ENGINEERING

McMaster University

Final Project Report

COMPENG 2DX3: Microprocessor System Project

Lab Section: L04

Connor Usaty - usatyc - 400409624

Submitted: April 15, 2024

Table of Contents

Device Overview - Part A: Features

The device consists of multiple components which each have the following respective features:

Texas Instrument MSP432E401Y SimpleLink™ Ethernet Microcontroller [1]:

- Cortex-M4F Processor Core with Floating-Point Unit (FPU)
- 24MHz bus speed (Up to 120MHz)
- I2C Interface (Up to 3.33MHz)
- UART Interface (Up to 128kbps)
- 4 user LEDs to indicate device status
- 2 user buttons to control device status

VL53L1X Time of Flight (ToF) Sensor [2]:

- Max accurate distance measurement of 4m
- Max ranging frequency of 50Hz
- I2C Interface (Up to 400kHz)
- 2.6V - 3.5V operating voltage

28BYJ-48 Stepper Motor:

- 4 LEDs on ULN 2003 Driver Board indicating current phase
- 5 - 12V operating voltage

Serial Communication Protocols:

- I2C protocol and ToF API allows for data transfer between board and ToF
- UART protocol and PySerial allows for data transfer between board and PC

3D Visualization Program:

- Python script parses data, colour codes the datapoints, and writes datapoints into .xyz file
- Open3D reads .xyz file data, connects adjacent points, and plots data visualization

Device Overview - Part B: General Description

The system is an easy to use, cost efficient, and high-performance 3D spatial mapping system, composed of state-of-the-art components such as the MSP432E401Y SimpleLink™ Ethernet Microcontroller, the VL53L1X Time-of-Flight (ToF) laser-ranging sensor, and the 28BYJ-48 Stepper Motor in combination with the ULN 2003 Driver Board. Taking advantage of the components I2C and UART communication capabilities for efficient communication and data transfer.

The system interface consists of 2 user buttons, PJ0 and PJ1, onboard the MSP432E401Y SimpleLink™ Ethernet Microcontroller that allow the user to control the system, and 2 user LEDs, D3 and D4, also onboard the MSP432E401Y SimpleLink™ Ethernet Microcontroller that indicate the current status of the system to the user.

PJ1 can be pressed by the user to initiate each clockwise 360-degree scan of the zy-plane. Each scan should be taken a fixed distance away from the last to incorporate the x-axis. While PJ0 is used to either void and restart single 360-degree scans or to finish the data collection process and initiate the data visualization process. LED D3 is used to indicate UART transmissions occurring, while LED D4 is used to indicate a distance measurement from the ToF sensor in progress. To prevent wires from tangling, each time a valid scan has finished, the motor will do a 360-degree rotation counterclockwise, returning to home.

