Real-Time Display of Carbon Dioxide Contents at Local, National, and Global Levels

Created By:
Ricardo Calderon,
Ellis Chen,
Cordero Forgach,
Luis J. Moreno,
Tejpal S Tut

Faculty Sponsor:
Dr. Rahim Khoie

Abstract

This report discusses the motivations behind the design of a real-time display of carbon dioxide contents at local, national, and global levels. Project implementation, component selection, component testing, overall system integration, and analysis of system evaluation are discussed in detail. The objective of this project is to create an autonomous system that collects CO2 and temperature readings wirelessly through distributed sensor nodes, and collects data at national and global levels from reputable online sources; such contents are all displayed to a general audience, all while the system runs entirely on renewable energy. A discussion of viable hardware selection was motivated by the prospect of having a sustainable system that is long-lasting, low-cost, and power-efficient. A detailed evaluation of power testing and hardware testing showed that the system under test produced local CO2 data and temperature data. Software components such as the wireless transfer of sensor readings and processing of multiple samples of data to a display via bash scripting and Python automation are also discussed in great detail. The system as a whole conforms to project objectives, and test results in this report prove that the system is sufficiently powered and operable.

Project Overview

What we fundamentally lack as human beings, are a sense of scale and a clear resolution to see the fine details of a vastly bigger picture. We are, in this instance, focusing our attention upon the growing CO2 levels and the environmental impact this has on the ecosystem. Even if we arm our citizenry with knowledge and skip the middlemen of the media by allowing them to see the raw data themselves, the scale is meaningless. A graph from a satellite cannot, effectively, monitor local scale for every township, much less every city block. The costs of attempting to do so in every township, for every hour and for every day would simply be too great. A distributed model, however, allows for a much finer resolution. It creates engagement for locals to witness the output from a sea of sensors for the sea of CO2 that floats and drifts on currents—and to see their own contributions to it. As our population grows and as we replace the existing biomes with those of our own manufacture, we must, in turn, assume the need to monitor, replace, or foster the existing ecosystems we might otherwise callously pave over. As stewards of civilization and our beloved Terra, it is the fundamental responsibility of humans to assume ownership of the environment we are creating together. Therefore we intend to inform the public by designing an automated hybrid system consisting of a kiosk, website, and sensor networks to record CO2 levels. We look to further bring awareness to the average person with such valuable information and to better inform them about how trends in carbon dioxide levels affect the environment.

Project Objectives

This system will support a series of autonomous sensor nodes that will each collect CO2 and temperature readings three times each day. All of these sensor nodes will be solar powered, and will be able to hold enough power in a battery to last throughout the night. This node network will span roughly the area of the University of the Pacific campus. Collected data will be passed through the nodes using a wireless communication protocol, and will also be stored locally in a main hub. The main hub will house a display that shows the real-time CO2 contents on a local, national, and global level. The data collected from the sensor nodes will be uploaded to a web server to be viewable worldwide. The web server will also host a repository containing national and global CO2 contents from trusted sources like NOAA, EIA, NASA, etc.. All the information stored in the web server should be compiled into a presentable format that is accessible to the average person.

Background

"Human activities have increased the concentration of carbon dioxide in our atmosphere, amplifying Earth's natural greenhouse effect. The ocean has absorbed enough carbon dioxide to lower its pH by 0.1 units, a 30% increase in acidity" [1]. It is important to consider that most of our society is unaware of the role carbon dioxide gasses play in the planet's ecosystems, and as such this system shall inform the public about the severity of this greenhouse effect with real-time data. The unit of measurement used to quantify the concentration of carbon dioxide in this system is in parts per million (ppm), which is how many parts a certain molecule or compound makes up within the one million parts of the whole solution [2]. For example, 400 ppm of carbon dioxide means there are 400 carbon dioxide molecules out of 1 million molecules within a certain volume of space. Higher concentrations of CO2 are attributed to a greater measurement in ppm.

Renewable energy is an important aspect of this system. Using solar as the energy source to power our system promotes the welfare of the environment because these sources do not release harmful byproducts into the atmosphere. "Solar energy is clean, efficient, and sustainable for your household or workplace. Carbon dioxide is produced mostly during the generation of electricity and also during consumption. Solar panels have no emissions whatsoever hence a guarantee of no carbon footprint if you depend on natural energy" [3].

Global, Social, and Environmental Impact

Applications in Real World and Global Impact

This project provides several real-world applications that give credibility to its existence. The first and most important application is the ability of the project to provide environmental information to the public that may normally be difficult or confusing to find. The general audience may not have a complete understanding of the scale of CO2 contents prevalent in the world. This project is designed to educate viewers through showing them how CO2 and temperature levels are changing both within their immediate proximity as well as throughout the nation and globe. The second real-world application of this project is the scientific documentation of environmental data. Because the nodes are planned to be collecting data points for a long period of time, eventually a sizable record of CO2 and temperature data will be recorded and stored in our system storage for the benefit of future scientific endeavors.

Societal Impact

This project aims to positively impact society by raising awareness of CO2 levels throughout the world, and to bring light to the related environmental issues. Possible negative impacts on society include the necessary space required to establish the system in order to allow the nodes to collect their data, as well as the space required for the hub to be viewable in a public area. There may be some people who believe that the public space could be better used for another purpose. However, awareness is a critical step to a better, more eco-friendly society, which is why the implementation of this real-time display of CO2 contents is important.

Environmental Impact

This project will likely not have any direct impacts on the environment, whether they be positive or negative. The main hub and sensor nodes will all draw power from renewable sources. Additionally, any construction materials used will not be harmful to the environment. The system is not actively aiding the environment either, as it is simply measuring and recording local data. However, as mentioned previously, the project will aim to indirectly aid the environment in the long term by way of increasing social awareness of environmental issues, and thus prompting an increasingly healthier attitude towards the environment.

Design Implementation

System Structure

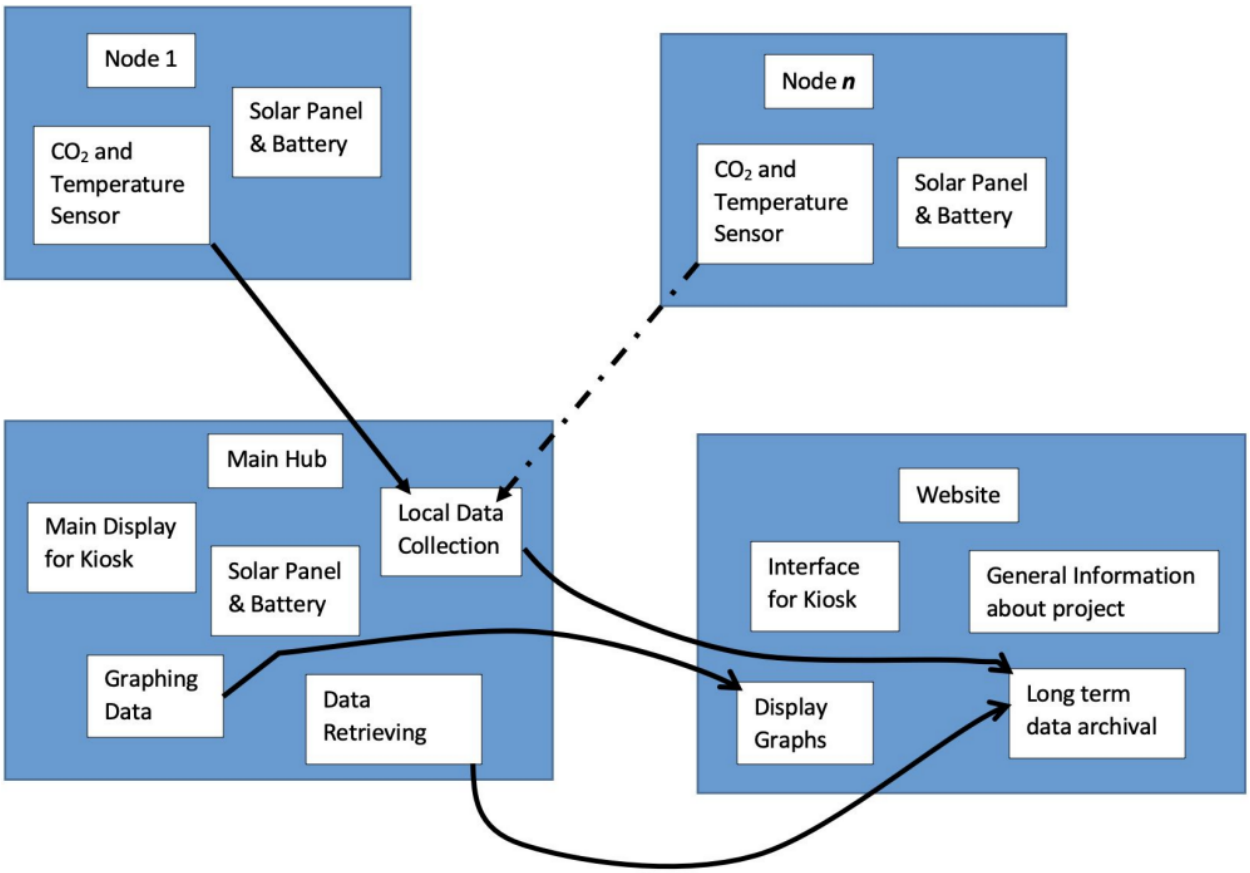


Figure 1. System Block Diagram showing the abstract connections between various modules of the CO2 monitoring system.

The overall system is broken down into 3 main subsystems that each serve different purposes. Figure 1 shows the block diagram of the overall system. The main hub (bottom left) is the subsystem responsible for the display of the CO2 contents, communication with a website that houses national and global databases, and collecting CO2 and temperature data from local sensor nodes. The sensor nodes (top left and top right) are subsystems that contain sensors that read local CO2 and temperature data in real-time. The system utilizes a network of these nodes to collect data from various geographical locations. The overall system is designed to be easily scalable in terms of the quantity of sensor nodes established in the network. The website (bottom right) is the subsystem responsible for extracting the global and national CO2 measurements from sources such as NOAA, EIA, and NASA. The contents of the website are compiled into a graph that is displayed on the main hub. The local CO2 data from the sensors are also uploaded to the website database and displayed on the main hub.

Hardware Description

The CO2 monitoring system requires two main physical subsystems that run independently of each other. The first subsystem is the main hub, which houses a microcontroller that drives the collection of CO2 and temperature data readings on a local, national, and global level. As shown in figure 2, the raspberry pi serves as the microcontroller for the main hub and it derives its power from solar and battery sources. The main node receives the local CO2 and temperature data via an XBee wireless communication module; the data is post-processed and stored locally, where it will be uploaded to the website database.

The second subsystems are the sensor nodes, which houses a microcontroller that reads data from sensors and sends them wirelessly to the main node via the same XBee wireless communication module. As shown in figure 2, the microcontroller in the sensor nodes are also powered by solar and battery. The microcontroller used for the design of this subsystem is the Seeeduino Stalker, which proves to be power-efficient and reliable for usage in a remote sensor network. The sensors and wireless communication modules derive power from the 3.3V pin on the microcontroller, but the communication protocols are all different for each component.

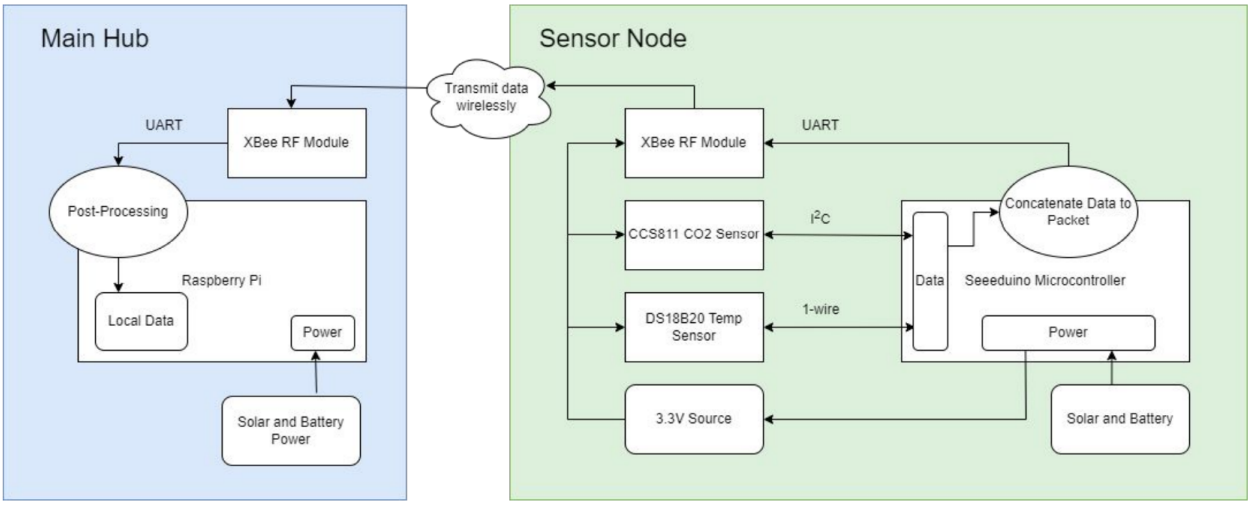


Figure 2. Flow diagram showing the flow of local CO2 and temperature sensor data across various hardware components of a sensor node and the main hub.

