# Setting up the Companion Computer

## Download the base code to your Raspberry Pi

To start, you will want to download the available code from the GitHub repository : <https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/tree/main/CompanionSystem> . If the link is not functioning, a current version of the code for each file is available in the appendix.

Inside there are several files and folders. The main ones we will be working with is the “Sensors” folder and the “Companion.py” and “MavVehicle.py” files. Make sure to also read the “Code\_README.md” file as it contains some instructions on setting up services and some additional modules you will need to install using your command prompt. However, we will go over that in this documentation as well.

A screenshot of a computer program

Description automatically generated

Next, we want to open up our command prompt to install the additional modules using the pip install command. In command prompt simply type each line and hit enter, this will install the modules to your raspberry pi.

* VL53L1X: sudo pip install VL53L1X
* smbus2: sudo pip install smbus2
* bme280: sudo pip install pimoroni-bme280
* pymavlink: sudo python -m pip install --upgrade pymavlink

Once these are all successfully downloaded, we can move on to setting up the hardware connections of our companion.

## Setting up Connections

There are three main connections we must make for the companion which include: power source, sensors, and Navio2 communication. Shown in the figure below are the wirings for each connection to the Raspberry Pi.

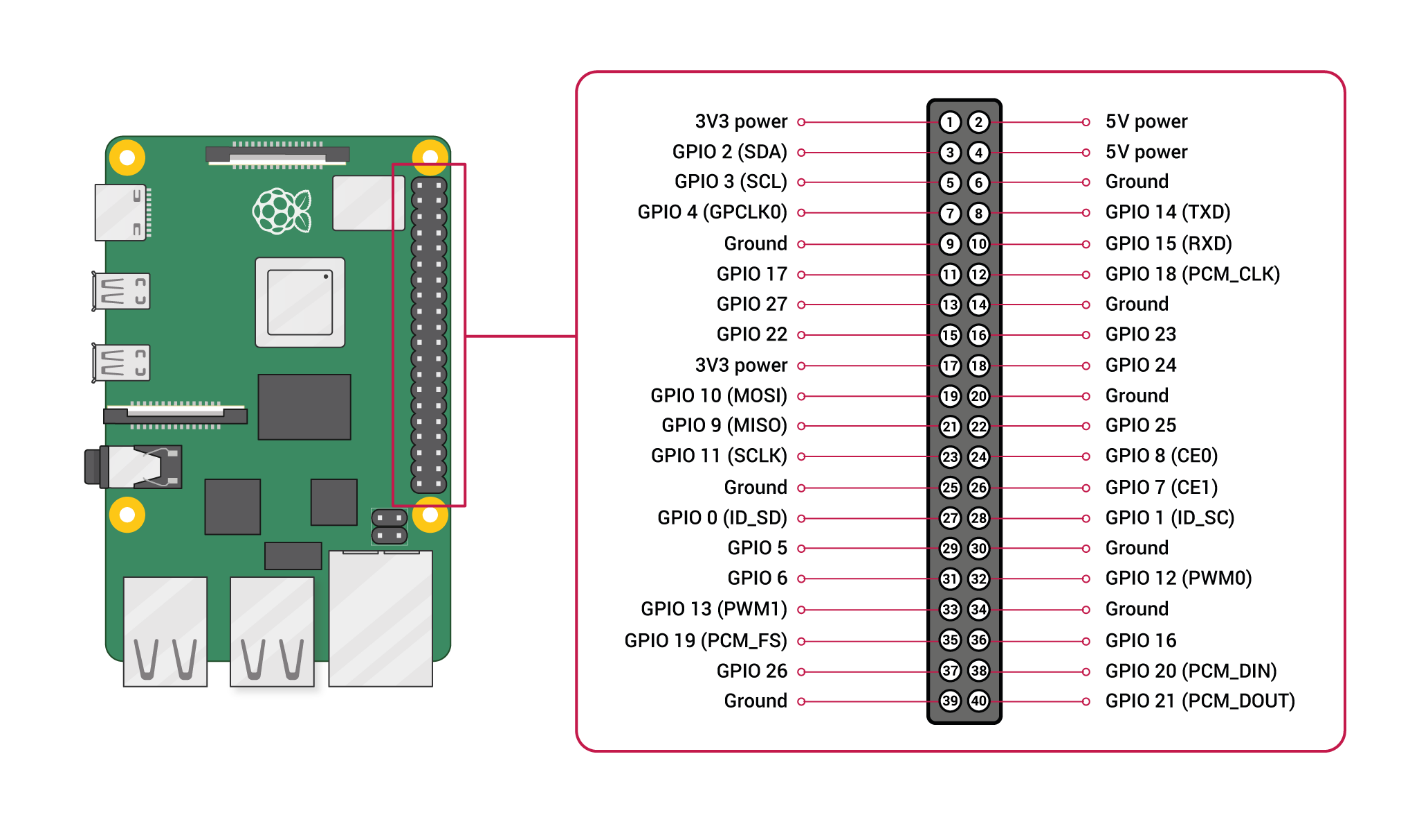
For the sensors, we are mainly using a set of Sparkfun sensors that use the I2C communication protocol with an easy-to-use connection style called QWIIC. This allows us to daisy chain sensors in series to all operate on a single I2C bus.

For the Navio2 communication, we must have an external USB-UART converter to connect the UART Serial pins of our Raspberry Pi to the USB port of our Navio2. Notice we do not attach power from this cable, but must share ground. This is because we are attaching a separate power supply to the Companion as a form of redundancy.

Finally, we must attach a power supply. Currently, we are using a second Power Buck module in parallel with the Power Buck of our Navio’s power supply to run an additional 5V power supply to the companion computer.

The complete wiring list is as follows:

* Sensors
  + 3.3 V Power to 3.3V Power (Pin 1)
  + I2C SDA to GPIO23 (Pin 16)
  + I2C SCL to GPIO24 (Pin 18)
  + Ground to Ground (Pin 20)
* Navio2 Communication
  + RX to TXD/GPIO14 (Pin 8)
  + TX to RXD/GPIO15 (Pin 10)
  + Ground to Ground (Pin 9)
* Power Supply
  + 5 V to 5V Power (Pin 2)
  + Ground to Ground (Pin 6)



A screenshot of a computer

Description automatically generated

A diagram of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

## Follow Task Requirement Instructions

Now that the hardware connections are complete, we will go through and setup all the software configurations for each section of the Companion Computer’s code. The following information covers a brief introduction of the operation of the Companion System and its sub-tasks. Inside each task is a section of Requirements that must be completed before the system will run correctly.

# The Companion System Code Breakdown

## Introduction

The Companion System can be looked at as a set of three tasks operating in parallel. We accomplish this using **threading**, a python module that enables us to run each task as an independent “thread” that can interact with each other(threading --- Thread-based parallelism, n.d.). The three tasks are based on the three goals of our companion: to communicate with a Mavlink-based Flight Controller, collect environmental data using additional onboard sensors, and to log communication and data collection in a single file for later analysis.

## Task 1: Mavlink Communication

### MAVLink - Requirements

The task of communicating with the flight controller has several requirements that we must keep in mind. The first being that it must be able to communicate using the Mavlink messaging protocol (MAVLINK Micro Air Vehicle Communication Protocol, n.d.). This is a common protocol prevalent in many commercially available flight controllers such as Arducopter, Pixhawk, and Navio. For us to communicate with the flight controller, we must use a python-based message processing library that enables us to send and receive MAVLink messages from our Flight Controller that will be sent from a Universal Serial Bus (USB) port of the Navio2 Flight Controller. Due to the nature of our raspberry pi-based companion system, we must convert the incoming USB signals into Universal Asynchronous Receiver/Transmitter (UART) signals of one of the Companion’s serial ports.

An additional step that must be taken is to tell the flight controller to send and receive communication via the USB port of the Navio. We do this by adding an additional Telemetry Configuration to the “/etc/default/ardurover” text file that lists the USB port “/dev/ttyUSB0” as an additional telemetry port.   
A screen shot of a computer

Description automatically generated

Figure . Ardurover Telemetry Configuration.

Then we also will go to into a Mission Control software such as Mission Planner, and ensure that our USB port, which is listed as Serial 1, is set to the desired Baud rate (115,200) and Protocol (MAVLink2).

A screenshot of a computer program

Description automatically generated

Figure . Mission Planner Serial Configuration.

Now when using our companion, we should be able to interact with the flight controller through MAVLink communication.

### MAVLink – Initialization

In order for our raspberry pi-based companion system to communicate with the Navio2 flight controller, we will use a library called pymavlink (Pymavlink, n.d.) to create a connection to the flight controller and begin sending and receiving information sent through the telemetry ports.   
In the code we create the connection using the pymavlink library with the serial port of the companion and expected baud rate. From there, we will wait for what is called a heartbeat which is basic message that the flight controller sends once per second as indicator that the flight controller is present and operating. If the companion detects a heartbeat, it will continue with initialization.

The last portion of the initialization is the creation of a header string and desired MAVLink data and data rates. There are a large variety of messages we can ask to receive from the flight controller, so we will select the ones we are looking for based on what vehicle frame (rover, drone, plane, etc) the companion is mounted on. We will also specify the data rate for each message type. As you increase the variety and rate of data being sent, there will be a point at which the flight controller will begin to view the desired data rates as suggestions. It will send the high priority messages first and from there it will try to send the other message types at a regular rate but may not be able to hit the desired data rate. For our purposes, we are only looking for data sets to be sent roughly once a second.

After we have set the data rate, we will then append the header of each expected data set to a single string. This will include all units inside the message that will then be sent to the data logging task along side the sensor task’s header to create the first line of the logged data file. For example, for the plane configuration we are receiving a message type called VFR\_HUD (Messages (common) - VFR\_HUD, n.d.). It will add 7 items to the header: mav packet type, airspeed, ground speed, heading, throttle, altitude, and climb rate.

Once all the items have been added to the header and data rates set, the task is declared to be successfully initialized and can begin its normal operation.

### MAVLink – Normal Operation

The normal operation of the MAVLink task is based on a simple three-step process: read, parse, and store.