Device Overview - Part C: Block Diagram and Data Flow Graph

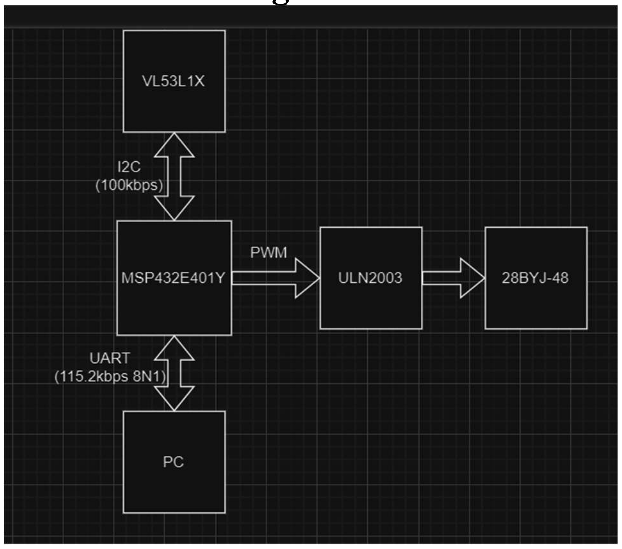


Figure 1: Component Block Diagram

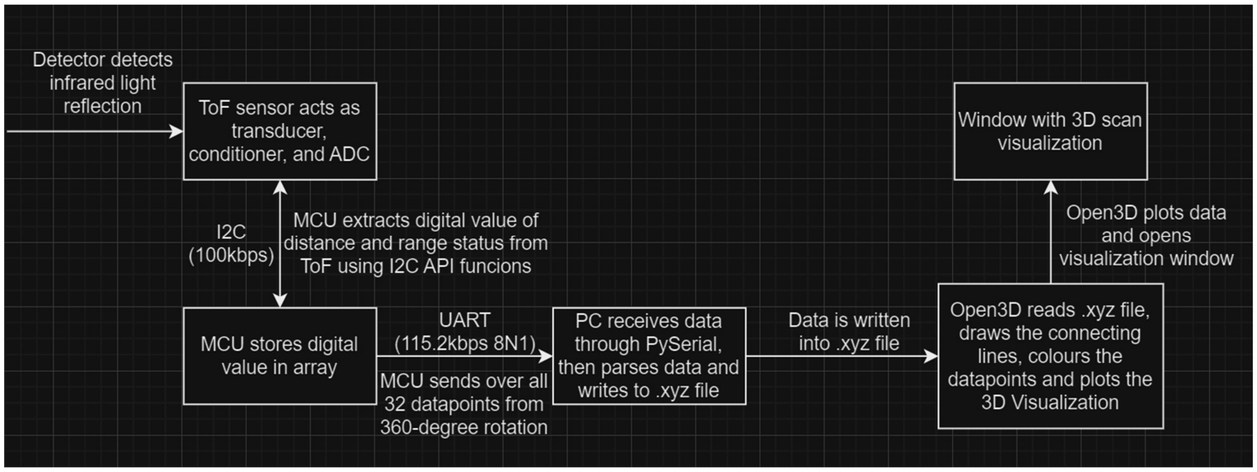

Detector detects infrared light reflection
ToF sensor acts as transducer, conditioner, and ADC
I2C (100kbps)
MCU extracts digital value of distance and range status from ToF using I2C API funcions
MCU stores digital value in array
UART (115.2kbps 8N1)
MCU sends over all 32 datapoints from 360-degree rotation
PC receives data through PySerial, then parses data and writes to .xyz file
Data is written into .xyz file
Open3D reads .xyz file, draws the connecting lines, colours the datapoints and plots the 3D Visualization
Open3D plots data and opens visualization window
Window with 3D scan visualization

Figure 2: Data Flow Graph from Light Detection to Visualization

Device Characteristics

Feature	Description
General	
UART Transmission Settings	115.2kbps 8N1
I2C Transmission Speed	100kbps
Datapoints per Scan	32
Angle Between Measurements	11.25°
MSP432E401Y Microcontroller	
Clock Speed	24MHz Bus Speed
Measurement Status LED	PF4 (D3)
UART Transmission Status LED	PF0 (D4)
Start Button	PJ1
Void/Stop and Finish Button	PJ0
VL53L1X Time of Flight (ToF) Sensor	
Vin	2.6 – 3.5V Operating Range
GND	GND
SDA	PB3
SCL	PB2
ULN 2003 Driver Board	
In1	PH0
In2	PH1
In3	PH2
In4	PH3
V+	5 – 12V Operating Range
V-	GND

Table 1: Device Characteristics

Detailed Description – Part A: Distance Measurement

The functionality of the VL53L1X Time-of-Flight (ToF) sensor hinges on its utilization of infrared light, emitted through its "emitter", to create a beam. This beam interacts with nearby objects and reflects back to the sensor's "detector". By precisely gauging the time between the emission and detection of this beam, the sensor can deduce the distance between itself and the object. Mathematically, this distance $D = \frac{photo \quad travel\, time}{2} * speed\ of\ light$ because the photons are travelling at the speed of light, and we divide the time by two as the light travelled there and back but we just want the distance there. Subsequent to this initial time-of-flight measurement, the sensor undergoes several processes including transduction, conditioning, and Analog-to-Digital Conversion to convert the analog distance measurement into a digital format. This digital data is then transmitted to the microcontroller via I2C serial communication [2].

However, before we can start this process of data collection and utilize the ToF sensor there is a sequence of steps that we must do to properly initialize the ToF sensor as seen in Figure 3 [2]. Steps 1 and 2 happen right after the code is flashed to the microcontroller and RESET is pressed, step 3 is not done as none of the optional functions are needed.

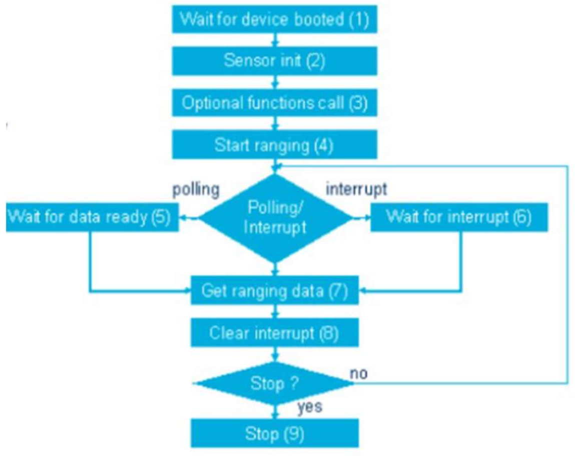


Figure 3: ToF Sensor Ranging Flow

Despite the sensor being active and poised for distance measurements now, the rest of the process doesn't commence immediately. Instead, the microcontroller is currently idle and continuously polls waiting for an input from the user through either of the 2 user buttons. While in this idle state a PJ1 press will make the 28BYJ-48 Stepper Motor commence a 360-degree clockwise rotation, where it will stop every 11.25 degrees to take a distance measurement, thus capturing 32 data points in total per rotation. A PJ1 press in any state except idle will do nothing.