Photovoltaic Panels

The Node and Main Hub will run independently of any power outlet and be placed in remote, easily accessible spots on campus. To achieve this, photovoltaics are used to charge and power both components of the system.

The Main Hub uses more power than the node, therefore a larger photovoltaic panel is used to power the hub and charge the hub's battery. While booting, the hub draws roughly 1.3A and levels off at about 750-800mA of current with 5V. To keep things powered with the solar panel and/or charge the battery, A 9W, 6V photovoltaic panel is used for the main hub.

The Node uses less power than the Main Hub, so it uses a smaller, slightly less power driven photovoltaic than the Main Hub. The Seeeduino, along with the sensors and Xbee antenna, operates on 5V and takes in about 350mA of current. For this case, a 5V, 500mA solar panel is used to power the node and charge the node's battery. This photovoltaic panel comes with a duplicate panel that can be wired in parallel to increase the current or in series to increase the voltage output of the solar energy components.

Power Storage

By using photovoltaics, there are some drawbacks that we have to consider. The main disadvantage being the limited time in which the sun is in sight of the photovoltaics. Due to the sun's path across the sky, the time where the photovoltaics are hit with perpendicular sunlight is limited to a short period during the day. A lithium polymer battery is used to compensate for the times where the sunlight is not sufficient to power the node or the main hub.

The main hub is the central point of the whole system, so it is expected that it will be in operation the longest throughout the day. It was projected that the most possible use the hub will have will be during normal campus hours from 9am to 5pm. In addition, the charge controller for the both devices requires external power storage devices be at 3.7V or 4.2 V. Given the Solar Panels can drop to 4.2V or lower, the 3.7V batteries were chosen for both components. The main hub's battery was chosen to be a 3.7V 10Ah battery that will allow it to work all day with two hours of spare time. The node battery uses a 3.7V 2.2Ah battery.

Charge Control/Power Conversion

Due to the variable output of the photovoltaics, a DC to DC step up/step down solid state power converter was used in conjunction with the Main Hub's power components. The purpose for using this power converter is to maintain as much of a consistent output as possible from the photovoltaics at any point of the day. The power converter is then attached to a MakerFocus H.A.T. (hardware attached on top) charge controller that is then used to power the Raspberry Pi and the display while also charging the battery.

The Seeeduino board is designed with the intention of being a remote sensor node. Due to this design specification the PCB board for this microcontroller has a corner dedicated to taking in solar energy and charging a lithium polymer battery. The Seeeduino board uses a built-in CN3065 solar charge controller with a voltage rating of 6V .

Microcontroller

The Raspberry Pi 4 Model B with 4GB of RAM is the controller used in the main hub. The Raspberry Pi 4 has the advantage of displaying graphics onto a display and it has a Broadcom BCM2711 quad-core processor. This amount of processing power is required to perform multiple tasks simultaneously, such as maintaining wireless communication with sensor networks and displaying CO2 contents from the website. The raspberry pi may not be the most power efficient option compared to other microcontrollers, but the benefit of having graphics and parallel processing capabilities makes the raspberry pi a valuable choice. The raspberry pi 4 is also one of the only options capable of running a full desktop operating system, which is important for bash script development and execution.

The Seeeduino Stalker v2.x is the microcontroller used in the sensor nodes. The seeeduino is a low-cost, energy efficient arduino-based microcontroller that is designed for network sensor projects. The microcontroller supports numerous communication protocols for attaching sensors, such as I2C, SPI, UART, and serial interfaces. The main advantage of the seeeduino is its built-in solar to battery charge controller, this built-in feature saves time and costs of researching and acquiring a third-party one. Also, the seeeduino has a built-in Bee series socket that accepts any wireless module such as ZigBee, WifiBee, GPSBee, XBee, etc.. The seeeduino makes a convincing statement as an effective wireless sensor system due to many of its built-in features.

Carbon Dioxide Sensor

The CCS811 was the CO2 sensor selected for this project. The CCS811 is a low-power digital gas sensor solution that contains a microcontroller unit and an Analog-to-Digital converter. These components allow the sensor to monitor the local environment and provide an indication of the air quality via an equivalent CO2 output over a standard I2C digital interface. The sensor is also self-calibrating, and the baseline it uses for its measurements evolves over time. The main advantages to using this CO2 sensor is its small footprint and very-low power design. The sensor provides an output range of 400 to 8192 ppm of CO2, which is suitable for the implementation of this project. I2C is a widely-used protocol that is supported by many devices, which makes establishing and debugging communication related issues between the microcontroller (seeeduino) and the sensor more straightforward.

Temperature Sensor

The DS18B20 is a digital thermometer that supports 9-bit to 12-bit Celsius temperature measurements. The temperature sensor communicates over a 1-wire bus, which means the data line of the temperature sensor can be attached directly to the microcontroller. The sensor can derive all its power from the data line, and does not need an external voltage source. The advantage of this sensor is its scalability, in which each DS18B20 sensor has its own unique 64-bit serial code that allows numerous DS18B20 sensors to run on the same 1-wire bus. The sensors have an accuracy of 0.5℃ with a range of -10℃ to 85℃, which makes this sensor suitable for the implementation of this project.

Wireless Communication

The XBee Pro S1 RF modules were implemented in the sensor node and the main hub to establish a wireless network between the two. The XBee modules were chosen because the seeeduino stalker board had a Bee series socket that allowed any wireless module, such as the XBee, to mount onto for convenient integration with the microcontroller. The XBee Pro S1 also provides long range signal transmission and reception, up to 90 m in urban areas and a theoretical 1600 m for outdoor RF line-of-sight. XBee modules are also easily configurable for multiple modules running on the same network. These configurations are facilitated by a software called XCTU, which is not only a software that configures the XBees, but the software also contains numerous tools for testing wireless signal strength or tools for debugging network data packets.

Display

A 7-inch capacitive touch screen LCD was chosen as the primary display for the system. The display draws near 500mA @ 5V with the backlight on. The 2.5W of power draw is justified because the display runs on a bright LCD panel that can be clearly seen from at least 10 feet away with 170° viewing angles. The display is also capable of capacitive touch, which makes implementing a kiosk-like display with navigation by touch possible. The display works with the HDMI interface, which makes it compatible with the graphics coming from the raspberry pi in the main hub.

Software Description

Wireless communication is facilitated by using the XBee Pro S1 RF modules. Each sensor node will have an XBee module attached to form a wireless network. The Raspberry Pi in the main hub will also have an XBee module attached, and that is treated as the central device of the network. These modules require a software called XCTU by Digi to configure their radio frequency settings. As shown in figure 3, multiple parameters must be set within the XCTU software for the XBee to work properly with other XBees. To create a network for multiple XBees to talk to one another, each XBee must have matching settings for the Channel, PAN ID, and Destination Address High. Only two parameters need to be set differently; the Destination Address Low for one XBee must be the 16-bit Source Address of the other, and vice versa. Figure 3 shows example configuration settings for one XBee module.

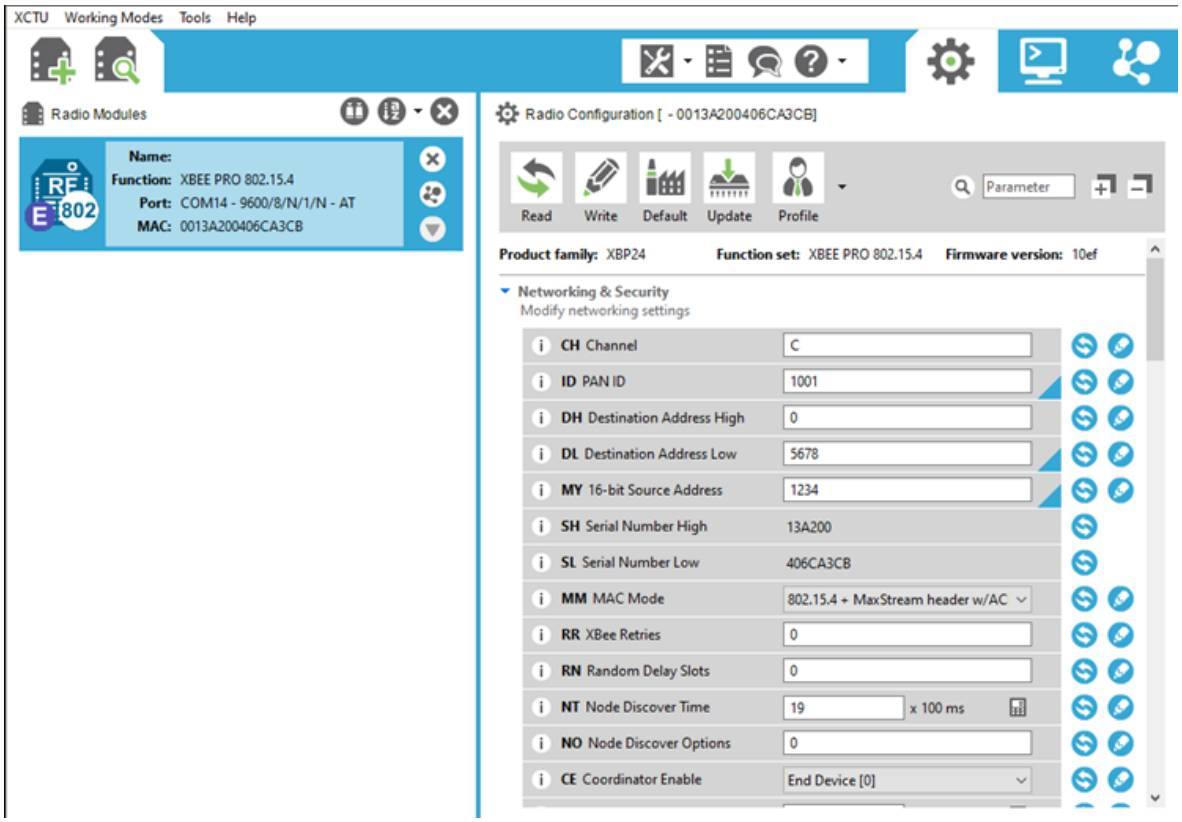
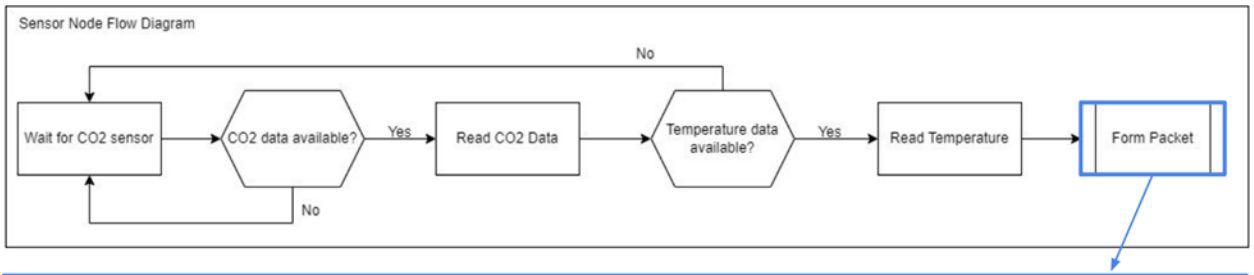


Figure 3: XBee radio configuration page within the XCTU software

Transmitting any type of data, let alone wirelessly, requires a structure that can be read properly by other machines on the receiving end. Figure 4 shows the flow diagram that the seeeduino will be programmed to follow. The seeeduino checks if the CO2 sensor is ready to read data after the sensor finishes warming up. The seeeduino reads the CO2 data when it is available and stores it locally in memory. Then the seeeduino proceeds to read from the temperature sensor. In the event the temperature sensor is unable to provide valid measurements, then all measurements including the CO2 readings are discarded, and the process starts over. In the event both the CO2 and temperature readings are valid, then the floating point values will be stored and post-processed into a packet-like structure.



