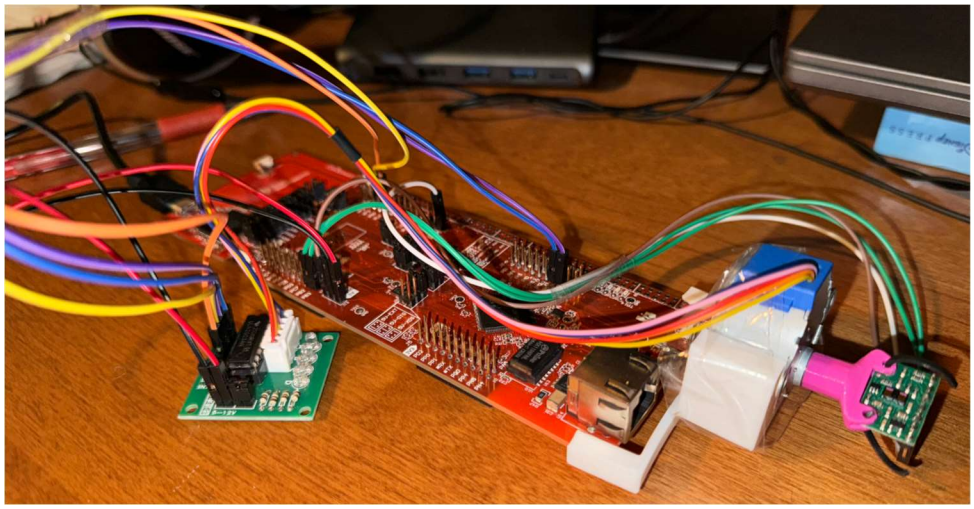
3D Lidar Scanning and Mapping

Annie Phan

Device Overview



Features

- 360-Degree Scanning Capability: Measures distances around a single level with full 360-degree coverage.
- Orthogonal Depth Measurement: Takes measurements at right angles to determine depth, enhancing the 3D mapping accuracy.
- USB Data Communication: Enables seamless data transfer for processing and visualization.
- Texas Instruments MSP432E401Y Microcontroller: Utilizes an Arm Cortex-M4F Processor Core running at 96 MHz for efficient data management.
- UNL2003 Stepper Motor Controller: Supports precise 360-degree rotation in 512 steps, with LED indicators for operation status. Operates on 5V-12V.
- VL53L1X Time-of-Flight Sensor: Offers accurate distance measurements (±20 mm error margin) up to 4 meters, powered by 2.6V-3.5V.
- Data Communication Protocol: Features I2C serial communication between the microcontroller and the ToF sensor, and UART serial communication for PC connectivity, supported by Python scripts.
- 3D Visualization: Utilizes the Open3D Python library for processing and visualizing 3D data, compatible with Python 3.6, providing detailed representations of scanned areas.

General Description

The lidar system is an advanced 3D scanning tool designed to capture distances across multiple 360-degree planes along a perpendicular axis and process the data to create a 3D map of the scanned area. It consists of three main components: a microcontroller, a stepper motor, and a time-of-flight sensor.

Apart from handling 3D visualization, the microcontroller is responsible for managing all system operations, providing a power source to other components, and transmitting data to an external device via serial communication. The system utilizes the stepper motor to enable a complete 360-degree rotation, allowing the time-of-flight sensor to capture distance measurements across the entire vertical plane.

Using infrared laser pulses, the time-of-flight sensor calculates object distances by measuring the time it takes for the pulses to reflect back to a detector. This data is then sent to the microcontroller via I2C communication.

To interact with the data, the system connects to a PC running specific Python scripts. The microcontroller transmits data to the PC through UART, allowing users to visualize the mapped space using the included Python scripts. Overall, the Lidar system offers a comprehensive solution for capturing and visualizing spatial data with precision and accuracy.