Each time the ToF sensor is capturing a datapoint it is doing steps 4, 5, 7, 9 in that order from Figure 3 as it was implemented with the polling method. The data is then sent to the board via I2C communication where it is stored in an array of MeasurementPair structs until the full rotation is completed. While a distance measurement is being taken, the onboard LED D3 (PF4) will turn on, indicating that a distance measurement is currently being taken and/or processed.

If a PJ0 press occurs during the 360-degree clockwise rotation the measured data will be voided (meaning this rotations data is never sent to the PC to be included in the visualization data, although the individual points up until now were sent to the microcontroller), and the motor will perform a counter-clockwise rotation until it returns back to home (0-degree position), where the system will be back in idle state polling for user input. If no PJ0 press occurs throughout the rotation, then it is considered valid and the 32 datapoints will be sent to the PC via UART serial communication for processing and visualization. During the UART transmission user LED D4 (PF0) will turn on, indicating a UART data transmission from the MCU to PC.

The data transmitted from microcontroller to PC after each rotation is an array of 32 MeasurementPair structs. The MeasurementPair struct simply stores a measurements Distance as well as the RangeStatus, the only 2 variables needed for plotting as seen in Figure 4. These measurements were extracted from the ToFs API functions specifically steps 4, 5, and 7 from Figure 3.

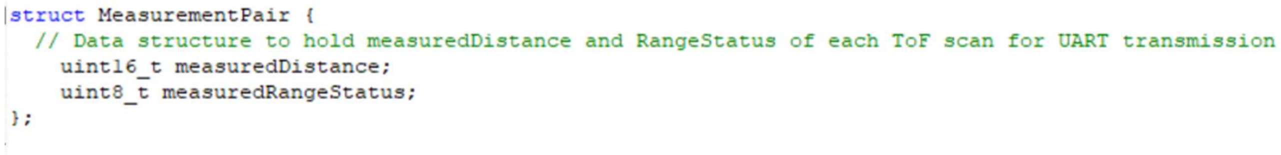
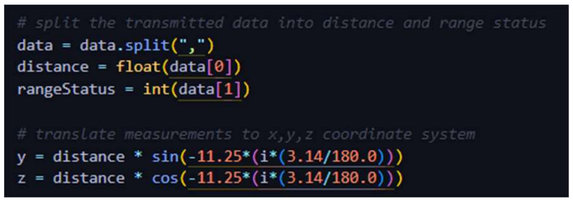

```
struct MeasurementPair {
  // Data structure to hold measuredDistance and RangeStatus of each ToF scan for UART transmission
    uint16_t measuredDistance;
    uint8_t measuredRangeStatus;
};
```

Figure 4: MeasurementPair Struct

Detailed Description – Part B: Visualization

The 32 datapoints of a valid rotation are sent to the PC via UART stable serial communication implemented with PySerial. Each datapoint is read and unpacked from byte to string. then parsed, separating the Distance and RangeStatus back into their own float and int variables from the single transmission. The Distance measurement is used to plot the points by converting them into zy-plane cartesian coordinates using simple trigonometry as seen in Figure 5, while the x distance is incremented every 32 points by a value set at the top of the script. These 3 variables are then written into a .xyz file to be read and visualized by Open3D later. The RangeStatus measurement is used to fill the colours array, where colours[i] will hold the visualization colour for that datapoint to represent the quality of that point. A 0 RangeStatus (Good scan) will be green, while 1 or 2 (Small error) will be yellow, and 4 or 7 (Large error) will red.



```
# split the transmitted data into distance and range status
data = data.split(",")
distance = float(data[0])
rangeStatus = int(data[1])

# translate measurements to x,y,z coordinate system
y = distance * sin(-11.25*(i*(3.14/180.0)))
z = distance * cos(-11.25*(i*(3.14/180.0)))
```

Figure 5: Parsing and Distance to Cartesian Coordinate Conversion

To finish the data collection process and initiate the visualization process, press PJ0 while in the idle state. This will send "Measurements done\n" from the microcontroller to the PC. When the PC receives this data and unpacks it, it will break out of the data collection loop, close the serial port, close the .xyz file that was being written to, and commence the visualization process.

The visualization process utilizes the Python library Open3D to read the coordinates in the .xyz file and turn it into a point cloud as well as read through colours array and assign each corresponding datapoint that colour.

The script then goes through each datapoint and connects it to the point after it, as well as the next iteration of that datapoint (i.e. datapoint 1 from rotation 1 will be connected to datapoint 2 from rotation 1, and datapoint 1 from rotation 2). Again, Open3D is used to add these connecting lines to the visualization.

Finally using Open3D a window is opened with both the coloured data points and connecting lines plotted to create an accurate 3D representation of the area scanned.