Packet Structure

	NODE ID	\t	TEMP (°C)	\t	CO2 (PPM)	\t	CHECKSUM
Example 1	1		22.55		420		14
Example 2	33		31.87		1132		16

Figure 4. The script follows the above flow diagrams to make sure sensors provide CO2 and temp data upon each request from the main hub. When the node microcontroller reads CO2 and temp data from its serial buffer, then the data is parsed into a packet form.

All the data at this point was recorded as floating-point precision. Floating point precision numbers take up 4 bytes. This may be ideal for easy byte transmission over the serial port of the microcontrollers; however, floating point precision numbers can start to become more memory intensive as the number of data requests and the addition of more sensor nodes increase. Converting the sensor data from floating point to string will be more cost-effective in terms of memory. A string is a concatenation of characters, and a char data type only requires 1 byte each. A list of data types and their sizes in bytes are tabulated in table 1. A string is also more useful when concatenating data into a packet-like structure, which is the motive for data transmission for this project.

Type	Size (bits)	Size (bytes)	Range
char	8	1	-128 to 127
unsigned char	8	1	0 to 255
int	16	2	-2^{15} to $2^{15}-1$
unsigned int	16	2	0 to $2^{16}-1$
short int	8	1	-128 to 127
unsigned short int	8	1	0 to 255
long int	32	4	-2^{31} to $2^{31}-1$
unsigned long int	32	4	0 to $2^{32}-1$
float	32	4	3.4E-38 to 3.4E+38
double	64	8	1.7E-308 to 1.7E+308
long double	80	10	3.4E-4932 to 1.1E+4932

Table 1. The sizes in bytes are shown for each data type as well as the range of values they may occupy. The main data types of interest are highlighted in yellow [4]

The packet structure from figure 4 is stored as a string data type, which contains 1 byte for each char. This data type by itself may be useful for programs like python to work with because the IDE is designed to work with object-oriented data types. However, because the XBee modules do not work with string objects; they can only work with raw bytes sent to its serial buffer. Further manipulation of data is required to encode the string packet to bytes. The UART buffer on the XBee requires 1 start bit, 8 data bits, and 1 stop bit. Figure 6 shows a diagram of the XBee UART buffer with an example data packet for demonstration. Since the buffer requires the data to be in a form of bits instead of string, the string needs to be encoded with UTF-8 format. Figure 5 shows a simplified flow diagram of this requirement.

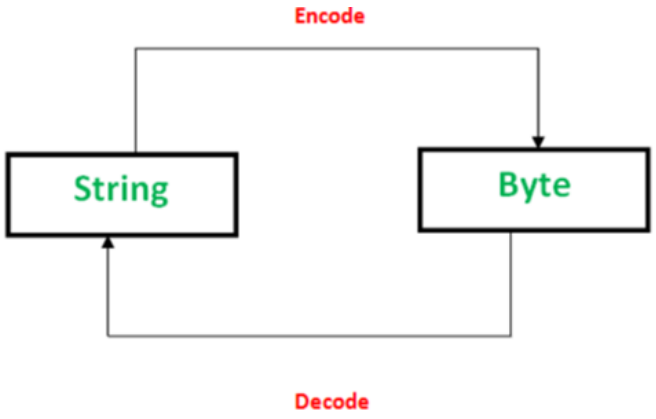


Figure 5. Strings in the packet must be encoded to Bytes to send over the serial port (UART) for transmitting data. Bytes are decoded to strings for receiving data.

Additionally, since the packets constructed are larger than the size of the buffer in terms of bits, the XBee will have to send the packets into chunks. This break in chunks may cause timing issues when the main node is attempting to spool data from the nodes; this is the motivation for implementing some sort of error checking.

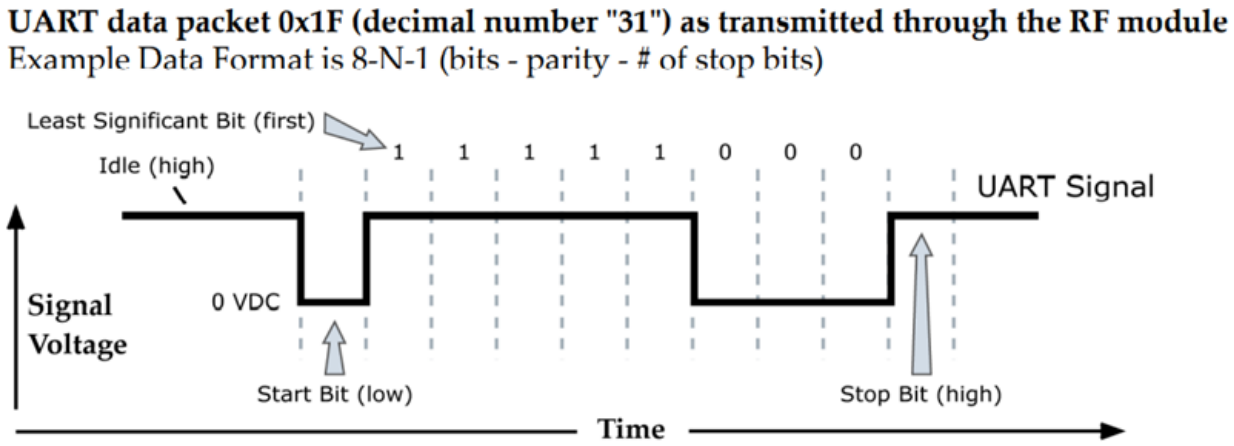


Figure 6: Diagram from: product-manual_XB_802.15.4_OEM_RF-Modules_v1.xAx.book (sparkfun.com) [5]

After the packet of data from the sensor nodes have been sent, the XBee on the main hub receives the encoded bytes from the node XBees and is converted back to a string object using python. Python is a powerful programming language that the raspberry pi can use to perform many post-processing tasks. Python works well with data type parsing and logic implementation. Python takes these packets and parses them and performs error-checking on the packet to see if there are any missing bytes lost in transmission. On successful verification of the packets, the contents are post-processed, and stored locally to a file much easier than bytes.

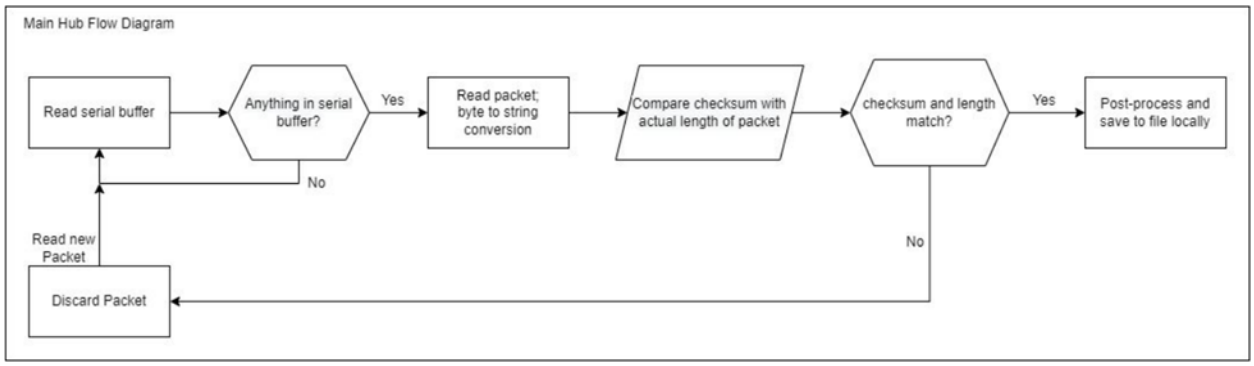


Figure 7. The python script performs a check on the packets to determine how many characters are actually in the received packet string. From figure 4, there was a checksum value inside the packet structure, that value denotes the expected number of characters that are sent over the XBees. The checksum is compared to the python scripts packet size checking functionality, if the values match then the packet did not lose any bits during transmission.

Verification and Evaluation

Photovoltaics

In confirming the power needs for the node and kiosk, both boards were powered using power supplies from the laboratory. The node microcontroller was connected to a DC power supply that gave us a current measurement along with the given operating voltage. The kiosk used a power measuring device that connects between the power supply and the raspberry pi. The power sources were chosen based on these values, in addition to the operation time of both the node and kiosk. From figure 8, the required voltage for the node was at 5.49V with .084A. The Kiosk measured voltage and current came out to be 5.13V at 1.1A.

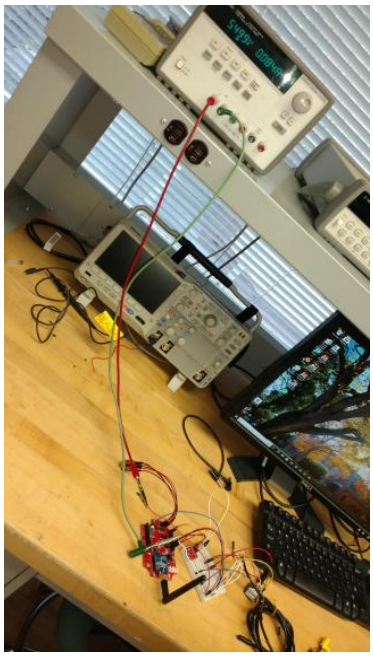


Figure 8:
Measured Voltage and current of the kiosk(left) and node(right) with all components operating.

Figure 9: (Left) Measurement of the open source voltage of the Kiosk photovoltaic under peak sunlight showing a voltage of 7.2V. (Right) Measurement of the short circuit voltage of the Kiosk photovoltaic under peak sunlight showing a current of 1.365A.

Figure 10: (Left) Measurement of the open source voltage of the node photovoltaic under peak sunlight showing 6.047V. (Right) Measurement of the short circuit voltage of the node photovoltaic under peak sunlight showing .5317A .

Charge Controllers and Power Storage

Determining the necessary power storage devices and by correlation the power charging circuits required looking into the same documentation as with the photovoltaics as shown in figure 8. Charging the battery in the node was simplified by an integrated charge controller on the Seeeduino board. The photovoltaic and battery voltages and currents were chosen based on the datasheet's recommended values. The capacity of the battery was determined by the expected runtime of the devices. The equation, $Capacity = Current(I)\ x\ Time(hours)$, was used to determine the amount of current the devices will need to run between their max estimated run time of 9am-5pm.

It was determined that the node battery would require a capacity of $.084A\ x\ 8hours \sim 100mAh$. To compensate for the days and hours where the photovoltaic would not be able to charge the battery, a capacity of 2200mAh was chosen. The storage capacity needed for the kiosk is $1.1A\ x\ 8hours \sim 8.8Ah$. For this size in capacity a 10Ah battery was chosen for the same reason as the node battery. By taking this path, the option to choose a charge controller for the kiosk battery was narrowed down as the manufacturer of the battery also makes Raspberry Pi compatible charge controllers.

Previous attempts at powering the kiosk were unsuccessful and were narrowed down to the charge controller's current limitations. With choosing the charge controller from the battery's manufacturer, we traded a compact device for a slightly larger one that could handle our requirements. There was only one set back to choosing this charge controller, which was the required input values need to be at 5V and 2A. This was solved by including a step up/step down DC-DC power converter connected between the photovoltaic and the charge controller. With this specific power converter, the output of the photovoltaic is kept at about 5V while the powerpath management system in the charge controller automatically adjusts the necessary current.

Hub Display

Verification of the Display's functionality was a one to three step process that took minimal time and effort to complete. After selecting the display we were going to implement with our kiosk, we were able to effortlessly check the viability of this component by plugging in the required hdmi and power cords to the Raspberry Pi. After reaching the required voltage and current requirements for the node and hub, the first attempt at powering the display was a success. Further configuration of the display is optional. These configured settings came via a potentiometer directly on the display to adjust brightness. This could also be done through the config file on the Pi's SD card or through other software means. In figure 11, the hardwiring for the display is shown.

