COMPENG 2DX3

Final Project Report

April 17, 2023

Ivan Lange – 400303704

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 our own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario.

Table of Contents

[1 Device Overview 3](#_Toc132469821)

[1.1. Features 3](#_Toc132469822)

[1.2. General Description 3](#_Toc132469823)

[1.3. Block Diagram 4](#_Toc132469824)

[2 Device Characteristics Table 5](#_Toc132469825)

[3 Detailed Description 6](#_Toc132469826)

[3.1. Distance Measurement 6](#_Toc132469827)

[3.2. Visualization 7](#_Toc132469828)

[4 Application Example, Instructions, and Expected Output 9](#_Toc132469829)

# Device Overview

# Features

|  |  |
| --- | --- |
| * All-In-One LIDAR System   + Suitable for Indoor Use   + Close Range LIDAR Scanning and Visualization Package   + Connectivity to PC for Visualization of Data * Texas Instruments MSP432E401Y SimpleLink™ Ethernet Microcontroller   + Arm Cortex-M4F Processor Core   + 12-MHz Bus Speed   + 1024KB of Flash Memory, 256KB of SRAM, 6KB EEPROM   + Input Voltage: 4.75 VDC to 5.25 VDC Via XDS-110 USB Micro-B   + Status LEDs for Data Scanning Ready and Transmission of Data to PC   + Programmed in C language (optionally Assembly or C++) | * VL53L1X Time-Of-Flight Sensor   + Up to 400 cm Distance Measurement   + Up to 50 Hz Ranging Frequency   + I²C Interface (up to 400 KHz)   + Input Voltage: 2.6 VDC to 5.5 VDC * MOT-28BYJ48 Stepper Motor with ULN2003 Driver   + Input Voltage: 5-12 VDC   + 512 Steps Per Rotation * Visualization   + Python 3.8 (64-bit) IDLE   + Open3D package for model creation * Communication   + I2C Communication Between Microcontroller and ToF   + UART Communication Between Microcontroller and PC, with 115,200 bps baud rate |

# General Description

This all-in-one embedded spatial measurement system scans indoor spaces and creates 3D reconstructions of a particular area. It records 360-degree distance measurements in the YZ plane and is manually moved in the X-plane for each new 360-degree measurement. The number of samples taken per 360-degree measurement depends on the step-size of the motor used and affects the resolution of the final image created. This data is then sent to a PC for visualization using Open3D on Python.

The hardware of the system is comprised of a TI MSP432E401Y microcontroller for control of the embedded system, a VL53L1X Time of Flight sensor for distance data, a MOT-28BYJ48 stepper motor with the ULN2003 driver for 360-degree scanning, two user pushbuttons, and two user LEDs. The microcontroller is programmed in C-language for the control-loop and communication between the ToF and PC. Python is used on the PC for visualization with the Open3D package for mesh-creation using transmitted data.

The VL53L1X package calculates the distance measurement from the time of flight of a light pulse. The sensor angle is variable and is adjusted by rotating the stepper with the desired step size. The data is acquired via I2C and stored locally on the embedded platform; this is repeated for the desired number of measurements in the X-plane. When done, the data can be sent to the PC via UART to be visualized in Open3D and Python.

