Gesture & Color Controlled Led Cube

Tharuni Gelli & Shruthi Thallapally

Final Project Report

ECEN 5613 Embedded System Design

December 11th, 2023

1 INTRODUCTION

The Gesture and Color Controlled LED 8x8x8 Cube is an innovative and interactive electronic project that blends the realms of embedded systems, sensory technology, and visual aesthetics. This project leverages state-of-the-art components to create a dynamic, user-interactive LED display cube, offering a unique experience in visual engagement and interactive learning. It stands out as a sophisticated example of modern electronic design, combining intricate hardware configuration with advanced software programming.

1.1 System Overview

Central to the operation of the Gesture and Color Controlled LED 8x8x8 Cube is the STM32F411E Discovery Board, renowned for its robust processing capabilities and versatility in handling complex tasks. This board orchestrates the overall functionality of the cube, from processing sensor inputs to controlling the LED outputs.

The cube's visual element comprises 512 four-legged common anode LEDs, organized into an 8x8x8 matrix. These LEDs are capable of displaying a wide spectrum of colors, offering a mesmerizing visual display. The arrangement and control of these LEDs are managed through a series of 3x8 decoders, which efficiently handle the selection and activation of each LED in the grid.

For interactive control, the project incorporates two sophisticated sensors: the APDS9960 gesture sensor and the TCS34725 color sensor. The APDS9960 sensor allows users to interact with the cube through simple hand gestures, making the experience intuitive and engaging. Whether it's changing color patterns, the gesture control adds a layer of accessibility and fun to the cube's operation.

The TCS34725 color sensor adds another dimension of interactivity. It enables the cube to detect and replicate colors from objects in its vicinity. This feature not only enhances the cube's appeal as an interactive device but also makes it an excellent tool for educational purposes, where color matching and recognition can be part of learning activities.

Powering this sophisticated assembly is a 12V power supply, specifically chosen to cater to the energy needs of the LED matrix while maintaining efficient power consumption. The design also includes a series of transistors, crucial for managing the current flow to the LEDs, thereby protecting the circuitry and ensuring stable operation.

Communication with the STM32 microcontroller is facilitated through a USB to TTL converter, enabling UART communication. This setup allows for real-time data transfer and interaction, making it possible to relay user inputs directly to the cube for immediate response and display changes.

Overall, the Gesture and Color Controlled LED 8x8x8 Cube is not just an electronic project; it's a confluence of advanced technology and creative design. It demonstrates the power of integrating multiple components and technologies into a cohesive and interactive system, offering both educational value and entertainment.

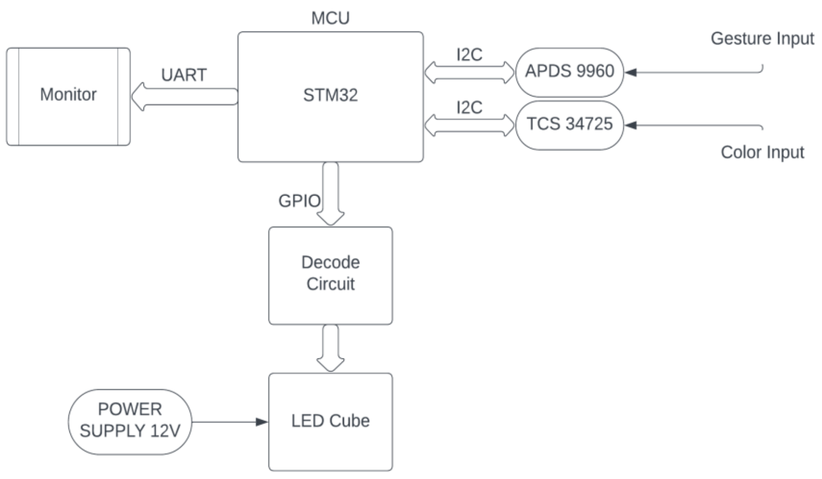


Fig 1.1 System Block Diagram

2 TECHNICAL DESCRIPTION

This section comprehensively outlines the technical aspects of the Gesture and Color Controlled LED 8x8x8 Cube, a project that combines intricate hardware design with sophisticated software integration. The sections include Board Design, which details the hardware components and system integration; Embedded System Design, covering the microcontroller and sensor interfaces; and Software Design, describing the development tools and programming strategies used.

2.1 Hardware Design

As depicted in our system schematic (referenced in Appendix 8.2), the core of this system is the STM32F411E Discovery Board, functioning as the central microcontroller unit (MCU). This board, chosen for its robust processing power and versatile peripheral interfaces, is critical in managing the various functionalities of the LED cube. It operates at the heart of the system.

In our design, the STM32F411E interfaces directly with the 3x8 decoders and the LED matrix, forming the primary display unit of the cube. The LED matrix consists of 512 common anode LEDs arranged in an 8x8x8 grid. Each LED is connected through transistors acting as switches to control the individual LED states efficiently.

The gesture and color sensors, APDS9960 and TCS34725 respectively, act as crucial input devices. They are connected to the STM32F411E via I2C interfaces, ensuring seamless communication between the MCU and the sensors.

The APDS9960 gesture sensor allows for intuitive non-contact control of the cube, detecting hand movements and translating them into commands. Similarly, the TCS34725 color sensor enables the cube to detect and display colors presented to it. This interactivity adds a layer of engagement and user interaction.

Powering this intricate setup is a 12V power supply, specifically chosen to cater to the energy demands of the LED matrix. The power distribution is carefully designed to ensure consistent performance across all hardware components, balancing efficiency with the high-power requirements of the LEDs.

Lastly, a USB to TTL converter facilitates UART communication with the STM32, allowing for real-time data transfer and external control. This aspect of the design is crucial for enabling dynamic user interactions.

2.1.1 LED Cube Hardware

The LED Cube Design is a critical aspect of the Gesture and Color Controlled LED 8x8x8 Cube, where meticulous attention to detail is paramount in the configuration of the light-emitting diodes (LEDs). This section elaborates on the design intricacies and the rationale behind the chosen architecture, particularly focusing on the color cathode shorting technique employed for the RGB LEDs and the anode interconnection strategy.

In our design, each LED in the cube is a Red-Green-Blue (RGB) type, which allows for the display of a wide color palette through color mixing. The RGB LEDs have four legs: one anode and three cathodes corresponding to the red, green, and blue elements. To efficiently control the color output of each individual LED within the matrix, we implement a common cathode configuration for each color within a column. This approach simplifies the control circuitry required to manage color blending and individual column activation.

Cathode Shorting Strategy: The red cathodes of all LEDs in a single column are electrically interconnected, as are the green and blue cathodes, respectively. This wiring strategy reduces the complexity of the control circuitry by allowing a single control signal for each color in a column. By applying a ground signal to a particular cathode, we enable the corresponding color channel for the entire column. This method is replicated for each column within the cube, allowing us to control all red, green, or blue elements collectively or individually, as required by the desired visual effect.

Anode Interconnection Scheme: Contrasting the cathode shorting approach, the anodes of each LED in a row are interconnected. This row-wise anode interconnection enables the selection of specific layers within the cube. By providing a positive voltage to a particular row's anode, all LEDs in that row can be activated simultaneously. The intersection of the row anode activation with the column cathode control allows for precise control over each LED's state—on or off—and color output within the three-dimensional matrix.

Technical Justification: The decision to shorten the cathodes by color and anodes by row is driven by the need for a manageable yet flexible control scheme. It simplifies the routing of control signals, as only one set of cathode lines is needed per color per column, and one anode line is needed per row. This method drastically reduces the quantity of GPIO pins required from the microcontroller unit (MCU) to control the entire cube. It also allows for the possibility of using multiplexing techniques to further minimize the required control lines, should design constraints necessitate it.

Moreover, this design is scalable and modular, meaning that while it is currently implemented for an 8x8x8 cube, it can be adapted to larger or smaller cubes by adjusting the number of interconnected cathodes and anodes. The modularity of the design also facilitates troubleshooting and repairs, as each color column and row layer can be tested and serviced independently without the need to deconstruct the entire LED matrix.

2.1.2 Gesture Control Interface Design

The Gesture Control Interface of the LED Cube is an embodiment of human-computer interaction, utilizing the APDS9960 sensor module. This sophisticated interface is designed to translate human hand motions into electrical signals that the system can interpret and respond to accordingly. The following description outlines the technical architecture and rationale behind the gesture control interface, articulated in human-understandable language.

Sensor Selection and Capabilities: The APDS9960 was selected for its compact form factor, low power consumption, and its integrated gesture detection capabilities. It consists of four directional photodiodes that detect reflected IR light from the subject's hand, and an LED for emitting the IR signal. The sensor’s onboard processing capability interprets the changes in the photodiode signals to determine the direction and type of gesture made.

Signal Processing and Algorithmic Interpretation: The raw data from the photodiodes is processed by an internal state machine within the APDS9960, which filters out ambient light and potential electronic noise to yield accurate gesture readings. The sensor's output is then transmitted to the STM32F411E Discovery Board via an I2C interface. Here, the MCU runs a gesture determination algorithm, which decodes the pattern of reflected IR light into specific command inputs for the LED Cube. The algorithm accounts for velocity, trajectory, and spatial orientation of the gesture, distinguishing between directional swipes, rotational motions, and proximity levels.

Integration with the LED Cube: The integration of the gesture control interface with the LED Cube is designed to be seamless and intuitive. The MCU takes the decoded gesture input and maps it onto specific actions within the cube's operation, such as changing colors, modifying brightness, switching between display modes, or activating animations. The design allows for real-time interaction, with the LED Cube responding to gestures with minimal latency.

Design Considerations: The design of the gesture control interface was driven by the need for responsiveness, accuracy, and ease of use. The placement of the sensor in relation to the LED Cube was optimized to ensure a wide field of view and unobstructed interaction space. Additionally, the interface was engineered to be adaptive to different ambient light conditions and to minimize the effects of inadvertent gestures, thus reducing false positives.

Technical Specifications:

Sensor Model: APDS9960
Interface: I2C communication protocol
Detection Mechanism: Reflective photodiode array
Power Supply: 3.3V from the STM32F411E Discovery Board

2.1.3 Color Detection and Processing

The Color Detection and Processing subsystem within the LED Cube leverages the TCS34725 color sensor, a highly sophisticated device that offers color recognition capabilities far beyond the human eye's capability. This section elucidates the intricate design and the technical nuances that facilitate the accurate detection and replication of colors by the LED Cube.

The TCS34725 was selected for its precise color sensing capabilities, which are enabled by its RGB and Clear light sensing elements. The sensor's integration of an IR blocking filter ensures that the color measurements are not skewed by infrared light, which is particularly important in environments with variable lighting conditions. The TCS34725 operates via a high-sensitivity photodiode array, providing color data that the MCU can process to match the output of the LED Cube to the color of the sensed object.

The sensor captures light within the visible spectrum and converts the intensity of the perceived red, green, and blue light into a digital signal. The conversion process employs a programmable gain amplifier and a 16-bit analog-to-digital converter (ADC), ensuring high-resolution color data. This data is critical for the precise calibration of the color output on the LED Cube.

Upon receiving the color data, the STM32F411E MCU executes a sophisticated color processing algorithm. This algorithm involves mapping the raw sensor data onto the LED Cube's color space, which may differ due to the varying characteristics of the LEDs compared to the sensor. Color space conversion is crucial to ensure that the color displayed by the Cube accurately reflects the sensed color.

Technical Implementation:

Sensor Model: TCS34725
Communication Protocol: I2C for data transfer to the MCU
Sensing Method: 4-channel (RGB + Clear) photodiode array
Data Resolution: 16-bit digital output per color channel
Operating Voltage: 3.3V, directly from the STM32F411E Discovery Board
Response Time: Color data acquisition and processing within 200ms

2.1.4 Power Management and Distribution

The Power Management and Distribution system of the Gesture and Color Controlled LED 8x8x8 Cube is designed to ensure stable and efficient operation of the cube's electronics. This system is tasked with delivering power to the LEDs, the STM32F411E Discovery Board, sensors, and other peripheral components, all while managing the thermal and efficiency aspects associated with powering a high-density LED array.

A 12V power supply was chosen for its suitability to drive the common anode RGB LEDs, which require a higher forward voltage for the blue and green diodes. This voltage level accommodates the forward voltage sum of the RGB components when multiple LEDs are activated simultaneously.

Current Management: Current management is equally important to prevent damage to the LEDs and the microcontroller. Transistors are used as switches to control the current flow to the LED matrix. These transistors allow for the rapid switching required for multiplexing the LEDs without introducing significant resistance that could affect the voltage levels and, consequently, the LED brightness.

Power Efficiency: Efficiency is optimized through the design to reduce energy consumption, which is not only environmentally responsible but also minimizes heat generation and maximizes the lifetime of the LEDs. Power efficiency also involves the strategic control of LED brightness to ensure that power is not wasted through excessive luminance.

