ECE Capstone Team Project User Manual

Designed for ECE 4905 @ The Ohio State University

Team Members: Tyler Crawford, Zachary Boyd, Nicholas Boenau, and Clarke Clemmons

Purpose Statement:

This document is intended for the members of the FABE department responsible for working with the greenhouse gas sensor device that was done in partnership with Makel Engineering and RedNOx including, but not limited to, Chris Tkach and Dr. Sami Khanal.

Contained within this brief document are instructions and helpful pointers to how to best make use of the ECE Undergraduate Capstone team's project that had been developed throughout the 2023-2024 academic year at The Ohio State University. Due to various conflicts, delays, and project shifts throughout the school year, the finished product does not necessarily represent the full extent of what was originally conceived during talks with FABE; however, the team hopes that it will at least help provide the FABE team with a general framework to develop a wireless communication platform for their project.

Product Usage

Requirements

To ensure the steady operation of the product developed by the team, the following is needed:

Hardware:

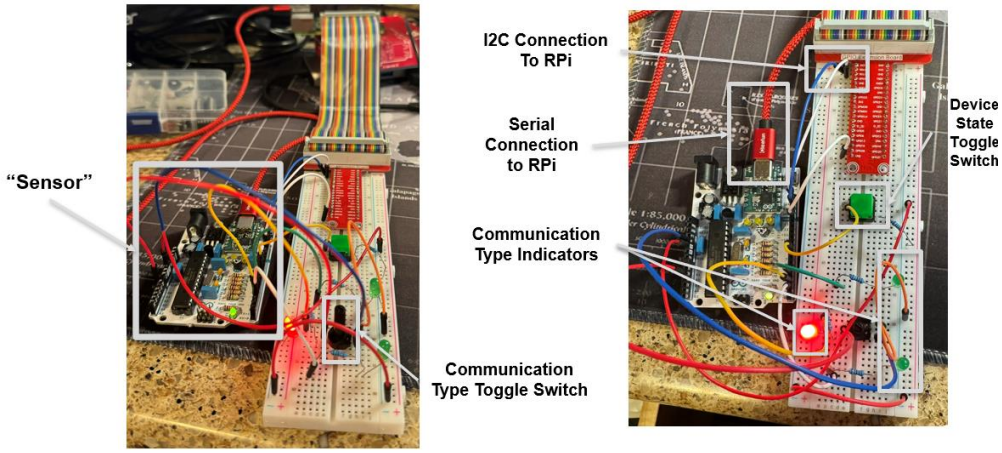
- **A Raspberry Pi 4B** (The model can be adjusted as seen fit; however, there may be small software issues if deviating to previous Raspberry Pi generations. One has been provided to FABE which can be used.)
    - This device serves as the main processing unit for the wireless communications system. It contains all relevant software that communicates with I/O devices such as the Arduino microcontroller, Solid-State relay board, and OLED screen.
- **An Arduino Uno R3** (or any equivalent board with an Atmega328p microcontroller)
    - Due to limitations in the team's access to equipment related to the original project, they decided to develop a mock sensor on the Arduino platform. This system operates in its own state machine and responds to commands from the Raspberry Pi. It generates pseudo sensor data that is meant to be collected by the Raspberry Pi.

- **A Solid-State relay Pi Hat** that is controlled via the I2C protocol
    - The individual relays on this device are used to turn the Arduino and OLED components on/off by using the Raspberry Pi's onboard 3.3- and 5-Volt rails. This device is attached as an extension to the Raspberry Pi's GPIO pins.
- (Optional) **OLED Screen (SSD1306)**
    - This device serves as the visual debugging tool for the overall system; however, if the system is brought out into the field, then this will not be very useful unless someone is looking at it.
    - It is listed as optional because the system can also be debugged over a serial terminal connection when running the team's main software program. Implementation of the OLED screen proved to be too time consuming for the team; therefore, it has not been properly integrated into the software.

Required Hardware Circuit Connections:

To properly operate the team's project in its current state, three major components must be created.