The MAVLink task will first read whatever the latest sent message is from the flight controller. This will store the message as a dictionary file type comprised of key:value pairs. We are able to parse the dictionary based on the first key that is sent with each message, “mavpackettype”, whose corresponding value is a string containing the message name. For example, the VFR\_HUD mavpackettype has a value of “VFR\_HUD”. Using this, we can then perform parsing for only the key:value pairs that are expected from the VFR\_HUD message type. We have an individual variable for each possible data type for all our desired messages, this means that we will have a value inside each variable that will only update the next time the same message is received. This will allow for the data logging task to have access to the latest data for each message type without waiting for each one individually or having empty data sets that haven’t been received yet.

After parsing the latest message, MAVLink stores this data in an updating array that will contain all data types and messages we have received with their latest received message. What this means is that we will have an array with data from VFR\_HUD, GPS, ETC and only the latest received message’s data will be updated. That way an array is always available to the data logging task at any moment since all three tasks are running in different points in their individual code execution. Take a look at the example below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| mavpackettype | groundspeed | climb | mavpackettype | omegaIx | omegaIy |
| VFR\_HUD | 0.284499943 | -0.018549522 | AHRS | 0.002826187 | -0.001374913 |
| VFR\_HUD | 0.284499943 | -0.018549522 | AHRS | 0.002828867 | -0.001375037 |

In this example, we have two rows of data that have been recorded. Notice that both recorded datasets for VFR\_HUD packets are the same, this is because a new VFR\_HUD packet has not been received yet. Meanwhile, the AHRS packet had a new message received in the second row, which is then updated individually.

### MAVLink – Code

The entire code for this task is located in Appendix X.

## Task 2: Sensor Data Collection

### Sensor - Requirements

As a sensor collection task, we are looking to ensure we have the capability to interact with sensors of a variety of different protocols. This could include Serial Peripheral Interface (SPI), UART, Inter-Integrated Circuit (I2C), and Analog. We also want to include the possibility of attaching camera systems to the Companion. With all of these sensors, we must make some changes to our raspberry pi configuration to allow for these sensors and to provide a work-around to a conflict between I2C and camera operation.

For SPI, we can simply go into the Raspberry Pi configuration and enable SPI such as shown in the figure below.

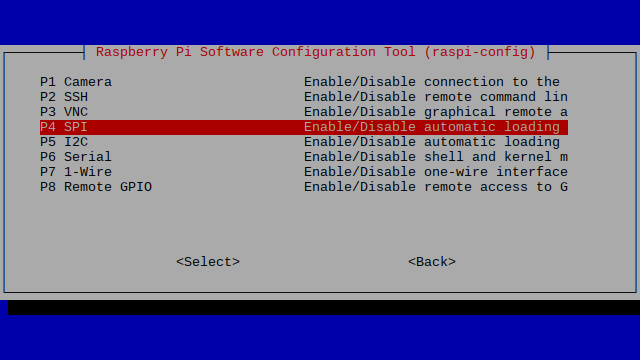


Figure . SPI Configuration (Matt, 2014)

However, the raspberry pi can not use the regular I2C General Purpose Input Output (GPIO) pins while also having the camera enabled. This is due to both pins using the same I2C system to run either component. To correct this, we will add additional I2C buses that are not connected to the Camera Bus.

The full instructions for this are in the Appendix, however the main result is the creation of two additional I2C busses that can be used for sensors while leaving the camera bus available. The last portion, shown in Appendix 2, will be installing any additional modules and libraries that are not present in the default raspberry pi operating system. These mainly include pre-made modules for sensors we have integrated, but also includes some communication protocols such as smbus2 which we use to interact with I2C sensors.

### Sensor - Initialization

To initialize our Sensor Task, we are focused on checking against a list of potential sensors to identify which is connected to the companion. If it detects a sensor, it will perform that sensors initialization procedure and upon success will add that sensor to the list of active sensors from which to collect data. After identifying all connected sensors, the task will then generate an initial header array to send to the data logging task which is appended to the header from the MAVLink task to create a total array of all data points being logged.

### Sensor - Normal Operation

During normal operation, the task will begin polling each sensor for its respective data. After it has finished a cycle of polling for data, it will update its internal data array with all newly received data and have it readily available for the data logging task to record. Some sensors have longer delays between data availability, this means that, much like the MAVLink data arrays, there will be rows of data where some sensors may have the same data repeated.

### Sensor – Full Code

The full code is available in appendix Y.

Unlike the MAVLink Task, the code for the Sensor Task is currently stored in the main file alongside the Data Logging Task. This is due to some required flags that are passed by the MAVLink Module and the update for this has not been implemented yet.

## Task 3: Data Logging

### Data Log- Initialization

To initialize our data logging we need to create the initial log file where all the data from our sensor task and MAVLink task feed in to. To make the data easy to read and analyze, we will be using a Comma-separated values (CSV) text file. A CSV is written where each column of data in a row is separated by some symbol, generally a comma. We will use a semicolon to indicate separation as some of data received from the MAVLink telemetry is sent in arrays with commas, which would create confusion in the parsing of the CSV file in software such as Excel. For the file name, we use a combination of the date, time, and a random number generator. This is to separate multiple tests being conducted around the same time. From there, we insert the created file inside a directory containing all logs within the raspberry pi to aid in organization. With the files created, we can move on to normal operation of the Data Logger.

### Data Log - Normal Operation

During normal operation, we focus on two main potential paths: armed/disconnected and disarmed. This is based on data received from the flight controller to indicate whether the system is armed or disarmed. Like our flight controller, we want to create an individual log file for each arming, aiding us in separating data from different missions. However, if we do not have a MAVLink connection, we will continue as though the system were armed, else the data logger would not collect any information.

When logging data, the data logger will take the data array from each task (Sensor and MAVLink) and combine them into a single larger array which is then fed into the data logger functions to write a semicolon separated row to the CSV file. After successfully writing, the system will pause for roughly 200 milliseconds (ms). We implemented a delay due to the natural speed of the tested vehicle, the rover, being quite slow. Five data samples per second was more than enough to gather information regarding the rovers current environment even while at max speed.

Plans have been made to adjust this data logging rate with respect to vehicle type to account for how quickly the vehicle moves through the environment. For faster vehicles, such as quadcopters and planes, we would aim for likely twice the data rate.

### Data Log – Full Code

The full code is available in appendix Y.

Unlike the MAVLink Task, the code for the Data Logging Task is currently stored in the main file alongside the Data Logging Task. This is due to some required flags that are passed by the MAVLink Module and the update for this has not been implemented yet.

### Armed vs. Disarmed vs. Disconnected Operation

A key redundancy to the companion system is that it can operate and record data in the Data Logger task as long as either the MAVLink task or the Sensor Task is operational. This allows for sensor collection to be continued even if connection to the flight controller is lost, or to continue collection flight controller information from the MAVLink task should the Sensor Task fail. This also provides a method for testing individual tasks without needing the whole system connected and functioning if we want to work on one Task at a time.

# Individual Sensor Modules

Inside our code folder are individual modules that contain the code needed to interact with each sensor. This is to make it easier to navigate the system and to make changes and test individual sensors without affecting the whole system. In this section we will be going through each currently available sensor to do a quick breakdown of its functions, outputs, and methods.

## TMP117 Temperature Sensor

The TMP117 Is a high-accuracy, low-power digital temperature sensor that provides 16-bit temperature results with a resolution of 0.0078 C and accuracy of up to +- 0.1 C (Instruments, Texas, n.d.). The basic operation of the TMP117 is to check the data register of the sensor and determine if there is data available to read. If there is, then the result is read and converted from its binary value to Celsius. This is then made available to the Sensor Bus task to be added to the data array being logged.

Full Code is available in appendix Z.

## SHTC3

The SHTC3 is a Temperature and Humidity Sensor with a +-2% relative humidity and +-0.2 C temperature measurements (Sensirion, n.d.). The operation of this sensor is simple in that we just read the data from the sensor at any point. The result is a six-byte array that contains the information for both temperature and humidity. Both results must then be converted from their binary value to their respective units, Celsius and percent humidity. From there the data can be made available to the sensor bus.

Full code is available in the appendix.

## SCD4X

The SCD4X is a line of sensors that can measure Temperature, Humidity, and CO2. The CO2 measurement is in units known as parts per million (ppm) with a range detection between 0 and 40,000 ppm (Sensirion, n.d.). The measurement function reads the data from the data register and then converts the binary values for temperature and humidity over to their respective units to be used in the data logging portion of the code.

Full code is available in the appendix.

## BME280

The BME280 is a temperature, pressure, and humidity sensor that can be used to calculate altitude (Bosch, n.d.). Due to the complexity of the sensor, we are using an already-made library in integrating our desired I2C bus into it. In the measurement function we use the modules built-in function to check whether data is available. If it is, the system will convert the data into its correct form to be used in the data collection portion of the code.

Full code is available in the appendix.

## Micropressure MPR

The MPR Micropressure Sensor is a high-accuracy pressure sensor that is capable of measuring between 1 and 30 PSI with temperature compensation and calibration (Honeywell, n.d.). In the measurement portion we must read the data then use the 3 bytes which is then converted into a PSI value using the provide constants available on the sensor datasheet to calculate a calibrated result.

Full code is available in the appendix.

## CCS811

The CCS811 is an environmental sensor that can measure CO2 concentrations in parts per million (ppm) and Total Volatile Organic Compounds (TVOC) in parts per billion (ppb) (AMS, n.d.). The basic operation of the sensor is to check the Status register and determine if there is data available to be read. If it is available, then we read the Result Register which contains both the CO2 and TVOC readings as two-byte values. No additional conversion needs to be done as the binary values translate to their individual units.

Full code is available in the appendix.

# Running the Companion Computer

## Running sensor and navio2

When we run the code in a regular IDE such as Thonny or Geany which are already on the Raspberry Pi, we will see several steps take place in terms of setup followed by normal operation.

In the setup, we will see each task initialized in the serial monitor. The companion will simultaneously attempt to connect to the Navio2, determine the available sensors to collect data from, and create the initial CSV file to store data inside on the Companion’s SD card.

In the normal operation, the serial monitor will only display the written lines of data showing the user what is written each time to the SD card. This is generally not visible during operation when you do not have a monitor hooked up but is good for debugging.

Attached inside the MavVehicle code is a portion to send a text packet to Mission Planner which you can view in the Mavlink Inspector. This will be useful for determining whether your system is Armed or Disarmed without needing to connect a monitor to your Companion.

