Ayaan Khan 400459403 April 10th, 2024

COMPENG 2DX3

ToF Spatial Mapping Device

Technical Report and Datasheet

DEVICE OVERVIEW

Features

- RAPID 3D SPATIAL MAPPING: Creates a 3D visualization of the surroundings by scanning the location with a Time-of-Flight (ToF) sensor.
- INTUITIVE OPERATION: Start and reset the scanner easily with dedicated buttons.
- FULL 360° SCANNING: Achieves precise scanning using a 28BYJ-48 Stepper Motor with 2048-step precision, controlled in Full Step mode for accurate positioning and smooth operation, supported by a stable ULN2003 Driver ensuring reliable performance.
- ADVANCED DATA CAPTURE: Utilizes VL53L1X ToF sensor for distances up to 4 meters, with efficient I²C communication for seamless data transfer. Supports 2.6V to 5.5V voltage input and offers up to 50Hz ranging frequency for rapid data capture.
- PC CONNECTIVITY: Establishes serial communication (115200bps baud rate) with Python interface for data transfer.
- MICROCONTROLLER EFFICIENCY: MSP432E401Y microcontroller operates at 60MHz for optimized performance.

General Description

The spatial mapping device is a microcontroller-based tool designed for 360° spatial scanning as it traverses an area. It is intended to be paired with a PC, allowing users to capture multiple scans and utilize integrated data mapping software to create a 3D visualization of the scanned environment.

Operation is straightforward: in the software, specify the port, set the scan interval, and designate the number of scans. Then, with a single push button, initiate a scan. The 28BYJ-48 Stepper Motor performs scans in both clockwise and counter clockwise directions, optimizing speed.

Out of the box, the device is configured to take 32 measurements per 360° scan using the VL53L1X ToF sensor, which precisely captures distances using high-speed infrared. The ToF sensor enables distance measurements up to 4 meters at a ranging frequency of up to 50Hz.

This device connects to any PC with the necessary Python libraries—PySerial and Open3D—installed and requires an available USB port.

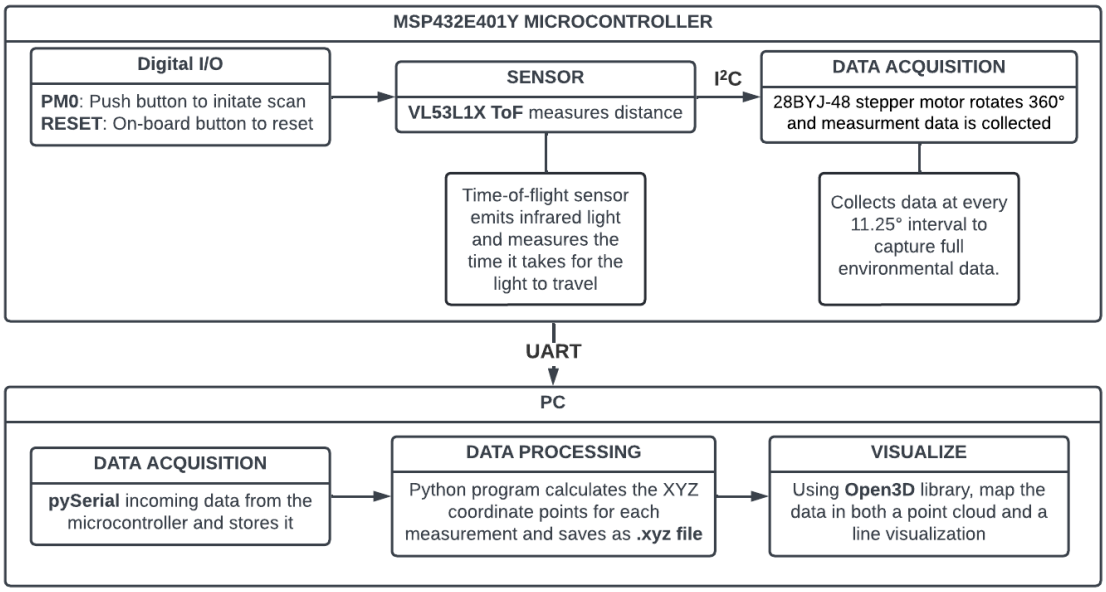

FIG 1 BLOCK DIAGRAM

DEVICE CHARACTERISTIC TABLE

Characteristic	Description					
Microcontroller	MSP432E401Y SimpleLink™ Microcontroller					
	Baud Rate	Bus Speed	Memory			
	115200 bps	60 MHz	Flash Memory: 1024 KB SRAM: 256 KB EEPROM: 6 KB			
Stepper Motor	28BYJ-48 Stepper Motor					
Driver for Stepper Motor	ULN2003 Driver					
- Driver Connections	+ TERM	- TERM	IN0	IN1	IN2	IN3
- MCU Connections	+5 Supply	GND	PH0	PH1	PH2	PH3
Sensor	VL53L1X Time-of-Flight (ToF) sensor					
- Sensor Connections	VIN	GND	SDA	SCL		
- MCU Connections	+3.3 Supply	GND	PB3	PB2		
Sensor Information	Measurement Range	Ranging Frequency	I²C Bus Speed			
	4 meters	50 Hz	400 KHz			
User Interaction	Single Push-Button (PM0)					
	Reset Button					
LEDs	LED PF0 blinks on every measurement					
	LED PF4 blinks on every full scan					
Supported Libraries	Requires PySerial and Open3D Python libraries on connected PC					
Python Version	3.10.1					