Component #1: Mock Sensor

- The Mock Sensor consists of the Arduino and a communication selection switch (as pictured in the Figure below).
    - 

- **For I2C:** The Arduino is connected to the Raspberry Pi's SDA1, SCL1, and ground pins to allow for proper I2C communication.
- **For Serial (UART):** The Arduino should be connected to the Raspberry Pi via USB (provides power) or to its Tx and Rx pins (does not provide power). This component is not completely configured, however, so stick with the I2C communication.
- The communication selection switch consists of the following components:
    - A potentiometer
    - 3 220 Ω resistors (1 for each indicator LED)
    - 2 Green LEDs (one to indicate I2C communication and another to indicate UART communication)
    - 1 Red LED (used to indicate a communication type has not been selected yet
- The pin connections between the communication selection switch and Arduino are as follows:

- Potentiometer:
  - Wiper (variable end) pin is connected to pin A1 on the Arduino
  - Fixed End pin 1 (forward most pin on the above diagram) is connected to the 5 Volt line of the Arduino
  - Fixed End pin 2 (backward most pin on the above diagram) is connected to a ground line of the Arduino
- Green LEDs:
  - The LED indicating a UART communication type is connected over a 220 Ω resistor to digital pin 4 of the Arduino.
  - The LED indicating an I2C communication type is connected over a 220 Ω resistor to digital pin 2 of the Arduino.
- Red LED:
  - The LED is connected over a 220 Ω resistor to digital pin 3 of the Arduino.

Component #2: Raspberry Pi 'Sleep' State Wake-Up Button

- The Wireless Network Application program uses a state machine that defaults to leaving the Raspberry Pi in 'Sleep' mode. The team opted to implementing a hardware solution that relies on the user to press down and release a toggle button (the green button shown in the figure above).
- GPIO pin 24 is attached to one end of the button and is configured to read a falling edge and set the Raspberry Pi to State 1 within the Wireless Network Application. This works properly because an internal pull up resistor is set on the GPIO pin which results in a HIGH signal on the pin that is brought to ground when the button is pressed.

Component #3: Solid-State Relay (SSR) Pi Hat for the Raspberry Pi

- The Raspberry Pi system requires a SSR Pi Hat to channel its 5- and 3-Volt rails through to power both the mock sensor. Each of the 4 relays on the SSR must be turned on and off by using the I2C protocol which allows for programmatic control of each connected device. A brief description of which relays are used is provided below:
  - **Relay #1**: This **must be** connected to the 5-Volt rail of the Raspberry Pi to supply power to the mock sensor device.
    - One lead is placed in the *Normally Closed (NC)* port of the relay and fed into the 5-Volt rail of the Raspberry Pi.
    - Another lead is placed in the *Common (Com)* port of the relay and fed into the *Vin* input pin on the Arduino.
  - **Relay #2:** This **must be** connected to either the 3.3 or 5-Volt rails of the Raspberry Pi to supply power to the SSD1306 OLED screen (either voltage appears to work find through testing).
    - One lead is placed in the *Normally Closed (NC)* port of the relay and fed into either the 3.3 or 5-Volt rail of the Raspberry Pi.
    - Another lead is placed in the *Common (Com)* port of the relay and fed into the *VCC* input pin on the OLED screen.
- While the Raspberry Pi is in State 0 (Sleep mode), the relays remain off. When it leaves State 0 and enters State 1, both relays 1 and 2 are switched on. This was implemented for the ease of operating the system; however, further experimentation may reveal that the components can be turned on at different times. At a bare minimum, the mock sensor must be turned on for State 1 to be exited as soon as possible.

Software:

- SSH is required to be enabled on the Raspberry Pi device being used for actual use. This is used as the protocol to handle the transmission of the collected sensor data files to the remote research data collection point. OpenSSH should be enabled on this receiving machine.
- Necessary software libraries:
    - BCM2835-1.75[1] (This should already be installed on the Raspberry Pi system provided to FABE)
        - This C/C++ library was used to program the necessary I2C communication and GPIO access on the Raspberry Pi 4B. It is required to use most of the key features of the team's software program.
        - To install the most recent version of this library, the following commands must be issued in the Linux command terminal on the Raspberry Pi[2]:
            - "wget http://www.airspayce.com/mikem/bcm2835/bcm2835-1.75.tar.gz"
            - "tar zxvf bcm2835-1.75.tar.gz"
            - "cd bcm2835-1.75/"
            - "sudo ./configure && sudo make && sudo make check && sudo make install"
    - OLED Libraries (Choose one based on the desired means of integrating it with the team's code)
        - SSD1306_OLED_RPI[3] (This should already be installed on the Raspberry Pi system provided to FABE)
            - This C++ library was chosen due to its tight integration with the BCM2835 library discussed above. Due to the team's timeline, development for the OLED component using this library did not end up resulting in a complete product; however, it is strongly recommended to use this library for C++ development of this device (at least as a starting point). As will be described later, a C++ interface was made to interact with the OLED for the team's project.
        - Adafruit's SSD1306[4] (This should be installed on the Raspberry Pi system provided to FABE, but so will CircuitPython which is

