HW5 - RGB Sensor

San Francisco State University

CSC 615 | Professor Robert Beirman

Group | **Robert RedBull Racing**

Zach Howe (923229694)

Yu-Ming Chen (*923313947*)

Aditya Sharma (917586584)

James Nguyen (922182661)

Github | *ZHowe1*

Table Of Contents

Description 3
Approach / What We Did 3
Issues and Resolutions 6
Analysis/How Library Works **7**
Photo of the completed circuit 8
Hardware Diagram 9
Screenshot of compilation 10
Screenshot of the execution of the program 11

Description

The goal is to build a library for the TCS34725 RGB sensor and use the sensor to detect colors.

Approach / What We Did

Here is our plan to complete this assignment and build a library for future projects:

Aditya Sharma

The first thing I needed to do was clearly understand how the TCS34725 RGB sensor's data translated into the actual colors humans perceive. Initially, this seemed straightforward—just read the sensor's red, green, blue, and clear values. However, upon closer inspection, I realized the raw RGB sensor readings alone wouldn't represent colors accurately to human eyes due to the nonlinear nature of our vision.

To figure out how to handle this, I researched examples of practical RGB sensor implementations. I came across an Adafruit TCS34725 library designed for Arduino, which utilized something called a "gamma table" for correcting sensor values. Initially, I didn't know exactly why gamma correction was necessary, so I dove deeper into gamma tables. I learned that human vision is nonlinear: we perceive brightness differently at varying intensities, and this difference is approximated well with a gamma correction curve. Thus, raw sensor values would not directly match perceived colors, necessitating gamma correction. At this point, I decided my first step should be to port the Arduino's gamma table concept into C code that we can use to get correct values. To achieve good modularity and ease of integration in our larger group project, I structured this logic into clearly separate header and source files—color_converter.h and color_converter.c. This organization allowed anyone in my group to easily integrate and reuse this module in different contexts, promoting clean code management.

Next, I defined a simple data structure, color_result, in the header file, encapsulating all essential information we would need: the hexadecimal representation of the color, an approximated human-readable color name, and a numeric confidence value indicating how accurately the color had been approximated. I deliberately included the confidence metric because I foresaw that sensor readings might fluctuate due to ambient lighting variations or shadows, providing a measure of approximation reliability that seemed valuable.

When I was defining this structure, I implemented the gamma correction logic in color_converter.c. Here, I referenced the Adafruit gamma table calculation as a reliable basis, carefully translating its logic from Arduino C++ into plain C suitable for the Pi environment. Specifically, I initialized a static lookup table (gamma_table) with 256 entries to efficiently map raw RGB sensor data to corrected, visually-accurate color values. My choice of static initialization made sure the gamma table would only be computed once, optimizing performance during repeated sensor readings needed for real-time responsiveness.

An important decision arose here regarding the "COMMON_ANODE" setting. The original Arduino library included logic for driving common anode LED arrays, which inverted the gamma values because lower numerical values corresponded to higher brightness on these types of LEDs. However, our TCS34725 sensor setup included an onboard white LED purely for illumination purposes rather than a controlled RGB LED requiring inversion. Understanding this distinction was critical—I recognized that we didn't need inverted gamma correction values. Therefore, I explicitly set COMMON_ANODE to 0 to ensure proper gamma correction tailored specifically to our sensor scenario. Without this careful consideration, colors would appear incorrectly reversed or distorted. After applying gamma correction, my next consideration was the normalization of RGB values. Through earlier experiments, I noticed sensor RGB values would vary significantly depending on ambient brightness. To compensate for this, I normalized RGB values by dividing each by the sensor's clear reading, which represents overall ambient light intensity. This normalization step was essential for consistent color detection across varying lighting conditions, ensuring our sensor would reliably detect colors regardless of external illumination changes. Once normalized and gamma corrected, the RGB values needed translation into a format easily understandable by humans. Thus, I formatted these corrected RGB values into a standard hexadecimal string, widely recognized as a convenient way to represent colors succinctly in digital contexts. Additionally, to satisfy our assignment requirements of providing a human-readable color approximation, I implemented a straightforward color matching algorithm.

