2DX4: Microprocessor Systems -- Final Project

Instructor: Drs. Doyle, Haddara, and Shirani
Noah Betik – L06 – betikn - 400246583

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 Noah Betik (betikn, 400246583).

1.0 -- Device Overview:

1.1 -- Device Features:

- Texas Instruments MSP432E401Y microcontroller based on Cortex M4F used for data acquisition and inter-device communications
- STM VL53L1X time-of-flight sensor used for accurate ranging up to 4m on a custom breakout board to deal with signal conditioning and ADC
- Sensor mounted on stepper motor for x-y plane measurement with accuracy up to 1.4 degrees or 256 measurements per rotation
- I2C communications between VL53L1X and the MSP432E401Y
- UART serial communications between MSP432E401Y and a Windows 10 PC
- Custom Python application based on the Open3D API for processing of raw data and visualization in a point-cloud format

1.2 -- General Description & Block Diagram:

This device was designed to be used as a spatial mapping system ideal for mapping a hallway or small room to a variable degree of accuracy, and consists of three separate COTS components. Firstly, the STM VL53L1X sensor acquires a distance measurement by measuring the flight time of a laser reflected off a nearby surface, and transmits that data over an I2C bus to the TI MSP432E401Y microcontroller. This board, which is the heart of the system, requests a measurement from the sensor each time after rotating a specified number of degrees, and stores each measurement received over I2C in an array. Each full rotation is triggered by an active-low pushbutton; once each rotation is completed, it will sequentially send each measurement in the array to a Windows PC over UART. To coordinate this data transfer, a custom Python application on the PC sends a signal character over UART to acknowledge that it is ready to receive.

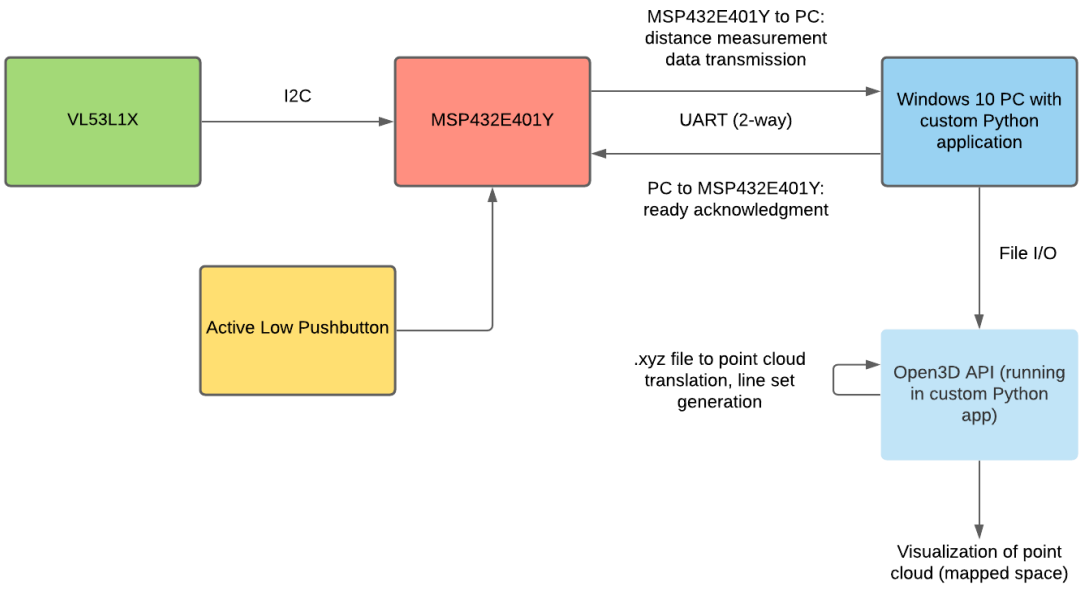
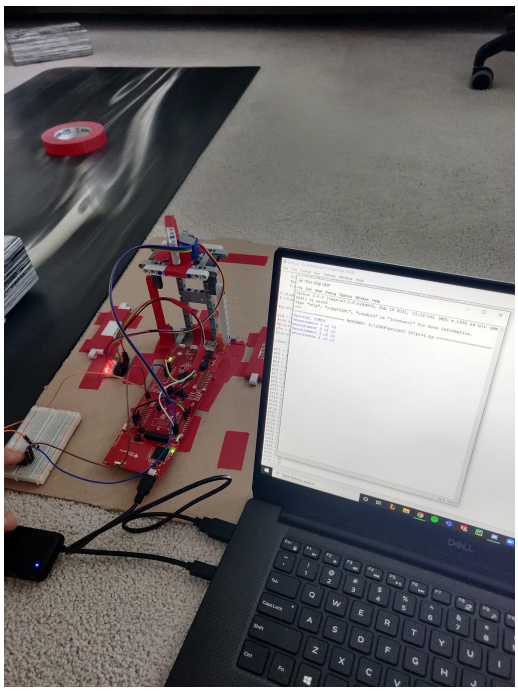