# Block Diagram

Timeline

Description automatically generated

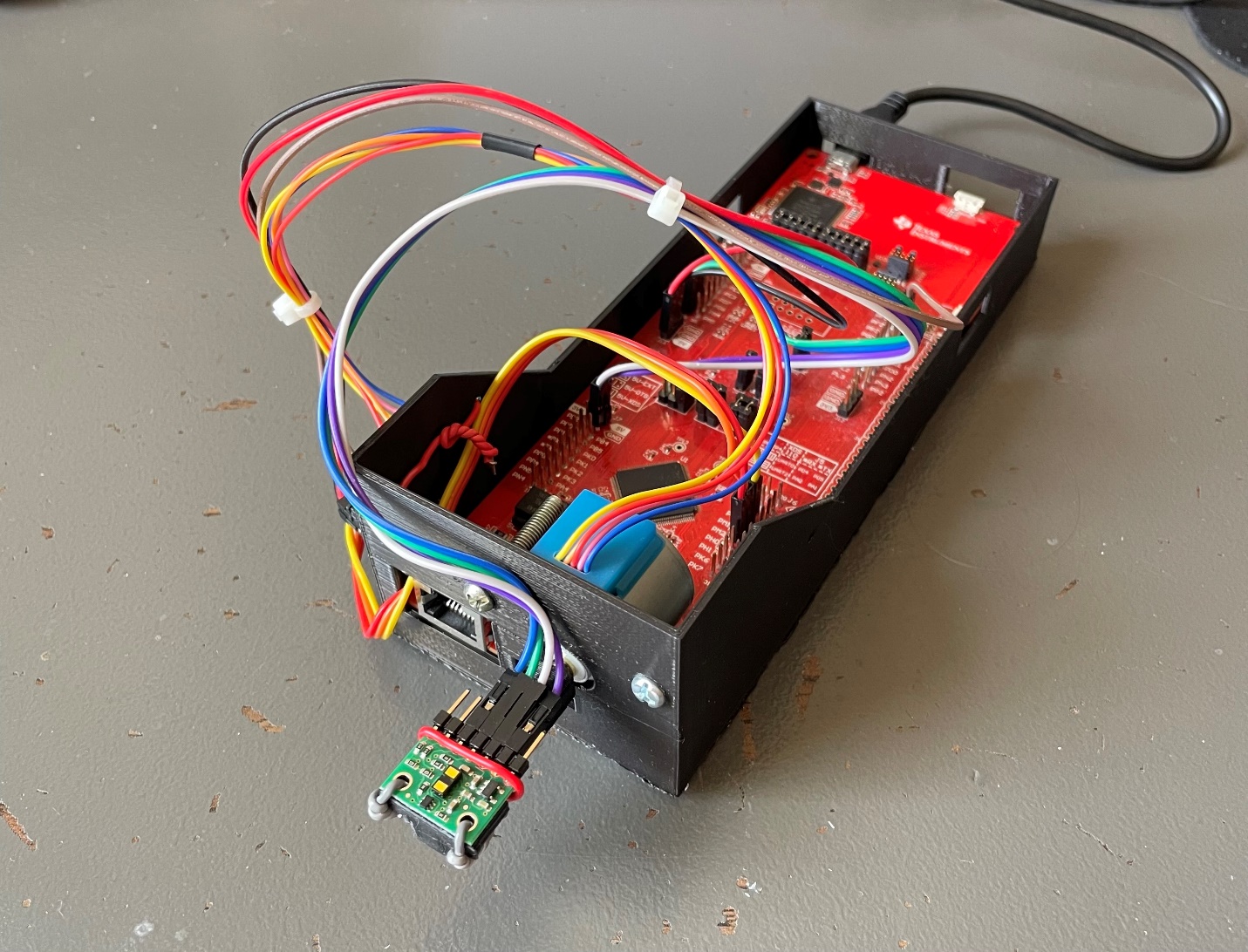
Figure 1: Data Flow Graph of System

Figure 2: Image of Scanner

# Device Characteristics Table

|  |  |
| --- | --- |
| General System Setup | |
| Feature | Description |
| Bus Speed | 12 MHz |
| UART Baud Rate | 115,200 bps |
| Python Version | 3.8 (64-bit) |
| COM Port | COM5 (Device Dependent) |
| Microcontroller Power | USB to PC (5 VDC) |
| VL53L1X Setup | |
| VL53L1X | Microcontroller |
| VDD | - |
| VIN | 5V |
| GND | GND |
| SDA | PB3 |
| SCL | PB2 |
| XSHUT | - |
| GPIO1 | - |
| LED Configuration | |
| User LED | Assignment |
| PN0 | Measurement Status |
| PN1 | Data Transmission Status |
| Pushbutton Configuration | |
| User LED | Assignment |
| PJ0 | Start Data Acquisition |
| PJ1 | Initiate Data Transfer to PC |
| ULN2003 Setup | |
| ULN2003 | Microcontroller |
| + | 5V |
| - | GND |
| IN1 | PM0 |
| IN2 | PM1 |
| IN3 | PM2 |
| IN4 | PM3 |

Table 1: Device Characteristics

# Detailed Description

# Distance Measurement

The embedded spatial measurement system is equipped with a VL53L1X LIDAR sensor for acquiring distance measurements. It comes in a fully integrated package, including the transducer stage, conditioning, and analog-to-digital conversion. This sensor relies on the time of flight of 940nm light pulses to determine the relative distance from the sensor; it considers the time taken for the light to travel and return from the object as well as the speed of light to determine the approximate distance. The data is then communicated to the microcontroller through I­2C, utilizing the UM2510 API for the VL53L1X by STMicroelectronics. A variety of functions are available to initialize the device to a particular specification as well as acquire distance and confidence data regarding the corresponding measurement. The confidence of the measurement is measured with the range status variable; it returns a value of 0 with no error, 1 of 2 with a warning, and 4 or 7 if there is an error. Verification that the range status has no errors is done within the software to ensure an accurate measurement is acquired at each scanning stage. Once this is verified, the distance measurement is recorded in millimeters and saved in its corresponding position in the two-dimensional array for transmission later.

