COMPENG 2DX3: Microprocessor Systems

Final Project: 3D Spatial Mapping LIDAR Device

Instructors: Drs. Athar, Doyle, Haddara

Veronica Marrocco

1.0. Device Overview 1.0.

1.1. Features

- Integrated LIDAR system
  - 360 degree distance measurements to fully map y-z plane at a particular x-location
  - Accuracy of 11.25 degrees, i.e. 32 y-z scans per 360 degree revolution
  - Orthogonal x-displacements for full 3D mapping of space
  - 3D visualization of data
  - Total cost of $145.00 per device (not including PC)
- Texas Instruments MSP432E401Y microcontroller Launchpad
  - Arm Cortex-M4F Processor Core
  - 60 MHz bus speed
  - Onboard LEDs; PN1 for when the device is actively collecting data (i.e., rotating motor); PN0 for every 11.25deg during measurement periods to indicate current sensor measurement (x-y-z point) complete
  - Onboard push button; PJ1 for pausing rotation/measurement
  - 3.3V and 5V tolerance
- Active-low external push button on PM0 to start measurement periods
- VL53L1X ToF sensor with built-in ADC
  - Provides accurate distance measurement up to 4m in range
  - Sampling frequency of up to 50Hz
  - 2.6-3.5V operating voltage; 3.3V from microcontroller used in this system
- 28BYJ-48 stepper motor and ULN2003 driver chip
  - 2048 full steps required for 360 degree rotation
  - 5-12V operating voltage; 5V from microcontroller used in this system
- Plastic housing to mount the ToF sensor atop the rotating stepper motor (**figure 6.5**)
  - Designed in AutoCAD and 3D printed
- Inter-device communications
  - I2C communication between the VL53L1X sensor and the MSP432E401Y
  - UART serial communication between the MSP432E401Y and a Windows 10 PC with baud rate 115200 bps over USB A 2.0
- Data visualization
  - Custom Python application based on Open3D API for processing of raw (x,y,z) data and 3D visualization in a point-cloud format with connector lines

1.2. General Description

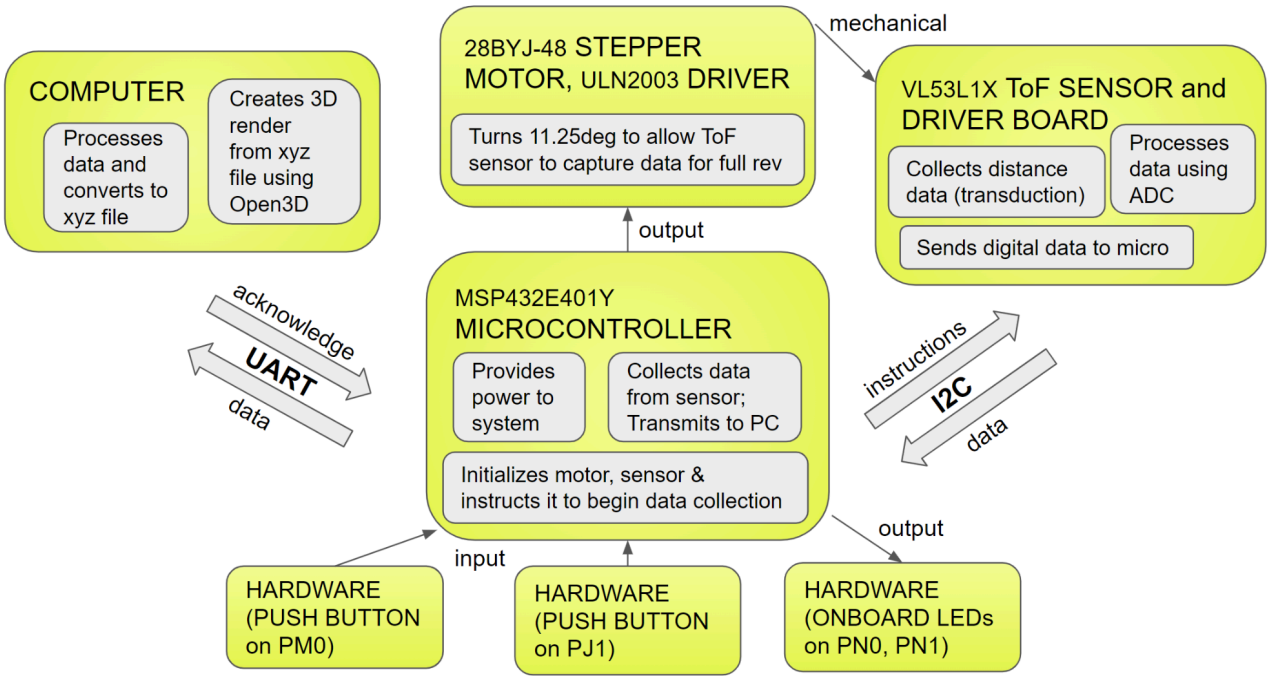
This is an integrated 3D scanning system that is capable of recording distance measures in successive 360 degree planes (**y-z**) along an orthogonal axis (**x**) and process the recorded data to produce a 3D render of the space that was mapped. The system consists of an active-low push button to initiate measurement cycles, the VL53L1X time-of-flight sensor, the 28BYJ-48 stepper motor, the MSP432E401Y microcontroller, its onboard elements (a push button for pausing measurements and LEDs for visual status updates) and a PC for data visualization. The microcontroller is responsible for managing all operations of the system excluding visualization, providing power to the peripheral components and communicating with them in serial to control functionality and acquire data measurements from the ToF sensor. Indeed, the distance data is communicated to the microcontroller from the ToF sensor over I2C and processed to obtain cartesian coordinates, which are subsequently sent to the PC over UART at a baud rate of 115200 bps so that they may be visualized.

