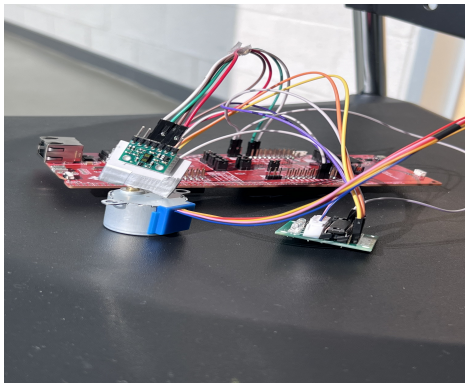
Time of Flight Spatial Mapping System

1 Overview



1.1 Features

- Compatible with Windows 10/11
- Open source Keil uVision5 C project supported with Python 3.10
- Running Python APIs Open3D v0.17.0 and PySerial v3.10
- Polling implementation
- Texas Instruments MSP-EXP432E401Y Microcontroller
  - Arm Cortex-M4F Processor Core
  - 80 MHz bus speed
  - Floating point capability
  - USB connection to computer for power and communication
  - 256 KB SRAM
  - 1024 KB Flash Memory
  - 6KB EEPROM
  - Two 12-bit successive approximation (SAR) ADC modules with max sample rate 2 Msps each
  - $39.99 USD
- VL53L1X Time-of-Flight Sensor Breakout Board with Voltage Regulator (3415-POLOLU)
  - Default long-distance mode LIDAR sensor ranging up to 4m
  - Default 10 Hz ranging frequency in long-distance mode
  - 2.6 V - 3.5V operating voltage (3.3 V chosen)
  - $18.95 USD
- MOT-28BYJ48 Stepper Motor and ULN2003 Driver
  - 512 steps for 360 degree rotation
  - 64 steps per scan (range every 45 degrees)
  - LED status indicators
  - 5 V - 12 V operating voltage (5 V chosen)
  - $6.95 CAD
- Serial Communication
  - I^2C serial communication on [PB2:PB3] of MSP-EXP432E401Y and VL53L1X
  - UART serial communication between MSP-EXP432E401Y and computer via Python
  - 115200 bps (baud rate)

1.2 General Description

1.2.1 ADC Sample Acquisition

The Spatial Mapper embedded system uses the MSP-EXP432E401Y microcontroller, VL53L1X Time-of-Flight sensor and ULN2003 driven stepper motor to acquire distance samples of the outside environment. The samples are used to map an interactive 3D simulation of the environment in Open3D via Python. The ToF sensor is mounted to the stepper motor to enable a 360 degree range of motion. Distance readings are collected plane by plane, which are connected along the depth axis to make a 3D visual. The system mechanism of action is the LIDAR ToF sensor emitting infrared light with a laser, and then receiving it back with detector lens. The sensor and module are controlled by the microcontroller as the points in time the IR light is released are dictated by the C program. The sensor's breakout board module transduces the returning analog light wave to a continuous voltage wave. The voltage is then conditioned to be within the range threshold of the microcontroller's operating voltage by with an ADC routine that operates on the sensor module. The difference in time of the microcontroller's instruction and the returning discrete voltage wave are used with the speed of light constant to calculate the distance of an object. Distance is calculated with

$$distance = \frac{speed}{time} = \frac{c}{\Delta t} = \frac{299792458}{\Delta t}.$$

1.2.2 Serial Communication

The I^2C communication protocol is used between the microcontroller and ToF sensor. The microcontroller acts as a leader, where each system clock cycle the 1 of 8 bits of data is transmitted/received between it and the sensor. The ToF sensor is a follower, meaning that it has no control over the frequency of data communication. I^2C is bidirectional, meaning either the leader and follower can both transmit and receive data. This enables the microcontroller to tell the sensor module when to take a distance reading, and receive it later on with the same GPIO pin. Once the microcontroller receives distance data, it is sent to Python on the computer for cardinal coordinate derivation and visualization. UART is an asynchronous communication protocol, thus no handshaking takes places between the computer and microcontroller. Data is sent on the computer's communication port from the microcontroller on bit at a time at a predetermined baud rate of 115200 bps. The Python program continuously polls for if the distance data has been sent due to the protocol being asynchronous where it then processes the data.