I defined a set of common reference colors within a static array, such as red, green, blue, white, black, and so on. Initially, I considered several methods to match sensor readings against these reference colors. Ultimately, I decided to use Euclidean distance because it was intuitive and straightforward, measuring how "close" a sensed color was to each predefined reference. I computed the RGB distance from each reference color to our gamma corrected RGB sensor values, selecting the smallest distance to determine the most accurate color match. With the closest reference color identified, my final task was calculating a meaningful confidence metric.

Initially, I thought a simple binary match (exact color or not) might suffice, but I realized real-world sensor data would rarely match perfectly due to subtle lighting shifts and sensor noise. To account for this realism, I derived a confidence value scaled between 0 (no match) and 1 (perfect match), using the computed Euclidean distance normalized against the maximum possible RGB distance. Thus, colors extremely close to the reference would yield high confidence, while ambiguous readings due to shadows, lighting changes, or sensor anomalies would logically lower the confidence.

Lastly, I accounted for possible sensor inconsistencies related to "shadow registers." The TCS34725 provides redundancy checks via shadow registers to validate the sensor's internal consistency. I introduced a check for matching shadow registers—a mismatch would indicate potential errors in sensor readings, such as noise or hardware malfunction. Therefore, if the shadow registers didn't match, I deliberately set confidence to 0.

Zachary Howe

I did a lot of the library stuff and helped set up the foundation for the project. I did some of the research, finding the datasheet and looking into a bit of optics to help the sensor be more accurate. Via the gamma table. I made sure that the GPIO and I2C access was usable.

I planned with James to help set up the tcs controller. Including the defines and debugging. I also helped set up the architecture, setting up the include and src for the makefile. Having to research to make files in the process.

I'm currently working on building out the tcs_controller module for our project. I've already implemented the core of the RGB color sensing logic for the TCS34725 sensor using I²C. My focus has been on making the system modular and easy to use by exposing clean functions like rgb_init, sense_rate, and sense_color.

To do this, I wrote helper functions like rgb_write_16 and rgb_read_16, which abstract away the raw I²C communication and handle things like register OR-ing with 0x80 (as required by the datasheet). I keep these functions internal to the C file so the public interface stays clean.

In rgb_init, I configure the sensor by setting the target address and enabling power and ADC. In sense_rate, I convert human-readable millisecond values into the sensor's internal ATIME register format (which uses a 2.4ms step size). I also choose an appropriate gain value based on the sensing window to balance sensitivity and speed. The idea is to keep each step isolated — timing first, then gain, so it's easier to debug and modify later.

The sense_color function handles reading the 8-bit high/low pairs for each color channel (red, green, blue, and clear). It validates success at each step and builds an RGB_color struct with the values and a flag indicating if the reading was valid.

I'm also helping define the overall structure of the sensor driver, making sure it's reusable and fits cleanly into our Makefile-based build system. I worked with James to plan out the header and source separation, and we added organized #defines to make register and mode setup much more readable. Along the way, I researched the TCS34725 datasheet and reviewed integration time math and gain behavior, including some basic optics concepts to make sure we're interpreting values correctly.

My goal is to keep this modular enough that we could later add gamma correction or color calibration through a lookup table or linearization pass.

James Nguyen

I plan to implement communication with the TCS34725 color sensor using the I2C protocol on the Raspberry Pi.