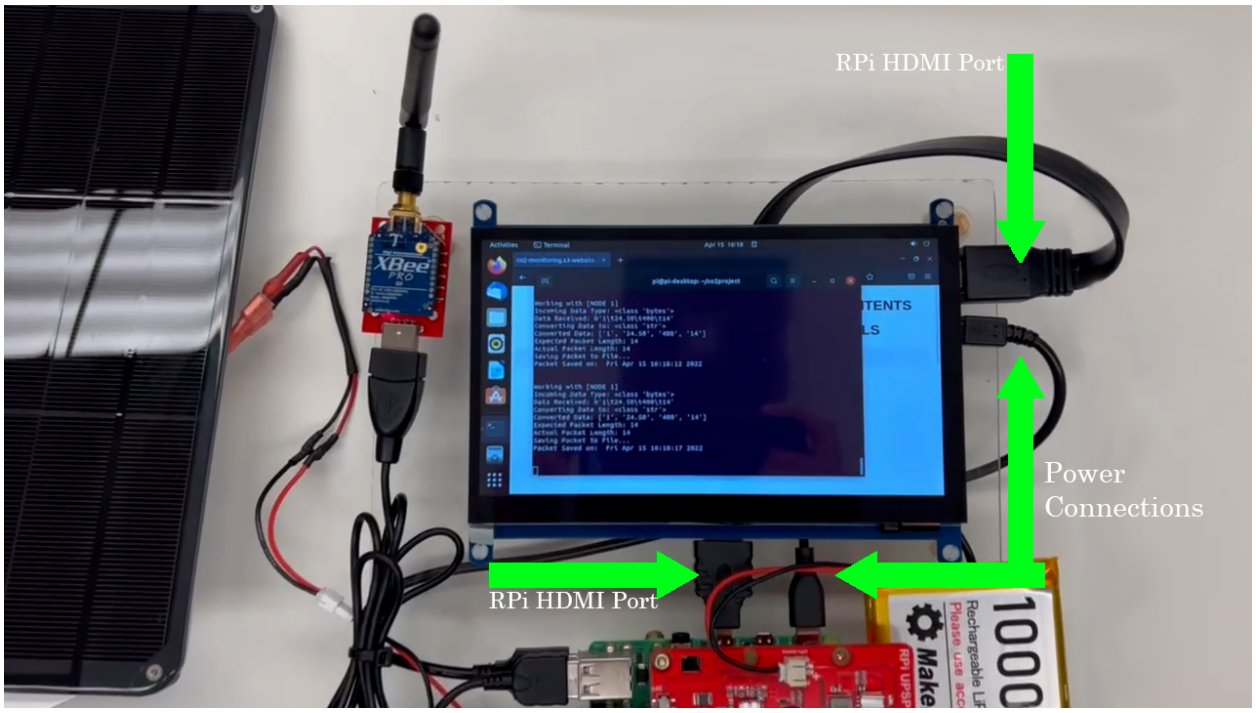


Figure 11: Diagram showing wiring for the Display and Raspberry Pi

Node Sensors

Establishing a working sensor node was the most important aspect to the project. Initial construction of the sensor node consisted of a seeeduino stalker microcontroller, a CCS811 CO2 sensor, and a DS18B20 temperature sensor. As shown in figure 12, the CCS811 connects to the seeeduino I2C pins, and derives power from the board's 3.3V supply. The DS18B20 has its data pin tied to 3.3V and a digital pin on the board with a 4.7kΩ pullup resistor. Initial test procedures involved verifying the functionality of both sensors with the same setup shown in figure 12.

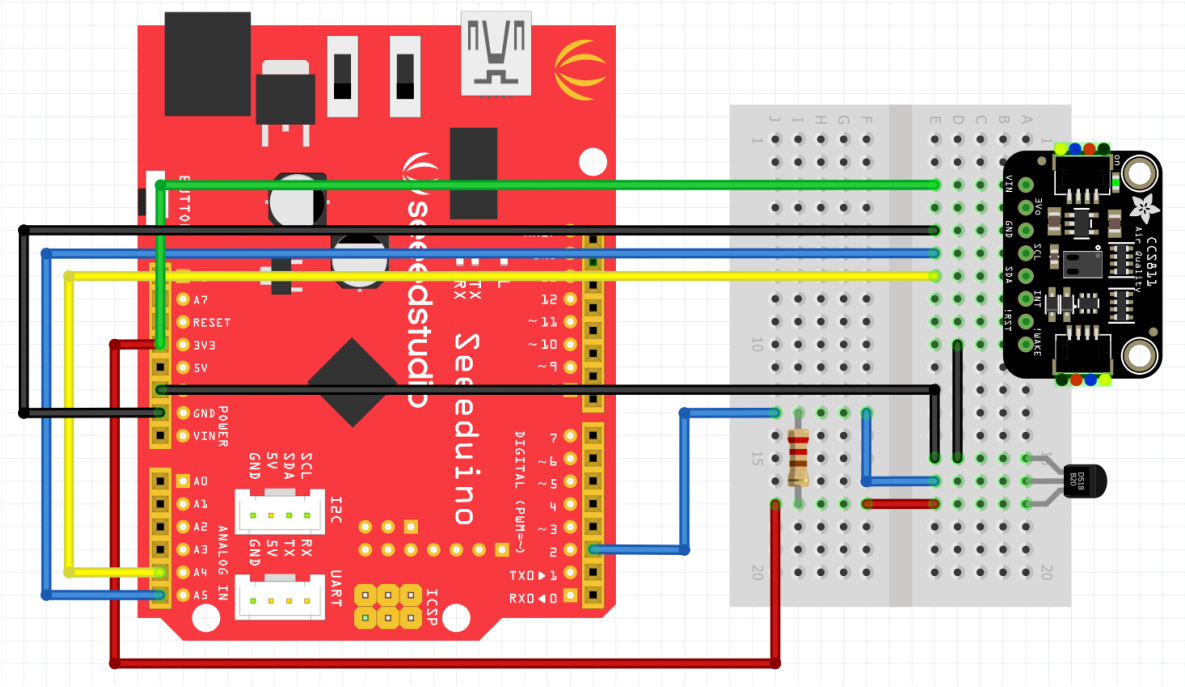


Figure 12: Diagram showing the physical connections between the CO2 sensor, temperature sensor, and the seeeduino stalker board.

Initial testing of the CCS811 CO2 sensor required configuring the I2C address. The code that was used to search through all the available I2C addresses on the I2C bus is in the Appendix B. The code was written and uploaded to the seeeduino using the arduino IDE. As shown in figure 13, there were three I2C devices found. The I2C device found at address 0x5B was the CCS811 sensor; this address would be used to read from the sensor in later test procedures.

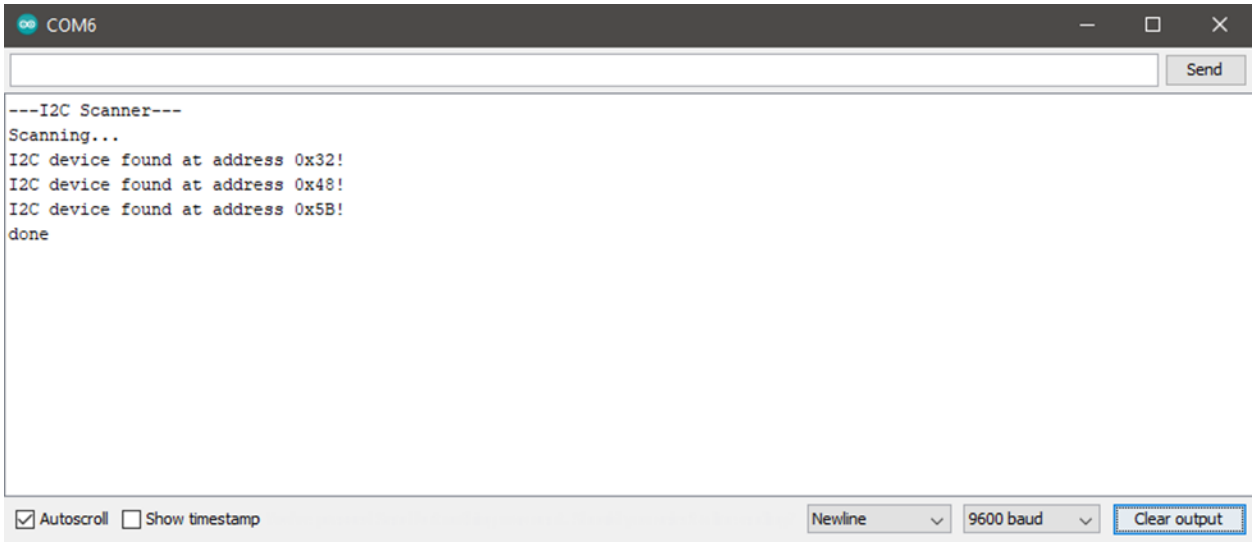


Figure 13: The arduino IDE serial monitor console showing the available I2C devices and their associated addresses.

Figure 14 shows the readings from the CCS811 sensor in the arduino serial monitor. The first column shows the CO2 readings in ppm. The average CO2 measurements floated around 405 ppm at the time this test was conducted. The second column shows the readings for total volatile organic compounds (tVOC) in parts per billion. The value of this measurement only varied between 0 to 1. However, these measurements would not be incorporated into the final design of the system because the data was irrelevant to the project objectives. The third column shows the time in milliseconds at the time an instance of data was acquired. The code for running this test can be found in the appendix C. These data points show that the CCS811 sensor is functional and capable of providing real-time CO2 readings.

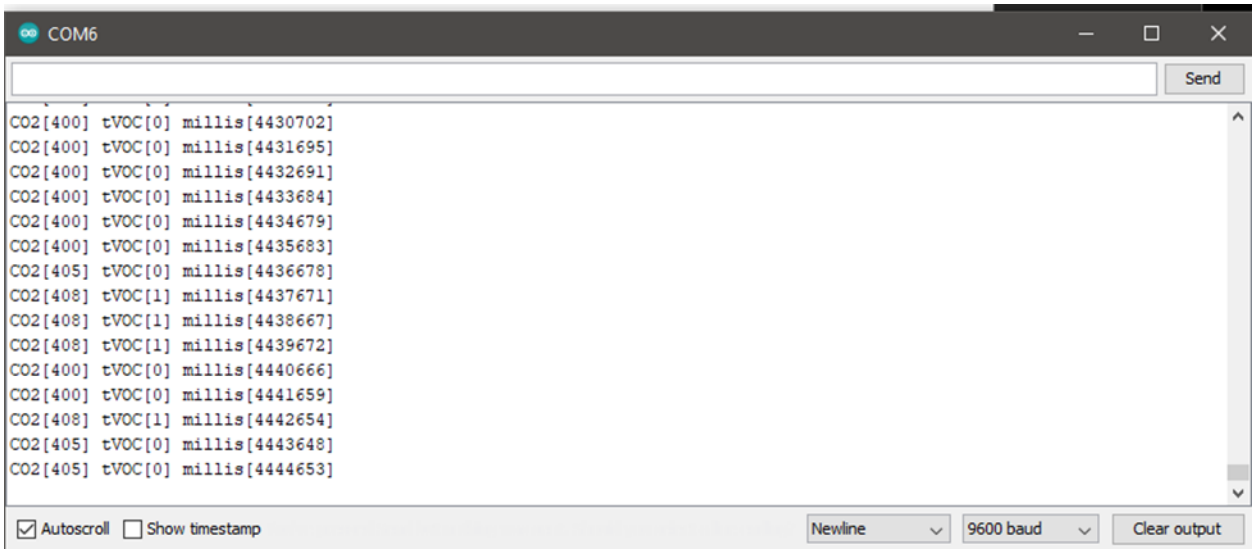


Figure 14: Example Serial Monitor output with CO2 and tVOC (total Volatile Organic Compounds) sensor readings from the CCS811.

The DS18B20 temperature sensor measurements are read by the seeeduino and programmed in the arduino IDE in similar practice to the CCS811 CO2 sensor. As shown in figure 15, the ambient temperature of the test environment was estimated to be around 20℃ with slight variations to the tenth of a degree in celsius. These temperature readings show that the DS18B20 sensor is functional and sufficiently accurate for the purposes of this project. The code for reading temperature measurements is found in Appendix D.

