SMART SPACE MAPPER

ARJUN SUKUMARAN

JITHENDRA HALENAHALLI SOMASHEKARAIAH

Final Project Report

[ECEN 5613] Embedded System Design

December 11,2023

1 INTRODUCTION

The Smart Space Mapper project originated from a practical and compelling motivation—observing workers struggling with conventional methods to measure ceiling height and gather floor dimensions for tasks such as purchasing tiles. This real-world challenge inspired the development of an easily accessible device capable of simplifying these specific applications and serving as a versatile solution for a range of scenarios. The subsequent sections of this report explore the intricacies of the Smart Space Mapper, delving into its system overview and the collaborative roles of its integrated hardware components.

1.1 System Overview

The Smart Space Mapper integrates advanced hardware components to create a cohesive system designed for indoor space mapping and optimization. At the heart of the system is the RP2040 microcontroller, serving as the computational powerhouse to orchestrate the functionalities of the entire device.

To address the challenge of measuring distances accurately, the project incorporates the TF-Luna LiDAR sensor, utilizing Time-of-Flight technology to provide precise distance measurements. Additionally, the MPU6050 sensor contributes critical data regarding the device's orientation, enhancing spatial awareness for comprehensive mapping.

The user interface is realized through a 1.5-inch OLED black and white display, complemented by two user-friendly buttons that facilitate intuitive navigation through the system's menu. This interactive display not only offers real-time information but also serves as a platform for user input and feedback.

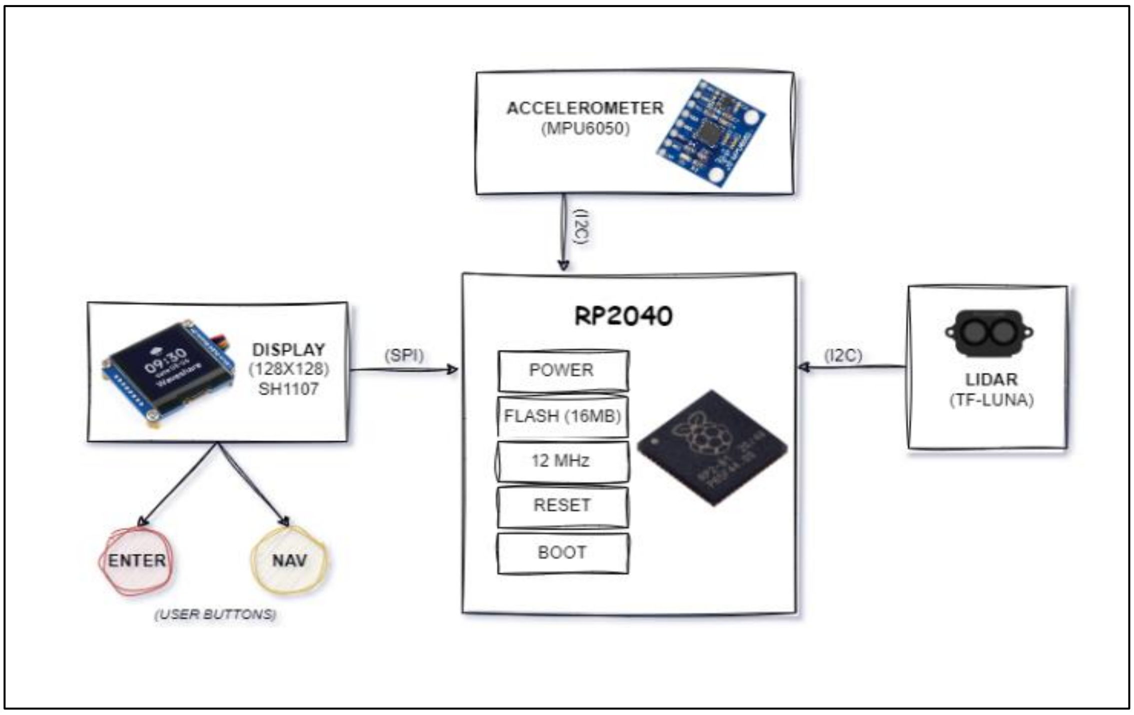


Figure 1.1: Smart Space Mapper System Block Diagram

This block diagram illustrates the key components of the Smart Space Mapper system and their interconnected relationships. The RP2040 microcontroller implemented on a custom-designed board tailored to the specific needs of the Smart Space Mapper project and serves as the central processing unit, orchestrating the functionalities of the TF-Luna LiDAR sensor for distance measurements, the MPU6050 sensor for orientation data, and the 1.5-inch OLED display with user interface and navigation buttons. This cohesive integration forms the foundation for the Smart Space Mapper's capabilities in mapping and optimizing indoor spaces.

The integration of these hardware components aims to create a user-friendly and accessible solution, simplifying specific tasks while laying the foundation for intelligent and optimized indoor spaces. The following sections provide a detailed exploration of each hardware component's functionalities and their collaborative roles within the Smart Space Mapper system.

2 TECHNICAL DESCRIPTION

The technical description of the Smart Space Mapper project encompasses various components and functionalities crucial to its operation. The following sections provide an in-depth overview of each aspect.

2.1 Board Design

The Board Design section delves into the intricacies of the custom-designed board housing the RP2040 microcontroller. This portion addresses critical considerations such as power distribution, signal routing, and compatibility with additional hardware components. The design choices aim for efficiency, reliability, and scalability, laying the foundation for seamless integration and future enhancements.

2.1.1 RP2040 Microcontroller Overview:

The RP2040 microcontroller is an economical yet powerful device offering versatile digital interfaces. Highlighted features include:

- **Dual ARM Cortex M0+ Processors:** Capable of running up to 133MHz.
- **Embedded SRAM:** Providing 264kB of storage across six banks.
- **GPIO and SPI Flash Support:** Equipped with 30 multifunction GPIO pins and six dedicated I/O for SPI flash (supporting XIP).
- **Peripheral Hardware:** Includes dedicated hardware support for commonly used peripherals.
- **Programmable I/O Support:** Allows for extended peripheral functionalities.
- **ADC:** Features a 4-channel ADC with a 12-bit conversion rate of 500ksps.
- The RP2040 microcontroller supports **USB Type-C** connectivity.

The RP2040 supports the execution of code directly from external memory via dedicated SPI, DSPI, or QSPI interfaces. It incorporates a small cache for improved performance in typical applications. Debugging is facilitated through the SWD interface.

Internal SRAM banks can store code or data, accessed via dedicated AHB bus fabric connections, enabling bus masters to access separate bus slaves without being stalled. DMA bus masters are available for repetitive data transfer tasks offloading from processors.

GPIO pins are directly drivable or can be utilized for various dedicated logic functions. Dedicated peripheral IP provides fixed functionalities like SPI, I2C, and UART. Configurable PIO controllers enable a wide range of I/O functions.

The integrated USB controller with an embedded PHY allows for FS/LS host or device connectivity under software control. Additionally, GPIOs and ADC inputs share package pins, while PLLs provide a flexible clock up to 133MHz and a fixed 48MHz clock for USB or ADC.

2.1.2 *Power Regulatory circuit:*

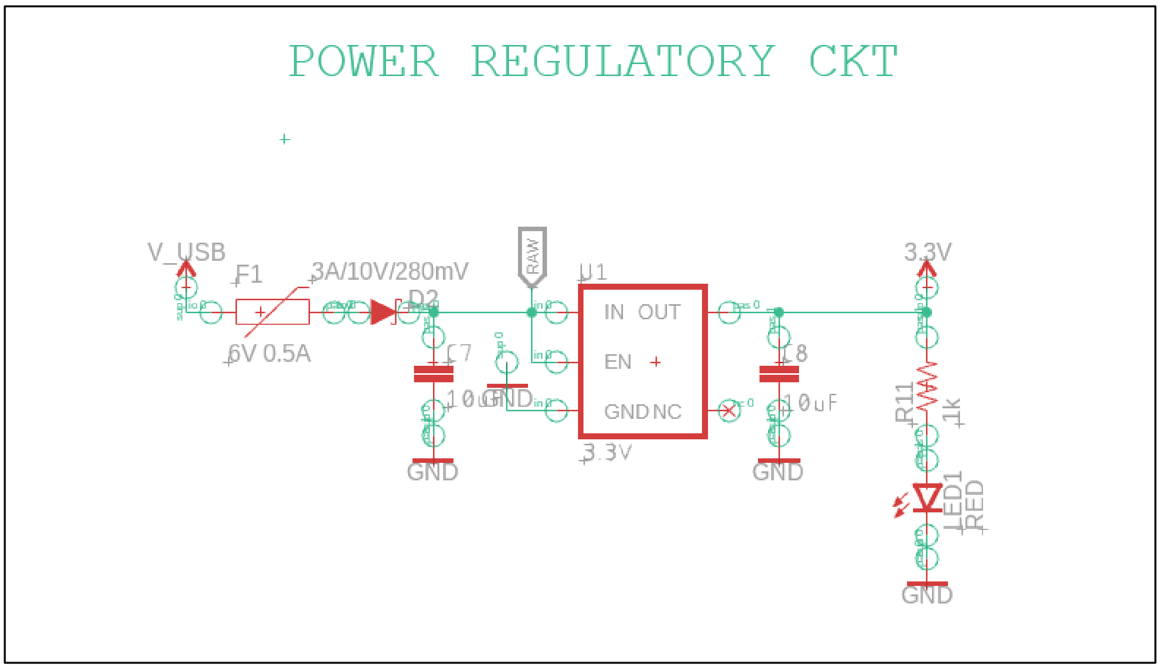


Figure 2.1: Power Regulatory circuit

The power regulatory circuit designed for the RP2040 microcontroller, driven from the USB-C source, encompasses essential protective components and precise voltage regulation mechanisms, ensuring stable power distribution and safeguarding against potential electrical anomalies. The detailed functionalities include:

- **Power Path and Configuration:** Originating from the USB-C connector's differential pairs linked to the microcontroller, the input power channel undergoes grounding in configuration channels, establishing a robust grounding framework within the circuit.

- **Voltage Regulation Pathway:** The power stream from the USB-C source traverses a series of protective components, commencing with a fuse, a Schottky diode, and a decoupling capacitor. The Schottky diode, engineered with specifications of 3A/10V/280mV, effectively prevents reverse current flow while presenting minimal forward voltage drop, bolstering the circuit's protection against voltage irregularities. Complementing the diode's functionality, the 6V, 0.5A fuse acts as a crucial safety measure, providing a barrier against overcurrent scenarios, thus shielding the circuit's vital components. The placement of a decoupling capacitor strategically filters out undesirable noise and stabilizes the input voltage, ensuring a consistent and pristine power supply to the subsequent components.

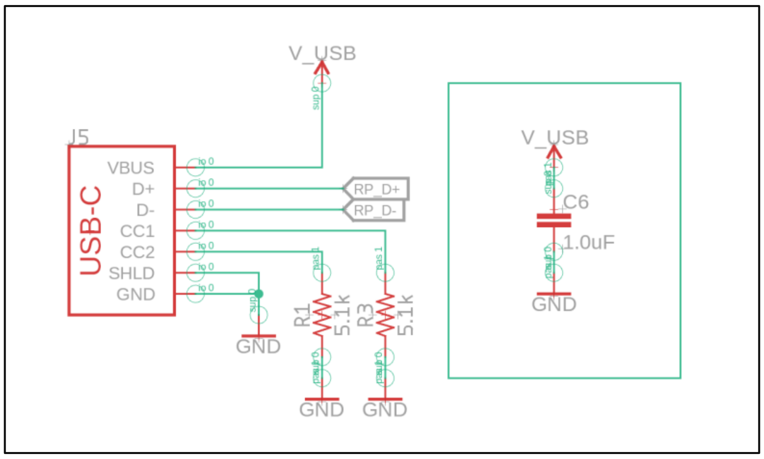


Figure 2.2: USB-C interface with board

- **CMOS Low Dropout Regulator (AP2112):** Positioned within the circuitry, the CMOS low dropout regulator, notably the AP2112, assumes a pivotal role in refining the incoming voltage. This regulator offers a reliable and stable output voltage of 3.3V, serving as the primary VCC for the interconnected components throughout the system. Leveraging its unique features, including precise output voltage accuracy at ±1.5%, low quiescent current consumption of 55µA, and superior load and line regulation, the AP2112 ensures optimal voltage delivery under varying operational conditions.

- **Visual Power Status Indication:** The regulated 3.3V output stream is ingeniously routed to power a red LED, serving as an intuitive indicator of the circuit's power status. This LED illumination confirms successful power regulation, enhancing the user's ability to monitor the system's functionality effortlessly.

2.1.3 W25Q128JV Serial Flash Memory:

The W25Q128JV Serial Flash Memory serves as a cornerstone within our project, facilitating critical data storage and retrieval functionalities. Its robust features and compatibility render it an ideal choice for our application's memory requirements.

Key Features:

- ***Capacity***: With its expansive 128M-bit capacity, the W25Q128JV accommodates our project's substantial data storage needs.
- ***Dual/Quad SPI Support***: The memory's Dual/Quad SPI compatibility ensures efficient data transfer, aligning perfectly with our system's performance criteria.
- ***Operating Environment***: Operating across an industrial-grade temperature range, the memory assures reliability in diverse environmental conditions.

Pin Configuration and Connectivity:

Understanding the memory's pin layout as given in Figure 2.3, especially the Chip Select (/CS), Serial Data Input (DI), and Serial Data Output (DO), is crucial for seamless integration with the RP2040 microcontroller. Proper alignment ensures effective communication and data exchange between the components.