Application Overview

Lidar scanning technology harnesses laser light to gauge distances and craft intricate 3D representations of landscapes and objects, rendering it a potent tool in remote sensing with multifaceted applications. Among its primary functions is its pivotal role in mapping and surveying, wherein it excels at generating precise digital elevation models crucial for tasks such as urban planning, flood prediction, and resource allocation. Moreover, lidar's integration into the realm of autonomous vehicles is burgeoning, furnishing real-time environmental data vital for navigating obstacles and ensuring safe traversal. Its versatility extends further into domains like archaeology and forestry. In the realm of archaeology, lidar's prowess lies in its ability to unveil intricate maps of historical sites, uncovering hidden relics and structures concealed beneath the surface. Similarly, within forestry, lidar aids in crafting meticulous and accurate representations of wooded areas, facilitating resource monitoring, hazard identification—including wildfires and landslides—and overall forest management.

Instructions

This instruction and setup process assumes that the user has all of the necessary software downloaded and configured for their specific system. This includes the installation of Kiel Development Environment, an IDE of their choice to run the Python file, as well as a Python version compatible with Open3D.

1) Plug the microcontroller into a computer and determine the COM port by pressing the windows key on the keyboard, then type in 'Device Manager' and hit enter. Once in the device manager menu, scroll down to 'Ports (COM & LPT)' and then expand the arrow. In the drop-down menu, note down the value of # in the statement 'XDS110 Class Application/User UART (COM #)'.

2) Open the Python script and change line 30 "s = serial.Serial('COMX')" so that X corresponds to the index of the COM port that you determined in step 1. Now, serial communication should be all set up.

3) Configure the circuit by following the circuit schematic shown as Figure 12 on page 12.

4) Open the Kiel project. Select 'Options for Target' then 'C/C++' tab, then ensure your optimization is '-O0'. Then select the 'Debug' tab, from the drop-down menu beside the 'Settings' button and select 'CMSIS-DAP Debugger', then click the 'Settings' button and ensure the dropdown menu in the top left says 'XDS110 with CMSIS-DAP'. Then click 'OK' on both menus.

5) Click 'Translate' → 'Build' → 'Load' in the top left of the IDE.

6) Flash the project onto the board by clicking the RESET button.

7) In the Python script change line 25 "X_INCREMENT = 100" to equal the x-axis distance in mm between scans. (X_INCREMENT = 300, in my provided scans in Figures 6 and 7)

8) Ensure you are using a version of Python compatible with Open3D (I used 3.11.3)

9) Ensure you have all the dependencies installed. If you do not have any of the following 4: serial, math, numpy, open3d; Open a terminal and type 'pip install X', where X is your missing dependency.

10) Run the Python script.

11) Hit enter to confirm that you are ready to receive data from the microcontroller.

12) Press PJ1 to start a 360-degree rotation scan.

13) (Optional step) IF you wish to void and redo a scan, press PJ0 *while the scan is still occurring* to reset the motor back to its starting position and void the scan.

14) Move forward X_INCREMENT millimeters (If X_INCREMENT=100, go forward 100mm)

15) Repeat steps 12-14 until you reach the number of scans you want.

16) Press PJ0 while in idle to finish data collection and visualize the collected data.

Expected Output

The expected output of the device can be seen below in Figures 6 and 7. To obtain these Figures an X_INCREMENT of 300mm, approximately one foot, was used. My assigned scan location was location E in ETB as seen in Figure 8. Figures 9 through 11 show the scan location in real life. Comparing Figure 6 to Figure 8 you can see how the scan captures the shape of the hallway including finer details such as the curvature on the left side of the hallway, and the indentation from the smaller hallway in the bottom right. Figure 7 shows that it correctly mapped a level floor and roof that is seen in Figure 10.