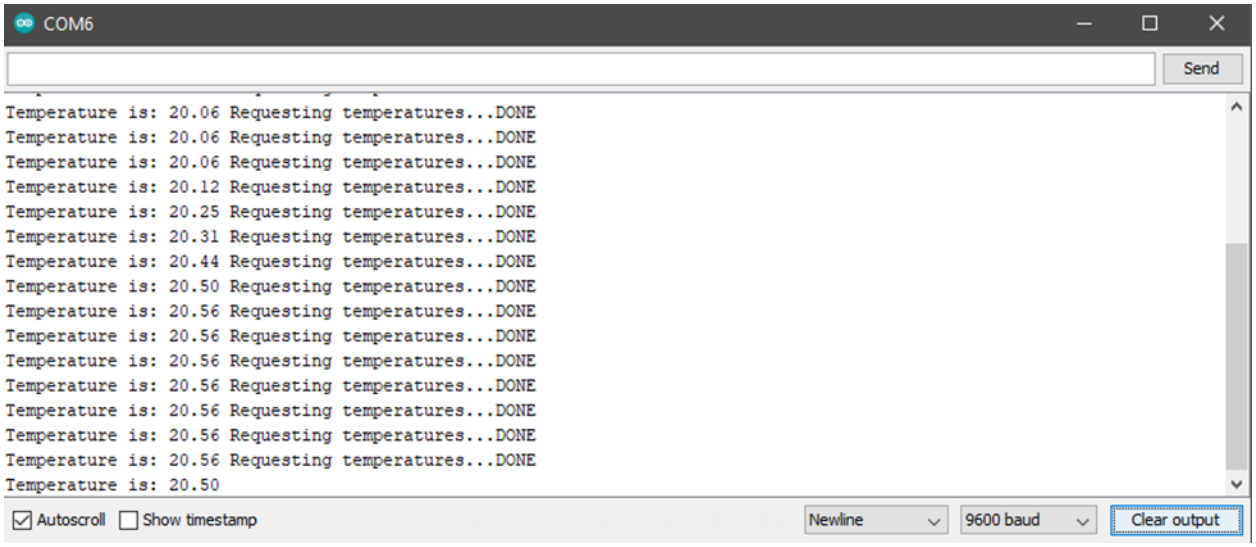


Figure 15: Example Serial Monitor output with temperature sensor readings in degrees celsius from the DS18B20

The seeeduino has shown to be capable of reading in live measurements by checking the values in the serial monitor of the arduino IDE. However, taking measurements continuously over a long time period in an outdoor environment is not practical when the system is tethered to a computer. The seeeduino has a built-in micro SD card reader. This was used for testing the sensors over a long period of time. All the data was collected and stored into a text file. After letting the seeeduino collect the data, the contents of the micro SD card can be plotted to show the behavior of the sensors over time. Figure 16 shows the contents collected from the CO2 and temperature sensors over time.

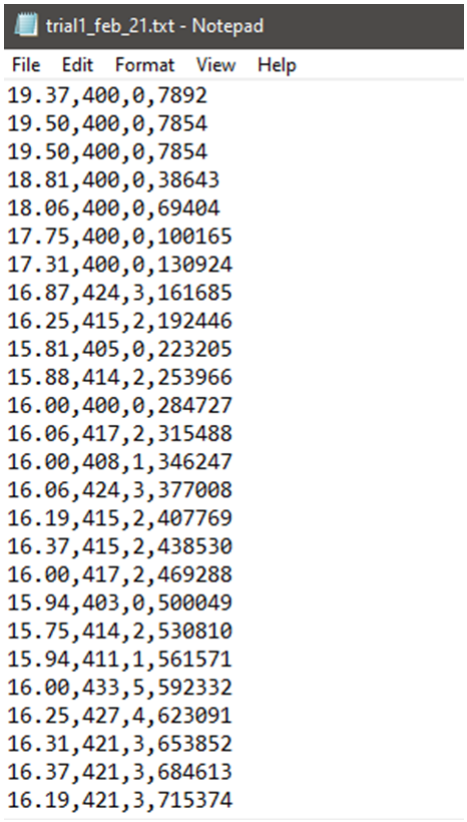


Figure 16. Sensor data collected and stored in a txt file on an SD Card. The 1st column shows temperature data in Celsius, the 2nd column shows co2 contents in ppm, the 3rd column shows total volatile organic compounds (tVOC), and the 4th column shows time in milliseconds. Each successive row represents another point in time when data was collected.

After verifying the functionality of the sensors, the next step was to gauge the variability of the sensors as they take measurements for an extended period of time. The testing period spanned 6 hours from approximately 11:30 am to 5:30 pm on a relatively clear day. The seeeduino and temperature sensors were placed outside and powered by an external battery pack. The solar and battery power components were implemented at this point, as the focus was to verify the temperature and CO2 sensors operated as intended. Throughout the experiment, the seeeduino pulled readings from both sensors at the same time in 10 minute intervals. As shown in figure 17 the sensor exhibits a lot of variation in its measurements throughout the day. The measurements steadily increased from 400 ppm and peaked within 3 hours to approximately 1,500 ppm. Measurements thereafter steadily decreased until the measurements leveled off to an average of 800 ppm. There are no conclusive observations to be made based on the plot, but it does show that the sensor is able to respond to changes in the air from an outdoor environment. These variations may be attributed to factors such as cars driving by or changes in ambient temperature. Considering these possibilities, a more controlled experiment may be required to make more conclusive observations on the data.

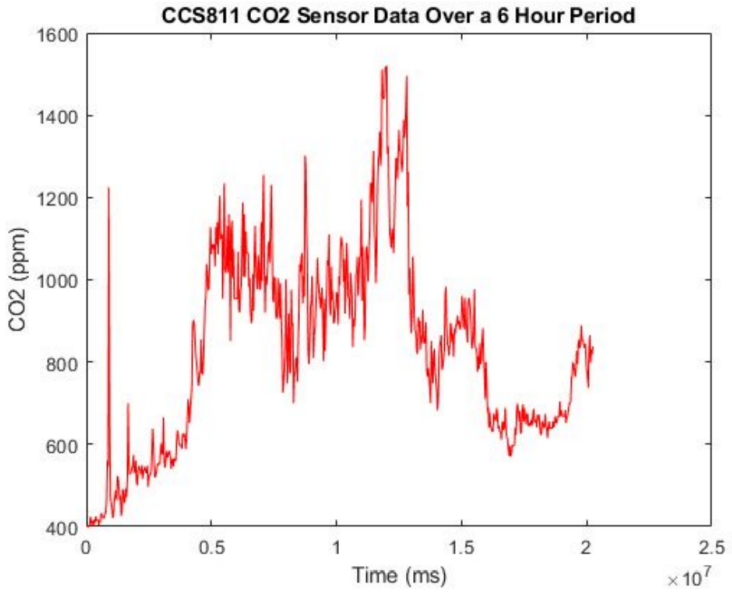


Figure 17: The plot shows the trend of outdoor CO2 measurements measured by the CCS811 sensor over a 6 hour period

The temperature recordings shown on the plot from figure 18 ranged from 15℃ to approximately 34℃ throughout the testing period. The shape of the overall curve indicates that the ambient temperature was steadily rising for the first couple hours and then spiked during the middle of the day, from 1 pm to 3 pm, and then the temperature tapers off after 3 pm. This plot of data shows the DS18B20 sensor can respond to changes in outdoor temperature.

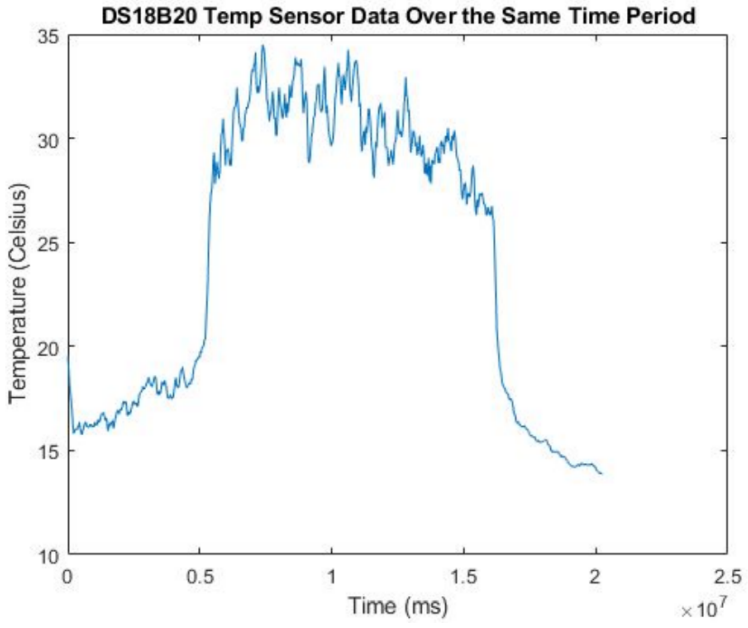


Figure 18: The plot shows the trend of outdoor temperature measurements measured by the CCS811 sensor over a 6 hour period

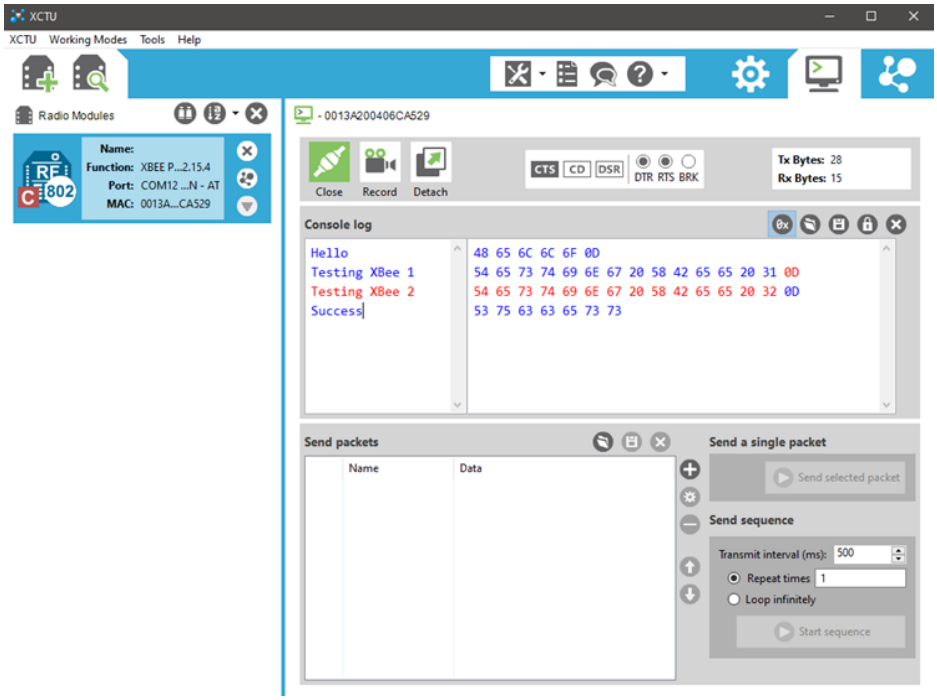


Figure 19: Console log showing data transmitted and received from the "coordinator" XBee. 28 Bytes were sent and 15 Bytes were received

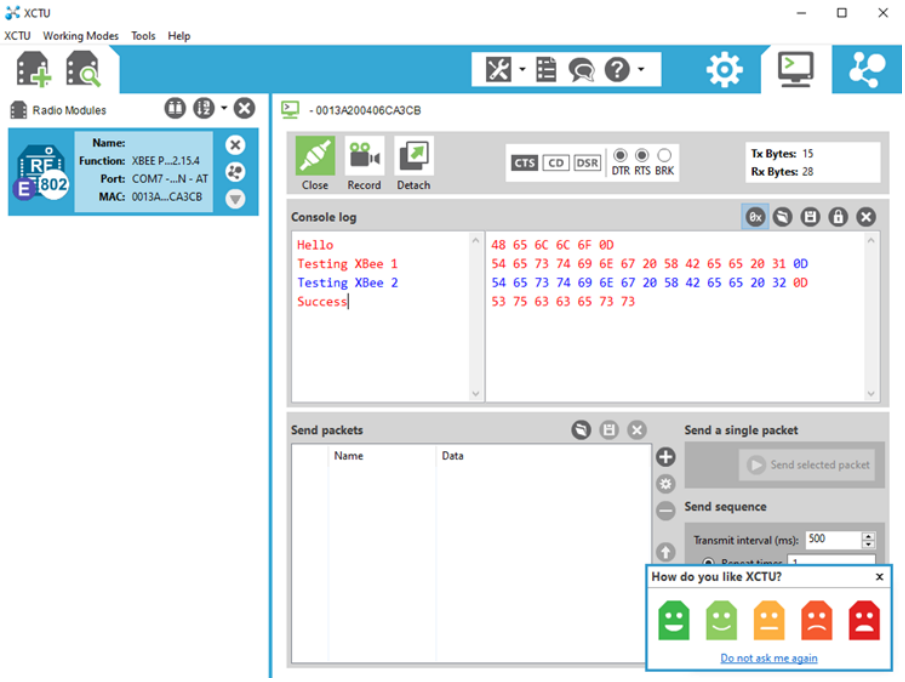


Figure 20: Console log showing data transmitted and received from the "end device" XBee. 15 Bytes were sent and 28 Bytes were received

After performing tests on the CO2 and temperature sensors to verify their functionality, the next step was to configure the wireless communication part of the system. Two XBees were acquired to test the feasibility of a wireless network connection. The two XBee modules were configured independently using the XCTU software. This software ran on a Windows 10 machine, so the XBee modules were wired onto USB breakout boards. An example of the configuration settings are shown in figure 3. After configuring the modules, the next step was to send a message from one to the other. This action was performed by typing in a string of text in the XCTU console shown in figure 19. The messages typed into the console were highlighted in blue text. Figure 20 shows the received message, which is denoted by the red highlighted text. Establishing a connection that enables one XBee to send messages to the other was a success.