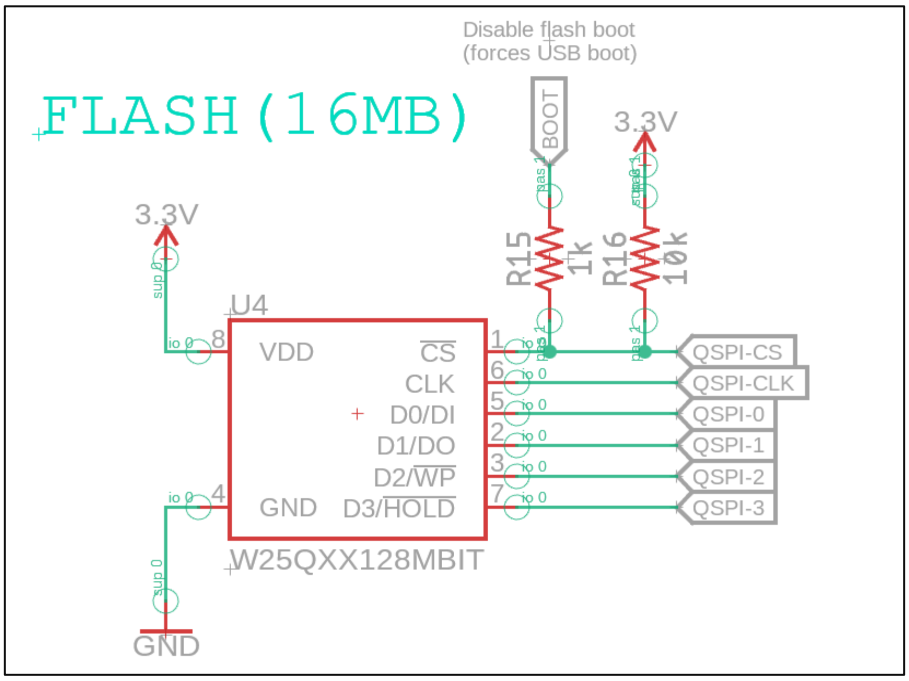


Figure 2.3: W25Q128JV Serial Flash Memory integration

Operational Specifications:

- ***Voltage Range***: Operating within a 2.7V to 3.6V power supply range, the memory suits the standard voltage levels prevalent in embedded systems.
- ***Write Protection***: Robust write protection mechanisms, including the Status Register Write Enable Latch (WEL), guarantee data integrity and security, critical for our project's operations.
- ***Status Register***: The Status Register controls essential operational parameters, enabling vital operational control and management of the memory.
- ***Power Efficiency***: Efficient power management, including low standby and power-down currents, aligns with our project's energy-efficient objectives, minimizing power consumption during operation.

Integration with RP2040:

Integration involves establishing a seamless SPI interface between the W25Q128JV and the RP2040 microcontroller. Utilizing the SPI protocol facilitates efficient data storage and retrieval operations, emphasizing the importance of interface considerations for successful integration.

2.1.4 *Crystal oscillator:*

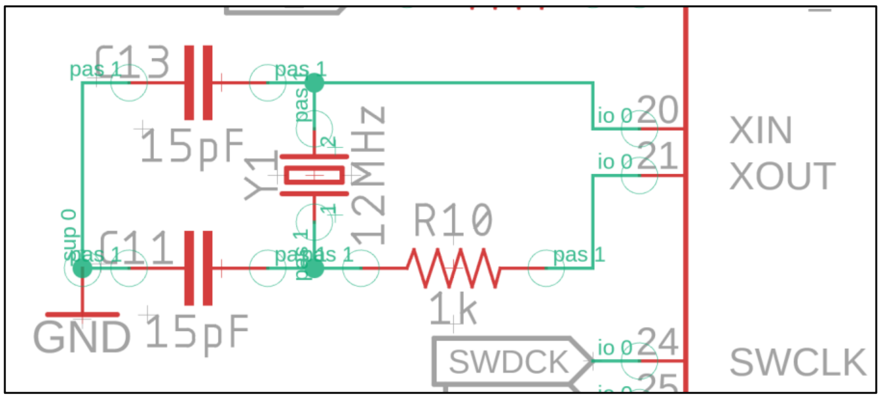


Figure 2.4: Crystal oscillator circuit for external clock

RP2040 does have an internal oscillator but using an external frequency source is recommended due to variations in the internal oscillator's frequency across different chips, supply voltages, and temperatures.

For precise applications, particularly those reliant on exact frequencies like USB, an external frequency source is crucial.

External Frequency Source Options:

- Recommended options for providing an external frequency source are either using a clock source with a CMOS output (3.3V square wave) into the XIN pin or using a 12MHz crystal connected between XIN and XOUT.
- The preferred option is used due to its cost-effectiveness and high accuracy. The crystal, a commonly available 12MHz crystal with a 50ppm tolerance, was chosen for the design.

2.1.5 *Board Layout*

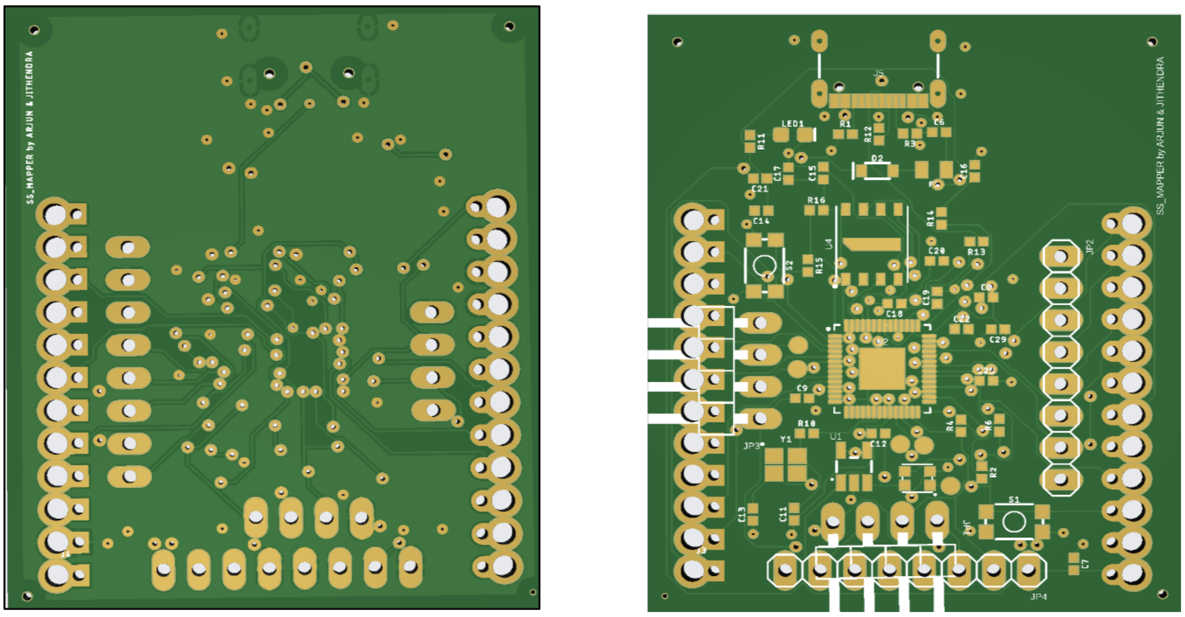


Figure 2.5: Bottom and Top view of the designed board

2.2 TF-Luna LiDAR Sensor

This section outlines the purpose, specifications, and integration of the LiDAR sensor into the Smart Space Mapper system. It highlights specific features such as measurement range, accuracy, and communication protocols, showcasing how this hardware component enhances the overall functionality of the device.

The selection of this specific LiDAR sensor stemmed from its exceptional combination of accuracy, versatile operating range, and high-resolution output. Its adaptability to diverse environmental conditions, coupled with a wide-ranging frame rate and efficient power consumption, perfectly aligns with the project's requirements. Additionally, the sensor's compliance with safety standards, such as its Class 1 photobiological safety rating, and the reliability of its VCSEL light source were pivotal factors influencing the choice. Overall, the sensor's robust specifications and reliability make it an ideal fit for fulfilling the precise distance measurement needs of the project while ensuring consistent performance in varying scenarios.

2.3 MPU6050 Sensor

This portion elucidates the role of the sensor in providing orientation data, contributing to the spatial awareness of the Smart Space Mapper. Specifications including sensitivity, range, and communication interfaces are detailed, showcasing the significance of this hardware component.

The selection of the MPU6050 accelerometer was driven by several key features critical to our project's requirements. Its versatile programmable full-scale range from ±2g to ±16g ensures adaptability for various motion sensing needs. Additionally, the integrated 16-bit ADCs facilitate simultaneous sampling of accelerometers, eliminating the need for an external multiplexer and simplifying the circuit design. The accelerometer's low power consumption modes, with currents as low as 10µA at lower frequencies, align perfectly with our project's emphasis on energy efficiency.

2.4 OLED Display and Navigation Buttons

It provides insights into the user interface aspects of the Smart Space Mapper, detailing the display's specifications (such as resolution and color depth) and the buttons' role in facilitating user interaction. This section emphasizes the synergy between these components for an intuitive user experience.

The selection of the 128x128 OLED display with an SH1107 driver stemmed from its compact 1.5-inch size, ensuring it fits seamlessly into our project's design. The display's compatibility with SPI/I2C, although we utilized SPI through bit banging, offered versatile communication options for interfacing with our system. With a capacity of 16 pages, this display accommodates adequate information for our application's needs, enabling effective data presentation. Furthermore, the display's suitable operating voltage range aligns well with our project's power requirements, ensuring optimal performance within specified power parameters.

2.5 Hardware setup

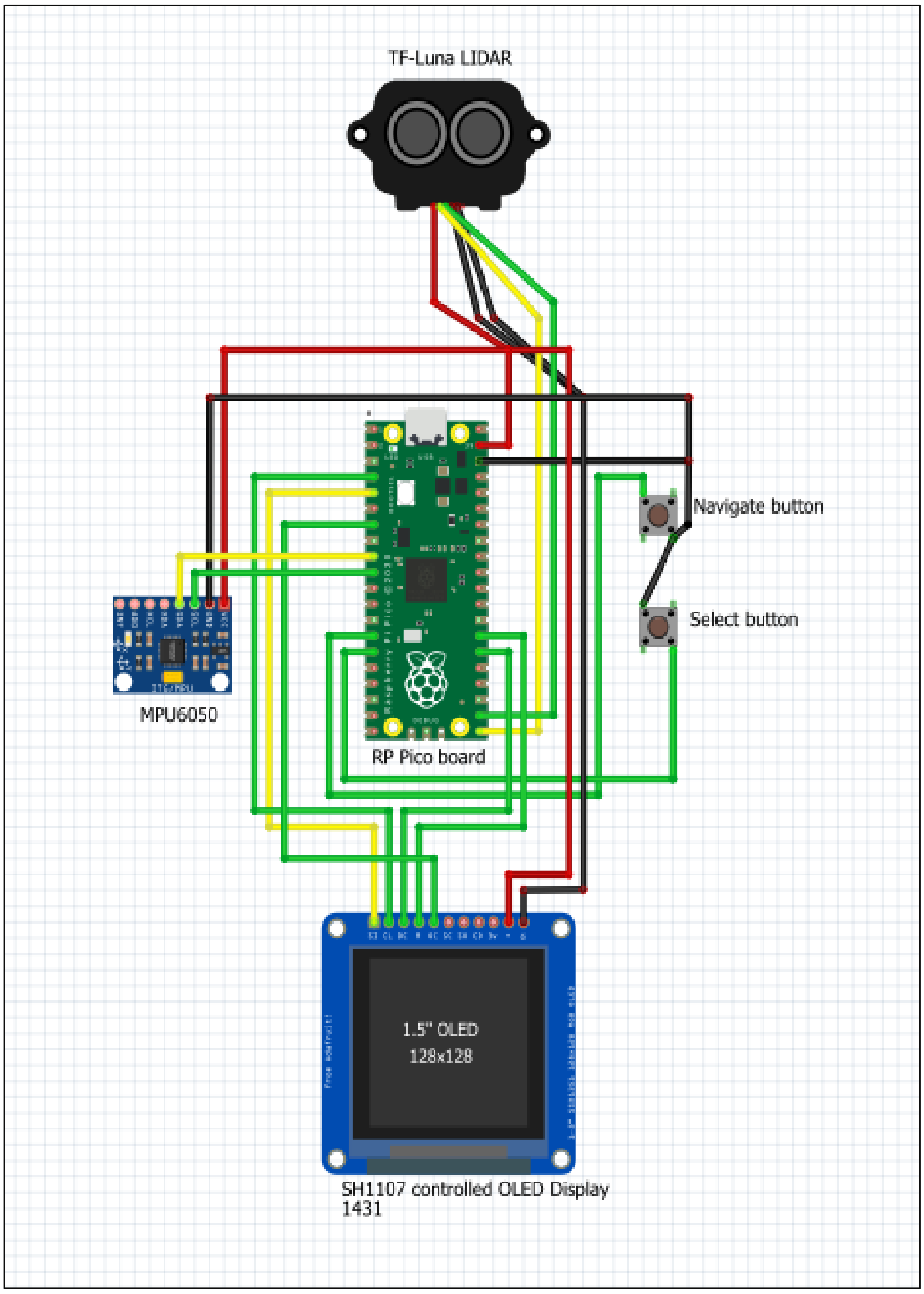


Figure 2.5.1 Circuit connections

The SS Mapper project circuit, created with Fritzing, centers around the Raspberry Pi Pico microcontroller. Key elements include I2C0 for LIDAR interfacing, I2C1 for MPU6050 interfacing, and bit-banged SPI for dynamic display. Yellow wires denote data lines, while green wires signify control lines, ensuring a clear and organized layout.

2.6 Firmware Design

The Firmware Design section plays a pivotal role in orchestrating the low-level programming aspects of the Smart Space Mapper. This segment outlines the intricate details of how the firmware facilitates seamless communication and coordination among the hardware components, highlighting the nuanced considerations for optimal device functionality.

In the context of interfacing the TF-Luna LiDAR sensor and MPU6050 sensor using I2C communication, the firmware implements robust protocols to establish a reliable data exchange. Specifically, the I2C communication protocol is harnessed to ensure efficient and synchronized interaction between the RP2040 microcontroller, TF-Luna LiDAR, and MPU6050 sensor. This includes addressing considerations such as addressing, data transmission, and acknowledgment.

Furthermore, the firmware considers real-time data processing to handle the continuous stream of information from the LiDAR and MPU6050. This involves efficient algorithms for data interpretation, calibration, and synchronization to ensure the accuracy and responsiveness of the spatial data collected by the sensors.

A distinct facet of the firmware design involves the bare-metal coding approach for interfacing with the display. Utilizing the SPI protocol through bit-banging, the firmware precisely controls the communication with the 1.5-inch OLED display. This bare-metal coding strategy eliminates the need for complex software libraries, offering a streamlined and resource-efficient method to drive the display. The firmware ensures synchronization between the sensor data and the display output, providing real-time feedback to the user.