1.2.3 Visualization

Once Python receives data via UART, it converts the distance measurement to X and Y coordinates with trigonometry. The calculations are

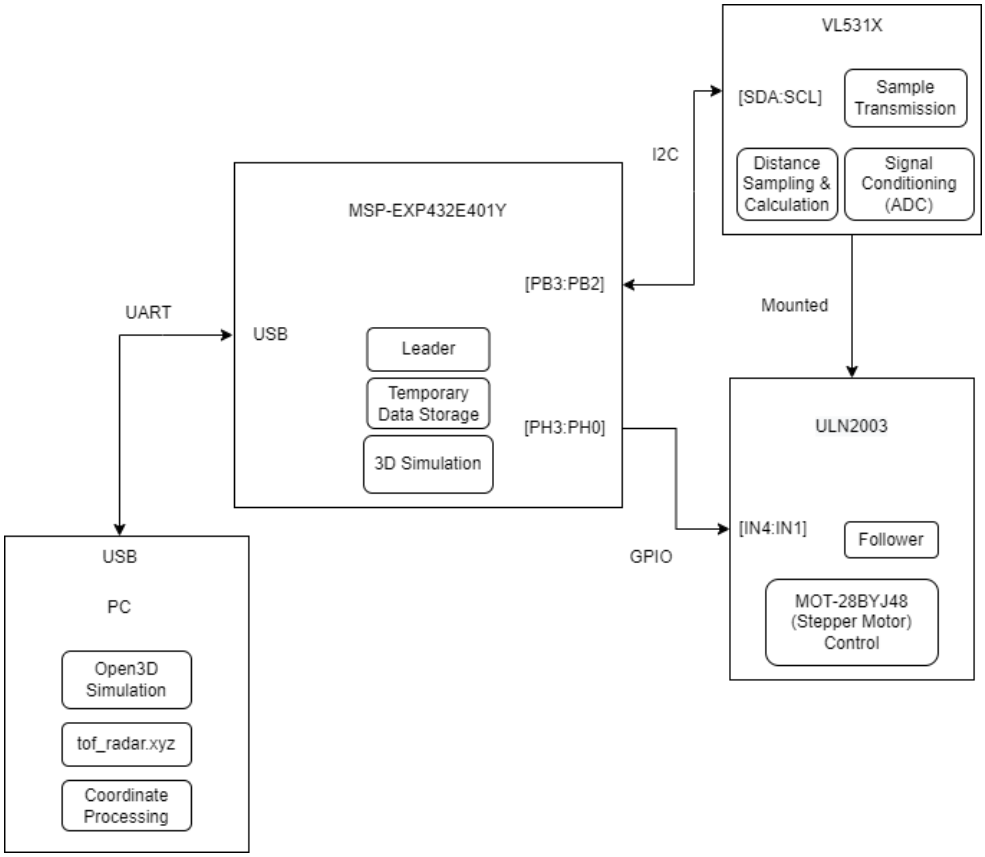
$$x = distance * sin(angle),$$

and

$$y = distance * cos(angle),$$

where `angle` is the angle that the sensor points at a given time. `angle` is tracked with a variable in Python that increments by $\frac{\pi}{4}$ every time distance data is received, starting at 0. This follows the stepper motor's 45 degree rotations. The Z coordinate automatically increases by 2.36 m under the assumption the user travels that distance between planar scans. Finally, each set of coordinates is exported to a file `tof_radar.xyz`. The produced point cloud is then read using the using functions from the Open3D API to create a 3D simulation of the environment.

1.3 Data Flow Graph



2 Device Characteristics Table

MSP-EXP432E401Y		VL53L1X		ULN2003	
Feature	Spec.	Device Pin	MSP-EXP432E401Y Pin	Device Pin	MSP-EXP432E401Y Pin
Bus Speed	80 MHz	VIN	3.3V	VIN	5V
Serial Port	COM3	GND	GND	GND	GND
Baud Rate	115200	SDA	PB3	IN1	PH0
Meas. Status	PN1	SCL	PB2	IN2	PH1
Addit. Status	PF4			IN3	PH2
				IN4	PH3

3 Detailed Description

3.1 Distance Measurement

Behind this embedded system, two programs run to first acquire distance data of the outside environment, and another uses the distance data to create a virtual 3D mapping. The first of the two is a C program running on the MSP-EXP432E401Y interfaces with the VL53L1X time of flight sensor and ULN2003 stepper motor driver to periodically collect scans of the room. With the ToF mounted to the stepper motor, a reading is taken followed by the motor spinning 45 degrees. Sequential distance readings are taken in this manner repeatedly. The program is organized based on the format specified in the VL53L1X.pdf documentation, and it uses the Ultra Light Driver for the VL53L1X sensor.

