COMPENG 2DX3
Final Project Report

Yazan Khatib
Instructor. Dr. Yasser M. Haddara

Department of Electrical and Computer Engineering, McMaster University
April 5, 2025

Device Overview

The 2DX4 Final Project Lidar system is a compact, integrated 3D scanning device capable of measuring distances across several 360-degree planes positioned orthogonally. It processes these measurements to generate a three-dimensional map of the scanned space.

The system is composed of a microcontroller, a stepper motor, and a time-of-flight (ToF) sensor. The microcontroller manages all system operations apart from the 3D visualization, supplying power to the components and sending all collected data to an external device via serial communication. The stepper motor enables full 360-degree rotational motion, allowing the ToF sensor to collect distance readings throughout an entire vertical plane.

At the core of the system, the VL53L1X ToF sensor emits infrared laser pulses to measure the distance to surrounding objects. It determines distance by calculating the time it takes for the reflected pulses to return to the sensor. This data is processed based on the sensor's configuration and converted to a digital signal via an ADC and sent to the microcontroller through an I²C interface.

All processed distance data is transmitted to the PC using UART (Universal Asynchronous Receiver/Transmitter) serial communication over COM4 at 115200 baud. The MSP432 microcontroller formats and sends each data packet—consisting of distance measurements and positional metadata—over UART. On the PC side, a MATLAB or Python script reads the data stream, ensuring synchronization and storing the values for visualization.

Once received by the PC, the data is processed and plotted using MATLAB or Python scripts. These scripts interpret the 1D distance measurements in combination with the stepper motor's angular position (based on its step count), converting polar data into Cartesian coordinates (x, y, z). The result is a full 3D point cloud, which can be viewed and analyzed to interpret the scanned environment. The script can also generate an .xyz file for further use with external 3D visualization tools.

Features

- **Integrated 3D Scanner**
  - **Motor/TOF System capable of taking distance measurements**
  - **Full 360 coverage**
  - **Orthogonal displacement samples**
  - **VL53L1X ToF sensor**
    - **Up to 4m range**
    - **Accurate within +- 20 mm ranging error**
    - **2.6 - 3.5V Operating Voltage**
    - **8 - 15$ CAD**
  - **ULN2003 Stepper Motor**
    - **512 steps at 2ms delay speed**
    - **5 - 12v Operating Voltage**

        - **5 - 10$ CAD**

- **MSP432E401Y Microcontroller**
    - **24 Mhz bus Speed (Capable of up to 120Mhz)**
    - **ARM Cortex-M4F Processor**
    - **Integrated LEDs PN0, PF4, PN1 for Data Transmission indicator, 11.25 deg Displacement Indicator, Return Indicator.**
    - **Serial Port COM4**
    - **115200 BPS Baud Rate (8-N-1)**
    - **2 Onboard buttons PJ1/PJ0 for controllering the system**
    - **~75$ CAD**
    - **Flash Memory: 1 MB**
    - **SRAM: 256 KB**

- **Data Communication**
    - **I2C Communication between MSP Micro and TOF sensor**
    - **UART Serial communication between PC and MSP controller**

- **MATLAB/Python Visualization script**
    - **Creates 3D Visualization of mapped data (xyz file made from python script)**