For sensor initialization, I plan to write a function called rgb_init to handle the initial setup of the TCS34725. This will include setting the sensor's I2C address and enabling power and ADC functionality by writing to the ENABLE register using a helper write function. For sampling rate and gain, I plan to create a function named sense_rate that sets the integration time for the sensor based on a provided number of milliseconds. This time will be converted to the appropriate register value and written to the TIMER register. Based on the time value, I will also select and write a corresponding gain setting to the GAIN register to optimize sensor accuracy. To handle color data retrieval, I plan to read RGB values from the sensor. I plan to implement a function called sense_colo,r, and it will read from both low and high data registers for clear, red, green, and blue channels. If all the reads are successful, the data will be marked as valid in a struct named rgb_color, which I plan to define to hold the sensor data. Lastly, for the I2C Read/Write Abstraction, I plan to implement two helper functions: rgb_write_16 and rgb_read_16. These should abstract the details of I2C communication by formatting the register addresses correctly and sending and receiving data using the driver functions, which will make the higher-level sensor functions more straightforward to use.

By structuring the code this way, I aim to create a clean and reusable interface for working with the TCS34725 color sensor through I2C.

Yu-Ming Chen (Jim)

I did the diagrams and most of the formatting/cleanup work of the program. There was a lot of tangling within the codes, so I organized them and added descriptions to each function for better documentation.

I also handle most of the debugging. See problem and solution below.

Issues and Resolutions

Issue 1: Clear values are too low, causing our colors to all be read as black
Solution 1:
Because the common anode was set to the incorrect values, we were receiving flipped values and calculating incorrect color conversion. We were getting the correct values, but due to the encoder being flipped, we were calculating values incorrectly, and we noticed that it led to the opposite color being recognized.

Issue 2: The sensor is very sensitive to ambient lighting and people standing next to it casting shadows onto the object, which affects the accuracy of the RGB sensor
Solution 2:
We turned off the lights and stood back when running the code to reduce the ambient light and shadows cast by us

Issue 3: We were getting bad data from the sensor, which made us double-check multiple of our variables, but we were still getting inconsistent results.
Solution 3:
We swapped out our sensor with another sensor and began to get consistent results, which led to our realization that we had a bad sensor that was feeding us garbage data.

Analysis/How Library Works

Running The library:

The root folder has a makefile which builds the library. You can run make. This will build the library and a main executable for testing the file.

The executable main can be run normally like ./main within the root folder.

We also create .o files for each .c when compiled.

Testing the library yourself can be down by either editing the main or building a new file and targeting the library location. This can be done in Make by calling

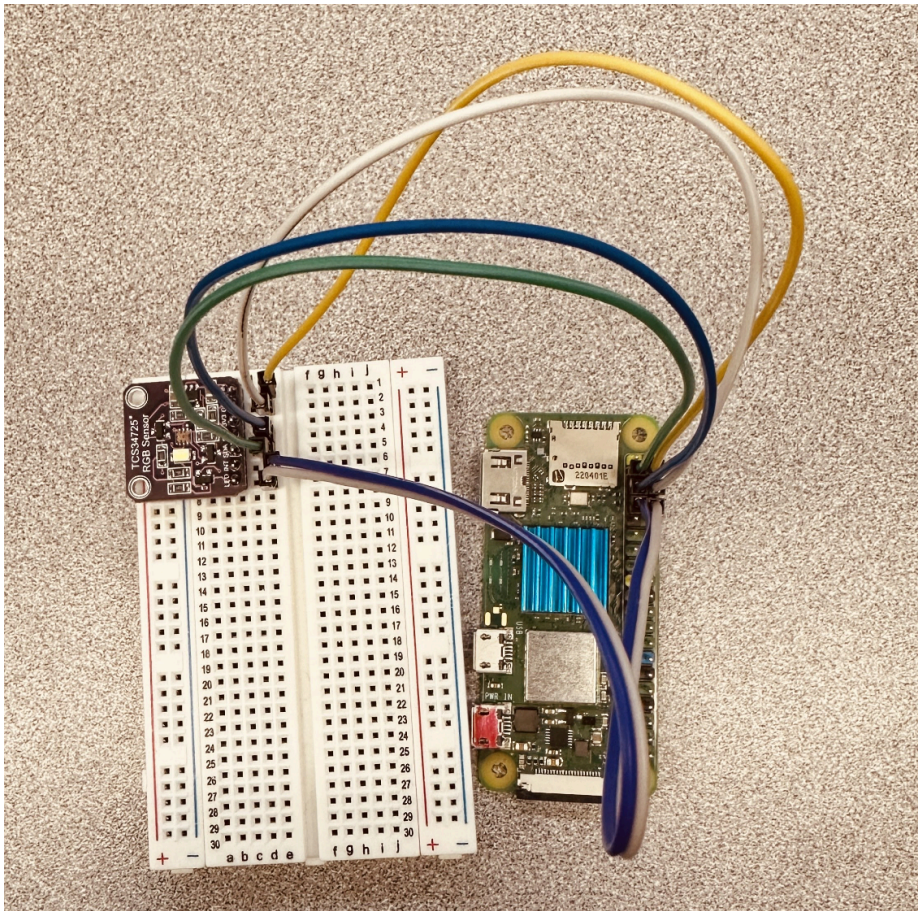
```
$(<name of your executable>): <your main file(s)> $(<path/to/library>)
    $(CC) $(CFLAGS) -o $@ $< -L. -lRGB_lib $(LDFLAGS)
```