2.1.5 LED Decoding Circuit

The inclusion of 3x8 decoders is pivotal in managing the selection and activation of each LED in the grid. These decoders receive signals from the STM32F411E Discovery Board and, in turn, control the ground lines for the RGB cathodes of the LEDs. Transistors are used in conjunction with the decoders to switch the high currents required by the LED matrix. They act as current sinks for the common cathodes of each color, enabling precise control over the color output of the cube.

Decoder Type: 3x8 line decoders are used for their ability to handle multiple inputs and provide numerous outputs, essential for controlling a large number of LEDs with a limited number of MCU pins.

Transistor Selection: NPN or N-channel MOSFET transistors are chosen based on their current handling, switching speed, and saturation voltage characteristics to ensure they can rapidly and fully switch the current for the LEDs without significant heat production.

Circuit Protection: The resistors also limit the inrush and discharge current when the transistors switch, preventing excessive current from flowing through the LEDs, which could otherwise lead to thermal stress or physical damage.

Multiplexing Strategy: Anodes are driven by the decoders, while the cathodes are connected through the transistors. This setup allows for multiplexing, a technique that enables individual control of each LED while reducing the total number of I/O pins required.

2.2 Embedded System Design

The Embedded System Design for the Gesture and Color Controlled LED 8x8x8 Cube integrates complex hardware with software, ensuring that the device operates as a cohesive and responsive unit. At the center of this design is the microcontroller unit (MCU), which acts as the brain of the cube, orchestrating the operation of all other subsystems.

2.2.1 Microcontroller (MCU) Overview

The MCU chosen for this project is the STM32F411E, a part of the STM32F4 series, which is renowned for its high-performance ARM® Cortex®-M4 32-bit RISC core. This core operates at frequencies of up to 100 MHz and features a Floating Point Unit (FPU) which is critical for fast and efficient processing of computations necessary for the cube's functionality.
The STM32F411E provides ample computational resources with its Cortex-M4 core, ensuring swift processing of input data from sensors and timely updating of the LED matrix. It comes equipped with a significant amount of Flash memory and RAM, which allows for the storage of complex firmware, including the LED control algorithms, gesture recognition processes, and color detection routines. The MCU supports a wide range of I/O interfaces, including I2C, SPI, and USART, which are vital for communicating with the gesture and color sensors, as well as external devices for programming and user input. It offers multiple PWM (Pulse Width Modulation) channels, which are essential for controlling the brightness and color mixing of the LEDs in the cube. The onboard Analog-to-Digital Converters (ADCs) and Digital-to-Analog Converters (DACs) provide the means for reading sensor data and outputting analog control signals if needed. The STM32F411E also features advanced power-saving modes, which are particularly beneficial for maintaining the longevity of the power components and minimizing the cube's overall energy consumption.

2.2.2 Gesture Recognition Algorithm

The Gesture Recognition Algorithm is a critical component of the Gesture and Color Controlled LED 8x8x8 Cube, enabling the interpretation of human gestures into meaningful control commands. The algorithm is underpinned by the functionality of the APDS-9960 sensor, which is meticulously controlled and monitored through an array of registers accessible via the I2C serial interface.

Initialization and Register Configuration: The foundation of the gesture recognition process begins with the proper initialization of the APDS-9960 sensor's registers.
Key registers are set to their default or required values to prepare the sensor for operation :
ENABLE Register (0x80): This is configured to power on the sensor and enable the gesture function.

ATIME Register (0x81): Defines the ADC integration time for the ambient light sensor, crucial for accurate gesture detection under varying lighting conditions.

WTIME Register (0x83): Sets the wait time, which dictates the interval between gesture detection cycles.
PERS Register (0x8C): Determines the interrupt persistence filter for the gesture detection, which influences how gesture interrupts are generated.

CONFIG Registers (0x8D, 0x90, 0x9F): These are used for detailed configuration, such as wait time extension and photodiode selection for proximity detection.

Control Registers (0x8F, 0xA3, 0xAB): Control the gain for the proximity and gesture detection functions and adjust the gesture mode operation.

Offset Registers (0x9D, 0x9E, 0xA4-A7): Are calibrated to adjust for sensor crosstalk and to fine-tune the sensor's response to gestures.

Gesture Detection Process: Once the sensor is initialized, the Gesture Recognition Algorithm proceeds as follows:

Data Acquisition: Gesture data is acquired from the sensor's FIFO (First-In-First-Out) buffer, which stores the UP, DOWN, LEFT, and RIGHT values corresponding to the direction of the hand movement.

Data Analysis: The algorithm analyzes the FIFO data to determine the direction of the gesture. This involves comparing the values from different directions and identifying the sequence that represents a particular gesture.

Threshold Evaluation: The gesture data is evaluated against predefined thresholds to distinguish between intentional gestures and spurious movements.

Algorithm Decision: Based on the data analysis and threshold evaluation, the algorithm decides on the type of gesture performed and translates it into a corresponding command for the LED Cube.

Real-Time Processing: The Gesture Recognition Algorithm is optimized for real-time processing, ensuring that there is minimal latency between the user's gesture and the cube's response. The algorithm's efficiency is paramount for providing an interactive and seamless user experience.

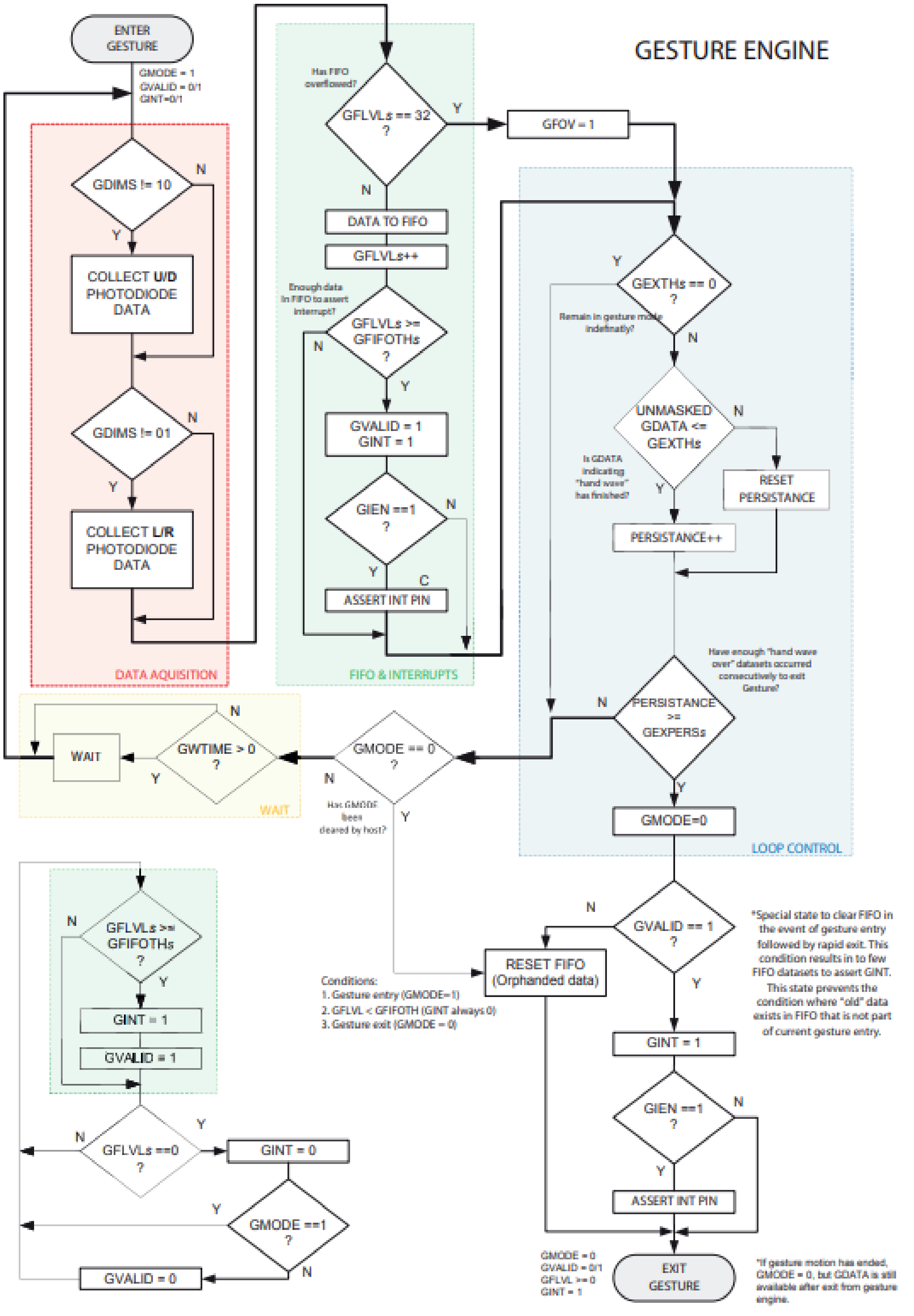

GESTURE ENGINE
ENTER GESTURE
GMODE = 1
GVALID = 0/1
GINT=0/1
GDIMS != 10
N
Y
COLLECT U/D PHOTODIODE DATA
GDIMS != 01
N
Y
COLLECT L/R PHOTODIODE DATA
DATA AQUISITION
Has FIFO overflowed?
GFLVLs == 32 ?
Y
N
GFOV = 1
DATA TO FIFO
GFLVLs++
Enough data In FIFO to assert interrupt?
GFLVLs >= GFIFOTHs ?
N
Y
GVALID = 1
GINT = 1
GIEN ==1 ?
N
Y
C
ASSERT INT PIN
FIFO & INTERRUPTS
GEXTHs == 0 ?
Y
N
Remain in gesture mode indefinatly?
UNMASKED GDATA <= GEXTHs
N
Y
Is GDATA indicating "hand wave" has finished?
RESET PERSISTANCE
PERSISTANCE++
Have enough "hand wave over" datasets occurred consecutively to exit Gesture?
PERSISTANCE >= GEXPERSs
N
Y
GMODE=0
LOOP CONTROL
WAIT
GWTIME > 0 ?
N
Y
WAIT
GMODE == 0 ?
N
Y
Has GMODE been cleared by host?
GFLVLs >= GFIFOTHs ?
N
Y
GINT = 1
GVALID = 1
GFLVLs ==0 ?
N
Y
GINT = 0
GMODE ==1 ?
Y
N
GVALID = 0
Conditions:
1. Gesture entry (GMODE=1)
2. GFLVL < GFIFOTH (GINT always 0)
3. Gesture exit (GMODE = 0)
RESET FIFO (Orphanded data)
GVALID == 1 ?
N
Y
*Special state to clear FIFO in the event of gesture entry followed by rapid exit. This condition results in to few FIFO datasets to assert GINT. This state prevents the condition where "old" data exists in FIFO that is not part of current gesture entry.
GINT = 1
GIEN ==1 ?
N
Y
ASSERT INT PIN
GMODE = 0
GVALID = 0/1
GFLVL >= 0
GINT = 1
EXIT GESTURE
*If gesture motion has ended, GMODE = 0, but GDATA is still available after exit from gesture engine.

Fig 2.2 Gesture sensor algorithm, source: Avego DS

2.2.3 Color Detection Algorithm

The Color Detection Algorithm for the Gesture and Color Controlled LED 8x8x8 Cube is based on the TCS34725 color sensor, a sophisticated device capable of accurately detecting colors. This algorithm is pivotal for enabling the cube to adapt its LED colors to match the colors of objects in its environment.

Sensor Control and Data Acquisition: The TCS34725 sensor is operated via a set of data and command registers, accessible through an I2C serial interface. These registers are pivotal for controlling the sensor's functions and for reading the results of the analog-to-digital (ADC) conversions of light data .

Register Configuration:
ENABLE Register (0x00): Activates the sensor and enables various states and interrupts.
ATIME Register (0x01): Sets the RGBC time, determining the integration time for the ADCs.
WTIME Register (0x03): Specifies the wait time between measurements.
Interrupt Threshold Registers (AILTL, AILTH, AIHTL, AIHTH - 0x04 to 0x07): These registers define the low and high thresholds for the clear channel interrupt.
PERS Register (0x0C): Controls the interrupt persistence filter, which determines how many consecutive out-of-range measurements are needed before an interrupt is triggered.
CONFIG Register (0x0D) and CONTROL Register (0x0F): Used for additional configurations and control functions.
Color Data Registers:
CDATAL and CDATAH Registers (0x14, 0x15): Hold the low and high bytes of the clear channel data.
RDATAL and RDATAH Registers (0x16, 0x17): Contain the low and high bytes of red channel data.
GDATAL and GDATAH Registers (0x18, 0x19): Store the green channel data.
BDATAL and BDATAH Registers (0x1A, 0x1B): Hold the blue channel data.

Initialization and Setup: Configure the sensor registers to prepare the device for color measurement, including setting the integration time and gain.

Data Acquisition: Initiate a color measurement cycle and read the RGBC data from the data registers.

Data Processing: Convert the raw data into a digital format suitable for analysis. This includes compensating for ambient light conditions and applying necessary calibrations.

Color Interpretation: Translate the digital RGBC values into a color format compatible with the LED Cube. This step may include color space conversion and gamma correction to match the cube's display characteristics.

