COMPENG 2DX3

Final Project Report: Spatial Mapping System

Aakash Jain – Jaina78 – 400495996

Instructors: Dr. Haddara, Dr. Athar, Dr. Doyle
Date: Tuesday, April 8th, 2025

Device Overview

Features

The mapping system is driven by a Texas Instruments MSP432E401Y Microcontroller, running C code written in Keil uVision software. The microcontroller has the following specifications:

- Cost –$72.54 CAD (DigiKey Electronics)
- Processor – Cortex-M4 Processor
- Max Bus Speed 120MHz
- Memory – 1024 KB flash memory, SRAM – 256KB, EEPROM – 6KB
- Voltage - Recommended DC operating range 2.97V - 3.63V (3.3V standard)
- Analog to Digital type – 12-bit SAR
- Communication: I2Cs and UART – Serial Communication

The time-of-flight sensor used is the VL53L1X from Pololu, with the following specifications:

- Maximum Measurable distance – 4m and has ranges: short, medium, and long
- Operating Voltage – 2.6V-5.5 V, VDD 2.6V-3.5V when Vin disconnected
- Communication: I2C with a maximum frequency of 400KHz – Serial Communication (TOF to microcontroller)
- Cost –$18.95 USD (Manufacturer: Pololu Robotics and Electronics)
- Up to 50 Hz ranging frequency

This sensor is mounted on a 28BYJ-48 unipolar stepper motor, allowing for 360-degree scanning across a single plane. The stepper motor and its ULN2003 driver board are specified as follows:

- The motor has a 64:1 gear ratio, with two center-tapped coils that control an internal gear, which in turn drives an outer gear
- Operates in full-step mode, with 2048 steps per revolution of the internal gear, corresponding to 512 steps per revolution of the outer gear
- Controlled by the ULN2003 driver board using six wires (four for controlling the coils, one for power, and one for ground)
- The driver operates at 5V (project purposes) or 12V

The system's components communicate using the following protocols:

- I2C serial communication is between the TOF sensor and the microcontroller, with transmission at 100k bits/s configured in C.
- UART serial communication between the microcontroller and a PC, with a baud rate of 115200 bps configured in Python.
- For 3D visualization, Python 3.10 is used with Pyserial and open3D.

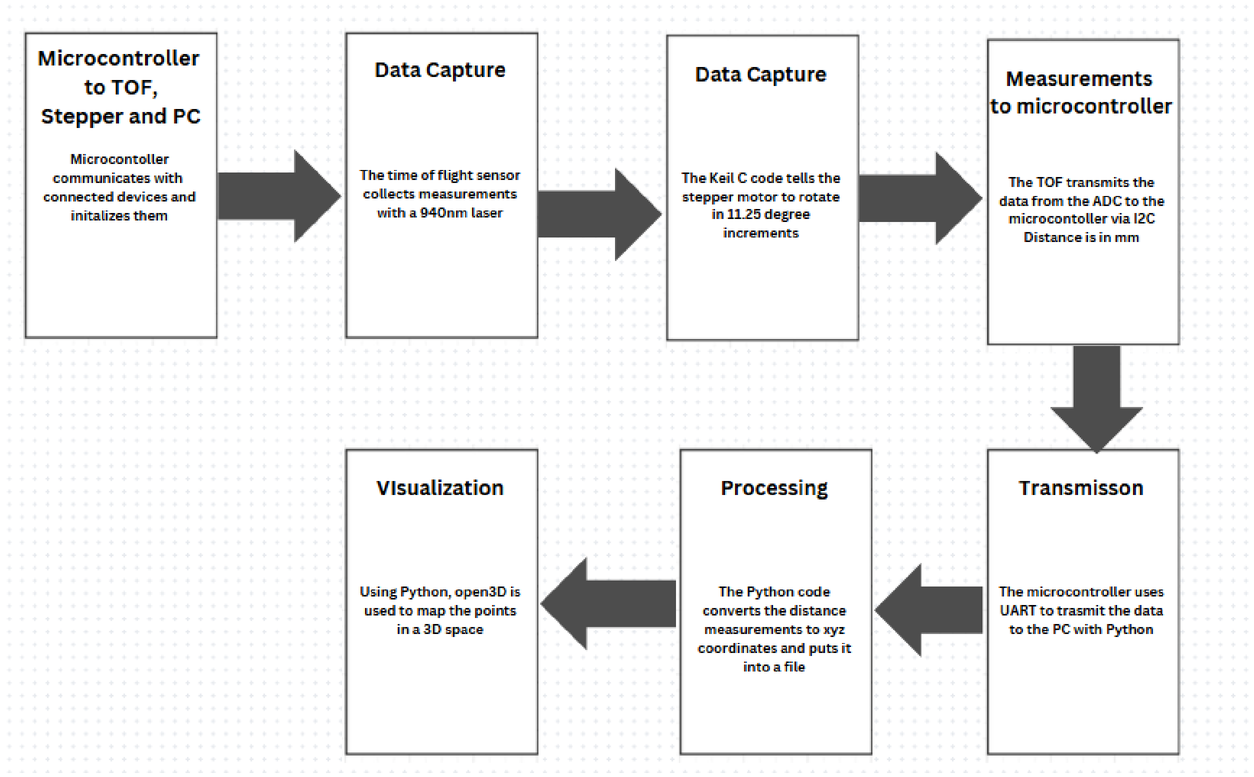
General Description

The goal of this project is to create a 3D model of indoor spaces like hallways and rooms using a time-of-flight (TOF) sensor. The system uses an MSP432E401Y microcontroller, and the VL53L1X sensor to measure distances, an alternative to expensive LIDAR technology. The sensor emits light, which reflects off objects and is detected by the onboard receiver. The distance calculated is in millimeters. Mounted on a stepper motor with a 3D-printed mount, the sensor rotates 360 degrees, taking measurements every 11.25 degrees (32 points per rotation).

