<u>Valencia Team</u>

IEEE Orlando Section

<u>Progress Report 3: March</u>

Project Addressing Climate Change:

Solar Powered Ventilation with Controlled Airflow for Parked Cars

Daniel Eisenbraun [IEEE # 98706125]

Ian Matheson [IEEE # 98579639]

Supervisor:

Dr. Masood Ejaz [IEEE # 92131157]

Department of Electrical & Computer Engineering Technology (ECET)

Division of Engineering, Computer Programming, & Technology

Valencia College

2022 - 2023

<u>Contact: Daniel Eisenbraun & Ian Matheson</u>
deisenbraun@mail.valenciacollege.edu
imatheson@valenciacollege.edu

Introduction:

This brief report covers important project developments in March. A review of spent budget and added costs will be given, followed by build progress.

Spent Budget and Added Costs:

The group incurred one additional cost this month when they swapped the IR proximity sensors for HC-SR04 Ultrasonic proximity sensors. The cost is given in the "Added Costs" section of the Project Budget.

Progress:

Build Progress

The group made the final push to complete the full system build this month. This includes some software modifications such as cooling mode hysteresis, MCU on-board temperature sensing, and datalogging capabilities. Hardware modifications include the replacement of IR proximity sensors with Ultrasonic proximity sensors and full system assembly. The vent-visors were mounted to the test vehicle and any resulting damages were amended.

Cooling Mode Hysteresis

The flowchart in *figure 1* shows how the hysteresis is implemented. Two global variables hold the previous mode and previous temperature standard deviation. At the start of the function, the new temperature standard deviation is taken. If the difference between them is greater than 3, then a cooling mode will be selected. The previous mode will then be set to the selected mode, and the previous temperature deviation will be set to the new temperature deviation. If the condition is not satisfied, then the mode returned by the function will be set equivalent to the previous mode.

The printout below shows the hysteresis in action. With no temperature sensors plugged in, the T values vary wildly. The cooling mode is only selected when the difference between new_t and prev_t is greater than 3. This is observed from the first to the second iteration and from the third to the fourth iteration.

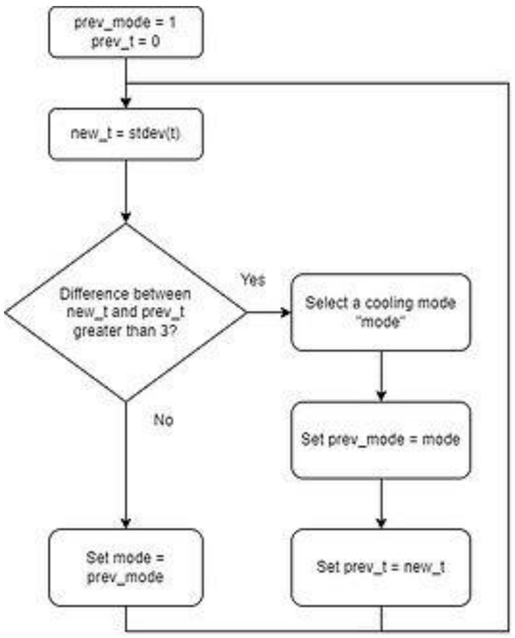
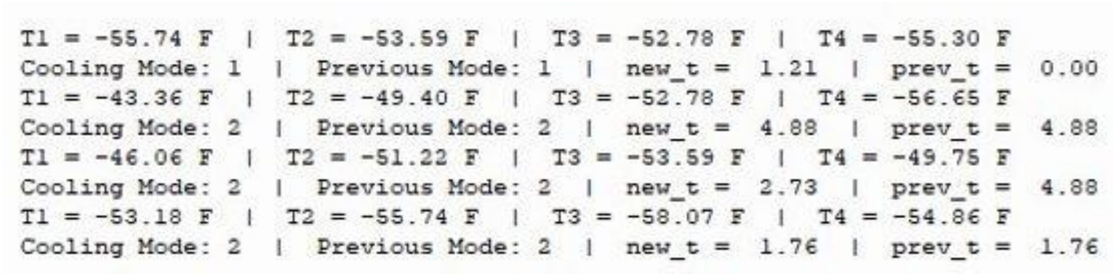