The time of flight sensor captures distance data using LIDAR technology. It emits a 940nm infrared lightwave, and measures the time it takes for this wave to return. The obstacle distance (mm) is hence the speed of light (mm/s) multiplied by one half the roundtrip travel time (s). The VL53L1X sensor's internal Analog-to-Digital converter (ADC) digitizes the initial analog data (in millimeters) such that the microcontroller receives digital values when communicating with the VL53L1X module and no additional signal conditioning nor ADC is required.

The stepper motor gives the ToF sensor a 360 degree range of motion such that it may record distance measurements in the full y-z (vertical) plane. The motor rotates 11.25 degrees between each measurement, allowing the sensor to capture a total of 32 data points per 360 degree revolution. The displacement (x) data is captured manually by moving the device in increments of 800mm along the x-axis and repeating the 360 degree scan to map the y-z plane in this incremented x-location. Once he/she is set up and ready in the x-domain, the user must press the external push button to initiate the new y-z series of measurements, and continues as such until sufficient x-locations have been scanned.

By interfacing with the microcontroller over UART, a Python program is able to read the cartesian coordinates line by line as they are processed in real-time. These datalines are parsed and split into their x, y and z components, subsequently written to an .xyz file. We feed this file to the Open3D software, creating a mesh by connecting points within an individual plane and then connecting the planes. Finally, a 3D render is generated for visualization.

1.3. Block diagram (data flow graph)



2.0. Device Characteristics 2.0.

Device	Specification	
MSP432E401Y Microcontroller	Bus Speed	60 MHz
	Baud Rate	115200 bps
Computer	Serial Port	COM4
Software	Python	3.7.4
	Open3D	0.7.0.0
	Numpy	1.16.5
	Pyserial	3.4
VL53L1X ToF Sensor	Pins	Microcontroller Connection
	VIN	3.3V
	GND	GND
	SDA	PB3

	SCL	PB2
28BYJ-48 Stepper Motor and ULN2003 Driver	Pins	Microcontroller Connection
	VIN	5V
	GND	GND
	IN1	PH0
	IN2	PH1
	IN3	PH2
	IN4	PH3
Miscellaneous	Circuit Part	Microcontroller Connection
	Measurement LED (flashes every 11.25° of motion)	PN0
	Status LED (ON when device is in action)	PN1
	External Push Button	PM0
	Onboard Push Button	PJ1

3.0. Detailed Description 3.0.

3.1. Data Acquisition

The data required to generate a 3D mapping of space is provided by the VL53L1X ToF sensor, which returns digital values of surrounding object distances using LIDAR technology. The sensor works by emitting a 940nm infrared lightwave towards the target and waiting for the reflected wave to return, thereby acquiring the wave's total, roundtrip travel time. The target distance (in milliliters) is then computed by the VL53L1X sensor module as the speed of light (mm/s) multiplied by one half the total roundtrip travel time (s).

$$distance = \frac{time\ of\ flight}{2} * speed\ of\ light$$

The VL53L1X sensor IC contains an internal transduction and ADC unit, hence all the signal processing is completed automatically and a single, digital distance measure is relayed to the microcontroller over I2C. This distance measure is expressed in millimeters.