A screenshot of a computer

Description automatically generated

In order for the code to run on boot, we will set up something called a Service to run the main companion code after the raspberry pi finishes booting whenever power is applied. This will allow the system to run automatically without needing constant user interaction; we just turn power on for the vehicle and the system should begin functioning. We have a 30 second window in attempting to connect to the Navio2 which should be enough time for the Navio2 to boot up and connect.

## Setting up service

By setting up a service, we create a method to run the code without needing to be manually started each time. To do so, we must first create a service file in the command prompt.

We create the file using the sudo nano command, in place of “ServiceName” you would put whatever you want it to be called.

**sudo nano /lib/system/system/ServiceName.service**

Inside we will add the following text:

[Unit]

Description=My Sample Service

After=multi-user.target

[Service]

Type=idle

ExecStart=**/usr/bin/python /home/pi/Companion.py**

[Install]

WantedBy=multi-user.target

The bolded portion “ExecStart” is the path to the location of your python file you want to run, if the path location is different you will need to adjust the “/home/pi/Companion.py” portion to include the full path to the file.

You will then save and exit the text editor and finally change the permission of the service to be useable with the “chmod” command

**sudo chmod 644 /lib/system/system/ServiceName.service**

Finally, we will enable the service so it will begin on boot

**sudo systemctl daemon-reload**

**sudo systemctl enable ServiceName.service**

At this point, after rebooting the Raspberry Pi, the service should start on its own as long as there are no errors in the code.

Appendix

## Appendix 1: Companion Main Code

1. # Import Libraries / Modules

2. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Companion.py

3.

4. import os # Operating System Interfaces

5. from picamera import PiCamera # Raspberry Pi Camera

6. from picamera import Color # Raspberry Pi Camera

7. import threading # Thread-Based Parallelism

8. import VL53L1X # Time-of-Flight sensor

9. import time # Time access and conversion module

10. from datetime import datetime # Date and time module

11. import random

12. import TMP117

13. import SHTC3

14. import CCS811

15. import self\_bme280

16. import Micropressure

17. import MavVehicle as Vehicle

18. bus=3

19. startTime=time.time() # Use to mark a starting time for code, used for timing and data recording.

20.

21. # from sensors import TMP117, SHTC3, BME280, CCS811, Micropressure This might be an interesting thing to do where we put all sensors in a folder, and either a folder within that?

22.

23. class SensorBus(threading.Thread):

24. """

25. SensorBus class for managing multiple sensors in a threaded environment.

26.

27. Attributes:

28. - checkArmed (int): Flag to check if the system is armed.

29. - dataFull (list): List to store recorded data.

30. - header (list): List to store header information.

31. - timer1 (float): Timer for marking the current time.

32. - timer2 (float): Timer for marking the end time.

33. - timeTotal (float): Total time taken for sensor readings.

34. - \_running (bool): Indicates whether the thread is running.

35.

36. Methods:

37. - \_\_init\_\_(self): Initialize the SensorBus.

38. - makeHeader(self): Create a header for recorded data based on initialized sensors.

39. - fillData(self): Create a data array to be recorded based on initialized sensors.

40. - run(self): Threaded function for continuously reading sensor data.

41. - checkConn(self): Attempt to initialize all possible sensors.

42. """

43.

44. def \_\_init\_\_(self):

45. """

46. Initialize the SensorBus.

47. """

48. threading.Thread.\_\_init\_\_(self)

49. self.checkArmed = 0

50. self.dataFull = [] # Initialize data array

51. self.header = [] # initialize header array

52. self.makeHeader() # Create header

53. self.\_running = True

54.

55. def makeHeader(self):

56. """

57. Create header for recorded data based on what sensors passed initialization.

58. """

59. self.checkConn() # Attempt to initialize all possible sensors

60. self.header = []

61. self.header.append("Time (s)")

62. if self.tmp.\_running:

63. self.header.append("TMP117 Temp (C)")

64. if self.shtc3.\_running:

65. self.header.append("SHTC3 Temp (C)")

66. self.header.append("SHTC Humidity (%)")

67. if self.micro.\_running:

68. self.header.append("Micro Pressure (psi)")

69. if self.bme.\_running:

70. self.header.append("BME280 Temp (C)")

71. self.header.append("BME280 Press. (Pa)")

72. self.header.append("BME280 Humidity (%)")

73. self.header.append("BME280 Altitude (m)")

74. if self.ccs.\_running:

75. self.header.append("CCS811 tVOC (ppb)")

76. self.header.append("CCS811 CO2 (ppm)")

77.

78. def fillData(self):

79. """

80. Create Data array to be recorded based on what sensors passed initialization.

81. """

82. self.dataFull = []

83. self.dataFull.append((self.timer1 - startTime))

84. if self.tmp.\_running:

85. self.dataFull.append(self.tmp.temp\_c)

86. if self.shtc3.\_running:

87. self.dataFull.append(self.shtc3.temp)

88. self.dataFull.append(self.shtc3.humidity)

89. if self.micro.\_running:

90. self.dataFull.append(self.micro.Pressure)

91. if self.bme.\_running:

92. self.dataFull.append(self.bme.temperature)

93. self.dataFull.append(self.bme.pressure)

94. self.dataFull.append(self.bme.humidity)

95. self.dataFull.append(self.bme.altitude)

96. if self.ccs.\_running:

97. self.dataFull.append(self.ccs.eTVOC)

98. self.dataFull.append(self.ccs.eco2)

99.

100. def run(self):

101. """

102. Threaded function for continuously reading sensor data.

103. """

104. while self.\_running:

105. self.timer1 = time.time() # Mark current time

106. if not navio.\_armed:

107. self.checkArmed = 1

108.

109. if navio.\_armed and self.checkArmed == 1:

110. self.makeHeader()

111. self.checkArmed = 0

112. if navio.\_armed or not navio.\_running:

113. # Read latest available data from all sensors

114. self.micro.read\_pressure() # 10ms

115. self.shtc3.measure() # 10-30 ms

116. self.tmp.tempReading() # 1 ms

117. self.bme.get\_bme280\_data() # 5-10 ms

118. self.ccs.read\_gas() # 1 ms

119.

120. # For timing purposes

121. self.timer2 = time.time()

122. self.timeTotal = self.timer2 - self.timer1

123.

124. self.fillData() # Create data array from newly recorded sensors

125.

126. time.sleep(0.05) # Sleep 50ms

127.

128. def checkConn(self):

129. """

130. Attempt to initialize all possible sensors.

131. """

132. self.tmp = TMP117.TMP117(bus)

133. self.shtc3 = SHTC3.SHTC3(bus)

134. self.bme = self\_bme280.BME280\_Sensor()

135. self.ccs = CCS811.CCS811(bus)

136. self.micro = Micropressure.microPressure(bus)

137.

138.

139. class DataRecordBus(threading.Thread):

140. """

141. DataRecordBus class for managing data recording in a threaded environment.

142.

143. Attributes:

144. - name\_file (str): Name of the log file and directory.

145. - log\_file (str): Path to the log file.

146. - video\_file (str): Path to the video file.

147. - makeNew (bool): Flag to indicate whether a new log should be created.

148. - newCount (int): Counter for creating a new log.

149. - \_running (bool): Indicates whether the thread is running.

150.

151. Methods:

152. - \_\_init\_\_(self): Initialize the DataRecordBus.

153. - makeFile(self): Create a new log file and directory.

154. - combineHeader(self, header1, header2): Combine headers from different sources.

155. - run(self): Threaded function for continuously recording sensor data.

156. - log\_data\_file(self, data): Save data in the log text file.

157. """

158.

159. def \_\_init\_\_(self):

160. """

161. Initialize the DataRecordBus.

162. """

163. threading.Thread.\_\_init\_\_(self)

164. self.makeFile()

165. self.newCount = 0

166. self.\_running = True

167.

168. def makeFile(self):

169. """

170. Create a new log file and directory.

171. """

172. global startTime

173. startTime = time.time()

174. self.name\_file = datetime.now().strftime('%Y%m%d\_%H%M%S') # Create file name

175. random.seed()

176. self.name\_file = self.name\_file + "\_" + str(random.randrange(0, 10001, 2))

177. os.mkdir("/home/pi/Documents/logs/" + self.name\_file) # Create log directory

178. self.log\_file = self.name\_file + "/" + self.name\_file + ".txt" # Create log file

179. self.video\_file = "/home/pi/Documents/logs/" + self.name\_file + "/" + self.name\_file + ".h264" # Create video file

180. self.makeNew = False

181.

182. def combineHeader(self, header1, header2):

183. """

184. Combine headers from different sources.

185.

186. Args:

187. - header1 (list): Header from the first source.

188. - header2 (list): Header from the second source.

189. """

190. self.Header = sensor.header + navio.header

191.

192. def run(self):

193. """

194. Threaded function for continuously recording sensor data.

195. """

196. time.sleep(10)

197. print(sensor.header)

198. print(navio.header)

199. self.combineHeader(sensor.header, navio.header)

200. self.log\_data\_file(self.Header) # Log header data into log file

201. while self.\_running:

202. if navio.\_running:

203. if not navio.\_armed and self.makeNew == False:

204. self.newCount = self.newCount + 1

205. if self.newCount > 5:

206. self.makeNew = True

207. self.newCount = 0

208. print("NEW LOG")

209. if self.makeNew and navio.\_armed:

210. self.makeFile()

211. print(sensor.header)

212. print(navio.header)

213. self.combineHeader(sensor.header, navio.header)

214. self.log\_data\_file(self.Header) # Log header data into log file

215. print("New Log Made")

216. if ((navio.\_running == False) or navio.\_armed):

217. print(sensor.dataFull)

218. self.log\_data\_file(sensor.dataFull + navio.dataFull) # Log current sensor data

219. else:

220. print("System Disarmed and Navio Connected, No Logging")

221. time.sleep(0.5)

222. time.sleep(0.2) # Sleep 200ms

223.

224. def log\_data\_file(self, data):

225. """

226. Save data in the log text file.

227.

228. Args:

229. - data (list): Data to be logged.

230. """

231. file = open(r"/home/pi/Documents/logs/" + self.log\_file, "a") # open log file

232. for L in range(len(data)): # Append data

233. if L == len(data) - 1:

234. file.write(str(data[L]) + "\r\n")

235. else:

236. file.write(str(data[L]) + ";")

237. file.close() # Close file

238.