Figure 1 - Flowchart for cooling mode hysteresis.



```
T1 = -55.74 F  |  T2 = -53.59 F  |  T3 = -52.78 F  |  T4 = -55.30 F
Cooling Mode: 1  |  Previous Mode: 1  |  new_t =  1.21  |  prev_t =  0.00
T1 = -43.36 F  |  T2 = -49.40 F  |  T3 = -52.78 F  |  T4 = -56.65 F
Cooling Mode: 2  |  Previous Mode: 2  |  new_t =  4.88  |  prev_t =  4.88
T1 = -46.06 F  |  T2 = -51.22 F  |  T3 = -53.59 F  |  T4 = -49.75 F
Cooling Mode: 2  |  Previous Mode: 2  |  new_t =  2.73  |  prev_t =  4.88
T1 = -53.18 F  |  T2 = -55.74 F  |  T3 = -58.07 F  |  T4 = -54.86 F
Cooling Mode: 2  |  Previous Mode: 2  |  new_t =  1.76  |  prev_t =  1.76
```

Reading the Onboard Temperature Sensor

To read the temperature in Fahrenheit from the Raspberry Pi Pico onboard sensor, the following two lines of code are used. The sensor is reasonably accurate and matched the room temperature closely.

```
import microcontroller
internal_T = microcontroller.cpu.temperature*(9/5) + 32
```

Writing Data to a .txt File

Writing data to files using CircuitPython is tricky. CircuitPython cannot normally write data to files from within the program; the tradeoff is that the programmer has read and write access to the whole file structure on the board. Luckily, there is a simple workaround. First, a file named boot.py is created in the main directory. This file is the first thing that executes when the Pico gets power. The following code is written in boot.py. This allows the Pico to write.

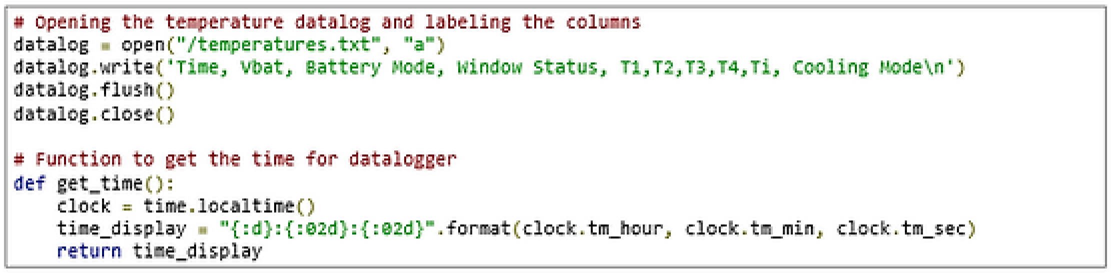
```
import storage
storage.remount("/", readonly=False)
```

The tricky part about this is that once the Pico has write access, the user can no longer write files, so no changes can be made to the code in this mode. To make changes, the user can use the CircuitPython terminal to execute the following code, which changes the name of boot.py so that it doesn't run on startup. To switch the mode back again, the user can simply change the name back to boot.py from the file explorer.

```
import os
os.rename("boot.py", "boot1.py")
```

Another tricky thing about this method is that once the Pico has write access, the user has no way to know whether the code is executing properly. To do that, a try/except method is used in the main loop to change the blink speed of the onboard LED according to the error status.

With the above functionality in place, writing data to a txt file is simple. The following code opens the datalog file in append mode and places the headers in the first line. The get_time function is called as the first argument in every iteration of the datalog.write code to get the time from Pico startup. Unfortunately, the Pico does not have a real-time clock, so the start time must be noted manually anytime we run tests.



```
# Opening the temperature datalog and labeling the columns
datalog = open("/temperatures.txt", "a")
datalog.write('Time, Vbat, Battery Mode, Window Status, T1,T2,T3,T4,Ti, Cooling Mode\n')
datalog.flush()
datalog.close()

# Function to get the time for datalogger
def get_time():
    clock = time.localtime()
    time_display = "{:d}:{:02d}:{:02d}".format(clock.tm_hour, clock.tm_min, clock.tm_sec)
    return time_display
```

Running the Pico standalone in write mode, the following .txt file is generated. This is easily converted to spreadsheet format by importing the .txt file in Excel and using commas as the column delimiter.

