2DX4: Microprocessor Systems Project Final Project

Instructors: Dr. Bruce, Dr. Haddara, Dr. Hranilovic, Dr. Shirani

Aaron Pinto – L03

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. Submitted by **[Aaron Pinto]**

Question 1 Video link:
https://drive.google.com/file/d/1CS2EtZJeP8xgb8vXeHuGbhO9tS5VDwR6/view?usp=sharing

Question 2 Video link: https://drive.google.com/file/d/1Pdem5vtm65Fs-o1ehFqSs7K4qqTg-gKu/view?usp=sharing

Question 3 Video link: https://drive.google.com/file/d/1daoA_D-suU48tZ58RWw0vm5RWe2gixX-/view?usp=sharing

Device Overview

Features

- Cost-effective all-in-one lidar system
  - Suitable for indoor exploration and navigation
  - Communicates with a PC via USB
  - 3D visualization of the collected data in Python
- Texas Instruments MSP432E401Y Microcontroller
  - ARM Cortex-M4F Processor Core
  - 12 MHz clock speed
  - 256KB of SRAM
  - LEDs for data transmission and acquisition states
  - Programmed in C (optionally Assembly or C++)
- VL53L1X Time-of-Flight Sensor
  - Up to a 4m range
  - Up to 50 Hz ranging frequency
  - ±20 mm of ranging error
  - 2.6V–3.5V operating voltage
- 28BYJ-48 Stepper Motor & ULN2003 Stepper Motor Driver
  - 512 steps per rotation
  - LED indicators for the step state
  - 5V–12V operating voltage
- Data communication
  - I^2C serial communication between the microcontroller and ToF sensor
  - UART serial communication between the microcontroller and PC
  - Baud rate of 115200 bps between the microcontroller and PC
- Visualization
  - Written in Python 3, specifically 3.6.8
  - Uses the Open3D python package
  - On-the-fly updating as new data is received

General Description

This lidar system is an embedded spatial measurement system which uses the VL53L1X time-of-flight sensor to acquire information about the area around it. It uses a stepper motor to provide a 360-degree measurement of distance within a single vertical geometric plane (y-z) and requires to be manually moved 20cm along the orthogonal axis (x-axis), for each 360-degree measurement. The mapped spatial information is streamed to a connected PC via USB for reconstruction and graphical presentation in the form of a 3D visualization.

The system consists of a microcontroller, a stepper motor and a ToF sensor. The MSP432E401Y microcontroller manages and controls the entire embedded system, distributes power from the PC to the other components, and transmits the measured data to the PC via USB. The stepper motor allows the mounted ToF sensor to have a 360-degree range of motion in 1 axis so that it can record distance measurements in a single vertical plane.

The VL53L1X ToF sensor emits pulses of infrared light and determines the distance to objects by measuring the amount of time the pulse is "in-flight" before being detected. The sensor calculates the distance by converting the measured value from analog to digital and processing it in conjunction with its configuration settings. It sends this data to the microcontroller via I^2C.

The system must be connected to a PC capable of serial communication via USB and running the included python script, lidar_visualization.py. The microcontroller transmits displacement, distance and angle data to the PC via it's UART and USB port. The data is then read, converted into x-y-z values and visualized by the python script.