239.

240.

241.

242. sensor=SensorBus()

243. navio=Vehicle.Vehicle('rover')

244. record=DataRecordBus()

245. sensor.start()

246. navio.start()

247. record.start()

248. restart=0

## Appendix 2: MAVLink Vehicle Code

1. # Import Libraries / Modules

2. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/MavVehicle.py

3.

4. import threading # Thread-Based Parallelism

5. import time # Time access and conversion module

6. from pymavlink import mavutil # Communication for Ardupilot Mavlink Protocol

7.

8. class Vehicle(threading.Thread):

9. """

10. Class representing a vehicle with MAVLink communication capabilities.

11.

12. Parameters:

13. - frame (str): The type of vehicle frame ('plane', 'rover', etc.).

14.

15. Attributes:

16. - exit (bool): Flag indicating whether the vehicle thread should exit.

17. - \_armed (bool): Flag indicating whether the vehicle is armed.

18. - header (list): List to store headers of different MAVLink message types.

19. - dataFull (list): List to store data from MAVLink messages.

20. - frame (str): Type of vehicle frame.

21. - heart (bool): Flag indicating whether a heartbeat signal has been received.

22. - nav (mavutil.mavlink\_connection): MAVLink connection object.

23. - heartVal (mavutil.mavlink\_heartbeat): Heartbeat message received

24. from the system.

25. - \_running (bool): Flag indicating whether the vehicle thread is running.

26.

27. Methods:

28. - \_\_init\_\_(self, frame): Constructor method for the Vehicle class.

29. - attemptConnection(self): Attempts to establish a MAVLink connection with

30. the vehicle.

31. - initializeFields(self): Initializes MAVLink message types and sets

32. message intervals based on the vehicle frame.

33. - createEmpty(self, keys): Creates an empty dictionary with keys

34. initialized to empty strings.

35. - set\_Message\_Interval(self, message\_id, frequency\_Hz): Sets the message

36. interval for a specific MAVLink message type.

37. - getMessage(self): Retrieves and stores a MAVLink message from the

38. vehicle.

39. - createData(self): Creates a list of data from various MAVLink messages

40. if a heartbeat is received and the vehicle is armed.

41. - parseMessage(self): Parses the received MAVLink message and updates

42. relevant attributes based on the message type.

43. - run(self): Main execution method for the vehicle thread,

44. continuously collects and processes MAVLink messages.

45. """

46. def \_\_init\_\_(self,frame):

47. try:

48. self.exit = False

49. self.\_armed = False

50. self.header = []

51. self.dataFull = []

52. self.frame = frame

53. self.heart = False

54.

55. threading.Thread.\_\_init\_\_(self)

56. self.nav = mavutil.mavlink\_connection("/dev/ttyS0", baud=115200)

57. print("Waiting for Heartbeat")

58.

59. self.heartVal = self.nav.wait\_heartbeat(timeout=30)

60. if self.heartVal == None:

61. print("Failure")

62. self.\_running = False

63.

64. else:

65. self.\_running = True

66. print("Heartbeat from system: ",self.nav.target\_system,

67. " and component: ", self.nav.target\_component)

68. print(self.heartVal)

69. self.initializeFields()

70.

71. except:

72. self.\_running = False

73.

74. def attemptConnection(self):

75. """

76. Attempts to establish a MAVLink connection with the vehicle.

77.

78. If successful, initializes necessary attributes and starts MAVLink

79. message streams.

80. """

81. try:

82. self.nav = mavutil.mavlink\_connection("/dev/ttyS0", baud=115200)

83. print("Waiting for Heartbeat")

84. self.heartVal = self.nav.wait\_heartbeat(timeout=30)

85. if self.heartVal == None:

86. print("Failure to find Heartbeat")

87. self.\_running = False

88.

89. else:

90. self.\_running = True

91. print("Heartbeat from system: ",self.nav.target\_system,

92. " and component: ", self.nav.target\_component)

93. print(self.heartVal)

94. self.initializeFields()

95. except:

96. self.\_running = False

97. print("Mavlink Error in Attempting Connection")

98.

99. def initializeFields(self):

100. """

101. Initializes MAVLink message types and sets message intervals based

102. on the vehicle frame.

103. """

104.

105. print("All Stream Halted")

106.

107. # Do Plane Initialization

108. if self.frame == 'plane' or self.frame == 'rover':

109.

110. # Set VFR HUD

111. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_VFR\_HUD,1)

112. print("VFR HUD Stream Started")

113. self.VFR\_HUD\_HEADER=['mavpackettype', 'airspeed', 'groundspeed',

114. 'heading', 'throttle', 'alt', 'climb']

115. self.VFR\_HUD=self.createEmpty(self.VFR\_HUD\_HEADER)

116. self.header = self.header + self.VFR\_HUD\_HEADER

117.

118. if self.frame == 'plane':

119. #Set POSITION\_TARGET\_GLOBAL\_INT

120. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_POSITION\_TARGET\_GLOBAL\_INT,1)

121. print("POSITION\_TARGET\_GLOBAL\_INT Stream Started")

122. self.POSITION\_TARGET\_GLOBAL\_INT\_HEADER=['mavpackettype',

123. 'time\_boot\_ms', 'coordinate\_frame', 'type\_mask',

124. 'lat\_int', 'lon\_int', 'alt', 'vx', 'vy', 'vz', 'afx',

125. 'afy','afz', 'yaw', 'yaw\_rate']

126. self.POSITION\_TARGET\_GLOBAL\_INT=self.createEmpty(self.POSITION\_TARGET\_GLOBAL\_INT\_HEADER)

127. self.header = self.header + self.POSITION\_TARGET\_GLOBAL\_INT\_HEADER

128.

129. # Set AHRS

130. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_AHRS,1)

131. print("AHRS Stream Started")

132. self.AHRS\_HEADER=['mavpackettype', 'omegaIx', 'omegaIy',

133. 'omegaIz', 'accel\_weight',

134. 'renorm\_val', 'error\_rp', 'error\_yaw']

135. self.AHRS=self.createEmpty(self.AHRS\_HEADER)

136. self.header = self.header + self.AHRS\_HEADER

137.

138. # Set GPS\_RAW\_INT

139. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_GPS\_RAW\_INT,1)

140. print("GPS\_RAW\_INT Stream Started")

141. self.GPS\_RAW\_INT\_HEADER=['mavpackettype', 'time\_usec', 'fix\_type',

142. 'lat', 'lon', 'alt','eph', 'epv', 'vel',

143. 'cog', 'satellites\_visible']

144. self.GPS\_RAW\_INT=self.createEmpty(self.GPS\_RAW\_INT\_HEADER)

145. self.header = self.header + self.GPS\_RAW\_INT\_HEADER

146.

147. # Set RC\_CHANNELS

148. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_RC\_CHANNELS,1)

149. print("RC\_CHANNELS Stream Started")

150. self.RC\_CHANNELS\_HEADER=['mavpackettype', 'time\_boot\_ms', 'chancount', 'chan1\_raw',

151. 'chan2\_raw', 'chan3\_raw', 'chan4\_raw', 'chan5\_raw', 'chan6\_raw',

152. 'chan7\_raw', 'chan8\_raw', 'chan9\_raw', 'chan10\_raw', 'chan11\_raw',

153. 'chan12\_raw', 'chan13\_raw', 'chan14\_raw', 'chan15\_raw', 'chan16\_raw',

154. 'chan17\_raw', 'chan18\_raw', 'rssi']

155. self.RC\_CHANNELS=self.createEmpty(self.RC\_CHANNELS\_HEADER)

156. self.header = self.header + self.RC\_CHANNELS\_HEADER

157.

158. # Set MISSION\_CURRENT

159. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_MISSION\_CURRENT,1)

160. print("MISSION\_CURRENT Stream Started")

161. self.MISSION\_CURRENT\_HEADER=['mavpackettype', 'seq']

162. self.MISSION\_CURRENT=self.createEmpty(self.MISSION\_CURRENT\_HEADER)

163. self.header = self.header + self.MISSION\_CURRENT\_HEADER

164.

165. if self.frame == 'plane':

166. # Set NAV\_CONTROLLER\_OUTPUT

167. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_NAV\_CONTROLLER\_OUTPUT,1)

168. print("NAV\_CONTROLLER\_OUTPUT Stream Started")

169. self.NAV\_CONTROLLER\_OUTPUT\_HEADER=['mavpackettype', 'nav\_roll', 'nav\_pitch', 'nav\_bearing',

170. 'target\_bearing', 'wp\_dist', 'alt\_error', 'aspd\_error', 'xtrack\_error']

171. self.NAV\_CONTROLLER\_OUTPUT=self.createEmpty(self.NAV\_CONTROLLER\_OUTPUT\_HEADER)

172. self.header = self.header + self.NAV\_CONTROLLER\_OUTPUT\_HEADER

173.

174. # Set SYS\_STATUS

175. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_SYS\_STATUS,1)

176. print("SYS\_STATUS Stream Started")

177. self.SYS\_STATUS\_HEADER=['mavpackettype', 'onboard\_control\_sensors\_present', 'onboard\_control\_sensors\_enabled',

178. 'onboard\_control\_sensors\_health', 'load', 'voltage\_battery', 'current\_battery',

179. 'battery\_remaining', 'drop\_rate\_comm', 'errors\_comm', 'errors\_count1', 'errors\_count2',

180. 'errors\_count3', 'errors\_count4']

181. self.SYS\_STATUS=self.createEmpty(self.SYS\_STATUS\_HEADER)

182. self.header = self.header + self.SYS\_STATUS\_HEADER

183.

184. # Set GLOBAL\_POSITION\_INT

185. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_GLOBAL\_POSITION\_INT,1)

186. print("GLOBAL\_POSITION\_INT Stream Started")

187. self.GLOBAL\_POSITION\_INT\_HEADER=['mavpackettype', 'time\_boot\_ms', 'lat', 'lon', 'alt',

188. 'relative\_alt', 'vx', 'vy', 'vz', 'hdg']

189. self.GLOBAL\_POSITION\_INT=self.createEmpty(self.GLOBAL\_POSITION\_INT\_HEADER)

190. self.header = self.header + self.GLOBAL\_POSITION\_INT\_HEADER

191.