The analog data is converted to digital by the ToF sensor and transmitted to the microcontroller via I2C, then sent to a PC via UART serially (COM3, baud of 115200bps). Using Python 3.10, the x and y coordinates are calculated and stored in an .xyz file, while the sensor's movement along the z-axis is manually incremented in the code. The data is processed after each measurement and are combined to create a 3D representation of area, visualized using Open3D on the PC by connecting points from the .xyz file after the scan is complete.

This approach provides a cheaper, more compact alternative to commercial LIDAR systems. The components include the PC, microcontroller, stepper motor, and ToF sensor, all communicating through GPIO, I2C, and UART.

Block Diagram (Data flow chart)



Microcontroller to TOF, Stepper and PC
Microcontoller communicates with connected devices and initalizes them
Data Capture
The time of flight sensor collects measurements with a 940nm laser
Data Capture
The Keil C code tells the stepper motor to rotate in 11.25 degree increments
Measurements to microcontroller
The TOF transmits the data from the ADC to the microcontoller via I2C Distance is in mm
Transmisson
The microcontroller uses UART to trasmit the data to the PC with Python
Processing
The Python code converts the distance measurements to xyz coordinates and puts it into a file
VIsualization
Using Python, open3D is used to map the points in a 3D space

Figure 1: Block Diagram and Data Flow of System

Device Characteristics Table

Table 1: Device Characteristics

Microcontroller TI MSP432E401Y	
Bus Clock Speed	14 MHz
Measurement Status LED	Onboard LED D2 (PN0)
UART Tx LED	Onboard LED D1 (PN1)
Additional Status LED (Motor Running)	Onboard LED D3 (PF4)
Onboard Pushbuttons	PJ0, PJ1
Baud Rate, I2C Transmission Rate	115200 bps baud, 100k bits/s I2C
VL53L1X Time-of-Flight Sensor Pololu	
Supply Voltage (VIN)	3.3V pin
Ground (GND)	Ground (GND) pin
SDA Pin, SCL pin	PB3 pin, PB2 pin
ULN2003 Stepper Motor Driver	
Supply Voltage (VIN)	5V pin
Ground (GND)	Ground (GND) pin
IN1 – IN4 Pin	PH0 – PH3 pins
Software and Tools	
Microcontroller Code	Keil uVision (in C)
Python Version	3.10 with any IDE
Python Tool for 3D Visualization	Open3D
Python Tool for Serial Communication	Pyserial, COM3

Detailed Description

Distance Measurements

In this project, the VL53L1X sensor was used to measure distances. The sensor works by emitting 940nm infrared light pulses and measuring the time it takes for the pulses to reflect off a surface and go back to the sensor. With the known speed of light, the sensor calculates the distance to the surface. Each time the C program requests a new measurement via the I2C communication protocol, the TOF sensor performs the calculation. Once the measurement is complete, the data is sent back to the microcontroller using I2C. The serial data line (SDA) is used to transmit data, and the serial clock line (SCL) provides the clock signal that synchronizes the data transmission between the microcontroller and sensor. The onboard LEDs (PN0 and PN1) blink to signal that the sensor has successfully completed the measurement, and the data has been transmitted. To calculate the x and y coordinates (slices), trigonometric functions are applied, factoring in the motor's angle. The sensor scans at 11.25-degree intervals, collecting 32 measurements per complete rotation. After each scan frame, the z-coordinate (displacement) is incremented by 1000mm. The motor then rotates counterclockwise for one full 360-degree revolution to prevent the wires from tangling up and no measurements are taken during this process.

The distance formula can be defined as $\frac{Time * Speed\ of\ light}{2}$ that translates the value of time to a measurement in millimeters. Example: If the time for a light to go and come back to the sensor is 10ns, we can sub it into the formula above to obtain:

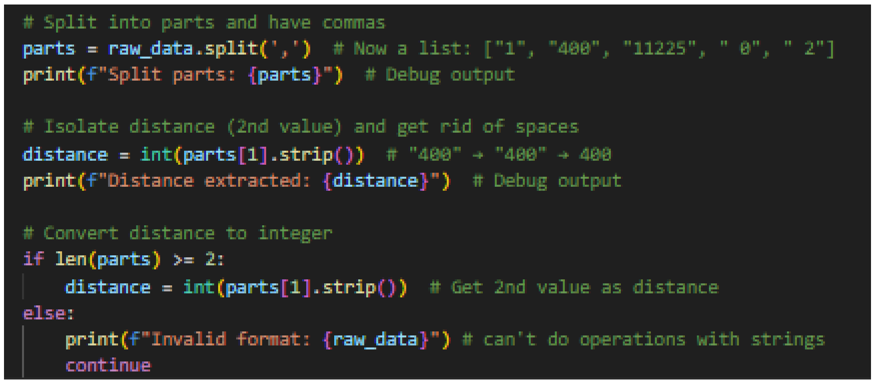
$$\frac{Time * Speed\ of\ light}{2} = \frac{10ns * 300{,}000{,}000.000\ mm/s}{2} = 1500\ mm \text{ (Distance returned is in mm)}$$