The data acquisition program is written in c under a Keil software project, as outlined in the flowchart in **figure 6.1**. The MSP432E401Y microcontroller is first initialized by setting up all the relevant GPIO ports (B, J, H, M, N), I2C and UART functionality. The ToF sensor is

booted and initialized under default settings using functions from the VL53L1 Core API as distributed by STMicroelectronics. The microcontroller begins to poll for a user input through PMO which is connected to an active-low push button. Once the button has been pressed, a new revolution is initiated for a particular x-value; the microcontroller reinitializes the ToF sensor and instructs it to start ranging. The distance measurement is acquired, and its corresponding x-y-z coordinate is computed using the trigonometric functions from <math.h>. Once this datapoint is obtained, the stepper motor is rotated clockwise by 11.25 degrees so that the next distance measure may be acquired from the sensor. This process takes place 32 times to achieve (x,y,z) data points over a full 360 degrees (11.25deg x 32 = 360deg). Once the full revolution is complete, the user is instructed to move the device by an increment of 800mm along the x-axis and this is manually accounted for in the c code by adding 800 to the current x-value. In addition, the stepper motor is rotated counterclockwise by 360 degrees in order to return the sensor to its home position and avoid further entanglement of the rotating wires.

Finally, the microcontroller returns to its polling state, waiting for an input from the push button to commence a new revolution at the incremented x-location. This algorithm is repeated until sufficient x-locations have been scanned, as defined (and modifiable) in the Python script.

3.1.1. Useful Formulae

The (x,y,z) coordinates are obtained using trigonometry. Where the ToF returns the distance as a radius from the sensor, the appropriate angle can be determined by summing the number of 11.25 degree increments the motor has moved from its initial position.

$$\theta = i \cdot 11.25 \text{ in degrees}$$

$$\theta = i \cdot 11.25 \cdot 2pi/360 \text{ in radians}$$

The cartesian coordinates in mm can then be obtained by using:

$$x = increment \cdot number\ of\ revolutions\ completed$$

$$y = distance \cdot sin(\theta)$$

$$z = distance \cdot cos(\theta)$$

For example; for the first scan, the ToF sensor is oriented perpendicular to the ground, meaning it should capture horizontal (z) information *only*. Indeed, when **i = 0**, the angle formula above returns 0 x 11.25 = **0deg**. Hence the coordinate formulae return a y-value of y = distance x sin(0) = **0** and a z-value of z = distance x cos(0) = **distance**, as expected.

3.1.2. Data Acquisition from Microcontroller to PC

Once each x-y-z datapoint is processed and collected by the microcontroller, it is sent out through UART to the PC on COM4. A Python program was redacted to intercept this data for storage and visualization. It reads the cartesian coordinates line by line as they are processed in real-time. These datalines are then parsed as strings and split into their x,y and z components, subsequently written to an .xyz file. By default, after 6 sets of data have been received by the PC, i.e. 6 full revolutions at 6 different x-locations (for a total x-displacement of 4.8m assuming increments of 800mm), the data is visualized into a 3D render. Should the user complete another

6 full revolutions, or another, the new data obtained would be appended to the .xyz file and an updated 3D render would be generated. Additionally, the Python script can be modified by the user to personalize the x-increment itself (default 800mm) and/or the total number of x-increments (number of full revolutions) which is by default set to 6 (see section **4.3.6**).

3.2. Data Visualization

After compiling all the necessary data measurements into an .xyz file, they are ready to be visualized. This stage is essential as it allows for the representation and interpretation of the 3D data in a way that can be easily comprehended and validated. Visualization is achieved through a Python program that calls upon Open3D and Numpy libraries to generate a point cloud, as summarized in the flowchart in **figure 6.2**. Numpy is used for the manipulation of arrays and data structuring, while Open3D allows for the reading of the .xyz file as a point cloud and for the 3D rendering itself.

This portion of the Python code utilizes two sets of nested for loops. The first nested loop draws connections between points on the same y-z measurement plane (same x-location) by setting two variables as points, where one point is a step ahead of the other. Such point-pairs are connected together one by one until 32 lines have been drawn across each full revolution plane, where the final point is connected back to the first in order to achieve a closed structure. Therefore, the outer loop runs N times, where N is the number of revolutions (unique x-locations) and the inner loop runs 32 times for 32 points per revolution.

The second nested loop connects each point to its corresponding points in adjacent y-z measurement planes (neighboring x-locations). The outer loop runs N-1 times, where N is the total number of revolutions, since there are only N-1 connections between the N number of planes. The inner loop runs 32 times for 32 points per revolution.

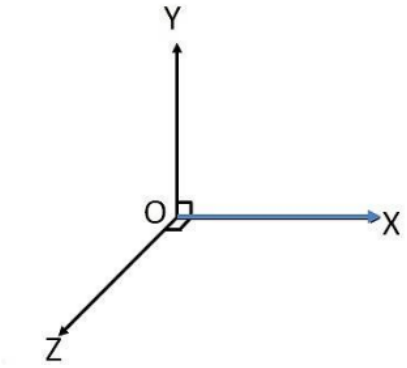
Finally, the visualization function in Open3D is used to initialize and plot a line-set visualization object using the points in the .xyz file. The 3D plot is immediately outputted, providing an easy-to-understand and visually appealing representation of the data.