TABLE 1

DETAILED DESCRIPTION

Distance Measurement

With this device, once a scan is initiated via the push button, a 360° rotation begins either clockwise or counter clockwise, depending on the direction of the previous rotation. At intervals of 11.25° during rotation, the VL53L1X ToF sensor takes a new distance measurement. This measurement is obtained by emitting infrared light and calculating the time it takes for the light to bounce back to the detector from an object. The measured photon travel time is halved (divided by two) because the ToF sensor only requires the time for the light to reach the object. The resulting value is then multiplied by the speed of light to determine the distance from the sensor to the object. The VL53L1X board performs these calculations, and the microcontroller receives the distance value in millimeters via I²C protocol.

The sensor's SCL signal manages the clock for synchronizing device communication, while the SDA signal handles data transfer. After receiving data, the microcontroller communicates it to a connected PC using UART protocol, facilitating data transfer into Python. This serial communication process involves Python utilizing the pySerial module to access the UART port connecting the microcontroller and PC, operating at a baud rate of 115200 bps. It's important to note that the specific UART port on one's system may differ; users can locate the corresponding UART port via the device manager.

The data transmitted from the microcontroller is in the format "W, (distance), (z coordinate), (degree)." This is to ensure simplicity of operation, three measurements are being done by the microcontroller. Before starting the program, the user specifies the interval between measurements, this value is pre-set at 10 centimeters. Then for each 360° scan, the z coordinate is increased by that interval. Additionally, for each of the 32 scans in a full rotation, the degree value is being incremented by 11.25 degrees. For clockwise rotations, the degree value starts at zero and increases for each measurement, for counter clockwise rotations, the degree value starts at 360 and decreases for each measurement.

For example, consider the results of two consecutive scans on a hypothetical 30cm radius cylinder, each initialed by the push button:

[Start of scan]
“W, 300, 0, 0.00”
“W, 300, 0, 11.25”
“W, 300, 0, 22.50”
…
“W, 300, 0, 348.75”
[End of scan]
[Start of scan]
“W, 300, 100, 360”
“W, 300, 100, 348.75”
“W, 300, 100, 337.5”
…
“W, 300, 100, 11.25”
[End of scan]

In this example, the first set of measurements is taken clockwise, with the degree value increasing for each measurement while the z coordinate remains at zero for the initial scan. The second scan

is taken counter clockwise, with the degree value decreasing for each measurement and the z coordinate increased by 10 cm. Both distance measurements are in millimeters.
When reading the port in Python, each incoming line from the microcontroller is checked for a leading 'W' character, indicating that the data following is a measurement. Lines starting with 'W' are added to an array until the specified number of scans is reached, at which point the Python program stops reading data. It then parses each entry of that created array into its three values, creating a new 2D array, where each item consists of a three item list of [distance, z coordinate, degree].
Finally, each entry in the 2D array is used to calculate the 3D vector data point [x, y, z] for each measurement. The x and y values are computed using equations (1) and (2) respectively, with the distance measurement and degree values from the embedded list. The z value remains unchanged from the initial reading. The function for this calculation is shown in FIG 2, where "pcoords" represents the initial 2D array and "coords" represents the final measurement 2D array.

$$x = distance * \cos(2\pi * degree) \quad \textbf{(1)}$$
$$y = distance * \sin(2\pi * degree) \quad \textbf{(2)}$$

```
61  coords = []
62
63  for coord in pcoords:
64      temp = [coord[0]* math.cos(math.radians(coord[2])), coord[0] * math.sin(math.radians(coord[2])), coord[1]]
65      coords.append(temp)
```

FIG 2 PYTHON SNIPPET CALCULATING POINT VECTOR

Visualization
After obtaining the final vector measurements for each point in the previously created array, there is an additional array manipulation step required before the data is ready for visualization. Specifically, every odd set of data needs to be flipped. This adjustment is necessary because every alternate set of measurements was conducted in the opposite direction, or counter clockwise, resulting in backward data sets. This configuration leads to a visualization where each point in a frame connects to a point 32 positions out of alignment, creating the appearance of frames colliding into each other. The function to flip every odd set is depicted in FIG 3.

```
70  # Flip every odd set
71  # Iterate over the list in steps of 32
72  for i in range(0, len(coords), 32):
73      # Check if the current set is odd
74      if i // 32 % 2 == 1:  # If the index divided by 32 has a remainder of 1, it's an odd set
75          # Reverse the sublist of 32 values
76          coords[i:i+32] = reversed(coords[i:i+32])
```

FIG 3 PYTHON SNIPPET FIXING DATA SET

Next, the finalized data array can be written to an .xyz file, serving as the storage method for 3D scans. This allows any set of scans to be re-visualized using the following Python program.