At every 11.25-degree interval, the distance measurement is transmitted to the Python code via UART. The received data is in the form of a string containing multiple values, including RangeStatus, Distance, SignalRate, AmbientRate, and SpadNum. Using Python's string functions, such as strip() and split(), the relevant information was extracted and organized into an array. The distance value, located at index 1, was then retrieved for further processing.

```
sprintf(printf_buffer, "%u, %u, %u, %u, %u\r\n", RangeStatus, Distance, SignalRate, AmbientRate, SpadNum);

UART printf(printf buffer);
```

Figure 2: Data String through UART



```python
# Split into parts and have commas
parts = raw_data.split(',')  # Now a list: ["1", "400", "11225", " 0", " 2"]
print(f"Split parts: {parts}")  # Debug output

# Isolate distance (2nd value) and get rid of spaces
distance = int(parts[1].strip())  # "400" → "400" → 400
print(f"Distance extracted: {distance}")  # Debug output

# Convert distance to integer
if len(parts) >= 2:
    distance = int(parts[1].strip())  # Get 2nd value as distance
else:
    print(f"Invalid format: {raw_data}") # can't do operations with strings
    continue
```

Figure 3: Python's String Functions

To calculate the coordinates in the x and y plane, the following equations were applied after converting the angle values to radians using the math library in Python:

$$x = \cos(\text{angle}) \times \text{distance}$$

$$y = \sin(\text{angle}) \times \text{distance}$$

For the z-coordinate (displacement), a predefined array was used, incremented by 1 after every 32 data points collected. This approach allowed the system to track the progression along the z-axis, with each new set of 32 measurements representing one complete frame of data. The displacement essentially adds layers to the scan and turns the imaging 3D.

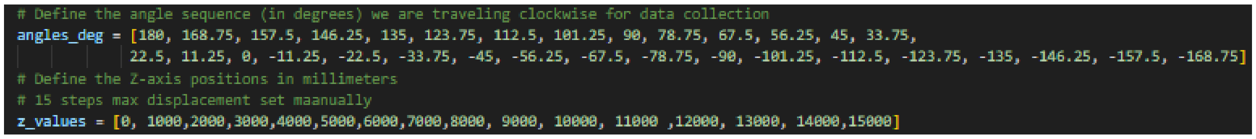
Example calculation - Assume first distance measurement from TOF is 2234mm, the corresponding x, y and z values are as follows for the angle of 180 degrees:

$$x = \cos(\pi) \times 2234\text{mm} = \text{-}2234\text{mm}$$

$$y = \sin(\pi) \times 2234\text{mm} = 0\text{mm}$$

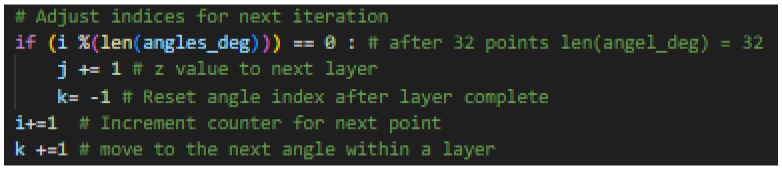
$$z = 0\text{mm (manually set)}$$

These values give you the x-y-z coordinate plane.



```
# Define the angle sequence (in degrees) we are traveling clockwise for data collection
angles_deg = [180, 168.75, 157.5, 146.25, 135, 123.75, 112.5, 101.25, 90, 78.75, 67.5, 56.25, 45, 33.75,
                22.5, 11.25, 0, -11.25, -22.5, -33.75, -45, -56.25, -67.5, -78.75, -90, -101.25, -112.5, -123.75, -135, -146.25, -157.5, -168.75]
# Define the Z-axis positions in millimeters
# 15 steps max displacement set maanually
z_values = [0, 1000,2000,3000,4000,5000,6000,7000,8000, 9000, 10000, 11000 ,12000, 13000, 14000,15000]
```

Figure 4: Angle and Displacement Array



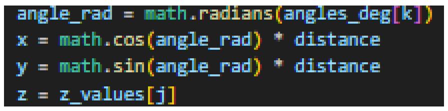
```
# Adjust indices for next iteration
if (i %(len(angles_deg))) == 0 : # after 32 points len(angel_deg) = 32
    j += 1 # z value to next layer
    k= -1 # Reset angle index after layer complete
i+=1  # Increment counter for next point
k +=1 # move to the next angle within a layer
```

Figure 5: Indices Adjusted after 32 (X, Y) Readings

Visualization

To generate a 3D visualization of the data collected, Python was used as the primary programming language in VScode. The data flow and visualization process leveraged several key libraries:

1. **NumPy**: Used for numerical operations and for converting the raw point cloud data into a structured NumPy array, allowing for efficient manipulation and processing.
2. **PySerial**: Facilitated communication with the microcontroller via a serial UART connection, enabling the exchange of distance measurements captured by the VL53L1X sensor.
3. **Open3D**: Utilized to read, process, and visualize the point cloud data, providing an interactive 3D view of the scanned environment.
4. **Math**: Used for basic mathematical calculations and operations