The library still needs to be built, however.

The -L. Includes the current directory. The -lm allows us to link the c
math library to our build.

We are using a couple of flags, including werror-unused, because one of the variables was required by signal but unused in the final file.

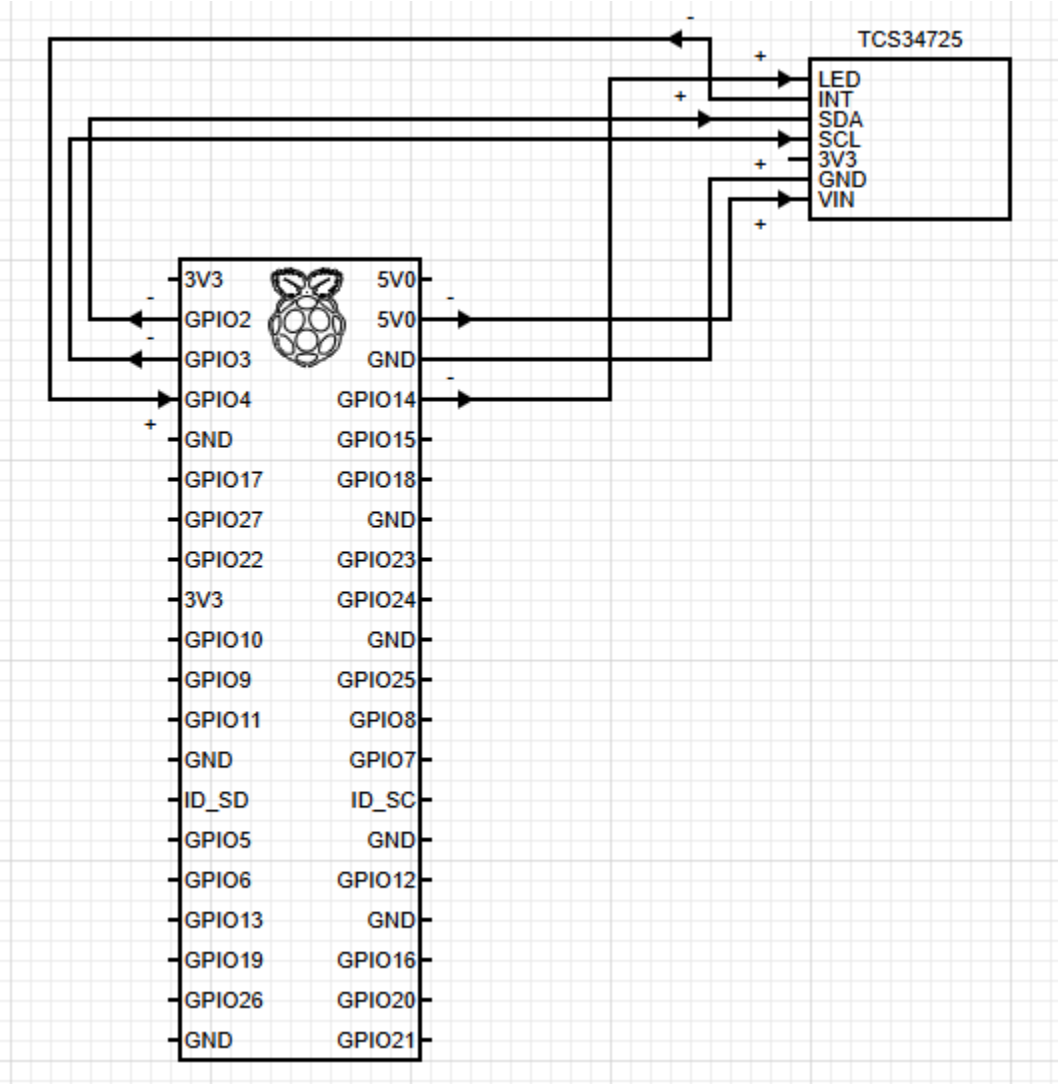
Photo of the completed circuit



Hardware Diagram

Tool used: Circuit Diagram www.circuit-diagram.org/

Author: Yu-Ming



Screenshot of compilation

```
pi38@raspberrypi38:~/code/assignment-5 $ make
gcc -I./include -g -Wno-unused-parameter -c src/gpio_library/core/pins.c -o src/
gpio_library/core/pins.o
gcc -I./include -g -Wno-unused-parameter -c src/gpio_library/core/timer.c -o src
/gpio_library/core/timer.o
gcc -I./include -g -Wno-unused-parameter -c src/gpio_library/core/i2c_access.c -
o src/gpio_library/core/i2c_access.o
gcc -I./include -g -Wno-unused-parameter -c src/gpio_library/TCS34725/tcs_contro
ller.c -o src/gpio_library/TCS34725/tcs_controller.o
gcc -I./include -g -Wno-unused-parameter -c src/gpio_library/TCS34725/color_conv
erter.c -o src/gpio_library/TCS34725/color_converter.o