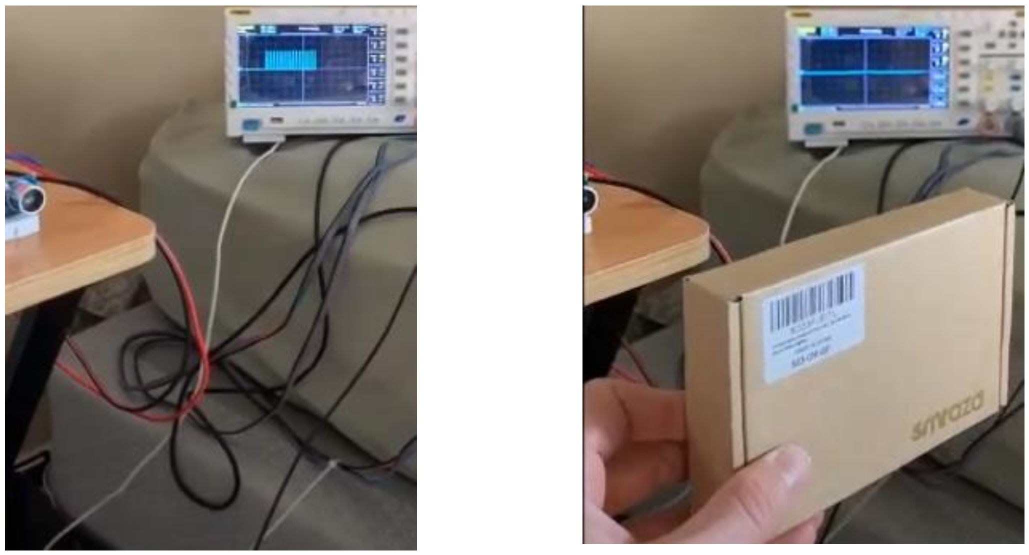
```
Time, Vbat, Battery Mode, Window Status, T1,T2,T3,T4,Ti, Cooling Mode
0:00:03,0.200, 0,Down,-58.07,-49.04,-60.10,-49.04,70.74,3
0:05:04,0.169, 0,Down,-50.85,-44.83,-51.22,-48.35,73.27,3
0:10:04,0.175, 0,Down,-50.11,-45.13,-46.06,-48.01,73.27,3
```

Switch to Ultrasonic Proximity Sensors

Following the confirmation of a major design flaw with the IR proximity sensors, the group sought an alternative. Dr. Radu Bunea, a professor at Valencia College, suggested that ultrasonic sensors be used in place of IR sensors. They effectively perform the same task but use sound instead of light. They are also far cheaper than quality optical filters.

Swap to Ultrasonic Proximity Sensor

To replace the IR Proximity sensor, the HC-SR04 Ultrasonic Proximity Sensors were acquired. A test was performed which confirmed their functionality as proximity sensors with a function generator and an oscilloscope.

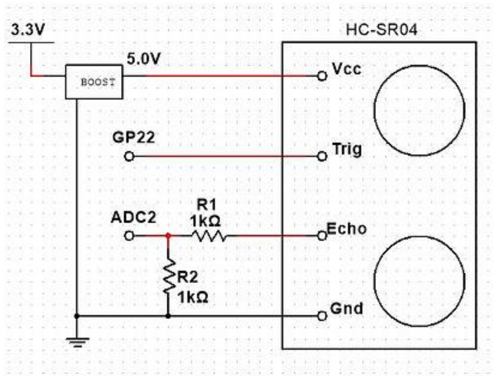


(a) (b)

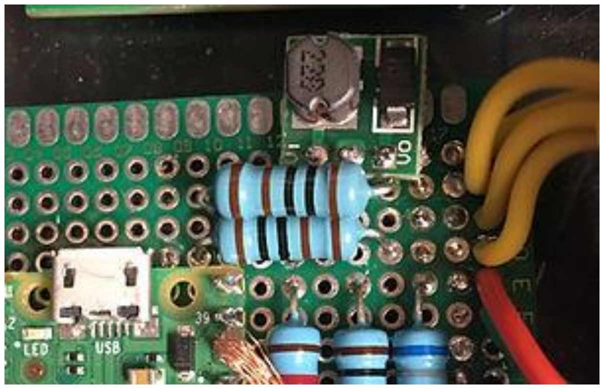
Figure 2 - (a) Ultrasonic Echo signal with no object present and (b) with object present.

System Integration of Ultrasonic Sensors

Luckily, this sensor has the same number of pins that the IR sensor had, so implementing it into the control module was relatively painless. To start, the schematic in *figure 3a* was drawn up to use as a reference while soldering. Next, the IR circuit components were desoldered from the MCU board, and the Ultrasonic circuit components were added. The completed circuit is shown in *figure 3b*.