Block Diagrams

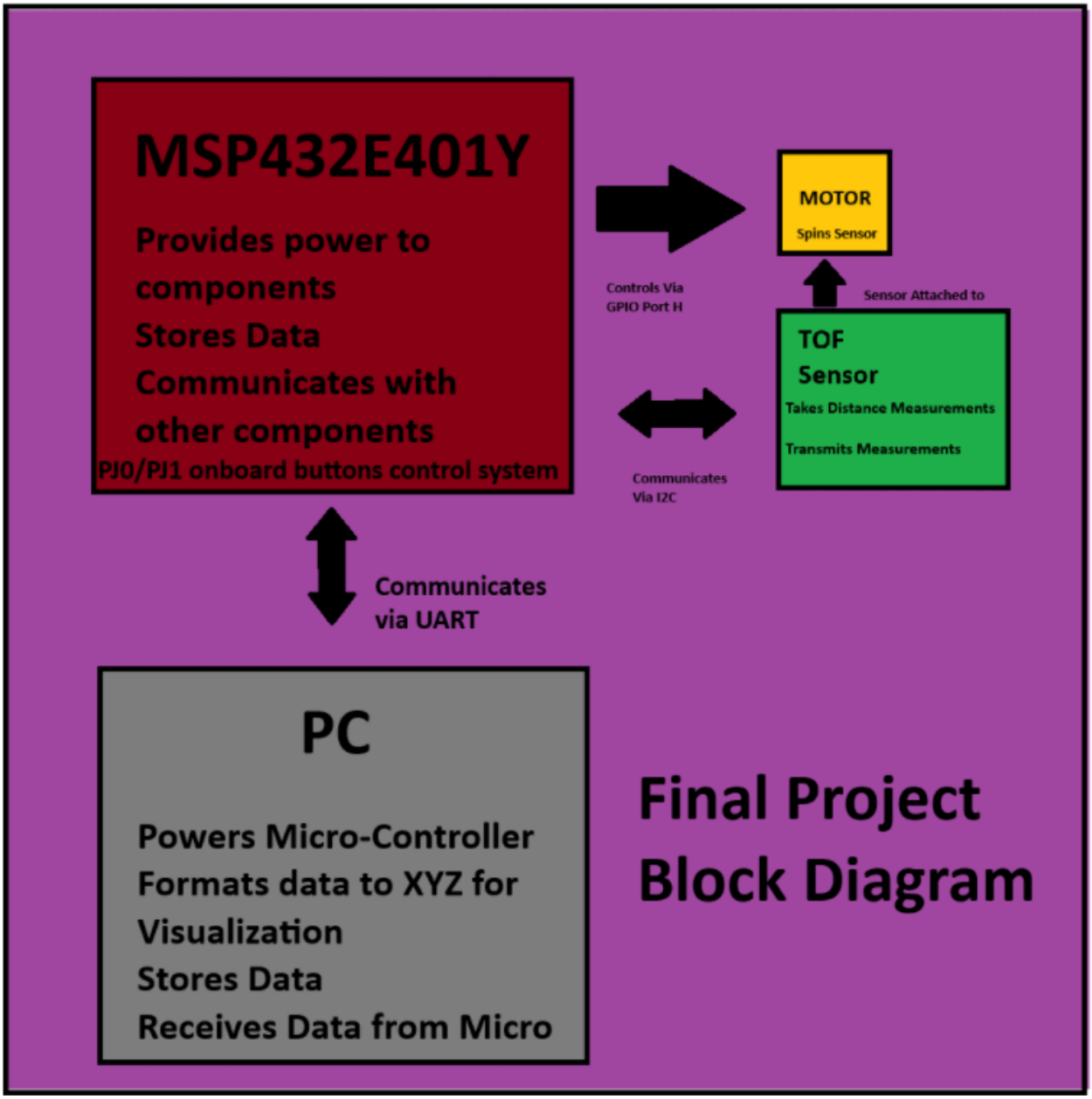


Figure 1: System Block Diagram

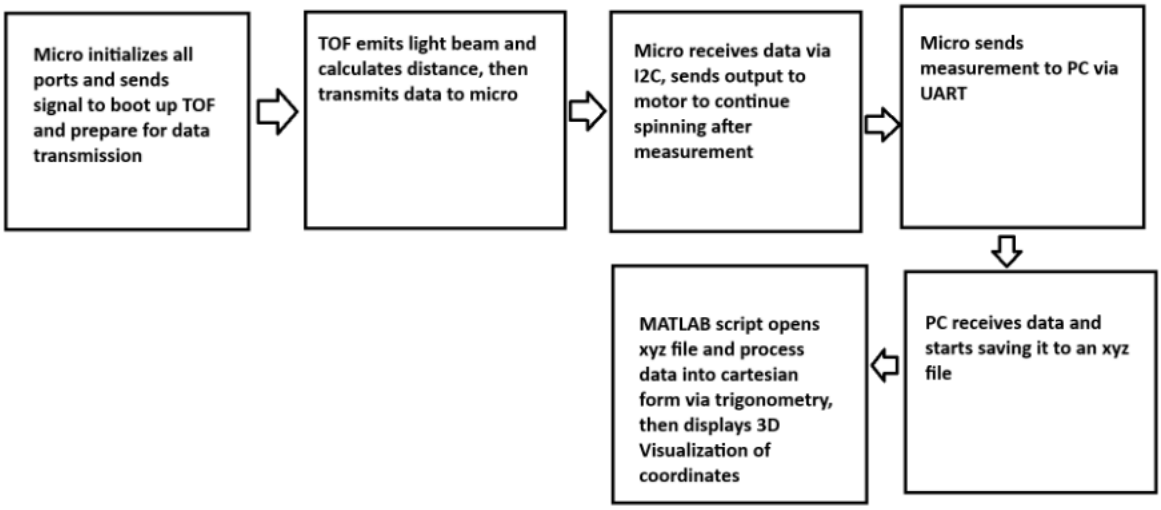


Figure 2: Data flow Block Diagram

Device Characteristics Table

MSP432E401Y		ULN2003		VL53L1X	
Bus Speed	24Mhz	+	5V	Vin	3.3V
Baud Rate	115200	-	GND	GND	GND
Measurement Status	PN1	IN1	PH0	SDA	PB3
UART Tx Status	PF4	IN2	PH1	SCL	PB2
Returning Home Status	PN0	IN3	PH2	Measurements per scan	32
Serial Port	COM4	IN4	PH3	Max range	4m
Start	PJ1	Angle between measureme	11.25 deg		

		nt			
Stop	**PJ0**				

Figure 3: Device Characteristics Table

Detailed Description

Part A: Distance Measurement

The VL53L1X Time-of-Flight (ToF) sensor is responsible for measuring the distance. The sensor operates by utilizing infrared light, emitting it through its emitter to reflect photons that are then reflected back into the sensor's receiver. By measuring the time that it takes between emitting and receiving the photons, the sensor can figure out distance as the speed of photons is known, and thus the formula for distance D is ((Speed of light (speed of photons)) * (Time between emitting and receiving)) / 2. We Divide by 2 because the photons travel to and back from whatever object is reflecting, meaning they cover that distance twice. After the ToF sensor calculates the distance, it then converts the measurement into a digital signal that is then transmitted to the microcontroller via I2C.

Before the sensor can be used, various hardware and software components must be properly initialized. The microcontroller's system clock is configured through PLL_Init() and a delay timer is established with Systick_Init() to allow for timed operations. Several ports are then set up for specific roles. Port B is configured for I2C communication, which is the primary interface between the microcontroller and the VL53L1X sensor, Port G is used to control the sensor's XSHUT pin, which allows hardware resets, Ports H is used to control the stepper motor, and Port J is configured to enable the