Block Diagram

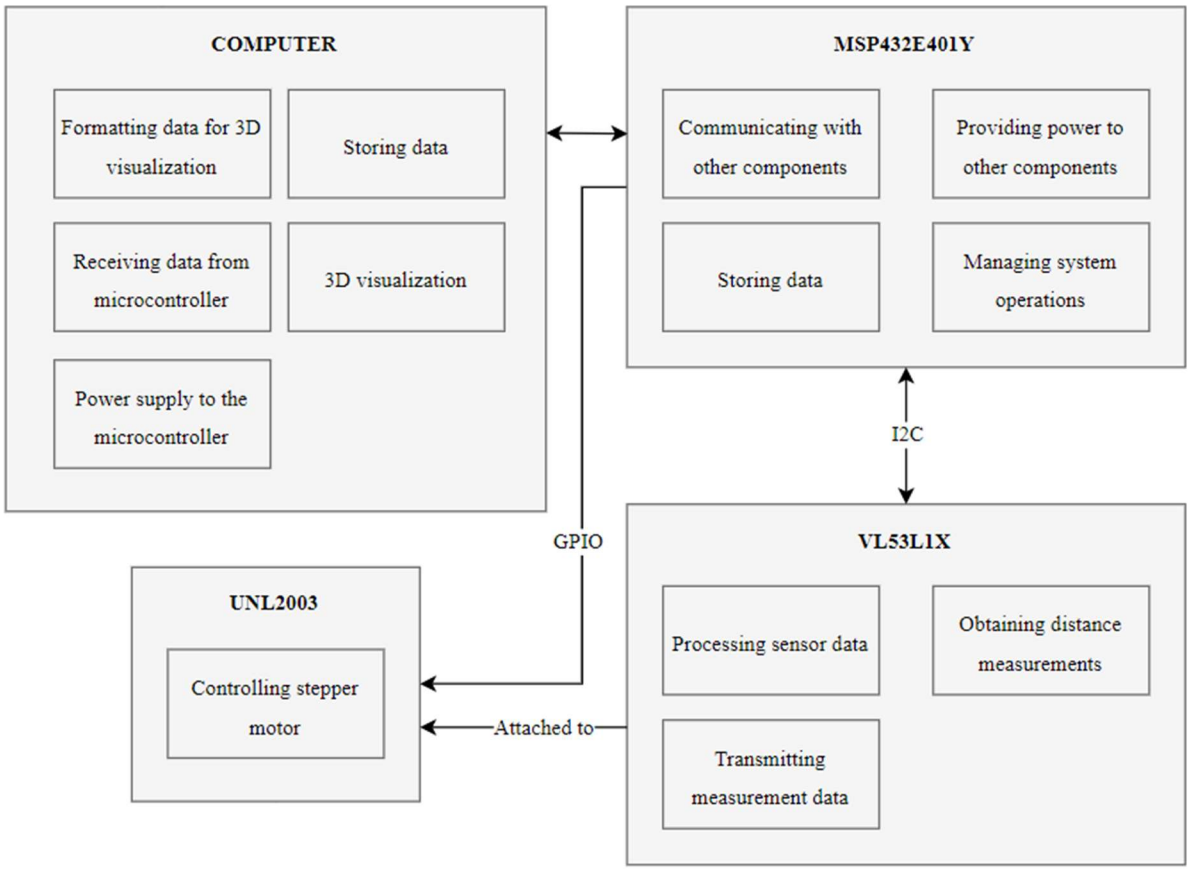


Figure 1. Block diagram for Lidar system.

Device Characteristics Table

Device	Feature	Detail	Device Pin	Microcontroller Pin
MSP432E401Y (Microcontroller)	Bus Speed	80 MHz	-	-
	Serial Port	COM4	-	-
	Baud Rate	115200	-	-
	Distance Status	Flash LED with each measurement	-	PF4
	Rotate Status	Blink every rotation	-	PF0
UNL2003 (Stepper Motor)	Power (VIN)	Flash LED with each complete turn	VDD	5V
	Ground (GND)	0 V	GND	GND
	Phase A	-	IN1	PH0
	Phase B	-	IN2	PH1
	Phase C	-	IN3	PH2
	Phase D	-	IN4	PH3
VL53L1X (Time-of-Flight Sensor)	-	-	VDD	-
	Power (VIN)	3.3 V	VIN	3.3 V
	Ground (GND)	0 V	GND	GND
	Serial Data Line (SDA)	-	SDA	PB3
	Serial Clock Line (SCL)	-	SCL	PB2
	-	-	XSHUT	-
	-	-	GPIO1	-