192. # Set RAW\_IMU

193. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_RAW\_IMU,1)

194. print("RAW\_IMU Stream Started")

195. self.RAW\_IMU\_HEADER=['mavpackettype', 'time\_usec', 'xacc', 'yacc', 'zacc', 'xgyro', 'ygyro',

196. 'zgyro', 'xmag', 'ymag', 'zmag']

197. self.RAW\_IMU=self.createEmpty(self.RAW\_IMU\_HEADER)

198. self.header = self.header + self.RAW\_IMU\_HEADER

199.

200. # Set BATTERY\_STATUS

201. self.set\_Message\_Interval(mavutil.mavlink.MAVLINK\_MSG\_ID\_BATTERY\_STATUS,1)

202. self.BATTERY\_STATUS\_HEADER=['mavpackettype', 'id', 'battery\_function', 'type', 'temperature',

203. 'voltages', 'current\_battery', 'current\_consumed', 'energy\_consumed',

204. 'battery\_remaining']

205. self.BATTERY\_STATUS=self.createEmpty(self.BATTERY\_STATUS\_HEADER)

206. print("BATTERY\_STATUS Stream Started")

207. self.header = self.header + self.BATTERY\_STATUS\_HEADER

208.

209. self.HEARTBEAT\_HEADER=['mavpackettype','type','autopilot','base\_mode','custom\_mode','system\_status','mavlink\_version']

210. self.HEARTBEAT=self.createEmpty(self.HEARTBEAT\_HEADER)

211. print("Heartbeat Stream Started")

212. self.header = self.header + self.HEARTBEAT\_HEADER

213.

214. def createEmpty(self, keys):

215. """

216. Creates an empty dictionary with keys initialized to empty strings.

217.

218. Parameters:

219. - keys (list): List of keys for the dictionary.

220.

221. Returns:

222. - dict: Empty dictionary with keys.

223. """

224. return {keys[x]:'' for x in range(len(keys))}

225.

226. def set\_Message\_Interval(self, message\_id, frequency\_Hz):

227. """

228. Sets the message interval for a specific MAVLink message type.

229.

230. Parameters:

231. - message\_id (int): ID of the MAVLink message.

232. - frequency\_Hz (float): Desired frequency of the message in Hertz.

233. """

234. if self.\_running:

235. self.nav.mav.command\_long\_send(1,self.nav.target\_component,

236. mavutil.mavlink.MAV\_CMD\_SET\_MESSAGE\_INTERVAL,

237. 0,

238. message\_id,

239. 1e6/frequency\_Hz,

240. 0,0,0,0,

241. 2)

242. time.sleep(0.1)

243.

244. def getMessage(self):

245. """

246. Retrieves and stores a MAVLink message from the vehicle.

247. """

248. if self.\_running:

249. self.msg=self.nav.recv\_match().to\_dict()

250. #print(self.msg)

251.

252. def createData(self):

253. """

254. Creates a list of data from various MAVLink messages if a heartbeat is received and the vehicle is armed.

255. """

256. if self.heart:

257. if self.\_armed:

258. self.dataFull = list(self.VFR\_HUD.values())

259. if self.frame=='plane':

260. self.dataFull= self.dataFull + list(self.POSITION\_TARGET\_GLOBAL\_INT.values())

261. self.dataFull=self.dataFull +list(self.AHRS.values())+list(self.GPS\_RAW\_INT.values()) + list(self.RC\_CHANNELS.values()) + list(self.MISSION\_CURRENT.values())

262. if self.frame == 'plane':

263. self.dataFull = self.dataFull + list(self.NAV\_CONTROLLER\_OUTPUT.values())

264. self.dataFull = self.dataFull + list(self.SYS\_STATUS.values()) + list(self.GLOBAL\_POSITION\_INT.values()) + list(self.RAW\_IMU.values())+list(self.BATTERY\_STATUS.values())+list(self.HEARTBEAT.values())

265. #print(self.dataFull)

266.

267. def parseMessage(self):

268. """

269. Parses the received MAVLink message and updates relevant attributes based on the message type.

270. """

271. if self.\_running:

272. if self.msg['mavpackettype']=='VFR\_HUD':

273. #print(self.msg)

274. self.VFR\_HUD=self.msg

275. elif self.msg['mavpackettype']=='POSITION\_TARGET\_GLOBAL\_INT':

276. #print(self.msg)

277. self.POSITION\_TARGET\_GLOBAL\_INT=self.msg

278. elif self.msg['mavpackettype']=='AHRS':

279. #print(self.msg)

280. self.AHRS=self.msg

281. elif self.msg['mavpackettype']=='GPS\_RAW\_INT':

282. #print(self.msg)

283. self.GPS\_RAW\_INT=self.msg

284. elif self.msg['mavpackettype']=='RC\_CHANNELS':

285. #print(self.msg)

286. self.RC\_CHANNELS=self.msg

287. elif self.msg['mavpackettype']=='MISSION\_CURRENT':

288. #print(self.msg)

289. self.MISSION\_CURRENT=self.msg

290. elif self.msg['mavpackettype']=='NAV\_CONTROLLER\_OUTPUT':

291. #print(self.msg)

292. self.NAV\_CONTROLLER\_OUTPUT=self.msg

293. elif self.msg['mavpackettype']=='SYS\_STATUS':

294. #print(self.msg)

295. self.SYS\_STATUS=self.msg

296. elif self.msg['mavpackettype']=='GLOBAL\_POSITION\_INT':

297. #print(self.msg)

298. self.GLOBAL\_POSITION\_INT=self.msg

299. elif self.msg['mavpackettype']=='RAW\_IMU':

300. #print(self.msg)

301. self.RAW\_IMU=self.msg

302. elif self.msg['mavpackettype']=='BATTERY\_STATUS':

303. #print(self.msg)

304. self.BATTERY\_STATUS=self.msg

305. elif self.msg['mavpackettype']=='HEARTBEAT':

306. #print(self.msg)

307. self.HEARTBEAT=self.msg

308. self.heart=True

309. if self.HEARTBEAT['type'] == 10 and self.heart== True and self.HEARTBEAT['base\_mode']>128:

310. self.\_armed=True

311. #print("Armed")

312. elif self.HEARTBEAT['type'] == 10 and self.heart== True and self.HEARTBEAT['base\_mode']<128:

313. self.\_armed=False

314. #print("Disarmed")

315. else:

316. pass

317. #print(self.\_armed)

318. else:

319. pass

320.

321. def run(self):

322. """

323. Main execution method for the vehicle thread.

324.

325. Continuously collects and processes MAVLink messages while the thread is running.

326. """

327. while not self.exit:

328. if self.\_running:

329. nextTime=time.time()

330. while self.\_running:

331. curTime=time.time()

332. if curTime-nextTime > 5:

333. if self.\_armed:

334. self.nav.mav.statustext\_send(mavutil.mavlink.MAV\_SEVERITY\_NOTICE,"System Collecting Data".encode())

335. #print("Hellooo")

336. else:

337. self.nav.mav.statustext\_send(mavutil.mavlink.MAV\_SEVERITY\_NOTICE,"System Not Collecting Data".encode())

338.

339. nextTime=time.time()

340. #time.sleep(.01)

341. #print(navio.nav.messages)

342. try:

343. self.getMessage()

344. except:

345. pass

346. self.parseMessage()

347. self.createData()

348. if not self.\_running:

349. time.sleep(60)

350. self.attemptConnection()

## Appendix 3: I2C Bus Creation

1. We will open the configuration file using the command: **sudo nano /boot/config.txt**
2. This will open a text file with a large number of options. Use the arrow keys to move down until you get to a portion with the following lines shown in the figure below:   
   A computer screen with white text

   Description automatically generated
3. We want to make sure that the **dtparam** options here are as follows:  
   **dtparam=i2c\_arm=on  
   dtparam=i2s=on  
   dtparam=spi=on**You will remove the “#” to uncomment the lines.
4. Next, we are going to add in our additional I2C busses by adding the following line  
   **dtoverlay=i2c-gpio,bus=4,i2c\_gpio\_delay\_us=1,i2c\_gpio\_sda=23,i2c\_gpio\_scl=24**This will create a new open bus “bus=4” using GPIO Pins 23 and 24 as the SDA and SCL of your I2C bus.  
   A computer screen with white text

   Description automatically generated  
   You can not use numbers 2 or lower.  And you must start at the highest bus you will add and work your way down.   
     
   As we are creating two buses for extra ports to use, we will now create a new line under the previous and create our next bus, bus #3  
   **dtoverlay=i2c-gpio,bus=3,i2c\_gpio\_delay\_us=1,i2c\_gpio\_sda=10,i2c\_gpio\_scl=9**  
   This will now have an additional bus on GPIO Pins 10 and 9.
5. Now that we are finished, you can close the file using “CTRL+X” and then you will press “Y” to save the modified buffer and then ENTER to save it without changing the name of the file.
6. The next time we reboot, we will have two additional I2C buses active to work with for our system.
7. You can add additional busses by using the same format and changing the following items
   1. bus=”3” , Otherwise you can keep incrementing the numbers of buses, but the first one should always be your highest number and then you work your way down. Do not use 2 or lower.
   2. i2c\_gpio\_sda/scl = “23/24” you are using available GPIO pins. There are many.
   3. You can find pins by using the command “pinout”  
      A screenshot of a computer

      Description automatically generated

## Appendix 4: Install Libraries

We need to download all the libraries used. Now many of them are built in such as Time, but others specific to sensors we must download. To do so, open up command prompt. We will use the pip installer, type the following inputs and hit enter for each line. This will download the library to your system.

1. sudo pip install VL53L1X
2. sudo pip install smbus2
3. sudo pip install pimoroni-bme280
4. sudo python -m pip install --upgrade pymavlink

## Appendix 5: Sensor Task Code

1. class SensorBus(threading.Thread):

2. """

3. SensorBus class for managing multiple sensors in a threaded environment.

4.

5. Attributes:

6. - checkArmed (int): Flag to check if the system is armed.

7. - dataFull (list): List to store recorded data.

8. - header (list): List to store header information.

9. - timer1 (float): Timer for marking the current time.

10. - timer2 (float): Timer for marking the end time.

11. - timeTotal (float): Total time taken for sensor readings.