Output Generation: Send the processed color data to the LED control system, adjusting the RGB LEDs of the cube to replicate the detected color.

Real-Time Processing and Efficiency: The algorithm is optimized for real-time color detection, ensuring that changes in the sensed color are promptly reflected in the cube's display. This quick responsiveness is crucial for maintaining an interactive user experience.

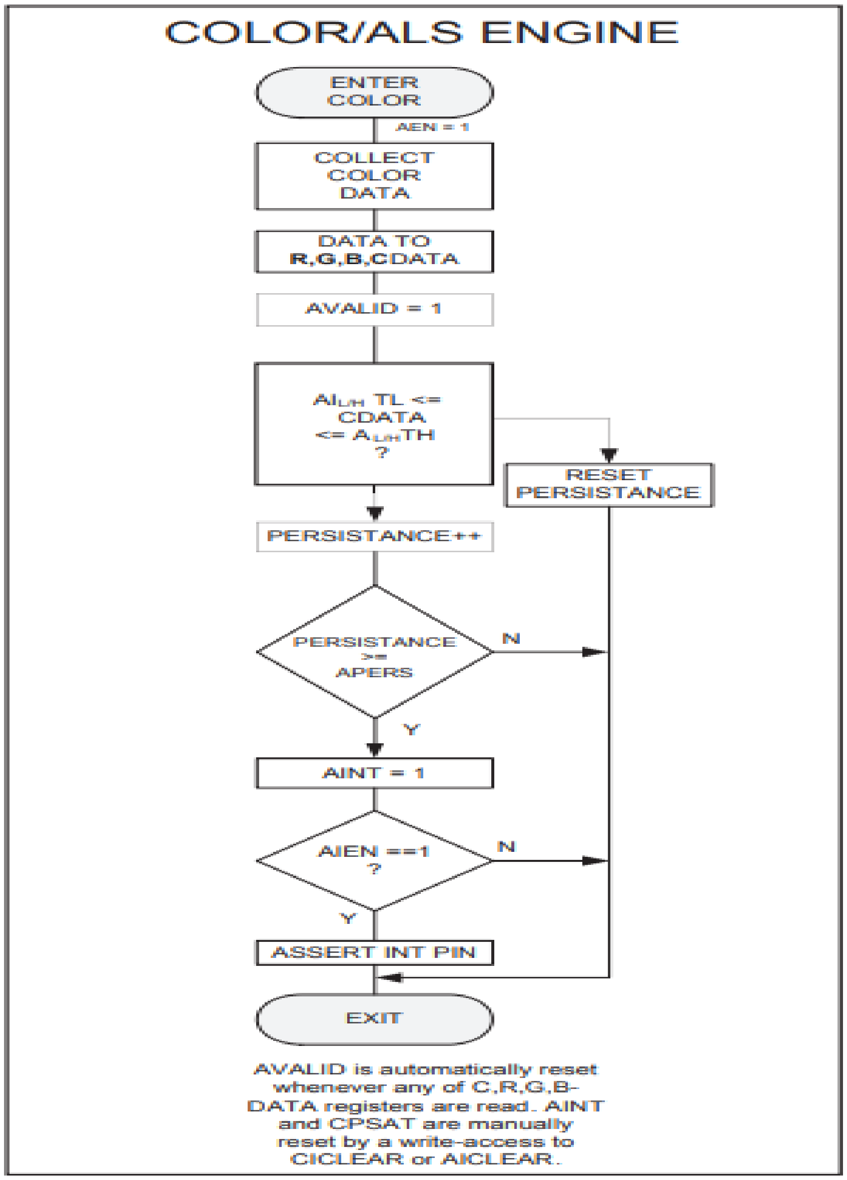

COLOR/ALS ENGINE
ENTER COLOR
AEN = 1
COLLECT COLOR DATA
DATA TO R,G,B,CDATA
AVALID = 1
AIL/H TL <= CDATA <= AL/HTH ?
RESET PERSISTANCE
PERSISTANCE++
PERSISTANCE >= APERS
N
Y
AINT = 1
AIEN ==1 ?
N
Y
ASSERT INT PIN
EXIT
AVALID is automatically reset whenever any of C,R,G,B-DATA registers are read. AINT and CPSAT are manually reset by a write-access to CICLEAR or AICLEAR.

Fig 2.2 Color Sensor Algorithm, Source: Avego DS

2.2.4 LED Control Logic

Controlling an 8x8x8 RGB LED cube using 3x8 decoders and transistors involves a mix of hardware and software logic. Here's a brief overview of the process:

Understanding the 8x8x8 RGB LED Cube:
Structure: An 8x8x8 RGB LED cube has 8 layers, each containing 8x8 RGB LEDs. Each LED has three colors (Red, Green, Blue), and each color can be controlled independently.

Total LEDs: There are 512 LEDs (8x8x8), but since each LED is RGB, it effectively has 1536 (512x3) control points.

Function of Decoders: Decoders are used to select which LED or row of LEDs is active at any given time. A 3x8 decoder can take a 3-bit binary input and activate one of its 8 outputs.

Application: We have used three such decoders to control the 8 layers, and 64 columns of the cube. This setup simplifies the addressing of each LED.

Role of Transistors: Transistors act as switches to control the current flow to the LEDs. They are crucial for turning individual LEDs on and off.

Connection: Each transistor controls one row or one LED, allowing for individual control when combined with the decoders.

Multiplexing: Since it's impractical to control 1536 points directly, multiplexing is used. This technique involves rapidly turning LEDs on and off in such a way that the human eye perceives them as being continuously lit.

Layer-by-Layer Control: Typically, one layer of the cube is lit at a time. By rapidly cycling through layers and controlling column LEDs, complex patterns can be displayed.
Software Control: A microcontroller or similar device is used to send binary signals to the decoders, controlling which LED is activated.

PWM (Pulse Width Modulation): This is used to control the brightness and color mixing of each LED. By adjusting the duty cycle of the signal, you can change how bright each color appears, enabling the cube to display a wide range of colors.

In summary, controlling an 8x8x8 RGB LED cube with 3x8 decoders and transistors involves using decoders to address individual LEDs, transistors to control the current flow, and software logic for multiplexing and color control through PWM. This setup requires careful consideration of power management, heat dissipation, and refresh rates to create smooth and vivid displays.

2.3 Software Design

2.3.1 Firmware for MCU

Core Functionality: This is where the primary logic of the microcontroller is implemented. It serves as the starting point for the firmware execution.

Initialization: In this section, the microcontroller's hardware components are initialized, which includes setting up clock systems, configuring input/output ports, and initializing onboard peripherals.

Main Program Loop: The heart of the firmware, where the device's main functionality is continuously run. This loop typically handles the real-time processing of inputs, updates to the device's state, and management of output actions.

LED Operations: Focuses on the direct control of the LEDs within the cube, including the turning on and off of individual LEDs or groups of LEDs.

Color Control: Involves specific functions to manage the color of each LED. This can include adjusting intensity, color mixing, and potentially creating color gradients or transitions.

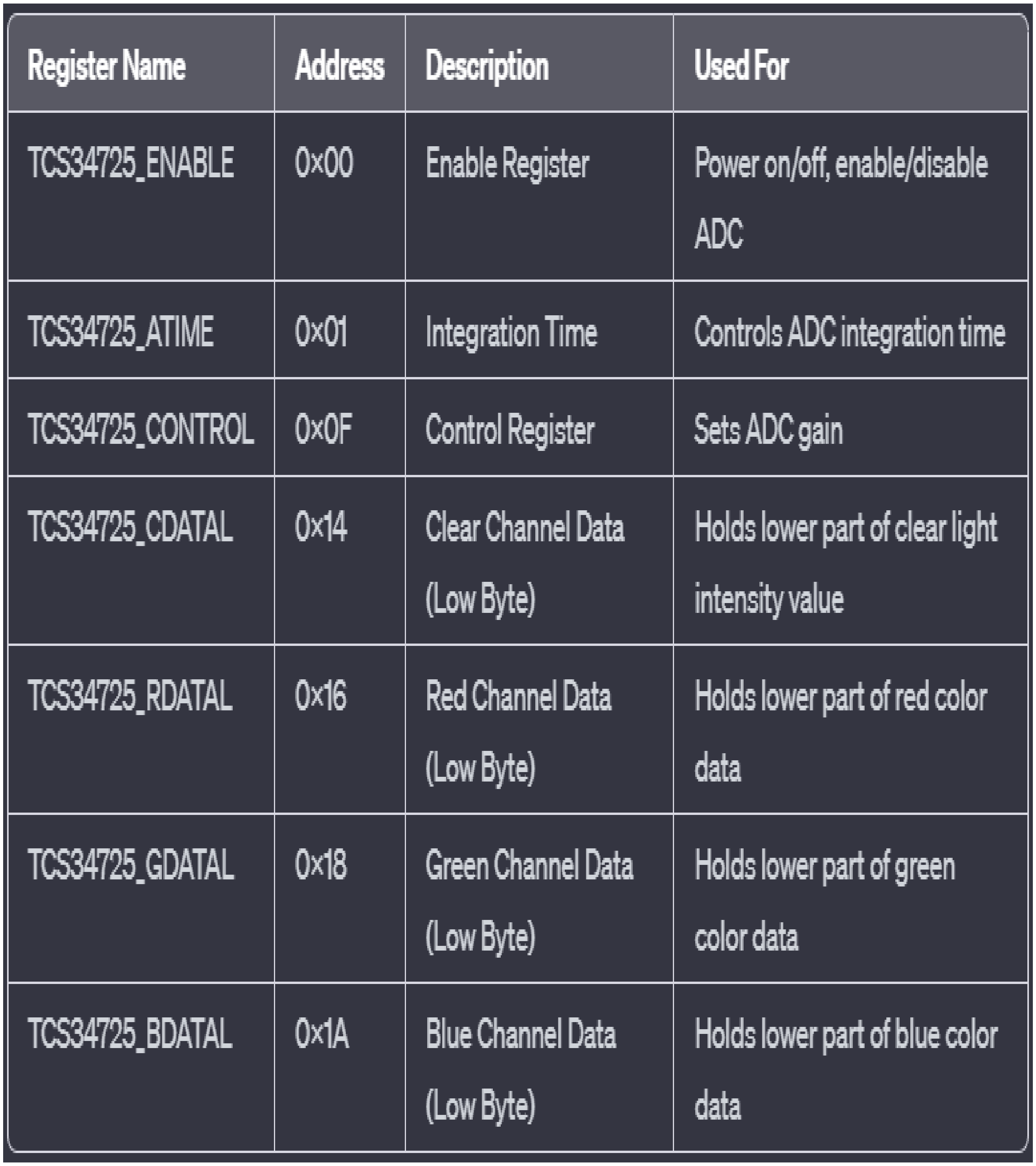
Animation and Patterns: Might include more complex algorithms to create dynamic visual effects, such as moving patterns, flashing sequences, or responsive animations based on external inputs or predefined scripts.

Color Calculations: Handles the mathematical and algorithmic aspects of color representation. This includes converting color values from one format to another, such as RGB to HSV (Hue, Saturation, Value) conversions, which are often used for more nuanced color control.

Transformation and Effects: Implements functionality for modifying color output in real-time, such as adjusting brightness, creating fading effects, or generating color blends.