Figure 1: Block Diagram

Once all values have been received, the same application takes the raw distance measurement and converts it into a set of [x,y,z] coordinates. From here, the data is written to a .xyz file. Using the Open3D API, the data in the .xyz file is first converted into a point cloud format. Next, the set of lines connecting each point to its immediate neighbours is generated. Finally the set of lines and the point cloud data is passed to the Open3D geometry drawing function to give a visualization of the scanned area.

1.3 -- Photos:



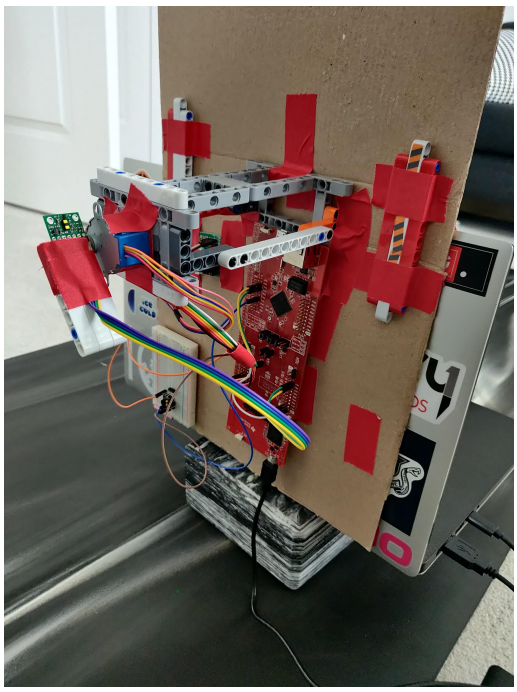


Figure 2 (left) and Figure 3 (right) -- full system image
Note: left image has the system deconstructed to show both the sensor and laptop screen

Figure 4 -- laptop screenshot of data collection

2.0 -- Features:

Bus/Clock Speed:

- 60 MHz

Operating Voltages:

- MSP432E401Y, VL53L1X, pushbutton -- 3.3V
- Stepper Motor -- 5V

Memory Resources (MSP432E401Y):

- uint16_t array to store distances measurements for one rotation -- 16 bits * number of measurements per angle = 2 bytes per measurement/rotation
- Various integers (uint8_t, uint16_t)
- Total flash memory of the MSP432E401Y is 1024 KB

ADC Channels:

- 2 SAR-based 12-bit ADC channels at up to 2 million samples per second

Cost:

- MSP432E401Y -- $65.53
- VL53L1X with breakout board -- $15.93
- Generic stepper motor & custom ULN2003AN-based control board -- ~$20
- Miscellaneous breadboard/wires/pushbutton/etc -- ~$10
- TOTAL: ~$111.46

Languages:

- MSP432E401Y/VL53L1X data acquisition system -- C / Cortex M4F native assembly
- Open3D custom application -- Python

Serial Communication Details (UART):

- Baud rate of 115200, half-duplex configuration
- 8-bit packets, 1 stop bit, no parity bits
- Terminator character “\0” (null character)

3.0 -- Device Characteristics Table:

Note that all GPIO pins stated are on the MSP432E401Y microcontroller.