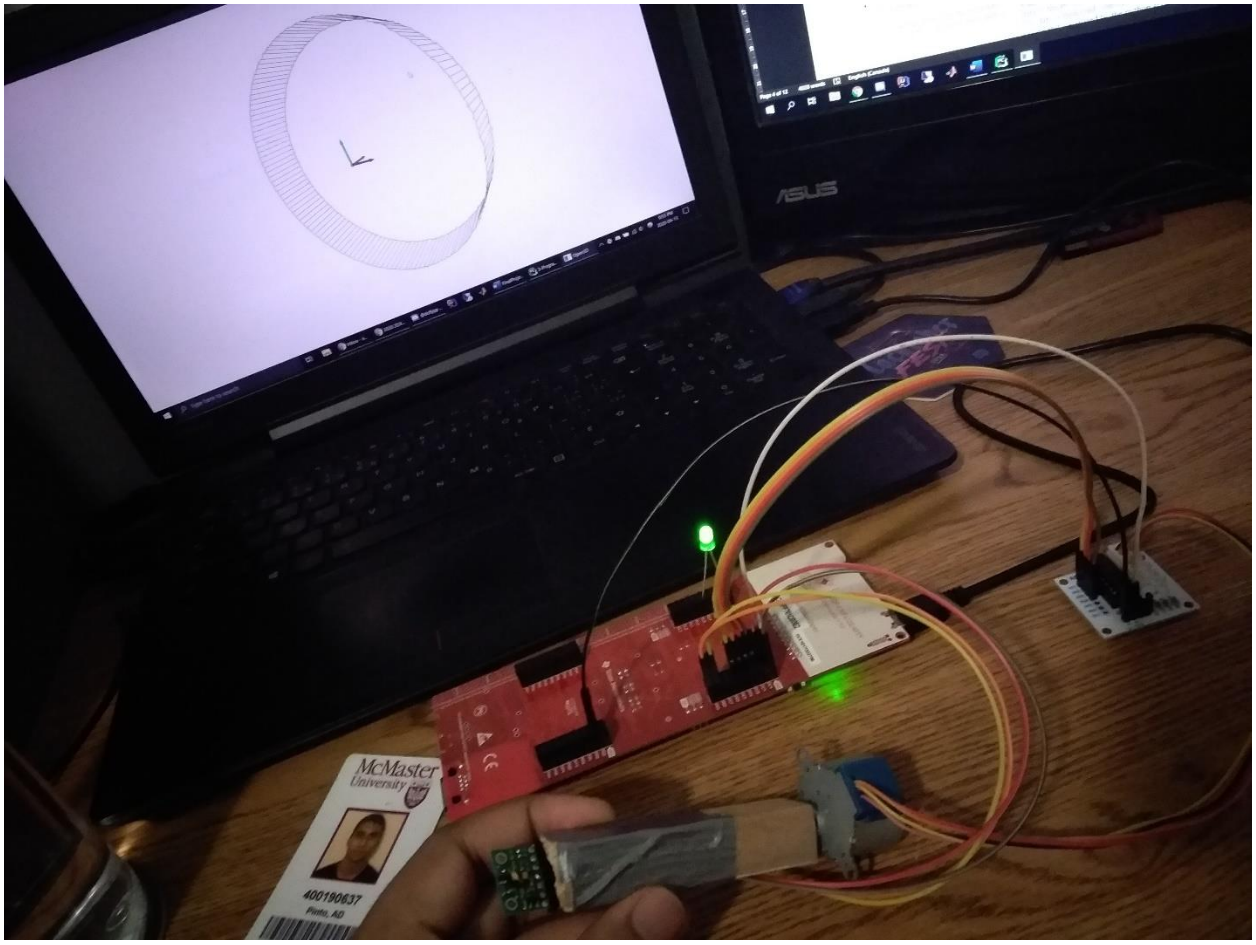

Figure 1. The lidar system hooked up to a laptop displaying a captured visualization

