Theme Report 3 - Act

COMPENG 2DX3
Dr. Yaser Haddara

April 6th, 2025
Edison He (400449140)

Theme

The theme of this report revolves around the ability of a system to act autonomously based on the data it receives. In Lab 8 and deliverable 2, the microcontroller gathers inputs from various sensors and performs specific actions using outputs. The ability to act is critical in numerous fields such as robotics, automation, and control systems. In industrial automation, machines rely on sensors to detect changes in the environment. With this information, they make onboard decisions and adjust their actions. This includes toggling motors, moving a robotic arm, or triggering flags. Lab 8 and deliverable 2 demonstrate how sensor data is translated into physical actions, enabling the device to respond appropriately to its surroundings. This capacity to act in response to data is foundational in systems that require immediate and accurate responses to optimize performance and ensure operational efficiency.

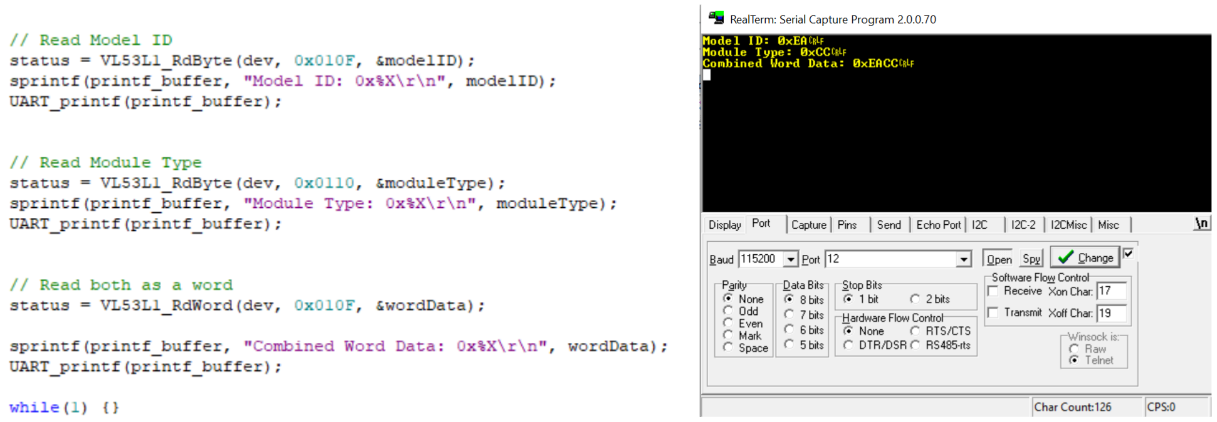
Background

For autonomous systems to operate effectively, they must be able to act on the data they collect. In autonomous technology like self-driving cars, the ability to make decisions based on the environment is crucial for safe and efficient operation. These systems rely on a range of sensors, cameras, and radar to continuously monitor their surroundings, collecting information about traffic, road conditions, pedestrians, and obstacles. With this data, the system can make real-time decisions that ensure safety and versatility in all driving conditions. For example, a Tesla uses its onboard sensors to gather information, allowing it to determine its position on the road, adjust speed, and navigate around obstacles autonomously, all without human intervention. This continuous data stream and immediate action is essential to the functionality of autonomous systems, ensuring they can respond appropriately to dynamic environments.

In Labs 8 and deliverable 2, the focus is on event-based programming, where the microcontroller responds to event triggers, building on the reasoning and acting concepts from Lab 4 and Lab 6. In Lab 8, the system processes GPIO interrupts triggered by button presses. When a button is pressed, the microcontroller transmits data to the Time-of-Flight sensor via I2C. The interrupt triggers an interrupt service routine in which a specific action is executed based on the button pressed. Deliverable 8 expands on this concept by sending data using UART to the pc. These labs demonstrate how embedded systems combine observe, reason, and act by collecting inputs, decision making based on the input, and then executing actions accordingly in real-time. These actions come in forms of LEDs or sending data over communication interfaces.