(a)



(b)

Figure 3 - Ultrasonic sensor components soldered to the MCU PCB.

2-pin connectors were used to connect the ultrasonic sensors to the control module. These were soldered and wrapped directly to the pins. The green indicator LED's cathode was also soldered directly to the ground pin of the window 1 ultrasonic sensor. A single-pin connector was used to connect the LED's anode.

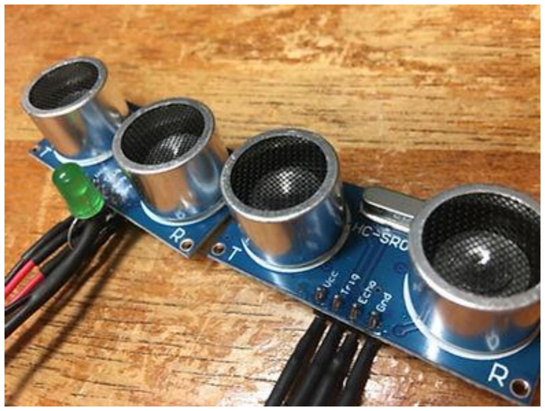
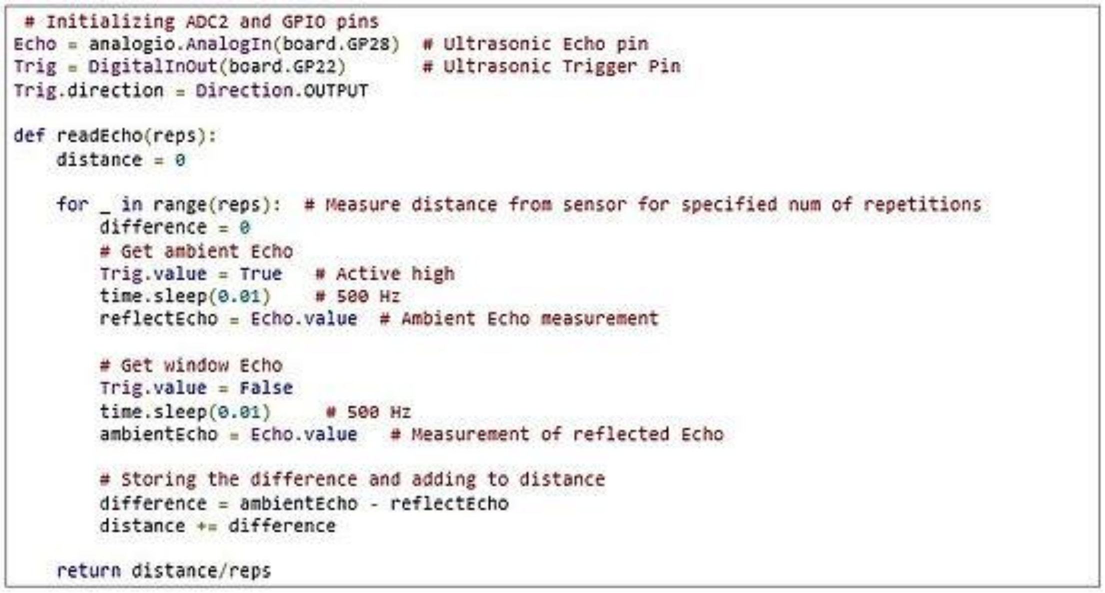


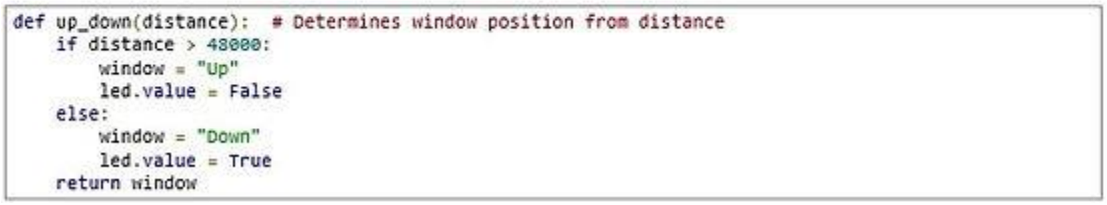
Figure 4 - Wiring of Ultrasonic sensor and LED.