Characteristic	Specification
GPIO Pins for VL53L1X	3.3V, GND, PB2 (SCL), PB3 (SDA)
GPIO pins for stepper motor	5V, GND, PH0-PH3 (IN1-IN4 on breakout board)
GPIO pins for pushbutton	PM0, 3.3V, GND
Bus Speed	60 MHz (PSYSDIV = 7 in PLL.h)
Serial Port	COM10 (USBSER000) at 115200bps (COM port may vary by PC)

4.0 -- Detailed Description:

4.1 -- Distance Measurement:

In this section, please refer to the flowchart provided in Figure 8 (in Appendix) as an overview of the system. Before any of the distance measurements are done, all GPIO ports in this system are initialized, and the setup process for the time-of-flight sensor is run. From here, the main body of the program is centred around a nested double for loop. The outer loop controls the number of iterations in the z-axis (the direction moved by the user), and contains an if statement whose sole purpose is to wait for a logic low signal (as the pushbutton is wired as negative logic). Upon receiving this signal, the inner loop is entered.

In this inner loop, several functions from the VL53L1X API are used. Firstly, *VL53L1X_CheckForDataReady(dev, &dataReady)* ensures that the sensor is available to receive data. Once the sensor is ready, *VL53L1X_GetRangeStatus(dev, &RangeStatus)* and *VL53L1X_GetDistance(dev, &Distance)* are used to confirm that the sensor is in range and to take a distance measurement, respectively. The variable *Distance*, whose reference is passed to the distance request function, is used to temporarily store this measurement before placing it in the array named *distances[]*. At this point, the function to spin the stepper motor clockwise is called to advance the sensor to its next position. This function takes two arguments -- the first is how many steps to rotate (2048 steps = 360 degrees) and the second is the delay between steps. Experimental testing showed that 2ms is the smallest delay for the motor to spin without error. By default, the motor is set to spin 64 steps, which corresponds to an angle of 11.25 degrees. This default angle was chosen because it provided reasonable accuracy while maintaining a reasonably short operation time. As a result, 32 measurements are taken per rotation.

In particular, the VL53L1X takes its distance measurement by reflecting a Class 1 940nm laser off a nearby surface, and measuring the time it takes to return to the SPAD (single photon avalanche diode) array sensor. The distance can then be calculated with the following formula:

$$d = c * (t/2)$$

where d is the distance, c is the speed of light, and t is the elapsed time. As the sensor is effectively light-based, it is most effective in darker environments, where it is accurate up to 4m. In ambient light, it is likely to be accurate only up to about 170cm in long-distance mode. All signals are sent via I2C to the microcontroller, where the sensor uses a device address of 0x52. Information is stored in 8-bit packets followed by an acknowledgement bit. More info about the VL53L1X can be found in the datasheet provided in the appendix.

After the 32 distance measurements for a full rotation are taken, the microcontroller sends the values in *distances[]* to the PC using UART serial communication at a baud rate of 115200 and settings of 8-N-1. An overview of this can be found in the flowchart provided in Figure 8. To prevent data loss due to timing errors, a handshake system was implemented between the PC and the micro. Firstly, the micro waits to receive a predefined input character from the PC as an acknowledgement that it is ready to receive. Once this character is received, the micro sends the *printf_buffer* which contains the i^{th} value of *distances[]* using the *UART_printf()* function. Since the PC can only receive one byte at a time using the pyserial library's *read()* function, and since

the distance measurement may be anywhere from 1 byte to 4 bytes (as each digit is a character requiring 1 byte to store), it effectively checks to see if the received byte is "," inside an infinite while loop before either concatenating the non-comma digit to the current value, or converting the current received buffer to an integer and breaking out of the while loop (this process is more clearly viewed in Figure 9). At this point, the raw distance measurement is converted into a set of [x,y,z] coordinates. The angle is known based on the iteration of the for loop (*angle = 360*i/spins* where *i* the iteration variable and *spins* is the number of measurements per rotation), and the coordinates are calculated as follows. Note that the Python math library functions used are defined for radians, which is why the conversion from degrees is necessary.