Communication protocols, including I2C for sensors and SPI for the display, are carefully managed within the firmware to create a cohesive and synchronized operation of the entire Smart Space Mapper system. This meticulous approach to firmware design is fundamental in realizing the project's goals of accurate indoor space mapping and optimization. The Firmware Design section serves as a comprehensive guide to the intricate coding strategies and protocols employed to harmonize the diverse hardware components seamlessly.

2.6.1 *SPI Communication Implementation: Bit-Banging Approach*

In the context of the project, the display interface is established using a software-based Serial Peripheral Interface (SPI) technique known as bit-banging. This approach enables the emulation of SPI communication by directly controlling the state of individual bits on the General-Purpose Input/Output (GPIO) pins. The relevant functions in the code initialize and manage the SPI communication for the display module.

```
/**
 * @brief Initializes the SPI module for bit-banging mode.
 */
void SPI_Module_Init_BIT_BANGING() {
    // Set GPIO functions for SPI pins
    iobank0_hw->io[SPI_SCK_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_TX_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_CS_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_DC_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_RESET_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;

    // Set output enables for SPI pins
    sio_hw->gpio_oe_set = (1ul << SPI_SCK_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_TX_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_CS_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_DC_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_RESET_PIN);

    // Clear SPI pins
    sio_hw->gpio_clr = (1ul << SPI_SCK_PIN);
    sio_hw->gpio_clr = (1ul << SPI_TX_PIN);
    sio_hw->gpio_clr = (1ul << SPI_CS_PIN);
}
```

Label 2.1 SPI Init code function implementation.

The *SPI_Module_Init_BIT_BANGING()* function is responsible for configuring the GPIO functions for essential SPI pins, including the clock (SCK), data transmission (TX), chip select (CS), data/command mode (DC), and reset (RESET). The output enables for these pins are set accordingly.

```
/**
 * @brief Sends a byte of data over SPI using bit-banging.
 *
 * @param data The byte of data to be sent.
 */
void SPI_send_byte(uint8_t data) {
    // Reverse the bits of the data byte
    data = reverse(data);

    for(int i = 0; i < BYTE_SIZE; i++) {
        // Extract the least significant bit
        uint8_t bit = data & 0x01;

        // Set or clear the SPI_TX_PIN based on the bit value
        gpio_put(SPI_TX_PIN, bit);

        // Set or clear the GPIO corresponding to SPI_TX_PIN
        if(bit) {
            sio_hw->gpio_set = (1ul << SPI_TX_PIN);
        } else {
            sio_hw->gpio_clr = (1ul << SPI_TX_PIN);
        }
        // Wait for a short duration
        sleep_us(5);
        // Set the SPI clock signal high
        sio_hw->gpio_set = (1ul << SPI_SCK_PIN);
        // Wait for a short duration
        sleep_us(10);
        // Clear the SPI clock signal
        sio_hw->gpio_clr = (1ul << SPI_SCK_PIN);
        // Wait for a short duration
        sleep_us(5);
        // Shift to the next bit
        data = data >> 1;
        // Wait for a short duration
        sleep_us(2);
    }
}
```

Label 2.2 SPI bit banging write function implementation.

Subsequently, the *SPI_send_byte()* function facilitates the transmission of a byte of data over SPI by sequentially manipulating the individual bits. This function forms the core of the bit-banging SPI communication.

```c
/**

 * @brief Writes a command to the SPI module.
 *
 * @param data  The command to write.
 */
void SPI_WriteCommand(const uint8_t data) {
    // Set DC pin low to indicate command mode
    sio_hw->gpio_clr = SET_SPI_DC_PIN;

    // Set CS pin low to enable SPI communication
    sio_hw->gpio_clr = SET_SPI_CS_PIN;

    // Write command data to SPI
    SPI_send_byte(data);

    // Set CS pin high to disable SPI communication
    sio_hw->gpio_set = SET_SPI_CS_PIN;
}
```

Label 2.3 SPI Command Write function implementation.

For communication with the display, two key functions are employed. The *SPI_WriteCommand()* function is used to write commands to the SPI module. It sets the data/command (DC) pin low to indicate command mode, enables SPI communication by setting the chip select (CS) pin low, sends the command data using the SPI_send_byte() function, and then disables SPI communication by setting the CS pin high.

```c
/**

 * @brief Writes data to the SPI module in data mode.
 *
 * @param data  The data to write.
 */
void SPI_WriteData(const uint8_t data) {
    // Set DC pin high to indicate data mode
    sio_hw->gpio_set = SET_SPI_DC_PIN;

    // Set CS pin low to enable SPI communication
    sio_hw->gpio_clr = SET_SPI_CS_PIN;

    // Write data to SPI
    SPI_send_byte(data);

    // Set CS pin high to disable SPI communication
    sio_hw->gpio_set = SET_SPI_CS_PIN;
}
```

Label 2.4 SPI Data Write function implementation.

Similarly, the SPI_WriteData(const uint8_t data) function is utilized to write data to the SPI module in data mode. It sets the DC pin high to indicate data mode, enables SPI communication, transmits the data using SPI_send_byte(), and subsequently disables SPI communication.

2.7 Software Design

In the software design phase, we focus on the high-level programming and user interface logic, detailing the development of the user interface, navigation algorithms, and specialized algorithms for space mapping and optimization. This section underscores how the software translates user inputs into actionable commands, providing a cohesive and intuitive user experience.

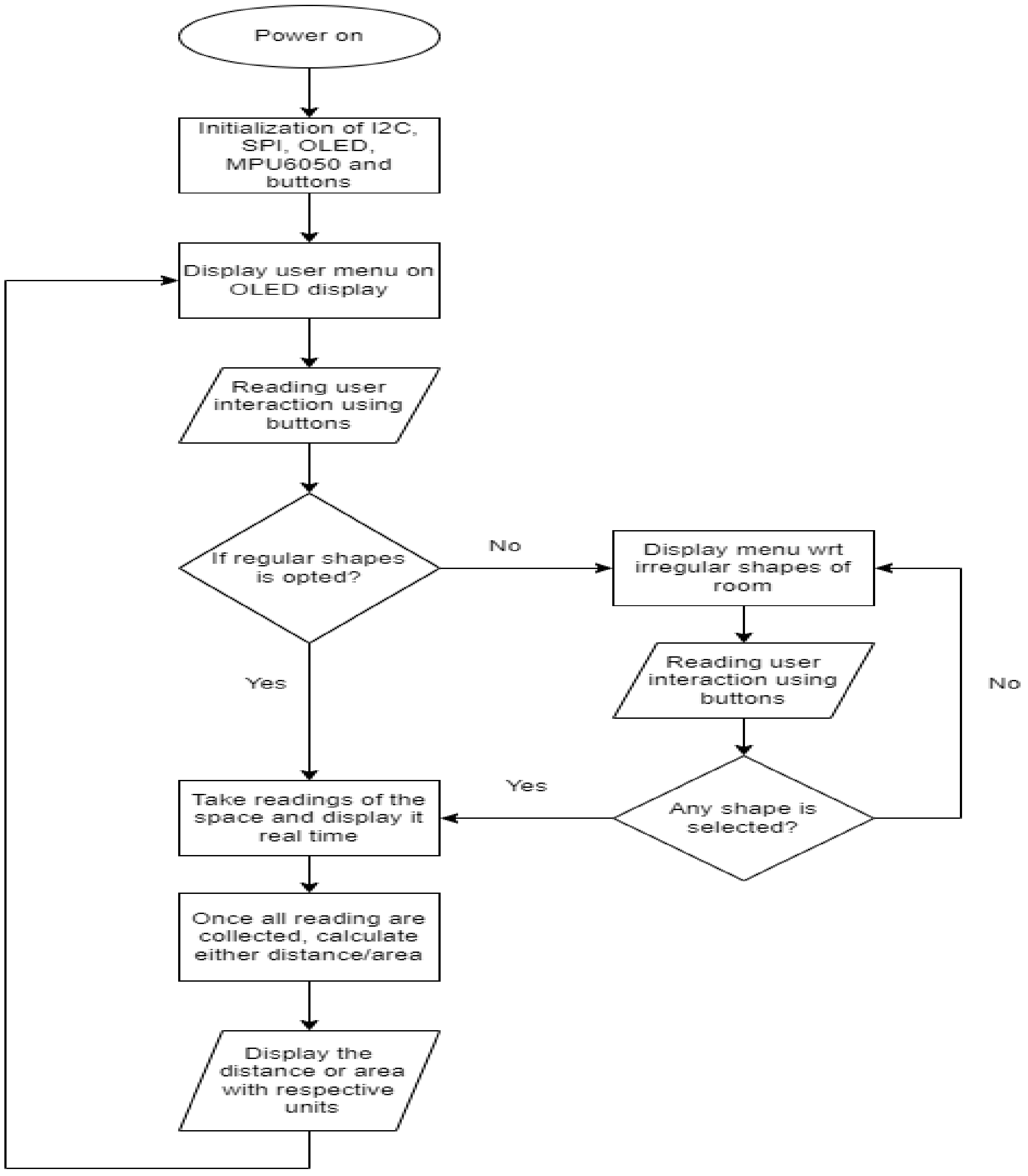

Power on
Initialization of I2C, SPI, OLED, MPU6050 and buttons
Display user menu on OLED display
Reading user interaction using buttons
If regular shapes is opted?
No
Display menu wrt irregular shapes of room
Reading user interaction using buttons
Yes
No
Yes
Any shape is selected?
Take readings of the space and display it real time
Once all reading are collected, calculate either distance/area
Display the distance or area with respective units

Figure 2.7.1: Software Flow chart

As defined in the above figure 2.7.1, the process begins with powering on the system, configuring I2C, SPI, OLED, MPU6050, and buttons, ensuring a stable hardware environment. Once initialized, the system displays a user menu on the OLED display, providing an interactive interface for users.

Users can navigate through the menu and make selections using the buttons. The system reads user inputs through buttons, facilitating a seamless and responsive user experience. Depending on user preferences, the system checks if regular shapes are selected for the space mapping process.

If regular shapes are chosen, the system proceeds to execute specific algorithms tailored for regular shapes. In cases where regular shapes are not selected, the system executes algorithms designed for mapping irregular room shapes.

The software collects readings from the space and displays them in real-time on the OLED, providing users with immediate feedback. Upon collecting all readings, the system calculates either distance or area based on user preferences.

The results, along with respective units, are displayed on the OLED, ensuring clarity and usability. This structured software design ensures a logical and intuitive flow, providing users with a seamless experience in interacting with the system and obtaining valuable information about their environment.

This enhanced software design optimizes the user interface and ensures efficient processing of inputs for a robust and user-friendly application.

2.8 Testing Process

The testing process was conducted with a meticulous and incremental approach. I initially interfaced the LIDAR using the I2C protocol, ensuring accurate readings in both centimetres and feet. Subsequently, I tackled the display interface using SPI bit-banging, addressing timing challenges related to the rise time and fall time of the clock signal. Microsecond delays were strategically introduced, resolving timing issues and facilitating seamless data transmission to the display.

Once the display functionality was confirmed, I proceeded to integrate the MPU 6050 using the I2C protocol, expanding the project's capabilities. The incorporation of buttons was also seamlessly executed. Each stage of the testing process underwent thorough verification to ensure the robustness and reliability of the overall system. This dynamic testing approach contributed to valuable insights and identified areas for refinement, leading to the successful completion of the project.

3 ENVIRONMENT AND TOOLCHAIN SETUP FOR RP2040 DEVELOPMENT

The development environment for RP2040 was set up through a series of essential steps to ensure a smooth coding, building and debugging process. The following steps detail the configuration process:

- ***Installed GNU Arm Embedded Toolchain:*** Downloaded and installed the GNU Arm Embedded Toolchain, which contained the Arm GCC compiler necessary for compiling C and C++ code for the RP2040.
- ***Installed MinGW-w64 GCC Tools:*** Installed MinGW-w64 GCC Tools to obtain compilers and linkers, essential for compiling elf2uf2 and pioasm tools during Pico project builds.
- ***Installed CMake:*** Installed CMake, a versatile tool that automated the build process, facilitating the generation of directory structures and files required by the Raspberry Pi Pico SDK.
- ***Installed Python:*** Python was installed as a prerequisite for the Pico SDK, used for scripting and automating certain build functions.
- ***Installed Git:*** Git simplified the process of downloading the Pico SDK. The Windows version included Git Bash, a useful shell. Configured VS Code to use Git Bash as the default shell.
- ***Downloaded Pico SDK and Examples:*** Obtained the Pico SDK, a comprehensive collection of tools and libraries that streamlined Pico and RP2040 development. Additionally, downloaded a set of C/C++ examples for practical demonstrations.
- ***Installed Visual Studio Code (VS Code):*** Downloaded and installed Visual Studio Code, a cross-platform text editor with configurable plugins. Selected the right plugins to transform VS Code into a powerful integrated development environment (IDE) with features like step-through debugging.
- ***Built Blink Example in VS Code:*** Verified the setup by building a simple project, such as the Blink example, in VS Code. This step ensured that code could be seamlessly compiled for the RP2040, providing insights into the development workflow within the chosen environment.

With these steps completed, the development environment was ready for RP2040 coding and project building using Visual Studio Code.

In addition to the fundamental steps outlined earlier, several additional tools were installed and configured to enhance the RP2040 debugging environment:

- ***Installed Git for Windows SDK:*** Recognizing the unique features of Git for Windows SDK, which includes tools facilitating program builds on Windows along with a package manager. This SDK was preferred over regular Git Bash due to its additional capabilities.
- ***Installed Packages with Pacman Package Manager:*** Leveraged the pacman package manager within Git for Windows SDK to install libraries and software, mimicking a Linux-like package installation process.
- ***Built OpenOCD for RP2040:*** Recognizing that the RP2040 was not officially supported in the mainline OpenOCD, a special branch of OpenOCD was built to enable communication with the RP2040 using GDB commands over JTAG or SWD.