onboard buttons PJ1/PJ0 for user input. The I2C_Init() function sets up I2C0 on Port B (PB2: SCL, PB3: SDA) by configuring the alternate functions, open-drain settings for SDA, digital I/O, and the master clock (MTPR) for a 100 kbps I2C clock rate. This enables communication with the VL53L1X, which acts as an I2C slave device.

The ToF sensor is initially reset using the VL53L1X_XSHUT() function. This function pulls the XSHUT pin low via PG0 to disable the sensor and then returns it to high-impedance mode, allowing it to boot. After the reset, the sensor's boot state is checked in a loop using VL53L1X_BootState(). The microcontroller repeatedly polls the sensor until it returns a non-zero state, indicating it has completed its boot sequence. Following this, VL53L1X_GetSensorId() is used to retrieve the sensor ID and ensure correct communication. This step verifies that the sensor is responding with the correct device ID (expected to be 0x29). If successful, the code proceeds to initialize the sensor with VL53L1X_SensorInit(), which loads the default firmware settings onto the VL53L1X and configures internal registers required for standard operation.

Once the sensor is initialized, distance measurement is conducted using several key functions. VL53L1X_StartRanging(dev) initiates the continuous ranging process. The sensor begins emitting laser pulses and timing the return of reflections from the target. In this system, the ToF sensor is not static. Instead, it is physically mounted on a rotating motor shaft. The motor is controlled to incrementally rotate the sensor by 11.25 degrees per step. This angular movement is deliberate and systematic: 360° divided by 32 intervals results in 11.25° per step. Therefore, one full rotation of the motor allows the sensor to collect 32 distance readings, each corresponding to a unique angle around the full circle. This setup essentially transforms the ToF sensor into a rudimentary LiDAR system, capable of scanning its surroundings in all directions.