4.0. Application Example (Setup Guide) 4.0.

The following section takes the user through the installation procedure and device instructions in order to use the embedded device as intended.

4.1. Axis Definitions

- **X-axis:** translated along by user (manual).
- **Y-axis:** captured by rotational scan, perpendicular to the floor (sine of motor angle).
- **Z-axis:** captured by rotational scan, parallel to the floor (cosine of motor angle).



4.2. Software requirements (Windows 10 PC):

- XDS Emulation software for your computer to communicate with the microcontroller through the XDS110 UART port.
- Python (preferably ***not*** 3.10 as there have been issues in its interfacing with Open3D).
- IDLE or other Python IDE.
- 3 Python Libraries: Open3D, Numpy and pySerial. (From the command line, simply enter **pip install open3d** and wait for a confirmation that all packages have been correctly installed. Similarly, enter **pip install pyserial** to install the pySerial package and **pip install numpy** for Numpy).

4.3. Device Setup:

1. Assemble the microcontroller circuit shown as shown in the **figure 6.3. schematic** and **6.4. physical build**. Note that the pin connections are also described in Section 2, device characteristics.
2. Attach the VL53L1X sensor to the stepper motor using the 3D-printed housing (as seen in **figure 6.5**). Make sure your system box and/or housing has a hole carved out for the stepper motor (**figure 6.5**).
3. Connect the MSP432E401Y to the PC via USB.
4. Determine the USB serial communication port that the MSP432E401Y is connected to by following these steps:
    a. Open your PC's Device Manager.
    b. Select "Show hidden devices" under the "View" tab.
    c. Expand the "Ports" dropdown if needed and find the port associated with UART, listed "XDS110 Class Application/User UART". This is your MSP432E401Y port number.
5. Open the Keil project and flash the project c code onto the MSP432E401Y. Don't forget to press the RESET button on the microcontroller.
6. Open the Python script in your choice of IDE (e.g. IDLE), and make adjustments to the code to personalize your use case.
    a. *Absolutely essential* to modify line 7, where it is written
    "ser = serial.Serial("COM4", 115200)",
    such that "COM4" is replaced with your serial port as obtained in step **4.c.** above.
    b. *Optional* to modify the total number of x-locations to be scanned for y-z surroundings. This would be achieved by changing the value of the "**totalrevs**" variable (line 15), which represents the number of full revolutions (at different x-locations) that must be acquired before the 3D visualization begins rendering. Note that "totalrevs" is initially set to 6.
    c. *Optional* to modify the x-increment by editing the variable "**increment**" (line 16) which is set to 800 by default. The value is in millimeters. You can choose to displace the device along the x-axis in larger or smaller steps.

******If you choose to modify “increment” in the Python script, you must also change it in the c code, where the variable name is also “increment”, and reload/reset.

   d. *Optional* to modify the file name where the data is being stored. You can name it anything you like, but the default is “2DXToFdata.xyz”. If you modify the file name, make sure to do so in both lines where it is invoked (19 and 49).

7. The embedded device is now set up and ready to be used.

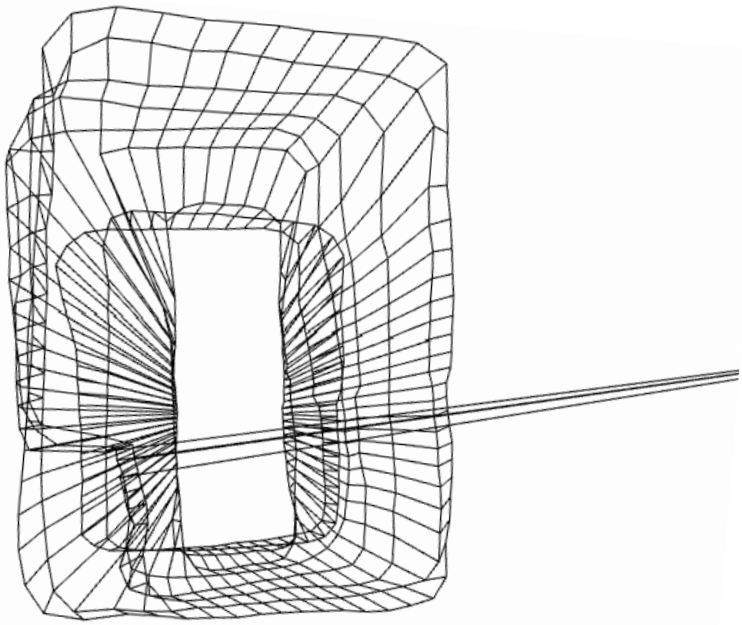
4.4. Taking measurements and obtaining data results:

1. Now that it is ready, you may **run your modified Python script** in your IDE and wait for “Opening: COMQ” to be outputted, where Q is your specific serial port number. Once this message is read, you can be sure the device is prepped for data collection.
2. **Place the physical device** where you want the first sensor reading to take place, that is, somewhere along the orthogonal (x) axis to the plane of measurement. This location will correspond to x = 0mm.
3. **Press the external push button** to initiate the first revolution.
   a. The LED on PN1 (associated with device state) should light up to indicate data collection mode.
   b. Examine the motor and make sure it rotates, then pauses for distance measure, before rotating again.
   c. The LED on PN0 should flash every time an (x,y,z) datapoint is successfully mapped (i.e. with every 11.25deg during the scan).
   d. Watch the Python shell to further ensure these (x,y,z) measurements are coming through as expected.
4. Once the revolution is complete (i.e., 32 measurements), and no new data points are appearing in the Python shell, **move the device forward** along the x-axis by your pre-defined increment (this is 800mm by default).
5. **Press the push button** again to initiate the next revolution.
6. **Repeat steps 4-5** until you have completed the pre-defined number of full revolutions, stored in the “totalrevs” variable in the Python script.
7. Once all revolutions are complete, the **3D render is automatically generated** in Open3D. Look for a new Python window that contains this visualization (it may not pop up in front of the existing windows). You will be able to use your mouse to translate, rotate and zoom in/out of the Open3D render.

****** Note that, if you ever want to **pause measurements** for any reason, you can press the microcontroller’s onboard button **PJ1**, which triggers a GPIO interrupt. To resume measurements, once again, press the external button on PM0.

3.2.1. Data Visualization - Hallway Mapping Example

Below, the hallway (left) was appropriately 3D mapped, as shown to the right. Additional images are available in **figure 6.6**. Observe the shape of the garbage cans carved into the bottom left, and also how the opening becomes smaller as the device scans the narrower portion of the hall. Note that the protruding line seemingly outwards to infinity is an out of range error due to a hole in the garbage cans.



5.0. Limitations 5.0.

5.1. Computational Considerations: The MSP432E401Y microcontroller uses a 32-bit ARM Cortex M4 which supports add, subtract, multiply, divide, multiply-accumulate and square root operations. It is an EEE754-compliant, single-precision (32-bit) floating-point unit (FPU). Hence, there are only 32 registers which are each 32-bits wide, i.e. they can each store a 32-bit integer or float. Since the IEEE 754 standard 32-bit float has an integer component ranging from 0 to 255, this is a limitation for the application at hand, which must be able to return values up to **4000** in millimeters (maximum range of VL53L1X sensor) with trigonometric manipulation (see formulae in section 3). Thankfully, the architecture of the FPU is such that there is a group of sixteen ***doubleword*** registers, i.e. registers that pair up to store 64-bit floats, also known as doubles, within which mathematical operations may be executed. This precision is sufficient for the application at hand since the data in millimeters being stored and/or fed to trig functions is limited to 4000, which is attainable using 64-bit floats.

Therefore, trigonometric functions from the **math.h** library can be performed on **integer** and **double** variables in the MSP432E401Y, as demonstrated in this project code, to obtain

cartesian coordinates from a radius (VL53L1X distance measure) and angle. Indeed, most variables were initialized as '**doubles**' in the c code, including x, y, z and theta, since the math.h trig functions take and return values in this 'double' format. Moreover, upon each iteration, these variable values were saved to the external PC such that the micro's FPU registers may be freed and reused, never surpassing the occupancy of 32.

5.2. Quantization Error: The ToF sensor has 16 ADC bits, with $V_{max} = V_{in} = 3.3V$, $V_{min} = 0V$ and hence a full scale voltage of 3.3V. Moreover, it measures distances up to 4 meters. Therefore, the maximum quantization error is 0.05 in mV and 0.061 in mm, as per the calculations below.