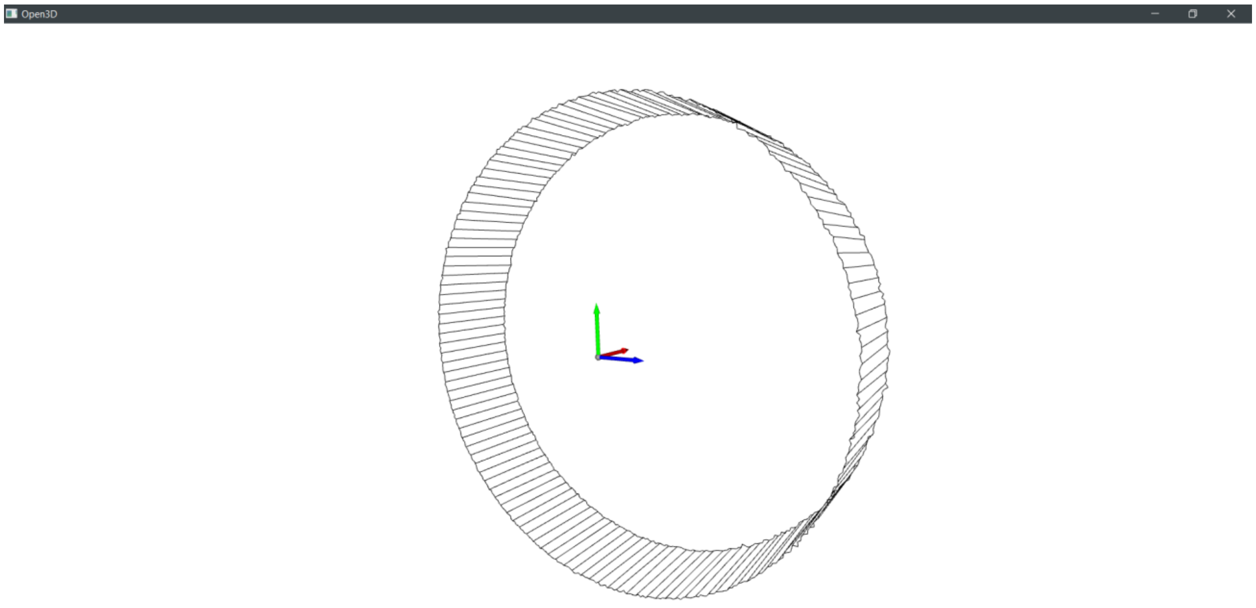

Figure 2. A screenshot of the captured visualization on the PC

Block Diagram

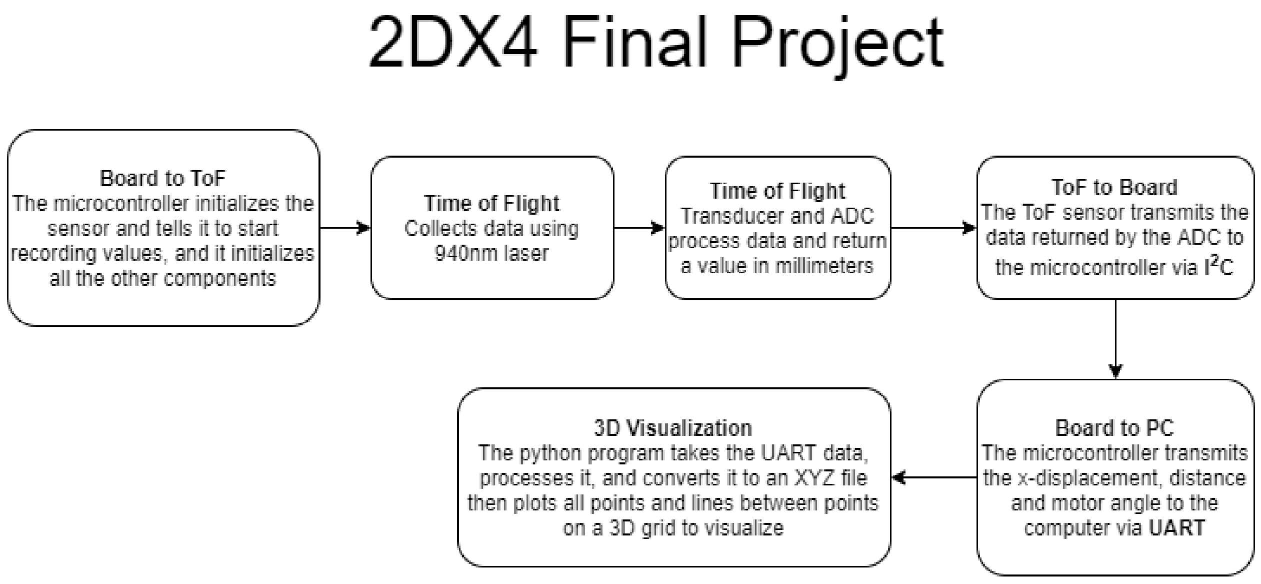
2DX4 Final Project
Board to ToF
The microcontroller initializes the sensor and tells it to start recording values, and it initializes all the other components
Time of Flight
Collects data using 940nm laser
Time of Flight
Transducer and ADC process data and return a value in millimeters
ToF to Board
The ToF sensor transmits the data returned by the ADC to the microcontroller via I²C
Board to PC
The microcontroller transmits the x-displacement, distance and motor angle to the computer via UART
3D Visualization
The python program takes the UART data, processes it, and converts it to an XYZ file then plots all points and lines between points on a 3D grid to visualize

Figure 3. System block diagram

Device Characteristics Table

Specification	Detail	
Bus Speed	12 MHz	
Serial Port	COM3 (UART Port)	
Baud Rate	115200 bps	
Python Version	3.6.8	
VL53L1X Pin Map	Device	Microcontroller
	VDD	-
	VIN	3.3V
	GND	GND
	SDA	PB3
	SCL	PB2
	XSHUT	-
	GPIO1	-
ULN2003 Driver Pin Map	Device	Microcontroller
	IN1	PE3
	IN2	PE2
	IN3	PE1
	IN4	PE0
	-	GND
	+	5V
Green Displacement LED Pin Map	Device	Microcontroller
	Anode	PL4
	Cathode	GND

Table 1. Technical Specification

Detailed Description

Distance Measurement

Distance measurements are performed by the VL53L1X time-of-flight sensor to acquire information about the area around it. It uses a stepper motor to provide a 360-degree measurement of distance within a single vertical geometric plane.

The VL53L1X ToF sensor emits pulses of infrared light and determines the distance to objects by measuring the amount of time the pulse is “in-flight” before being detected. A general equation for ToF sensors is $\mathrm{D} = \frac{t}{2} \cdot c$, where D is the measured distance, c is the speed of light and t is the travel time of the emitted light in seconds. The sensor calculates the distance by converting the measured time value from analog to digital and then passing it through that equation while using its configuration settings to check if the final distance value is out of range or otherwise incorrect. It sends this data to the microcontroller via I^2C.

The ToF settings used to initialize the sensor are the default configuration settings provided in the VL53L1X core API and can be found in the UM2356 (VL53L1X API) user manual. All functions used in this system to communicate with the ToF sensor are provided in the VL53L1X core API distributed by STMicroelectronics or in the platform code provided by the 2DX4 course, with the implementation of these functions slightly modified to decrease the delays in order to improve performance.

The entire process from the acquisition of distance measurement data to the visualization of this data on the PC is outlined in the flowcharts in Figures 5 and 6. In the initial microcontroller initialization process, the I^2C functionality of the microcontroller is enabled, allowing the microcontroller to communicate with the ToF sensor. All variables used for handling the values for the distance data and ToF functions are also initialized. The ToF sensor is booted, initialized and starts ranging, however nothing is done with the acquired data yet.

The onboard button of GPIO PJ1 is polled and used as a toggle to start and stop data capture for the ToF sensor. In order to use the button, the interrupt functionality of the button must be enabled. The current state of this lidar system is stored in the variable **running** of type **bool**, which is **false** (default) when this system is not capturing data and **true** when this system is capturing data. When GPIO PJ1 is pressed and **running** is **false**, it will turn the displacement LED off and set **running** to **true**. When GPIO PJ1 is pressed and **running** is **true**, it will set **running** to **false** and turn the displacement LED on until the button is pressed again, where it will turn the LED off and set **running** to **true** again. All the functions of this system are controlled and managed in 1 main loop, broken up into different blocks that each handle a different part of the system. This main loop always checks the value of GPIO PJ1 at the beginning of the loop and sets the **running** state accordingly, then it handles the data capture based on the current value of **running**, and then based on if its done 1 full rotation.

When this system is in the data capture state (**running** is **true**), the program waits for the sensor data to be ready, extracts the range status and measured distance information when the data is ready and clears the interrupt that is set on the ToF sensor. This communication between the microcontroller and the ToF sensor utilizes the I^2C protocol on both devices. The extracted distance data is stored on the microcontroller and then transmitted to the PC via the microcontroller’s UART and USB port, where this distance measurement data is received and processed by the PC running the lidar_viaulization.py python script.