The ToF sensor is mounted to the MOT-28BYJ48 motor through a custom-built mount. The stepper motor is then used to precisely change the angle of the ToF at each measurement. Beginning by facing 90 degrees to the horizontal, the ToF determines the distance, and stores it in the array, *transmissionData*. The resolution of the final image is proportional to the number of measurements taken per displacement in the X-plane, or inversely proportional to the angle per step of the motor. Therefore, to achieve a higher resolution, the stepper motor must step at a smaller angle for each measurement of the LIDAR sensor. In order to achieve this, a macro is defined in the *scanner.h* file called STEP\_SIZE. The value assigned determines the number of times the stepper motor steps between each measurement. The user should refer to the lookup table shown below when setting the STEP\_SIZE for their application.

|  |  |  |
| --- | --- | --- |
| *Number of Samples* | *Degree between scans* | *Value of STEP\_SIZE* |
| *128* | *2.8125* | *16* |
| *64* | *5.625* | *32* |
| *32* | *11.25* | *64* |
| *16* | *22.5* | *128* |
| *8* | *45* | *256* |

Table 2: Assigned Step Size for Desired Resolution per Scan in X-plane

An interrupt-driven solution was used for the device firmware. The system is first configured through the initialization of the system clock, UART, I2C, LEDs, stepper motor, LIDAR sensor, and pushbuttons. Interrupts are then configured for pushbuttons PJ0 and PJ1. Additionally, two global structs are declared for the system – *motor* of type *TsStepParameters* and *scanner* of type *TsScannerParameters.* The first struct contains information regarding the state of the motor, while the second struct stores the state of the scanner. The definitions for these structs are shown below. Note that enumerations are defined to define the state of each struct parameter.

volatile TsStepParameters motor **=** **{**

**.**angle **=** STEP\_SIZE **,**

**.**dir **=** STEP\_CW**,**

**.**currentStep **=** 0**,**

**.**op **=** STEP\_NORMAL**,**

**};**

volatile TsScannerParameters scanner **=** **{**

**.**status **=** SC\_COMPLETE**,**

**.**state **=** SC\_SCAN\_MODE

**};**

The system is now in a resting state until further action from the user. User LED D2 will be on, signaling that the system is ready to begin scanning. When the user wants to begin acquiring distance data, button PJ0 is depressed and a GPIO interrupt is triggered, flipping the status of the *scanner.status* field. The main program loop then senses the change, and runs the *scanYZ()* function, which scans the Y-Z plane with the assigned STEP\_SIZE using the stepper and ToF sensor. The distance data is acquired through the use of the API and I2C communication from the microcontroller to the sensor. Each distance measurement is stored in the global array *transmissionData.* The size of this two-dimensional array is determined by a macro called MAX\_MEASUREMENTS (number of rows) and the STEP\_SIZE (number of columns). The size of this array is determined at compile time, and adapts to the user defined constants. MAX\_MEASUREMENTS is defaulted to 20, however, can be changed by the user in the *scanner.h* file. This constant determines the maximum number of scans taken in the X-plane, but the user can choose to take any number of scans up to this value. When the motor has rotated a full 360 degrees, it returns home by spinning 360 degrees in the opposite direction, and the program returns to the previous state of rest. To acquire more scans at a new location in the X-plane, the user should physically increment the location of the system in the X-direction (see 3.2 Visualization for the actual distance) without moving the relative location in the Z-plane from previous scans (see the relative coordinate system in Section 4). When ready in the new location, the user should repeat this process by depressing PJ0 again.

# Visualization

After the distance data is acquired through the process outlined above, it must be transmitted to the user’s PC for visualization. The visualization code was written in Python 3.8 (64-bit), in the file *main.py*. For testing, the program was run on a Dell XPS 15, with an Intel Core i7, 16GB of RAM, and a 512GB SSD. The Python file should then be run using IDLE.