Detailed Description

Distance Measurement

Distance measurements were obtained using a VL53L1X sensor. The VL53L1X sensor, instrumental in the Lidar system, leverages infrared laser pulses to capture precise distance readings. With each micro-step of the attached stepper motor, the sensor updates its measurement, enabling a detailed scan over a 360-degree horizontal plane.

Upon emitting a laser pulse, the sensor begins a timer. It halts the timer once the pulse reflects back from an object within its field of view. The time recorded (how long the light travels before returning to the sensor) is used to calculate the distance to the object, utilizing the speed of light as a constant. These calculations conform to the sensor's default settings, ensuring accuracy and consistency. Following the distance calculations, the ToF sensor relays this information to the microcontroller through an established I2C protocol, a standard communication method that ensures swift and error-free data transfer.

The system follows the VL53L1 Core API settings provided by the manufacturer, STMicroelectronics, coupled with additional platform code from 2DX design studios. These settings are maintained, with no deviations or custom modifications, to preserve the sensor's intended operation as designed by the original developers.

Outlined in a step-by-step flowchart (referenced as Figure 4), the sequence begins with the microcontroller initializing its I2C interface, a prerequisite for communicating with the ToF sensor. Concurrently, it establishes variables that will be later utilized for processing ToF data. Once the microcontroller is primed, the ToF sensor enters a standby mode, anticipating a user command in the form of a button press to commence the distance measurement process.

The Lidar system's state is monitored by a specific integer variable within the microcontroller, indicative of whether the system is currently active in data collection (1) or in a standby state (0). The transition between these states is governed by an interrupt-driven button mechanism, which, upon engagement, toggles the system's data recording functionality.

When in data capture mode, the system sequences through distance measurements as the stepper motor completes its rotation cycle, each step triggers a new distance reading. The sensor data readiness is continuously checked, and once the data is available, it is promptly fetched through the ToF sensor's I2C function and transferred to the microcontroller. This data is then packaged and sent to the PC through UART protocol, where a specialized Python script named data_collect.py processes the information, structuring it into a .xyz file. The script ensures that each set of data corresponds with a specific step of the motor, maintaining spatial representation.

Then the Python script uses trigonometric principles to transform the linear distance data into three-dimensional coordinates. By using the stepper motor's steps and applying them against the full rotational degree it calculates the angle in radians. With the sensor's upward-facing

starting position, the script then uses this angle to compute the horizontal ($y = d*\cos\theta$) and vertical ($z = d*\sin\theta$) components of each data point, assuming a spherical coordinate system.

The system encountered mechanical constraints with the mounting mechanism, limiting the acquisition to two complete sets of distance data. The latter data set showed discrepancies due to these mechanical impediments, leading to a subset of the distance measurements to have errors.

The Lidar system's data acquisition software currently requires manual input for the x-displacement value within the data_collect.py script. This manual input method ensures that the x-coordinate in the resulting .xyz data file accurately reflects the measured displacement without needing additional programmatic adjustments prior to its incorporation into the dataset.

Visualization

3D visualization is executed by importing the collected .xyz data into a Python environment through the Open3D library. This library facilitates the generation of a point cloud (a collection of data points in space representing the external contours of the scanned object or area). Additionally, the library creates a corresponding line set that links these data points, illustrating the shape and structure of the object or area with lines that connect adjacent slices of the scan. The visualization functions within the library then render these points and lines into a coherent 3D model.

The 3D visualization capacity of the system was verified on a Lenovo Yoga 7 14IAL7 laptop equipped with a 12th Gen Intel(R) Core(TM) i7-1255U processor, an Integrated Intel® Iris® Xe Graphics, and 16 GB of RAM. All Python code execution was facilitated through Visual Studio Code Version 1.87.0. The Python environment for running the data_collect.py script was set to version 3.8.10, while the 3d_plot.py script supports Python 3.8 and higher, dictated by the Open3D library's specific version requirements and compatibility constraints.