The VL53L1X driver module returns distance measurements in mm to the microcontroller with the use of functions from the VL53L1X C API file. The I2C communication protocol between the microcontroller and ToF sensor allow them to send and receive data from one another. The MSP-EXP432E401Y microcontroller acts as a leader and the VL53L1X sensor is a follower device on the I2C bus, meaning the microcontroller controls the rate of serial communication.

The program starts by including the necessary header files for the MSP microcontroller, such as PLL.h, SysTick.h, uart.h, onboardLEDs.h, and tm4c1294ncpdt.h, as well as the VL53L1X_api.h header file for the VL53L1X sensor. These files contain supporter functions for system clock timing, UART serial communication with the computer, LED flashing, and the sensor's operation.

The program defines some constants for I2C communication, as well as the maximum number of receive attempts before giving up. It also includes function prototypes for I2C initialization, PortG and PortH initialization, and a delay function.

The I2C_Init() function initializes the I2C0 module of the MSP microcontroller by configuring the necessary GPIO pins (PB2 and PB3) as I2C pins and setting the I2C Master Control Register (MCR) to enable the I2C Master Function. It also sets the I2C Master Timer Period Register (MTPR) to configure the I2C clock speed to 100 kbps with glitch suppression of 8 clocks.

The PortG_Init() initializes the PG0 GPIO pin for controlling the XSHUT pin of the VL53L1X sensor. This optional pin is used to reset the sensor before communication. PortH_Init() initializes pins [PH3:PH0] which control the stepper motor. The function stepperSpin() makes the the motor rotate a single step (0.7 degrees). motorPoll() performs continous steps to rotate a total of 45 degrees. This is so that the ToF sensor takes distance readings in 45 degree increments. Following each step, PJ1 is polled for if the button has been pressed, signifying the motor should stop measurements and the program should restart. PortJ_Init() is also necessary to initialize the onboard push button J1 of the MSP microcontroller.

The main() function starts by calling the SysTick_Init() function to initialize the SysTick timer for timekeeping. It then calls the I2C_Init() function to initialize the I2C module of the MSP microcontroller. Next, it calls the PortG_Init(), PortH_Init(), and PortJ_Init() and then prints welcome messages with UART and a serial print buffer to signify the beginning of the program and test functionality of the serial communication port. The sensor boot is attempted repeatedly if failed. The onboard LEDs are flashed and another message is sent serially to signify successful sensor initialization.

The program enters an infinite loop and repeatedly performs the following steps:

1. Calls the VL53L1X_StartRanging() function from the VL53L1X_api.h library to enable distance ranging
2. Polls for if J1 is pressed to begin data acquisition. Program waits until pressed
3. Serially prints a flag to tell the PC to get ready for data storage with Python
4. Enters a loop to get 8 readings from the ToF sensor. Performs the following steps:
    - Calls the VL53L1X_GetDistance() function from the VL53L1X_api.h library to start distance measurement with the ToF sensor
    - Calls the VL53L1X_CheckForDataReady() function from the VL53L1X_api.h library and waits for if a distance measurement from the ToF sensor is ready for transmission
    - Flash the Measurement LED to signify successful measurement acquisition
    - Sends the distance value over UART for further processing and visualization.
    - Blinks the onboard LEDs to indicate the progress of the program.
    - motorPoll() is called to rotate the motor and see if J1 has been pressed to reset the program. If so, the infinite loop is restarted

5. Calls the VL53L1X_StopRanging() function from the VL53L1X_api.h to shut down the ToF sensor

Every run of the inner loop, it is assumed the that the user manually displaces the device such that a new plane of distance measurements are acquired. The Python program stores 24 sets of coordinates before mapping the point cloud, so with the ToF sensor transmitting 8 distance reading per plane, that totals 3 planes. The PC tracks the plane positions; the displacement of the sensor's Z coordinate. The user can modify the automatic Z coordinate increments for desired sequential Z-axis displacements. The variable increments every 8 distance readings such that the visualization accurately maps a 3D environment with vectors. X and Y coordinates are calculated with the formulas