Utilizing the Open3D Python library, the data can be visualized by reading from the .xyz file and creating a point cloud, transforming each previously computed data vector into a point in space. This cloud can be visualized, without any connecting lines between points, and still show an

accurate capture of the scanned environment. The Python snippet for point cloud visualization can be seen in FIG 4.

```
88    # Get Point cloud data from file
89    point_cloud_data = o3d.io.read_point_cloud(filename, format="xyz")
90
91    # Print out data to make sure it exists
92    print("Point cloud array: ")
93    print(np.asarray(point_cloud_data.points))
94
95    # Display the data
96    print("Spawining separate window with point cloud: ")
97    o3d.visualization.draw_geometries([point_cloud_data])
```

FIG 4 POINT CLOUD VISUALIZATION IN PYTHON

For a more detailed and visually appealing model, lines can be drawn between points within a single frame (defined as the 32 data points in a scan) and between frames. This process is straightforward in Python, by creating a line array where every line is a vector with the start and end point [start, end]. A function iterates through the points in a frame to generate line vectors between adjacent points and connects the first and last points in the frame. It also establishes connections between corresponding points across multiple frames and incorporates these line vectors into the array. This new visualization is then output, this snippet is shown in FIG 5.

```
100   # Define connections
101   lines = []
102   y = 0
103   num_scans = 32
104   for x in range(len(coords)):
105       print(y, ((len(coords)/num_scans)-1))
106       if (((x+1)%num_scans) != 0) or (x==0):
107           # Connect adjacent points
108           print("General case: ", x)
109           lines.append([[x], [x+1]])
110       else:
111           # Connect first and last points in a frame
112           print("Special Case: ", x)
113           lines.append([[x], [0+(y*num_scans)]])
114           y += 1
115           print(y)
116           # Connect frames
117       if(y < ((len(coords)/num_scans)-1)):
118           print("Connecting Lines: ", x, (x+num_scans))
119           lines.append([[x], [x+num_scans]])
120
121   line_set = o3d.geometry.LineSet(points=o3d.utility.Vector3dVector(np.asarray(point_cloud_data.points)),
122                                   lines=o3d.utility.Vector2iVector(lines))
123
124   print("Spawining separate window with lines")
125   o3d.visualization.draw_geometries([line_set])
```

FIG 5 LINE VISUALIZATION IN PYTHON

APPLICATION NOTE, INSTRUCTIONS, AND EXPECTED OUTPUT

Application Note

The designated testing area for this device was in the *Information Technology Building (ITB)* at *McMaster University*, specifically on the second floor in section 'D'. The scanning process involved 17 scans with a z interval of 45 centimeters. Below are images comparing the line visualization created using Open3D with photographs of the actual area.

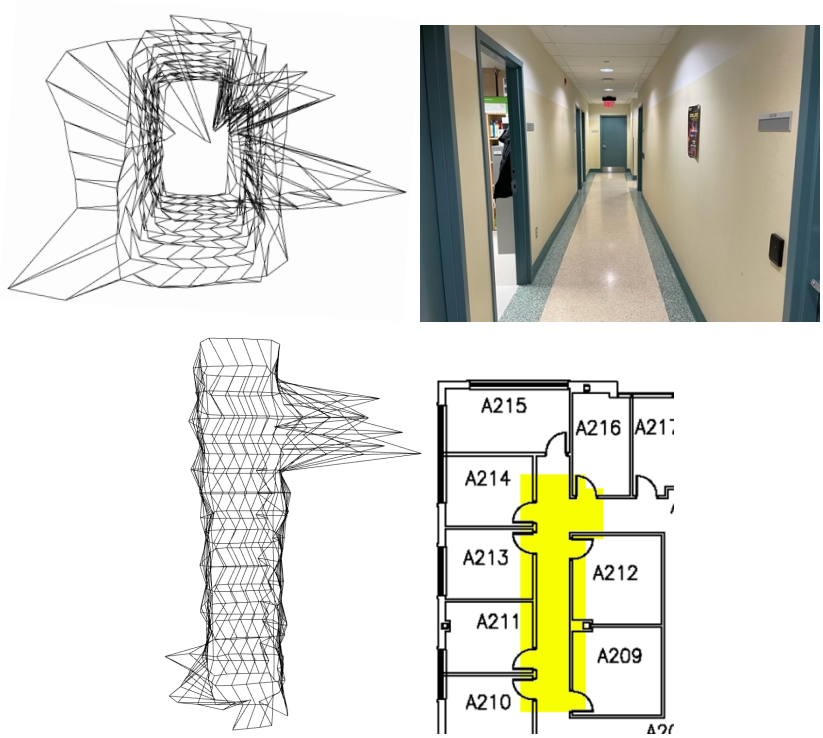

FIG 6 VISUALIZATION COMPARISON IMAGES

Instructions

If the device is disassembled, follow these steps:

1. On a breadboard attach a single push button in a pull-up setup, with a 1KΩ resistor connected to ground and a wire connecting to PM0 on the microcontroller.
2. Carefully attach the ToF sensor to the motor shaft, ensuring a secure connection.
3. Connect the ULN2003 Driver to the microcontroller (MSP432E401Y) as follows:
   - + Terminal to +5V Supply
   - - Terminal to GND
   - IN0 to PH0
   - IN1 to PH1
   - IN2 to PH2
   - IN3 to PH3
4. Connect the VL53L1X ToF sensor to the microcontroller (MSP432E401Y) as follows:
   - VIN to +3.3V Supply
   - GND to GND
   - SDA to PB3
   - SCL to PB2

5. Securely mount the motor base to the side of a box, ensuring the box is comfortable to hold.

Software Setup:

1. Connect the microcontroller to a PC with KEIL installed using a micro-USB cable.
2. In the main program file, adjust the variable "z_interval" on line 167 to set the desired distance between scans. This value is preset to 100, corresponding to 10 centimeters.
3. Verify that the correct target, compiler, and debugger settings are selected. For the components used, use default compiler 6 and XDS-110 for CSMIS-DAP Debugger.
4. Translate, build, and download (in that order) the KEIL program onto the microcontroller.
5. Ensure the correct Python version is installed along with the required libraries (pySerial and Open3D).
6. Use the following command in Command Prompt to identify the correct port: "python3 -m serial.tools.list_ports --v"
7. Modify line 10 in the Python program so that "s.port" matches the correct port (defaulted to 'COM3').
8. Update line 29 in the Python program to set "setsOfScan" to the desired number of scans (defaulted to 3).
9. Save the Python file.

Operation:

1. In Command Prompt, navigate to where the Python program is held.
2. Run the Python program.
3. When positioned at the desired starting location, press 'ENTER' on the PC to allow the Python program to begin reading data.
4. Press the 'RESET' button on the microcontroller to ensure all values are reset.
5. Press the push button to start scan, hold still for an optimal scan. Confirm functionality by observing the onboard LEDs; PF0 will blink with each measurement, and PF4 will be on while a scan is in progress.
6. Once PF4 is turned off and the motor has stopped running, move to the next location for scanning (this should be the same displacement as the KEIL program).
7. Press the push button to start another scan.
8. Repeat steps 5-7 until all the desired scans are done. Once the specified number of scans is completed the Python program will immediately break and display the point cloud visualization in a new popup window.
9. Closing the point cloud window will pop up a new line visualization window.
10. Done.

Expected Output

There are three desired outputs from the device: a point cloud visualisation, a connected line visualization, and a saved .xyz file with the scanned data. Below are the same results from the scan done of the assigned environment, shown in the Application Note.

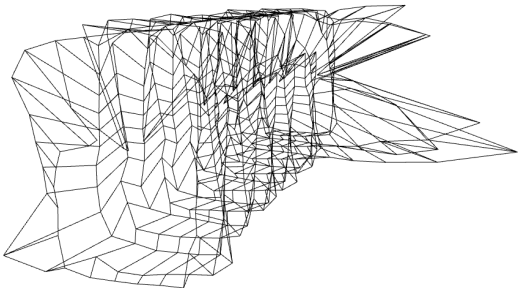

FIG 7 LINE VISUALIZATION

Shown above in FIG 7 is another angle of the line visualization made by Open3D. These line visualizations give a clearer view of the scanned area, compared to a point cloud display.