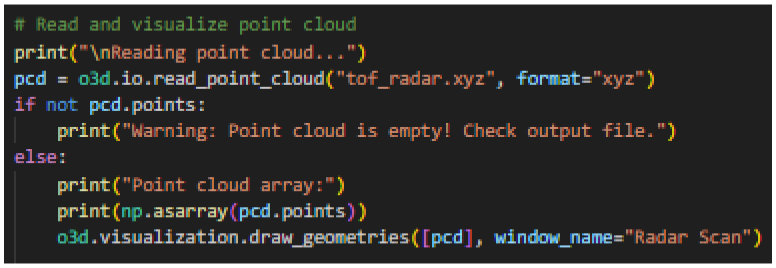
```
angle_rad = math.radians(angles_deg[k])
x = math.cos(angle_rad) * distance
y = math.sin(angle_rad) * distance
z = z_values[j]
```

Figure 6: Math Operations to Covert to Cartesian Points with Math Library

Visualization Process

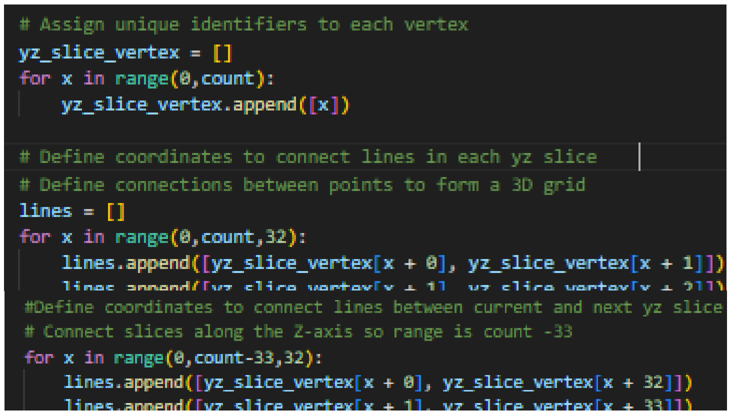
1. **Data Acquisition**: The serial communication between the microcontroller and the Python code is handled using the **PySerial** library. The microcontroller sends distance data, which is decoded and processed into Cartesian coordinates (x, y, z) based on the sensor's measurements.
2. **Data Processing**: The distance values are converted from polar coordinates (angle, distance) to Cartesian coordinates. The z-coordinate is incremented after each set of 32 measurements, corresponding to a new layer of the 3D grid. NumPy is used to store the processed 3D coordinates in an array format making it easier to store, manipulate, and visualize the data in the terminal.

3. **Point Cloud Visualization**: Using the Open3D library, the point cloud data is read from the file “tof_radar.xyz” that is written to and then visualized in 3D. The points are connected with lines to form a 3D mesh, providing a clear representation of the scanned environment. Each point is plotted in the 3D space, and lines are drawn to connect neighboring points, creating a mesh-like structure that reveals the shape and structure of the scanned area.



```python
# Read and visualize point cloud
print("\nReading point cloud...")
pcd = o3d.io.read_point_cloud("tof_radar.xyz", format="xyz")
if not pcd.points:
    print("Warning: Point cloud is empty! Check output file.")
else:
    print("Point cloud array:")
    print(np.asarray(pcd.points))
    o3d.visualization.draw_geometries([pcd], window_name="Radar Scan")
```

Figure 7: Open3D Data Reading from XYZ File

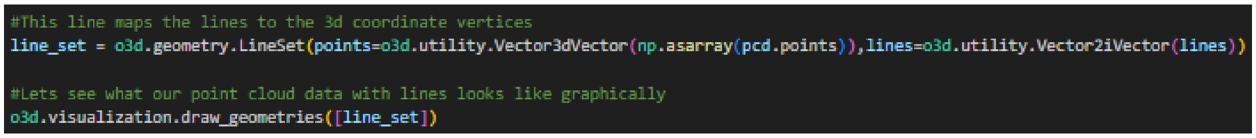


```python
# Assign unique identifiers to each vertex
yz_slice_vertex = []
for x in range(0,count):
    yz_slice_vertex.append([x])

# Define coordinates to connect lines in each yz slice
# Define connections between points to form a 3D grid
lines = []
for x in range(0,count,32):
    lines.append([yz_slice_vertex[x + 0], yz_slice_vertex[x + 1]])
    lines.append([yz_slice_vertex[x + 1], yz_slice_vertex[x + 2]])
#Define coordinates to connect lines between current and next yz slice
# Connect slices along the Z-axis so range is count -33
for x in range(0,count-33,32):
    lines.append([yz_slice_vertex[x + 0], yz_slice_vertex[x + 32]])
    lines.append([yz_slice_vertex[x + 1], yz_slice_vertex[x + 33]])
```

Figure 8: Line Connections between Points

4. **Interactivity**: The Open3D visualization window allows for interactive exploration of the 3D point cloud, enabling users to rotate, zoom, and pan to view the model from different angles.



```python
#This line maps the lines to the 3d coordinate vertices
line_set = o3d.geometry.LineSet(points=o3d.utility.Vector3dVector(np.asarray(pcd.points)),lines=o3d.utility.Vector2iVector(lines))

#Lets see what our point cloud data with lines looks like graphically
o3d.visualization.draw_geometries([line_set])
```

Figure 9: Visualization Window Code

Application Note, Instructions, and Expected Output

This application note describes the operation and expected output of the Spatial Mapping System, which uses a Texas Instruments MSP432E401Y Microcontroller, a VL53L1X Time-of-Flight (TOF) sensor, and a 28BYJ-48 stepper motor to create a 3D model of indoor spaces. The system serves as a more affordable alternative to commercial solutions, providing distance measurements in real-time and creating a visual 3D representation of an environment using Python-based visualization tools.