Lab 7

Lab 7 included 2 milestones which used the theme of act. In milestone 1, the microprocessor on startup sends a request to the TOF sensor using I2C. When the TOF sends its model information back to the microprocessor, the microprocessor instantly sends this information to the PC using UART. The communication between the TOF is done with SDA and port PB2 of the Micro. Figures 1 and 2 below show the code, connections between the microprocessor and TOF sensor, and RealTerm terminal.



```
// Read Model ID
status = VL53L1_RdByte(dev, 0x010F, &modelID);
sprintf(printf_buffer, "Model ID: 0x%X\r\n", modelID);
UART_printf(printf_buffer);


// Read Module Type
status = VL53L1_RdByte(dev, 0x0110, &moduleType);
sprintf(printf_buffer, "Module Type: 0x%X\r\n", moduleType);
UART_printf(printf_buffer);


// Read both as a word
status = VL53L1_RdWord(dev, 0x010F, &wordData);

sprintf(printf_buffer, "Combined Word Data: 0x%X\r\n", wordData);
UART_printf(printf_buffer);

while(1) {}
```

RealTerm: Serial Capture Program 2.0.0.70

```
Model ID: 0xEA
Module Type: 0xCC
Combined Word Data: 0xEACC
```

Baud 115200 Port 12

Parity: [✓ None] Odd Even Mark Space

Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits

Stop Bits: [✓ 1 bit] 2 bits

Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts

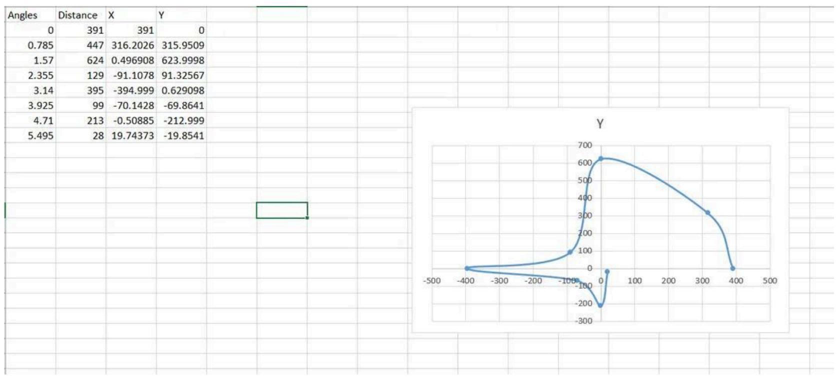
Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19

Winsock is: Raw [✓ Telnet]

Char Count:126 CPS:0

Figure 1: Keil Code and RealTerm Terminal

Milestone 2 further applied this concept by introducing a stepper motor on port H0 to H3, which is attached to the TOF sensor. With this setup, the TOF sensor is able to scan multiple points in one 360 degree scan. In this setup, the TOF sensor sends a point to the microprocessor, and in turn, the microprocessor rotates the motor. This is repeated 7 more times. Once all the data is scanned, a graph was manually made on excel. Given a distance from the time of flight sensor, and angle of the motor, the polar coordinates were converted into rectangular coordinates and the result is seen in the figure below.



Angles	Distance	X	Y
0	391	391	0
0.785	447	316.2026	315.9509
1.57	624	0.496908	623.9998
2.355	129	-91.1078	91.32567
3.14	395	-394.999	0.629098
3.925	99	-70.1428	-69.8641
4.71	213	-0.50885	-212.999
5.495	28	19.74373	-19.8541

Figure 2: 2D Scan

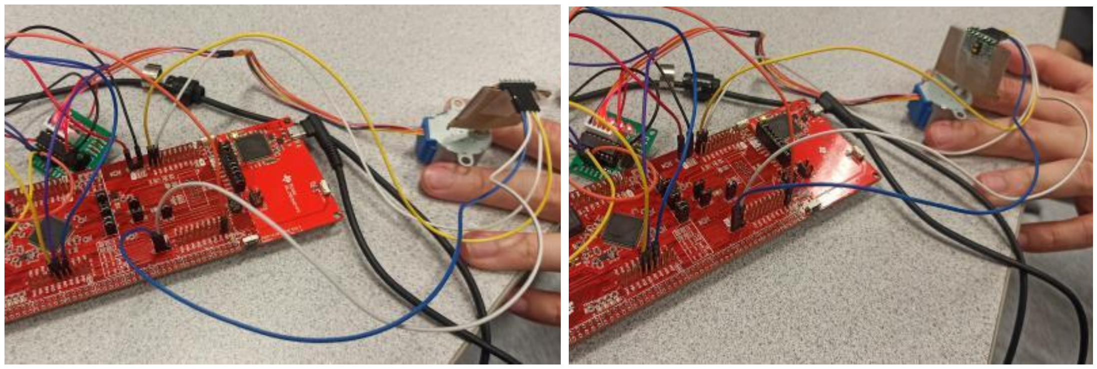
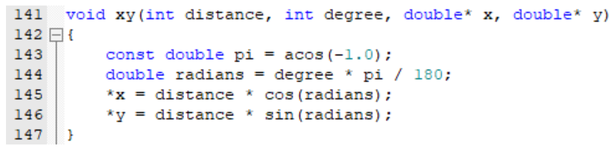


