# 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.

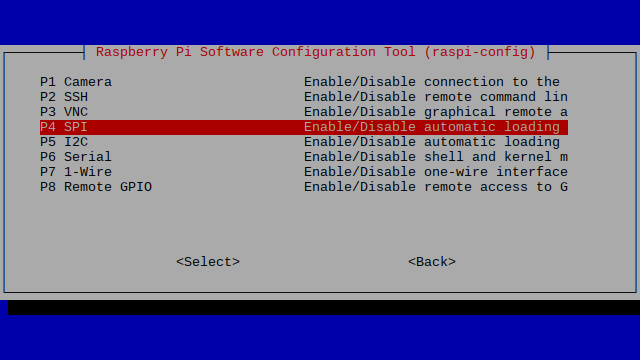


Figure . SPI Configuration (Matt, 2014)

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, 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 two-byte 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.

We do a basic read on the data register to confirm it does not produce an error then move on to fully configuring the sensor.

When we first initialize the sensor, we will configure the data measurement and test read temperature data from the sensor.  We will have a flag dataHere that will be used to read the config register for whether new data is ready to be read. Next we will configure the sensor by setting the 16 bit config register to [0000000010111100] This corresponds in the config datasheet as a Continuous Conversion Mode (constantly reading data) with a conversion cycle of 125ms and 8 averaged conversions per result.  
  
 Once the configuration has been set successfully we can set the sensor’s \_running flag to True and it can now function in the normal sensor code.

With reading the data, we want to read the config register before attempting to read data to make sure there is data available. This is done in the dataReady function which will read the register and check Bit 13 of the register which is set to “1” if there is new data available to read. If there is data available, we will set a dataHere flag to true to indicate to the class that it can read data from the data register.

With the data reading function, tempReading, we will read the 16-bit data register. We then must combine the two bytes together then take the twos complement to get the true value based on whether it is a positive or negative temperature.  
  
 From there we multiply by a provided value to get the temperature in Celsius and store it to a variable available to the sensor class.

Full Code :<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/TMP117.py>

## 

## SHTC3

The SHTC3 is a Temperature and Humidity Sensor with a +-2% releative humidity and +-0.2 C temperature measurements. The key parts of the module we have made for this sensor is the initialization (\_init\_\_) and the measurement function (measure).  
  
 In the initialization section, we attempt to initialize the device by sending a wakeup message, followed by starting a measurement then reading the result. If this passes we will set the device’s “\_running” flag to true.

The measure function is the exact same as the measurement process of the initialization with the Wakeup, Measure, and Read. The results are then multiplied by a given constant to convert from the received binary value to relative humidity percentage and temperature in Celsius.

Full SHTC3 Code :<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/SHTC3.py>

## 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.

The basic initialization of the sensor starts periodic measurements and reads the data from the sensors twice. As long as the sensor is functioning correctly, we will set the running flag to true and allow data measurements to include the sensor.

The measurement function (readSensor) once again reads the data 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 :<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/SCD4X.py>

## BME280

The BME280 is a temperature, pressure, and humidity sensor that can be used to calculate altitude. Due to the complexity of the sensor, we are using an already-made library in integrating our desired I2C bus into it.

The main functions are the initialization and measurement functions. The initialization contains the startup for the pre-done module and measures temperature to test functionality before setting the running flag.

In the measurement function (get\_bme280\_data) 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:<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/self_bme280.py>

## 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.

The initialization portion of the sensor is to do a basic read of the status register of the sensor. If that comes back without issue, the sensor is declared running.

In the measurement portion (read\_pressure) 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:<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/Micropressure.py>

## CCS811

The CCS811 is an environmental sensor that can measure CO2 concentrations in parts per million (ppm) and Total Volatile Organic Compounds (eTVOC) in parts per billion (ppb).

In the initialization portion of this sensor, we must read the data several times to transition from boot mode to operation. At the start, the sensor will read a CO2 value of 0 ppm, while this occurs we will continue to read the sensor constantly to wait for it to go to the sensors actual minimum CO2 reading of 400 ppm. Once it has clear this, we will take an additional measure of CO2 and eTVOC to ensure the sensor performs correctly. Once it has passed this measurement, we set the sensor as running.

Inside the measurement portion of the sensor we have a function to check if there is data available based on a flag in the status register. If it is not showing that data is available, the code will not try to collect data as it would result in an error.

If there is data available, we only need to read the data register and use the first two bytes as the CO2 value and the last two as the eTVOC value. The binary result represents the correct units of ppm and ppb.

Full Code:<https://github.com/Brandonh291/Masters-Project-for-Raspberry-Pi-Based-Companion-Computer/blob/main/CompanionSystem/CCS811.py>

## 

# Main Code Starter

Df

Code

# Communication with Navio

Required items

* USB to Serial TTL Cable
* Navio2 System running

# Appendix

## Appendix 0: MAVLink Task Code

1. # Import Libraries / Modules

2.

3. import threading # Thread-Based Parallelism

4. import time # Time access and conversion module

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

6.

7. class Vehicle(threading.Thread):

8. """

9. Class representing a vehicle with MAVLink communication capabilities.

10.

11. Parameters:

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

13.

14. Attributes:

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

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

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

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

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

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

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

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

23. from the system.

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

25.

26. Methods:

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

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

29. the vehicle.

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

31. message intervals based on the vehicle frame.

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

33. initialized to empty strings.