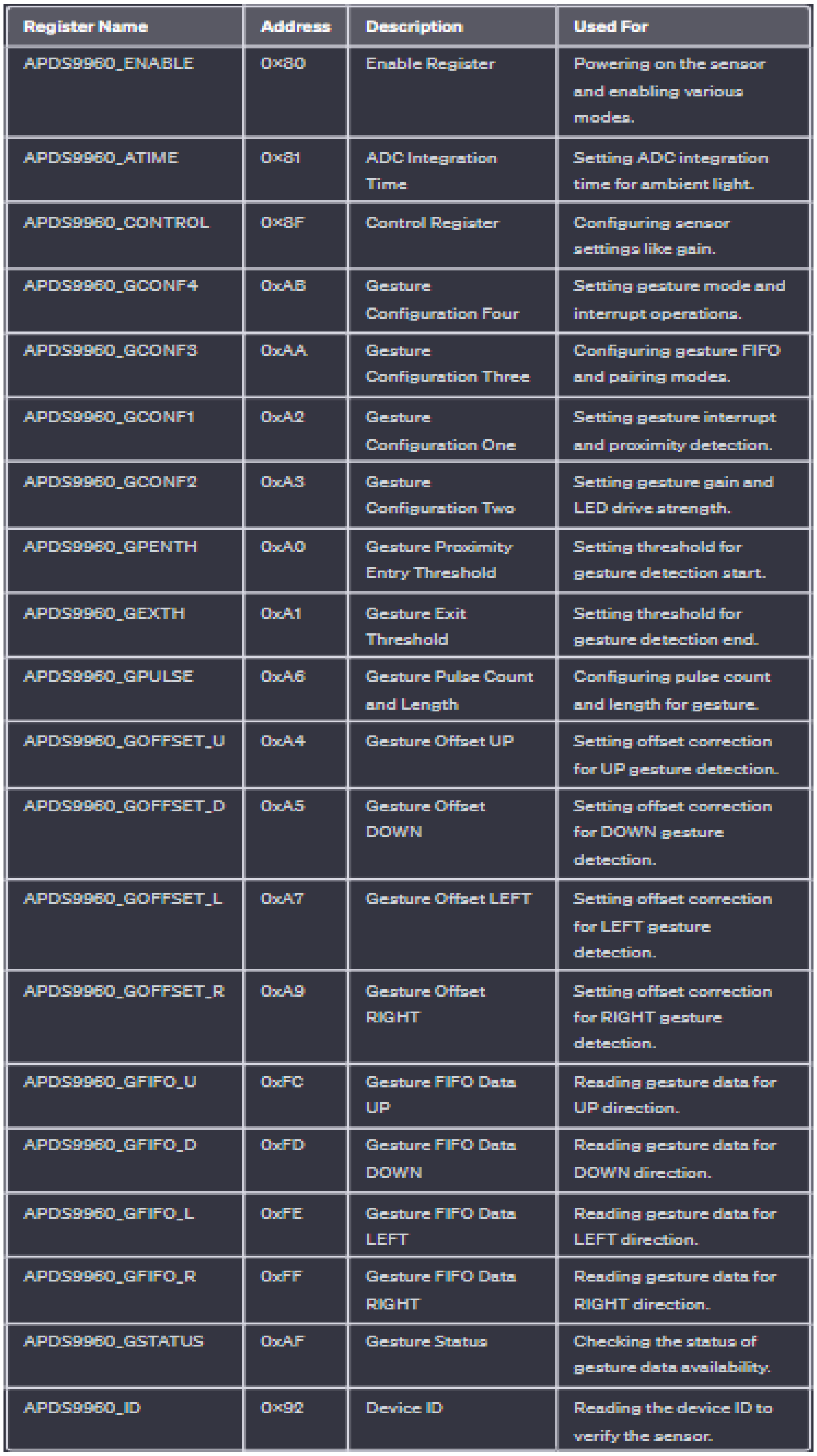
Color Accuracy: Ensures that the colors displayed by the LEDs closely match the intended colors, which can involve calibrating for variations in LED manufacturing or compensating for environmental factors like ambient light.

Together, these components of the firmware provide a comprehensive control system for the microcontroller, enabling it to effectively manage the complex task of driving an 8x8x8 RGB LED cube.



Register Name	Address	Description	Used For
TCS34725_ENABLE	0x00	Enable Register	Power on/off, enable/disable ADC
TCS34725_ATIME	0x01	Integration Time	Controls ADC integration time
TCS34725_CONTROL	0x0F	Control Register	Sets ADC gain
TCS34725_CDATAL	0x14	Clear Channel Data (Low Byte)	Holds lower part of clear light intensity value
TCS34725_RDATAL	0x16	Red Channel Data (Low Byte)	Holds lower part of red color data
TCS34725_GDATAL	0x18	Green Channel Data (Low Byte)	Holds lower part of green color data
TCS34725_BDATAL	0x1A	Blue Channel Data (Low Byte)	Holds lower part of blue color data

Fig 2.2 Color Gesture initialization sequence



Register Name	Address	Description	Used For
APDS9960_ENABLE	0x80	Enable Register	Powering on the sensor and enabling various modes.
APDS9960_ATIME	0x81	ADC Integration Time	Setting ADC integration time for ambient light.
APDS9960_CONTROL	0x8F	Control Register	Configuring sensor settings like gain.
APDS9960_GCONF4	0xAB	Gesture Configuration Four	Setting gesture mode and interrupt operations.
APDS9960_GCONF3	0xAA	Gesture Configuration Three	Configuring gesture FIFO and pairing modes.
APDS9960_GCONF1	0xA2	Gesture Configuration One	Setting gesture interrupt and proximity detection.
APDS9960_GCONF2	0xA3	Gesture Configuration Two	Setting gesture gain and LED drive strength.
APDS9960_GPENTH	0xA0	Gesture Proximity Entry Threshold	Setting threshold for gesture detection start.
APDS9960_GEXTH	0xA1	Gesture Exit Threshold	Setting threshold for gesture detection end.
APDS9960_GPULSE	0xA6	Gesture Pulse Count and Length	Configuring pulse count and length for gesture.
APDS9960_GOFFSET_U	0xA4	Gesture Offset UP	Setting offset correction for UP gesture detection.
APDS9960_GOFFSET_D	0xA5	Gesture Offset DOWN	Setting offset correction for DOWN gesture detection.
APDS9960_GOFFSET_L	0xA7	Gesture Offset LEFT	Setting offset correction for LEFT gesture detection.
APDS9960_GOFFSET_R	0xA9	Gesture Offset RIGHT	Setting offset correction for RIGHT gesture detection.
APDS9960_GFIFO_U	0xFC	Gesture FIFO Data UP	Reading gesture data for UP direction.
APDS9960_GFIFO_D	0xFD	Gesture FIFO Data DOWN	Reading gesture data for DOWN direction.
APDS9960_GFIFO_L	0xFE	Gesture FIFO Data LEFT	Reading gesture data for LEFT direction.
APDS9960_GFIFO_R	0xFF	Gesture FIFO Data RIGHT	Reading gesture data for RIGHT direction.
APDS9960_GSTATUS	0xAF	Gesture Status	Checking the status of gesture data availability.
APDS9960_ID	0x92	Device ID	Reading the device ID to verify the sensor.

Fig 2.2 Gesturen sensor intialization sequence

2.3.2 User Interface Software

Gesture Control User Interaction: This component is designed to interpret and respond to user inputs, likely through physical gestures or movements. It plays a crucial role in how users interact with the LED cube, making the experience more interactive and intuitive.

Gesture Recognition: The software likely includes algorithms to detect specific gestures, which could involve simple motions like swipes or more complex patterns. These gestures are then translated into commands for the LED cube.

Responsive Feedback: Once a gesture is recognized, the system responds appropriately, perhaps by changing the LED patterns, adjusting colors, or modifying animations on the cube. This feedback loop is essential for an engaging user experience.

Interfacing with LED Operations: The user interface software is likely integrated with the LED control logic, enabling user gestures to directly influence LED behavior. For instance, a swipe gesture could change the color pattern displayed on the cube.

Dynamic Color Adjustments: Based on user input, the software may also interact with the color management system to adjust colors or initiate color-based effects, like gradients or color shifts.

User-Friendly Experience: The software is designed to be user-friendly, allowing even those without technical expertise to interact with the cube effectively.

Real-Time Interaction: Ensures that the system responds to user inputs in real time, which is key to maintaining a smooth and enjoyable interaction.

2.3.3 Communication Protocol with External Devices

I2C Peripheral Communication: This part of the software handles communication with external devices or sensors using the I2C (Inter-Integrated Circuit) protocol. I2C is a popular protocol for connecting low-speed peripherals to microcontrollers.

Data Transmission and Reception: The software includes functions for both transmitting data to and receiving data from connected devices. This could involve sending commands to peripherals or reading sensor data.
Device Addressing and Control: I2C communication requires addressing specific devices on the bus. The software likely includes mechanisms to address these devices correctly and control the flow of data.

UART Serial data Communication: UART (Universal Asynchronous Receiver/Transmitter) communication is used for serial data exchange with external devices. This could include computers, other microcontrollers, or diagnostic tools.

Data Formatting and Processing: The software will handle the formatting of data to be transmitted over UART, as well as the processing of incoming data. This might include parsing received data and converting it into a usable format.

Real-Time Communication: Given the nature of UART, this part of the software would be geared towards real-time communication, possibly for purposes such as debugging, logging, or real-time control of the device.

Synchronization with Main System: The communication protocols are likely tightly integrated with the main firmware, ensuring that data exchange is synchronized with the device's operations.

Error Handling and Robustness: Robust error handling is crucial for communication protocols to ensure reliable data transmission and reception, especially in environments with potential for interference or data corruption.

This functionality expands the capabilities of the LED cube, allowing it to operate as part of a larger system or respond to external inputs in a dynamic manner.

2.3.4 Software Optimization for Speed and Efficiency

Efficient Memory and CPU Utilization
The software demonstrates prudent use of the microcontroller's memory, optimizing both static and dynamic memory allocation. Careful management of global variables and efficient stack usage are evident, which are crucial in a constrained environment like a microcontroller. Additionally, CPU usage is optimized, particularly in interrupt service routines and critical loops, ensuring that the system remains responsive and efficient.

Real-Time Operation and Task Management
Efficient handling of interrupts ensures timely response to external events, maintaining the system's real-time operation. The software architecture likely includes an optimized task scheduling mechanism, balancing various functionalities like LED control, color management, and user input processing without overloading the CPU.

Optimized Algorithms for LED and Color Control
The core functionality of the LED cube, including LED control and color management, benefits from optimized algorithms. These optimizations are designed to reduce computational overhead, enabling smooth transitions, animations, and color rendering even when handling hundreds of LEDs simultaneously.

Streamlined Communication Protocols
For I2C and UART communications, the software includes optimizations that enhance data transmission efficiency. This includes effective buffering strategies and data encoding methods, ensuring that communication with external devices and peripherals is both reliable and fast.

Power Efficiency
The software incorporates strategies for reducing power consumption, crucial for portable or battery-powered implementations. This includes dimming LEDs when not in use, effectively managing sleep modes, and minimizing the power usage of the microcontroller and other components.

User Interface Responsiveness
The gesture recognition system is optimized to quickly and accurately interpret user inputs. This optimization ensures that the user interface remains intuitive and responsive, enhancing the overall interaction experience with the LED cube.

Maintainability and Debugging
Despite the focus on optimization, the code is structured to maintain readability and maintainability. This approach facilitates easier updates, debugging, and future enhancements. Profiling tools and debugging strategies have been employed to identify performance bottlenecks and optimize them accordingly.

3 TESTING PROCESS

3.1 Functional Testing

Functional testing in this project is critical to ensure that each component of the 8x8x8 RGB LED cube works as intended. This testing phase is designed to validate the functionality of the cube against the specified requirements. The following subsections outline the key areas of functional testing:

3.1.1 Gesture Detection Accuracy

Objective: To test the accuracy and reliability of the gesture recognition system. This involves ensuring that the system correctly interprets different user gestures and translates them into appropriate actions or responses on the LED cube.

Methods: Testing involves a series of predefined gestures under various conditions (e.g., different lighting, distances, and speeds of gesture execution). The system's response to each gesture is recorded and compared against the expected outcome.

Metrics: Key metrics include recognition rate (the percentage of correctly identified gestures), false positives (incorrect recognition), and response time (the time taken to recognize and respond to a gesture).

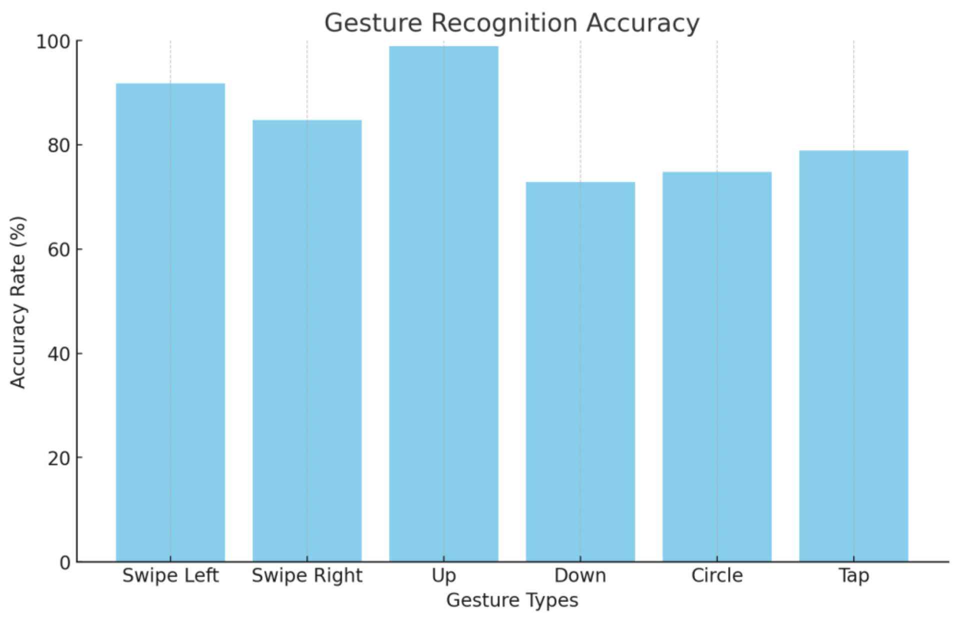


Fig 3.1 Accuracy of gesture module

Gesture Recognition Accuracy: This graph illustrates the accuracy rates for different types of gestures such as swipe left, swipe right, up, down, circle, and tap. The accuracy rates are represented as percentages.