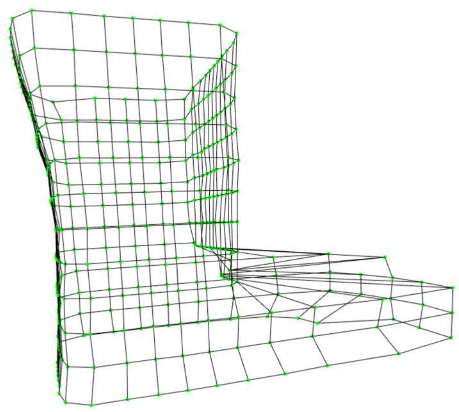


Figure 6: 3D Visualization - Birds Eye View

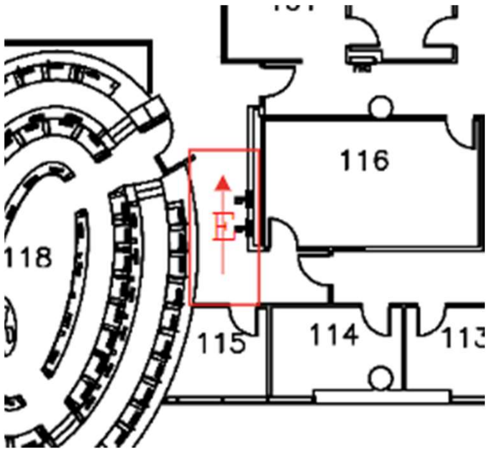


Figure 8: Location E - Birds Eye View

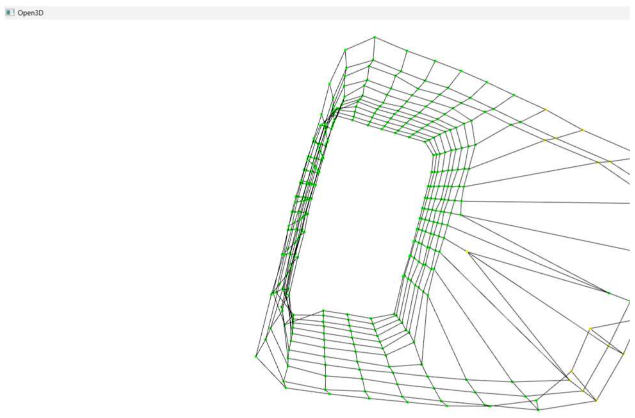


Figure 7: 3D Visualization - Front View

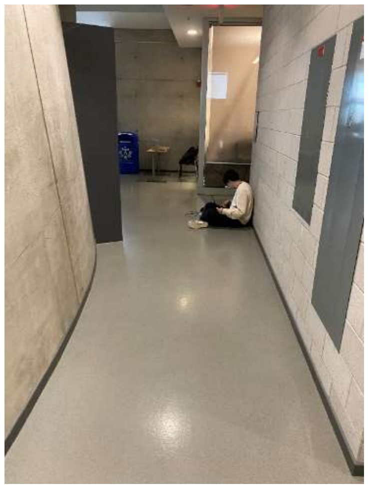


Figure 9: Location E – Front View

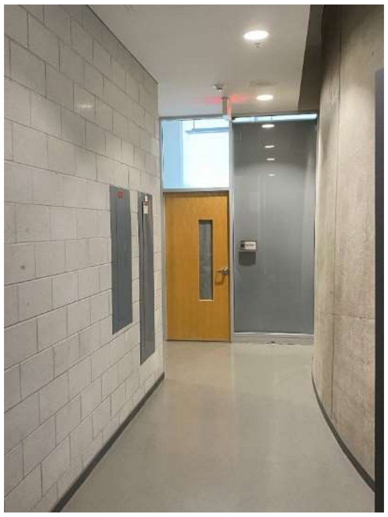


Figure 10: Location E – Back View

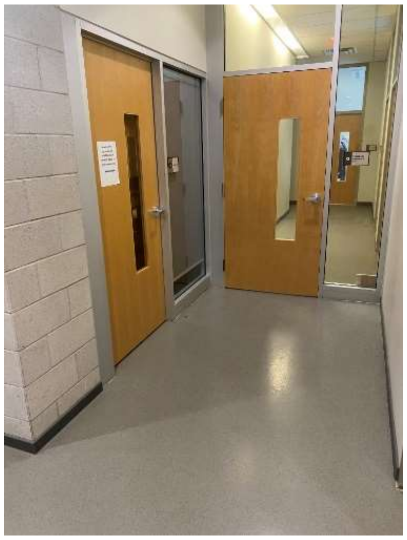


Figure 11: Location E - View of Side Hallway

Limitations

The microcontroller features an Arm Cortex-M4F which has an IEEE 754-compliant single-precision (32-bit) Floating-Point Unit (FPU) [1]. This means that the microcontroller fully supports single-precision add, subtract, multiply, divide, multiply and accumulate, and square-root operations. The FPU also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions [1]. Despite the fact that this makes the microcontroller capable of handing trigonometric operations and calculations to a 32-bit precision there is no reason to perform the calculations on the microcontroller as we still risk losing accuracy past the 32nd bit. Thus, all expensive calculations such as the conversion from distance to cartesian coordinates through trigonometric functions were offloaded to the PC and performed in the Python script ensuring 64-bit precision and faster calculation speed.

The maximum quantization error is the same as the resolution of the system or 1 LSB which we know $\Delta = \frac{Max\ Output}{2^n}$ where the Max Output in our case is 4m, the max ranging distance output of the ToF sensor, and n is 16, the number of bits that the ToF sensor returns to our microcontroller after converting the signal from the infrared light emission to a digital value. Therefore, $\Delta = \frac{4000mm}{2^{16}} = 0.06mm$ is our resolution and the maximum quantization error.

The maximum speed that my PC can implement is 128kbps. This was verified by going into device manager, then ports, then clicking on the XDS110 Class Application/User UART port, then going to the port settings tab, then opening up the bits per second drop down menu and observing that the largest option is 128kbps.