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

35. interval for a specific MAVLink message type.

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

37. vehicle.

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

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

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

41. relevant attributes based on the message type.

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

43. continuously collects and processes MAVLink messages.

44. """

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

46. try:

47. self.exit = False

48. self.\_armed = False

49. self.header = []

50. self.dataFull = []

51. self.frame = frame

52. self.heart = False

53.

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

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

56. print("Waiting for Heartbeat")

57.

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

59. if self.heartVal == None:

60. print("Failure")

61. self.\_running = False

62.

63. else:

64. self.\_running = True

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

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

67. print(self.heartVal)

68. self.initializeFields()

69.

70. except:

71. self.\_running = False

72.

73. def attemptConnection(self):

74. """

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

76.

77. If successful, initializes necessary attributes and starts MAVLink

78. message streams.

79. """

80. try:

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

82. print("Waiting for Heartbeat")

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

84. if self.heartVal == None:

85. print("Failure to find Heartbeat")

86. self.\_running = False

87.

88. else:

89. self.\_running = True

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

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

92. print(self.heartVal)

93. self.initializeFields()

94. except:

95. self.\_running = False

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

97.

98. def initializeFields(self):

99. """

100. Initializes MAVLink message types and sets message intervals based

101. on the vehicle frame.

102. """

103.

104. print("All Stream Halted")

105.

106. # Do Plane Initialization

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

108.

109. # Set VFR HUD

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

111. print("VFR HUD Stream Started")

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

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

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

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

116.

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

118. #Set POSITION\_TARGET\_GLOBAL\_INT

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

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

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

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

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

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

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

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

127.

128. # Set AHRS

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

130. print("AHRS Stream Started")

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

132. 'omegaIz', 'accel\_weight',

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

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

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

136.

137. # Set GPS\_RAW\_INT

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

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

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

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

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

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

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

145.

146. # Set RC\_CHANNELS

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

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

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

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

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

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

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

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

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

156.

157. # Set MISSION\_CURRENT

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

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

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

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

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

163.

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

165. # Set NAV\_CONTROLLER\_OUTPUT

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

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

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

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

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

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

172.

173. # Set SYS\_STATUS

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

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

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

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

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

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

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

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

182.

183. # Set GLOBAL\_POSITION\_INT

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

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

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

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

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

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

190.

191. # Set RAW\_IMU

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

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

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

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

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

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

198.

199. # Set BATTERY\_STATUS

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

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

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

203. 'battery\_remaining']

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

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

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

207.

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

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

210. print("Heartbeat Stream Started")

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

212.

213. def createEmpty(self, keys):

214. """

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

216.

217. Parameters:

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

219.

220. Returns:

221. - dict: Empty dictionary with keys.

222. """

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

224.

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

226. """

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

228.

229. Parameters:

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

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

232. """

233. if self.\_running:

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

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

236. 0,

237. message\_id,

238. 1e6/frequency\_Hz,

239. 0,0,0,0,

240. 2)

241. time.sleep(0.1)

242.

243. def getMessage(self):

244. """

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

246. """

247. if self.\_running:

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

249. #print(self.msg)

250.

251. def createData(self):

252. """

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

254. """

255. if self.heart:

256. if self.\_armed:

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

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

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

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

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

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

263. 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())

264. #print(self.dataFull)

265.

266. def parseMessage(self):

267. """

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

269. """

270. if self.\_running:

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

272. #print(self.msg)

273. self.VFR\_HUD=self.msg

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

275. #print(self.msg)

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

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

278. #print(self.msg)

279. self.AHRS=self.msg

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

281. #print(self.msg)

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

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

284. #print(self.msg)

285. self.RC\_CHANNELS=self.msg

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

287. #print(self.msg)

288. self.MISSION\_CURRENT=self.msg

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

290. #print(self.msg)

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

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

293. #print(self.msg)

294. self.SYS\_STATUS=self.msg

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

296. #print(self.msg)

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

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

299. #print(self.msg)

300. self.RAW\_IMU=self.msg

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

302. #print(self.msg)

303. self.BATTERY\_STATUS=self.msg

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

305. #print(self.msg)

306. self.HEARTBEAT=self.msg

307. self.heart=True

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

309. self.\_armed=True

310. #print("Armed")

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

312. self.\_armed=False

313. #print("Disarmed")

314. else:

315. pass

316. #print(self.\_armed)

317. else:

318. pass

319.

320. def run(self):

321. """

322. Main execution method for the vehicle thread.

323.

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

325. """

326. while not self.exit:

327. if self.\_running:

328. nextTime=time.time()

329. while self.\_running:

330. curTime=time.time()

331. if curTime-nextTime > 5:

332. if self.\_armed:

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

334. #print("Hellooo")

335. else:

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

337.

338. nextTime=time.time()

339. #time.sleep(.01)

340. #print(navio.nav.messages)

341. try:

342. self.getMessage()

343. except:

344. pass

345. self.parseMessage()

346. self.createData()

347. if not self.\_running:

348. time.sleep(60)

349. self.attemptConnection()

## Appendix 1: I2C Bus Creation

* + 1. We will open the configuration file using the command  
       sudo nano /boot/config.txt
    2. This will open up 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   
       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 can 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 buses 1 or 2.  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

1. I2C Bus 3: Sensors

## Appendix 2: Install Libraries

* 1. First, 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
  2. 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 3: 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 4: 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

df

d

f

df