- ***Built Picotool:*** Incorporated Picotool, a program designed to inspect RP2040 binaries and interact with the RP2040 in bootloader mode, into the development environment.
- ***Updated USB Driver for Picotool:*** Addressed the absence of a native Windows driver for USB communication with the Pico from picotool. A manual installation of the required driver was performed.
- ***Built and Uploaded Picoprobe:*** Configured and uploaded picoprobe, treated like any other Pico firmware, to the debugger Pico. This step facilitated interaction with the RP2040 in bootloader mode.
- ***Configured VS Code for Step-Through Debugging:*** With all the tools built and installed, configured Visual Studio Code (VS Code) to enable step-through debugging. This enhancement aimed to streamline the debugging process during code development.
- ***Started Debugging with Modified Code:*** Made a specific modification to the code by disabling USB serial and enabling UART serial. This alteration was crucial for successful debugging and interactively stepping through the code during the development process.

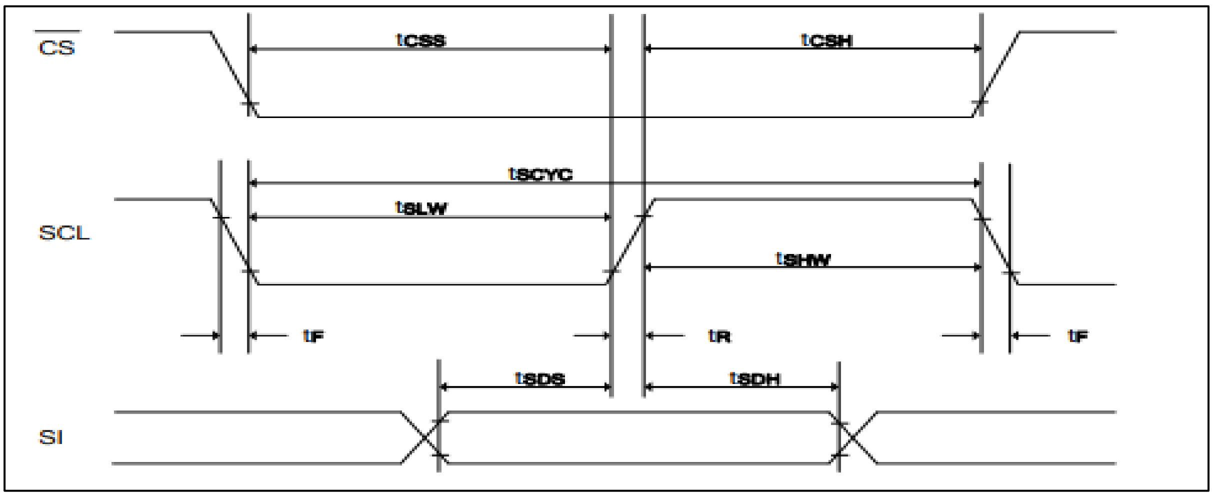
These supplementary steps further fortified the RP2040 development environment, providing advanced debugging capabilities and enabling seamless interaction with the RP2040 microcontroller.

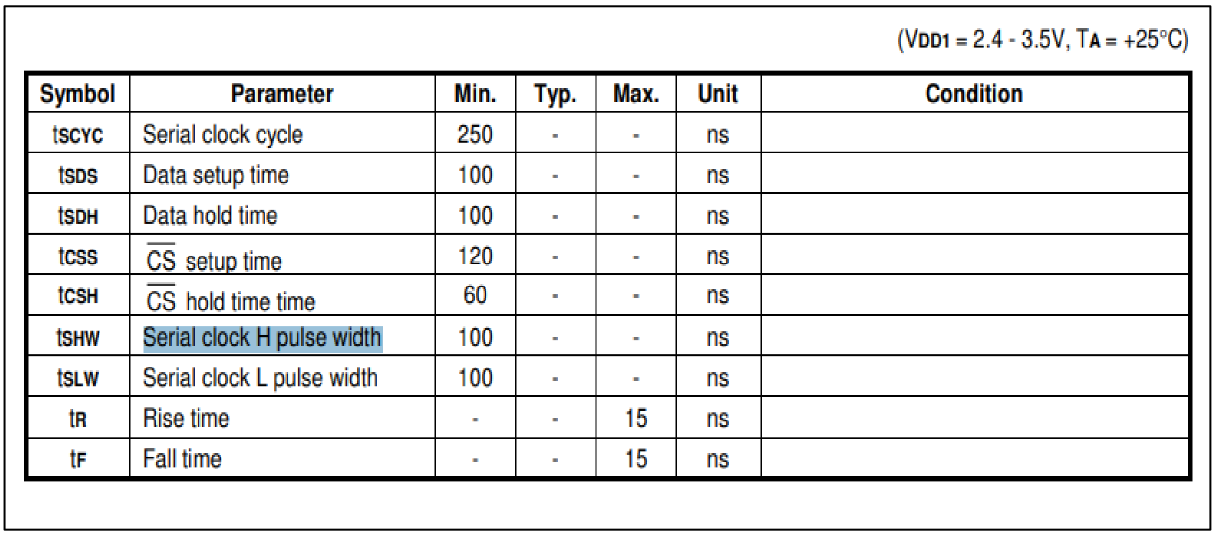
4 RESULTS AND ERROR ANALYSIS

The project outcomes align closely with the anticipated results, demonstrating the successful implementation of the SS Mapper system. However, a minor deviation of approximately 50 square centimetres and 1.7 square feet was observed in the calculated values. This discrepancy can be attributed to the fixed integer values obtained from the distance sensor. To enhance precision, a decision was made to type cast these values to double during calculations. This casting operation ensures that the final values are optimized for accuracy.

The RP2040 processor, based on the Arm Cortex M0+ architecture, operates with a clock speed of 125MHz, allowing each instruction to be executed in a swift 96ns, a verification corroborated by a logic analyzer. However, an issue surfaced during communication with the display device. Without any delay between clock pulses, the logic analyzer revealed a consistent transition between clock HIGH and LOW states. Subsequent analysis of the display device's datasheet unveiled a crucial requirement: the Serial Clock High (SCK-H) pulse width and Serial Clock Low (SCK-L) pulse width should be a minimum of 100ns each. Please find the relative information from below figures.

Source of images (Datasheet: SH1107V2.3.pdf)





(V_{DD1} = 2.4 - 3.5V, T_A = +25°C)

Symbol	Parameter	Min.	Typ.	Max.	Unit	Condition
t_{SCYC}	Serial clock cycle	250	-	-	ns	
t_{SDS}	Data setup time	100	-	-	ns	
t_{SDH}	Data hold time	100	-	-	ns	
t_{CSS}	$\overline{CS}$ setup time	120	-	-	ns	
t_{CSH}	$\overline{CS}$ hold time time	60	-	-	ns	
t_{SHW}	Serial clock H pulse width	100	-	-	ns	
t_{SLW}	Serial clock L pulse width	100	-	-	ns	
t_R	Rise time	-	-	15	ns	
t_F	Fall time	-	-	15	ns	

Figure 3.1 SPI Signal timing characteristics

In response to this discovery, a deliberate delay of approximately 10 microseconds was strategically introduced between the pulse's HIGH and LOW conditions. This critical adjustment rectified the communication anomaly, ensuring the display device could reliably receive and process the data. This incident underscores the importance of meticulous adherence to datasheet specifications, particularly in real-time systems, and highlights the significance of precise timing considerations in achieving seamless component integration. The successful resolution further attests to the effectiveness of systematic troubleshooting and validation processes in embedded systems development.

5 CONCLUSION

The SS Mapper project offered a comprehensive exploration of embedded systems, providing valuable insights ARM cortex M0+ and the RP2040 MCU. From the initial interfacing of LIDAR using the I2C protocol to overcoming timing challenges in display interfacing, each phase contributed to the project's overall success.

The seamless integration of the MPU 6050 with I2C and buttons showcased adaptability and effective problem-solving skills, essential in real-world embedded system design. The project's significance was further heightened by the incorporation of the RP2040 MCU, underscoring the importance of staying informed about the latest technologies in the field.

Throughout the development process, valuable insights were gained into the intricacies of optimizing sensor data for accurate calculations. The experience of identifying and addressing a SPI timing mismatch highlighted the significance of thorough datasheet analysis for robust system integration. Moving forward, the insights gained from overcoming these challenges contribute to a deeper understanding of embedded systems and pave the way for future enhancements to the SS Mapper project. The SS Mapper project stands as a testament to the fusion of traditional embedded system concepts with the contemporary capabilities of the RP2040 MCU.

6 FUTURE DEVELOPMENT IDEAS

Looking ahead, there are several areas where the project could evolve to enhance its functionality and maintainability:

6.1 Hardware area

Certainly, for future development and board upgrades, several key improvements can be considered to enhance the overall design:

6.1.1 Via Reduction

Implementing techniques to minimize the number of vias can help reduce signal interference, decrease manufacturing complexity, and enhance signal integrity throughout the board.

6.1.2 Signal Integrity Focus

Placing a greater emphasis on signal integrity by employing controlled impedance traces, differential pairs, and signal isolation techniques can significantly enhance the performance of high-frequency circuits, ensuring reliable data transmission and reception.

6.1.3 Cleaner Design

Striving for a cleaner design layout involves optimizing component placement, reducing unnecessary traces, and organizing the board layout more efficiently. This not only enhances aesthetics but also contributes to better electrical performance and manufacturability.

By prioritizing these upgrades in future iterations, the board's performance, reliability, and manufacturability can be substantially enhanced, paving the way for more efficient and robust electronic systems.

6.2 Software area

6.2.1 Unified I2C Module Implementation

It would be beneficial to streamline the communication process by implementing a single I2C module to handle interactions with both the LIDAR and MPU6050 sensors. This approach can simplify the code structure and improve overall efficiency.

6.2.2 State Machine Integration

To enhance user experience and project organization, I recommend incorporating a state machine. This would facilitate smooth transitions between different functionalities, such as moving from the menu to area calculation and vice versa.

6.2.3 Interrupts for Button Handling

Implementing interrupt-driven mechanisms for button inputs can significantly improve the responsiveness of the user interface. This approach ensures quicker and more efficient processing of button presses compared to traditional delay-based methods.

6.2.4 Minimize the Use of Delays

Instead of relying on explicit delays, consider adopting interrupt-based or event-driven techniques for precise timing. This adjustment can enhance the overall efficiency of the code and make it more responsive to external events.

6.2.5 Fixed-Point Arithmetic

Transitioning from floating-point to fully fixed-point arithmetic is a worthwhile optimization. This adjustment can improve computational efficiency and potentially reduce the overall code size, particularly in scenarios where floating-point precision is not crucial.

Addressing these points in future development iterations will contribute to the project's functionality, maintainability, and performance.

7 ACKNOWLEDGEMENTS

We extend our sincere gratitude to Prof. Linden McClure, the course instructor, for imparting valuable knowledge and guidance throughout the duration of the course. His support and insights have been instrumental in the successful completion of this project.

Special thanks are due to our flat mates for their unwavering support and assistance in managing household responsibilities, allowing us to focus on the project with dedication.

Lastly, we express our acknowledgment to the authors and organizations whose contributions and detailed documentation, including datasheets, have served as crucial references for our project. Their work has enriched our understanding and provided essential information for the successful implementation of our ideas.

Our heartfelt thanks to everyone who has contributed to our learning and project development journey.

8 REFERENCES

1. Toolchain setup for RP2040:
   https://www.digikey.com/en/maker/projects/raspberry-pi-pico-and-rp2040-cc-part-1-blink-and-vs-code/7102fb8bca95452e9df6150f39ae8422
2. Debug environment setup:
   https://www.digikey.com/en/maker/projects/raspberry-pi-pico-and-rp2040-cc-part-2-debugging-with-vs-code/470abc7efb07432b82c95f6f67f184c0
3. Example codes referred from here:
   GitHub - raspberrypi/pico-sdk
4. Information about I2C Protocol:
   https://www.analog.com/en/technical-articles/i2c-primer-what-is-i2c-part-1.html
5. Information about SPI protocol:
   https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html
6. Raspberry Pi Documentation:
   https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html
7. PCB board design tutorial:
   https://youtube.com/@AutodeskEAGLE?si=fNNkUsEq2Z6lnUpM
8. Referred existing designs around RP2040:
   https://www.adafruit.com/
   https://www.sparkfun.com/

9 APPENDICES

9.1 Appendix - Bill of Materials

Part Detail	Top Designator	Qty	Source	Ext. Price
PSMD050	F1	7	JLCPCB	$0.27
CL05A106MQ5NUNC	C7,C8	20	JLCPCB	$0.0880
CL05A105KA5NQNC	C6	20	JLCPCB	$0.0720
0402CG150J500NT	C11,C13	20	JLCPCB	$0.0180
TPAP2112K-3.3TRG1	U1	7	JLCPCB	$0.81
KMR231GLFS	S2,S1	10	JLCPCB	$4.26
0402WGF1001TCE	R10,R11,R15	20	JLCPCB	$0.01
CL05B104KO5NNNC	C9,C12,C14,C15,C16,C17,C18,C19,C29	50	JLCPCB	$0.05
0402WGF2000TCE	R14	20	JLCPCB	$0.0100
0402WGF5101TCE	R1,R3	20	JLCPCB	$0.0100
W25Q128JVPIQ	U4	5	JLCPCB	$4.57
RP2040	U2	5	JLCPCB	$5.07
0402WGF1002TCE	R2,R16	20	JLCPCB	$0.0100
KT-0603R	LED1	20	JLCPCB	$0.1040
0402WGF2201TCE	R4,R6	20	JLCPCB	$0.0100
USB-TYPE-C-019	J5	7	JLCPCB	$0.30
CL05A225MQ5NSNC	C20,C21,C22,C23	25	JLCPCB	$0.06
0402WGF270JTCE	R12,R13	20	JLCPCB	$0.0100
BAT60AE6327	D2	5	JLCPCB	$0.83
8W12000010	Y1	5	JLCPCB	$3.80
WS2812B-2020	U3	7	JLCPCB	$0.66
LIDAR	TFLuna	1	AMAZON	$22.99
MPU6050		1	AMAZON	$4.00
OLED DISPLAY	Waveshare	1	AMAZON	$18.99
PCB		5	JLCPCB	$2.00
TOTAL				**$69.852**