$$x = distance * sin(angle),$$

and

$$y = distance * cos(angle),$$

by the PC on Python after a distance measurement is serially received. `angle` represents the angle that the sensor points at a given time. `angle` is tracked with a variable in Python that increments by $\frac{\pi}{4}$ every time distance data is received, starting at 0. This follows the stepper motor's 45 degree rotations. For example, pointing the first measurement would have `angle` = 0 degrees and Z = 0. So, if for example the distance measurement was 500 mm, the X and Y coordinates would be,

$$x = 500 * sin(0) = 500mm,$$

and

$$y = 500 * cos(0) = 0mm.$$

Each set of XYZ coordinates is exported to a file `tof_radar.xyz`. Each 8 entries in the file represent a single plane.

3.2 Visualization

Environment visualization is handled by the PC with Python. The program uses Open3D v0.17.0 API for 3D simulation. The program serially receives the distance readings from the microcontroller unit (MCU) to create a 3D point cloud. Python creates a new file named `tof_radar.xyz` for writing the coordinate sets, where each newline has a set. Distance data is read from the MCU for 24 iterations, each time converting the received bytes into a string and extracting a numerical value. The received data represents distance measurements in polar coordinates (distance and calculated angle as mentioned earlier) from VL53L1X sensor. The polar coordinates are converted to Cartesian coordinates using the trigonometric functions as mentioned before. The Cartesian coordinates are written to the file and then closes after all 24 vertices have been written. The file data is then used to create a point cloud visualization with Open3D. The point cloud only maps disconnected vertices, so the program defines lines connecting the vertices in the point cloud to visualize the structure. This takes the form of a nested array LineSet object that holds two Cartesian coordinates to be connected at each array outer array index. Using the point cloud data and the LineSet, Open3D's draw_geometries function is called to display the point cloud with vectors connecting the vertices to resemble the physical environment.

4 User Guide with Application Example

To setup the Time of Flight Spatial Mapping System some programs need to be preemptively installed. The following steps will ready the system on Windows 10/11

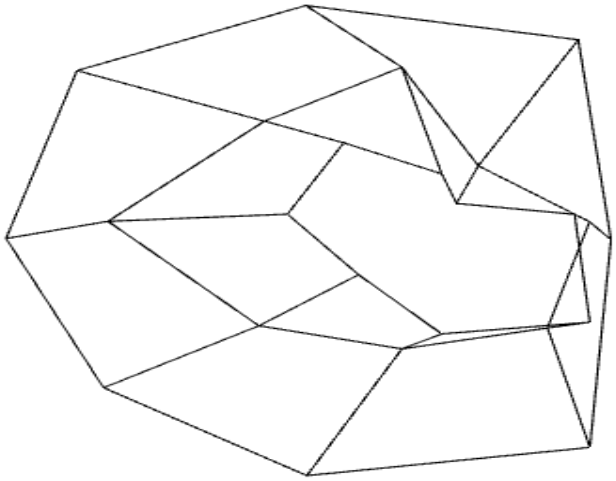
1. Download and install Python 3.10. The latest release is available at https://www.python.org/downloads/release/python-3100/.
2. Run the downloaded installer executable file.
3. Select any desired custom installation settings. Once you have made your selections, click on the "Next" button.
4. Click on the "Install Now" button.
5. After Python is installed, open command prompt as an administrator mode to install Open 3D and PySerial. In the command prompt, type in `pip install open3d` and hit enter. Wait for all the packages to install correctly. Then type `pip install pyserial` and press enter. Once again, wait until installation has concluded. Both APIs should be installed

Your Windows PC should now be compatible with the Time of Flight Spatial Mapping System. To use the system follow these steps:

1. Open the 3D `Mapper.py` file in an IDLE of choice. IDLE is part of your Python installation. Right-click the file and select "Edit with IDLE". Certain parameters can be changed, like `z` and its increment values
    - The increment of the Z displacement can be changed by editing the value on line 60. The value is in mm
    - The COM port being used must be set on line 12. To determine the COM port being used for UART serial communication, enter your PC's Device Manager and select "Show hidden devices" under View. The COM port can be found in "Ports (COM & LPT). Change the COM port on line 4 to the one listed by "XDS110 Class Application/User UART"
2. Once 3D `Mapper.py` has been modified, hit run
3. Connect your PC to the MCU by using the USB-A to micro USB-B cord
4. Press reset on the MCU
5. Aim the sensor at 0 degrees relative to the floor
6. Press J1 to begin distance capture
7. A successful scan is indicate by a flash of the measurement LED D1. Once 8 flashes occur displace the sensor by walking forward the value specified by the `z` variable
8. Repeat the previous step once more and once capture is complete the interactive Open3D simulation will open on your PC

*Note: The stepper motor rotates along the XY plane. The device parses the full 3D simulation into vertical XY planes. A suitable `z` step should be determined by the user to have an accurate simulation. It should complement the displacement taken by stepping forward (Z axis).