After the data transmission, the stepper motor moves one step clockwise, the **stepCounter** variable is incremented by 1, and the distance LED flashes if the angle of the stepper motor shaft is a multiple of 45 degrees. If the motor has travelled 512 steps (one full rotation is comprised of 512 steps), **running** is set to **false**, **stepCounter** is reset to 0, **xDistMM**, which represents the x-displacement in millimetres, is incremented by 200, the distance LED is turned off, the displacement LED is turned on, and the stepper motor rotates 1 full rotation counter-clockwise so that the ToF sensor's wires do not risk being damaged. The program then stores the current level state (1 or 0) of GPIO PJ1 in a separate variable (**prevStartStop)**, which is used to determine if the button has been pressed or not on the next cycle, and then cycles back to the beginning of the loop.

The python script, lidar_visualization.py, receives the data sent via USB from the microcontroller by using the pySerial library. This data is in the format of "x-displacement, measured distance, motor angle\r\n" excluding the quotes. The \r and \n are special characters to signal a new line, where \r is CR (Carriage Return) and \n is LF (Line Feed). The python script processes the measurement data to write it to a .xyz file. The x-component of the data is discussed in the displacement measurement section. The y and z-components of the data are calculated using the distance measurement, parsing it to a float, and applying simple trigonometric functions on it. The motor angle is determined by counting the number of steps (**stepCounter)**, dividing by the number of steps in a rotation (512), and multiplying by 360 degrees. The y and z-components are determined by multiplying the parsed distance measurement by the appropriate trigonometric function (the argument of the function is the parsed angle converted to radians). The initial state of the ToF sensor is facing up (0 deg) and it rotates clockwise, so the y-component is the distance multiplied by the sin of the angle $y = d\,sin(\theta)$ and the z-component is the distance multiplied by the cosine of the angle $z = d\,cos(\theta)$.

Displacement

The displacement measurement is currently modified by manually changing the amount **xDistMM** is incremented by on line 114 in **main.c** of the microcontroller program and changing the value of **dist_per_slice** in lidar_visualization.py. When using this system, the x-displacement can be correctly achieved by measuring 200mm on a ruler or measuring tape. Because the x-displacement is manually measured, nothing else needs to be changed in either the Python or C programs before the data can be dumped into the .xyz file.

If the project were to be completed using the IMU sensor, the design of this system would change slightly. The IMU sensor would now determine the **running** state of the system, based on if the system is moving or not, instead of if a button was pressed or not. Also, the IMU's displacement measurement system would need to be integrated into the system, which would communicate with the microcontroller using the I^2C protocol and would eliminate the need of manual x-displacement measurements. This would most likely be implemented using the double integration of accelerometer data to get displacement data (on the microcontroller it would be summation instead of integration). Finally, a displacement-based interrupt system would need to be added, which would automatically begin data capture after the system moved a set x-displacement and stopped moving.

The microcontroller would transmit the new displacement information along with the current distance measurements during data collection, and this would change the implementation of the x-y-z coordinate generation in lidar_visualization.py to account for the angle offset in the x-y plane of the system, because that changes the y and z-components of the data.

Visualization

3D visualization of data is performed by reading the data from the microcontroller, writing it to a .xyz file, creating a point cloud from the data in the .xyz file using the Open3D library, and then using the library's functions to visualize the data.

The visualization is tested on a 15" Lenovo Ideapad 700 laptop running Windows 10 Education Version 2004, OS Build 19041.173. The laptop features a 4 Core/8 Thread Intel Core i7-6700HQ CPU @ 2.60GHz with Intel Turbo Boost enabled, an Intel HD Graphics 530 integrated graphics chipset, an NVIDIA GeForce GTX 950M dedicated GPU with 4GB of DDR3 VRAM, 12 GB of dual-channel DDR4 memory running at 2133 MHz, a Samsung PM951 MZVLV256HCHP 256GB NVMe SSD, and a Samsung 850 Evo 500GB SSD.

The IDE of choice, that all the Python code is run in, is PyCharm Professional version 2020.1. The lidar_visualization.py script was created and run on Python 3.6.8 because Open3D lacks support for Python 3.7+. It should be noted that the visualization script was using the integrated graphics chipset and not the dedicated GPU, and the laptop was plugged into a wall outlet via the included AC adapter to prevent any power throttling.

The Python script requires the Open3D, NumPy and pySerial open-source Python libraries, which include functions to read, format and visualize the measurement data. It also requires the threading, math and queue built-in Python libraries for processing and managing the data in a thread-safe manner.

The script leverages multiple threads to enable visualizing the data "on-the-fly" as soon as new data is received. The first thread is a daemon (runs in the background) that handles the serial I/O using the pySerial library and a custom **ReadLine** class and function to improve the performance over the **readline()** function from the pySerial library. Once it has received data, it splits it by ", " and adds the resulting list to the thread-safe queue. If it does not receive any data, it blocks until new data is available. The main thread handles the parsing of the data to x-y-z coordinates, writing the data to a file and then visualizing the data. It creates a **points** list to store the received points instead of writing it directly to the file so that it doesn't overwrite the file if the data acquisition is prematurely ended. It also defines 2 variables, **num_slices** and **dist_per_slice**, which control the number of slices in the visualization and the x-displacement per slice. If the queue contains data, the main thread removes that data from the queue and checks if the list contains 3 values. If it does it converts each to a float and decomposes the list into the **x**, **dist** and **angle** variables. It converts the angle to radians, and then breaks up the distance into its y and z-components, before appending the x, y, and z values to the **points** list. If the last point in the visualization is received from the microcontroller, all the points are written to a .xyz file and then imported as point cloud using the **read_point_cloud()** function from Open3D.