9.2 Appendix – Schematics

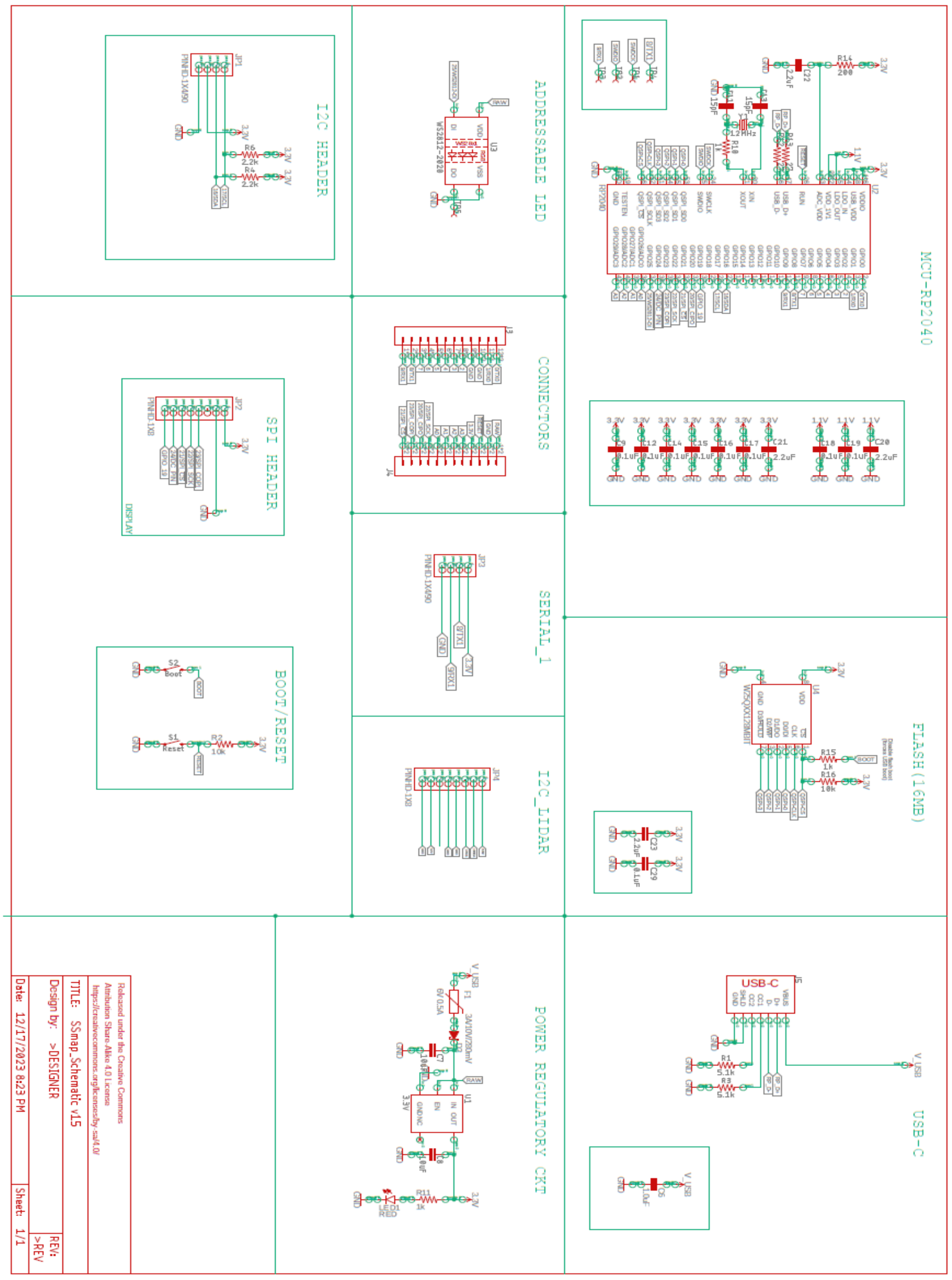


Figure 4: Project Schematics

9.3 Appendix - Firmware Source Code

main.c

```
/******************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file main.c
 * @brief Main file for the SS Mapper application.
 * @author Jithendra H S
 * @date 2023-12-15
 */
#include <stdio.h>
#include "pico/stdlib.h"
#include <stdlib.h>
#include "spi_code.h"
#include "i2c_code.h"
#include "lidar.h"
#include "area.h"
#include "button.h"
#include "user_interface.h"
#include "mpu6050.h"

/**
 * @brief Main function for the SS Mapper application.
 *
 * This function initializes the serial port, I2C module, SPI module, OLED
display,
 * MPU6050 sensor, buttons, and starts the user interface for the SS Mapper
application.
 * The application then enters the main loop where it continuously checks for
button
 * presses and updates the user interface accordingly.
 *
 * @return 0 upon successful execution.
 */
int main() {
    // Initialize chosen serial port
    stdio_init_all();
    printf(" !!!!!!!!!!!!!!!!!! SS Mapper started !!!!!!!!!!!!!!!!!!\n");
    // Initialize I2C module
    I2C_Module_Init();

    // Initialize SPI module using bit-banging
    SPI_Module_Init_BIT_BANGING();

    // Initialize OLED display
    OLED_init();
    OLED_Clear();

    // Reset MPU6050 sensor
    resetMPU6050(i2c1);

    // Initialize buttons
    Button_Init();

    // Start the user interface
    user_interface();

    return 0;
}
```

i2c_code.h

```
/******************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file i2c_code.h
 * @brief Header file for I2C communication functions.
 * @author Jithendra H S
 * @date December 15, 2023
*/
#include "stdint.h"
#include "hardware/i2c.h"
#include "pico/stdlib.h"

#define I2C_SDA (16)
#define I2C_SCK (17)
```

```c
#define I2C1_SDA_PIN (6)
#define I2C1_SCL_PIN (7)

/**
 * @brief Write data to the specified register over I2C.
 *
 * @param i2c       I2C instance.
 * @param addr      I2C device address.
 * @param reg       Register address to write to.
 * @param buf       Pointer to the data buffer.
 * @param nbytes    Number of bytes to write.
 * @return int      Number of bytes read.
 */
int reg_write(i2c_inst_t *i2c,
              const uint addr,
              const uint8_t reg,
              uint8_t *buf,
              const uint8_t nbytes);

/**
 * @brief Read data from the specified register over I2C.
 *
 * @param i2c       I2C instance.
 * @param addr      I2C device address.
 * @param reg       Register address to read from.
 * @param buf       Pointer to the data buffer.
 * @param nbytes    Number of bytes to read.
 * @return int      Number of bytes read.
 */
int reg_read(i2c_inst_t *i2c,
             const uint addr,
             const uint8_t reg,
             uint8_t *buf,
             const uint8_t nbytes);

/**
 * @brief Initialize I2C modules and pins.
 */
void I2C_Module_Init();
```

i2c_code.c

```c
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file i2c_code.c
 * @brief Source file for I2C communication functions.
 * @author Jithendra H S
 * @date December 15, 2023
*/
#include "stdio.h"
#include "i2c_code.h"

/**
 * @brief Write data to the specified register over I2C.
 *
 * This function writes data to a specific register of an I2C device.
 *
 * @param i2c       I2C instance.
 * @param addr      I2C device address.
 * @param reg       Register address to write to.
 * @param buf       Pointer to the data buffer.
 * @param nbytes    Number of bytes to write.
 * @return int      Number of bytes written.
 */
int reg_write(i2c_inst_t *i2c,
              const uint addr,
              const uint8_t reg,
              uint8_t *buf,
              const uint8_t nbytes) {
    // Initialize the number of bytes written
    int num_bytes_written = 0;

    // Create a message buffer including the register address
    uint8_t msg[nbytes + 1];

    // Check if the number of bytes is valid
    if (nbytes < 1) {
        return 0; // Return 0 if an invalid number of bytes is provided
    }
    // Append the register address to the front of the data packet
    msg[0] = reg;
    for (int i = 0; i < nbytes; i++) {
        msg[i + 1] = buf[i];
    }

    // Write data to register(s) over I2C
    num_bytes_written = i2c_write_blocking(i2c, addr, msg, (nbytes + 1),
false);

    return num_bytes_written; // Return the number of bytes written
}


/**
 * @brief Read data from the specified register(s) over I2C.
 *
 * This function reads data from a specific register of an I2C device.
 *
 * @param i2c       I2C instance.
 * @param addr      I2C device address.
 * @param reg       Register address to read from.
 * @param buf       Pointer to the data buffer.
 * @param nbytes    Number of bytes to read.
 * @return int      Number of bytes read.
 */
int reg_read(i2c_inst_t *i2c,
             const uint addr,
             const uint8_t reg,
             uint8_t *buf,
             const uint8_t nbytes) {
    // Initialize the number of bytes read
    int num_bytes_read = 0;

    // Check if the number of bytes is valid
    if (nbytes < 1) {
        return 0; // Return 0 if an invalid number of bytes is provided
    }

    // Read data from register(s) over I2C
    i2c_write_blocking(i2c, addr, &reg, 1, true);
    num_bytes_read = i2c_read_blocking(i2c, addr, buf, nbytes, false);

    return num_bytes_read; // Return the number of bytes read
}


/**
```

```
 * @brief Initialize the I2C modules and pins.
 *
 * This function initializes the I2C modules and pins for communication.
 * It sets the I2C0 port at 200 kHz and I2C1 port at 100 kHz.
 * It configures the corresponding SDA and SCK pins for I2C functionality.
 */
void I2C_Module_Init() {
    // Initialize I2C0 port at 200 kHz
    i2c_init(i2c0, 200 * 1000);

    // Initialize I2C0 pins
    iobank0_hw->io[I2C_SDA].ctrl = GPIO_FUNC_I2C <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[I2C_SCK].ctrl = GPIO_FUNC_I2C <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;

    // Initialize I2C1 port at 100 kHz
    i2c_init(i2c1, 100 * 1000);

    // Initialize I2C1 pins
    iobank0_hw->io[I2C1_SDA_PIN].ctrl = GPIO_FUNC_I2C <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[I2C1_SCL_PIN].ctrl = GPIO_FUNC_I2C <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
}
```

menu.h

```
/****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file menu.h
 * @brief Header file for SS Mapper menu functions.
 * @author Jithendra H S
 * @date 2023-12-15
*/
#include "oled.h"
```

```
// Maximum length for shape names
#define SHAPE_NAME_MAX_LENGTH 16

// External declarations for shape arrays
extern char shapes[][SHAPE_NAME_MAX_LENGTH];
extern char irr_shapes[][SHAPE_NAME_MAX_LENGTH];

// Function declarations

/**
 * @brief Displays the main menu on the OLED screen.
 *
 * This function displays the main menu on the OLED screen, including shape
options and
 * the cursor indicating the current selection. It also provides navigation
instructions.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 */
void menu(UBYTE *BlackImage);

/**
 * @brief Moves the cursor in the main menu and returns the selected shape.
 *
 * This function updates the cursor position in the main menu and returns the
selected shape
 * based on the new cursor position.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 * @return A pointer to the selected shape name.
 */
char *move_cursor(UBYTE *BlackImage);

/**
 * @brief Moves the cursor in the irregular menu and returns the selected
shape.
 *
 * This function updates the cursor position in the irregular menu and returns
the selected shape
 * based on the new cursor position.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 * @return A pointer to the selected irregular shape name.
 */
char *move_cursor_irr_menu(UBYTE *BlackImage);

/**
 * @brief Retrieves the current cursor position in the main menu.
 *
 * This function returns the current cursor position in the main menu,
allowing other
 * functions to query and use the cursor position information.
 *
 * @return The current cursor position in the main menu.
 */
int8_t get_cursor_pos();

/**
 * @brief Sets the cursor position in the main menu.
 *
 * This function sets the cursor position in the main menu to the specified
value.
 *
 * @param pos The new cursor position to set.
 */
void put_cursor_pos(int8_t pos);

/**
 * @brief Displays the irregular menu on the OLED screen.
 *
 * This function displays the irregular menu on the OLED screen, including the
shapes and
 * the cursor indicating the current selection. It also provides navigation
instructions.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 */
void irr_menu(UBYTE *BlackImage);
```

menu.c

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file menu.c
 * @brief Menu functions for SS Mapper application.
 * @author Jithendra H S
 * @date 2023-12-15
*/
#include "menu.h"
#include "pico/stdlib.h"
#include <string.h>

/**
 * @brief X-coordinate for the start position of shapes in the menu.
 */
#define SHAPES_START_X (24)

/**
 * @brief Y-coordinate for the start position of shapes in the menu.
 */
#define SHAPES_START_Y (35)

/**
 * @brief X-coordinate for the start position of the cursor in the menu.
 */
#define CURSOR_START_X (0)

/**
 * @brief Array of strings representing different shapes.
 */
char shapes[][16] = {"Distance", "Circle", "Rectangle", "Triangle", "Irregular
menu"};

/**
 * @brief Array of strings representing shapes in the irregular menu.
 */
char irr_shapes[][16] = {"shape1", "shape2", "shape3", "shape4", "shape5"};

/**
 * @brief Cursor position in the main menu.
 */
static volatile int8_t cursor_pos = 0;

/**
 * @brief Cursor position in the irregular menu.
 */
static volatile int8_t cursor_pos_irr_menu = 0;

/**
 * @brief Displays the main menu on the OLED screen.
 *
 * This function initializes the display with shapes, including the cursor for
shape selection.
```

```
 * It also provides instructions for navigation and selection using different
button presses.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 */
void menu(UBYTE *BlackImage) {
    // Initialize the OLED display
    Paint_NewImage(BlackImage, OLED_WIDTH, OLED_HEIGHT, 0, BLACK);
    Paint_SetScale(2);

    Paint_SelectImage(BlackImage);
    Paint_Clear(BLACK);

    // Display the application name
    Paint_DrawString_EN(10, 0, "SS Mapper", &Font16, WHITE, BLACK);
    Paint_DrawString_EN(0, 18, "--------------------", &Font12, WHITE, BLACK);

    // Display shapes with checkboxes
    for (int i = 0; i < sizeof(shapes) / sizeof(shapes[0]); i++) {
        Paint_DrawString_EN(CURSOR_START_X, SHAPES_START_Y + (i *
Font12.Height), "[ ]", &Font12, WHITE, BLACK);
        Paint_DrawString_EN(SHAPES_START_X, SHAPES_START_Y + (i *
Font12.Height), shapes[i], &Font12, WHITE, BLACK);
    }

    // Display cursor for shape selection
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y + (cursor_pos *
Font12.Height), "+", &Font12, WHITE, BLACK);

    // Display navigation instructions
    Paint_DrawString_EN(0, 112, "*Press Yellow to Navigate", &Font8, WHITE,
BLACK);
    Paint_DrawString_EN(0, 120, "*Press Red to select", &Font8, WHITE, BLACK);