12. - \_running (bool): Indicates whether the thread is running.

13.

14. Methods:

15. - \_\_init\_\_(self): Initialize the SensorBus.

16. - makeHeader(self): Create a header for recorded data based on initialized sensors.

17. - fillData(self): Create a data array to be recorded based on initialized sensors.

18. - run(self): Threaded function for continuously reading sensor data.

19. - checkConn(self): Attempt to initialize all possible sensors.

20. """

21.

22. def \_\_init\_\_(self):

23. """

24. Initialize the SensorBus.

25. """

26. threading.Thread.\_\_init\_\_(self)

27. self.checkArmed = 0

28. self.dataFull = [] # Initialize data array

29. self.header = [] # initialize header array

30. self.makeHeader() # Create header

31. self.\_running = True

32.

33. def makeHeader(self):

34. """

35. Create header for recorded data based on what sensors passed initialization.

36. """

37. self.checkConn() # Attempt to initialize all possible sensors

38. self.header = []

39. self.header.append("Time (s)")

40. if self.tmp.\_running:

41. self.header.append("TMP117 Temp (C)")

42. if self.shtc3.\_running:

43. self.header.append("SHTC3 Temp (C)")

44. self.header.append("SHTC Humidity (%)")

45. if self.micro.\_running:

46. self.header.append("Micro Pressure (psi)")

47. if self.bme.\_running:

48. self.header.append("BME280 Temp (C)")

49. self.header.append("BME280 Press. (Pa)")

50. self.header.append("BME280 Humidity (%)")

51. self.header.append("BME280 Altitude (m)")

52. if self.ccs.\_running:

53. self.header.append("CCS811 tVOC (ppb)")

54. self.header.append("CCS811 CO2 (ppm)")

55.

56. def fillData(self):

57. """

58. Create Data array to be recorded based on what sensors passed initialization.

59. """

60. self.dataFull = []

61. self.dataFull.append((self.timer1 - startTime))

62. if self.tmp.\_running:

63. self.dataFull.append(self.tmp.temp\_c)

64. if self.shtc3.\_running:

65. self.dataFull.append(self.shtc3.temp)

66. self.dataFull.append(self.shtc3.humidity)

67. if self.micro.\_running:

68. self.dataFull.append(self.micro.Pressure)

69. if self.bme.\_running:

70. self.dataFull.append(self.bme.temperature)

71. self.dataFull.append(self.bme.pressure)

72. self.dataFull.append(self.bme.humidity)

73. self.dataFull.append(self.bme.altitude)

74. if self.ccs.\_running:

75. self.dataFull.append(self.ccs.eTVOC)

76. self.dataFull.append(self.ccs.eco2)

77.

78. def run(self):

79. """

80. Threaded function for continuously reading sensor data.

81. """

82. while self.\_running:

83. self.timer1 = time.time() # Mark current time

84. if not navio.\_armed:

85. self.checkArmed = 1

86.

87. if navio.\_armed and self.checkArmed == 1:

88. self.makeHeader()

89. self.checkArmed = 0

90. if navio.\_armed or not navio.\_running:

91. # Read latest available data from all sensors

92. self.micro.read\_pressure() # 10ms

93. self.shtc3.measure() # 10-30 ms

94. self.tmp.tempReading() # 1 ms

95. self.bme.get\_bme280\_data() # 5-10 ms

96. self.ccs.read\_gas() # 1 ms

97.

98. # For timing purposes

99. self.timer2 = time.time()

100. self.timeTotal = self.timer2 - self.timer1

101.

102. self.fillData() # Create data array from newly recorded sensors

103.

104. time.sleep(0.05) # Sleep 50ms

105.

106. def checkConn(self):

107. """

108. Attempt to initialize all possible sensors.

109. """

110. self.tmp = TMP117.TMP117(bus)

111. self.shtc3 = SHTC3.SHTC3(bus)

112. self.bme = self\_bme280.BME280\_Sensor()

113. self.ccs = CCS811.CCS811(bus)

114. self.micro = Micropressure.microPressure(bus)

## Appendix 6: Data Logging Class Code

1. class DataRecordBus(threading.Thread):

2. """

3. DataRecordBus class for managing data recording in a threaded environment.

4.

5. Attributes:

6. - name\_file (str): Name of the log file and directory.

7. - log\_file (str): Path to the log file.

8. - video\_file (str): Path to the video file.

9. - makeNew (bool): Flag to indicate whether a new log should be created.

10. - newCount (int): Counter for creating a new log.

11. - \_running (bool): Indicates whether the thread is running.

12.

13. Methods:

14. - \_\_init\_\_(self): Initialize the DataRecordBus.

15. - makeFile(self): Create a new log file and directory.

16. - combineHeader(self, header1, header2): Combine headers from different sources.

17. - run(self): Threaded function for continuously recording sensor data.

18. - log\_data\_file(self, data): Save data in the log text file.

19. """

20.

21. def \_\_init\_\_(self):

22. """

23. Initialize the DataRecordBus.

24. """

25. threading.Thread.\_\_init\_\_(self)

26. self.makeFile()

27. self.newCount = 0

28. self.\_running = True

29.

30. def makeFile(self):

31. """

32. Create a new log file and directory.

33. """

34. global startTime

35. startTime = time.time()

36. self.name\_file = datetime.now().strftime('%Y%m%d\_%H%M%S') # Create file name

37. random.seed()

38. self.name\_file = self.name\_file + "\_" + str(random.randrange(0, 10001, 2))

39. os.mkdir("/home/pi/Documents/logs/" + self.name\_file) # Create log directory

40. self.log\_file = self.name\_file + "/" + self.name\_file + ".txt" # Create log file

41. self.video\_file = "/home/pi/Documents/logs/" + self.name\_file + "/" + self.name\_file + ".h264" # Create video file

42. self.makeNew = False

43.

44. def combineHeader(self, header1, header2):

45. """

46. Combine headers from different sources.

47.

48. Args:

49. - header1 (list): Header from the first source.

50. - header2 (list): Header from the second source.

51. """

52. self.Header = sensor.header + navio.header

53.

54. def run(self):

55. """

56. Threaded function for continuously recording sensor data.

57. """

58. time.sleep(10)

59. print(sensor.header)

60. print(navio.header)

61. self.combineHeader(sensor.header, navio.header)

62. self.log\_data\_file(self.Header) # Log header data into log file

63. while self.\_running:

64. if navio.\_running:

65. if not navio.\_armed and self.makeNew == False:

66. self.newCount = self.newCount + 1

67. if self.newCount > 5:

68. self.makeNew = True

69. self.newCount = 0

70. print("NEW LOG")

71. if self.makeNew and navio.\_armed:

72. self.makeFile()

73. print(sensor.header)

74. print(navio.header)

75. self.combineHeader(sensor.header, navio.header)

76. self.log\_data\_file(self.Header) # Log header data into log file

77. print("New Log Made")

78. if ((navio.\_running == False) or navio.\_armed):

79. print(sensor.dataFull)

80. self.log\_data\_file(sensor.dataFull + navio.dataFull) # Log current sensor data

81. else:

82. print("System Disarmed and Navio Connected, No Logging")

83. time.sleep(0.5)

84. time.sleep(0.2) # Sleep 200ms

85.

86. def log\_data\_file(self, data):

87. """

88. Save data in the log text file.

89.

90. Args:

91. - data (list): Data to be logged.

92. """

93. file = open(r"/home/pi/Documents/logs/" + self.log\_file, "a") # open log file

94. for L in range(len(data)): # Append data

95. if L == len(data) - 1:

96. file.write(str(data[L]) + "\r\n")

97. else:

98. file.write(str(data[L]) + ";")

99. file.close() # Close file

100.

## Appendix 7: TMP117 Sensor Code

1. # TMP117 Temperature Sensor

2. # Sparkfun Breakout Board

3. # Datasheet: https://www.ti.com/lit/ds/symlink/tmp117.pdf?ts=1701791984778&ref\_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FTMP117

4. # Github Link: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Sensors/TMP117.py

5. from smbus2 import SMBus # I2C Package

6. from smbus2 import i2c\_msg # I2C Package

7.

8. # Constants

9. DEVICE\_ADDRESS = 0x48 # Default Device Address

10. TEMP\_RESULT = 0x00 # Temperature result register

11. CONFIGURATION = 0x01 # Configuration register

12. THIGH\_LIMIT = 0x02 # Temperature high limit register

13. TLOW\_LIMIT = 0x03 # Temperature low limit register

14. TEMP\_OFFSET = 0x07 # Temperature offset register

15. DEVICE\_ID = 0x0F # Device ID register

16.

17. class TMP117:

18. """

19. Class representing a TMP117 temperature sensor.

20.

21. Parameters:

22. - busID (int): The bus ID for communication with the sensor.

23.

24. Attributes:

25. - busID (int): The bus ID for communication.

26. - bus (SMBus): The SMBus object for communication.

27. - tmp117\_addr (int): Device address for the TMP117 sensor.

28. - tmp117\_reg\_temp (int): Register address for temperature readings.

29. - tmp117\_reg\_config (int): Register address for configuration settings.

30. - dataHere (bool): Flag indicating whether new data is available for

31. reading.

32. - temp\_c (float): Temperature reading in Celsius.

33. - \_running (bool): Flag indicating whether the sensor initialization

34. succeeded.

35.

36. Methods:

37. - \_\_init\_\_(self, busID): Constructor method for the TMP117 class.

38. - configure(self, conversion\_mode, conversion\_cycle, conversion\_average):

39. Configures TMP117 Sensor.

40. - twos\_comp(self, val, bits): Calculate the two's complement of a value.

41. - dataReady(self): Check if new data is available for reading.

42. - tempReading(self): Read temperature data from the sensor and convert

43. it to Celsius.

44. """

45. def \_\_init\_\_(self, busID = 3, address = DEVICE\_ADDRESS):

46. """

47. Initialize the TMP117 sensor.

48.

49. Parameters:

50. - busID (int): The bus ID for communication with the sensor.

51. """

52. try:

53. self.busID = busID

54. self.bus = SMBus(busID)

55. self.address = address # Device Address

56.

57. self.dataHere = False # Initialize Flag for whether there is data to be read

58. self.temp\_c = 0 # Initialize data output......Removeable?

59.