An Open3D **Visualizer** is initialized to enable the non-blocking visualization which allows for updating the shown geometries in real-time, and a coordinate frame is created and added to the visualizer. The point cloud is also used to create a **LineSet** which contains all the points in the point cloud, as well as the lines that join all the points together in the visualization. The lines join all consecutive pairs of individual points in a slice, and every 4th point between slices. Once the line set is created for a slice, the previous line set geometry is removed and the new line set is added to the visualizer, which then polls for any input events and updates the renderer to show the new geometry. When all the slices are rendered, the visualization window remains open until it is manually closed. A general overview of the Python script is provided in the flowchart in Figure 6.

4) Application Example with Expected Output

In order to use this embedded lidar system, you must first go through a setup process that ensures you have all the necessary software installed on their computer. This setup process is required so that your PC will have all the required drivers to be able to receive and visualize the data from the lidar system. The steps in the setup process are for machines running Windows 10, as they have only been tested on Windows 10.

1. Download the 64-bit Windows XDS Emulation Software from https://software-dl.ti.com/ccs/esd/documents/xdsdebugprobes/emu_xds_software_package_download.html. Run the setup executable and complete the setup, accepting the license agreement, and using the "Typical" setup option. This will allow the user to communicate with and debug the microcontroller through the XDS110 UART port.
2. Download and install Python 3.6.8. The release, specifically the Windows x86-64 executable installer can be downloaded from https://www.python.org/downloads/release/python-368/. Run the setup and select "Add Python 3.6 to PATH" on the main screen. Then select Install Now.
3. After the Python installation is complete, disable the path length limit, then open command prompt in administrator mode by searching "Command Prompt" from the search menu, and right clicking the icon and selecting "Run as administrator." From the command prompt, change directory (cd) into the “3-Program_pintoa9” directory which will be wherever you extracted the project files. Then run “pip install pipenv”, which will install a very useful tool for developing in Python. When that’s done, run “pipenv install”, and wait for it to finish installing all the dependencies.

After the setup, your PC has all the software needed to interact with this lidar system and utilize all its features. Now you are ready to begin collecting and visualizing distance measurement data. The steps in this process are:

1. Connect your PC to the microcontroller by using the micro-USB port on the opposite side of the Ethernet port. It should turn on after you plug it in.
2. Open the lidar_visualization.py python file in an IDE of your choice, preferably one with line numbers. IDLE, which was installed during the Python installation is fine and can be accessed by right clicking the python file and selecting the "Edit with IDLE" option. In IDLE, you will be able to change the user-modifiable parameters of this system.
    - The delta x-displacement between slices can be changed by editing the variable **dist_per_slice** on line 54. Please refer to the Displacement section for more information on the other variable you are required to change if you change this one, or vice-versa. The value of the delta is in mm.
    - The file name where the x-y-z data is being stored can be changed on line 71. Change the value in “quotes” to your desired file name.
    - The COM port being used must be set on line 37. To determine the COM port being used for UART serial communication, open command prompt again, and run “python -m serial.tools.list_ports -v” with the microcontroller plugged into your computer and use the one which mentions UART in it. Alternatively, you can enter your PC's Device Manager and select "Show hidden devices" under View. The COM port can be found under "Ports (COM & LPT). Change the COM port on line 37 to the one listed by "XDS110 Class Application/User UART."

3. Once the lidar_visualization.py python file has been modified, it can be run by opening command prompt and running “pipenv shell”, followed by “python lidar_visualization.py”.
4. Press the button on the same side as the micro-USB port where you plugged in the microcontroller to restart the program. Messages will appear in the terminal of your IDE or the command prompt telling you about the status of your lidar system. Read through the messages and wait until you are given a prompt to press GPIO PJ1. If no messages appear in the terminal, there is likely an issue with the serial COM settings or with the XDS110 UART driver.
5. When you are given the prompt to GPIO PJ1 to start, you may move the lidar system and your PC to the location which you wish to map. You must make sure that any connections from the microcontroller to the PC and from the microcontroller to other components of the system do not come loose.
6. Align your ToF sensor's starting position to be facing up by continually starting and stopping data capture by pressing the GPIO PJ1 button. If the sensor has a different starting position, the visualization will still be correct, however the x-y-z coordinate frame will be with reference to a different orientation. For reference, the positive x-axis is the direction of the shaft of the stepper motor, the positive z-axis (should be straight up) is in the direction where the ToF sensor is initially aiming, and the positive y-axis is rotated 90 degrees clockwise from the z-axis about the x-axis. When the ToF sensor is facing straight up, restart the Python program and the microcontroller.
7. Press the GPIO PJ1 button to begin data capture when prompted again. It is vital to ensure that no wires become loose or unplugged during this stage and that valid distance measurements are appearing on the command prompt or terminal of your IDE. If the values appear incorrect, there may be issues with the ToF sensor or its connection with the microcontroller.
8. Once the stepper motor stops rotating and the displacement LED turns on, you can move the system to the next x-position that you want to map based on the delta set in your python file and **main.c** file. It is fine to record data from a different x-displacement than what was stated, however you will need to manually edit the collected data in the .xyz file to reflect this change.
9. When you’re in position, press the GPIO PJ1 button to start mapping again. Keep mapping for the number of slices that you set in the python file.