The motor's rotation and the sensor's data capture are tightly synchronized in code. After each motor movement off 11.25 degrees (number of steps % 16 == 0), the system triggers the ToF sensor to take a new measurement. This ensures that each distance reading is spatially aligned with a specific direction. The microcontroller waits for the measurement to complete using the VL53L1X_CheckForDataReady() function before retrieving the result. Once the measurement is ready, it is read using VL53L1X_GetDistance(), and then sent to the PC via UART for analysis.

Part B: Visualization

During each full 360° rotation, the system collects 32 valid distance measurements, which are transmitted to a PC via stable UART serial communication, using the PySerial library in Python. Each data packet is received as a byte stream and unpacked into a string format. From this string, two key components are extracted: the Distance and the RangeStatus values. These are parsed and converted back into a floating-point number (Distance) and an integer (RangeStatus), respectively.

The Distance measurement is then transformed into Cartesian coordinates on the zy-plane using basic trigonometric formulas (as shown in Figure 4). Meanwhile, the x-axis value is incremented every 32 measurements by a predefined amount set at the top of the script, simulating motion in the x-direction with each full rotation of the motor. The resulting three-dimensional coordinate values (x, y, z) are written to a .xyz file, which is later used for 3D point cloud visualization with the Open3D library.

```
data = data.split(",")
distance = float(data[0])
rangeStatus = int(data[1])


y = distance * sin(-11.25*(i*(3.14/180.0)))
z = distance * cos(-11.25*(i*(3.14/180.0)))


f.write('{0:.2f} {1:.2f} {2:.2f}\n'.format(x, y, z))
```

Figure 4: Converting Distance to Cartesian Coordinates.

Simultaneously, the RangeStatus is used to determine the quality of each data point. Based on its value, an associated color is stored in a colours array:

A RangeStatus of 0 (indicating a valid scan) assigns green,
Values 1 or 2 (minor errors) assign yellow,
Values 4 or 7 (major errors) assign red.

To complete the data collection phase and trigger visualization, the user presses the PJ0 button while the microcontroller is in the idle state. This action sends the message "Measurements done\n" over UART to the PC. Once received, the Python script exits the data collection loop, closes the serial port and the .xyz file, and proceeds to the visualization phase.

In the visualization process, Open3D is used to read the .xyz file and generate a 3D point cloud. The script also iterates through the colours array to apply the appropriate color to each point, representing its measurement reliability.

To enhance spatial perception, the script creates connecting lines between neighboring points. For each point, a line is drawn to the next point in the same rotation (e.g., point i to i+1) and to the corresponding point in the next rotation cycle (e.g., point i in rotation n to point i in rotation n+1). These connections, rendered via Open3D, help form a mesh-like structure that illustrates both horizontal and vertical continuity in the scanned data.

Finally, Open3D opens an interactive visualization window displaying the full 3D point cloud with color-coded quality indicators and connecting lines, providing a clear and accurate representation of the scanned environment.

Application Note, Instructions, and Expected Output.

A system like this, which uses a Time-of-Flight (ToF) sensor mounted on a rotating motor, can be applied in a robotic vacuum cleaner to help it scan and map its environment. As the motor spins the sensor 360 degrees in small steps (11.25° per step), the ToF sensor collects 32 distance measurements per full rotation. These distance values allow the robot to detect how far objects are in every direction around it. As the robot moves, it continues to take these circular scans, building up a full picture of the room.

The collected distance data is converted from polar coordinates into 3D cartesian coordinates using trigonometry. These points are used to form a point cloud, which shows the shape of the room and the location of objects like furniture and walls. Each measurement also comes with a quality indicator (called RangeStatus), which helps

the robot know how reliable the reading is. This makes the robot smarter, allowing it to ignore bad data and focus on accurate measurements.

This system helps the vacuum cleaner navigate without bumping into things, clean efficiently, and avoid areas that are already done or blocked. Over time, it can build a detailed map of the room and plan better cleaning paths. It can also send this data to a computer for visualization, making it easier to improve the robot's design or cleaning strategy. Overall, this kind of system makes robots more aware of their surroundings and better at completing tasks on their own.

Instructions

This setup process assumes that the user has installed all necessary software for their system, including the Kiel Development Environment, a suitable IDE for running the Python script, and a compatible version of Python with Open3D.