Instructions

1. Startup Installations Windows Only

Software Requirements for the project, install onto PC:

- Python Version 3.10 from Python Website
- PySerial
    - Type "pip install pyserial" in Command Prompt and press Enter to install PySerial. Read and resolve issues in terminal accordingly.
- Open3D
    - Type "pip install open3d" in Command Prompt and press Enter to install Open3D. Read and resolve issues in terminal accordingly.
- Keil uVision5 and C from corresponding websites

Python Installation and Verification:

1. Open Command Prompt on Windows.
2. Type "Python" and press Enter.
3. If Python is installed correctly, it will display the version number, otherwise, you will need to resolve installation issue Python before proceeding. More information can be found on their website.

Verifying Keil uVision Installation:

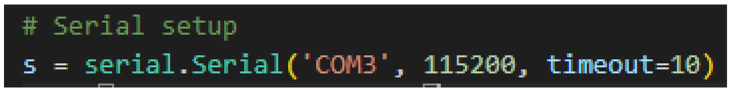
1. Connect the microcontroller to your PC.
2. Ensure settings are correct in Keil target options: O0 optimization, CMSIS-DAP Debugger and XDS110. Install XDS110 debugging tool if not done so, instructions on Keil website.
3. Load given C code in Keil onto microcontroller.
4. If the microcontroller responds, Keil is installed correctly.

2. Microprocessor Wiring to All Components

Wire all stepper motor, time of flight sensor and the PC to the microprocessor according to the device characteristics table or the circuit schematic in Figures 15-16. The pushbuttons are onboard and configured through code, no wiring is needed.

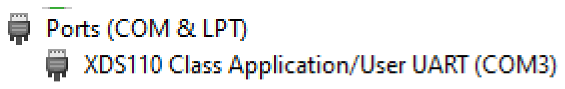
3. Parameter Correction for UART and Measurements

Change "COM3" to the corresponding UART port on your PC from Device Manager



```python
# Serial setup
s = serial.Serial('COM3', 115200, timeout=10)
```

Figure 10: Python Code COM Port Setup

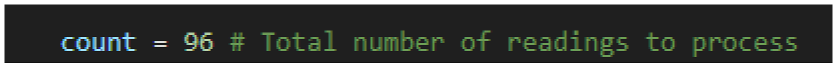


```
Ports (COM & LPT)
    XDS110 Class Application/User UART (COM3)
```

Figure 11: Device Manager Port Identification

X and Y are defined as the vertical distance slices that are calculated in the Python code, and Z is defined as the displacement.

Change the number of vertical distance scans. A count set to 96 is 3 vertical scans with 32 points per scan. To change the number of scans, increase or decrease count by multiples of 32.



```
count = 96 # Total number of readings to process
```

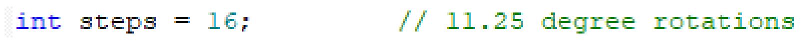
Figure 12: Number of X, Y Readings Set

Displacement is set to intervals of 1000mm, change this value to have shorter or longer displacement depending on room size. Currently it is manually set to a max of 16 steps, but you can increase or decrease by adding the same value for the indexes needed.

Note: The Python code only supports 11.25-degree steps for the stepper motor for 32 points for each scan.

You must implement additional code in Python program to support 45-degree steps.

To change step size, it must be done in Keil C code. For 45-degree steps the "steps" variable can be set to 64.



```
int steps = 16;          // 11.25 degree rotations
```

Figure 13: Steps Constant for 11.25 Degree Steps

4. Run Programs in Keil and Python

1. Load the C program onto the microcontroller.
2. Run/ Execute the Python script
3. Reset the microcontroller and press 'Enter' in the Python terminal.
4. Press PJ0 on Microcontroller to start/stop stepper motor and scanning
5. Optionally press PJ1 to stop/start data collection exclusively

5. Visualize the Scan

After the specified number of scans are completed, the python code will open a new window displaying the points of the scan automatically. The point data is also stored in a file called "tof_radar.xyz" on your PC. Once you close the point visualization window, the connection lines will appear.

Expected Output Scan

The expected output after the point visualization window is closed is shown below with the connected lines. An example application for a scan is below, which is of a hallway located in ETB at McMaster University.

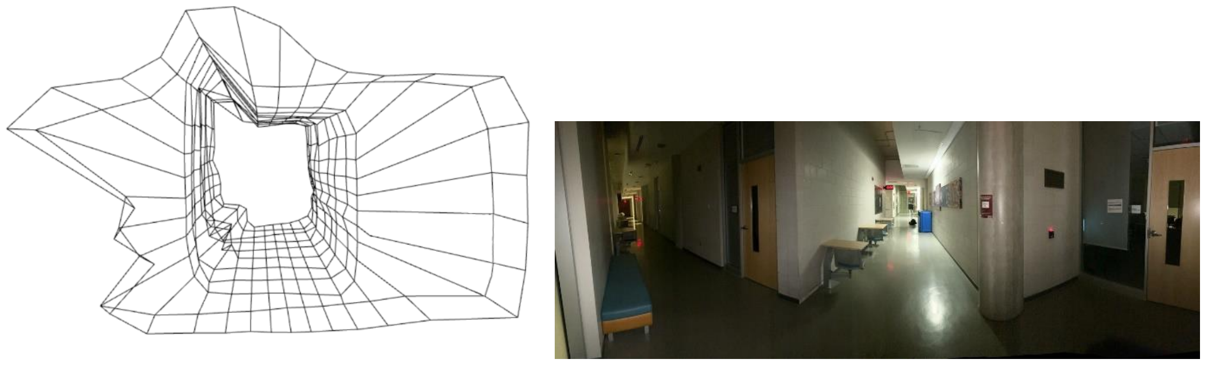


Figure 14: Expected Output Side-by-Side Comparison - ETB Hallway at McMaster University