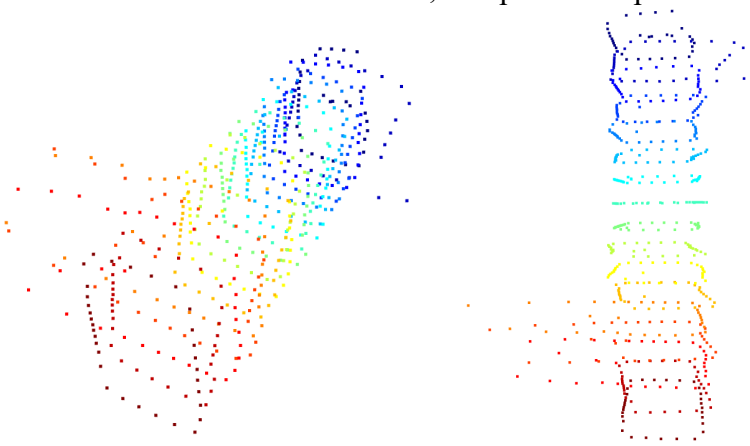

FIG 8 POINT CLOUD VISUALIZATION

Shown above in FIG 8 is the point cloud visualization of the same area, this is an interesting visualization that can still give a good sense of the area.

```
1081.0 0.0 0
991.573918487666 197.23631555830565 0
841.6542541177822 348.6246068845968 0
695.9400654972303 465.012285037407 0
362.7457787486989 362.7457787486989 0
247.228753693723 370.0039774746326 0
316.09651513356642 763.1244938543228 0
33.94571603080633 170.6566387901621 0
7.7152748346283 25e-15 126.0 0
-266.88356051806335 1341.7142635916193 0
-490.21747685967995 1183.4896811469582 0
-690.5737996433656 1033.5167280920637 0
-870.4484476406399 870.4484476406401 0
-196.2268285034007 131.1145749926261 0
-1006.1048109047913 416.7422578455829 0
-890.5530346061332 177.14201239064437 0
-775.0 9.491012693391988e-14 0
-692.4344079646806 -137.73376734338663 0
-601.4455756648478 -249.12691446967338 0
-531.3090822613264 -355.00937889952587 0
-463.15494167718873 -463.15494167718856 0
-385.0101714825843 -576.2084413256639 0
-290.83940859746804 -702.1484447085 78 0
-177.92237367870933 -894.4761757277461 0
-1.8663617219005662e-13 -1016.0 0
260.05539924749905 -1307.3867787775062 0
489.4521099949501 -1181.6419220819355 0
716.6856005952864 -1072.5957998702836 0
943.9875528840407 -943.9875528840412 0
1193.1588936541525 -797.2432843831291 0
1415.3834438072915 -586.2710183833173 0
1267.1745822809735 -252.0566960448383 0
1528.9060796679594 208.07498541235458 450
1389.9123061309958 478.58518705202033 450
1294.889422301389 755.1571915912439 450
1175.8604816387333 1040.3134756987079 450
1121.0044658554405 1474.19570869412 450
791.6982911010739 1623.2220476162956 450
398.9612528380618 1515.3609862781489 450
84.06383893949796 1374.4316174269109 450
-171.66523423844933 1261.3722873735012 450
-398.49542105555975 1157.3147365335637 450
-611.0778341562233 1047.832467812932 450
```

FIG 9 SNIPPET OF .XYZ DATA

Shown above in FIG 9 is the saved .xyz data of the scanned area. It is important to note that the same Python program cannot be used to visualize this, requiring a separate program built just for visualization such that the .xyz file does not get overwritten.

LIMITATIONS

(1) Our system adopts a strategy to ensure precision in trigonometric calculations and decimal handling by leveraging the computational power of the user's computer and operating system, rather than processing these tasks directly on the MSP432E401Y microcontroller. This approach is essential due to the microcontroller's 32-bit floating-point unit, which provides limited precision compared to personal computers that utilize larger floating-point units (64-bit or 80-bit) for enhanced accuracy in mathematical operations involving decimal numbers. By offloading these calculations to the user's system, we optimize accuracy when converting distance measurements from the ToF sensor into x and y coordinates

(2) Using equation (3) below, we calculate the maximum quantization error to be 0.06103515625 mm. This value is determined by substituting 16 for x, representing the number of resolution bits for the ADC, and 4000 mm for d, which is the maximum distance that can be measured by the ToF sensor.

$$Max\ Quantization\ Error\ = \frac{d}{2^x} \textbf{ (3)}$$

In this equation:

- d = 4000 mm (maximum distance measured by the ToF sensor)
- x = 16 (resolution bits for the ADC)

Substituting these values into equation (3), we find the maximum quantization error to be 0.061035156252164000 mm

(3) The maximum standard serial communication rate is 128000 bps, however in the program a standard serial communication rate of 115200 bps was selected. This can be verified by checking the bits per second shown by the properties manager on Windows.

(4) The communication method between the microcontroller and ToF module was I^2C with a maximum rate of 400KHz.

(5) The primary limitation on system speed was determined to be the ToF sensor and the stepper motor. Through testing various time delays with the motor and ToF function, it was observed that using time delays less than 10ms led to operational difficulties. For instance, the ToF sensor attempted to take measurements at a significantly faster rate than the designated degree intervals of the stepper motor. As a result, there were instances where the sensor would perform 8 measurements within a 45-degree rotation and then cease measurements for subsequent intervals.

(6) To configure the system bus speed, change the PLL.h header file in the KEIL project. In the header file set the PSYDIV variable value to 7, which corresponds to 60 MHz as calculated with equation (4).