    // Update the OLED display
    OLED_Display(BlackImage);
}

/**
 * @brief Moves the cursor to the next shape in the main menu.
 *
 * This function updates the display to move the cursor to the next shape in
the main menu.
 * It ensures that the cursor wraps around to the beginning if it reaches the
end.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 * @return A pointer to the string representing the currently selected shape.
 */
char *move_cursor(UBYTE *BlackImage) {
    // Clear the previous cursor position
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y + (cursor_pos * 
Font12.Height), " ", &Font12, WHITE, BLACK);

    // Move to the next shape in the main menu
    cursor_pos++;
    cursor_pos = cursor_pos % (sizeof(shapes) / sizeof(shapes[0]));

    // Display the cursor at the new position
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y + (cursor_pos * 
Font12.Height), "+", &Font12, WHITE, BLACK);

    // Update the OLED display
    OLED_Display(BlackImage);

    // Return the name of the currently selected shape
    return shapes[cursor_pos];
}

/**
 * @brief Moves the cursor to the next shape in the irregular menu.
 *
 * This function updates the display to move the cursor to the next shape in 
the irregular menu.
 * It ensures that the cursor wraps around to the beginning if it reaches the 
end.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 * @return A pointer to the string representing the currently selected shape 
in the irregular menu.
 */
char *move_cursor_irr_menu(UBYTE *BlackImage) {
    // Clear the previous cursor position
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y + 
(cursor_pos_irr_menu * Font12.Height), " ", &Font12, WHITE, BLACK);

    // Move to the next shape in the irregular menu
    cursor_pos_irr_menu++;
    cursor_pos_irr_menu = cursor_pos_irr_menu % (sizeof(irr_shapes) / 
sizeof(irr_shapes[0]));

    // Display the cursor at the new position
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y + 
(cursor_pos_irr_menu * Font12.Height), "+", &Font12, WHITE, BLACK);

    // Update the OLED display
    OLED_Display(BlackImage);

    // Return the name of the currently selected shape in the irregular menu
    return irr_shapes[cursor_pos_irr_menu];
}


/**
 * @brief Retrieves the current cursor position in the main menu.
 *
 * This function returns the current cursor position in the main menu,
allowing other
 * functions to query and use the cursor position information.
 *
 * @return The current cursor position in the main menu.
 */
int8_t get_cursor_pos() {
    return cursor_pos;
}

/**
 * @brief Sets the cursor position in the main menu.
 *
 * This function sets the cursor position in the main menu to the specified
value.
 *
 * @param pos The new cursor position to set.
 */
void put_cursor_pos(int8_t pos) {
    cursor_pos = pos;
}

/**
 * @brief Displays the irregular menu on the OLED screen.
 *
 * This function displays the irregular menu on the OLED screen, including the
shapes and
 * the cursor indicating the current selection. It also provides navigation
instructions.
 *
 * @param BlackImage Pointer to the image buffer for the OLED display.
 */
void irr_menu(UBYTE *BlackImage) {
    // Reset cursor position in the irregular menu
    cursor_pos_irr_menu = 0;

    // Clear OLED display
    OLED_Clear();
    // Use memset to set all values to 0x00
    memset(BlackImage, 0x00, OLED_IMAGE_SIZE);

    // Display the title
    Paint_DrawString_EN(10, 0, "SS Mapper", &Font16, WHITE, BLACK);
    Paint_DrawString_EN(0, 18, "--------------------", &Font12, WHITE, BLACK);

    // Display shapes and cursor in the irregular menu
    for (int i = 0; i < sizeof(irr_shapes) / sizeof(irr_shapes[0]); i++) {
        Paint_DrawString_EN(CURSOR_START_X, SHAPES_START_Y + (i *
Font12.Height), "[ ]", &Font12, WHITE, BLACK);
        Paint_DrawString_EN(SHAPES_START_X, SHAPES_START_Y + (i *
Font12.Height), irr_shapes[i], &Font12, WHITE, BLACK);
    }

    // Display the cursor at the initial position
    Paint_DrawString_EN(CURSOR_START_X + 8, SHAPES_START_Y +
(cursor_pos_irr_menu * Font12.Height), "+", &Font12, WHITE, BLACK);

    // Display navigation instructions
    Paint_DrawString_EN(0, 112, "*Press Yellow to Navigate", &Font8, WHITE,
BLACK);
    Paint_DrawString_EN(0, 120, "*Press Red to select", &Font8, WHITE, BLACK);

    // Update the OLED display
    OLED_Display(BlackImage);
}
```

area.h

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file area.h
 * @brief Functions for calculating areas of different shapes and distances.
 * @author Jithendra H S
 * @date December 15, 2023
 */
#include "stdint.h"
```

```
#include "menu.h"
#include "string.h"

// Structure to store the result of area calculations
typedef struct double_array double_array;

struct double_array{
    double result[2];
};

/**
 * @brief Calculate the area based on the specified shape.
 *
 * @param shape         A string identifier for the shape to be calculated.
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area(char *shape, UBYTE *BlackImage);

/**
 * @brief Calculate the area based on the specified irregular shape.
 *
 * @param shape         A string identifier for the irregular shape to be
calculated.
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_irr_shape(char *shape, UBYTE *BlackImage);

/**
 * @brief Capture the distance using LIDAR sensor.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return uint16_t     The captured distance in centimeters.
 */
uint16_t capture_distance(UBYTE *BlackImage);

/**
 * @brief Calculate the distance and convert it to both centimeters and feet.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated distance in
centimeters
 *                      (result[0]) and feet (result[1]).
 */
double_array calculate_distance(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a circle based on the captured diameter.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_circle(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a rectangle based on the measured width and
length.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_rectangle(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a triangle based on the measured distances at
three corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_triangle(UBYTE *BlackImage);

/**
 * @brief Calculate the area of an irregular shape based on the user's
selection.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                       (result[0]) and square feet (result[1]).
 */
double_array calculate_area_Irregular(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a specific irregular shape (Shape 1) based on
the measured
 *        distances at six corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                       (result[0]) and square feet (result[1]).
 */
double_array calculate_area_shape1(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a specific irregular shape (Shape 2) based on
the measured
 *        distances at eight corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                       (result[0]) and square feet (result[1]).
 */
double_array calculate_area_shape2(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a specific irregular shape (Shape 3) based on
the measured
 *        distances at eight corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                       (result[0]) and square feet (result[1]).
 */
double_array calculate_area_shape3(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a specific irregular shape (Shape 4) based on
the measured
```

```
 *        distances at twelve corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_shape4(UBYTE *BlackImage);

/**
 * @brief Calculate the area of a specific irregular shape (Shape 5) based on
the measured
 *        distances at three corners.
 *
 * @param BlackImage    Pointer to the image data.
 *
 * @return double_array A structure containing the calculated area in square
centimeters
 *                      (result[0]) and square feet (result[1]).
 */
double_array calculate_area_shape5(UBYTE *BlackImage);
```

area.c

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file area.c
 * @brief Functions for calculating and displaying area based on various
shapes.
 * @author Jithendra H S
 * @date 2023-12-15
 */
#include "area.h"
#include "string.h"
#include "stdio.h"
#include "pico/stdlib.h"
```

```
#include <math.h>
#include "lidar.h"
#include "mpu6050.h"

#define FOOT_IN_CM (30.48)
#define SQUARE_FEET (0.0010764)

/**
 * @brief Calculate area based on the selected shape
 *
 * This function determines the selected shape and calls the corresponding
area calculation function.
 * The result is returned as a double_array structure.
 *
 * @param shape A string indicating the selected shape
 * @param BlackImage A pointer to the image cache
 *
 * @return A double_array structure containing the calculated area values
 */
double_array calculate_area(char * shape, UBYTE *BlackImage){
    // Check the selected shape and call the corresponding area calculation
function
    if(strcmp(shape, "Distance") == 0){
        return calculate_distance(BlackImage);
    }else if(strcmp(shape, "Circle") == 0){
        return calculate_area_circle(BlackImage);
    }else if(strcmp(shape, "Rectangle") == 0){
        return calculate_area_rectangle(BlackImage);
    }else if(strcmp(shape, "Triangle") == 0){
        return calculate_area_triangle(BlackImage);
    }else if(strcmp(shape, "Irregular menu") == 0){
        return calculate_area_Irregular(BlackImage);
    }else{
        printf("Unknown command\n\r");
    }
}


/**
 * @brief Calculate area for irregular shapes based on the selected shape
 *
 * This function determines the selected irregular shape and calls the
corresponding area calculation function.
 * The result is returned as a double_array structure.
 *
 * @param shape A string indicating the selected irregular shape
 * @param BlackImage A pointer to the image cache
 *
 * @return A double_array structure containing the calculated area values for
the irregular shape
 */
double_array calculate_area_irr_shape(char * shape, UBYTE *BlackImage){
    // Check the selected irregular shape and call the corresponding area
calculation function
    if(strcmp(shape, "shape1") == 0){
        return calculate_area_shape1(BlackImage);
    }else if(strcmp(shape, "shape2") == 0){
        return calculate_area_shape2(BlackImage);
    }else if(strcmp(shape, "shape3") == 0){
        return calculate_area_shape3(BlackImage);
    }else if(strcmp(shape, "shape4") == 0){
        return calculate_area_shape4(BlackImage);
    }else if(strcmp(shape, "shape5") == 0){
        return calculate_area_shape5(BlackImage);
    }else{
        printf("Unknown command\n\r");
    }
}

/**
 * @brief Capture and display distance from the LiDAR sensor along with device
orientation
 *
 * This function continuously captures distance data from the LiDAR sensor and
displays it on the OLED screen.
 * Additionally, it reads and displays the device's X and Y orientation
angles.
 * The function returns the captured distance when the user presses the red
button (GPIO11).
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return The captured distance from the LiDAR sensor
 */
uint16_t capture_distance(UBYTE *BlackImage){
    uint16_t distance = 0;
    int16_t co_ordinate[3] = {0};

    // Display distance-related information on the OLED screen
    Paint_DrawString_EN(0, 12, "Distance:", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(24, 24, "cm", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(76, 24, "Feet", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(0, 60, "Device orientation:", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(0, 72, "X :", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(0, 84, "Y :", &Font12, WHITE, BLACK);
    Paint_DrawString_EN(0, 120, "*Press Red to Read", &Font8, WHITE, BLACK);
    // Continuously capture and display distance data
    while(true){
        distance = read_lidar();
        read_tilt_angle(co_ordinate);

        // Display captured distance in both centimeters and feet
        Paint_DrawString_EN(24, 36, "          ", &Font12, WHITE, BLACK);
        Paint_DrawNum(24, 36, distance, &Font12, 0, WHITE, BLACK);
        Paint_DrawString_EN(76, 36, "          ", &Font12, WHITE, BLACK);
        Paint_DrawNum(76, 36, (double)distance/FOOT_IN_CM, &Font12, 2, WHITE,
BLACK);

        // Display device orientation angles
        Paint_DrawString_EN(21, 72, "          ", &Font12, WHITE, BLACK);
        Paint_DrawNum(21, 72, co_ordinate[0], &Font12, 0, WHITE, BLACK);
        Paint_DrawString_EN(21, 84, "          ", &Font12, WHITE, BLACK);
        Paint_DrawNum(21, 84, co_ordinate[1], &Font12, 0, WHITE, BLACK);

        // Update the OLED display
        OLED_Display(BlackImage);

        // Read the state of GPIO11
        bool gpio11_state = gpio_get(11);

        // Return the captured distance if the user presses the red button
        if (!gpio11_state) {
            printf("Captured distance\n\r");
            sleep_ms(200);
            return distance;
        }

        // Wait for a short duration before the next iteration
        sleep_ms(100);
    }
}
/**
 * @brief Calculate and display distance and converted feet value
 *
 * This function calculates the distance using the `capture_distance` function
and converts it to feet.
 * It then displays the measured distance and converted feet value on the OLED
screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated distance in centimeters and
converted feet value
 */
double_array calculate_distance(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Display a message for distance measurement
    Paint_DrawString_EN(0, 0, "Measuring Distance", &Font8, WHITE, BLACK);

    // Capture distance using the capture_distance function
    double output = (double)capture_distance(BlackImage);

    // Store the calculated distance and converted feet value in the result
structure
    result.result[0] = output;
    result.result[1] = output / FOOT_IN_CM;

    // Return the result structure
    return result;
}

/**
 * @brief Calculate and display area of a circle
 *
 * This function calculates the area of a circle using the `capture_distance`
function to measure the diameter.
 * It then displays the measured diameter, calculated area, and converted area
value on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_circle(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Display a message for diameter measurement
    Paint_DrawString_EN(0, 0, "Measuring Diameter", &Font8, WHITE, BLACK);

    // Capture diameter using the capture_distance function
    uint16_t diameter = capture_distance(BlackImage);

    // Calculate the area of the circle
    double output = (double)(M_PI * diameter * diameter / 4);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;
    // Return the result structure
    return result;
}

/**
 * @brief Calculate and display area of a rectangle
 *
 * This function measures the width and length using the `capture_distance`
function,
 * calculates the area of the rectangle, and displays the measured values and
the area
 * on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_rectangle(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Display a message for width measurement
    Paint_DrawString_EN(0, 0, "Measuring Width", &Font8, WHITE, BLACK);

    // Measure the width using the capture_distance function
    uint16_t width = capture_distance(BlackImage);

    // Sleep for 500 ms to allow the user to reposition the sensor
    sleep_ms(500);

    // Display a message for length measurement
    Paint_DrawString_EN(0, 0, "Measuring Length", &Font8, WHITE, BLACK);

    // Measure the length using the capture_distance function
    uint16_t length = capture_distance(BlackImage);

    // Calculate the area of the rectangle
    double output = (double)(length * width);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}
```

```
/**
 * @brief Calculate and display area of a triangle
 *
 * This function measures distances at three corners using the
`capture_distance` function,
 * applies Heron's formula to calculate the area of the triangle, and displays
the measured values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_triangle(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store distances at three corners of the triangle
    uint16_t corner[3] = {0};

    // Index variable for looping through corners
    uint8_t index = 0;

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    while(index < 3){
        Paint_DrawNum(112, 0, index + 1, &Font8, 0, WHITE, BLACK);
        corner[index] = capture_distance(BlackImage);
        index++;
    }