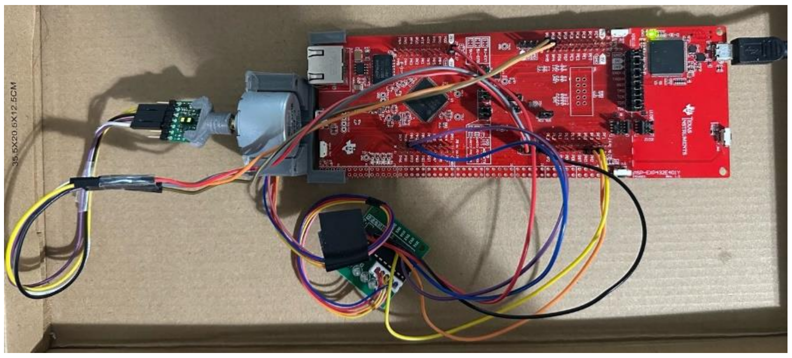


Figure 15: Real-World Component Connection

Limitations

1. **Summarize any limitations of the microcontroller floating point capability and use of trigonometric functions.**

   The MSP432E401Y microcontroller has limitations when it comes to floating-point operations. It includes a single-precision Floating Point Unit (FPU) that supports 32-bit floating-point arithmetic efficiently. This is generally sufficient for many applications, but it may not be ideal in cases that require double-precision (64-bit) floating-point operations, as those would need to be handled in software, leading to slower performance. Additionally, trigonometric functions (such as sine, cosine, and tangent) are computed using software-based math libraries rather than hardware acceleration. This makes trigonometric calculations relatively slow. When combined, the limitations of single-precision floating-point and the use of software-based trigonometric functions can result in reduced accuracy and slower performance, particularly in applications that require precise floating-point calculations or operations.

2. **Calculate your maximum quantization error for the TOF module.**

   The maximum quantization error can be defined as the biggest discrepancy that can arise when a continuous signal is changed into a discrete form during the quantization process. Since there is no ADC in the VL53L1X sensor in its hardware we cannot use the ADC formulas to quantify it. The maximum quantization error is determined by its resolution of 1mm, which is the unit of measurement it uses. The measurements round to the nearest millimeter and you get this error.

   What is the maximum standard serial communication rate you can implement with the PC. How did you verify?

   The communication between the microcontroller and PC is through UART. The maximum standard serial communication rate you can implement is 115,200bps. This was verified through the Device Manger application on the PC where the COM port allowed you to select a maximum baud rate of 128,000bps but communication failed at this speed when I tried to use it in the Python code. The next available option was 115,200bps which worked. It was also verified during the process of testing the transmission and setting the baud rate to different values to see which ones could be implemented.

3. **What were the communication method(s) and speed used between the microcontroller and the TOF modules?**
   The communication method between the microcontroller and the TOF sensor is the I2C protocol. I2C has multiple modes that are accessible such as the standard mode (100k bits/s) and fast mode (400k bits/s). The speed used for the transmission in the project was set to 100k bits/s, which was implemented by changing the MTPR value in the C code in Keil to correspond to the 14MHz bus speed.

4. **Reviewing the entire system, which elements are the primary limitation on speed? How did you test this?**
   The main factors limiting system speed are the time taken by the TOF sensor to acquire data and the time the stepper motor requires to rotate to the next position. The TOF sensor, in long-distance mode, typically takes 140ms per measurement. Similarly, the stepper motor's rotation introduces additional delays. These factors far exceed the time required for serial communication, making them the main constraints on system speed. This was tested by setting every component value speed as high as possible and seeing how long it took for the motor and TOF to give out readings.