$$max\ Q\ = 1\ LSB\ = \Delta = \frac{V_{FS}}{2^n} = \frac{3.3\ V}{2^{16}} = 0.05mV$$

$$\text{Or } \Delta = \frac{d_{FS}}{2^n} = \frac{4000\ mm}{2^{16}} = 0.061mm$$

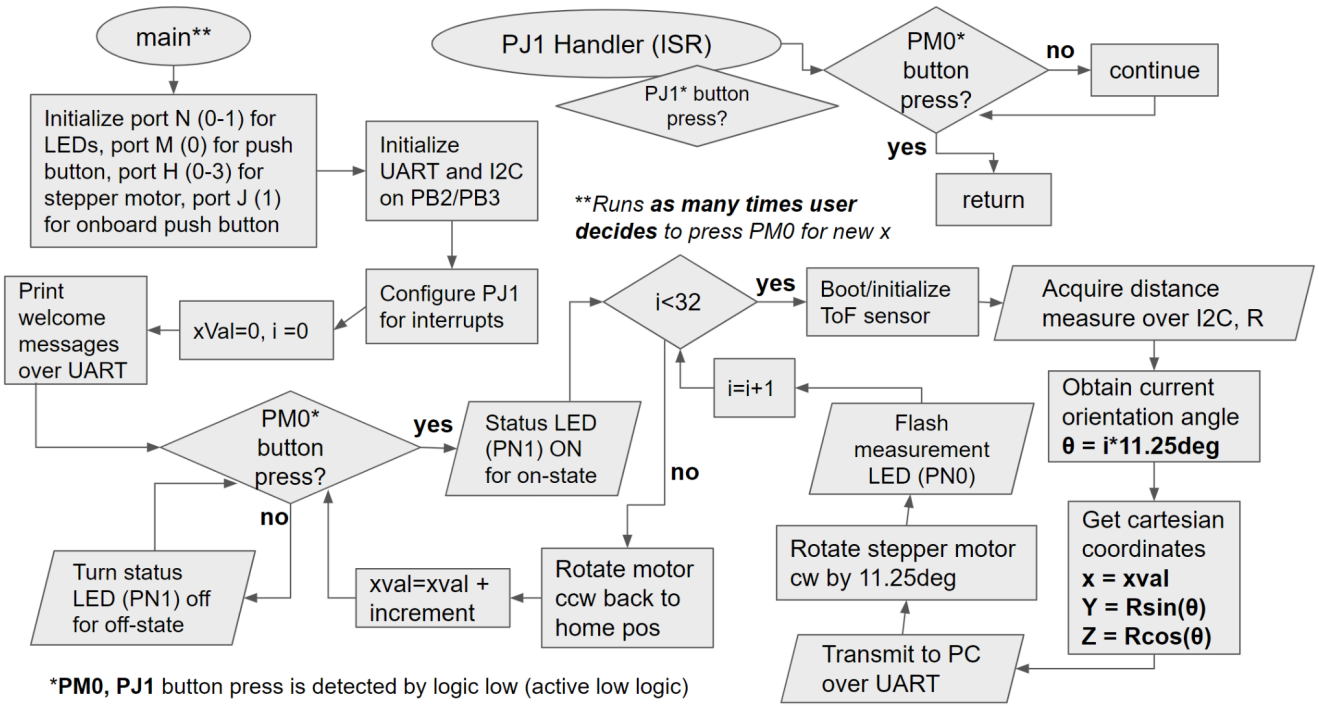
5.3. The maximum standard serial communication rate that can be implemented on the PC is 128 kbps. This was verified in the device manager settings for the XDS110 UART Port. However, this system implements the default, standard speed of 115.2 kbps as taught in lecture and verified in the device manager, plus over Realterm.

5.4. The microcontroller and ToF sensor communicate through I2C with the microcontroller registers used for I2C configured to have a 100 kbps clock.