[1] The official website hosting the library and related documentation can be found here: https://www.airspayce.com/mikem/bcm2835/

[2] The website where these instructions are referenced can be found here: https://www.waveshare.com/wiki/Libraries_Installation_for_RPi

[3] The GitHub hosting the library along with install instructions can be found here: https://github.com/gavinlyonsrepo/SSD1306_OLED_RPI/tree/main

[4] The GitHub hosting the library along with install instructions can be found here: https://github.com/adafruit/Adafruit_CircuitPython_SSD1306/tree/main

the new overarching framework for embedded Python programming)

- This is a Python-based library that the team briefly developed for; however, they could not get it to properly integrate with the mock sensor device over I2C. This prompted a switch over to the C++ library mentioned above.

A Brief Overview of the Software Design:

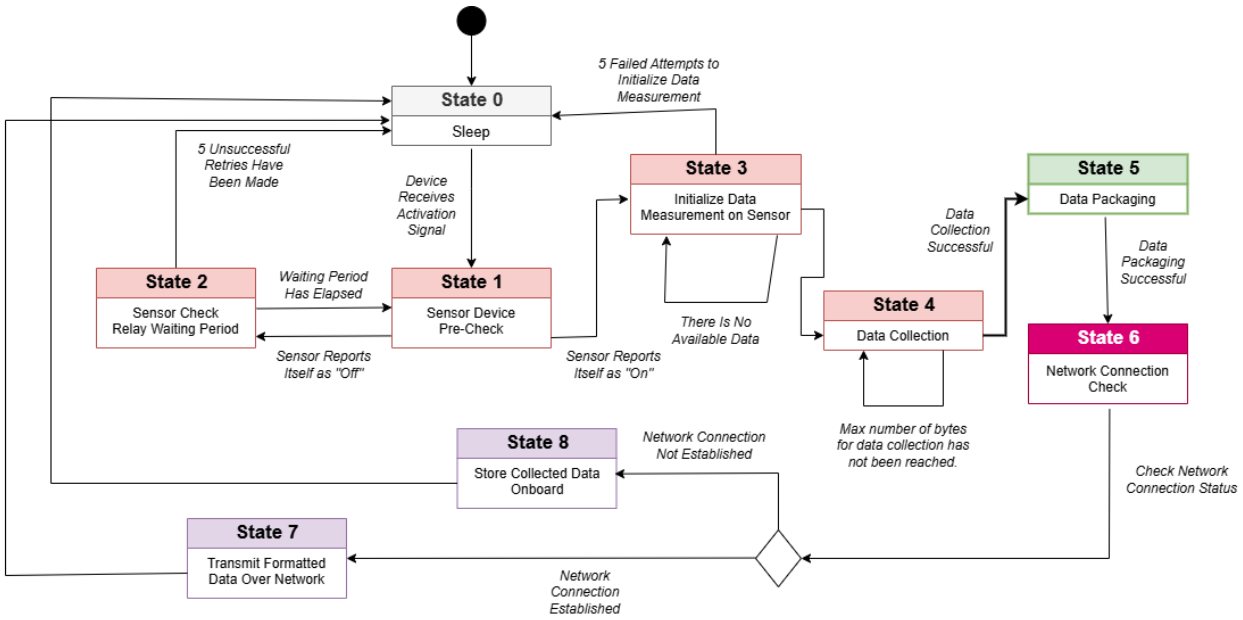


Figure 1 – The Wireless Communication Application State Diagram

Since the proposed Wireless Communication system was meant to run continuously when activated by the FABE researchers to collect their data, the team decided that a state machine, shown in Figure 1, was required to properly capture/implement this idea in software. The state machine consists of 9 total states, each reflecting a specific action that the Raspberry Pi running the application performs at any given point in the program's operation. As it currently stands, the implementation runs its main operations on a single thread; however, the team believes that a multi-threaded program would enhance the effectiveness of the network connection check. A network connection check thread could inform the main application thread could promptly inform the main application thread about a lack of a network connection which could help increase the reliability of the overall program's operation. Explanations of each state's operation have been provided in Figure A.1. **The intention of providing this framework is to provide a framework for further development of a wireless communication/data collection system.**

The OLED Screen

The OLED screen component of the project is intended to provide someone who is monitoring the system up close a view at specific system parameters of interest. These parameters include the following:

- The IPv4 address of the Raspberry Pi

- A general description of the current network connection to the server, which can be of the following 4 possibilities:
    - "Net: Disconnected"
    - "Net: Weak"
    - "Net: Average"
    - "Net: Strong"
- The current temperature reported by the onboard microprocessor temperature sensor (in degrees Celsius)
- The current state the Wireless Network Application is in which has also been simplified into one of the 6 following possibilities:
    - "State: Sleep"
        - When the application is in State 0.
    - "State: On"
        - When the application is in States 1-3.
    - "State: Data Collect"
        - When the application is in States 4-5.
    - "State: Data Transmit"
        - When the application is in State 7.
    - "State: Data Store"
        - When the application is in State 8.
    - "State: Unknown"
        - When the state is not represented by the current state machine. This was only added for future compatibility.

The team developed a small interface to manipulate the selected SSD1306 OLED component and called it *'oledInterface.cpp'*. This was effectively a wrapper API for an existing C++ library developed by Gavin Lyons (this library was previously mentioned in footnote 3 in the 'Software' section of this manual). Further development past the existing library was required due to how the library made use of the bcm2835 I2C module (from the library mentioned in footnote 1 in the 'Software' section). The team wanted to provide specific functionality for displaying the four main system characteristics which was convenient to put inside of their own functions.

After much trail and error, the team did not get the OLED screen properly integrated with the Wireless Network Application program; however, they have still provided the code developed to support it.

The Mock Sensor

The mock sensor was developed on the Arduino platform and configured with the necessary I2C receive and request handlers which have been provided in Figure B.1 (called *handleReceiveMessage* and *handleRequestMessage*, respectively). To provide effective communication between the Raspberry Pi and mock sensor, a set of 4 commands were created: *Read Device Status, Begin Data Collection, Collect Data,* and *Terminate Data Collection*. A table containing a brief description of these commands have been provided in Table B.1. Please refer to the comments in the *MockSensor/mockSensor.ino* code for further information about how it functions.

Final System Overview:

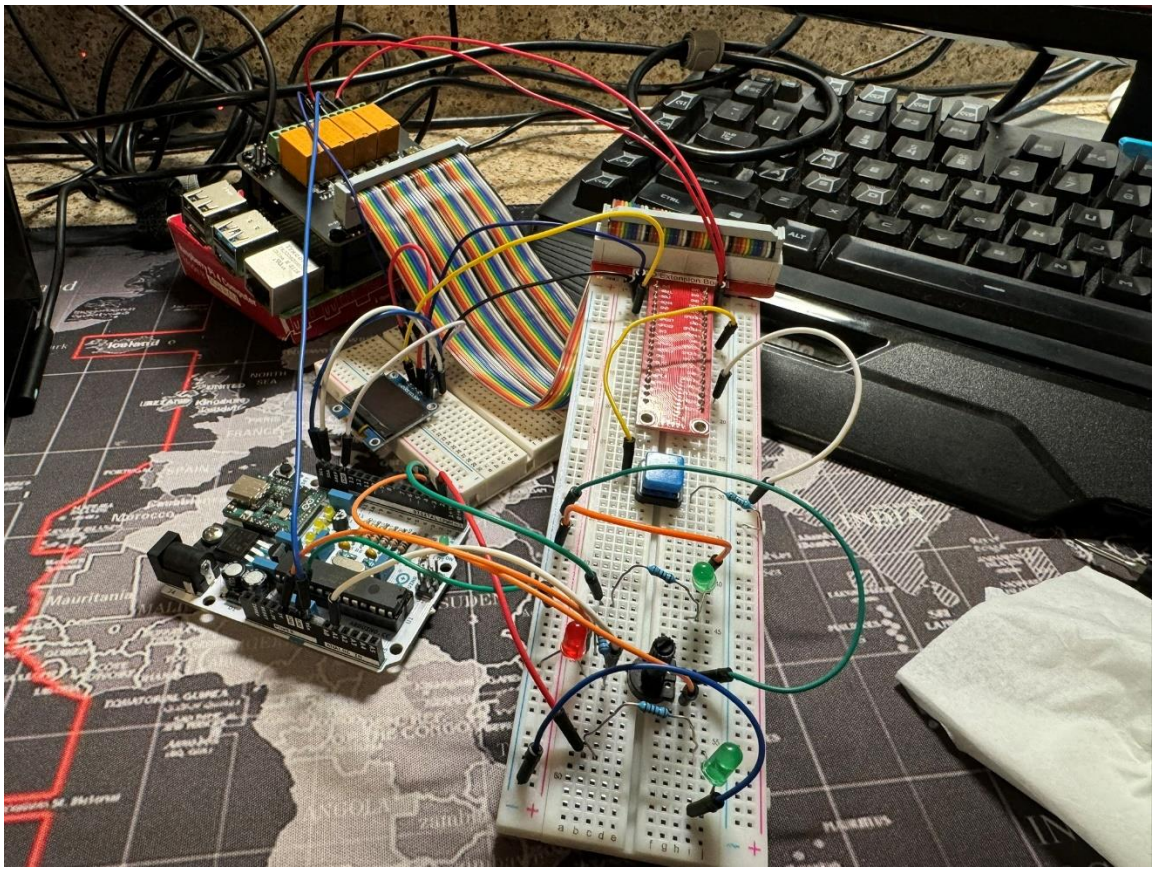


Figure 2 – The final system with all components connected to the Raspberry Pi via a GPIO breakout board

A final view of the wireless communication system is provided in Figure 2 with all components connected. The only components being provided to FABE are as follows:

- The Raspberry Pi
- The SSR Pi Hat
- The OLED screen

The remainder of the components needed to operate the system should be purchased and have been previously described above.

Product Usage + Troubleshooting Guide:

All instructions for how to properly install, build, and execute the Wireless Network Application program have been provided at the GitHub repository here: https://github.com/Triggs02/FABE_WirelessNetworkCommApp.

Problem 1

P: Program executes and initiates data transfer but no files are received.

A: To ensure that data transfer is successful, several actions must occur prior to running.

1. Firstly, ensure that any firewall/antivirus is disabled on the receiving machine. This is due to the nature of the Raspberry Pi and SSH being flagged as an untrusted device by most windows machines.
2. Ensure SSH has been enabled on your receiving device. To do so, see article here
3. Change the respective data in the "networkCredentials.h" file to fit your device including:
    a. Device username
    b. Device password, this will be used later during program execution
    c. Device IP address, to find this, open a terminal and run "ipconfig", then locate the IPv4 address as shown below in Figure 28

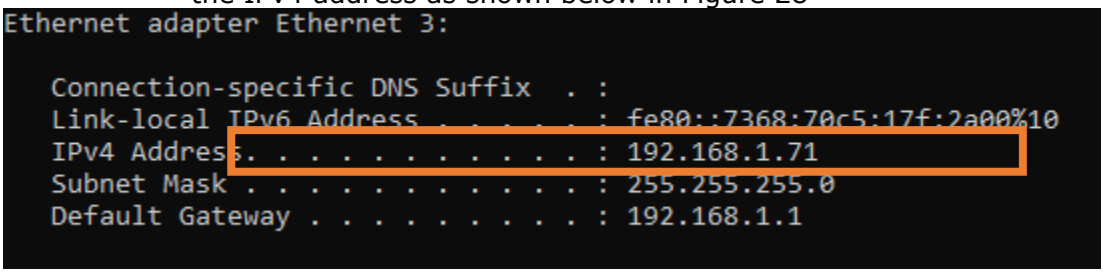


Figure 28 – Device IP Address

    d. Desired file directory destination for the file transfer. (Note: this must use '/' instead of '\' due to the nature of Linux, i.e. '/C:/Users/boena/Desktop/TestData/')
    e. Ensure both the wireless module and the computer are connected to the same network

Problem 2

P: Cannot log in to wireless module

A: Log in information for the Raspberry Pi 4B Module is as follows:

Username: pugAero
Password: pugAero24

Appendix A – Wireless Network Application's Software Program Operating State Descriptions

Table A.1 – The data collection/wireless communication system's states with descriptions