4.1 Application Example



Open3D Hallway Simulation

Scanned Hallway

5 Limitations

1. The MCU has a 32-bit Floating-Point Unit (FPU) (single-precision) that supports primitive math operations. The MCU sends and receives distance readings as integers to the PC, where the data is processed with trigonometric functions in Python to avoid inaccuracies on the MCU. Floating points on the PC are limited to the amount of bits used to represent a float, so error propagates for more precise number. Type conversion has been implemented to help mitigate this issue. Since coordinates are already on the measure of mm, no significant error propagates for minor rounding errors.

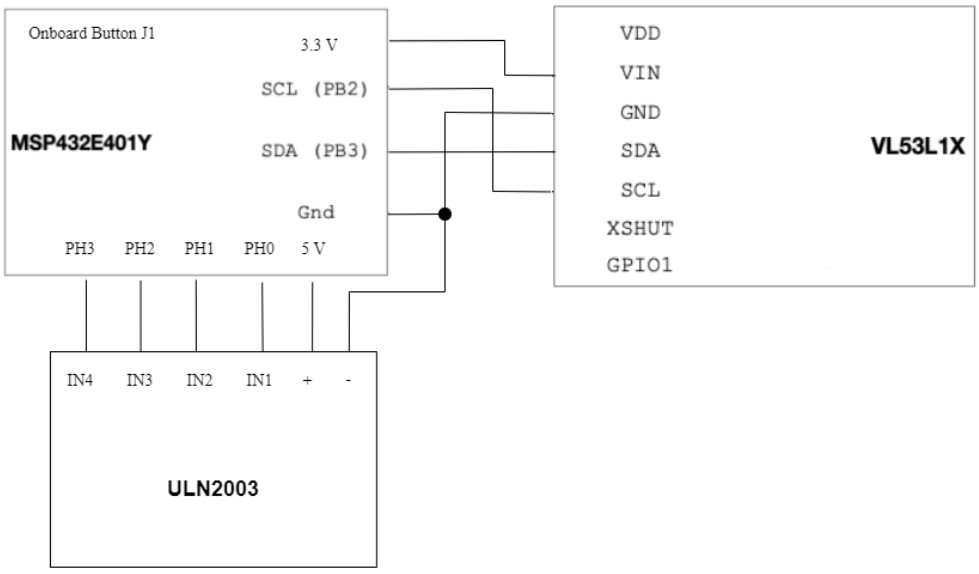
2. V_{FS} = full scale voltage = 3.3 V
   M = bits = 8
   Resolution = $V_{FS}/2^M$ = $3.3/2^8$ = 12.89 mV

3. The maximum standard serial communication rate that can be implemented with the PC is 128000 bits per second. This was verified by checking the port settings for the XDS110 UART Port with the PC's device manager.