Next, the proximity sensor code had to be modified. Ian kept the code very similar to the original proximity code. It essentially pulses the Trig pin at 50 Hz and reads the Echo pin at the same rate. When an object approaches the sensor, the average voltage at the Echo pin should increase.



```
 # Initializing ADC2 and GPIO pins
Echo = analogio.AnalogIn(board.GP28)  # Ultrasonic Echo pin
Trig = DigitalInOut(board.GP22)       # Ultrasonic Trigger Pin
Trig.direction = Direction.OUTPUT

def readEcho(reps):
    distance = 0

    for _ in range(reps):  # Measure distance from sensor for specified num of repetitions
        difference = 0
        # Get ambient Echo
        Trig.value = True   # Active high
        time.sleep(0.01)    # 500 Hz
        reflectEcho = Echo.value  # Ambient Echo measurement

        # Get window Echo
        Trig.value = False
        time.sleep(0.01)     # 500 Hz
        ambientEcho = Echo.value   # Measurement of reflected Echo

        # Storing the difference and adding to distance
        difference = ambientEcho - reflectEcho
        distance += difference

    return distance/reps
```

To determine whether the window is up or down, the threshold is set in the code below.



```
def up_down(distance):  # Determines window position from distance
    if distance > 48000:
        window = "Up"
        led.value = False
    else:
        window = "Down"
        led.value = True
    return window
```

To test the sensors, all four of them were connected and set up on the bench such that they were aimed out into the room. A hard reflective object was placed in front of each ultrasonic sensor and then removed one at a time. The distance value was observed. With no object present, the distance value came out to about 50000. When an object was placed right in front of any of the sensors, a drop of about 5000 occurred, putting the distance around 45000. The threshold was set so that if the distance exceeded 48000, the window would be registered as "Down". Else, the window registers as "Up".

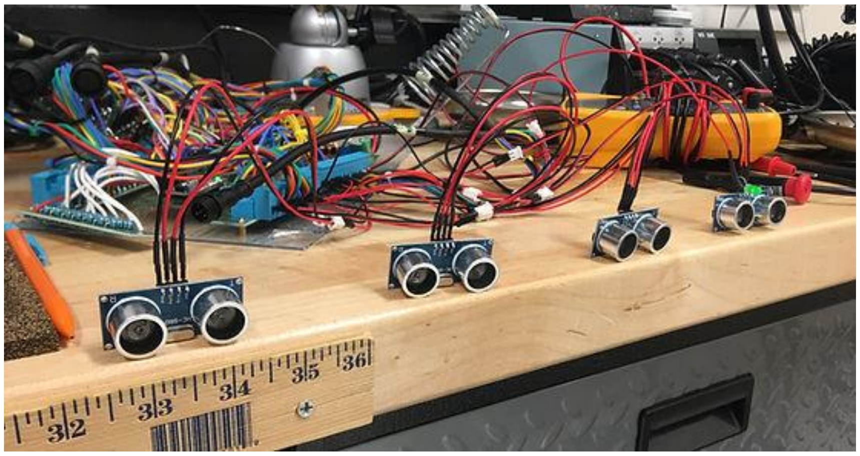
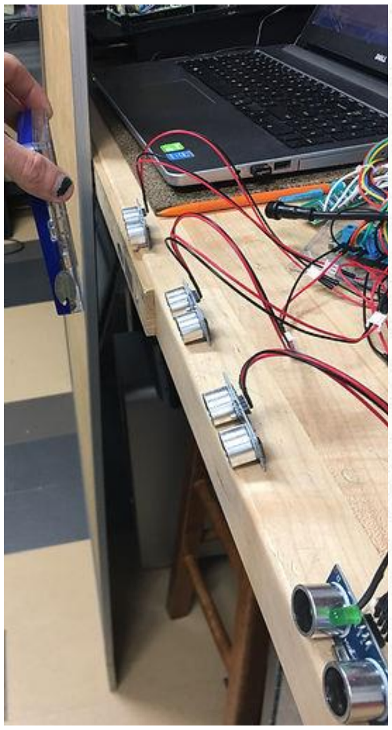
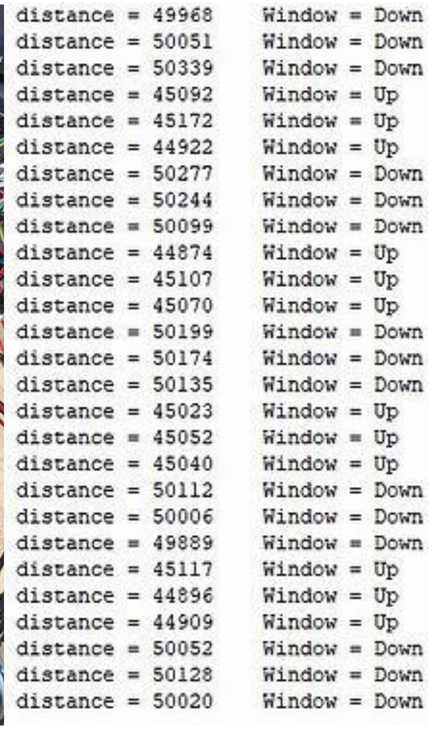


Figure 5 - Test setup for ultrasonic proximity sensors.