60. self.bus.read\_byte\_data(self.address, TEMP\_RESULT) # Read the temperature register.

61.

62.

63. self.configure()

64. self.\_running = True # Initialization succeeded

65. # System will take data from this sensor.

66.

67. print("TMP117 Pass") # Let the user know the sensor passed initialization.

68.

69. except:

70. self.\_running = False # Initialization failed, mark as non-functioning.

71. # System will not take data from this sensor.

72.

73. print("TMP117 Fail") # Let the user know the sensor failed initialization.

74.

75. def configure(self, conversion\_mode = 0b00, conversion\_cycle = 0b001,

76. conversion\_average = 0b01):

77. """

78. Configure Sensor.

79.

80. Parameters:

81. - conversion\_mode (binary): Sets conversion Mode.

82. 0b00: Continuous conversion (CC)

83. 0b01: Shutdown (SD)

84. 0b10: Continuous conversion (CC), Same as 00 (reads back = 00)

85. 0b11: One-shot conversion (OS)

86.

87. - Conversion Cycle Bit (binary): Stanby Time between Conversions. There

88. are a lot of option settings in here but we are focused on a cycle

89. time of about 125 ms. More information available on datasheet.

90.

91. - Conversion Averaging Mode(binary): Determines the number of

92. conversion results that are colelcted and averaged before updating

93. the temperature register.The average is an accumulated average and

94. not a running average.

95. 0b00: No averaging

96. 0b01: 8 Averaged conversions

97. 0b10: 32 averaged conversions

98. 0b11: 64 averaged conversions

99. """

100.

101. val=[0,1] # Create Register Variable

102. val[1] = ((conversion\_cycle & 0b001)<<7) \

103. | (conversion\_average<<5) | 0b00000 # LSB of Configuration Register

104. val[0] = (conversion\_mode<<2) | (conversion\_cycle>>1) # MSB of Configuration Register

105.

106. self.bus.write\_i2c\_block\_data(self.address,CONFIGURATION, val) # Write the new configuration to the register.

107.

108. def twos\_comp(self, val, bits):

109. """

110. Calculate the two's complement of a value.

111.

112. Parameters:

113. - val (int): The value to calculate the two's complement for.

114. - bits (int): The number of bits in the representation.

115.

116. Returns:

117. - int: The two's complement of the input value.

118. """

119. if (val & (1 << (bits - 1))) != 0:

120. val = val - (1 << bits)

121. return val

122.

123. def dataReady(self):

124. """

125. Check if new data is available for reading.

126.

127. Updates the dataHere attribute based on the configuration register.

128. """

129. val = self.bus.read\_i2c\_block\_data(self.address,

130. TEMP\_RESULT, 2) # Read the Configuration Register.

131.

132. if val[0] & 0b100000: # Check if Bit 13 of the configuration register is "1".

133. self.dataHere = True # If Bit 13 == 1, New data is available.

134. else:

135. self.dataHere = False # Else, new data is not available.

136.

137. def tempReading(self):

138. """

139. Read temperature data from the sensor and convert it to Celsius.

140.

141. If the sensor has been initialized successfully and new data is

142. available, read the temperature register, perform necessary bit

143. operations, and convert the result to Celsius.

144. """

145. if self.\_running: # Only reads data if system passed initialization.

146. self.dataReady() # Check if data is available to be read.

147. if self.dataHere: # If data is available, read from temperature register.

148. val = self.bus.read\_i2c\_block\_data(self.address,

149. TEMP\_RESULT, 2) # Read temperature data register and store the recieved 2 bytes.

150. self.temp\_c = (val[0] << 8) | (val[1] >> 0) # Combine MSB and LSB bytes of the data.

151. self.temp\_c = self.twos\_comp(self.temp\_c, 16) # Calculate Twos complement of combined data.

152.

153. self.temp\_c = self.temp\_c \* 0.0078125 # Convert to celsius.

## Appendix 8: SHTC3 Sensor Code

1. # SHTC3 Sensor

2. # Sparkfun Breakout Board

3. # Datasheet: https://www.sensirion.com/media/documents/643F9C8E/63A5A436/Datasheet\_SHTC3.pdf

4. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Sensors/SHTC3.py

5. from smbus2 import SMBus # I2C Package

6. from smbus2 import i2c\_msg # I2C Package

7.

8. # Constants

9. DEVICE\_ADDRESS = 0x70

10. SLEEP = [0xB0, 0x98]

11. WAKEUP = [0x35, 0x17]

12. NORMAL\_MODE\_RH\_FIRST\_CLOCK\_STRETCH = [0x5C, 0x24]

13.

14. class SHTC3:

15. """

16. SHTC3 Temperature and Humidity Sensor class.

17.

18. Parameters:

19. - busID (int): The bus ID for communication with the sensor.

20. - address (int): The Address of the sensor

21.

22. Attributes:

23. - busID (int): The bus ID for communication.

24. - bus (SMBus): The SMBus instance for communication.

25. - addr (int): Device address (default: 0x70).

26. - temp (float): Temperature value in Celsius.

27. - humidity (float): Humidity value in percentage.

28. - \_running (bool): Indicates whether the sensor is successfully

29. initialized.

30.

31. Methods:

32. - \_\_init\_\_(self, busID): Initialize the SHTC3 sensor.

33. - measure(self): Read temperature and humidity data from the sensor.

34. """

35.

36. def \_\_init\_\_(self, busID = 3, address = DEVICE\_ADDRESS):

37. """

38. Initialize the SHTC3 sensor.

39.

40. Parameters:

41. - busID (int): The bus ID for communication with the sensor.

42. """

43. try:

44. self.busID = busID

45. self.bus = SMBus(self.busID)

46. self.address = address

47.

48. self.temp = 0

49. self.humidity = 0

50.

51.

52.

53. self.\_running = True

54. self.measure()

55. print("SHTC3 Pass")

56.

57. except:

58. self.\_running = False

59. print("SHTC3 Fail")

60.

61. def measure(self):

62. """

63. Read temperature and humidity data from the sensor.

64. """

65. if self.\_running:

66. msg = i2c\_msg.write(self.address, WAKEUP)

67. self.bus.i2c\_rdwr(msg)

68. msg = i2c\_msg.write(self.address,\

69. NORMAL\_MODE\_RH\_FIRST\_CLOCK\_STRETCH)

70. self.bus.i2c\_rdwr(msg)

71. msg = i2c\_msg.read(self.address, 6)

72. self.bus.i2c\_rdwr(msg)

73.

74. data = list(msg) # Load data from command into an array

75.

76. self.temp = data[3] << 8 | data[4] # Combine Temperature Bytes

77. self.temp = (-45) + ((175 \* self.temp) / 65536) # Convert to Celsius Value

78.

79. self.humidity = data[0] << 8 | data[1] # Combine Humidity Bytes

80. self.humidity = 100 \* self.humidity / 65536 # Convert to % Humidity

## Appendix 9: SCD4X Sensor Code

1. # SCD40 CO2, Temp, Humidity

2. # Adafruit Breakout Board

3. # Datasheet: https://sensirion.com/media/documents/48C4B7FB/64C134E7/Sensirion\_SCD4x\_Datasheet.pdf

4. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Sensors/SCD4X.py

5. from smbus2 import SMBus # I2C Package

6. from smbus2 import i2c\_msg # I2C Package

7. import time

8.

9. # Constants

10. DEVICE\_ADDRESS = 0x62

11. START\_PERIODIC\_MEASUREMENT = [0x21, 0xB1]

12. READ\_MEASUREMENT = [0xEC, 0x05]

13. STOP\_PERIODIC\_MEASUREMENT = [0x3F, 0x86]

14. SET\_TEMPERATURE\_OFFSET = [0x24, 0x1D]

15. GET\_TEMPERATURE\_OFFSET = [0x23, 0x18]

16. SET\_SENSOR\_ALTITUDE = [0x24, 0x27]

17. GET\_SENSOR\_ALTITUDE = [0x23, 0x22]

18. SET\_AMBIENT\_PRESSURE = [0xE0, 0x00]

19. PERFORM\_FORCED\_RECALIBRATION = [0x36, 0x2F]

20. SET\_AUTOMATIC\_SELF\_CALIBRATION\_ENABLED = [0x24, 0x16]

21. GET\_AUTOMATIC\_SELF\_CALIBRATION\_ENABLED = [0x23, 0x13]

22. START\_LOW\_POWER\_PERIODIC\_MEASUREMENT = [0x21, 0xAC]

23. GET\_DATA\_READY\_STATUS = [0xE4, 0xB8]

24. PERSIST\_SETTINGS = [0x36, 0x15]

25. GET\_SERIAL\_NUMBER = [0x36, 0x82]

26. PERFORM\_SELF\_TEST = [0x36, 0x39]

27. PERFORM\_FACTORY\_RESET = [0x36, 0x32]

28. REINIT = [0x36, 0x46]

29. MEASURE\_SINGLE\_SHOT = [0x21, 0x9D]

30. MEASURE\_SINGLE\_SHOT\_RHT\_ONLY = [0x21, 0x96]

31.

32. class SCD4X\_CO2:

33. """

34. SCD4X\_CO2 class for interfacing with the SCD4X CO2 sensor.

35.

36. Parameters:

37. - busID (int): The bus ID for communication with the sensor.

38.

39. Attributes:

40. - busID (int): The bus ID for communication.

41. - bus (SMBus): The SMBus instance for communication.

42. - address (int): Device address (default: 0x62).

43. - CO2 (int): CO2 concentration data.

44. - Temp (float): Temperature data in Celsius.

45. - RH (float): Relative Humidity data in percentage.

46. - \_running (bool): Indicates whether the sensor is successfully

47. initialized.

48.

49. Methods:

50. - \_\_init\_\_(self, busID): Initialize the SCD4X\_CO2 sensor.

51. - startPeriodicMeasurement: Start measurement polling.

52. - getReadyStatus: Read the GET\_DATA\_READY\_STATUS Register to determine

53. if the first 11 bits of the first word are greater than 0.

54. - readSensor(self): Read CO2, temperature, and humidity data from the

55. sensor.

56. """

57.

58. def \_\_init\_\_(self, busID=3,address=DEVICE\_ADDRESS):

59. """

60. Initialize the SCD4X\_CO2 sensor.

61.