The communication method used between the microcontroller and the ToF sensor was Inter-Integrated Circuit (I2C) with the microcontroller as the leader setting the speed to 100kbps, and the ToF sensor as the follower. However, the microcontrollers I2C protocol supports up to 3.33Mbps (High-speed mode) [1] and the ToF sensors I2C protocol supports up to 400kbps (Fast mode) [2].

The primary limitation on system speed is the stepper motor due to the necessary delay between stepper motor phases. Between each of the 4 phases there must be a 2ms delay or else the motor will just vibrate rather than rotate. This was verified by individually commenting out portions of the code and running the system without it. Running the project without the UART transmissions saves less than 100ms per rotation. Running the project without the ToF sensor (commenting out all API calls) saves less than 200ms per rotation. While a single rotation with the stepper motor $= \frac{4phases}{step} * 512steps * \frac{2ms}{phase} = 1.024\ seconds$. Therefore, the stepper motor is the biggest limitation on system speed as it causes a minimum delay of 1.024 seconds per rotation which is much larger than the delay caused by other parts of the system.

My assigned system Bus Speed from Table 1 is 24MHz as the LSD in my student number, 400409624, is 4. To configure this bus speed, I changed line 29 of PLL.h to #define PSYSDIV 19, making this variable equal to 19 instead of 3. Thus the $bus\ speed = \frac{480MHz}{(PSYSDIV+1)} = \frac{480MHz}{(19+1)} = 24MHz$. However, the SysTick function depends directly on the bus speed and thus I had to change that as well. To re-configure my SysTick_Wait10ms function I simply changed the value in SysTick_Wait(Value) from 1.2M to 240k. I did this because previously $10ms * 120MHz = 1.2M$, and now $10ms * 24MHz = 240k$. To re-configure my SysTick_Wait1ms function I did the same. Changing the value in SysTick_Wait(Value) from 120k to 24k. This was again because previously $1ms * 120MHz = 120k$, and now $10ms * 24MHz = 24k$.