Circuit Schematic

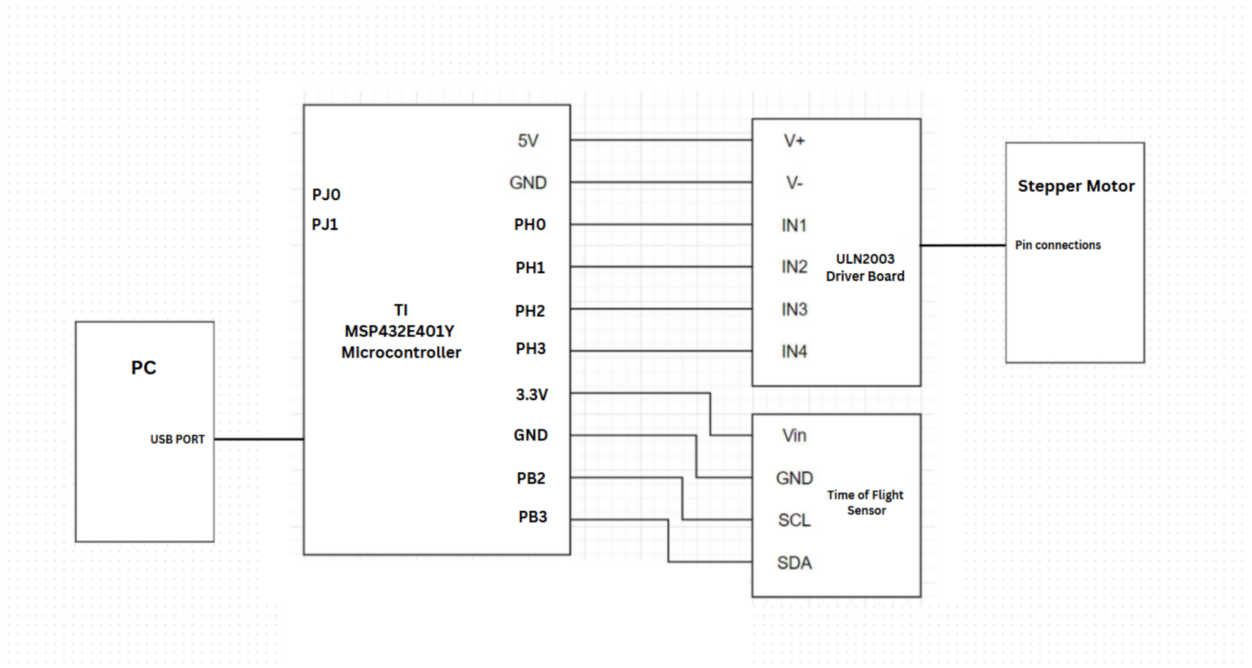
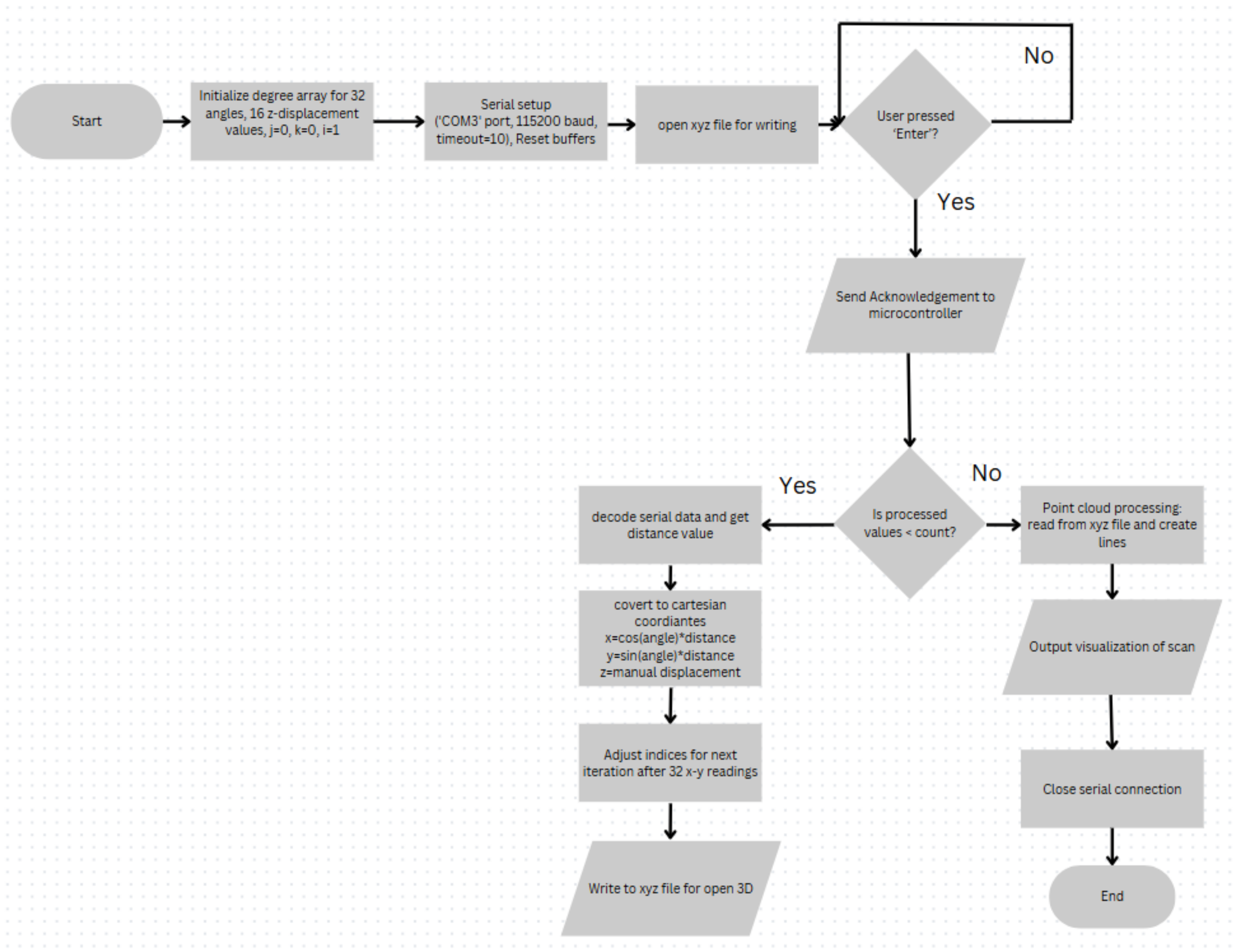


Figure 16: Circuit Schematic of System Design

PJ0 and PJ1 are onboard pushbuttons that are configured in the Keil code.