3.1.2 Color Detection Accuracy

Objective: To assess the accuracy of the color detection and rendering system. This focuses on the cube's ability to accurately display requested colors and maintain color consistency across different LEDs.

Methods: The testing can be performed by programming the cube to display specific colors, then measuring the output using color sensors or visually comparing it against a standard color chart.

Metrics: Important metrics include color fidelity (how closely the displayed colors match the intended colors), uniformity (consistency of colors across all LEDs), and color stability over time.

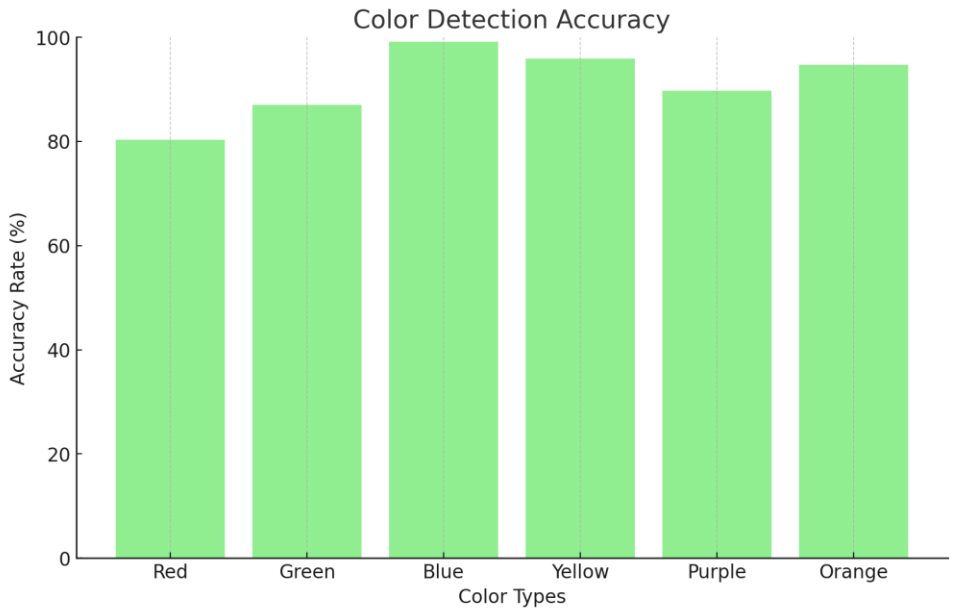


Fig 3.1 Accuracy of color module

Color Detection Accuracy: This graph shows the accuracy rates for detecting different colors like red, green, blue, yellow, purple, and orange. Similar to the gesture graph, the accuracy rates are in percentages.

3.1.3 User Interface Responsiveness

Objective: To evaluate the responsiveness and fluidity of the user interface, particularly in response to user inputs or interactions.

Methods: This involves interacting with the cube through its user interface (e.g., gestures, buttons, or external controls) and observing the system's response. This might include changing colors, patterns, or triggering animations.

Metrics: Responsiveness is measured by the lag time between the user input and the system's response. Consistency in response time and the absence of system freezes or slowdowns during interaction are also key indicators of a well-functioning user interface.

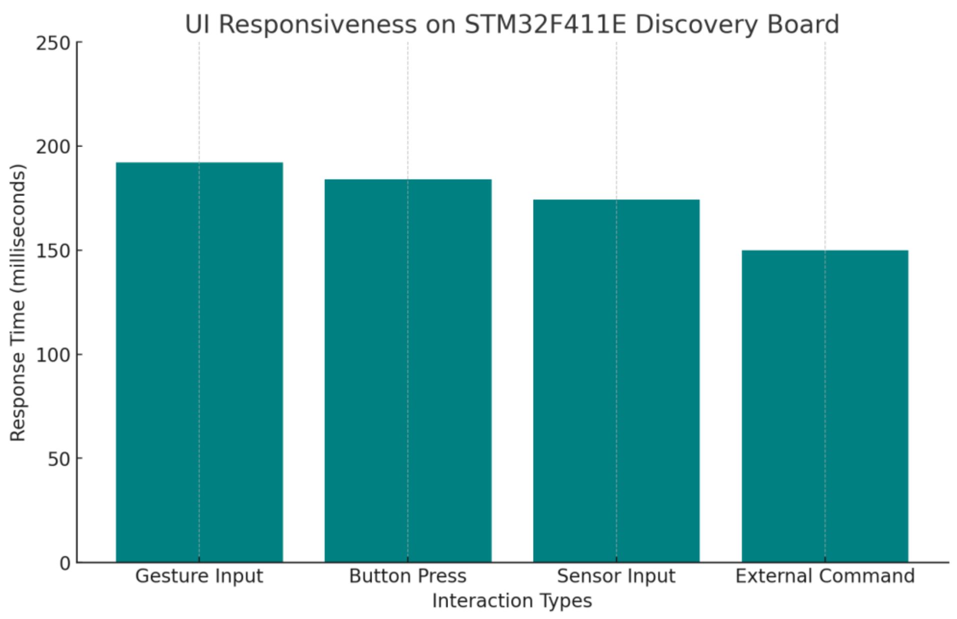


Fig 3.1 Responsiveness of User inputs

UI Responsiveness on STM32F411E Discovery Board Graph: This graph illustrates the response times for various types of user interactions with the STM32F411E Discovery board, including gesture inputs, button presses, sensor inputs, and external commands. The response times are presented in milliseconds (ms).

4 FUTURE DEVELOPMENT IDEAS

4.1 Advanced Gesture Controls

In the realm of enhancing user interaction, advanced gesture controls present a significant opportunity for future development. The current system's gesture recognition can be expanded to include more complex and nuanced gestures. This advancement could involve the use of more sophisticated sensors or the integration of machine learning algorithms for better accuracy and a wider range of detectable gestures.

3D Gesture Recognition: Implementing 3D gesture recognition using advanced sensors can allow users to interact with the cube in a more intuitive and natural way. This could include gestures like rotating the hand to change colors or moving the hand closer to the cube to adjust brightness.

Haptic Feedback: Incorporating haptic feedback to provide tactile responses to user gestures. This can enhance the user experience by confirming successful gesture recognition and making the interaction more immersive.

4.2 Integration with IoT Devices

Integrating the LED cube with IoT (Internet of Things) devices opens a plethora of possibilities for future development. This could turn the cube into a smart, connected device that can interact with other smart devices and systems.

Smart Home Integration: Connecting the cube with smart home systems, allowing it to act as a visual indicator for notifications, such as showing different colors for different types of notifications from smart home devices (e.g., security alerts, weather updates).

Voice Control: Implementing voice control through integration with virtual assistants like Amazon Alexa or Google Assistant. This would allow users to control the cube using voice commands, adding an extra layer of convenience.

Remote Control and Monitoring: Enabling remote control and monitoring of the cube via a smartphone app. This would allow users to interact with the cube even when they are not physically present, for example, setting mood lighting before arriving home.

4.3 Energy Efficiency Improvements

Improving energy efficiency is not only beneficial for reducing operational costs but also aligns with the growing need for environmentally friendly technology.

Low Power Modes: Implementing more aggressive low power modes, where the cube enters a sleep state with minimal energy consumption when not in use, can greatly reduce power usage.

Energy Harvesting: Exploring energy harvesting methods, such as solar power or kinetic energy from user interactions, to supplement or replace the power supply.

Efficient LED Technology: Utilizing the latest advancements in LED technology that offer higher luminosity with lower power consumption. This could include newer types of LEDs or improvements in color rendering efficiency.

Adaptive Brightness: Implementing adaptive brightness control, where the brightness of the LEDs is automatically adjusted based on ambient light conditions. This can conserve energy while maintaining visibility and user comfort.

5. ACKNOWLEDGMENTS

Academic Guidance and Technical Support

- Expressing gratitude to the course instructor Dr. Professor Linden McClure for their invaluable guidance throughout the course. Thankyou for your expertise, assistance in technical matters, and your role in shaping the project.
- Acknowledging the Teaching Assistants those who provided technical support, such as help with answering technical queries, and assisting in sourcing parts for the project.

6. REFERENCES

[1]. https://www.instructables.com/RGB-8x8x8-LED-Cube-1/

[2]. https://www.youtube.com/watch?v=guppB4cK3oU

[3]. https://www.hackster.io/john-bradnam/8x8x8-rgb-led-cube-91729c

[4]. https://github.com/adafruit/Adafruit_APDS9960

[5].https://github.com/adafruit/Adafruit_APDS9960/blob/master/examples/gesture_sensor/gesture_sensor.ino

[6]. https://github.com/adafruit/Adafruit_TCS34725

[7]. https://github.com/libdriver/tcs34725

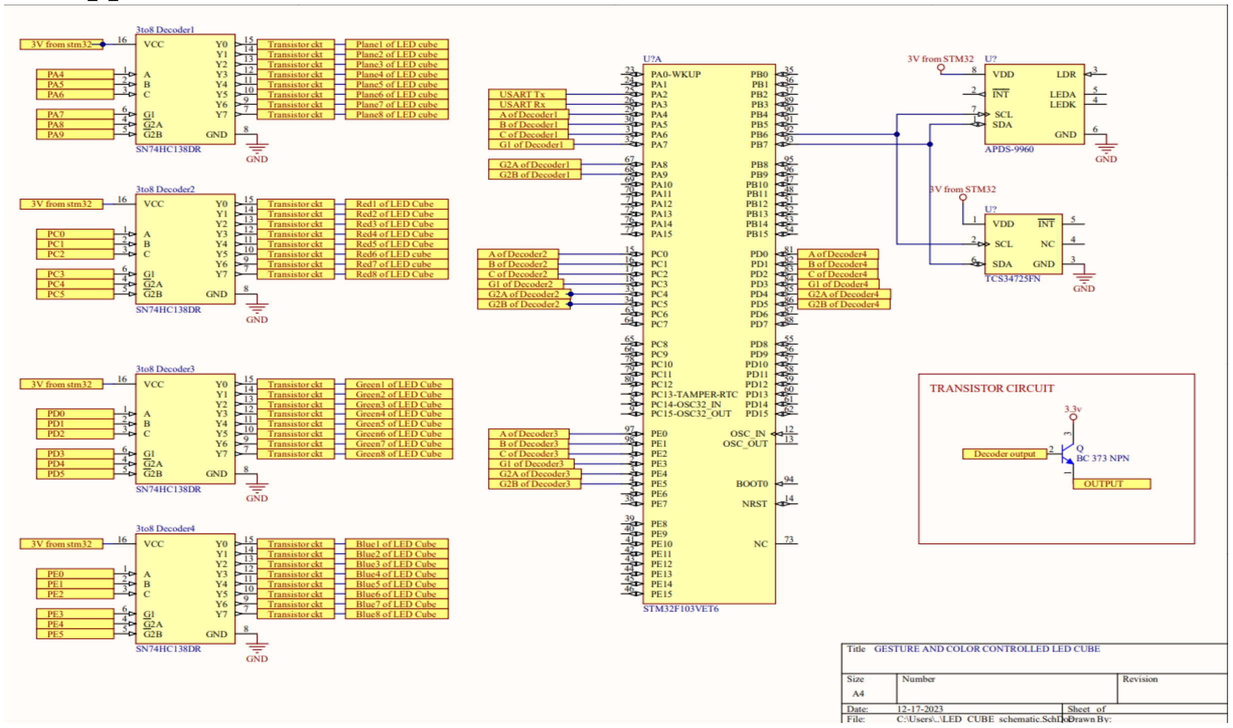
7 APPENDICES

Several appendices have been attached to this report in the order shown below

7.1 Appendix - Bill of Materials

Part Description	**Source**	**Cost**
Adafruit APDS9960 Sensor module	Amazon	13.34$
HiLetgo TCS-34725 Sensor module	Amazon	6.99$
STM32F411Ediscovery board	ESD Kit	0.00$
Bread board	loan from ITLL	0.00$
Decoders, transistors & resistors	loan from ITLL	0.00$
Jumper wires/ wire extensions	loan from ITLL	0.00$
Zero PCB board	loan from ITLL	0.00$
12V power adapter	ESD Kit	0.00$
Power Socket	Loan from ITLL	0.00$
	Total	20.34$