Circuit Schematic

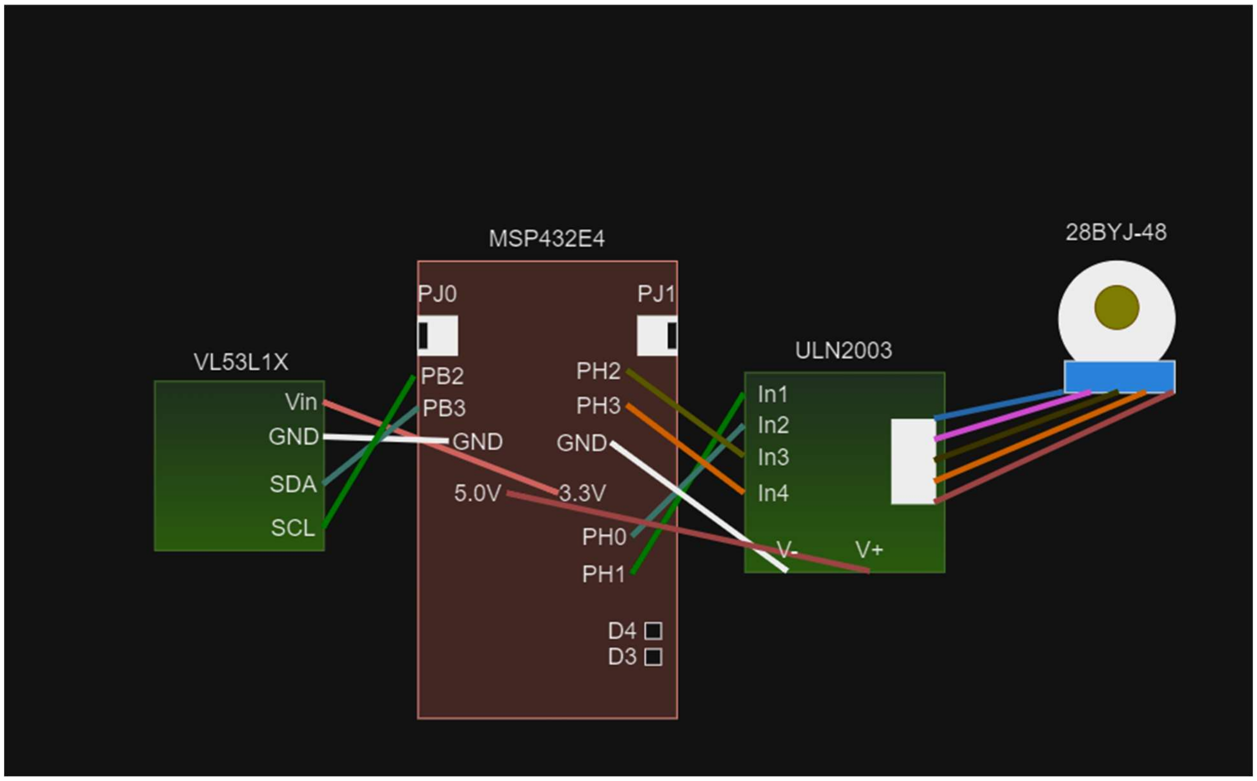


Figure 12: Circuit Schematic

Programming Logic Flowcharts

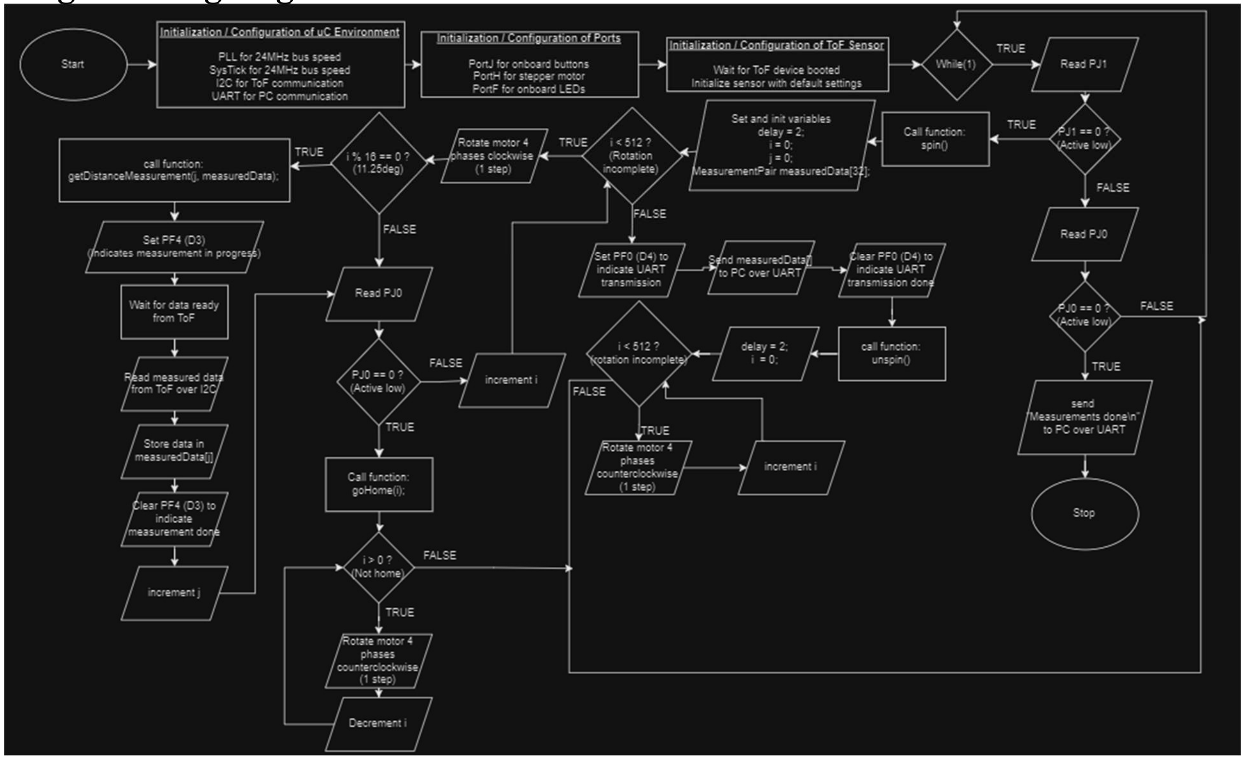

Start
Initialization / Configuration of uC Environment
PLL for 24MHz bus speed
SysTick for 24MHz bus speed
I2C for ToF communication
UART for PC communication
Initialization / Configuration of Ports
PortJ for onboard buttons
PortH for stepper motor
PortF for onboard LEDs
Initialization / Configuration of ToF Sensor
Wait for ToF device booted
Initialize sensor with default settings
While(1)
TRUE
Read PJ1
PJ1 == 0 ? (Active low)
TRUE
Call function: spin()
Set and init variables delay = 2; i = 0; j = 0; MeasurementPair measuredData[32];
i < 512 ? (Rotation incomplete)
TRUE
Rotate motor 4 phases clockwise (1 step)
i % 16 == 0 ? (11.25deg)
TRUE
call function: getDistanceMeasurement(j, measuredData);
Set PF4 (D3) (Indicates measurement in progress)
Wait for data ready from ToF
Read measured data from ToF over I2C
Store data in measuredData[j]
Clear PF4 (D3) to indicate measurement done
increment j
FALSE
Read PJ0
PJ0 == 0 ? (Active low)
FALSE
increment i
TRUE
Call function: goHome(i);
i > 0 ? (Not home)
FALSE
TRUE
Rotate motor 4 phases counterclockwise (1 step)
Decrement i
FALSE
Set PF0 (D4) to indicate UART transmission
Send measuredData[] to PC over UART
Clear PF0 (D4) to indicate UART transmission done
call function: unspin()
delay = 2; i = 0;
i < 512 ? (rotation incomplete)
FALSE
TRUE
Rotate motor 4 phases counterclockwise (1 step)
increment i
FALSE
Read PJ0
PJ0 == 0 ? (Active low)
FALSE
TRUE
send "Measurements done\n" to PC over UART
Stop