Figure 3: Time-of-Flight Sensor Mounted onto Stepper Motor

Deliverable 2

This deliverable is a full integration of project 1, demonstrating full functionality of a system using a microcontroller. It integrates all studios and lab concepts together. The main segments used are a stepper motor on port H, a TOF sensor on Ports B 2 and 3, serial communication, push buttons, and a PC running visualization code. The on board push button PJ0 acts as a start can button and PJ1 acts as a stop button, which the PC reads and visualizes the point cloud. The stepper motor rotates every 11.25 degrees while the TOF sensor scans 32 times per frame. After each frame, the data is converted from polar to rectangular coordinates and sent to the PC using UART. D2 blinks every time the stepper motor moves and D3 blinks every time the microprocessor waits for the TOF sensor to transmit data. The full embedded system can be seen in Figure 4 and 5.


```
void xy(int distance, int degree, double* x, double* y)
{
    const double pi = acos(-1.0);
    double radians = degree * pi / 180;
    *x = distance * cos(radians);
    *y = distance * sin(radians);
}
```

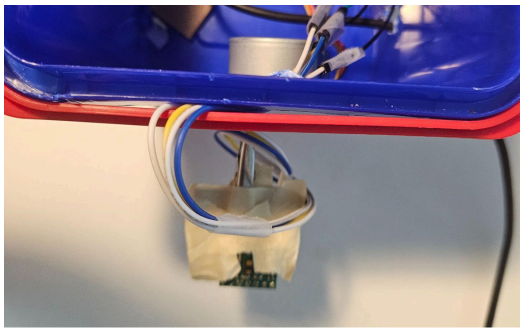



Figure 4: Data Conversion Code and Time-of-Flight Sensor Mount

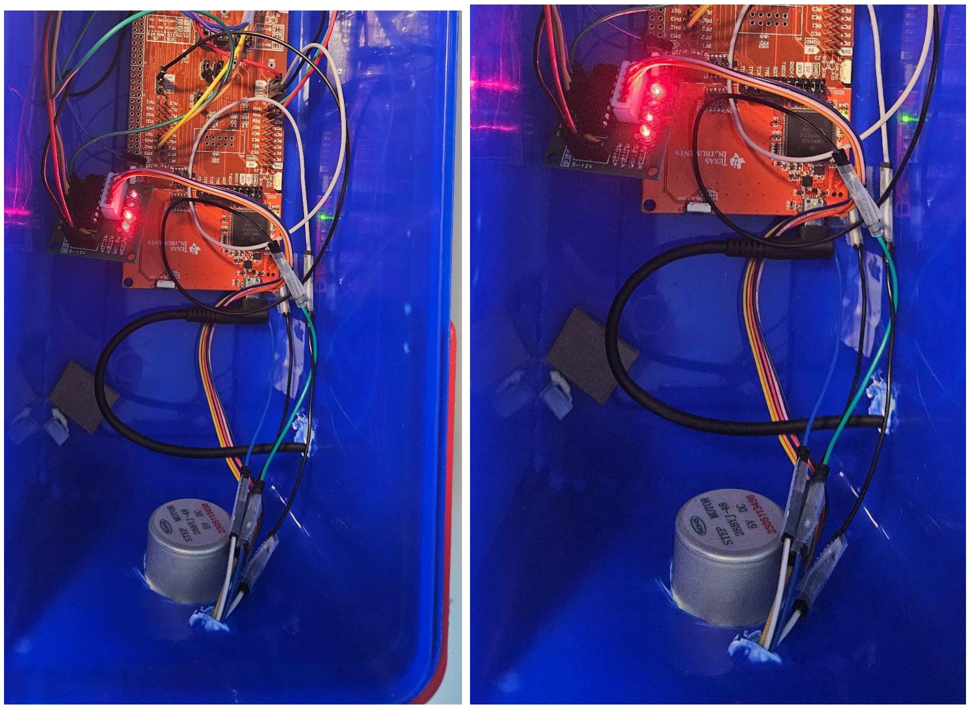
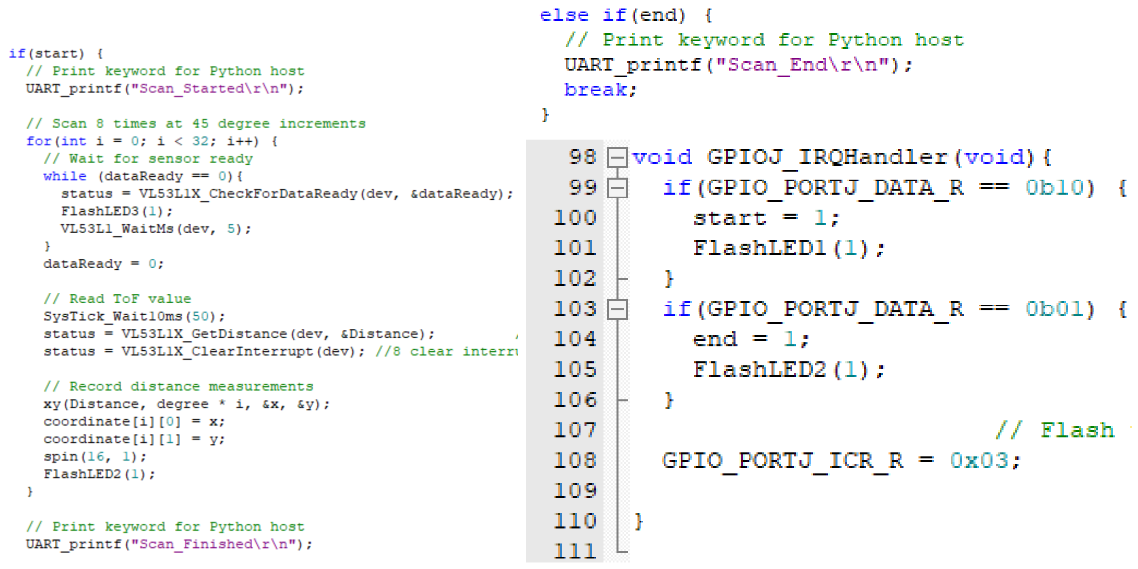


Figure 5: Full Embedded System with Motor Rotation

In order to initiate and stop the scan, interrupts were used. Whenever a button press is detected, the microprocessor detects the button, clears the flag, and runs the corresponding code. In this case, there were two options, PJ0 which sets start to 1 and initiates the scan, and PJ1 sets end to 1 which ends all scans for the data to be visualized. These can be seen in the Figure below.