Open3D's visualization capabilities involve crafting a detailed point cloud, which encompasses every discrete data point intended for visual representation. This cloud forms the basis for a line set, which interlinks points to represent the scanned environment's spatial relationships and geometry. The rendering of this line set, performed by the library's visualization tools, results in an interactive 3D model, providing a virtual depiction of the physical space scanned by the Lidar system. The visualization process is seen in the concluding sections of the flowchart presented in Figure 4.

In addition to measuring distances, the project managed to gather a detailed set of data of a hallway seen in Figure 3., even with difficulties related to the sensor's mounting mechanism. The set of data presented noticeable irregularities, with points indenting into the 3D structure. This issue was linked to inaccuracies in measuring distances, stemming from the mechanical issues with the mounting system and lighting. Despite these challenges, the dataset, which showcase both the strengths and limitations of the Lidar system's ability to visualize data, is accessible in the distance_data.xyz file located in the “Data & Visualization” folder.

Application Note, Instructions and Expected Output

Initial Setup for Lidar System Operation:

Before using the integrated Lidar system, users must install essential software on their computers to ensure compatibility for data reception and visualization. Follow these detailed steps on a Windows 10 device:

1. XDS Emulation Software Installation:
- Navigate to the Texas Instruments download page at TI Software Download to acquire the XDS Emulation Software. Install it by choosing the "Typical" setup, enabling debugging and UART communication with the microcontroller.
2. Python 3.6 Installation:
- Download Python 3.6 from Python.org. During installation, select "Add Python 3.6 to path" to ensure Python commands are accessible from the command line. Proceed with the default settings for the rest of the installation.
3. Python Libraries Installation:
- Open a command prompt as administrator. Install Open3D and pySerial libraries by executing pip install open3d followed by pip install pyserial, confirming successful installation.
4. Integrated Development Environment (IDE) Setup:
- The Python IDLE, included with Python installation, used for script editing and execution. Other IDEs can also be used if preferred.

Preparing for Data Collection:

Once the setup is complete, your PC is ready to interface with the Lidar system for data collection and visualization:

1. Configure the Data Collection Script:
- Open data_collection.py in your chosen IDE. Modify system parameters as follows:
    - Initial X-Displacement: Adjust the x variable (line 16) to set the starting x-coordinate in millimeters.
    - X-Displacement Increment: Edit the increment variable (line 17) to change how much x-coordinate increases with each measurement, in millimeters.
    - Output File Name: Change the filename variable (line 12) to your preferred output file name for data storage.
    - COM Port Setting: Determine the active COM port via Device Manager under "Ports (COM & LPT)" and update line 4 accordingly.
2. Running the Script:
- Execute the modified data_collect.py script within your IDE.
3. System Connection:
- Link a micro-USB cable to connect your PC to the microcontroller, making sure it's plugged into the port opposite the Ethernet connection.

4. Initiating the System:
- Press the RESET button on the microcontroller to start the system. Monitor the IDE terminal for system status updates and instructions, such as such as "Press GPIO J1 to start/stop data capture".
5. Location Setup for Data Capture:
- Upon receiving a data capture prompt, position the Lidar system at the desired mapping location. Ensure all connections remain secure.
6. Sensor Orientation:
- Although not mandatory, you can align the ToF sensor upwards by toggling data capture using the GPIO J1 button. Regardless of its initial position, the final visualization will be accurate, but x, y, z values will reference the starting orientation.
7. Begin Data Capture:
- Press the GPIO J1 button to start capturing data, watching for valid distance measurements in the IDE terminal. Investigate any discrepancies for potential sensor or connection issues.
8. Data Handling Post-Capture:
- Following a complete rotation and terminal notification, proceed to map another position as determined by your script's increment setting. Manual adjustments may be needed for discrepancies in displacement settings.