1) Connect the microcontroller to your computer and determine the COM port by pressing the Windows key, typing 'Device Manager', and hitting enter. In the Device Manager window, expand the 'Ports (COM & LPT)' section, then find and note the COM port number listed next to 'XDS110 Class Application/User UART (COM #)'.

2) Open the Python script and modify line 30, replacing "s = serial.Serial('COMX')" with the correct COM port number from step 1.

3) Set up the circuit according to the Device Characteristic table above

4) Open the Kiel project, go to 'Options for Target', select the 'C/C++' tab, and ensure the optimization is set to '-O0'. Then, under the 'Debug' tab, choose 'CMSIS-DAP Debugger' from the drop-down menu and click the 'Settings' button. Ensure that 'XDS110 with CMSIS-DAP' is selected in the top-left dropdown. Click 'OK' to confirm the settings.

5) In the IDE, click 'Translate' → 'Build' → 'Load' to compile and load the project onto the board.

6) Flash the project to the board by pressing the RESET button.

7) Modify line 25 of the Python script, changing “X_INCREMENT = 300” to the desired x-axis distance in millimeters between scans

8) Ensure all required dependencies are installed. If any are missing (serial, math, numpy, open3d), open a terminal and type ‘pip install X’, replacing ‘X’ with the missing dependency.

9) Run the Python script.

10) Press enter to confirm that you are ready to receive data from the microcontroller.

11) Press PJ1 to start the 360-degree rotation scan.

12) Move forward by X_INCREMENT millimeters (for example, 100mm if X_INCREMENT=100). (yz are used for vertical distance slices, x is displacement, must be set by user)

13) Repeat steps 12-13 until the desired number of scans is completed.

14) Press PJ0 while in the idle state to finish data collection and begin visualizing the data.

Expected Output

The expected output of this device can be seen below. My assigned location was in ETB section E, and I had to scan in the same direction with displacement of 300mm in the x direction. The scan was carried throughout the hallway up until the edge of the left wall ended, for which we were not allowed to go beyond. Thus leaving us with a simple hallway model with a curving right wall as can be seen below in the photo and scan.