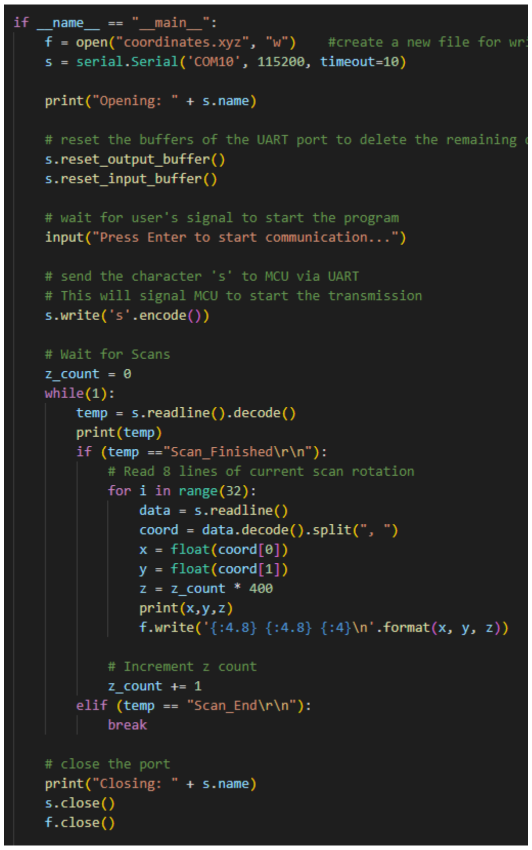
```
if(start) {
  // Print keyword for Python host
  UART_printf("Scan_Started\r\n");

  // Scan 8 times at 45 degree increments
  for(int i = 0; i < 32; i++) {
    // Wait for sensor ready
    while (dataReady == 0){
      status = VL53L1X_CheckForDataReady(dev, &dataReady);
      FlashLED3(1);
      VL53L1_WaitMs(dev, 5);
    }
    dataReady = 0;

    // Read ToF value
    SysTick_Wait10ms(50);
    status = VL53L1X_GetDistance(dev, &Distance);
    status = VL53L1X_ClearInterrupt(dev); //8 clear interru

    // Record distance measurements
    xy(Distance, degree * i, &x, &y);
    coordinate[i][0] = x;
    coordinate[i][1] = y;
    spin(16, 1);
    FlashLED2(1);
  }

  // Print keyword for Python host
  UART_printf("Scan_Finished\r\n");
```

```
else if(end) {
  // Print keyword for Python host
  UART_printf("Scan_End\r\n");
  break;
}
```

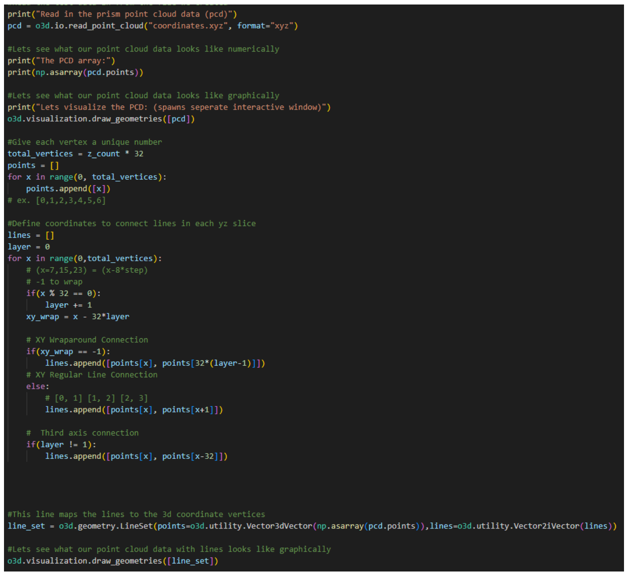
```
void GPIOJ_IRQHandler(void){
  if(GPIO_PORTJ_DATA_R == 0b10) {
    start = 1;
    FlashLED1(1);
  }
  if(GPIO_PORTJ_DATA_R == 0b01) {
    end = 1;
    FlashLED2(1);
  }
                                          // Flash
  GPIO_PORTJ_ICR_R = 0x03;

}
```