Creating 3D Visualizations:

To visualize the captured data in 3D:

1. Visualization Script Execution:
- Open and run 3d_plot.py in your IDE, ensuring any filename adjustments made in the data collect script are mirrored here.
2. 3D Model Inspection:
- An "Open3D" window displaying your data in 3D will open. Use the mouse controls to explore the model fully.

The Figure 2. presented below offers a visual comparison between an actual campus location (Information Technology Building) and the corresponding 3D model created from Lidar scan data. This side-by-side arrangement is intended to showcase the scanning accuracy of the Lidar system by directly comparing the real-world environment with its digital reconstruction obtained through Lidar.

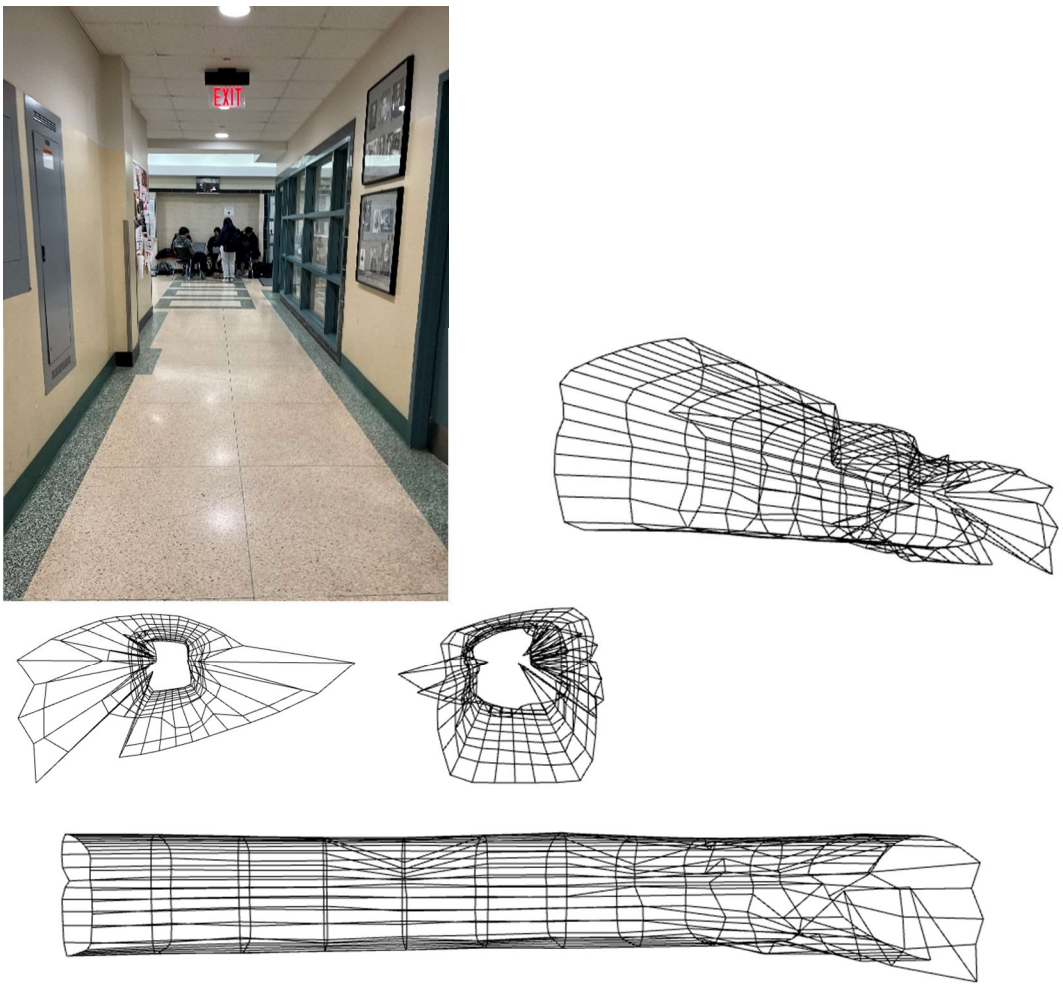


Figure 2. Comparison of a photo of the designated campus area and the 3D model created from scanned data.