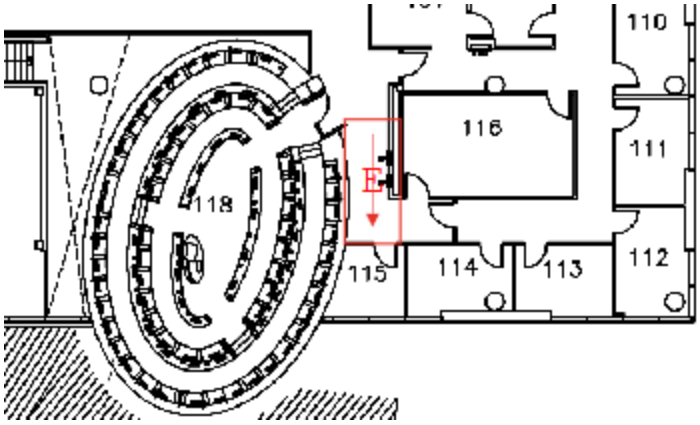


Figure 5: Hallway location

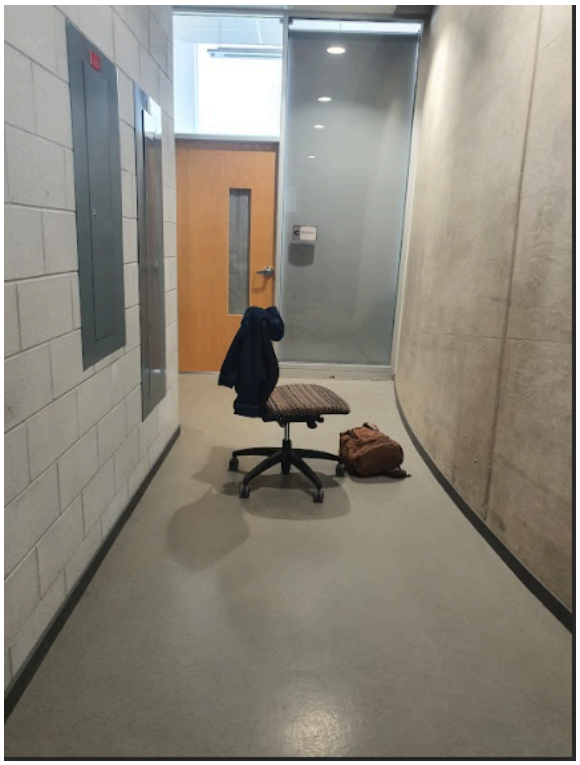


Figure 6: Hallway Front View

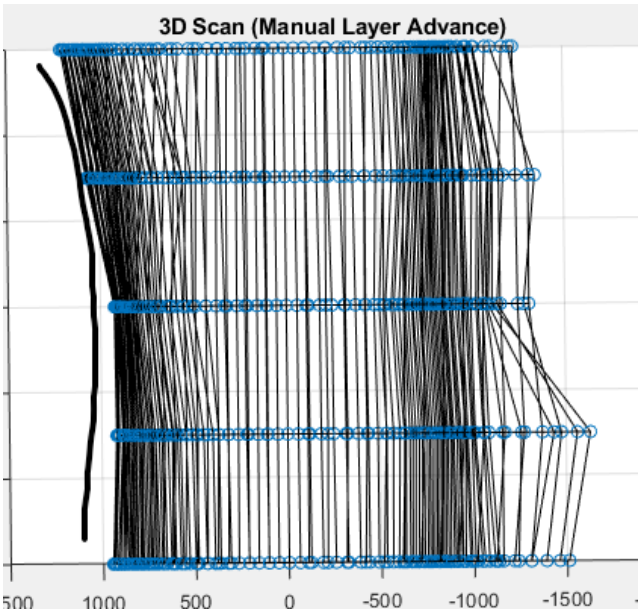


Figure 7: Hallway 3D Visualization bottom view (with curve wall marked for reference)

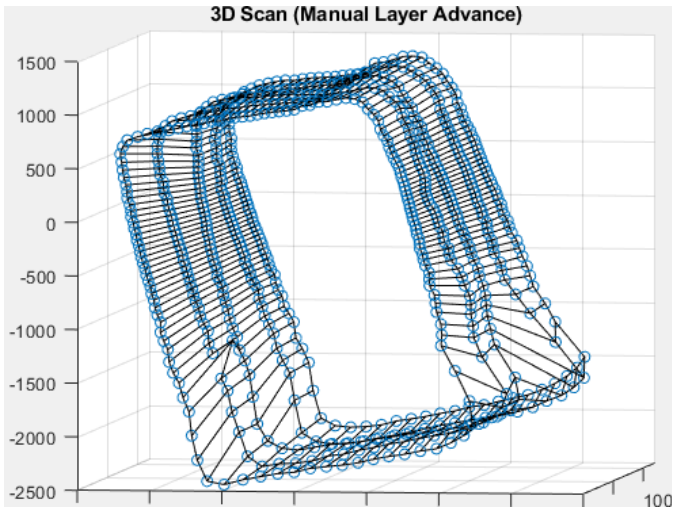


Figure 8: Hallway scan front view (xyz file opened in MATLAB)

Limitations

1. The Arm Cortex-M4 Processor features a 32-bit Floating point Unit (FPU) which is capable of floating point operations such as addition, subtraction, multiplication, division, etc. which can be performed with a precision of up to 32 bits.[1] Due to the use of trigonometric functions when converting coordinates, it is preferred to maximize accuracy as much as possible, and offloading the processing to the PC which has accuracy up to 64 bits gives much more accuracy. The PC also has much faster calculation speeds due to the greater computational power, which further improves the performance of the system.

2. The Max Quantization error can be calculated via (Max Distance/2^(ADC bits)). The maximum range on our TOF is 4m, and the given spec in lecture was that the TOF has a max resolution of 1mm (meaning it has a max quantization error of 1mm). This can be calculated as well (4000mm/2^12 = ~0.97mm = ~1mm)

