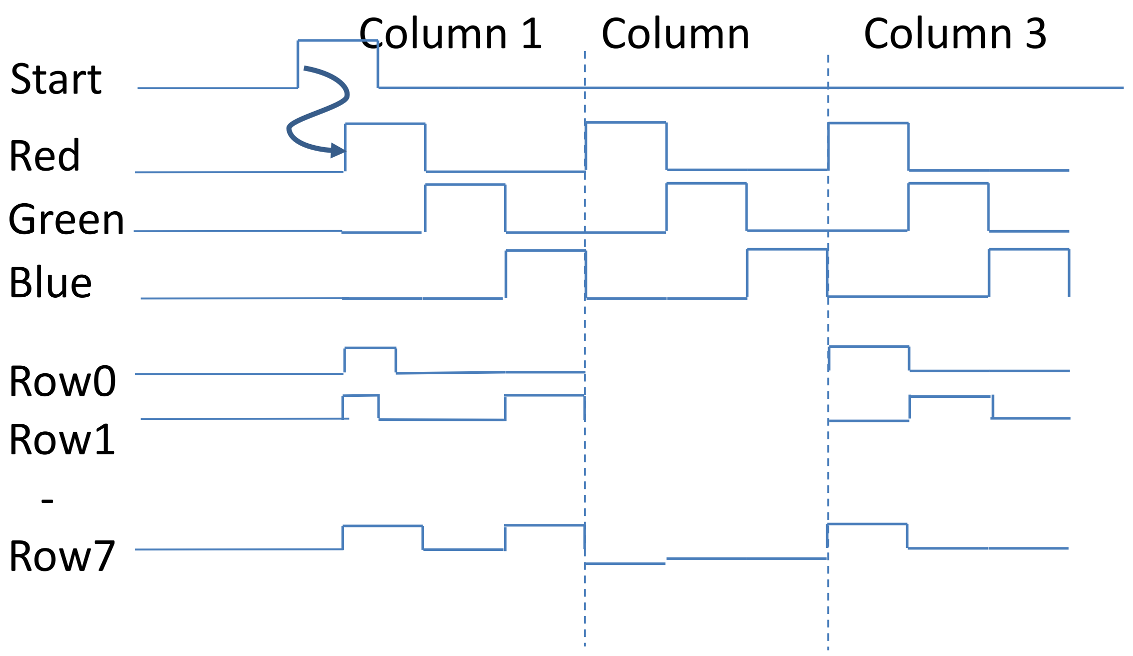
```
int getPixel(int row, int column, int color) {
  int index = (row * gimp_image.width + column ) *
      gimp_image.bytes_per_pixel;
  return gimp_image.pixel_data[index + color];
}
```

Programming: Time multiplexing

In the persistence-of-vision project we use time multiplexing to display each color of each column for a set a period of time before moving on to the next color/column. Working from the outside in, we scan through each column and then color. For each color we divide the time by 256 to allow for intensity variations and then scan each row. The following timing diagram shows how this works:

Color Wheel 4.8.1 Software: Images

Date____________________ Team __________



We can implement this timing diagram with nested for loops

6. Create a `for` loop that cycles through each of the columns in the picture (`gimp_image.width`).
7. Inside the column `for` loop create another `for` loop which cycles through the 3 colors (r, g and b). Turn the appropriate color pins on inside this loop.
8. Inside the color loop create a loop which counts from 0 to 255. This loop will adjust the intensity. If a pixel value is greater than the loop counter we turn it ON, otherwise we turn it off. A pixel with a value of 127 will be on for half of the time.
9. Create one final loop which cycles through the eight rows. In this loop use getPixel(row, column, color) and an `if()` and `else` command to check if we should turn the pixel on or off.
10. Clear the row and column pins after the loops have executed so that we don't leave anything on.

Programming: Finish and Save

11. **In your lab notebook** write down what you expect the code to do?
12. Test this code using your group's hardware.
13. **In your lab notebook** describe what your code actually did. If different from your expectations, explain. Share the results with your team.
14. How long does it take to draw a complete picture?
15. Explain your results to your team and discuss with them how you might speed thing up.

16. Save your code and upload it to your group's file space on Canvas. Use a new name so that prior versions of the code are not over-written.