Limitations

The microcontroller used includes a Floating-Point Unit (FPU) capable of handling 32-bit calculations. This unit performs basic arithmetic such as addition, subtraction, multiplication, and division as well as more complex operations like the multiply-and-accumulate function and the computation of square roots. It can also switch data between various floating-point formats seamlessly. As a result, executing trigonometric functions, which are part of the standard math.h library, on floating-point numbers is straightforward and error-free.

$$ToF\ Module\ Max\ Quantization\ Error = \frac{4000\ mm}{2^{16}} = 0.061\ mm$$

$$IMU\ Module\ Max\ Quantization\ Error = \frac{32\ g}{2^{16}} = \frac{32\ \times 9.81\ m/s^2}{2^{16}} = 0.00479\ \mathrm{m/s^2}$$

The Time-of-Flight (ToF) sensor demonstrates a maximum error in measurement (known as quantization error) of around 0.061 mm. This is deduced by dividing the sensor's furthest measurable distance of 4000 mm by the largest value representable by a 16-bit number. For the Inertial Measurement Unit (IMU), the maximum quantization error is estimated to be around 0.00479 m/s², factoring in the sensor's upper limit of acceleration (equivalent to 32 times the force of gravity) against the resolution afforded by 16 bits.

For standard serial communication between a PC and a device to occur, the connection's baud rate usually sets the maximum speed limit. Specifically, for the PC's XDS110 UART Port, the highest achievable serial communication speed is 128,000 bits per second (128 kbps). This rate can be confirmed by examining the port's configuration in the PC's device manager.

Data transfer between the microcontroller and the ToF sensor is managed via I2C serial communication, with the microcontroller's I2C ports configured for a 100 kilobits per second clock rate. The connection with the IMU sensor is likely to follow a similar protocol, though the actual speed may vary based on the configured clock speed. The I2C interface of the IMU sensor can support up to 400 kHz.

When evaluating the overall speed limitations of the system, the ToF sensor's range-finding function appears to be the primary constraint, more so than the speed of the stepper motor's rotation. If programmed delays are omitted, the ToF sensor might not output data, causing the system to stop. This problem doesn't affect the stepper motor; if delays are too short, the motor simply doesn't move, but the system continues to function.

To set the assigned system bus speed to 80 MHz on the TM4C1294 using the Phase-Locked Loop (PLL), the PSYSDIV value is defined as 5 within the PLL.h file, as indicated by the formula 480MHz/(PSYSDIV+1), effectively setting the PLL output. This adjustment requires recalibration of the SysTick timer to maintain accurate timing operations, given SysTick's dependency on the system clock. Modifications to SysTick involve setting the reload value to account for 800,000 clock cycles per 10 milliseconds, ensuring timing functions such as delays are accurate under the new 80 MHz bus speed. This thorough method ensures that both the PLL's

frequency and the accuracy of SysTick's timing align with the desired system performance standards.

Circuit Schematic

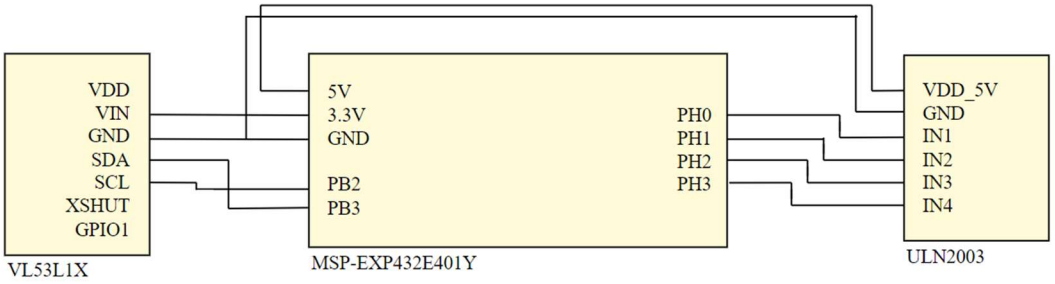


Figure 3. Circuit schematic.