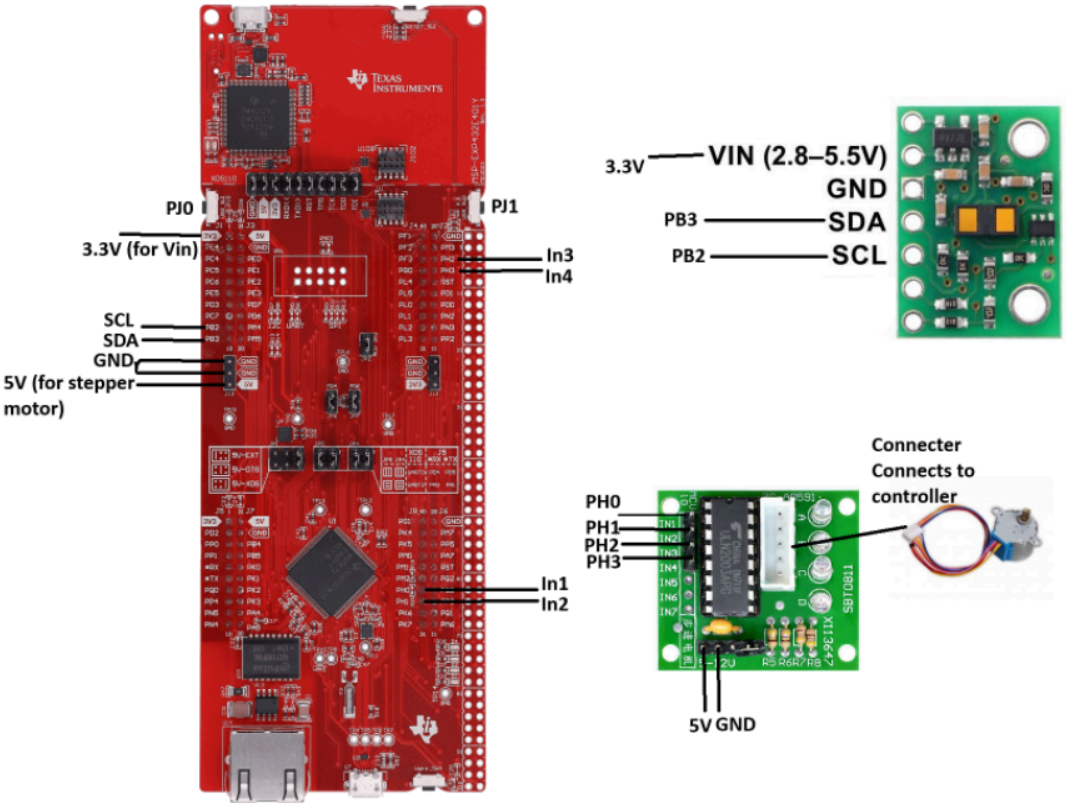
3. When checking device manager, and going to Ports (COM & LPT), and viewing the properties of Communications port, it can be observed that the maximum speed allowed is 128000 bps. In my project I implemented 115200 bps.

4. The microcontroller and the ToF module use I2C to communicate with each other. The microcontroller is the leader and the ToF is the follower, the speed set was 100kbps but the Micro is capable of up to 3.33Mbps (High speed mode I2C) and the ToF supports up to 400kbps (Fast Mode) I2C.[2]

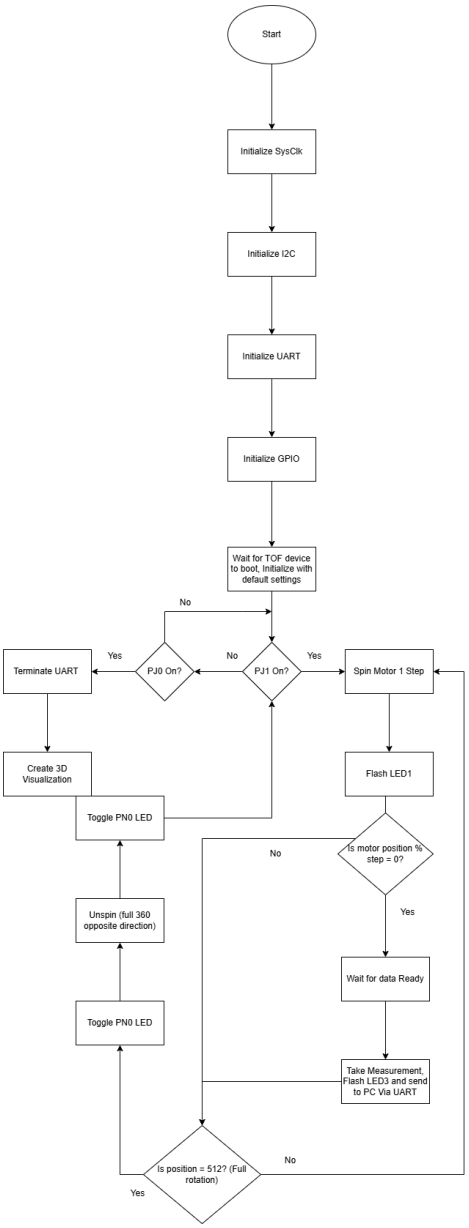
5. The primary limitation on system speed is the Stepper Motor and, as it forces at least a 2ms delay between each step to rotate or else it will skip steps and/or not move at all. This was tested by trying to run the sensor without the motor and seeing how much time can be saved, which for a single rotation the stepper motor adds (4 phases/step * 512 steps * 2ms/phase) = 1.024 seconds. Another bottleneck would be the TOF sensor, as it takes at least a 200ms delay for each rotation, this was also tested similarly to the stepper motor by adding up all the delays in a rotation.