$$Bus\ Frequency\ = \frac{480MHz}{PSYSDIV+1} \textbf{ (4)}$$

CIRCUIT SCHEMATIC

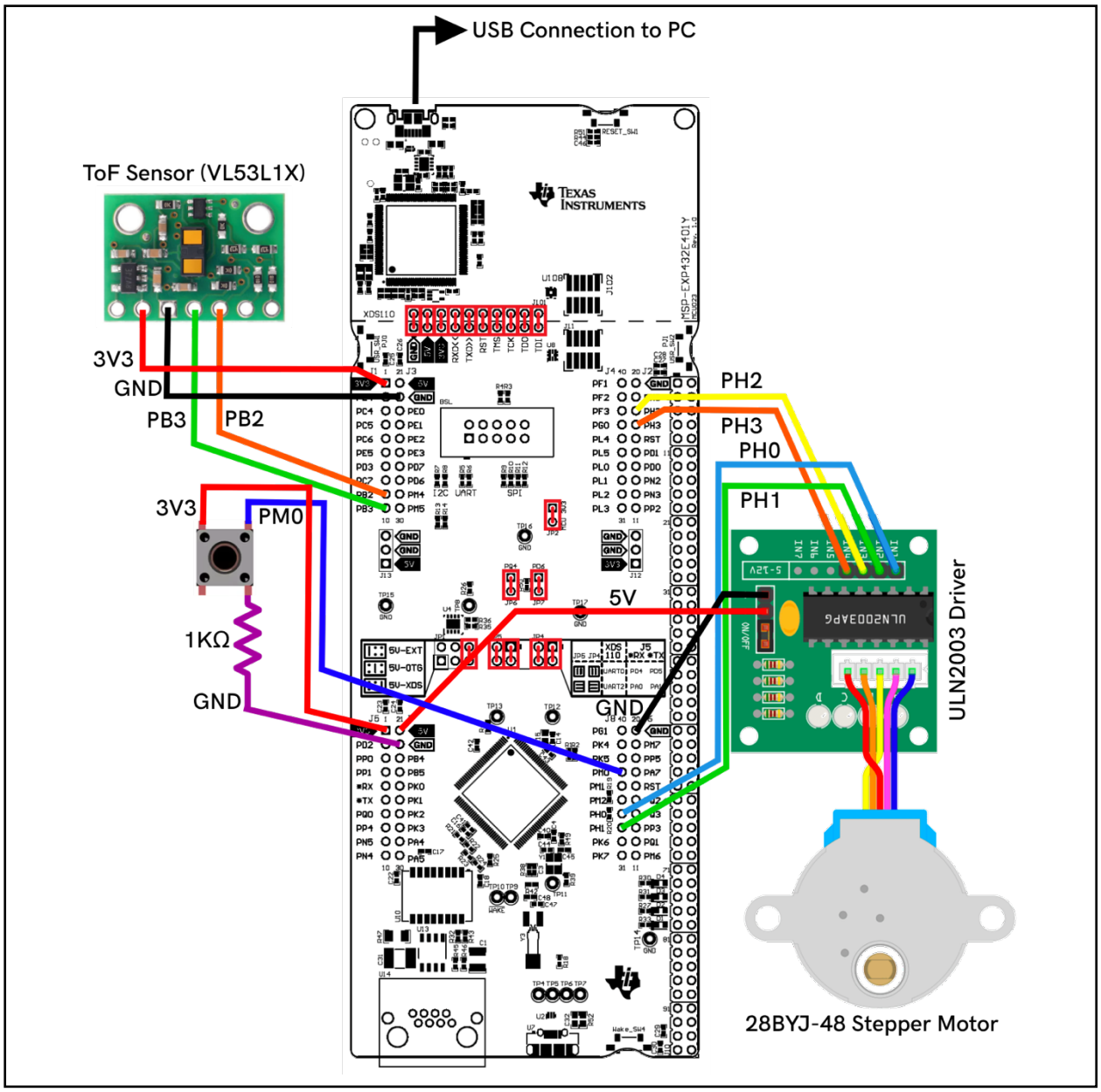

FIG 10 CIRCUIT SCHEMATIC

PROGRAMMING LOGIC FLOWCHARTS

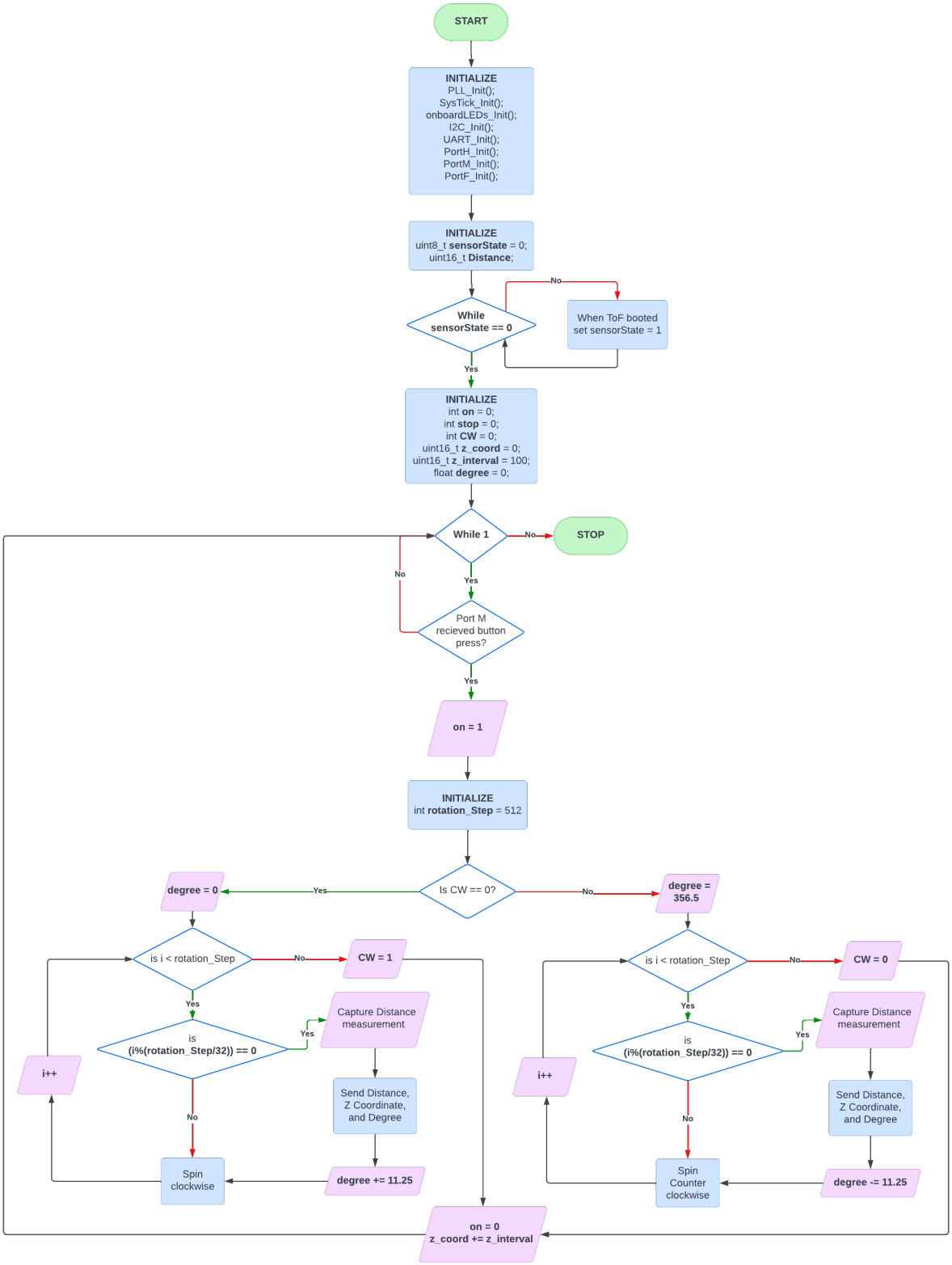