*x = r*cos((pi/180) * angle)*

*y = r*sin((pi/180) * angle)*

*z = [number of full rotations completed so far] * [Δz / rotation]*

For example, a distance measurement of 1678 mm at an angle of 22.5 degrees on the 4th rotation with Δz = 20cm would yield the following calculations:

*x = (1678) * cos((pi/180) * 22.5) = 1678*cos(pi/8) = 1678(0.9239) = 1550.3mm*

*y = (1678) * sin((pi/180) * 22.5) = 1678*sin(pi/8) = 1678(0.3827) = 642.3mm*

*z = 4 * (200mm) = 800mm*

Therefore, [x, y, z] = [1550.3, 642.2, 800.0]

In the Python code, values are rounded to one decimal place, as any further accuracy is meaningless considering the accuracy of the sensor. This entire process is repeated for each full rotation.

Once all rotations have been completed, Python file I/O is used to write the values of each coordinate to a local .xyz file. This file format takes each point on a new line with a space between each value. For example, a .xyz file with three measurements would appear as follows:

1550.3 642.3 800.0

373.6 987.8 400.0

975.1 2226.8 1200.0

The number of coordinates to be written to the file is the number of rotations times the number of measurements per rotation. For the default values of 10 rotations and 32 measurements per rotation, this is 320 points. From here, the data is considered processed and ready to pass to the visualization section of the Python application.

4.2 -- PC Specifications:

CPU -- Intel i5-8300H @ 2.30GHz
Memory -- 8GB RAM
Storage -- 256GB SSD
Relevant Peripherals -- USB-A 3.1
Operating System -- Windows 10 Home
Python Version -- 3.8.8

Required Python Libraries/Modules -- math, pyserial, Open3D, numpy (any supporting libraries/modules will be installed automatically when installing the above with pip)

4.3 -- Data Visualization:

For this section, please refer to the flowchart provided in Figure 9 (in appendix). The visualization of the data points mapped by the time of flight sensor is done in the same Python program used to receive the data from the microcontroller, and relies heavily on the Open3D Python API, as well as the NumPy library. Firstly, Open3D's *read_point_cloud()* method is used to generate a point cloud object from the .xyz file created in the previous step. This point cloud is then passed to the NumPy method *asarray()* to prepare it to be passed to the Open3D line set generator (which for points typically accepts NumPy arrays of size (n, 3). While a visualization of just the points could be generated here, in order to give a higher degree of clarity and orientation, lines to connect all adjacent points must be generated. Before the lines can be defined, arbitrary labels for each point must be created. The naming convention is "pt*x*" where (1 < *x* < n) is the number of the point and *n* is the number of points. Two for loops are then used to firstly connect adjacent points within a plane, and secondly, connect points of the same angle between planes. This set of lines is stored in an array *lines[]* and is used to create a mesh when passed to Open3D's geometry drawing method.

However, before this can be done, a *LineSet* object from the Open3D API must be created. This constructor uses the Open3D methods *Vector3DVector* and *Vector2iVector* in order to generate Open3D format sets of 3D lines and points inside a single object. Finally, the *LineSet* object is passed inside an array wrapper to Open3D's *draw_geometries()* method in order to generate a plot. This plot is opened in a new window, and can be viewed at any orientation by left-clicking and dragging to rotate, or right clicking and dragging to pan.

4.3 -- Data Visualization Example:

As an example, a basement hallway/room was scanned with the system. A rough multiview sketch of the area created using Solidworks is provided below, followed by the actual scan of the area.