6. My assigned bus speed was 24Mhz, which was achieved by changing the N value to 24, which gives us this calculation:
   (PSYSDIV = 3, Q = 0, N = 24, MINT = 96, MFRAC = 0)
   SysClk = (fVCO/(Psysdiv + 1))
   = ((25000000/((0 + 1)(24 + 1))) * (96 + (0/1024)))/(3 + 1))
   = 24000000 hz
   = 24Mhz

Circuit Schematic



Programming Logic Flowchart.



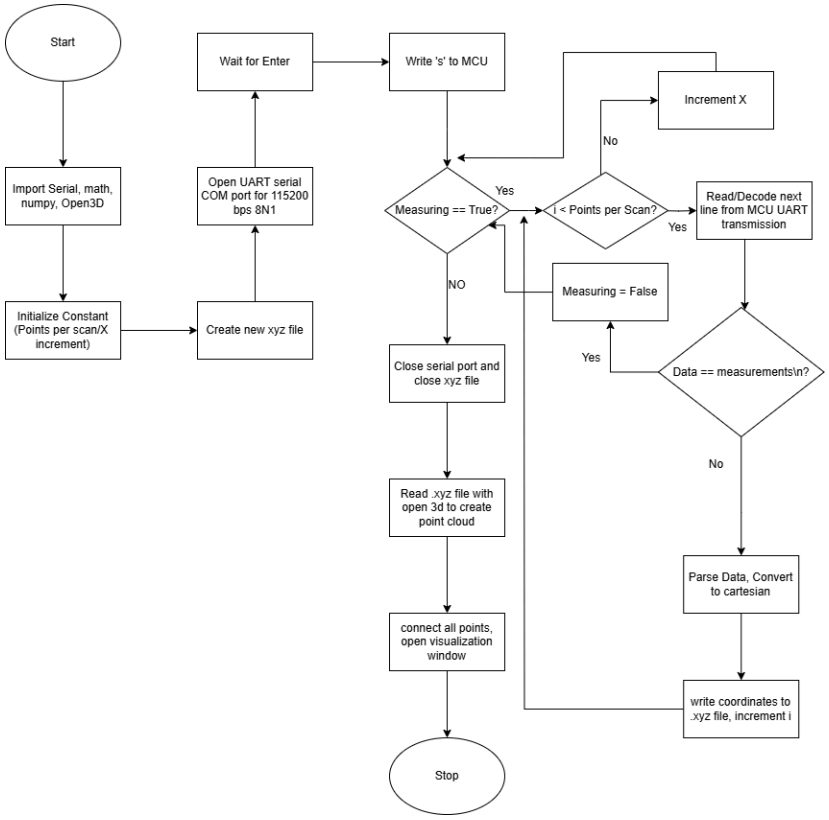


Figure 10: Visualization Program Flowchart

Figure 9: Keil Program Flowchart

References Cited

[1] "MSP432E401Y SimpleLinkTM Ethernet Microcontroller Data Sheet," Ti, https://www.ti.com/lit/ds/symlink/msp432e401y.pdf (accessed Apr. 6, 2025).

[2] “VL53L1X,” STMicroelectronics, https://www.st.com/en/imaging-and-photonics-solutions/vl53l1x.html (accessed Apr. 6, 2025).