Programming Logic Flowchart

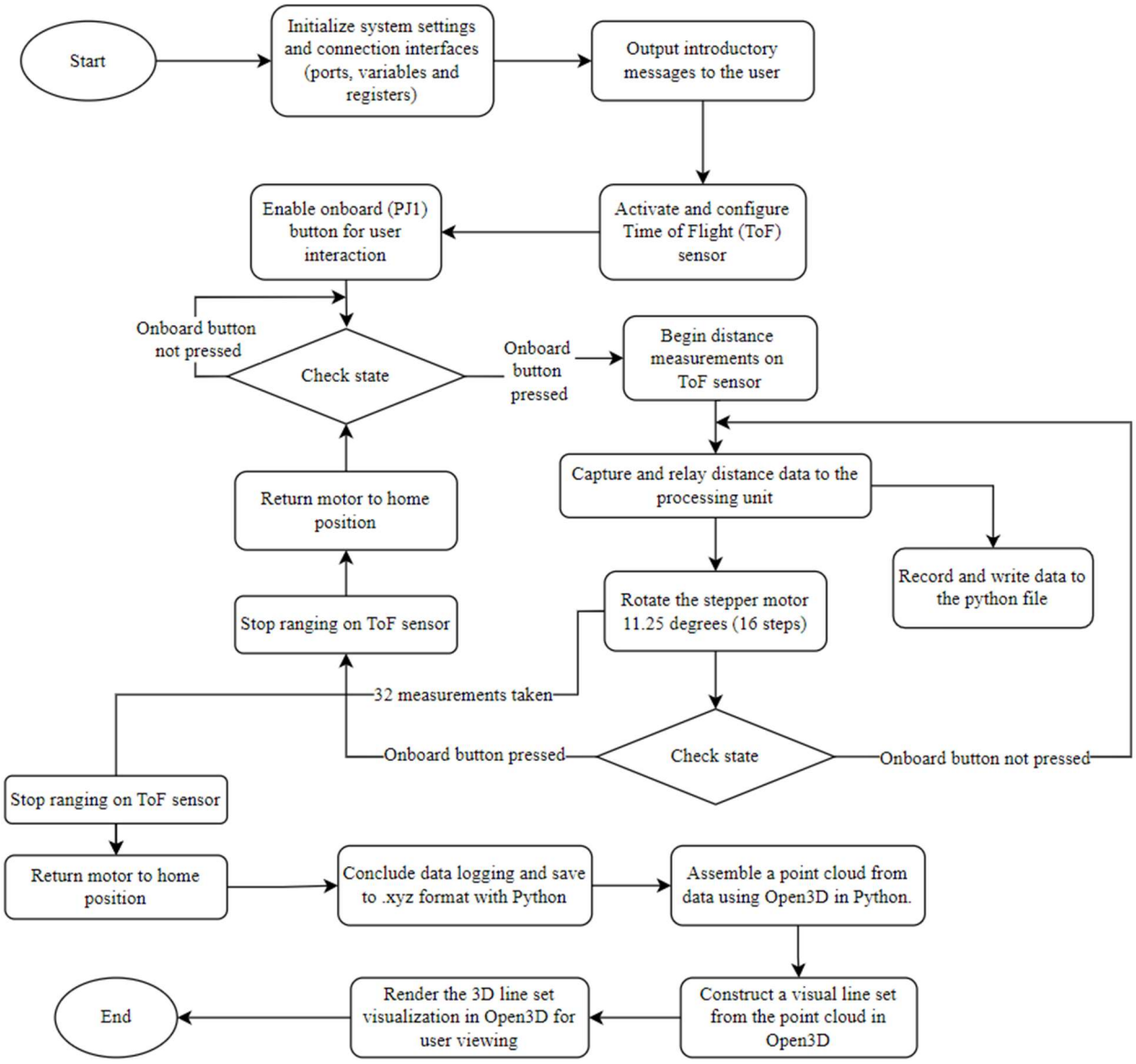


Figure 4. Flow diagram for code.