    // Using Heron's formula to calculate the area of the triangle
    double s = (corner[0] + corner[1] + corner[2]) / 2.0;
    double output = (double)sqrt(s * (s - corner[0]) * (s - corner[1]) * (s -
corner[2]));

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}


/**
```

```
 * @brief Calculate and display area of an irregular shape
 *
 * This function displays an irregular shape menu on the OLED screen,
 * allows the user to select a specific irregular shape, and calculates
 * and displays the area of the selected shape using the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_Irregular(UBYTE *BlackImage){
    // Display the irregular shape menu on the OLED screen
    irr_menu(BlackImage);

    // Initialize the selected shape to the first shape in the menu
    char *shape = irr_shapes[0];

    // Sleep for 300 ms to allow the user to read the menu
    sleep_ms(300);

    while(true){
        // Read the state of GPIO10
        bool gpio10_state = gpio_get(10);
        // Read the state of GPIO11
        bool gpio11_state = gpio_get(11);

        // Perform actions based on the GPIO10 state
        if (!gpio10_state) {
            sleep_ms(200);
            // Allow the user to move the cursor and select a shape
            shape = move_cursor_irr_menu(BlackImage);
            printf("GPIO10 is high (pressed)! and shape %s\n\r", shape);
        }

        // Perform actions based on the GPIO11 state
        if (!gpio11_state) {
            sleep_ms(200);
            // Clear the OLED screen and set all image values to 0x00
            OLED_Clear();
            memset(BlackImage, 0x00, OLED_IMAGE_SIZE);
            OLED_Display(BlackImage);
            // Calculate and return the area of the selected irregular shape
            return calculate_area_irr_shape(shape, BlackImage);
        }
    }
}

/**
```

```
 * @brief Calculate and display area of a specific irregular shape (Shape 1)
 *
 * This function measures distances at six corners of a specific irregular
shape,
 * applies a specific formula to calculate the area, and displays the measured
values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_shape1(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store measurements at six corners of the shape
    int16_t measurement[6] = {0};

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    for(int i = 0; i < sizeof(measurement) / sizeof(measurement[0]); i++){
        Paint_DrawNum(112, 0, i + 1, &Font8, 0, WHITE, BLACK);
        measurement[i] = capture_distance(BlackImage);
    }

    // Using a specific formula to calculate the area of the shape
    double sub_area1 = measurement[0] * measurement[5];
    double sub_area2 = measurement[2] * measurement[3];
    double output = (double)(sub_area1 + sub_area2);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}


/**
 * @brief Calculate and display area of a specific irregular shape (Shape 2)
 *
 * This function measures distances at eight corners of a specific irregular
shape (Shape 2),
 * applies a specific formula to calculate the area, and displays the measured
values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_shape2(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store measurements at eight corners of the shape
    int16_t measurement[8] = {0};

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    for(int i = 0; i < sizeof(measurement) / sizeof(measurement[0]); i++){
        Paint_DrawNum(112, 0, i + 1, &Font8, 0, WHITE, BLACK);
        measurement[i] = capture_distance(BlackImage);
    }

    // Using a specific formula to calculate the area of the shape
    double sub_area1 = measurement[0] * measurement[7];
    double sub_area2 = measurement[2] * measurement[3];

    double output = (double)(sub_area1 + sub_area2);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}

/**
 * @brief Calculate and display area of a specific irregular shape (Shape 3)
 *
 * This function measures distances at eight corners of a specific irregular
shape (Shape 3),
 * applies a specific formula to calculate the area, and displays the measured
values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_shape3(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store measurements at eight corners of the shape
    int16_t measurement[8] = {0};

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    for(int i = 0; i < sizeof(measurement) / sizeof(measurement[0]) ; i++){
        Paint_DrawNum(112, 0, i + 1, &Font8, 0, WHITE, BLACK);
        measurement[i] = capture_distance(BlackImage);
    }

    // Using a specific formula to calculate the area of the shape
    double sub_area1 = measurement[0] * measurement[1];
    double sub_area2 = (measurement[0] - measurement[2]) * (measurement[7] -
measurement[1] - measurement[5]);
    double sub_area3 = measurement[5] * measurement[6];

    double output = (double)(sub_area1 + sub_area2 + sub_area3);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}

/**
 * @brief Calculate and display area of a specific irregular shape (Shape 4)
 *
 * This function measures distances at twelve corners of a specific irregular
shape (Shape 4),
 * applies a specific formula to calculate the area, and displays the measured
values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_shape4(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store measurements at twelve corners of the shape
    int16_t measurement[12] = {0};

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    for(int i = 0; i < sizeof(measurement) / sizeof(measurement[0]) ; i++){
        Paint_DrawNum(112, 0, i + 1, &Font8, 0, WHITE, BLACK);
        measurement[i] = capture_distance(BlackImage);
    }

    // Using a specific formula to calculate the area of the shape
    double sub_area1 = measurement[0] * measurement[11];
    double sub_area2 = measurement[2] * measurement[3];
    double sub_area3 = measurement[5] * measurement[6];
    double sub_area4 = measurement[7] * measurement[8];
    double sub_area5 = measurement[2] * measurement[11];

    double output = (double)(sub_area1 + sub_area2 + sub_area3 + sub_area4 +
sub_area5);

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}

/**
 * @brief Calculate and display area of a specific irregular shape (Shape 5)
 *
 * This function measures distances at three corners of a specific irregular
shape (Shape 5),
 * applies a specific formula to calculate the area, and displays the measured
values
 * and the area on the OLED screen.
 *
 * @param BlackImage A pointer to the image cache for OLED display
 * @return A structure containing the calculated area in square centimeters
and converted square feet value
 */
double_array calculate_area_shape5(UBYTE *BlackImage){
    // Initialize the result structure
    double_array result = {0};

    // Array to store measurements at three corners of the shape
    int16_t measurement[3] = {0};

    // Display a message for corner measurement
    Paint_DrawString_EN(0, 0, "Measuring at Corner:", &Font8, WHITE, BLACK);

    // Loop to measure distances at each corner
    for(int i = 0; i < sizeof(measurement) / sizeof(measurement[0]) ; i++){
        Paint_DrawNum(112, 0, i + 1, &Font8, 0, WHITE, BLACK);
        measurement[i] = capture_distance(BlackImage);
    }

    // Using a specific formula to calculate the area of the shape
    double sub_area1 = (measurement[0] + measurement[1]) * measurement[2] / 2;
    double output = (double)sub_area1;

    // Store the calculated area and converted area value in the result
structure
    result.result[0] = output;
    result.result[1] = output * SQUARE_FEET;

    // Return the result structure
    return result;
}
```

button.h

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file button.h
 * @brief Header file for button initialization functions.
 * @author Jithendra H S
 * @date 2023-12-15
*/
// Define constants for GPIO pins
#define GPIO10 (10)
#define GPIO11 (11)

// Define constants for GPIO direction
#define GPIO_OUT 1
#define GPIO_IN  0

// Function prototype for initializing buttons
void Button_Init();
```

button.c

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file button.c
 * @brief Implementation of button initialization functions.
 * @author Jithendra H S
 * @date 2023-12-15
*/
#include "button.h"
#include "pico/stdlib.h"

#define bool_to_bit(x) ((uint)!!(x))

void Button_Init(){
    // Set GPIO10 as input
    sio_hw->gpio_oe_clr = (1ul << GPIO10);
    sio_hw->gpio_clr = (1ul << GPIO10);
    iobank0_hw->io[GPIO10].ctrl = GPIO_FUNC_SIO <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    // Enable pull-up resistor
    *(io_rw_32 *) ((void *)(REG_ALIAS_XOR_BITS | (uintptr_t)((volatile void *)
&padsbank0_hw->io[GPIO10])))
     = (padsbank0_hw->io[GPIO10] ^ ((bool_to_bit(true) <<
PADS_BANK0_GPIO0_PUE_LSB)
      | (bool_to_bit(false) << PADS_BANK0_GPIO0_PDE_LSB)))
      & (PADS_BANK0_GPIO0_PUE_BITS | PADS_BANK0_GPIO0_PDE_BITS);

    // Set GPIO11 as input
    sio_hw->gpio_oe_clr = (1ul << GPIO11);
    sio_hw->gpio_clr = (1ul << GPIO11);
    iobank0_hw->io[GPIO11].ctrl = GPIO_FUNC_SIO <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    // Enable pull-up resistor
    *(io_rw_32 *) ((void *)(REG_ALIAS_XOR_BITS | (uintptr_t)((volatile void *)
&padsbank0_hw->io[GPIO11]))) =
     (padsbank0_hw->io[GPIO11] ^ ((bool_to_bit(true) <<
PADS_BANK0_GPIO0_PUE_LSB)
     | (bool_to_bit(false) << PADS_BANK0_GPIO0_PDE_LSB)))
     & (PADS_BANK0_GPIO0_PUE_BITS | PADS_BANK0_GPIO0_PDE_BITS);

}
```

lidar.h

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file lidar.h
 * @brief Header file for LIDAR interface functions.
 * @author Jithendra H S
 * @date 2023-12-15
*/
#include "stdint.h"

/**
 * @brief Read distance from the LIDAR device.
 *
 * This function communicates with the LIDAR device over I2C to read the
distance.
```

```
 * It reads the low and high bytes from the specified registers and combines
them
 * to obtain the final distance value.
 *
 * @return The distance measured by the LIDAR device.
 */
uint16_t read_lidar();
```

lidar.c

```
/******************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file lidar.c
 * @brief Source file for LIDAR interface functions.
 * @author Jithendra H S
 * @date 2023-12-15
*/
#include "lidar.h"
#include "i2c_code.h"
#include "stdio.h"

// I2C address of the LIDAR device
static const uint8_t LIDAR = 0x10;

// Registers
static const uint8_t DIST_LOW = 0x00;
static const uint8_t DIST_HIGH = 0x01;

/**
 * @brief Read distance from the LIDAR device.
 *
 * This function communicates with the LIDAR device over I2C to read the
distance.
 * It reads the low and high bytes from the specified registers and combines
them
 * to obtain the final distance value.
 *
 * @return The distance measured by the LIDAR device.
 */
uint16_t read_lidar() {
    // I2C instance to use (in this case, i2c0)
    i2c_inst_t *i2c = i2c0;

    // Buffer to store raw reads
    uint8_t data[2];

    // Read distance low and high bytes
    reg_read(i2c, LIDAR, DIST_LOW, data, 2);

    // Combine low and high bytes to get the distance
    uint16_t distance = (data[1] << 8) | data[0];

    // Print the results (you can remove this if not needed)
    printf("Distance: %d\r\n", distance);

    return distance;
}
```

mpu6050.h

```
/******************************************************************************
**
 * Copyright (C) 2023 by Arjun Sukumaran
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file mpu6050.h
 * @brief Function declaration for MPU6050 sensor communication and data
processing.
 * @author Arjun Sukumaran
 * @date December 15, 2023
*/
#include "hardware/i2c.h"

/**
 * @brief Resets the MPU6050 device.
 *
 * This function sends the necessary I2C command to reset the MPU6050 device.
 *
 * @param i2c The I2C instance to use for communication.
 */
void resetMPU6050(i2c_inst_t *i2c);

/**
 * @brief Reads accelerometer data from the MPU6050.
 *
 * This function reads raw accelerometer data from the MPU6050 and stores it
in the provided array.
 *
 * @param i2c The I2C instance to use for communication.
 * @param accel An array to store the X, Y, and Z-axis accelerometer values.
 */
void readAccelData(i2c_inst_t *i2c, int16_t accel[]);

/**
 * @brief Calculates the tilt angle from accelerometer value.
 *
 * This function calculates the tilt angle in degrees from the given
accelerometer value.
 *
 * @param accel_value The raw accelerometer value.
 * @return The tilt angle in degrees.
 */
float calculate_tilt_angle(int16_t accel_value);

/**
 * @brief Reads the tilt angle from the MPU6050 accelerometer.
 *
 * This function reads the accelerometer data, calculates the tilt angles, and
stores them in the provided array.
 *
 * @param acceleration An array to store the X, Y, and Z-axis tilt angles.
 */
void read_tilt_angle(int16_t acceleration[]);
```

mpu6050.c

```
/******************************************************************************
**
 * Copyright (C) 2023 by Arjun Sukumaran
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Arjun Sukumaran and the University of Colorado are not liable for
 * any misuse of this material.
 *
******************************************************************************/
/**
 * @file mpu6050.c
 * @brief Source file for MPU6050 sensor communication and data processing.
 * @author Arjun Sukumaran
 * @date December 15, 2023
*/
#include "mpu6050.h"
#include "stdint.h"
#include "math.h"
#include <stdio.h>

// MPU6050 I2C address
#define MPU6050_ADDRESS (0x68)

/**
 * @brief Resets the MPU6050 device.
 *
 * This function sends the necessary I2C command to reset the MPU6050 device.
 *
 * @param i2c The I2C instance to use for communication.
 */
void resetMPU6050(i2c_inst_t *i2c) {
    // Data to send for the reset
    uint8_t resetData[] = {0x6B, 0x00};

    // Write the reset data to the MPU6050 device
    i2c_write_blocking(i2c, MPU6050_ADDRESS, resetData, sizeof(resetData),
false);
}


/**
 * @brief Reads accelerometer data from the MPU6050 device.
 *
 * This function reads accelerometer data from the MPU6050 device over I2C.
 *
 * @param i2c The I2C instance to use for communication.
 * @param accel An array to store the accelerometer data [X, Y, Z].
 */
void readAccelData(i2c_inst_t *i2c, int16_t accel[3]) {
    // Buffer to store raw accelerometer data
    uint8_t buffer[6];

    // Register address for accelerometer data
    uint8_t regAddress = 0x3B;
    // Write the register address to initiate reading
    i2c_write_blocking(i2c, MPU6050_ADDRESS, &regAddress, 1, true);

    // Read accelerometer data from the MPU6050 device
    i2c_read_blocking(i2c, MPU6050_ADDRESS, buffer, sizeof(buffer), false);

    // Combine high and low bytes for each axis and store in the accel array
    for (int i = 0; i < 3; i++) {
        accel[i] = (buffer[i * 2] << 8 | buffer[(i * 2) + 1]);
    }
}

/**
 * @brief Calculates the tilt angle in degrees based on accelerometer values.
 *
 * This function calculates the tilt angle in degrees using the arctangent
function.
 *
 * @param accel_value The accelerometer value to calculate the tilt angle for.
 * @return The calculated tilt angle in degrees.
 */
float calculate_tilt_angle(int16_t accel_value) {
    // Calculate the tilt angle in degrees using arctangent
    return atan2(accel_value, 16384.0) * (180.0 / M_PI);
}

/**
 * @brief Reads accelerometer data and calculates tilt angles.
 *
 * This function reads accelerometer data from the MPU6050 device, calculates
tilt
 * angles for each axis, and prints the results.
 *
 * @param acceleration An array to store the accelerometer data [X, Y, Z].
 */
void read_tilt_angle(int16_t acceleration[3]){
    // Read accelerometer data from the MPU6050 device
    readAccelData(i2c1, acceleration);

    // Calculate tilt angles for each axis and store the results in the array
    acceleration[0] = (int16_t)calculate_tilt_angle(acceleration[0]) * 2;
    acceleration[1] = (int16_t)calculate_tilt_angle(acceleration[1]) * 2;
    acceleration[2] = (int16_t)calculate_tilt_angle(acceleration[2]) * 2;

    // Print the calculated tilt angles for X and Y axes
    printf("Acc. X = %d, Y = %d\n", acceleration[0], acceleration[1]);
}
```