ar rcs libRGB_lib.a src/gpio_library/core/pins.o src/gpio_library/core/timer.o s
rc/gpio_library/core/i2c_access.o src/gpio_library/TCS34725/tcs_controller.o src
/gpio_library/TCS34725/color_converter.o
gcc -I./include -g -Wno-unused-parameter -o main src/main.c -L. -lRGB_lib -L. -l
m
pi38@raspberrypi38:~/code/assignment-5 $
```

Screenshot of the execution of the program

```
pi38@raspberrypi38:~/code/assignment-5 $ make run
./main

 ** LED ON or OFF? (Y/N) **
y
** LED turned ON **
DEBUG: Writing ENABLE register with value: 0x03
i2c complete...
RGB values:
Red: 0
Green: 0
Blue: 0
Hex: #000000
Color Name: Black
Mix: 0, 0, 0
Clear value: 0, 0
Confidence: 100.00%

RGB values:
Red: 255
Green: 255
Blue: 35
Hex: #FFFF23
Color Name: Yellow
Mix: 54, 249, 11
Clear value: 15, 27
Confidence: 92.08%

RGB values:
Red: 255
Green: 14
Blue: 255
Hex: #FF0EFF
Color Name: Magenta
Mix: 249, 17, 78
Clear value: 6, 62
Confidence: 96.83%

RGB values:
Red: 25
Green: 123
Blue: 255
Hex: #197BFF
Color Name: Azure
Mix: 74, 153, 252
Clear value: 4, 213
Confidence: 94.27%

^C
Thanks for using the RGB sensor...

pi38@raspberrypi38:~/code/assignment-5 $
```