With the threshold set properly, Ian ran the code and moved the object in front of each of the sensors in three-cycle intervals to produce the printout shown below. The green LED attached to the window 1 sensor lit up when the window was "Down" and turned off when the window was "Up," as expected.

```
distance = 49968    Window = Down
distance = 50051    Window = Down
distance = 50339    Window = Down
distance = 45092    Window = Up
distance = 45172    Window = Up
distance = 44922    Window = Up
distance = 50277    Window = Down
distance = 50244    Window = Down
distance = 50099    Window = Down
distance = 44874    Window = Up
distance = 45107    Window = Up
distance = 45070    Window = Up
distance = 50199    Window = Down
distance = 50174    Window = Down
distance = 50135    Window = Down
distance = 45023    Window = Up
distance = 45052    Window = Up
distance = 45040    Window = Up
distance = 50112    Window = Down
distance = 50006    Window = Down
distance = 49889    Window = Down
distance = 45117    Window = Up
distance = 44896    Window = Up
distance = 44909    Window = Up
distance = 50052    Window = Down
distance = 50128    Window = Down
distance = 50020    Window = Down
```

Figure 6 - Testing integrated operation of all four ultrasonic proximity sensors.

Ultrasonic Proximity Sensor Testing and Troubleshooting

Once the vent visors were patched up and adhered to the testing vehicle, the group set about testing the ultrasonic proximity sensors. The sensors immediately presented some issues. The distance value came back lower than it did in the lab and the sensors weren't as sensitive to objects in close proximity. Also, one of the sensors seemed to be far less sensitive than the others, making it impossible to properly set the switching threshold.

First, the group used an oscilloscope to verify that the correct signals were being sent and received by each sensor. Once this was verified, the group began modifying the program to adjust the sensitivity of the sensors. A variety of pulse frequencies and repetitions were tested to no avail. The group decided that a rewriting of the program could prove beneficial at this stage.

Going by the HC-SR04 datasheet (“Ultrasonic Ranging Module HC-SR04,” n.d.), the sensor can be triggered by a 10us or greater pulse. At the falling edge of this pulse, the transmitter will emit an 8-cycle sonic burst at 40 kHz. The Echo Pulse lever signal output which follows is proportional to the proximity of the sensor to the object it is directed towards.

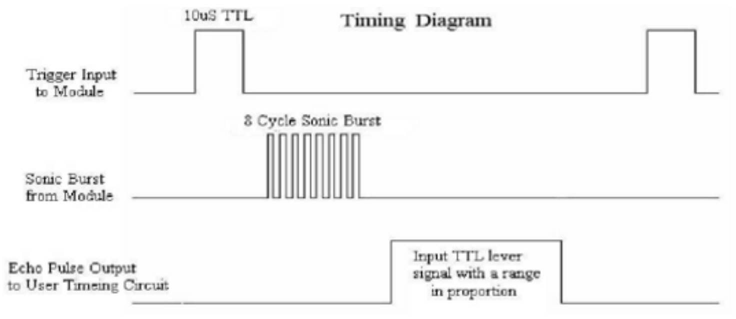

10uS TTL
Timing Diagram
Trigger Input to Module
8 Cycle Sonic Burst
Sonic Burst from Module
Echo Pulse Output to User Timeing Circuit
Input TTL lever signal with a range in proportion

Figure 7 - Timing diagram for HC-SR04 ultrasonic sensor.