7.2 Appendix - Hardware Schematics



7.3 Appendix - MCU Firmware Source Code

```
/*
 * color.c
 *
 * Created on: Dec 6, 2023
 * Author: Shruthi Thallapally
 *
 * Description: This module is designed to interact with the TCS34725 color
sensor.
 * It includes initialization and functions to read and interpret color data.
 */

#include "color.h"
#include "i2c.h"
#include "stm32f4xx.h" // Include appropriate header file
#include "stdio.h"
#include "string.h"
#include "uart.h"

uint16_t r = 0, g = 0, b = 0, c = 0; // Variables to hold color data (red, green,
blue, clear)

/**
 * Initializes the TCS34725 color sensor.
 * Sets up the sensor for color detection including power, integration time, and
gain control.
 */
void TCS34725_Init(void) {
    write_i2c(TCS34725_ADDRESS, (0x80|TCS34725_ENABLE), 0x03); // Power on and
enable ADC
    write_i2c(TCS34725_ADDRESS, (0x80| TCS34725_ATIME), 0xEB);  // Set
integration time
    write_i2c(TCS34725_ADDRESS, (0x80|TCS34725_CONTROL), 0x02); // Set gain
control to 16x
}

/**
 * Reads color data from the TCS34725 sensor and identifies the predominant
color.
 *
 * Returns:
 * PredominantColor - The identified predominant color or UNKNOWN if unable to
determine.
 */
PredominantColor TCS34725_ReadColorAndCheck() {

    // Read color data from the sensor
    c = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_CDATAL));
    r = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_RDATAL));
    g = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_GDATAL));
    b = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_BDATAL));
    // Determine the predominant color
    if(c > 2000) {
        printf("\n\rUnknown color\n\r");
        return UNKNOWN;
    } else if (r > g && r > b) {
        printf("\n\rDetected color is red\n\r");
        return RED;
    } else if (g > r && g > b) {
        printf("\n\rDetected color is green\n\r");
        return GREEN;
    } else if (b > r && b > g) {
        printf("\n\rDetected color is blue\n\r");
        return BLUE;
    }
}

/**
 * Reads raw color data (red, green, blue, clear) from the TCS34725 sensor.
 *
 * Parameters:
 * r, g, b, c - Pointers to store the read color data.
 */
void TCS34725_ReadColor(uint16_t *r, uint16_t *g, uint16_t *b, uint16_t *c) {
    *c = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_CDATAL));
    *r = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_RDATAL));
    *g = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_GDATAL));
    *b = read_i2c_word(TCS34725_ADDRESS, (0x80|TCS34725_BDATAL));

    // Optional: Print color data for debugging
    // printf("\n\rRed: %u, Green: %u, Blue: %u, Clear: %u\n\r", *r, *g, *b, *c);
}

/*
 * color.h
 *
 * Created on: Dec 6, 2023
 * Author: Shruthi Thallapally
 *
 * Description: Header file for color sensor module.
 * This file contains definitions and function prototypes for interfacing with
the TCS34725 color sensor.
 */

#ifndef SRC_COLOR_H_
#define SRC_COLOR_H_

#include "stdint.h"

// I2C address for the TCS34725 color sensor
#define TCS34725_ADDRESS 0x29
```

```c
// TCS34725 register addresses
#define TCS34725_ENABLE  0x00 // Enable register
#define TCS34725_ATIME   0x01 // Integration time register
#define TCS34725_CONTROL 0x0F // Control register (gain settings)
#define TCS34725_CDATAL  0x14 // Lower byte of clear channel data
#define TCS34725_RDATAL  0x16 // Lower byte of red channel data
#define TCS34725_GDATAL  0x18 // Lower byte of green channel data
#define TCS34725_BDATAL  0x1A // Lower byte of blue channel data

// Global variables for color data
extern uint16_t r, g, b, c; // Red, Green, Blue, and Clear color values

/**
 * Enumeration for predominant colors.
 */
typedef enum {
    RED,     // Predominant red color
    GREEN,   // Predominant green color
    BLUE,    // Predominant blue color
    UNKNOWN  // Color is unknown or not identifiable
} PredominantColor;

// Function prototypes

/**
 * Initializes the TCS34725 color sensor.
 */
void TCS34725_Init(void);

/**
 * Reads and identifies the predominant color from the TCS34725 sensor.
 *
 * Returns:
 * PredominantColor - the predominant color detected.
 */
PredominantColor TCS34725_ReadColorAndCheck();

/**
 * Reads raw color data (red, green, blue, clear) from the TCS34725 sensor.
 *
 * Parameters:
 * r, g, b, c - Pointers to store the read color data.
 */
void TCS34725_ReadColor(uint16_t *r, uint16_t *g, uint16_t *b, uint16_t *c);

// Optional SysTick_Handler function declaration (commented out as it may not be
needed in this context)
// void SysTick_Handler(void);

#endif /* SRC_COLOR_H_ */
```

```
/*
 * gesture.c
 *
 *  Created on: Dec 6, 2023
 *      Author: Shruthi Thallapally
 *
 * Description: This module is designed to interface with the APDS9960 sensor for
gesture detection.
 *
 */
#include "stdint.h"
#include "math.h"
#include "stm32f4xx.h"
#include "stdio.h"
#include "string.h"
#include "gesture.h"
#include "i2c.h"



uint8_t gestCnt;

  uint8_t UCount;
  uint8_t DCount;

  uint8_t LCount;
  uint8_t RCount;

 /**
 * Initializes the APDS9960 gesture sensor.
 * Configures the necessary registers and settings for gesture detection.
 */
void apds9960_init()
{
    // Enable the sensor and set up gesture detection parameters
    write_i2c(APDS9960_I2C_ADDRESS, 0x80, 0x01); // ENABLE register: Power ON

    // Configure gesture sensor settings
    write_i2c(APDS9960_I2C_ADDRESS, 0xAA, 0x00); // GCONF3: Both pairs active
    write_i2c(APDS9960_I2C_ADDRESS, 0xA2, 0x00); // GFIFOTHreshold
    write_i2c(APDS9960_I2C_ADDRESS, 0xA3, 0x57); // GCONF2: Gesture Gain Control,
LED Drive Strength, Gesture Wait Time
    write_i2c(APDS9960_I2C_ADDRESS, 0xA6, 0x80); // GPULSE: Gesture Pulse Count
and Length
    write_i2c(APDS9960_I2C_ADDRESS, 0xAB, 0x01); // GCONF4: GMODE set to 1

    // Set gesture proximity and offset parameters
    write_i2c(APDS9960_I2C_ADDRESS, 0xA0, 0x50); // GPENTH: Gesture Proximity
Entry Threshold
```

```
    write_i2c(APDS9960_I2C_ADDRESS, 0xA1, 0x1F); // GEXTH: Gesture Exit Threshold
    write_i2c(APDS9960_I2C_ADDRESS, 0xA4, 0x00); // GOFFSET_U: Gesture Offset UP
    write_i2c(APDS9960_I2C_ADDRESS, 0xA5, 0x00); // GOFFSET_D: Gesture Offset
DOWN
    write_i2c(APDS9960_I2C_ADDRESS, 0xA7, 0x00); // GOFFSET_L: Gesture Offset
LEFT
    write_i2c(APDS9960_I2C_ADDRESS, 0xA9, 0x00); // GOFFSET_R: Gesture Offset
RIGHT

    // Finalize configuration
    write_i2c(APDS9960_I2C_ADDRESS, 0x80, 0x45); // ENABLE register: Gesture
Enable + Power ON

    // Reset gesture detection counters
    resetCounts();
}
/**
 * Checks if the gesture sensor is properly initialized.
 *
 * Returns:
 * 1 if initialization is successful, 0 otherwise.
 */
int check_gesture_init() {
    uint8_t x = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_ID);
    return (x == 0xAB) ? 1 : 0;
}


/**
 * Detects and returns the type of gesture detected by the APDS9960 sensor.
 *
 * Returns:
 * gesture_t - the type of gesture detected (e.g., GESTURE_UP, GESTURE_DOWN,
etc.).
 */
gesture_t detect_gesture() {
    uint8_t fifo_level, u_data, d_data, l_data, r_data, up_down_diff,
left_right_diff;
    gesture_t gesture = GESTURE_NONE;

    // Read gesture FIFO level and data, then calculate differences
    fifo_level = read_i2c(APDS9960_I2C_ADDRESS, 0xAE);
    for (int i = 0; i < fifo_level; i++) {
        u_data = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_GFIFO_U);
        d_data = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_GFIFO_D);
        l_data = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_GFIFO_L);
        r_data = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_GFIFO_R);

        up_down_diff += u_data - d_data;
        left_right_diff += l_data - r_data;
    }
    // Determine gesture based on difference values
    if (abs(left_right_diff) > abs(up_down_diff)) {
        if (left_right_diff > GESTURE_THRESHOLD) {
            gesture = GESTURE_RIGHT;
        } else if (left_right_diff < -GESTURE_THRESHOLD) {
            gesture = GESTURE_LEFT;
        }
    } else {
        if (up_down_diff > GESTURE_THRESHOLD) {
            gesture = GESTURE_UP;
        } else if (up_down_diff < -GESTURE_THRESHOLD) {
            gesture = GESTURE_DOWN;
        }
    }

    return gesture; // Return the detected gesture (or GESTURE_NONE if no gesture
detected)
}

/**
 * Resets the gesture detection counters.
 */
void resetCounts() {
    gestCnt = 0;
    UCount = 0;
    DCount = 0;
    LCount = 0;
    RCount = 0;
}
/**
 * Checks if gesture data is available from the sensor.
 *
 * Returns:
 * 1 if data is available, 0 otherwise.
 */
uint8_t gesture_data_available() {
    uint8_t gstatus = read_i2c(APDS9960_I2C_ADDRESS, APDS9960_GSTATUS);
    return (gstatus & 0b00000001) ? 1 : 0;
}


/*
 * gesture.h
 *
 * Created on: Dec 6, 2023
 * Author: Shruthi Thallapally
 *
 * Description: Header file for the gesture detection module.
 * This file contains definitions, constants, and function prototypes
 * for interfacing with the APDS9960 gesture sensor.
 */
```

```c
#ifndef SRC_GESTURE_H_
#define SRC_GESTURE_H_

#include "stdint.h"

// I2C and GPIO constants for APDS9960
#define APDS9960_I2C_ADDRESS  0x39 // I2C address of APDS9960
#define APDS9960_GSTATUS      0xAF // Address of the GSTATUS register
#define APDS9960_ID           0x92 // Device ID register address

// GPIO pin definitions (modify as per your hardware configuration)
#define I2C1_SCL_PIN          GPIO_PIN_6
#define I2C1_SDA_PIN          GPIO_PIN_7
#define I2C1_GPIO_PORT        GPIOB

// APDS9960 register addresses
#define APDS9960_ENABLE       0x80
#define APDS9960_ATIME        0x81
#define APDS9960_CONTROL      0x8F
#define APDS9960_GCONF4       0xAB
#define APDS9960_GCONF3       0xAA
#define APDS9960_GCONF1       0xA2
#define APDS9960_GCONF2       0xA3
#define APDS9960_GPENTH       0xA0
#define APDS9960_GEXTH        0xA1
#define APDS9960_GPULSE       0xA6
#define APDS9960_GOFFSET_U    0xA4
#define APDS9960_GOFFSET_D    0xA5
#define APDS9960_GOFFSET_L    0xA7
#define APDS9960_GOFFSET_R    0xA9
#define APDS9960_GFIFO_U      0xFC
#define APDS9960_GFIFO_D      0xFD
#define APDS9960_GFIFO_L      0xFE
#define APDS9960_GFIFO_R      0xFF

#define GESTURE_THRESHOLD 30 // Threshold for gesture detection

// Enumeration for possible gesture types
typedef enum {
    GESTURE_NONE,   // No gesture detected
    GESTURE_UP,     // Upward swipe gesture
    GESTURE_DOWN,   // Downward swipe gesture
    GESTURE_LEFT,   // Left swipe gesture
    GESTURE_RIGHT   // Right swipe gesture
} gesture_t;

/* Function Prototypes */

/**
 * Initializes the APDS9960 gesture sensor.
 */
void apds9960_init();

/**
 * Resets the internal gesture detection counts.
 */
void resetCounts();

/**
 * Checks if the gesture sensor is properly initialized.
 *
 * Returns:
 * 1 if initialization is successful, 0 otherwise.
 */
int check_gesture_init();

/**
 * Detects and returns the type of gesture detected by the APDS9960 sensor.
 *
 * Returns:
 * gesture_t - the type of gesture detected.
 */
gesture_t detect_gesture();

/**
 * Checks if gesture data is available from the sensor.
 *
 * Returns:
 * 1 if data is available, 0 otherwise.
 */
uint8_t gesture_data_available();

#endif /* SRC_GESTURE_H_ */


/*
 * led.c
 *
 *  Created on: Dec 10, 2023
 *      Author: Shruthi Thallapally
 *
 * Description:
 * This file contains functions for controlling LEDs on a custom LED matrix
setup.
 * It includes initialization of GPIOs, various LED display patterns, and helper
functions
 * for setting and clearing individual LEDs based on color.
 */

#include "stm32f4xx.h"
#include "led.h"
```

```c
volatile uint32_t msTicks;  // Variable to store elapsed milliseconds

/**
 * Initializes GPIO for LED control.
 * Sets up GPIO ports A, C, D, and E for output to control the LED matrix.
 */
void initGPIO() {
    // Enable GPIO clock for Ports A, C, D, and E
    RCC_AHB1ENR |= (1 << 0) | (1 << 2) | (1 << 3) | (1 << 4);

    // Set GPIOA, GPIOC, GPIOD, GPIOE pins as output
    GPIOA_MODER &= ~(0xFFFFFFFF);
    GPIOA_MODER |= 0x55555555; // GPIOA_MODER
    GPIOC_MODER &= ~(0xFFFFFFFF);
    GPIOC_MODER |= 0x55555555; // GPIOC_MODER
    GPIOD_MODER &= ~(0xFFFFFFFF);
    GPIOD_MODER |= 0x55555555; // GPIOD_MODER
    GPIOE_MODER &= ~(0xFFFFFFFF);
    GPIOE_MODER |= 0x55555555; // GPIOE_MODER
}


/**
 * Displays an upward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayUpPattern(PredominantColor color) {
    for (int layer = 0; layer < 8; layer++) {
        // Turn on all LEDs in the current layer
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                setLED(layer, row, col,color);
            }
        }
        Delay_ms(1000);
        // Turn off all LEDs in the current layer
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                clearLED(layer, row, col,color);
            }
        }
        Delay_ms(1000);
    }
}

/**
 * Displays a downward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayDownPattern(PredominantColor color) {
    for (int layer = 7; layer >= 0; layer--) {
        // Turn on all LEDs in the current layer
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                setLED(layer, row, col,color);
            }
        }
        Delay_ms(1000);

        // Turn off all LEDs in the current layer
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                clearLED(layer, row, col,color);
            }
        }
        Delay_ms(1000);
    }
}

/**
 * Displays a rightward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayRightPattern(PredominantColor color) {
    for (int layer = 0; layer < 8; layer++) {
        for (int col = 0; col < 8; col++) {
            // Turn on a vertical line in the current column
            for (int row = 0; row < 8; row++) {
                setLED(layer, row, col,color);
            }
            Delay_ms(1000);
            // Turn off the vertical line in the current column
            for (int row = 0; row < 8; row++) {
                clearLED(layer, row, col,color);
            }
            Delay_ms(1000);
        }
    }
}

/**
 * Displays a leftward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
```

```
void displayLeftPattern(PredominantColor color) {
    for (int layer = 0; layer < 8; layer++) {
        for (int col = 7; col >= 0; col--) {
            // Turn on a vertical line in the current column
            for (int row = 0; row < 8; row++) {
                setLED(layer, row, col,color);
            }
            Delay_ms(1000);
            // Turn off the vertical line in the current column
            for (int row = 0; row < 8; row++) {
                clearLED(layer, row, col,color);
            }
            Delay_ms(1000);
        }
    }
}

/**
 * Sets an individual LED to a specified color.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 * color - The color to set the LED.
 */
void setLED(int layer, int row, int col,PredominantColor color) {
    LEDPin redPin = getRedPin(layer, row, col);
        LEDPin greenPin = getGreenPin(layer, row, col);
        LEDPin bluePin = getBluePin(layer, row, col);

        // Set or clear the red component
        if (color == RED) {
            redPin.port->BSRR = (1U << redPin.pin); // Set red pin
        } else {
            redPin.port->BSRR = (1U << (redPin.pin + 16)); // Reset red pin
        }

        // Set or clear the green component
        if (color==GREEN) {
            greenPin.port->BSRR = (1U << greenPin.pin); // Set green pin
        } else {
            greenPin.port->BSRR = (1U << (greenPin.pin + 16)); // Reset green pin
        }

        // Set or clear the blue component
        if (color==BLUE) {
            bluePin.port->BSRR = (1U << bluePin.pin); // Set blue pin
        } else {
            bluePin.port->BSRR = (1U << (bluePin.pin + 16)); // Reset blue pin
        }
}

/**
 * Clears an individual LED, turning it off.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 * color - The color to clear from the LED.
 */
void clearLED(int layer, int row, int col, PredominantColor color) {
    // Get the GPIO pins for the red, green, and blue components
    LEDPin redPin = getRedPin(layer, row, col);
    LEDPin greenPin = getGreenPin(layer, row, col);
    LEDPin bluePin = getBluePin(layer, row, col);

    // Reset the color components
    redPin.port->BSRR = (1U << (redPin.pin + 16)); // Reset red pin
    greenPin.port->BSRR = (1U << (greenPin.pin + 16)); // Reset green pin
    bluePin.port->BSRR = (1U << (bluePin.pin + 16)); // Reset blue pin
}

/**
 * Gets the red component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the red component of the LED.
 */
LEDPin getRedPin(int layer, int row, int col) {
    LEDPin pin;
    //  GPIOA is used for red pins
    pin.port = GPIOC;

    // mapping: pin number is calculated based on layer, row, and column
    pin.pin = layer * 16 + row; // Simplified example
    return pin;
}

/**
 * Gets the green component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the green component of the LED.
 */
LEDPin getGreenPin(int layer, int row, int col) {
    LEDPin pin;
    // GPIOB is used for green pins
    pin.port = GPIOD;

    //  mapping
    pin.pin = layer * 16 + col; // Simplified example
    return pin;
}

/**
 * Gets the blue component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the blue component of the LED.
 */
LEDPin getBluePin(int layer, int row, int col) {
    LEDPin pin;
    //GPIOC is used for blue pins
    pin.port = GPIOE;


    pin.pin = (layer * 16 + row + col) % 16; // Simplified example
    return pin;
}

/**
 * Implements a delay in milliseconds.
 *
 * Parameters:
 * ms - The number of milliseconds to delay.
 */
void Delay_ms(uint32_t ms) {
    msTicks = ms;  // Set the global variable with the desired delay
    while (msTicks != 0) {
        // Wait until the SysTick_Handler decrements the counter to zero
    }
}
```