Once data collection has been completed, a new window will appear on the PC titled "Open3D". This window contains the 3D visualization of your data. You will be able to use your mouse to rotate and zoom in the visualization. You can also find other controls at http://www.open3d.org/docs/release/tutorial/Basic/visualization.html

5) Limitations

1) The MSP432E401Y has a Floating-Point Unit (FPU) that can support single-precision (32-bit) addition, subtraction, multiplication, division, multiply-and-accumulate, and square root operations. The FPU can also perform conversions between fixed-point floating-point data formats, and floating-point constant instructions. Trigonometric functions from the **math.h** library can be applied on **float** variables. Because the FPU is only 32-bits wide, 64-bit (double precision) operations will take more than 1 instruction to complete as the FPU must split it up into 32-bit words.
2) The maximum quantization error for the ToF module is $\frac{4000\text{mm}}{2^{16}} = 6.10 \times 10^{-2}$ mm. The maximum quantization error for the IMU module is $\frac{32\text{g}}{2^{16}} = \frac{32 \cdot 9.81\text{m}/\text{s}^2}{2^{16}} = 4.79 \times 10^{-3}\ \text{m}/\text{s}^2$.
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 in the PC's device manager.
4) The communication between the microcontroller and the ToF module uses the I^2C protocol with the clock speed configured to 100KHz for a transfer rate of 100kbps.
5) The element that is the primary limitation on speed is the ranging on the ToF sensor. This is because the VL53L1X sensor has a timing budget that can be set from 20ms up to 1000ms. The minimum timing budget that can work in all modes is 33ms, but 140ms is the minimum which allows the max distance of 4m to be reached in Long distance mode. The sensor also has a programmable inter-measurement period, which is the delay between two ranging operations, and the minimum value for this must be larger than the timing budget + 4ms. If we add the minimum delays required for a ranging operation, 140ms + 140ms + 5ms = 285ms is longer than any other operation on this system. In testing, trying to make any of these delays less than their minimum values would cause the ToF sensor to not report any data and cause the entire system to stop, which is expected as the sensor is not designed to operate under those conditions.
6) The necessary sampling rate required for the displacement module is dependent on the highest frequency that can be measured by the IMU. Given that the x-displacement delta is 200mm (0.2m) and the IMU's maximum acceleration rating is 16g, using the kinematic equation $\mathbf{x} = \frac{1}{2}\mathbf{a}t^2 + \mathbf{v_o}t + \mathbf{x_o}$, where x_o and v_o are 0, we are left with $\mathbf{0.2} = \frac{1}{2}\mathbf{(16 * 9.8)}t^2$. Re-arranging the equation and solving for t gives us $t = 0.0505s$ which means the input signal has a frequency of approximately 20Hz. Thus, the necessary sampling rate is twice that, about 40Hz, based on the Nyquist-Shannon sampling theorem.