Figure 6: Interrupts and Scan Keil Code

Once the end key is hit, all scans end and *"Scan_Finished\r\n"* is sent to the PC using UART. In addition, after every frame, the data is saved in *"coordinates.xyz"* through write mode and z is incremented by 400mm. When the end message is detected, the program uses Open3d and Numpy to visualize the data. The full python code can be seen in Figure 6 below.



```python
if __name__ == "__main__":
    f = open("coordinates.xyz", "w")    #create a new file for wr
    s = serial.Serial('COM10', 115200, timeout=10)

    print("Opening: " + s.name)

    # reset the buffers of the UART port to delete the remaining 
    s.reset_output_buffer()
    s.reset_input_buffer()

    # wait for user's signal to start the program
    input("Press Enter to start communication...")

    # send the character 's' to MCU via UART
    # This will signal MCU to start the transmission
    s.write('s'.encode())

    # Wait for Scans
    z_count = 0
    while(1):
        temp = s.readline().decode()
        print(temp)
        if (temp =="Scan_Finished\r\n"):
            # Read 8 lines of current scan rotation
            for i in range(32):
                data = s.readline()
                coord = data.decode().split(", ")
                x = float(coord[0])
                y = float(coord[1])
                z = z_count * 400
                print(x,y,z)
                f.write('{:4.8} {:4.8} {:4}\n'.format(x, y, z))

            # Increment z count
            z_count += 1
        elif (temp == "Scan_End\r\n"):
            break

    # close the port
    print("Closing: " + s.name)
    s.close()
    f.close()
```



```python
print("Read in the prism point cloud data (pcd)")
pcd = o3d.io.read_point_cloud("coordinates.xyz", format="xyz")

#Lets see what our point cloud data looks like numerically
print("The PCD array:")
print(np.asarray(pcd.points))

#Lets see what our point cloud data looks like graphically
print("Lets visualize the PCD: (spawns seperate interactive window)")
o3d.visualization.draw_geometries([pcd])

#Give each vertex a unique number
total_vertices = z_count * 32
points = []
for x in range(0, total_vertices):
    points.append([x])
# ex. [0,1,2,3,4,5,6]

#Define coordinates to connect lines in each yz slice
lines = []
layer = 0
for x in range(0,total_vertices):
    # (x=7,15,23) = (x-8*step)
    # -1 to wrap
    if(x % 32 == 0):
        layer += 1
    xy_wrap = x - 32*layer

    # XY Wraparound Connection
    if(xy_wrap == -1):
        lines.append([points[x], points[32*(layer-1)]])
    # XY Regular Line Connection
    else:
        # [0, 1] [1, 2] [2, 3]
        lines.append([points[x], points[x+1]])

    #  Third axis connection
    if(layer != 1):
        lines.append([points[x], points[x-32]])



#This line maps the lines to the 3d coordinate vertices
line_set = o3d.geometry.LineSet(points=o3d.utility.Vector3dVector(np.asarray(pcd.points)),lines=o3d.utility.Vector2iVector(lines))

#Lets see what our point cloud data with lines looks like graphically
o3d.visualization.draw_geometries([line_set])
```

Figure 7: PC Visualization Code

This whole process is done autonomously, excluding the button presses and movement of the system by the user. Figure 7 presents the hallway while Figure 8 shows a multiview of the scan by the system. With a 32 segmented scan per frame, the accuracy of the system can be seen. In the front view, it can be seen that the corners of the roof were detected and angled. In addition, the cubby, and the open doors out of view on the left side were out of range and were not detected. Finally in the side view facing down, the back can be seen at the top of the image. This back area also has a missing segment in it. This represents the hallway in the right of Figure 7, which was not detected as it was also out of range.