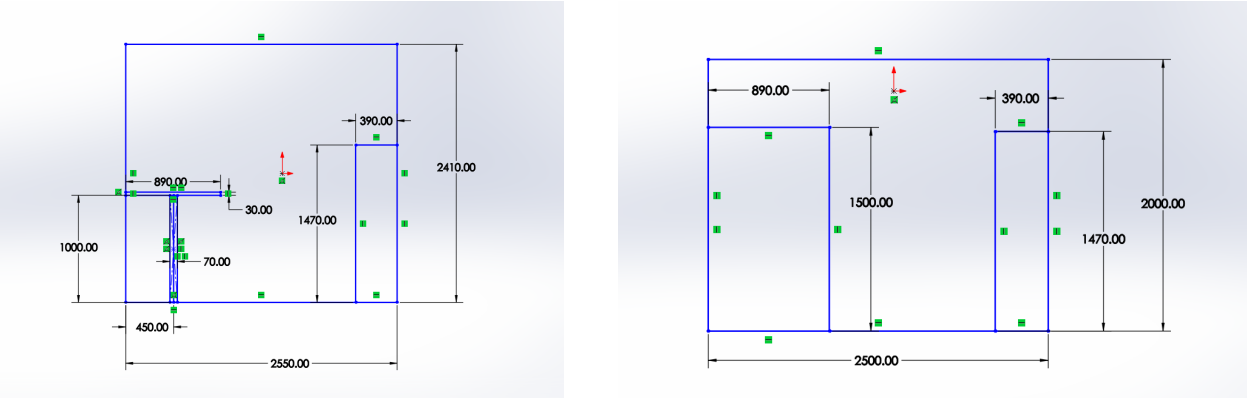


Figure 5 -- front view (left) and top view (right)

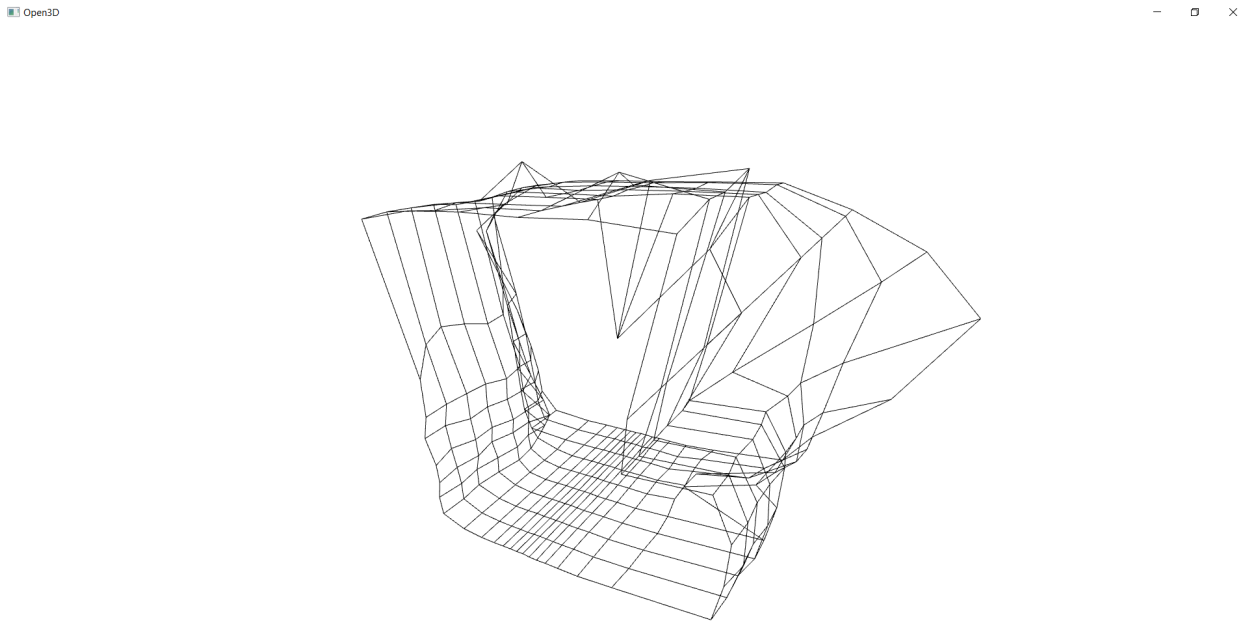


Figure 6 -- Scan of the room as seen in the video
Note: the resulting scan is mirrored across the y-axis and I was unable to diagnose and resolve the problem in time to submit.

5.0 -- Application Example:

5.1 -- Axis Definitions:

- x-axis: Captured by rotational scan, parallel to the floor,
- y-axis: Captured by rotational scan, perpendicular to the floor
- z-axis: moved along by the user