The next step in the test procedure was to get the sensor data transmitted wirelessly from one XBee to another. In this test, one XBee was still attached to the Windows 10 machine via the USB breakout board. The other XBee was attached to the seeduino's built-in Bee series socket. With this setup, the seeeduino collects sensor readings and writes them to the attached XBee via serial communication. Any data that is successfully received wirelessly on the other XBee would be populated on the XCTU console. The code for this implementation is shown in the Appendix E. Figure 21 shows messages and sensor data from the sensor node to the XCTU console. The console updates its contents immediately after the sensor node transmits its data.

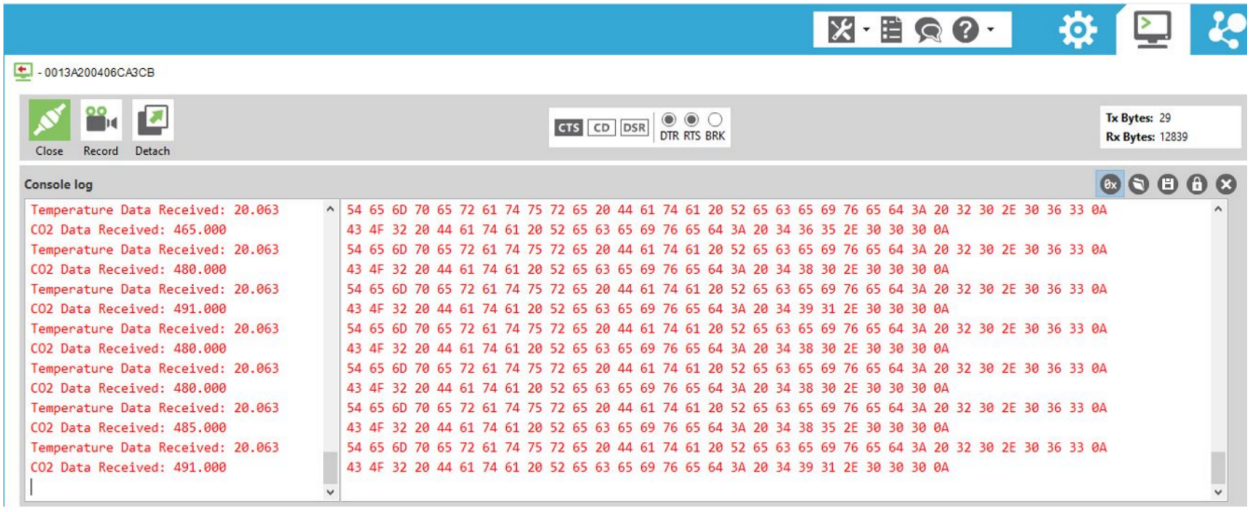


Figure 21: Console log showing CO2 and temperature sensor data received from the "end device" XBee. A total of 12839 Bytes of data was received. Reported temperature was consistently ~20C. Reported CO2 ranged from 400 to 500 ppm

A radio range test was conducted to see how far apart the sensor node and the main hub can be while sustaining a good connection. In the range test, 50 packets were sent for each test case. The tests were run with three ranges: 10 meters, 25 meters, and 50 meters. 3 Trails were performed for each range. As shown in figure 22, the radio range test provides signal strength statistics in dBm, and the success rate of packets sent and received are plotted over time. The results of each test are tabulated in table 2. The results show that signal integrity starts to degrade when the XBees are approximately 25 meters apart.

Figure 22: Radio range test showing the signal connection strength and packet loss statistics

Range		10m	25m	50m
Packets Received	Trial 1	50	45	18
	Trial 2	50	48	29
	Trial 3	50	45	24
Local RSSI (dBm)	Trial 1	-63	-76	-86
	Trial 2	-65	-72	-79
	Trial 3	-60	-69	-83
Remote RSSI (-dBm)	Trial 1	-65	-82	-85
	Trial 2	-66	-72	-77
	Trial 3	-62	-72	-81
Success Rate	Trial 1	100%	90%	36%
	Trial 2	100%	96%	58%
	Trial 3	100%	90%	48%

Table 2. Wireless range test results tabulated. Range test was performed under 3 ranges (10m, 25m, and 50m). A total of 50 packets were sent for each trial run. The table shows for each trial, the number of packets received out of the 50 packets sent, the Received Signal Strength Indicator levels (RSSI) in dBm, and the success rate of the range test.

After verifying that the CO2 and temperature measurements can be transmitted and received between two XBees, a prototype of the system containing all components was constructed. This prototype is shown in figure 23. The seeeduino has a built-in Bee series socket that accommodates one XBee. The other XBee was wired to the UART ports of the raspberry pi through the breadboard. The sensors were wired on the breadboard to the ports of the seeeduino as shown in figure 12.

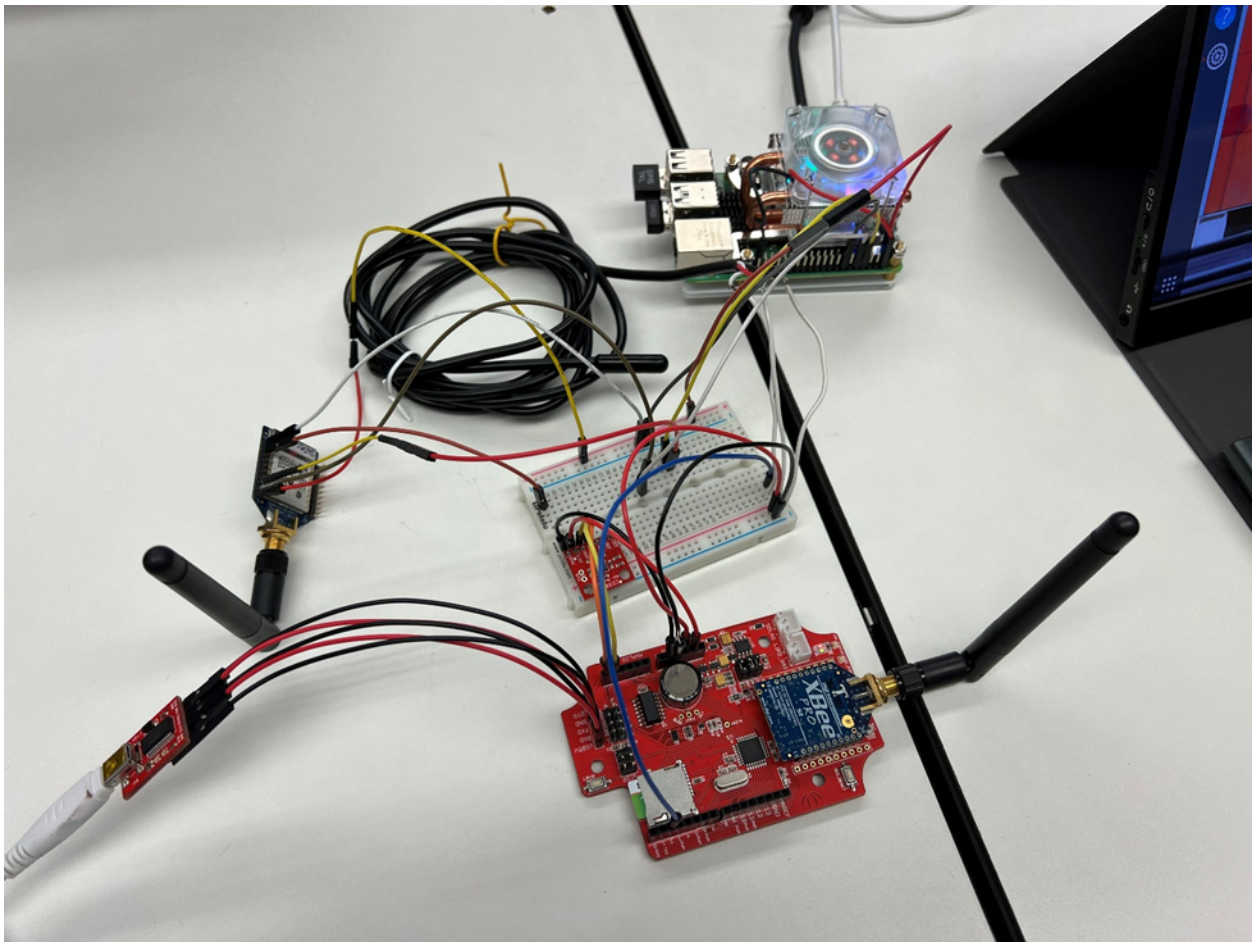


Figure 23. Physical prototype of the system containing one XBee module and CO2/temp sensors attached to the seeeduino & the one XBee module wired to the raspberry pi

Main Hub

As shown in figure 21, all sensor measurements were sent between the XBee attached on the seeeduino and the XBee attached to a Windows 10 machine via a USB breakout board. The final implementation of the system should have the seeeduino send sensor measurements to the raspberry pi. To enable wireless communication for the raspberry pi, the use of Python and bash scripts were necessary. Appendix G contains bash script commands that download and install the required libraries for Python. The Python script that is responsible for receiving data from the sensor node is found in Appendix F. The script is also responsible for checking the received packet for any errors. Due to reduced signal strength and packet loss as the distance between the XBees increased, details are tabulated in table 2, it is important to consider that there are chances that some bytes of the sensor data will be lost during transmission.

Two examples of packets produced by the seeeduino are shown in table 3. Following the flow diagram shown in figure 4, the packet consists of 4 components: the NODE ID, temperature readings, CO2 readings, and the checksum. The NODE ID is used to differentiate which node sent that specific series of data. As discussed in the software description, the entire packet is a string datatype. The more characters that need to be sent, the greater the checksum will be. According to table 3, example 2 has a checksum greater than example 1, and that is attributed to a greater amount of bytes contained in its packet.

	NODE ID	\t	TEMP (°C)	\t	CO2 (PPM)	\t	CHECKSUM
Example 1	1		22.55		420		14
Example 2	33		31.87		1132		16

Table 3. The packets sent wirelessly over the XBees are structured in the same manner as shown in the table. This table is a duplication of the one found in figure 4.

The Python script from Appendix F follows the flow diagram shown in figure 7. Each packet received on the raspberry pi is parsed and analyzed to ensure no bytes were lost during transmission. The checksum value shown in table 3 varies depending on the number of bytes that make up the packet that needs to be sent. The raspberry pi on the main node compares this checksum value from the packet, and compares its value with the actual length of the packet that is received. If the checksum doesn't match the length of the packet, then the packet can be considered incomplete as it is missing some data. As shown in figure 24, the terminal prints out each sample of data the raspberry pi receives, and displays a message indicating whether the data was saved or rejected.