6) Circuit Schematic

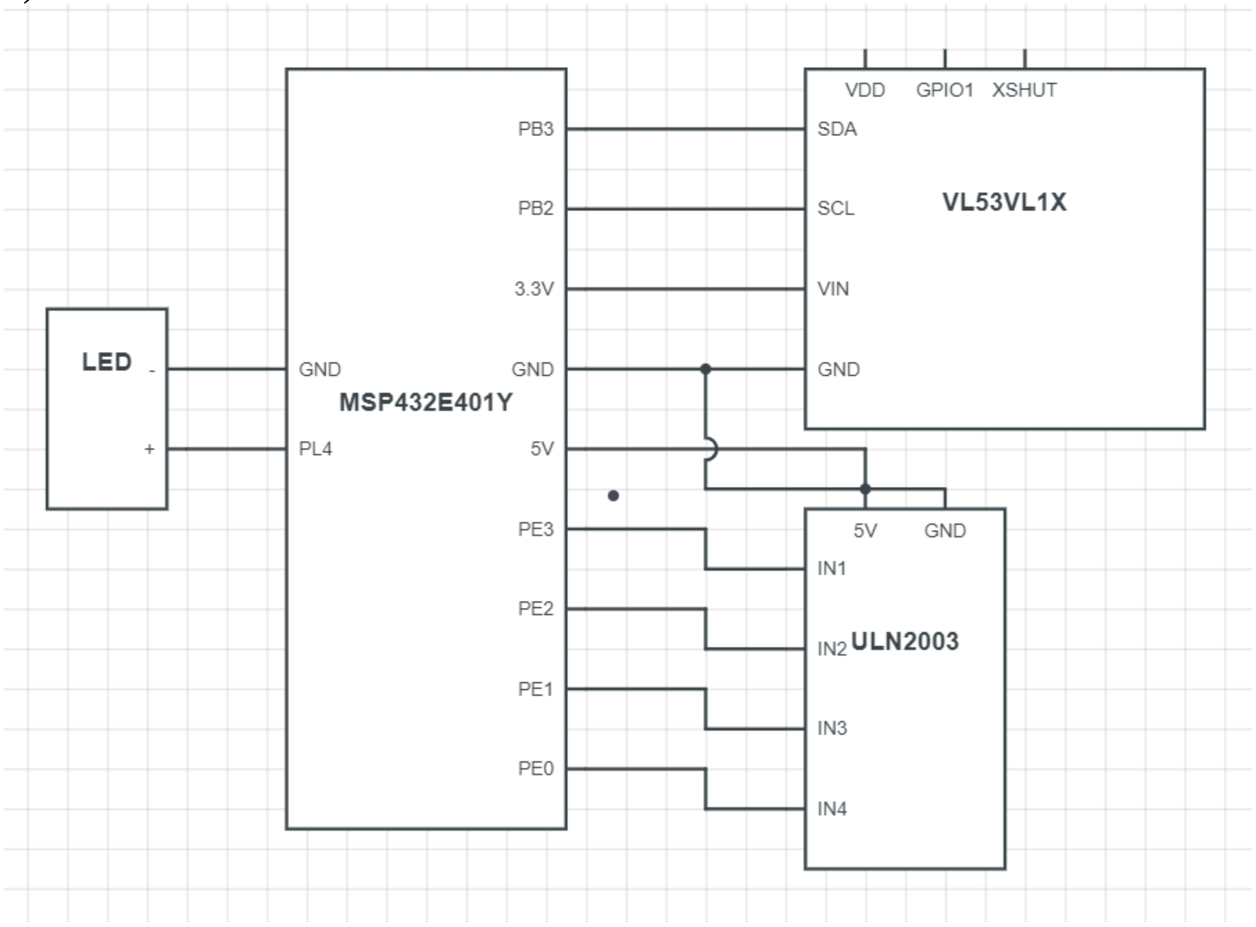

Figure 4. Schematic diagram of system

7) Programming Logic Flowchart(s)

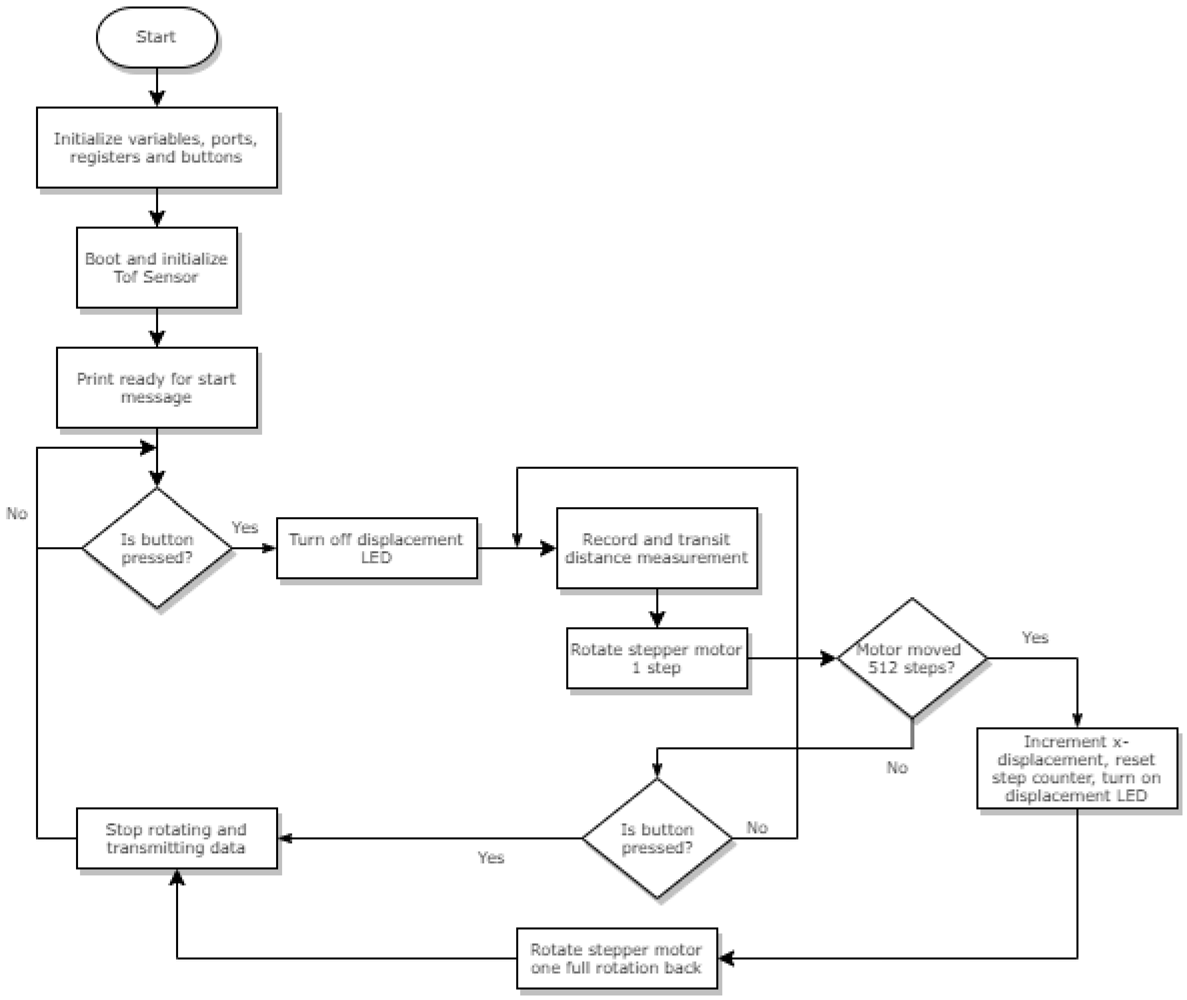
Start
Initialize variables, ports, registers and buttons
Boot and initialize Tof Sensor
Print ready for start message
No
Is button pressed?
Yes
Turn off displacement LED
Record and transit distance measurement
Rotate stepper motor 1 step
Motor moved 512 steps?
Yes
No
Increment x-displacement, reset step counter, turn on displacement LED
Is button pressed?
No
Yes
Stop rotating and transmitting data
Rotate stepper motor one full rotation back