7.4 Appendix - Software Source Code

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/**
 * @file main.c
 * @brief Main program for LED control with gesture recognition.
 * Created on: Dec 10, 2023
 * Author: Shruthi Thallapally
 *
 * This program initializes the necessary peripherals for LED control and gesture
recognition.
 * It uses an APDS9960 sensor for gesture detection and a TCS34725 sensor for
color recognition.
 * Based on the detected gesture, it displays different LED patterns on a custom
LED matrix.
 */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "led.h"
#include "color.h"
#include "uart.h"
#include "stdint.h"
#include "stm32f4xx.h"
#include "stdio.h"
#include "string.h"
#include "gesture.h"
#include "i2c.h"
```

```c
char rxData;

// Function Prototypes
void SystemClock_Config(void);
void SysTick_Handler(void);
void SysTick_Init(void);
void Delay_ms(uint32_t ms) ;

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  gesture_t gesture;
  // Initialize peripherals
  i2c_gpio_init();
  i2c_init();
  uint8_t a=0;
Here:
  apds9960_init();
  a=check_gesture_init();
  if(a==0){
    printf("Init failed for gesture sensor\n\r");
    goto Here;
  }
  PredominantColor color;
  TCS34725_Init();
  SysTick_Init();
  USART2_Config();
  initGPIO();
  printf("\n\rIn main function\n\r");
  printf("\n\rWaiting for Color input\n\r");

  while (1)
  {
  color=TCS34725_ReadColorAndCheck();
  Delay_ms(1000);
// Process the color data
// printf("\n\rRed: %u, Green: %u, Blue: %u, Clear: %u\n\r", r, g, b, c);

  while(color!= UNKNOWN){
    printf("\n\r Waiting for gesture\n\r");
    if (gesture_data_available()) {
            gesture = detect_gesture();
            Delay_ms(1000);

            switch (gesture) {
                case GESTURE_UP:
                   // Handle UP gesture
```

```
                    printf("\n\r UP\n\r");
                    displayUpPattern(color);
                    break;
                case GESTURE_DOWN:
                  // Handle DOWN gesture
                     printf("\n\r DOWN\n\r");
                     displayDownPattern(color);
                     break;
                case GESTURE_LEFT:
                  // Handle LEFT gesture
                     printf("\n\r LEFT\n\r");
                     displayLeftPattern(color);
                     break;
                case GESTURE_RIGHT:
                  // Handle RIGHT gesture
                     printf("\n\r RIGHT\n\r");
                     displayRightPattern(color) ;
                     break;
                default:
                     printf("\n\r Not A valid gesture\n\r");
                                  // No valid gesture detected
                     break;
                }
            printf("\n\r BLINKING LEDs\n\r");
            Delay_ms(1000);
            break;
        }

    Delay_ms(2000);
  }
  }

}

/**
 * @brief SysTick interrupt handler.
 * Decrements the millisecond counter.
 */
void SysTick_Handler(void) {
    if (msTicks != 0) {
        msTicks--;  // Decrement the milliseconds counter if not already zero
    }
}

/**
 * @brief Initializes SysTick with 1ms interval.
 * Sets up the SysTick timer for 1ms ticks.
 */
void SysTick_Init(void) {
    SystemCoreClockUpdate();  // Update system clock variable
    SysTick_Config(SystemCoreClock / 1000);  // Configure SysTick for 1ms
intervals
}


/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 192;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 8;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
  {
    Error_Handler();
  }
}
```

```c
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT

/**

  * @brief  Reports the name of the source file and the source line number

  *         where the assert_param error has occurred.

  * @param  file: pointer to the source file name

  * @param  line: assert_param error line source number

  * @retval None

  */

void assert_failed(uint8_t *file, uint32_t line)

{

  /* USER CODE BEGIN 6 */

  /* User can add his own implementation to report the file name and line number,

     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

  /* USER CODE END 6 */

}

#endif /* USE_FULL_ASSERT */


/*
 * uart.c
 *
 *  Created on: Dec 8, 2023
 *      Author: Shruthi Thallapally
 *
 *
 * Description:
 * This file contains the implementation of UART communication functions for
STM32F4xx microcontrollers.
 * It includes configuration of UART2 for basic transmit and receive
capabilities.
 *
 */

#include "stm32f4xx.h"
#include "uart.h"
#include "main.h"
#include "color.h"
#include "uart.h"
#include "stdint.h"
#include "stdio.h"
#include "string.h"
#include "i2c.h"

/**
 * Configures USART2 for UART communication.
 * Sets up GPIO pins for USART2, configures baud rate, and enables the
transmitter and receiver.
 */
void USART2_Config(void) {
    // Enable USART2 and GPIOA clocks
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

    // Configure GPIO pins for USART2 (PA2 = Tx, PA3 = Rx)
    GPIOA->AFR[0] = (7 << (4 * 2)) | (7 << (4 * 3));
    GPIOA->MODER = (GPIOA->MODER & ~(3 << (2 * 2))) | (2 << (2 * 2)) | (2 << (2 *
3));

    // Configure USART2
    USART2->BRR = 0x683; // 9600 baud rate @ 16MHz
    USART2->CR1 = USART_CR1_TE | USART_CR1_RE  | USART_CR1_UE; // Enable
transmitter and USART


}

/**
  * @brief Transmit a character via UART2
  * @param ch: Character to be transmitted
  */
void UART2_TxChar(char ch) {
    while (!(USART2->SR & USART_SR_TXE))
        {
        ;
        }
    USART2->DR = ch;
}

/**
  * @brief Receive a character via UART2
  * @return Received character
  */
char UART2_RxChar() {
    while (!(USART2->SR & USART_SR_RXNE));
    return USART2->DR;
}

/**
 * Overrides the _write function to redirect printf() to UART2.
 *
 * Parameters:
 * file - File descriptor.
 * ptr  - Pointer to the buffer containing characters to transmit.
 * len  - Number of characters to transmit.
 *
 * Returns:
 * int - The number of characters transmitted.
 */
int _write(int file, char *ptr, int len) {
    int i;
    for (i = 0; i < len; i++) {
        while (!(USART2->SR & USART_SR_TXE));  // Wait until TX buffer is empty
        USART2->DR = (ptr[i] & 0xFF);  // Send a character
    }
    return len;
}

/*

 * uart.h

 *

 *  Created on: Dec 8, 2023

 *      Author: Shruthi thallapally

 *

 * Description:

 * This header file defines the prototypes for UART communication functions using
USART2

 * on STM32F4xx microcontrollers. It includes functions for configuring UART,
transmitting
```

```
 * a single character, and receiving a single character. Additionally, it
provides a

 * function for redirecting standard output to UART for use with printf().