5.2 -- Step-By-Step Usage Guide

1. Connect MSP432E401Y to the PC via USB. The data acquisition program will begin running immediately, and the time-of-flight sensor will boot (shown by flashing LEDs)
2. Open plot.py in IDLE
3. Determine USB serial port to use by opening Windows Powershell
    a. Run the following command “python -m serial.tools.list_ports -v”
    b. Find the port associated with UART, as shown in the following image
    c. Change “s = serial.Serial("COM10", 115200)” such that “COM10” is replaced with your serial port. Note the red underline in the image below.

```
PS C:\Users\NWHAL> python -m serial.tools.list_ports -v
COM3
    desc: Standard Serial over Bluetooth link (COM3)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_VID&00010057_PID&0023\7&3322EC06&0&FCA89AED20F5_C00000000
COM4
    desc: Standard Serial over Bluetooth link (COM4)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&3322EC06&0&000000000000_00000002
COM5
    desc: Standard Serial over Bluetooth link (COM5)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_VID&000101DA_PID&0001\7&3322EC06&0&C0288DBAFB2F_C00000000
COM6
    desc: Standard Serial over Bluetooth link (COM6)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&3322EC06&0&000000000000_00000003
COM7
    desc: Standard Serial over Bluetooth link (COM7)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&001D\7&3322EC06&0&E8D03C436406_C00000000
COM8
    desc: Standard Serial over Bluetooth link (COM8)
    hwid: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&3322EC06&0&000000000000_00000004
COM9
    desc: XDS110 Class Auxiliary Data Port (COM9)
    hwid: USB VID:PID=0451:BEF3 SER=ME401023 LOCATION=1-1:x.3
COM10
    desc: XDS110 Class Application/User UART (COM10)
    hwid: USB VID:PID=0451:BEF3 SER=ME401023 LOCATION=1-1:x.0
8 ports found
```

Figure 7 -- Serial Port Selection

4. Save and run plot.py as soon as the LEDs stop flashing (sensor has booted)
5. Press the pushbutton to begin the first rotation
6. Once the motor has returned to its original position, slide the device forward for the defined z distance (200mm by default) and press the pushbutton again to start the next rotation. The IDLE console will display what number rotation is in progress.
7. Once all rotations are complete, look for a new Python window that will have the visualization of the scan. It may not pop up in front of the existing windows.

6.0 -- Limitations:

1. As a general rule, the accuracy of floating-point numbers is limited, especially on a microcontroller where space may be limited. However, by only recording integer measurements on the microcontroller, and offloading all floating-point operations to the Python script (which has use of the PC's considerably more vast resources), a higher degree of accuracy can be achieved. Python's math library, which includes the sin and cos functions, as well as a method to return a highly accurate value of pi, means that full precision is maintained until the final value is forcibly rounded to one decimal place as the final step.
2. Max quantization error = $\Delta = V_{FS}/2^m$
   V_{FS} = 5.5V - 2.6V = 2.9V
   Sensor send 2 bytes over I2C → 16 bit precision
   Δ = (2.9V) / 2^{16} = **0.044mV**

3. The maximum baud rate between the PC and the microcontroller is 115.2kbps. This was confirmed with RealTerm, as the maximum selectable baud rate was 115200bps.
4. The time-of-flight sensor and the microcontroller communicate over I2C at a rate of up to 400kHz
5. In the entire system, the two primary limitations on speed are the timing budget of the VL53L1X and the speed of the stepper motor. The timing budget for the VL53L1X is set by default at 100ms, and is followed by an inter-measurement time delay that must be greater than or equal to the timing budget (and is set to 200ms by default). This results in a time of at least 300ms by default where no actions are occurring. Furthermore, the stepper motor is limited to a speed of roughly 15RPM. By comparison, high-end commercial systems such as the RPLIDAR A3 run at a speed of 900RPM.