62. Parameters:

63. - busID (int): The bus ID for communication with the sensor.

64. """

65. self.busID = busID

66. self.bus = SMBus(self.busID)

67. self.address = address

68.

69. time.sleep(1)

70.

71. self.startPeriodicMeasurement()

72. self.\_running = True

73.

74. def startPeriodicMeasurement(self):

75. """

76. Starts periodic measurements. The data will become available roughly

77. every 5 seconds.

78. """

79. msg = i2c\_msg.write(self.address, START\_PERIODIC\_MEASUREMENT)

80. self.bus.i2c\_rdwr(msg)

81.

82. def getReadyStatus(self):

83. """

84. Read the GET\_DATA\_READY\_STATUS Register to determine if the first 11

85. bits of the first word are greater than 0. If they are, that means

86. that data is ready to be read.

87.

88. Returns:

89. - bool: If the data is available to be read, then True.

90. """

91. msg = i2c\_msg.write(self.address, GET\_DATA\_READY\_STATUS)

92. read = i2c\_msg.read(self.address, 3)

93. self.bus.i2c\_rdwr(msg)

94.

95. data = list(read)

96.

97. check= data[0] << 8 or data[1]

98. if (check & 0b0000011111111111) >0:

99. flag = True

100.

101. return flag

102.

103. def readSensor(self):

104. """

105. Read CO2, temperature, and humidity data from the sensor.

106. """

107. if self.\_running:

108. dataAvailable = self.getReadyStatus()

109. if dataAvailable:

110.

111. write = i2c\_msg.write(self.address, READ\_MEASUREMENT)

112. read = i2c\_msg.read(self.address, 9)

113. self.bus.i2c\_rdwr(write, read)

114.

115. time.sleep(0.01)

116.

117. data = list(read)

118.

119. self.CO2 = data[0] << 8 or data[1]

120. self.Temp = data[3] << 8 or data[4]

121. self.RH = data[6] << 8 or data[7]

122. self.Temp = -45.0 + 175 \* self.Temp / 65536

123. self.RH = 100.0 \* self.RH / 65536

## Appendix 10: Micro Pressure Sensor Code

1. # MPR Series - MPRLS0025PA00001A Micro pressure Sensor

2. # Sparkfun Breakout Board

3. # Datasheet: https://www.mouser.com/datasheet/2/187/HWSC\_S\_A0016036563\_1-3073392.pdf

4. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Sensors/Micropressure.py

5. from smbus2 import SMBus # I2C Package

6. from smbus2 import i2c\_msg # I2C Package

7. import time

8.

9. #Constants

10. DEVICE\_ADDRESS = 0x18

11. OUTPUT\_COMMAND = [0xAA , 0x00, 0x00]

12.

13. class microPressure:

14. """

15. microPressure class for interfacing with the microPressure sensor.

16.

17. Parameters:

18. - busID (int): The bus ID for communication with the sensor.

19.

20. Attributes:

21. - busID (int): The bus ID for communication.

22. - bus (SMBus): The SMBus instance for communication.

23. - address (int): Device address (default: 0x18).

24.

25. - Pressure (float): Pressure data in psi.

26. - \_running (bool): Indicates whether the sensor is successfully

27. initialized.

28.

29. Methods:

30. - \_\_init\_\_(self, busID): Initialize the microPressure sensor.

31. - read\_pressure(self): Read pressure data from the sensor.

32. """

33.

34. def \_\_init\_\_(self, busID = 3, address = DEVICE\_ADDRESS):

35. """

36. Initialize the microPressure sensor.

37.

38. Parameters:

39. - busID (int): The bus ID for communication with the sensor.

40. - address (int): The address for the sensor.

41. """

42. try:

43. self.busID = busID

44. self.bus = SMBus(self.busID)

45. self.address = address

46.

47. self.Pressure = 0

48.

49. self.\_running = True

50. print("Micropressure Sensor Passed")

51. except:

52. self.\_running = False

53. print("Micropressure Sensor Failed")

54.

55. def read\_pressure(self):

56. """

57. Read pressure data from the sensor.

58. """

59. if self.\_running:

60. msg = i2c\_msg.write(self.address, OUTPUT\_COMMAND) # Write a message to receive data.

61. self.bus.i2c\_rdwr(msg) # Send Message

62.

63. time.sleep(0.01)

64. val = self.bus.read\_byte(self.address) # Read Address

65.

66. msg = i2c\_msg.read(self.address, 4) # Write message to read 4 bytes from device

67. self.bus.i2c\_rdwr(msg) # Send Message and receive 4 bytes

68. content = list(msg) # Convert message to array

69. outputs = (content[1] << 16) | (content[2] << 8) | content[3] # Combine Data Bytes

70.

71. # 10% to 90% calibration

72. output\_max = 0xE66666

73. output\_min = 0x19999A

74. Pmax = 25.000 # max psi

75. Pmin = 0.000 # min psi

76.

77. self.Pressure = (outputs - output\_min) \* (Pmax - Pmin) # Pressure Calculation

78. self.Pressure = (self.Pressure / (output\_max - output\_min)) + Pmin # Pressure Calculation

## Appendix 11: CCS811 Sensor Code

1. # CCS811 Environmental Sensor

2. # Sparkfun Breakout Board

3. # Datasheet: https://cdn-learn.adafruit.com/assets/assets/000/044/636/original/CCS811\_DS000459\_2-00-1098798.pdf?1501602769

4. # Github: https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Sensors/CCS811.py

5. from smbus2 import SMBus # I2C Package

6. from smbus2 import i2c\_msg # I2C Package

7. import time

8.

9. # Constants

10. DEVICE\_ADDRESS = 0x5B

11. STATUS = 0x00

12. MEAS\_MODE = 0x01

13. ALG\_RESULT\_DATA = 0x02

14. RAW\_DATA = 0x03

15. ENV\_DATA = 0x05

16. THRESHOLDS = 0x10

17. BASELINE = 0x11

18. HW\_ID = 0x20

19. HW\_VERSION = 0x21

20. FW\_BOOT\_VERSION = 0x23

21. FW\_APP\_VERSION = 0x24

22. INTERNAL\_STATE = 0xA0

23. ERROR\_ID = 0xE0

24. APP\_ERASE = 0xF1

25. APP\_DATA = 0xF2

26. APP\_VERIFY = 0xF3

27. APP\_START = 0xF4

28. SW\_RESET = 0xFF

29.

30. class CCS811:

31. """

32. CCS811 Environmental Sensor class for Sparkfun Breakout Board.

33.

34. Parameters:

35. - busID (int): The bus ID for communication with the sensor.

36.

37. Attributes:

38. - busID (int): The bus ID for communication.

39. - bus (SMBus): The SMBus instance for communication.

40. - address (int): Device address (default: 0x5B).

41.

42. - eco2 (int): CO2 concentration in parts per million (ppm).

43. - eTVOC (int): Total Volatile Organic Compounds concentration in parts

44. per billion (ppb).

45. - dataReady (bool): Indicates whether data is ready for reading.

46. - \_running (bool): Indicates whether the sensor is successfully

47. initialized.

48.

49. Methods:

50. - \_\_init\_\_(self, busID): Initialize the CCS811 sensor.

51. - checkStatus(self): Check if data is available for reading.

52. - read\_gas(self): Read gas concentration values from the sensor.

53. """

54.

55. def \_\_init\_\_(self, busID=3, address = DEVICE\_ADDRESS):

56. """

57. Initialize the CCS811 sensor.

58.

59. Parameters:

60. - busID (int): The bus ID for communication with the sensor.

61. - address (int): The address for the sensor.

62. """

63. try:

64. self.busID = busID

65. self.bus = SMBus(self.busID)

66. self.address = address

67.

68. self.eco2 = 0

69. self.eTVOC = 0

70. self.dataReady = False

71.

72. self.configure()

73.

74. self.\_running = True

75. print("CCS811 Passed")

76. except:

77. self.\_running = False

78. print("CCS811 Failed")

79.

80. def configure(self, drive\_mode = 0b001):

81. """

82. Start the device, configure the measurement mode, then clear first

83. few readings.

84.

85. Parameters:

86. - drive\_mode (int): Sets the measurement mode and timing.

87. 0b000: Mode 0 – Idle (Measurements are disabled in this mode)

88. 0b001: Mode 1 – Constant power mode, IAQ measurement every second

89. 0b010: Mode 2 – Pulse heating mode IAQ measurement every 10 seconds

90. 0b011: Mode 3 – Low power pulse heating mode IAQ measurement every

91. 60 seconds

92. 0b100: Mode 4 – Constant power mode, sensor measurement every 250ms

93. 0b1xx: Reserved modes (For future use)

94. In mode 4, the ALG\_RESULT\_DATA is not updated, only RAW\_DATA;

95. the processing must be done on the host system.

96. """

97. self.bus.write\_byte(self.address, APP\_START)

98. self.bus.write\_byte(self.address, MEAS\_MODE)

99. self.bus.write\_i2c\_block\_data(self.address, MEAS\_MODE, [drive\_mode<<4])

100.

101. while self.eco2 == 0:

102. self.bus.write\_byte(self.address, ALG\_RESULT\_DATA)

103. val = self.bus.read\_i2c\_block\_data(self.address, ALG\_RESULT\_DATA, 4)

104. self.eco2 = ((val[0] << 8) | (val[1]))

105. time.sleep(.01)

106.

107. def checkStatus(self):

108. """

109. Check if data is available for reading.

110. """

111. if self.\_running:

112. self.bus.write\_byte(self.address, STATUS)

113. val = self.bus.read\_i2c\_block\_data(self.address, STATUS, 1)

114. if val[0] & 0b00001000:

115. self.dataReady = True

116. else:

117. self.dataReady = False

118.

119. def read\_gas(self):

120. """

121. Read gas concentration values from the sensor.

122. """

123. if self.\_running:

124. self.checkStatus()

125. try:

126. if self.dataReady:

127. self.bus.write\_byte(self.address, ALG\_RESULT\_DATA)

128. time.sleep(0.005)

129. val = self.bus.read\_i2c\_block\_data(self.address, ALG\_RESULT\_DATA, 4)

130. self.eco2 = ((val[0] << 8) | (val[1]))

131. self.eTVOC = ((val[2] << 8) | (val[3]))

132. except:

133. print("Error in Gas Read")