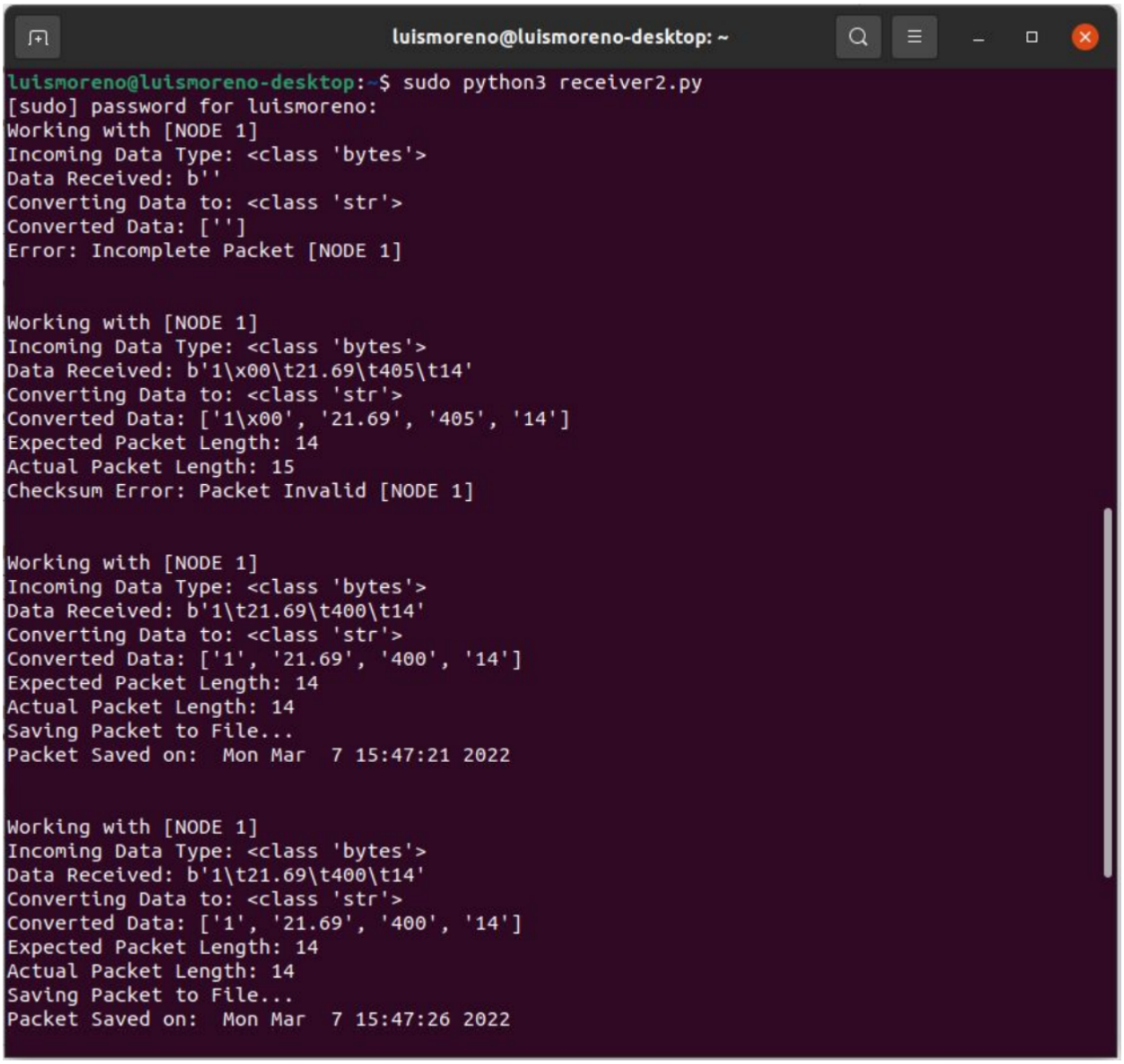


Figure 24. Terminal showing execution of the python script from Appendix F.

The packet data that was received by the raspberry pi is printed on the terminal. The packets are in a form of bytes, but the script converts them to strings and parses the data for post processing. To understand the structure of the packet, refer to figure 4. If the script determines that the received packet is valid, the script prints a message indicating that the data was successfully saved to a file locally. If the script determines that the packet is invalid, the script prints an error message indicating that there was either a loss of bytes or an excess of bytes in the packet.

After the packet is validated, the data is parsed and saved to a text file shown in figure 25. The first column is the ID of the node, the second column is the temperature data, the third column is the CO2 data, the fourth column is the checksum value, the fifth column is the year, the sixth column is the month, the seventh column is the day of the month, the eighth column is the hour, the ninth column is the minute, and the tenth column is the second.

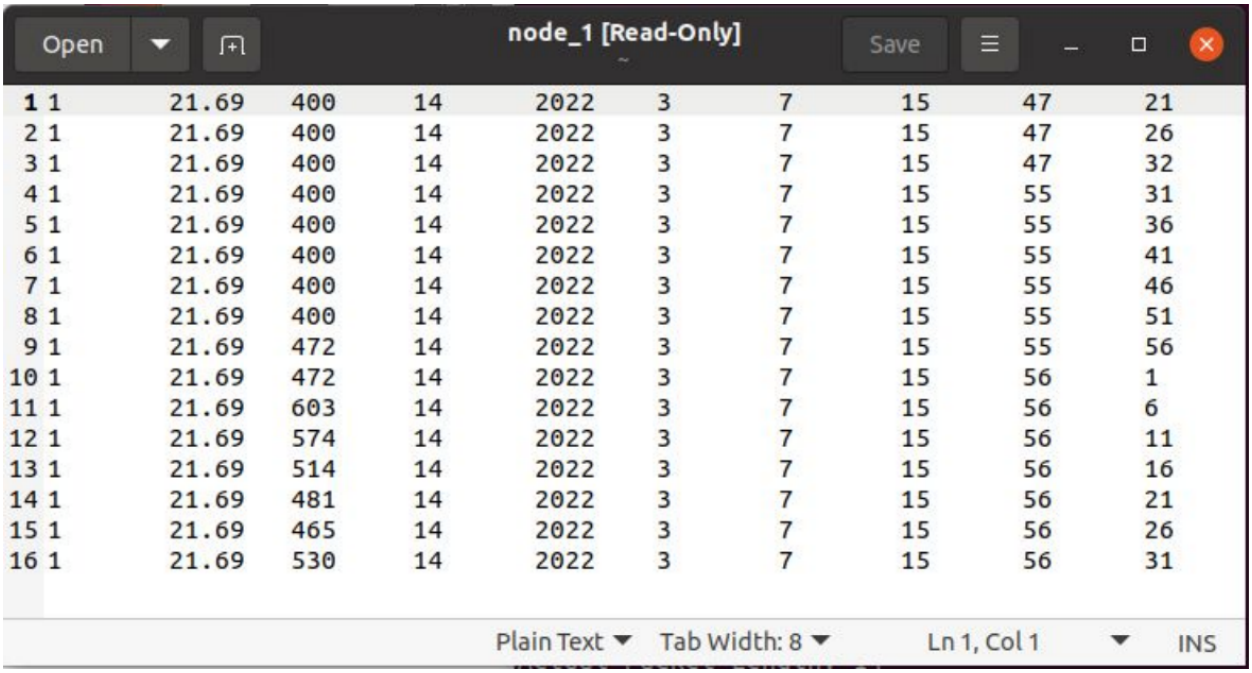


Figure 25: Example sensor data stored in a text file on the raspberry pi

The bash script from Appendix G also serves as a means to install the files required to connect the raspberry pi to the website database. The bash script also contains the access key and the secret access keys that serve as login credentials to the aws service. The script also contains configurations for the region and time. Figure 26 shows directories of the website database after successful installation and configuration. The raspberry pi now has privileges to write sensor data to the website database.

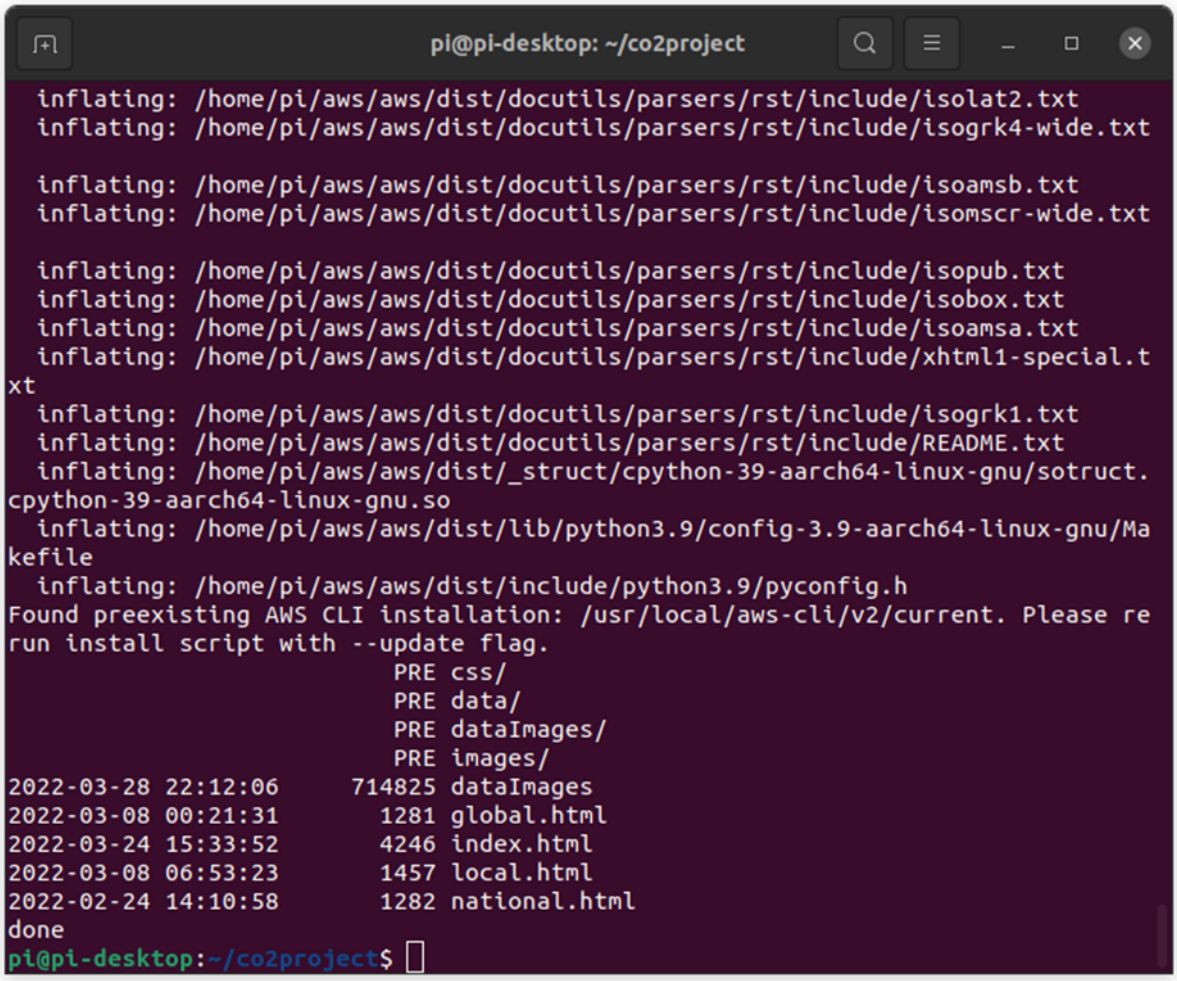


Figure 26. Raspberry pi terminal showing it has access to website directories

The final build of the sensor node is shown in figure 27. All components are mounted onto an acrylic sheet in a compact form factor, which allows the node to be relocatable with ease. The CCS811 CO2 sensor and the DS18B20 temperature sensor are both mounted to the seeeduino stalker. The photovoltaic panel (evaluation shown in figure 10) and the 2200mAh LiPo battery powers the seeeduino, XBee module, and sensors.