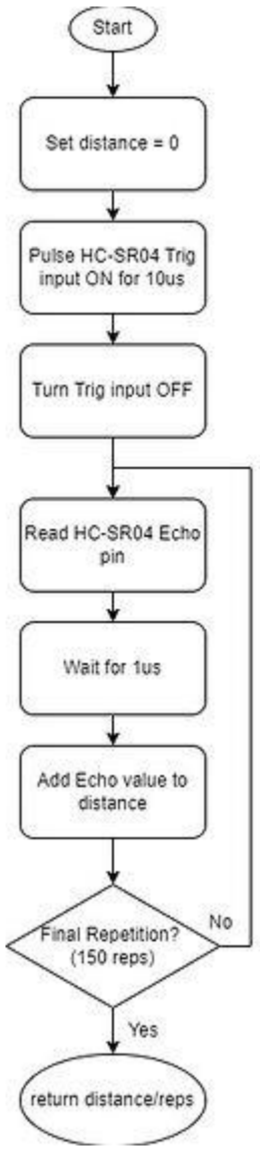


Figure 8 -Flowchart for Ultrasonic Proximity Sensor Software

The flowchart above shows how the new program works. After setting the distance variable to 0, the Pico sends a 10us pulse to the HC-SR04. Immediately after, the Pico reads the ADC connected to the HC-SR04 Echo pin every microsecond for a specified number of repetitions. Once the specified number of reps has occurred, the program returns the average distance value.

Testing the program in the vehicle, the sensors were far more sensitive and consistent, but one sensor had reduced sensitivity. The group adjusted the number of repetitions until the sensitivity of all the sensors was great enough. 150 repetitions worked very well, and a new switching threshold was set at 25000.

The group tested the functionality of the proximity sensors according to the test procedures laid out by Daniel: by rolling down all the windows and then rolling them up one at a time. Per the engineering requirements, power should be disconnected from the fans if any of the windows are rolled up too high for airflow. The green LED should also be on when the windows are down and off if they are up.

The results of the test indicate that the proximity sensor circuit and control are functioning exactly as intended. When any of the windows are rolled up too high for airflow, the fans and the green LED shut off. When all windows are low enough, the fans and green LED automatically turn back on. This constitutes a pass for the engineering requirement concerning window sensors.

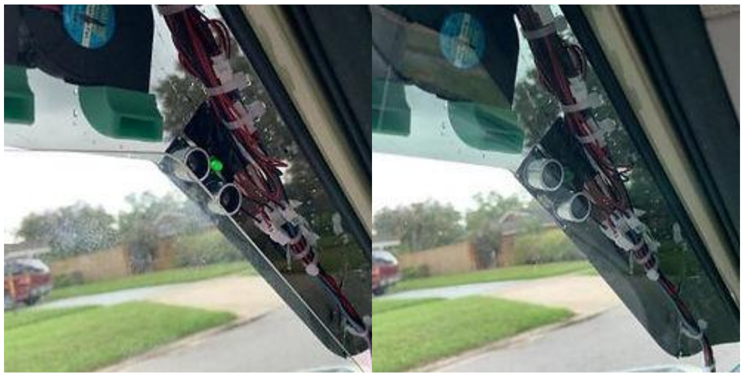


Figure 9 - The Green LED is on when the window is down and off when the window is up.

Preparing and Mounting the Vent-Visor Fan Assemblies

Using lots of zip ties, the group wrangled the fan cables into something manageable. Zip-tie surface mounts were attached to the vent visors using nuts and bolts, and these were used to zip-tie the cables in place.

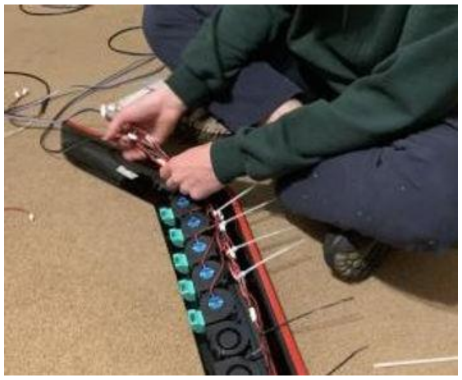


Figure 10 - Wire management on the vent-visor assembly.

Until this point, the vent visors had been quite resilient to drilling and mounting. Fastening the final screws for cable mounting put too much stress on the visors, however, and resulted in some terrifying cracks. Pictured below are two of the most gruesome examples.

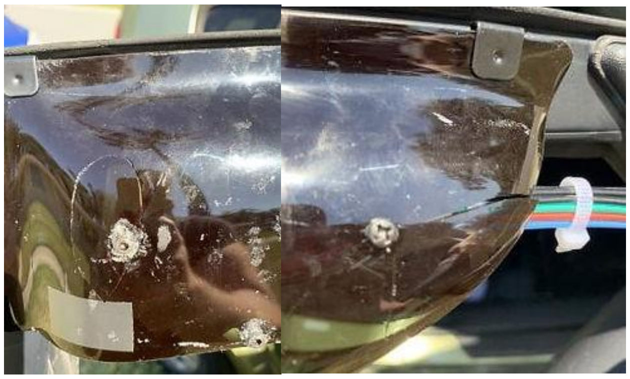


Figure 11 - Breaks in the vent-visors.