Figure 5. Microcontroller code flowchart

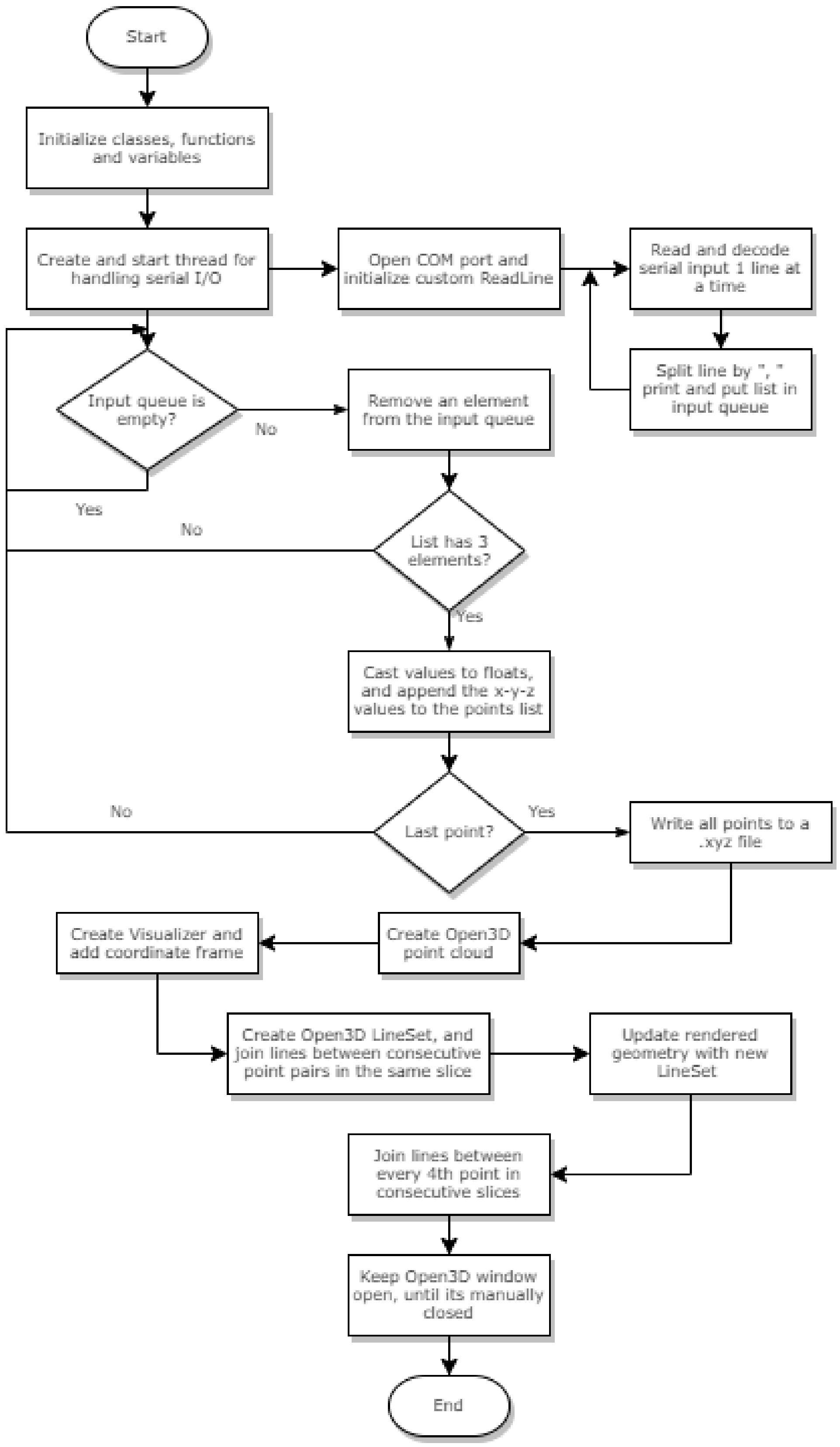
Start
Initialize classes, functions and variables
Create and start thread for handling serial I/O
Open COM port and initialize custom ReadLine
Read and decode serial input 1 line at a time
Split line by ", " print and put list in input queue
Input queue is empty?
No
Yes
Remove an element from the input queue
List has 3 elements?
No
Yes
Cast values to floats, and append the x-y-z values to the points list
Last point?
No
Yes
Write all points to a .xyz file
Create Open3D point cloud
Create Visualizer and add coordinate frame
Create Open3D LineSet, and join lines between consecutive point pairs in the same slice
Update rendered geometry with new LineSet
Join lines between every 4th point in consecutive slices
Keep Open3D window open, until its manually closed
End

Figure 6. Python code flowchart