7.0 -- Appendix

VL53L1X Datasheet: https://www.st.com/resource/en/datasheet/vl53l1x.pdf

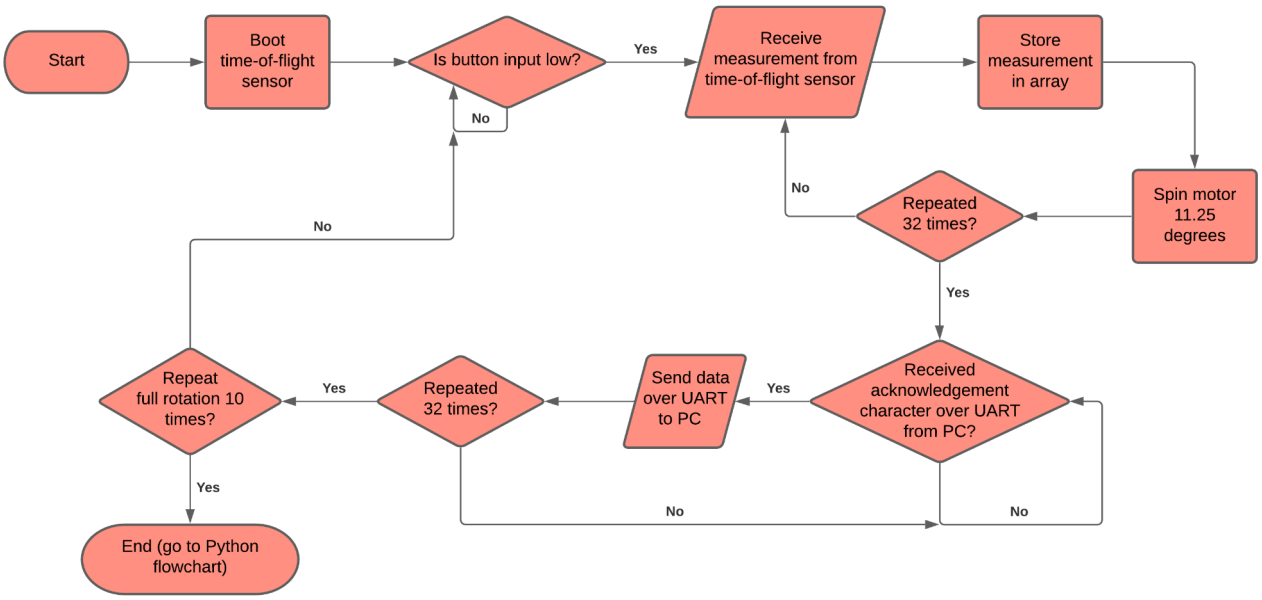


Figure 8 -- Microcontroller Flowchart

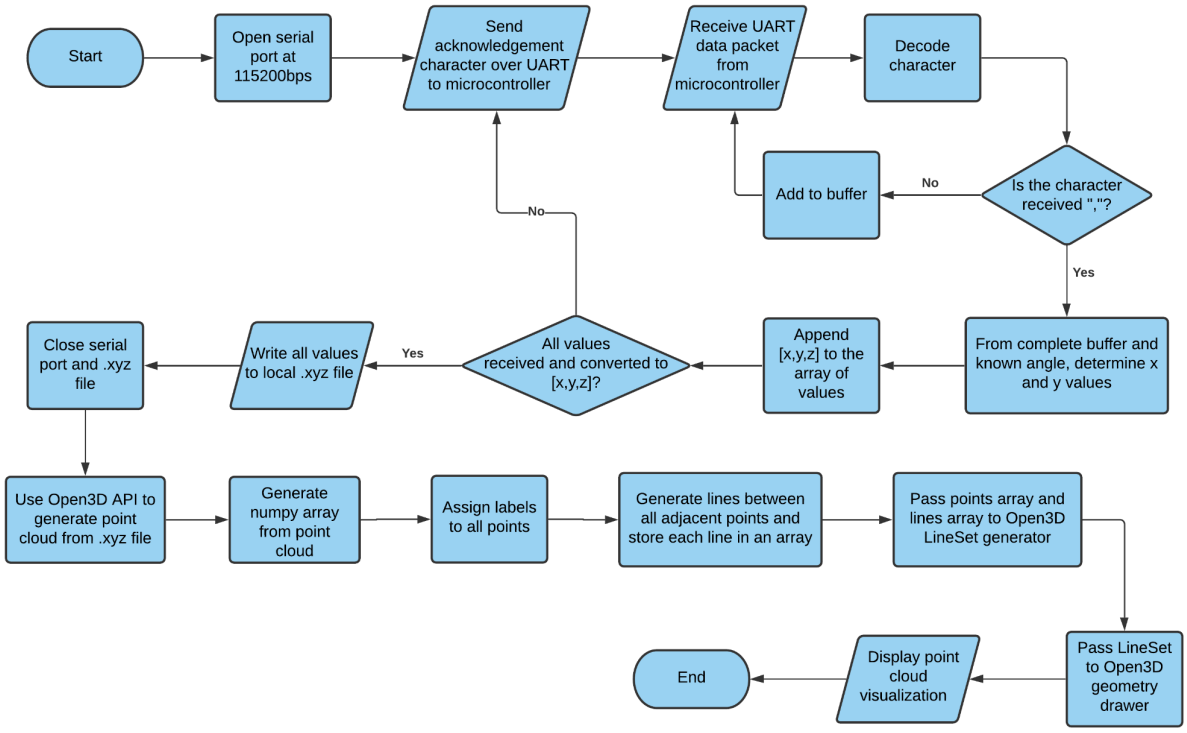


Figure 9 -- Python Flowchart

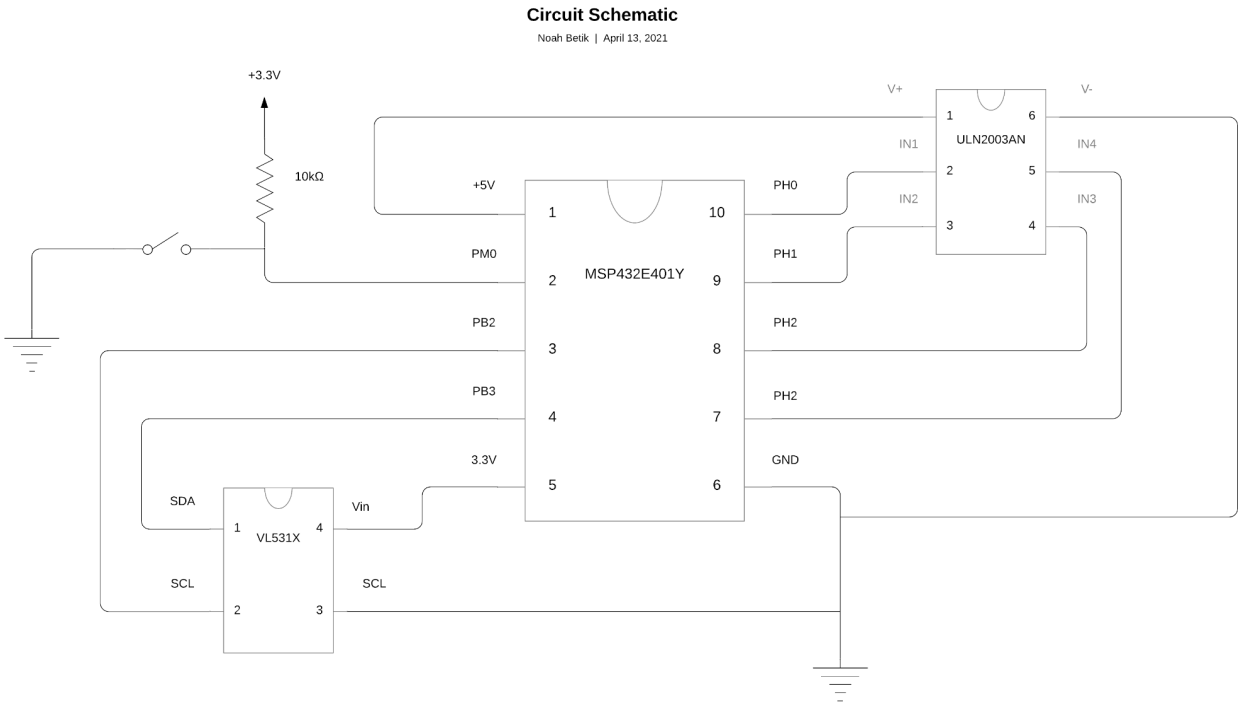


Figure 10 -- Circuit Schematic

Demonstration video link:
https://drive.google.com/file/d/1hDDuaF80Dhl69DNWYHtgbw944DvksEgp/view?usp=sharing

Question 1 video link:
https://drive.google.com/file/d/1hHPY_6cYCcJmi86KC_QAu6oguYNvMrWj/view?usp=sharing

Question 2 video link:
https://drive.google.com/file/d/1hSHvuYkXIq65kH1ACUDwGVuRl2ypHpUp/view?usp=sharing

Question 3 video link:
https://drive.google.com/file/d/1hIfsDY0mKfQVfRThkyDpmvy_pYtqazdz/view?usp=sharing