 *

 */


#ifndef SRC_UART_H_
#define SRC_UART_H_

/**
 * Configures USART2 for UART communication.
 * This function sets up the necessary registers and configurations for UART
communication
 * using USART2, including baud rate, mode, and enabling necessary peripherals.
 */
void USART2_Config(void);

/**
 * Transmits a single character over UART2.
 * This function sends a character via USART2, waiting until the transmission
buffer is empty.
 *
 * Parameters:
 * ch - The character to be transmitted.
 */
void UART2_TxChar(char ch);

/**
 * Receives a single character over UART2.
 * This function waits for and returns a character received via USART2.
 *
 * Returns:
 * char - The received character.
 */
char UART2_RxChar();

/**
 * Overrides the standard _write function for redirecting printf() output to
UART.
 * This function sends a string of characters via USART2, allowing printf() to
output
 * to the UART terminal.
 *
 * Parameters:
 * file - File descriptor (not used).
 * ptr  - Pointer to the character array to be transmitted.
 * len  - Number of characters to transmit.
 *
 * Returns:
 * int - The number of characters transmitted.
 */
int _write(int file, char *ptr, int len);

#endif /* SRC_UART_H_ */

/*
 * i2c.c
 *
 *  Created on: Dec 6, 2023
 *      Author: Shruthi Thallapally
 */
#include "i2c.h"
#include "stm32f4xx.h" // Include appropriate header file
#include "gesture.h"
#include "stdint.h"
#include "color.h"


/**
 * Initializes GPIO pins for I2C communication.
 * This function sets GPIOB pins 6 and 7 to alternate function mode for I2C
communication.
 */
void i2c_gpio_init() {
    // Enable GPIOB clock
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOBEN;

    // Set PB6 and PB7 to alternate function, open-drain, pull-up
    GPIOB->MODER &= ~(GPIO_MODER_MODER6 | GPIO_MODER_MODER7 );
    GPIOB->MODER |= (GPIO_MODER_MODER6_1 | GPIO_MODER_MODER7_1 );
    GPIOB->OTYPER |= (GPIO_OTYPER_OT_6 | GPIO_OTYPER_OT_7 );
    GPIOB->PUPDR &= ~(GPIO_PUPDR_PUPDR6 | GPIO_PUPDR_PUPDR7 );
    GPIOB->PUPDR |= (GPIO_PUPDR_PUPDR6_0 | GPIO_PUPDR_PUPDR7_0 );
    GPIOB->AFR[0] |= (0x04 << (4 * 6)) | (0x04 << (4 * 7)); // AF4 for I2C1
     //   GPIOB->AFR[1] |= (0x04 << ((8 - 8) * 4)) | (0x04 << ((9 - 8) * 4)); //
AF4 for I2C1 (PB8 and PB9)

}

/**
 * Initializes the I2C1 peripheral.
 * This function sets up I2C1 for standard mode with appropriate clock and timing
settings.
 */
void i2c_init() {
    // Enable I2C1 clock
    RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;
    // Reset I2C1
    RCC->APB1RSTR |= RCC_APB1RSTR_I2C1RST;
    RCC->APB1RSTR &= ~RCC_APB1RSTR_I2C1RST;

    // Configure I2C1
    I2C1->CR1 &= ~I2C_CR1_PE; // Disable I2C1
    I2C1->CR2 = 16; // Set APB1 clock frequency
    I2C1->CCR = 80; // Set CCR for standard mode
    I2C1->TRISE = 17; // Set TRISE
    I2C1->CR1 |= I2C_CR1_PE; // Enable I2C1
}

/**
 * Generates an I2C start condition.
 * This function sends an I2C start signal to begin a transmission.
 */
void i2c_start() {
    I2C1->CR1 |= I2C_CR1_START; // Generate start condition
    while (!(I2C1->SR1 & I2C_SR1_SB)); // Wait for start condition to be
generated
}

/**
 * Generates an I2C stop condition.
 * This function sends an I2C stop signal to end a transmission.
 */
void i2c_stop() {
    I2C1->CR1 |= I2C_CR1_STOP; // Generate stop condition
}

/**
 * Reads a byte from the I2C bus with acknowledgment.
 * This function reads a single byte and acknowledges the receipt.
 *
 * Returns:
 * uint8_t - The byte read from the I2C bus.
 */
uint8_t i2c_read_ack() {
    I2C1->CR1 |= I2C_CR1_ACK; // Enable ACK
    while (!(I2C1->SR1 & I2C_SR1_RXNE)); // Wait until data register is not empty
    return I2C1->DR;
}

/**
 * Reads a byte from the I2C bus without acknowledgment.
 * This function reads a single byte without acknowledging the receipt.
 *
 * Returns:
 * uint8_t - The byte read from the I2C bus.
 */
uint8_t i2c_read_nack() {
    I2C1->CR1 &= ~I2C_CR1_ACK; // Disable ACK
    i2c_stop(); // Send stop condition
    while (!(I2C1->SR1 & I2C_SR1_RXNE)); // Wait until data register is not empty
    return I2C1->DR;
}

/**
 * Writes a byte to a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to write to.
 * data       - The data byte to be written.
 */
void write_i2c(uint8_t deviceAddr, uint8_t reg, uint8_t data) {
    // Start I2C transmission
    i2c_start();

    // Send device address with write operation
    I2C1->DR = (deviceAddr << 1) | 0;
    while(!(I2C1->SR1 & I2C_SR1_ADDR));
    (void)I2C1->SR2;

    // Send register address
    I2C1->DR = reg;
    while(!(I2C1->SR1 & I2C_SR1_TXE));

    // Send data
    I2C1->DR = data;
    while(!(I2C1->SR1 & I2C_SR1_TXE));

    // Stop I2C transmission
    i2c_stop();
}

/**
 * Reads a byte from a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to read from.
 *
 * Returns:
 * uint8_t - The byte read from the specified register.
 */
uint8_t read_i2c(uint8_t deviceAddr, uint8_t reg) {
    uint8_t data;

    // Start I2C transmission
    i2c_start();
    // Send device address with write operation
    I2C1->DR = (deviceAddr << 1) | 0;
    while(!(I2C1->SR1 & I2C_SR1_ADDR));
    (void)I2C1->SR2;

    // Send register address
    I2C1->DR = reg;
    while(!(I2C1->SR1 & I2C_SR1_TXE));

    // Repeated start for read operation
    i2c_start();

    // Send device address with read operation
    I2C1->DR = (deviceAddr << 1) | 1;
    while(!(I2C1->SR1 & I2C_SR1_ADDR));
    (void)I2C1->SR2;

    // Prepare for reception
    I2C1->CR1 &= ~I2C_CR1_ACK;
    I2C1->CR1 |= I2C_CR1_STOP;

    // Receive data
    while(!(I2C1->SR1 & I2C_SR1_RXNE));
    data = I2C1->DR;

    i2c_stop();

    return data;
}

/**
 * Reads a 16-bit word from a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to read from.
 *
 * Returns:
 * uint16_t - The 16-bit word read from the specified register.
 */
uint16_t read_i2c_word(uint8_t deviceAddr, uint8_t reg) {
    uint16_t data = 0;
    uint8_t data1, data2;

    // Start I2C transmission
    i2c_start();

    // Send device address with write operation
    I2C1->DR = (deviceAddr << 1) | 0;
    while(!(I2C1->SR1 & I2C_SR1_ADDR));
    (void)I2C1->SR2;
    // Send register address
    I2C1->DR = reg;
    while(!(I2C1->SR1 & I2C_SR1_TXE));

    // Repeated start for read operation
    i2c_start();

    // Send device address with read operation
    I2C1->DR = (deviceAddr << 1) | 1;
    while(!(I2C1->SR1 & I2C_SR1_ADDR));
    (void)I2C1->SR2;

    // Enable ACK for reception of multiple bytes
    I2C1->CR1 |= I2C_CR1_ACK;

    // Read first byte (MSB)
    while(!(I2C1->SR1 & I2C_SR1_RXNE));
    data1 = I2C1->DR;

    // Disable ACK and set STOP bit to prepare for last byte reception
    I2C1->CR1 &= ~I2C_CR1_ACK;
    I2C1->CR1 |= I2C_CR1_STOP;

    // Read second byte (LSB)
    while(!(I2C1->SR1 & I2C_SR1_RXNE));
    data2 = I2C1->DR;

    // Combine the two bytes
    data = ((uint16_t)data2 << 8) |(uint16_t) (data1 & 0xff);

    return data;
}


/*
 * i2c.h
 *
 * Created on: Dec 6, 2023
 * Author: Shruthi Thallapally
 *
 * Description:
 * This header file defines the prototypes for I2C communication functions for
STM32F4xx microcontrollers.
 * It includes functions for initializing I2C peripheral and GPIO pins, and for
starting and stopping
 * I2C communication. Additionally, it provides functions for reading and writing
data over the I2C bus,
 * which can be used for interfacing with various I2C devices.
 */
```

```c
#ifndef SRC_I2C_H_
#define SRC_I2C_H_

#include "stdint.h"

/**
 * Initializes GPIO pins for I2C communication.
 */
void i2c_gpio_init();

/**
 * Initializes the I2C1 peripheral for communication.
 */
void i2c_init();

/**
 * Generates an I2C start condition.
 */
void i2c_start();

/**
 * Generates an I2C stop condition.
 */
void i2c_stop();

/**
 * Reads a byte from the I2C bus with acknowledgment.
 *
 * Returns:
 * uint8_t - The byte read from the I2C bus.
 */
uint8_t i2c_read_ack();

/**
 * Reads a byte from the I2C bus without acknowledgment.
 *
 * Returns:
 * uint8_t - The byte read from the I2C bus.
 */
uint8_t i2c_read_nack();

/**
 * Writes a byte to a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to write to.
 * data       - The data byte to be written.
 */
void write_i2c(uint8_t deviceAddr, uint8_t reg, uint8_t data);
```

```
/**
 * Reads a byte from a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to read from.
 *
 * Returns:
 * uint8_t - The byte read from the specified register.
 */
uint8_t read_i2c(uint8_t deviceAddr, uint8_t reg);

/**
 * Reads a 16-bit word from a specific register of a specified I2C device.
 *
 * Parameters:
 * deviceAddr - The I2C address of the device.
 * reg        - The register address to read from.
 *
 * Returns:
 * uint16_t - The 16-bit word read from the specified register.
 */
uint16_t read_i2c_word(uint8_t deviceAddr, uint8_t reg);

#endif /* SRC_I2C_H_ */


/*
 * led.h
 *
 *  Created on: Dec 10, 2023
 *      Author: Shruthi Thallapally
 *
 * Description:
 * This header file defines the function prototypes and structures for
controlling a custom LED matrix.
 * It includes functions for initializing GPIOs for LED control, displaying
patterns based on the
 * predominant color detected, and manipulating individual LEDs in the matrix.

 */

#ifndef SRC_LED_H_
#define SRC_LED_H_
#include "color.h"
#include "gesture.h"
#include "stdint.h"

// Register addresses for GPIO port modes and output data registers
#define RCC_AHB1ENR   (*((volatile uint32_t*)0x40023830))
#define GPIOA_MODER   (*((volatile uint32_t*)0x40020000))
#define GPIOC_MODER   (*((volatile uint32_t*)0x40020800))
```

```
#define GPIOD_MODER   (*((volatile uint32_t*)0x40020C00))
#define GPIOE_MODER   *((volatile uint32_t*)(0x40021000))
#define GPIOA_ODR (*((volatile uint32_t*)0x40020414))
#define GPIOC_ODR (*((volatile uint32_t*)0x40020814))
#define GPIOD_ODR (*((volatile uint32_t*)0x40020C14))
#define GPIOE_ODR (*((volatile uint32_t*)0x40021014))


// Structure for LED pin configuration
typedef struct {
    GPIO_TypeDef* port; // GPIO port
    uint16_t pin;       // GPIO pin number
} LEDPin;

extern volatile uint32_t msTicks;  // Variable to store elapsed milliseconds

/**
 * Initializes GPIO for LED control.
 * Sets up GPIO ports for output to control the LED matrix.
 */
void initGPIO();

/**
 * Displays an upward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayUpPattern(PredominantColor color);

/**
 * Displays a downward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayDownPattern(PredominantColor color);

/**
 * Displays a rightward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayRightPattern(PredominantColor color);

/**
 * Displays a leftward moving pattern on the LED matrix.
 *
 * Parameters:
 * color - The color in which the pattern is displayed.
 */
void displayLeftPattern(PredominantColor color);

/**
 * Sets an individual LED to a specified color.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 * color - The color to set the LED.
 */
void setLED(int layer, int row, int col, PredominantColor color);

/**
 * Clears an individual LED, turning it off.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 * color - The color to clear from the LED.
 */
void clearLED(int layer, int row, int col, PredominantColor color);

/**
 * Gets the red component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the red component of the LED.
 */
LEDPin getRedPin(int layer, int row, int col);

/**
 * Gets the green component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the green component of the LED.
 */
LEDPin getGreenPin(int layer, int row, int col);
```

```
/**
 * Gets the blue component pin for an individual LED.
 *
 * Parameters:
 * layer - The layer of the LED in the matrix.
 * row   - The row of the LED in the matrix.
 * col   - The column of the LED in the matrix.
 *
 * Returns:
 * LEDPin - The pin controlling the blue component of the LED.
 */
LEDPin getBluePin(int layer, int row, int col);

/**
 * Implements a delay in milliseconds.
 *
 * Parameters:
 * ms - The number of milliseconds to delay.
 */
void Delay_ms(uint32_t ms);

#endif /* SRC_LED_H_ */
```

7.5 Appendix - Data Sheets and Technical Notes

- STM32F411E discovery board data sheet:

https://www.st.com/content/ccc/resource/technical/document/datasheet/b3/a5/46/3b/b4/e5/4c/85/DM00115249.pdf/files/DM00115249.pdf/jcr:content/translations/en.DM00115249.pdf

- APDS 9960 sensor module data sheet:
https://www.mouser.com/datasheet/2/678/av02-4191en_ds_apds-9960_2015-11-13-1828458.pdf
- TCS 34725 sensor module data sheet: https://cdn-shop.adafruit.com/datasheets/TCS34725.pdf