Programming Logic Flowcharts



Start
Initialize degree array for 32 angles, 16 z-displacement values, j=0, k=0, i=1
Serial setup ('COM3' port, 115200 baud, timeout=10), Reset buffers
open xyz file for writing
User pressed 'Enter'?
No
Yes
Send Acknowledgement to microcontroller
Is processed values < count?
Yes
No
decode serial data and get distance value
covert to cartesian coordiantes x=cos(angle)*distance y=sin(angle)*distance z=manual displacement
Adjust indices for next iteration after 32 x-y readings
Write to xyz file for open 3D
Point cloud processing: read from xyz file and create lines
Output visualization of scan
Close serial connection
End

Figure 17: Python Code Flowchart

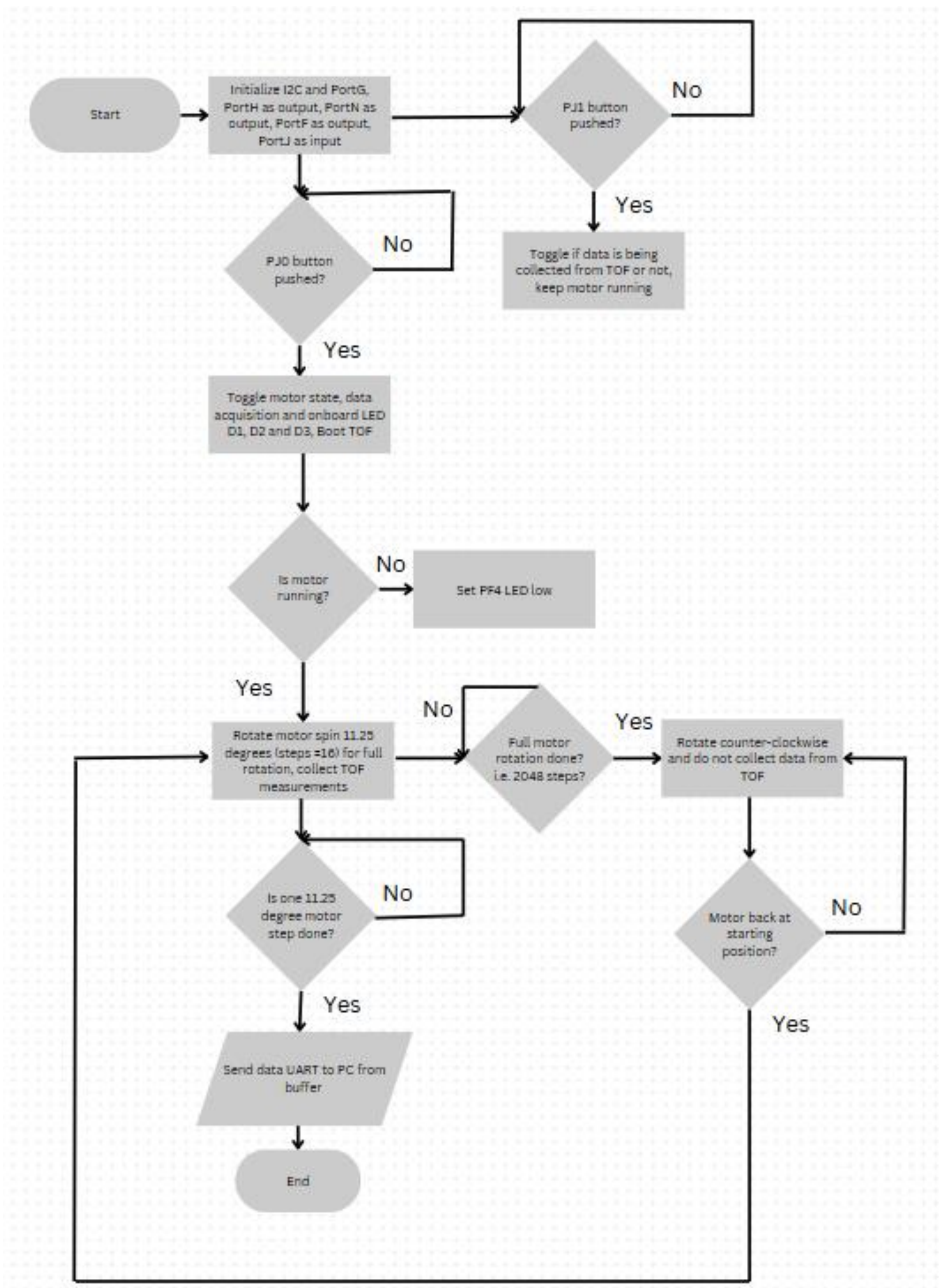

Start
Initialize I2C and PortG, PortH as output, PortN as output, PortF as output, PortJ as input
PJ1 button pushed?
No
Yes
Toggle if data is being collected from TOF or not, keep motor running
PJ0 button pushed?
No
Yes
Toggle motor state, data acquisition and onboard LED D1, D2 and D3, Boot TOF
Is motor running?
No
Set PF4 LED low
Yes
Rotate motor spin 11.25 degrees (steps =16) for full rotation, collect TOF measurements
No
Full motor rotation done? i.e. 2048 steps?
Yes
Rotate counter-clockwise and do not collect data from TOF
Is one 11.25 degree motor step done?
No
Yes
Motor back at starting position?
No
Yes
Send data UART to PC from buffer
End

Figure 18: Keil C code Flowchart