When the desired number of scans is achieved, the system will be in a state of rest. To transmit the data to the PC, button PJ1 should be depressed. An interrupt is triggered, and user LED D1 will turn on, signaling that the system is currently in a state of transmitting to the user. Also, the state of *scanner.state* field will be flipped, and the main program will loop will sense the change and call the *transmitDistanceSerial()* function. This function then waits to confirm that the user is ready for the data. On the PC end, with the shell running, the user should then click the enter key, signaling to the microcontroller that the PC is ready for transmission. The *transmitDistanceSerial()* method then transmits the number of X-plane measurements (measurementNum), the number of samples per X-plane measurement (NUM\_SAMPLES), and each 16 bit distance measurement from all the samples (determined by measurementNum\*NUM\_SAMPLES) via UART. Note that each distance measurement is two bytes large, and UART transmits one byte at a time. The 8 least significant bits are transmitted first, followed by the 8 most significant bits, for each distance measurement. The system is then reset and returns to its original state, ready for new measurements.

Using PySerial, Open3D, Struct, Math, and NumPy packages, the Python script then receives these UART frames and stores the necessary data. It stores the number of measurements and number of samples per measurement. Using these variables, it then loops through the necessary number of times to acquire every sample, reconstructing the two bytes into one integer value each time. At this stage, the distance data now needs to be converted into a coordinate for visualization after. Using the number of samples per measurement, the script finds the degree between scans that was set in the firmware. Assuming a starting angle of 90 degrees to the horizontal, the script then uses basic trigonometry to convert the distance (hypotenuse) and angle to an (x,y,z) coordinate, as shown below. Note that for each distance measurement sample, the angle is decreased by the degree between scans that was calculated earlier. Each set of measurements in the X-plane is assumed to have a constant X-distance between them, and is set to 600mm by default, but can be changed in line 56 of the script. Therefore, for each X-plane distance, x, angle, θ, and distance, d, the coordinate is found and saved on a new line of a .xyz file named test.xyz.

With the file created and updated with the coordinates, the Open3D package is then used to visualize the data. A point cloud is first created to show the relative position of each point found by the system. After this is closed, a line visualization is shown, connecting each point in the frame to the points adjacent to it. A loop is created for each set of measurements in the X-plane, connecting each point to the two points adjacent to it. Lines are also created between adjacent points in different X-locations by looping through each point and connecting the adjacent point in the following frame. These are outputted to the PC for the user to study.

# Application Example, Instructions, and Expected Output

# Limitations

1. The microprocessor used has limited floating point capabilities. Its floating-point unit (FPU) supports single-precision floating point values and operations. A single-precision floating point value occupies 32-bits in memory, with 23 of those bits being used for the fraction component of the stored number. However, the potential error caused by a floating point approximation in the microprocessor is not an issue that is present in the board. The firmware for the device only declares integer variables of 8 or 16 bits in size for the data flow. Since the floating point values and trigonometric functions are used only in the Python script, the source of the error would come from the visualization software on the PC. Therefore, since the x, y and z coordinates are stored in floating-point values, there is an approximation being made, and therefore a source of error. Additionally, when calculating the coordinates from the integer distance, an approximation of pi (to 15 decimal places) was used from the math package, and the functions for cosine and sine were approximations as well. This causes some potential inaccuracies in the final image recreation.
2. Another source of error is the quantization error for the data read by the ToF module. The maximum quantization error of the ToF readings is 0.0610 mm, or 5.035e-5 % of the full scale voltage of 3.3V.
3. The maximum standard baud rate that can be implemented by the PC is 128,000 bps. This speed was determined by looking at the properties of the COM port in the device manager, and finding the maximum baud rate that is supported. This was determined to be standard, since 128,000 bps was the largest baud rate in the drop-down selection. However, a baud rate of 115,200 bps was implemented for UART communication in the actual project. This was confirmed through RealTerm, by selecting that baud rate and confirming communication was working between it and the microcontroller, which also had UART configured at the same rate.
4. I2C was used to communicate between the ToF and the microcontroller at a clock speed of 100 kbps.
5. Looking at the entire system, the element that is the primary limitation on system speed is acquiring the distance data from the ToF and stepper motor setup. The ToF sensor has a programmable timing budget that can be set from 20 ms to 1000 ms. However, for long distance mode, the budget is 140 ms. This means the ranging duration for each measurement is 140ms. Additionally, the stepper motor needs time to rotate to the next position. These timings combined surpass the timing required for serial communication by a large factor; therefore, they have the largest effect on the speed of the system. This was tested by looking at the quantity of data being transmitted over serial, and looking at how long it would require to transmit it. Due to high data transmission speeds, it is much faster than the reading of distances.

# Graphical user interface Description automatically generatedCircuit Schematic

# Programming Logic Flowchart