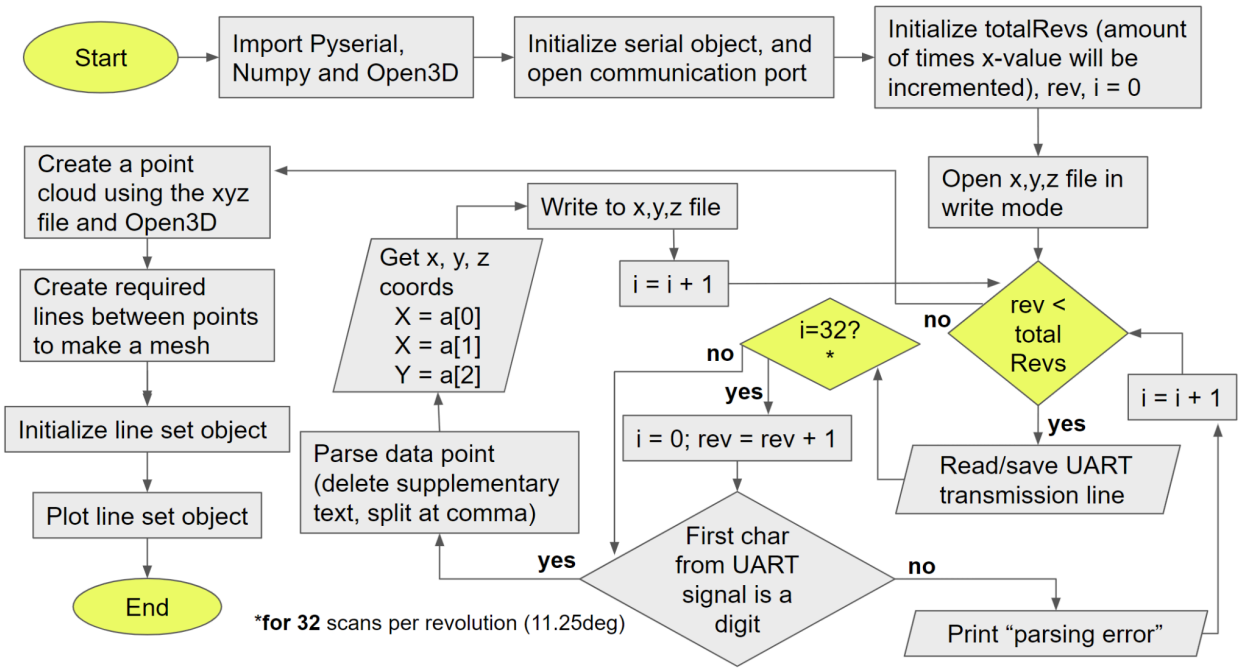
5.5. Speed Limitations: In this embedded system, the two primary limitations on speed are the timing budget of the VL53L1X sensor and the speed of the stepper motor. First, this project involves long-distance measurements, which require time in between sensor measurements for the lightwave to be emitted, reflected and detected. The timing budget for the VL53L1X is set by default at 100ms, and is followed by a between-measurement time delay that must be >= to the timing budget, set to 200ms by default. This results in a minimum time delay of 300ms due to the VL53L1X ranging before a new measurement can be taken. Furthermore, the stepper motor used here has a minimum delay requirement between full steps of approximately **1µs** until breakdown, as seen in testing. Per 2048-step revolution, this yields a total delay of 2048 x 1µs = **2ms** attributable to the motor. To make matters worse, due to the wiring between the stepper motor and microcontroller, the motor must take an extra rotation in the CCW direction after each full CW revolution in order to prevent wire entanglement. If the build quality was improved and this additional rotation was not required, the system's working speed could be enhanced. This was seen in testing when removing the CCW rotation and untangling the wires manually.

During testing, it was notably found that the primary limitation on speed in this system is the ToF sensor ranging delay. While the stepper motor delays could be successfully minimized down to 1µs without compromising the project integrity, reducing the sensor delay led to failure. Indeed, the VL53L1X timing budget is expensive (much more than the stepper motor's **2ms/rev**), as the time delay resulting from individual long-distance measures builds up quickly as the number of measurements per plane increases up to 32. Yet, it was found that trying to reduce this delay or remove it in the code rendered the ToF sensor useless, causing it to fail to report data altogether. The entire system was compromised, verifying that the VL53L1X timing budget is expensive and indeed the primary element blocking our usage of faster speeds.