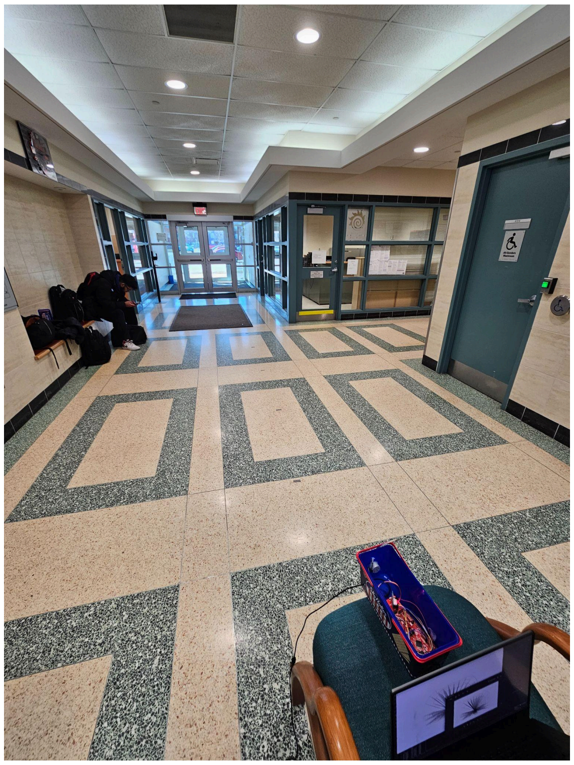


Figure 8: Hallway Located in ITB, Location A

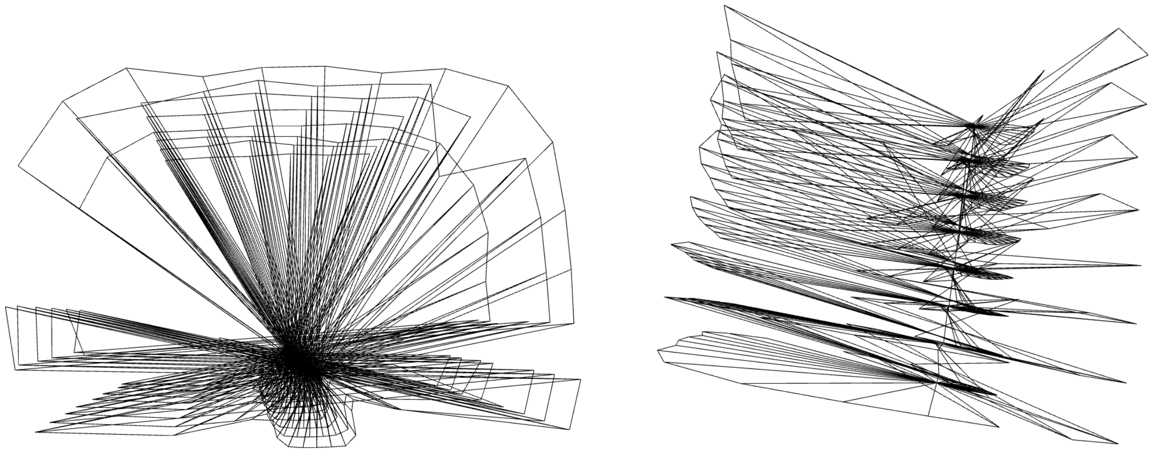
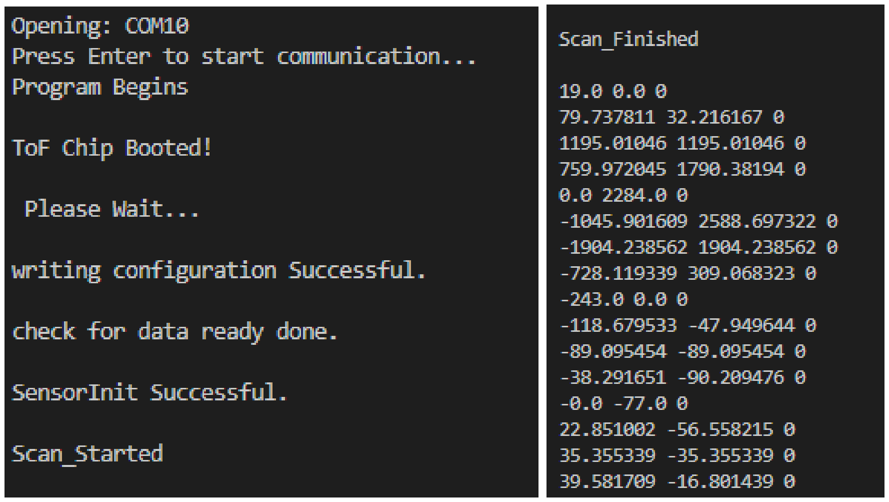


Figure 9: System Scan Front and Right View Facing Down

Debugging

Two debugging strategies were used in deliverable 2. Prompts sent from the microcontroller through UART were sent to the PC and viewed in the Visual Studio Code terminal. These prompts are viewed by the user and tell the user when the TOF boots, eligibility to start the scan, start scan, and end scan. In addition, once one frame of the scan is finished, all 32 coordinates can be viewed in this terminal. These allow the user to verify all states of the scanning process and to verify functionality of the TOF startup and the entire system.