Figure 13: Programming Logic Flowchart for FinalProject.c

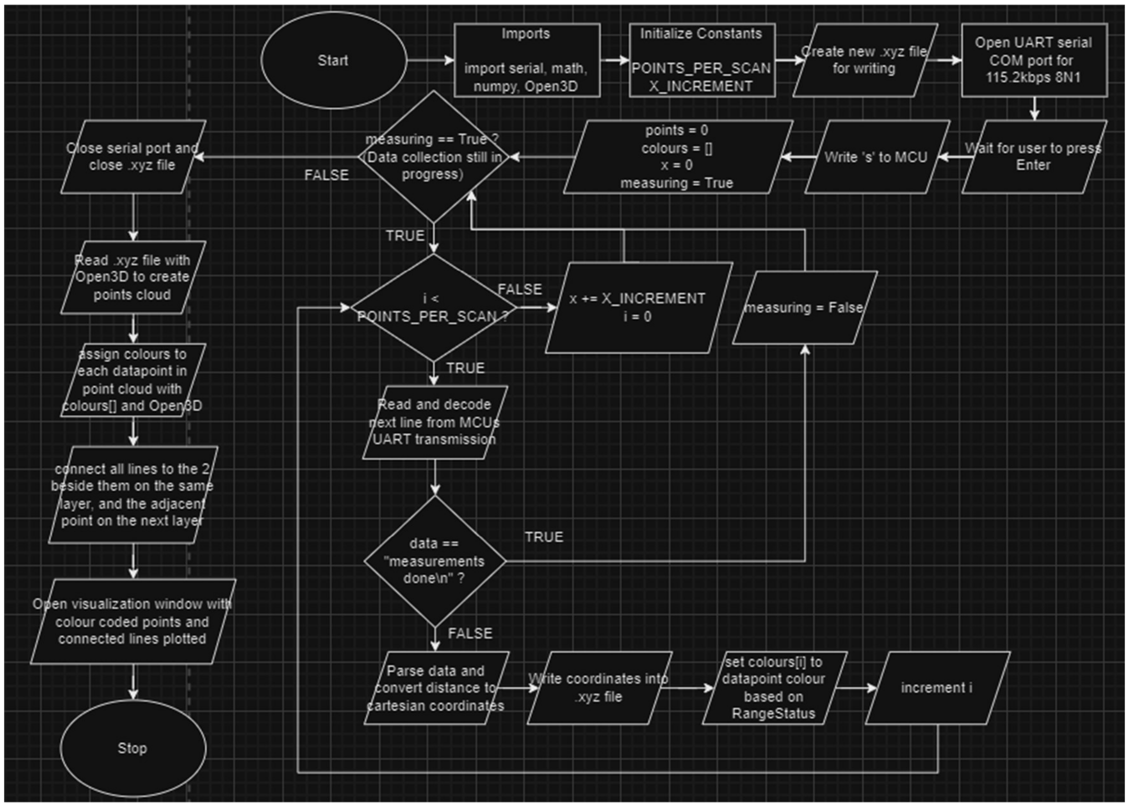

Start
Imports
import serial, math, numpy, Open3D
Initialize Constants
POINTS_PER_SCAN
X_INCREMENT
Create new .xyz file for writing
Open UART serial COM port for 115.2kbps 8N1
Wait for user to press Enter
Write 's' to MCU
points = 0
colours = []
x = 0
measuring = True
measuring == True ? (Data collection still in progress)
FALSE
Close serial port and close .xyz file
TRUE
i < POINTS_PER_SCAN ?
FALSE
x += X_INCREMENT
i = 0
measuring = False
TRUE
Read and decode next line from MCUs UART transmission
data == "measurements done\n" ?
TRUE
FALSE
Parse data and convert distance to cartesian coordinates
Write coordinates into .xyz file
set colours[i] to datapoint colour based on RangeStatus
increment i
Read .xyz file with Open3D to create points cloud
assign colours to each datapoint in point cloud with colours[] and Open3D
connect all lines to the 2 beside them on the same layer, and the adjacent point on the next layer
Open visualization window with colour coded points and connected lines plotted
Stop

Figure 14: Programming Logic Flowchart for FinalProject.py

References

[1] “MSP432E401Y SimpleLinkTM Ethernet Microcontroller Data Sheet,” 2017. Accessed: Apr. 14, 2024. [Online]. Available: MSP432E401Y SimpleLink™ Ethernet Microcontroller Data Sheet

[2] “This is information on a product in full production. VL53L1X A new generation, long distance ranging Time-of-Flight sensor based on ST’s FlightSenseTM technology Datasheet - production data Features,” 2018. Accessed: Apr. 14, 2024. [Online] Available: VL53L1X