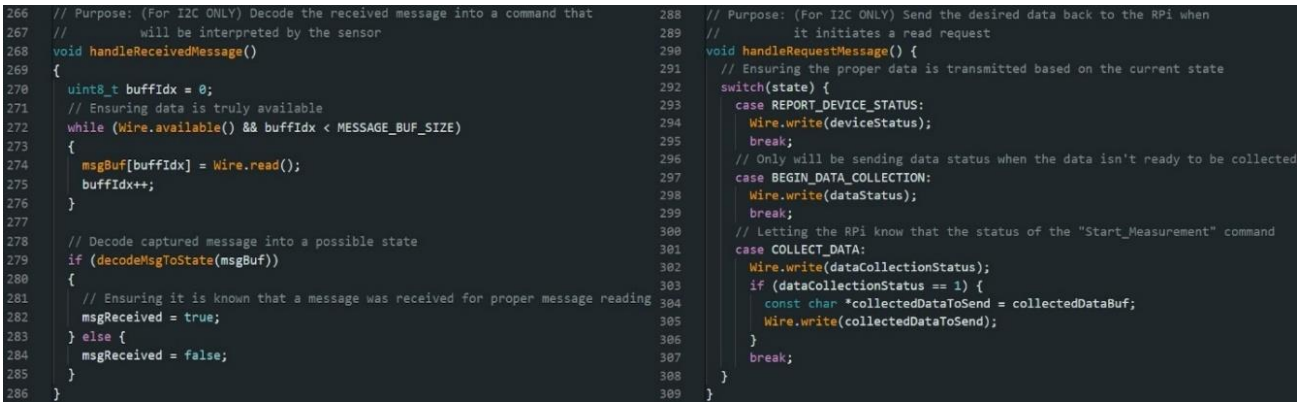
State	Description
S0: *"Sleep"*	The Raspberry Pi is in deep sleep mode until receiving an activation signal from an external hardware button.
S1: *"Sensor Device Pre-Check"*	Before data is collected, the RPi ensures the mock sensor is on.
S2: *"Sensor Check Retry Period"*	If the mock sensor is not on, then the RPi will try checking 5 more times.
S3: *"Initialize Data Measurement on Sensor"*	Once it determines the sensor is on, the RPi will send a command to the mock sensor to indicate that the mock sensor should expect data collection to begin.
S4: *"Data Collection"*	The RPi will continue collecting data in batches until all data has been received.
S5: *"Data Packaging"*[5]	The RPi will make sure the data is formatted and packaged in a file of the expected format on the receiving device.
S6: *"Network Connection Check"*	In a separate thread, the RPi will establish a connection to the FABE network and continuously poll if it is still available.
S7: *"Transmit Formatted Data Over Network"*	The RPi will transmit the collected data to the pre-determined collection server.
S8: *"Store Collected Data Onboard"*	If the RPi cannot connect to the network, then it will store the data locally on its SD card in a pre-defined directory.

[5] In its current state, the Wireless Network Application checks if the Raspberry Pi is connected to the Internet by pinging Google's infamous DNS server at 8.8.8.8. Only if this check fails will the Raspberry Pi decide to store the data locally. Since the team had little information about how the FABE research team wanted the collected sensor data packaged, they opted to configure the state in this way. Further modification to the state is required to add the intended functionality.

Appendix B – The Mock Sensor Software Program Specifics

Table B.1 – The commands used to communicate between the mock sensor and Raspberry Pi over I2C

Command	Description
Read Device Status	Checks whether the sensor is “on.”
Begin Data Collection	Initiates data collection on the sensor.
Collect Data	Collects the next available batch of data from the sensor.
Terminate Data Collection	Sets the sensor back into an “idle” state.



```
// Purpose: (For I2C ONLY) Decode the received message into a command that
//          will be interpreted by the sensor
void handleReceivedMessage()
{
  uint8_t buffIdx = 0;
  // Ensuring data is truly available
  while (Wire.available() && buffIdx < MESSAGE_BUF_SIZE)
  {
    msgBuf[buffIdx] = Wire.read();
    buffIdx++;
  }

  // Decode captured message into a possible state
  if (decodeMsgToState(msgBuf))
  {
    // Ensuring it is known that a message was received for proper message reading
    msgReceived = true;
  } else {
    msgReceived = false;
  }
}
```

```
// Purpose: (For I2C ONLY) Send the desired data back to the RPi when
//          it initiates a read request
void handleRequestMessage() {
  // Ensuring the proper data is transmitted based on the current state
  switch(state) {
    case REPORT_DEVICE_STATUS:
      Wire.write(deviceStatus);
      break;
    // Only will be sending data status when the data isn't ready to be collected
    case BEGIN_DATA_COLLECTION:
      Wire.write(dataStatus);
      break;
    // Letting the RPi know that the status of the "Start_Measurement" command
    case COLLECT_DATA:
      Wire.write(dataCollectionStatus);
      if (dataCollectionStatus == 1) {
        const char *collectedDataToSend = collectedDataBuf;
        Wire.write(collectedDataToSend);
      }
      break;
  }
}
```

Figure B.1 – The I2C communication handlers programmed for the mock sensor