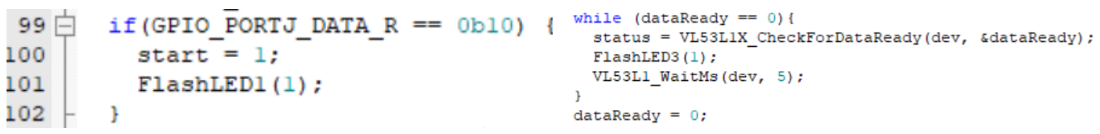


```
Opening: COM10
Press Enter to start communication...
Program Begins

ToF Chip Booted!

 Please Wait...

writing configuration Successful.

check for data ready done.

SensorInit Successful.

Scan_Started
```

```
Scan_Finished

19.0 0.0 0
79.737811 32.216167 0
1195.01046 1195.01046 0
759.972045 1790.38194 0
0.0 2284.0 0
-1045.901609 2588.697322 0
-1904.238562 1904.238562 0
-728.119339 309.068323 0
-243.0 0.0 0
-118.679533 -47.949644 0
-89.095454 -89.095454 0
-38.291651 -90.209476 0
-0.0 -77.0 0
22.851002 -56.558215 0
35.355339 -35.355339 0
39.581709 -16.801439 0
```

Figure 10: Visual Studio Code Terminal

LED debugging strategies were a secondary form of debugging used in deliverable 2. In the startup phase of the microprocessor, all LEDs flashed twice to tell the user the system was ready to collect an input. LED 1 flashes when the user pressed PJ0 to start the scan. LED 2 flashing was used when the stepper was in rotation. LED 3 flashing was used when the microprocessor awaited data from the TOF sensor. These were used a numerous amount of times in the testing phases. For example, the system would suddenly stop randomly in the testing phase. When checking the LEDs, it was easy to see the TOF sensor was not working as intended as LED 3 was blinking. This meant that the microprocessor did not receive a verification of new data being scanned from the TOF sensor.



```
if(GPIO_PORTJ_DATA_R == 0b10) {
  start = 1;
  FlashLED1(1);
}
```

```
while (dataReady == 0){
  status = VL53L1X_CheckForDataReady(dev, &dataReady);
  FlashLED3(1);
  VL53L1_WaitMs(dev, 5);
}
dataReady = 0;
```

Figure 11: LED 1 and 3 Blink

Synthesis

The theme of this report focuses on the ability of a system to act autonomously in response to input data. This is significant for real-time decision making in autonomous systems. In Lab 8 and Deliverable 2, this concept is demonstrated in the microcontroller's ability to detect, decide, and respond using outputs. In Lab 8, button presses trigger GPIO interrupts, prompting the microcontroller to immediately send a request to a TOF sensor via I2C. Each button initiates an interrupt service routine that returns the appropriate function.

Deliverable 2 expands on this by integrating these reactive behaviors into a complete system. Rather than only collecting data and processing it, the microcontroller responds to behaviour changes like start/stop commands. It then performs actions such as motor rotation, sensor scanning, and data transmission accordingly. The "Act" theme is shown in this deliverable by not just functionality, but the way the system transitions between states based on input and internal states. LED indicators and UART messages provide real-time confirmation of these actions, meaning the system is continually observing, interpreting, and reacting. This ability to process input and generate an immediate output is the essence of "act" in embedded systems.

Reflection

Lab 8 and Deliverable 2 deepened my understanding of how embedded systems must not only gather data but act accordingly in real time. At first, it seemed like acting on inputs was simply running functions after an event. But I quickly realized that timings, interruptions, and clear code execution in predictable ways. In Lab 8, handling GPIO interrupts emphasized how systems must be responsive to actions such as button presses. Deliverable 2 expanded on this developing a fully integrated system. The microcontroller had to balance and control motor movement, sensor communication, and data output based on user input, the environment, and current state. This process showed me how embedded systems are designed to act not just once, but repeatedly. In addition, they must be reliable based on evolving data to support the user.

In addition, it surprised me how important debugging was to ensure the system acts correctly. LED indicators and messages transmitted via UART provided me with feedback on the system state, which was essential for verifying that the microcontroller was working as intended in every state. For instance, LED blinks helped identify whether the system was rotating the motor or waiting for sensor data. These visual cues made it easier to isolate problems rather than guessing.

In conclusion, these assignments reinforced that acting in embedded systems is more than just outputting signals, but a systems ability to observe the environment around it and to respond reliably in a predictable manner. The experience highlighted the importance of timing, interrupts, and feedback in creating systems that behave autonomously making real-time decisions based on challenging environments.