spi_code.h

```
/***************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file spi_code.h
 * @brief Header file for SPI communication functions.
 * @author Jithendra H S
 * @date December 15, 2023
*/
#include "stdint.h"
#include "hardware/structs/iobank0.h"
#include "hardware/structs/spi.h"

// Define pin constants for SPI
#define SPI_PORT (spi0_hw)
#define SPI_SCK_PIN (2)
#define SPI_TX_PIN (3)
#define SPI_CS_PIN (5)
#define SPI_DC_PIN (20)
#define SPI_RESET_PIN (21)

#define SET_SPI_DC_PIN (1ul << SPI_DC_PIN)
#define SET_SPI_CS_PIN (1ul << SPI_CS_PIN);

/**
 * @brief Initializes the SPI module for normal operation.
 *
 * This function configures the SPI module for normal operation with the
specified
 * parameters, including baud rate, data format, and other settings.
 */
void SPI_Module_Init();

/**
 * @brief Writes a sequence of data to the SPI module.
 *
 * This function writes a sequence of data to the SPI module. It waits for the
 * transmit FIFO not full (TNF) condition and then writes data to the transmit
FIFO.
```

```
 * After completing the data transfer, it waits for the SPI module to finish
and
 * clears any overrun interrupt.
 *
 * @param spi The SPI hardware instance.
 * @param data A pointer to the data to be written.
 * @param len The length of the data to be written.
 */
void SPI_write(spi_hw_t *spi, const uint8_t *data, size_t len);

/**
 * @brief Writes a command to the SPI display.
 *
 * This function writes a command to the SPI display. It sets the DC pin low
to
 * indicate command mode, sets the CS pin low to enable SPI communication,
writes
 * the command data to SPI using bit-banging, and finally sets the CS pin high
to
 * disable SPI communication.
 *
 * @param data The command data to be written.
 */
void SPI_WriteCommand(const uint8_t data);

/**
 * @brief Writes data to the SPI display.
 *
 * This function writes data to the SPI display. It sets the DC pin high to
 * indicate data mode, sets the CS pin low to enable SPI communication, writes
 * the data to SPI using bit-banging, and finally sets the CS pin high to
 * disable SPI communication.
 *
 * @param data The data to be written.
 */
void SPI_WriteData(const uint8_t data);

/**
 * @brief Initializes the SPI module using bit-banging.
 *
 * This function configures the GPIO pins used for SPI communication and sets
 * up the SPI module using bit-banging.
 */
void SPI_Module_Init_BIT_BANGING();

/**
 * @brief Sends a byte of data over SPI using bit-banging.
 *
 * This function sends each bit of the data byte sequentially over SPI using
 * bit-banging. It assumes that the SPI pins and other configurations have
been
 * properly set up before calling this function.
 *
 * @param data The byte of data to be sent.
 */
void SPI_send_byte(uint8_t data);
```

spi_code.c

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file spi_code.c
 * @brief SPI communication functions for OLED display.
 * @author Jithendra H S
 * @date December 15, 2023
 */
#include "spi_code.h"
#include "pico/stdlib.h"
#include "oled.h"

// Constant for byte size
#define BYTE_SIZE (8)

/**
 * @brief Writes data to the SPI module.
 *
 * @param spi   Pointer to the SPI hardware structure.
 * @param data  Pointer to the data buffer.
 * @param len   Number of bytes to write.
 */
void SPI_write(spi_hw_t *spi, const uint8_t *data, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        // Wait for transmit FIFO not full (TNF)
        while (!(spi->sr & SPI_SSPSR_TNF_BITS));

        // Write data to transmit FIFO
        spi->dr = (uint32_t)data[i];
    }
    // Wait for SPI to finish
    while (spi->sr & SPI_SSPSR_BSY_BITS);

    // Clear overrun interrupt
    spi->icr = SPI_SSPICR_RORIC_BITS;
}

/**
 * @brief Writes a command to the SPI module.
 *
 * @param data  The command to write.
 */
void SPI_WriteCommand(const uint8_t data) {
    // Set DC pin low to indicate command mode
    sio_hw->gpio_clr = SET_SPI_DC_PIN;

    // Set CS pin low to enable SPI communication
    sio_hw->gpio_clr = SET_SPI_CS_PIN;

    // Write command data to SPI
    SPI_send_byte(data);

    // Set CS pin high to disable SPI communication
    sio_hw->gpio_set = SET_SPI_CS_PIN;
}

/**
 * @brief Writes data to the SPI module in data mode.
 *
 * @param data  The data to write.
 */
void SPI_WriteData(const uint8_t data) {
    // Set DC pin high to indicate data mode
    sio_hw->gpio_set = SET_SPI_DC_PIN;

    // Set CS pin low to enable SPI communication
    sio_hw->gpio_clr = SET_SPI_CS_PIN;

    // Write data to SPI
    SPI_send_byte(data);

    // Set CS pin high to disable SPI communication
    sio_hw->gpio_set = SET_SPI_CS_PIN;
}


/**
```

```
 * @brief Initializes the SPI module with the specified configuration.
 */
void SPI_Module_Init() {
    // Get a pointer to the SPI hardware
    spi_hw_t *spi = spi0_hw;

    // Disable the SPI
    spi->cr1 &= ~SPI_SSPCR1_SSE_BITS;

    // Set prescalar and postdiv for baud rate
    uint8_t prescalar = 2;
    uint16_t postdiv = 63;
    spi->cpsr = prescalar;
    spi->cr0 |= (postdiv - 1) << 8;

    // Configure data format (8 data bits, cpol, cpha)
    uint8_t data_bits = 8;
    uint8_t cpol = 0;
    uint8_t cpha = 0;
    spi->cr0 |= ((uint8_t)(data_bits - 1)) << SPI_SSPCR0_DSS_LSB |
                ((uint8_t)cpol) << SPI_SSPCR0_SPO_LSB |
                ((uint8_t)cpha) << SPI_SSPCR0_SPH_LSB;

    // Enable the SPI
    spi->cr1 |= SPI_SSPCR1_SSE_BITS;

    // Configure GPIO pins for SPI
    iobank0_hw->io[SPI_SCK_PIN].ctrl = GPIO_FUNC_SPI <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_TX_PIN].ctrl = GPIO_FUNC_SPI <<
IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
}


/**
 * @brief Initializes the SPI module for bit-banging mode.
 */
void SPI_Module_Init_BIT_BANGING() {
    // Set GPIO functions for SPI pins
    iobank0_hw->io[SPI_SCK_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_TX_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_CS_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_DC_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    iobank0_hw->io[SPI_RESET_PIN].ctrl = GPIO_FUNC_SIO <<
                                       IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
    // Set output enables for SPI pins
    sio_hw->gpio_oe_set = (1ul << SPI_SCK_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_TX_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_CS_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_DC_PIN);
    sio_hw->gpio_oe_set = (1ul << SPI_RESET_PIN);

    // Clear SPI pins
    sio_hw->gpio_clr = (1ul << SPI_SCK_PIN);
    sio_hw->gpio_clr = (1ul << SPI_TX_PIN);
    sio_hw->gpio_clr = (1ul << SPI_CS_PIN);
}


/**
 * @brief Sends a byte of data over SPI using bit-banging.
 *
 * This function sends each bit of the data byte sequentially over SPI using
 * bit-banging. It assumes that the SPI pins and other configurations have
been
 * properly set up before calling this function.
 *
 * @param data The byte of data to be sent.
 */
void SPI_send_byte(uint8_t data) {
    // Reverse the bits of the data byte
    data = reverse(data);

    for(int i = 0; i < BYTE_SIZE; i++) {
        // Extract the least significant bit
        uint8_t bit = data & 0x01;

        // Set or clear the SPI_TX_PIN based on the bit value
        gpio_put(SPI_TX_PIN, bit);

        // Set or clear the GPIO corresponding to SPI_TX_PIN
        if(bit) {
            sio_hw->gpio_set = (1ul << SPI_TX_PIN);
        } else {
            sio_hw->gpio_clr = (1ul << SPI_TX_PIN);
        }

        // Wait for a short duration
        sleep_us(5);

        // Set the SPI clock signal high
        sio_hw->gpio_set = (1ul << SPI_SCK_PIN);
        // Wait for a short duration
        sleep_us(10);

        // Clear the SPI clock signal
        sio_hw->gpio_clr = (1ul << SPI_SCK_PIN);

        // Wait for a short duration
        sleep_us(5);

        // Shift to the next bit
        data = data >> 1;

        // Wait for a short duration
        sleep_us(2);
    }
}
```

User_interface.h

```
/******************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file user_interface.h
 *
 * @brief Header file for the main user interface function.
 *
 * @author Jithendra H S
 *
 * @date December 15, 2023
*/
/**
 * @brief Main user interface function
 *
 * This function handles user interactions and displays information on the
OLED screen.
 * It continuously checks the states of GPIO10 and GPIO11 buttons to perform
corresponding actions.
 */
```

```
void user_interface();
```

user_interface.c

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra H S
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra H S and the University of Colorado are not liable for
 * any misuse of this material.
 *
****************************************************************************/
/**
 * @file user_interface.c
 *
 * @brief Main user interface function for handling user interactions and
 *        displaying information on an OLED screen.
 *
 * @author Jithendra H S
 *
 * @date December 15, 2023
*/
#include "user_interface.h"
#include "string.h"
#include <stdio.h>
#include "pico/stdlib.h"
#include <stdlib.h>
#include "button.h"
#include "area.h"

/**
 * @brief Main user interface function
 *
 * This function handles user interactions and displays information on the
OLED screen.
 * It continuously checks the states of GPIO10 and GPIO11 buttons to perform
corresponding actions.
 */
void user_interface() {
    // Create a new image cache
    UBYTE *BlackImage;
    BlackImage = (UBYTE *)malloc(OLED_IMAGE_SIZE);
    if (BlackImage == NULL) {
        // No enough memory
        printf(" No enough memory\n");
    }

    // Display the main menu
    menu(BlackImage);

    // Initial shape selection
    char *shape = shapes[0];

    bool gpio10_state;
    bool gpio11_state;

    while (true) {
        // Read the state of GPIO10
        gpio10_state = !!((1ul << GPIO10) & sio_hw->gpio_in);
        // Read the state of GPIO11
        gpio11_state = !!((1ul << GPIO11) & sio_hw->gpio_in);

        // Perform actions based on the GPIO10 state
        if (!gpio10_state) {
            sleep_ms(200);
            // Move cursor to select a shape
            shape = move_cursor(BlackImage);
            printf("GPIO10 is high (pressed)! and shape %s\n\r", shape);
        }

        // Perform actions based on the GPIO11 state
        if (!gpio11_state) {
            sleep_ms(200);
            OLED_Clear();
            // Use memset to set all values to 0x00
            memset(BlackImage, 0x00, OLED_IMAGE_SIZE);
            OLED_Display(BlackImage);

            // Calculate area based on the selected shape
            double_array area = calculate_area(shape, BlackImage);

            // Display the calculated area
            OLED_Clear();
            // Use memset to set all values to 0x00
            memset(BlackImage, 0x00, OLED_IMAGE_SIZE);
            printf("GPIO11 is high (pressed)! , shape : %s and area %f\n\r",
shape, area.result[1]);
            Paint_DrawString_EN(0, 12, "Final value :", &Font12, WHITE,
BLACK);
            Paint_DrawString_EN(93, 24, "     ", &Font12, WHITE, BLACK);
            Paint_DrawString_EN(79, 36, "        ", &Font12, WHITE, BLACK);
            if (strcmp(shape, "Distance") == 0) {
                Paint_DrawNum(0, 24, area.result[0], &Font12, 2, WHITE,
BLACK);
                Paint_DrawString_EN(114, 24, "cm", &Font12, WHITE, BLACK);
                Paint_DrawNum(0, 36, area.result[1], &Font12, 2, WHITE,
BLACK);
                Paint_DrawString_EN(100, 36, "foot", &Font12, WHITE, BLACK);
            } else {
                Paint_DrawNum(0, 24, area.result[0], &Font12, 2, WHITE,
BLACK);
                Paint_DrawString_EN(93, 24, "sq.cm", &Font12, WHITE, BLACK);
                Paint_DrawNum(0, 36, area.result[1], &Font12, 2, WHITE,
BLACK);
                Paint_DrawString_EN(79, 36, "sq.foot", &Font12, WHITE, BLACK);
            }

            Paint_DrawString_EN(0, 120, "*Press Red to exit", &Font8, WHITE,
BLACK);
            OLED_Display(BlackImage);

            // Wait for GPIO11 to be released before going back to the menu
            while (true) {
                // Read the state of GPIO11
                bool gpio11_state = !!((1ul << GPIO11) & sio_hw->gpio_in);
                // Perform actions based on the GPIO11 state
                if (!gpio11_state) {
                    printf("Exiting after area is displayed\n\r");
                    sleep_ms(500);
                    break;
                }
            }

            // Display the main menu
            menu(BlackImage);
        }
    }
}
```

9.4 Appendix - Data Sheets and Application Notes

(*Attached in respective folder)