FIG 11 Keil C Program Flowchart

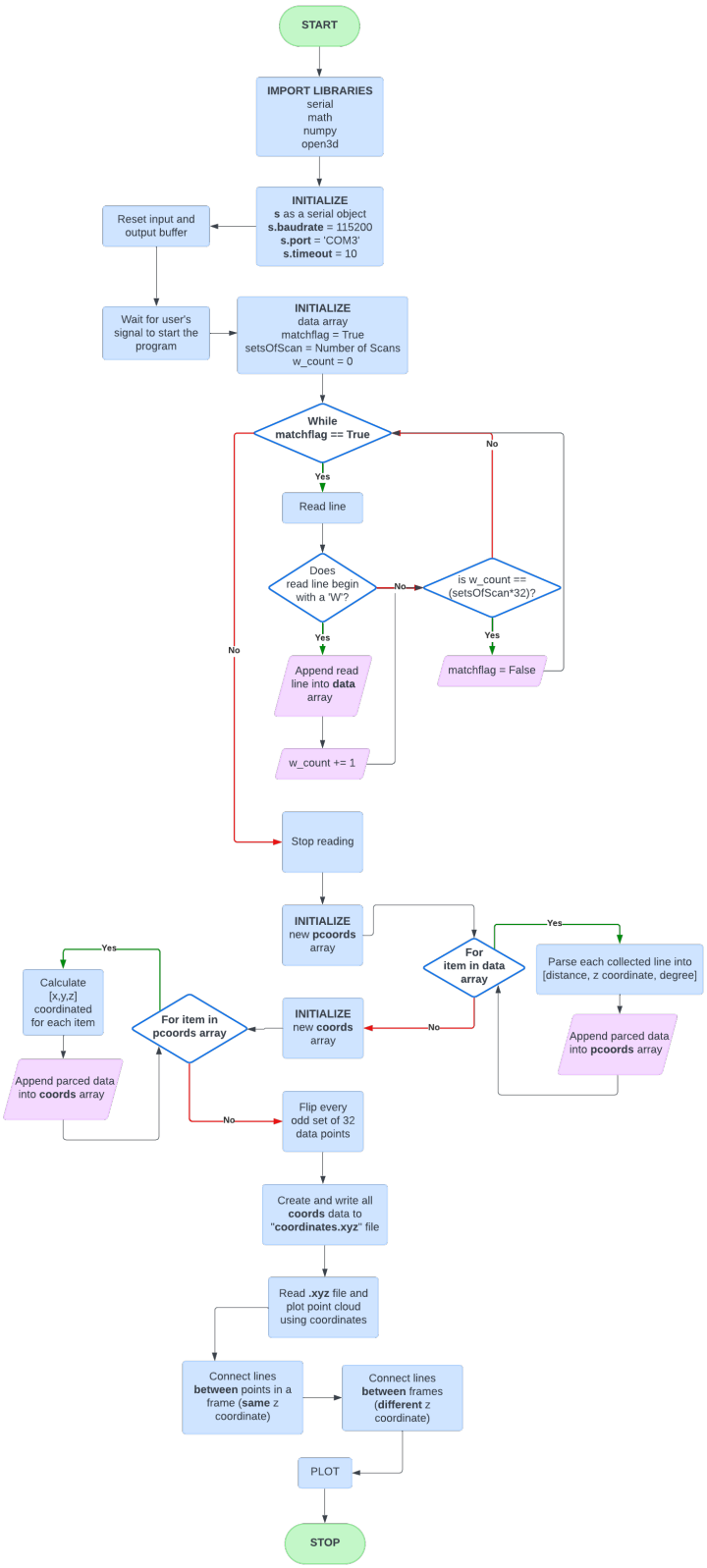

FIG 12 Python Program Flowchart

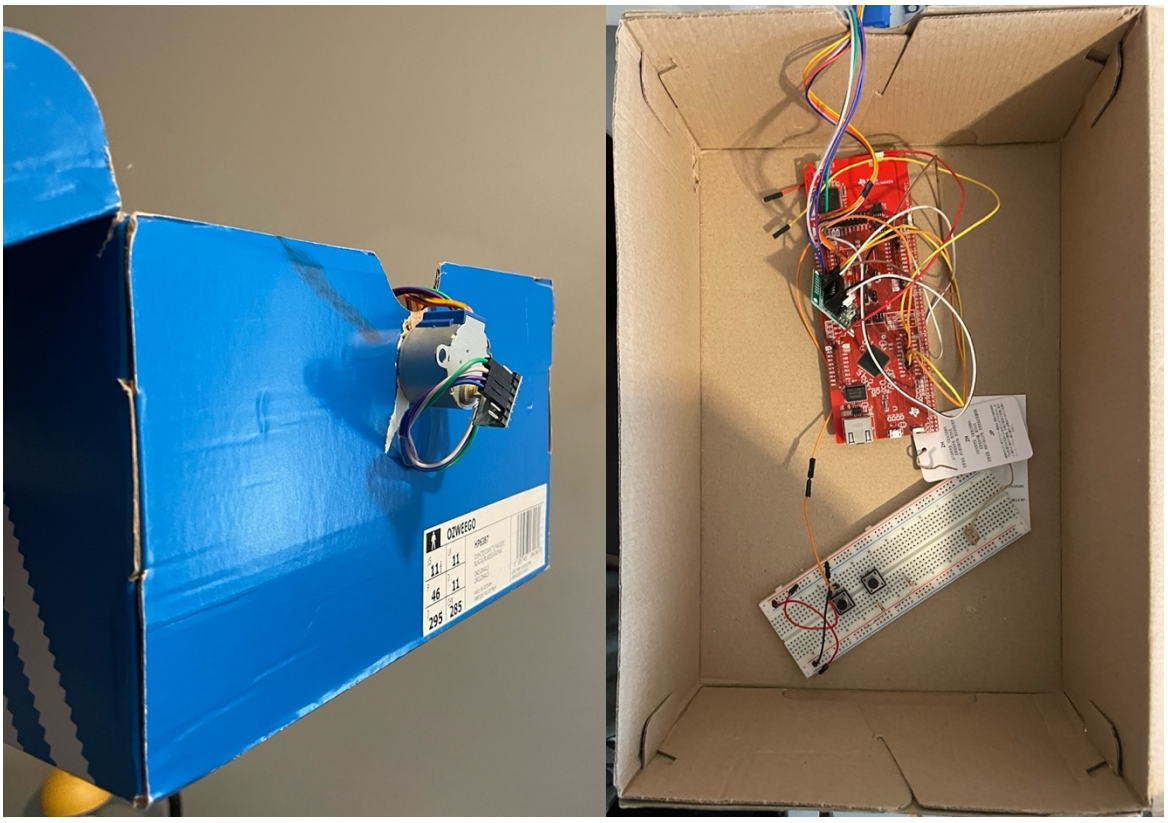

FIG 13 Complete Product Image

REFRENCES

[1] Texas Instruments, "MSP432E4 SimpleLink™ Microcontrollers Technical Reference Manual," Literature Number: SLAU723A, Oct. 2017, revised Oct. 2018. [Online]. Available: https://www.ti.com/lit/ug/slau723a/slau723a.pdf. [Accessed: April, 2024].