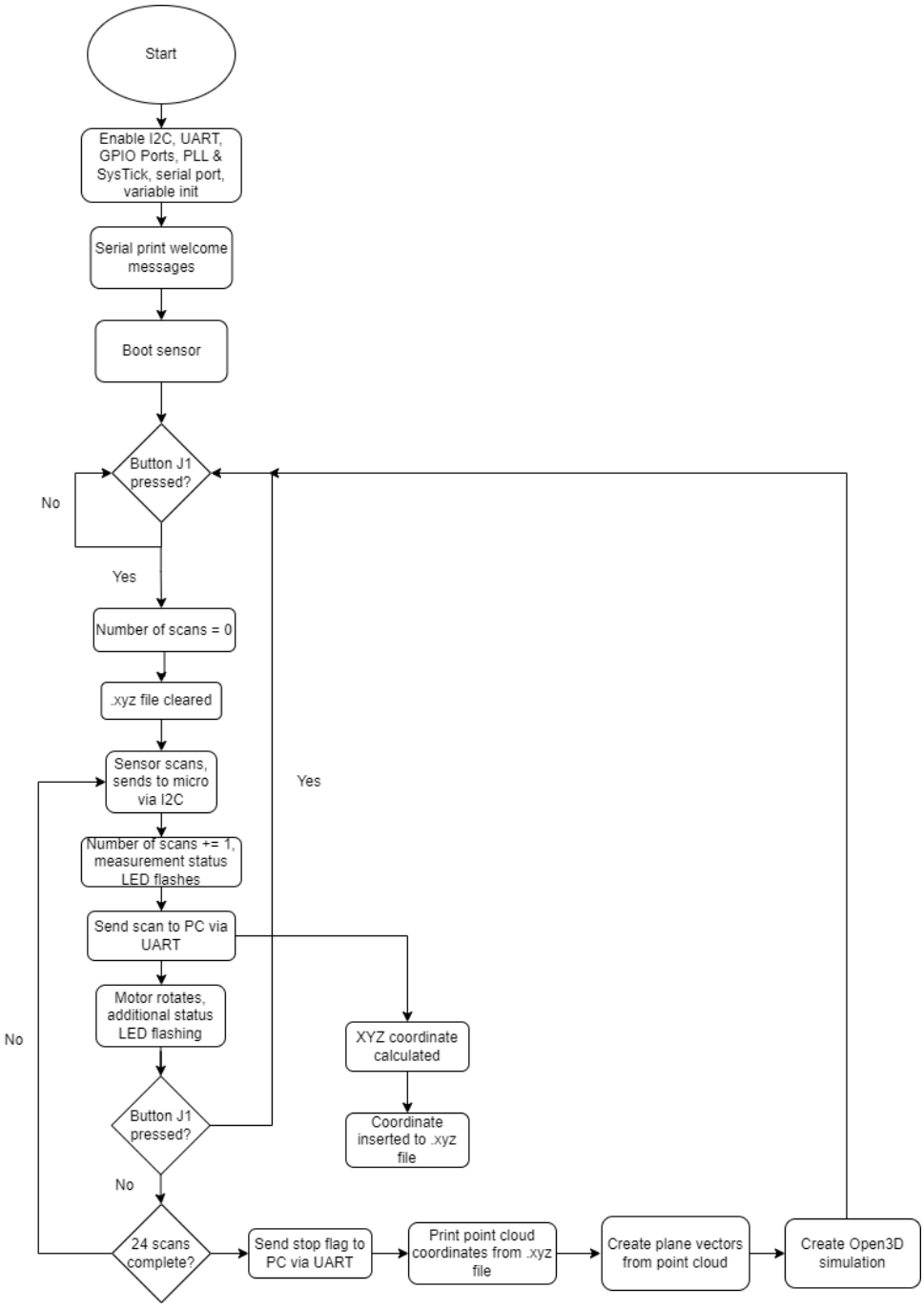
4. The The MCU and ToF sensor used I^2C communication. The speed of communication between the two is 400 kbps as the I^2C bus on the VL53L1X has a maximum speed of 400 kbps.

5. The ToF sensor and stepper motor are the bottleneck devices of the systems. The stepper motor's requirement for SysTick_Wait to rotate means that time has to elapse before it can sequentially function. The same goes for the ToF sensor, with booting for example. Overall, the ToF sensor takes more time to operate than the motor visibly when a distance reading is taken. ToF ranging still takes up a more significant portion of time at their maximum speeds. This was tested by tweaking SysTick_Wait values. For the stepper motor, if the delay was too short it would not rotate but the program would keep running, but for the ToF sensor it would not report any data and cause the system to be halt.

6 Circuit Schematic



7 Programming Logic Flowchart


Start
Enable I2C, UART, GPIO Ports, PLL & SysTick, serial port, variable init
Serial print welcome messages
Boot sensor
Button J1 pressed?
No
Yes
Number of scans = 0
.xyz file cleared
Sensor scans, sends to micro via I2C
Number of scans += 1, measurement status LED flashes
Send scan to PC via UART
Motor rotates, additional status LED flashing
Button J1 pressed?
Yes
No
XYZ coordinate calculated
Coordinate inserted to .xyz file
No
24 scans complete?
Send stop flag to PC via UART
Print point cloud coordinates from .xyz file
Create plane vectors from point cloud
Create Open3D simulation