To amend these breaks, the group applied hefty amounts of Flex-Glue to them. In some areas, some electrical tape and/or epoxy were also used. Later, spray-on adhesive was applied across each vent-visor, creating a protective layer over the breaks.

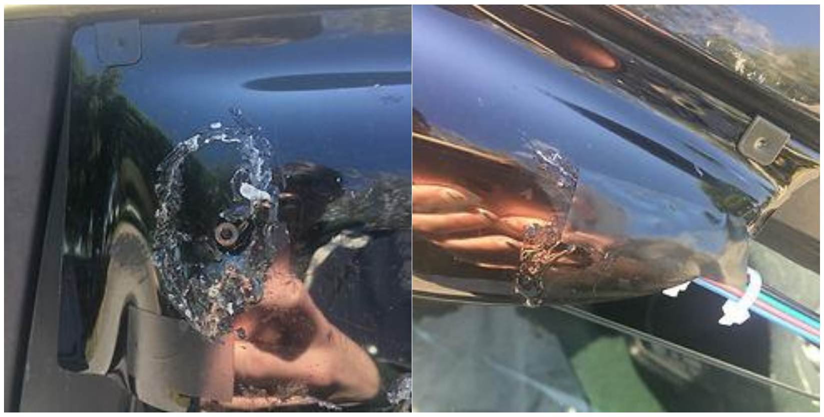
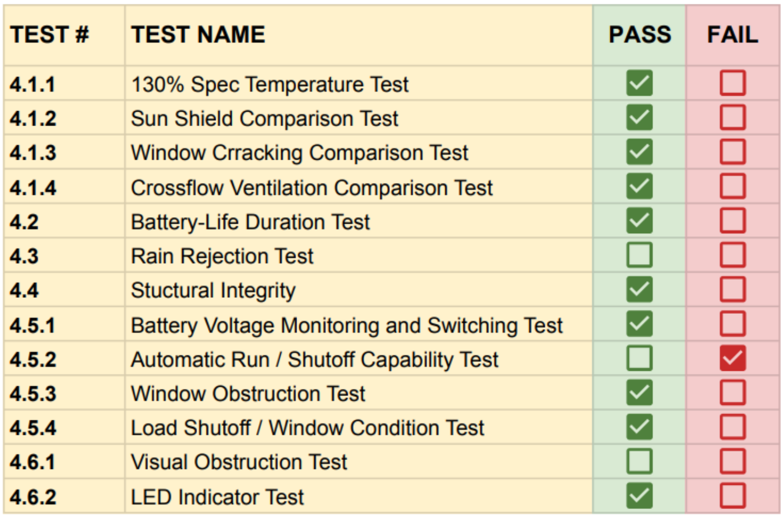


Figure 12 - Fixes for breaks in vent-visors.

<u>Full System Testing & Results</u>

With the full system assembled, the group set about testing the project requirements. Daniel wrote an Acceptance Test Procedure document to give the test procedures, rules, guidelines, as well as the pass or fail criteria. The full document can be found on the group's website. A list of completed tests and their results was created.

Table 1 – List of tests and PASS/FAIL status.



TEST #	TEST NAME	PASS	FAIL
4.1.1	130% Spec Temperature Test	✓	☐
4.1.2	Sun Shield Comparison Test	✓	☐
4.1.3	Window Crracking Comparison Test	✓	☐
4.1.4	Crossflow Ventilation Comparison Test	✓	☐
4.2	Battery-Life Duration Test	✓	☐
4.3	Rain Rejection Test	☐	☐
4.4	Stuctural Integrity	✓	☐
4.5.1	Battery Voltage Monitoring and Switching Test	✓	☐
4.5.2	Automatic Run / Shutoff Capability Test	☐	✓
4.5.3	Window Obstruction Test	✓	☐
4.5.4	Load Shutoff / Window Condition Test	✓	☐
4.6.1	Visual Obstruction Test	☐	☐
4.6.2	LED Indicator Test	✓	☐

<u>Summary</u>

This month, the group completed the full system build and performed the majority of the testing needed to verify the project engineering requirements.