6.1. Microcontroller Flowchart:

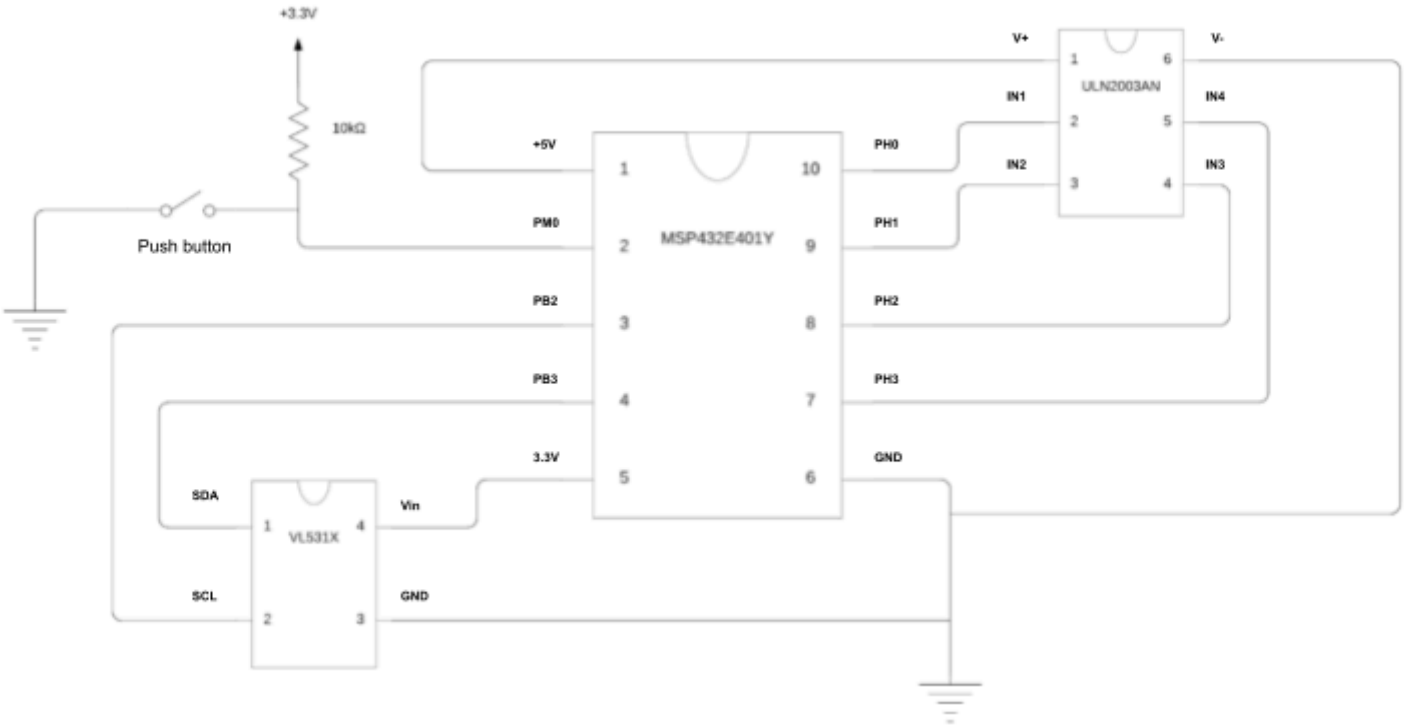


6.2. Python Flowchart:

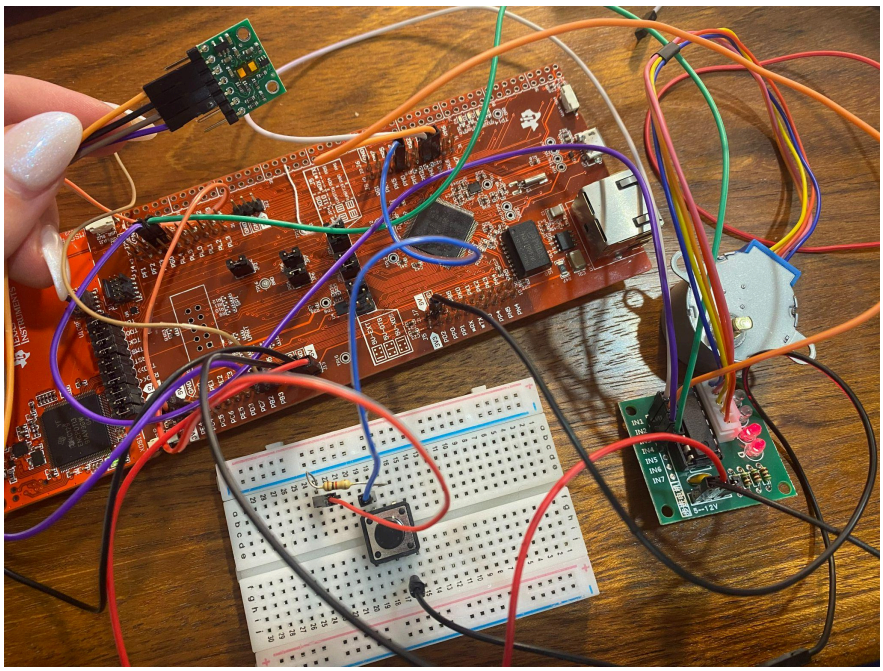


6.3. Circuit Schematic:

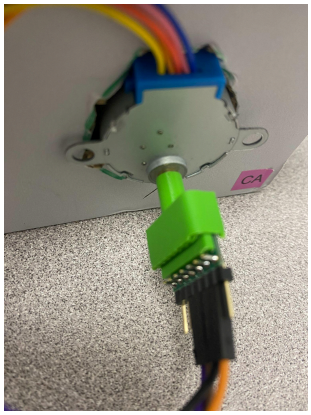
Not pictured in this schematic are the onboard elements; PJ1 onboard push button for interrupt-pause and PN0/PN1 for onboard measurement & additional status LEDs.



6.4. Physical Circuit Build:



6.5. 3D Printed Mounting Piece for ToF Sensor Atop Stepper Motor (Green):



6.6. Additional images of hallway mapping (application example)