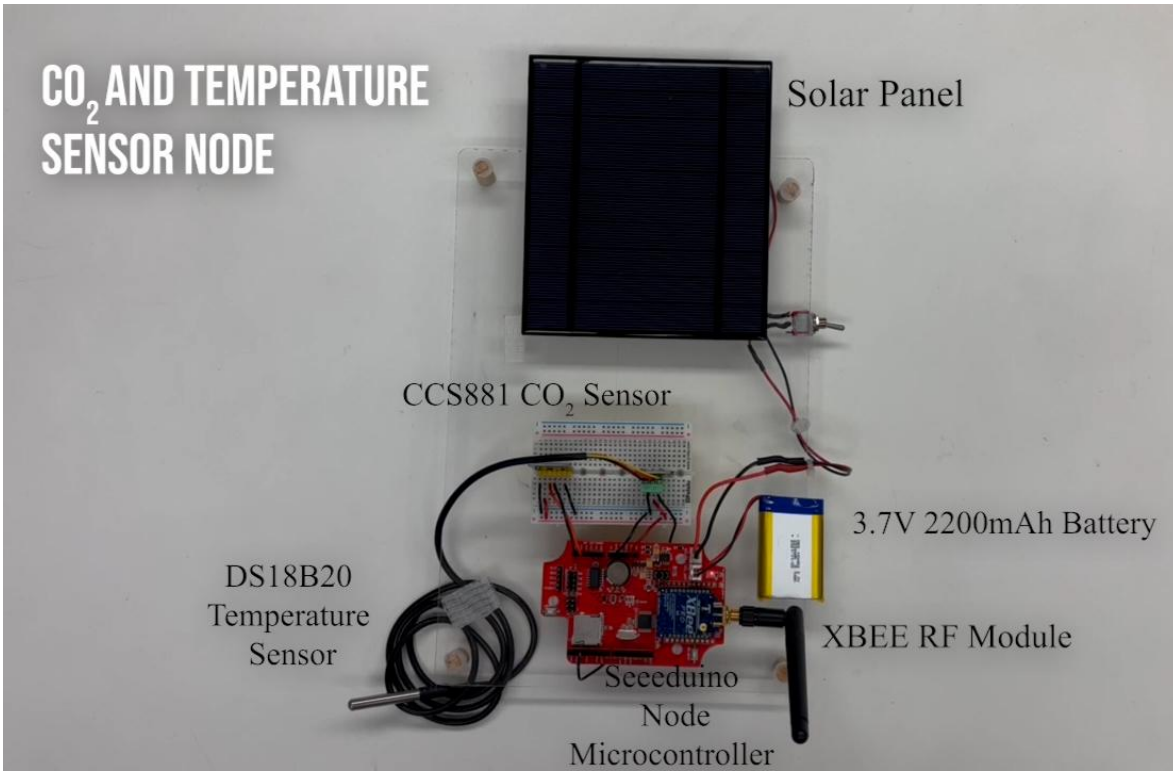


Figure 27: Completed sensor node with all sensors and solar/battery power components connected

The final build of the main hub, shown in figure 28, is also mounted onto an acrylic sheet. Since the power requirements for the main hub are significantly greater than the sensor node, the 9W photovoltaic panel (evaluation shown in figure 9) needed to be mounted separately. This is not an issue as the main hub will be stationary. Figure 28 also shows the 7-inch display is capable of showing contents on the screen.

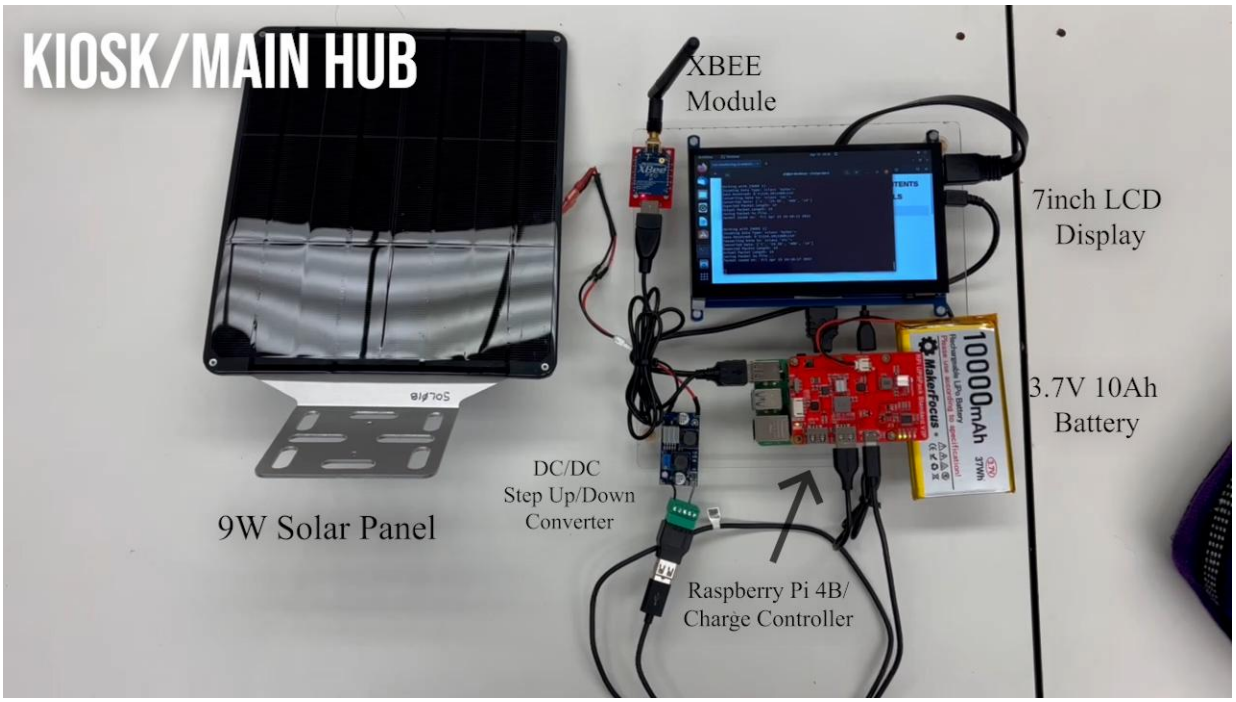


Figure 28: Complete build of the main hub with all components attached and display showing functionality of the system

Automation

Our system depends on the automation feature to present images and figures in real time. The code for all the automation scripts from Python and Bash programming languages, will be attached in Appendix A. This system is made up of three automated programs which are the Global referencing script, national referencing script and the local structure automation script with specific timings. All programs must be executed manually at the end of the initial setup phase in the terminal which at that point no longer requires human interaction.

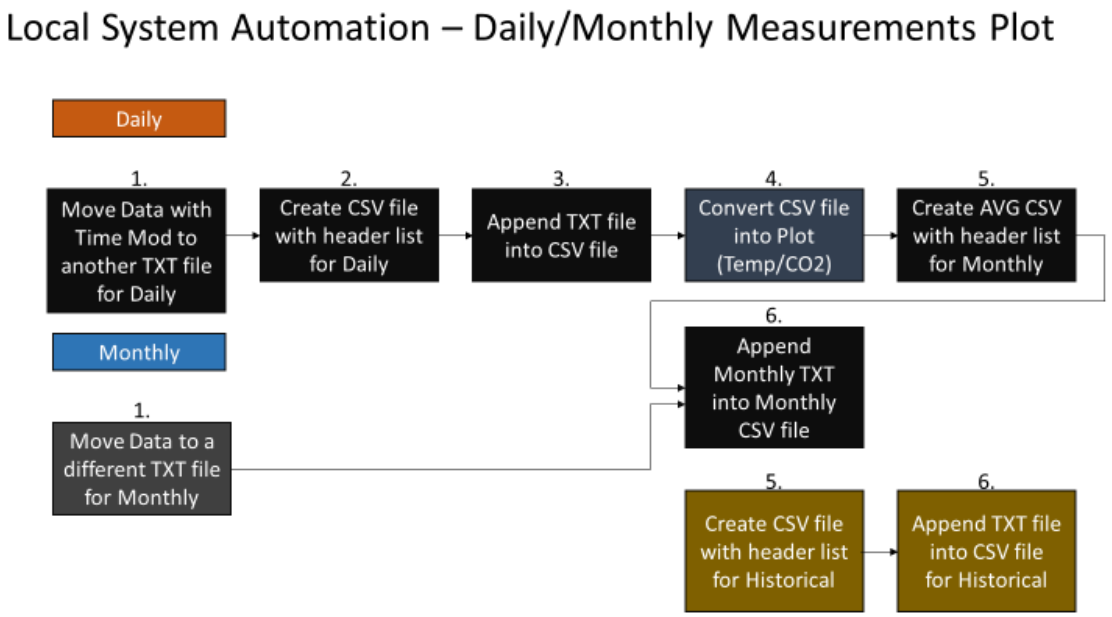


Figure 29: Local Automation of Plot Creation from Stored Data Received from Node - Daily

The local portion of the automation is handled in an organized manner to execute the file perpetually without error, assuming all conditions are uninterrupted. In figure 29, data from the node is pulled into a local directory on the main hub which then allows the first script to modify the time into a standard format for proper x-axis layout on the plotting stage. That data is appended after a CSV header is freshly created. Once the graph is pushed onto the main hub's directory, the plot is transferred onto the database. A CSV file is prepared for the monthly readings while simultaneously a historical file is stored in the database as a backup and for future customer usage.

Daily, and monthly readings have an interval of refreshing at the end of every day. Each daily graph displays the 3 daily measurements that are spread across the length of 25 to 30 minutes. Monthly plots record the average value from the daily file. The yearly file calculates the average value for the month which only refreshes at the end of each month for its plot. A Kiosk Mode Page script retrieves data from each section (local, national & global) and translates the content into an image. This is all automatically executed at the end of each day under the local bash script.

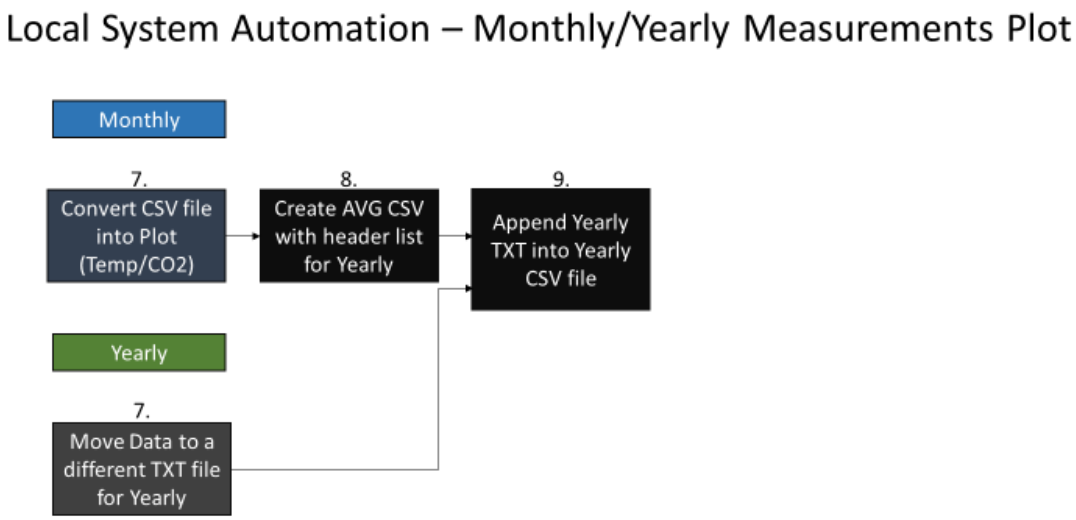


Figure 30: Local Automation of Plot Creation from Stored Data Received from Node - Monthly

Similar to the daily interval, the monthly measurements undergo the same process with the plot conversion from raw data, as shown in Figure 30. A monthly plot is shared onto the database for the local page on the website.

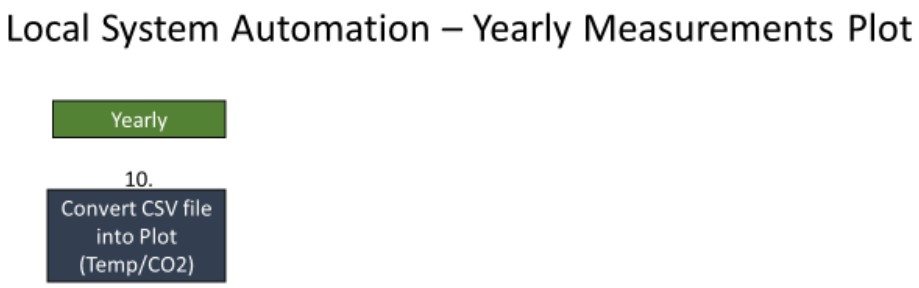


Figure 31: Local Automation of Plot Creation from Stored Data Received from Node - Yearly

As seen on Figure 29 and Figure 30, a yearly graph is extracted from the main hub totaling 3 pairs of images for the temperature and carbon dioxide measurements, as shown in Figure 31. Any viewer will be able to visualize the trend of the CO2 impact on a daily, monthly, and yearly basis.

Kiosk Mode Page

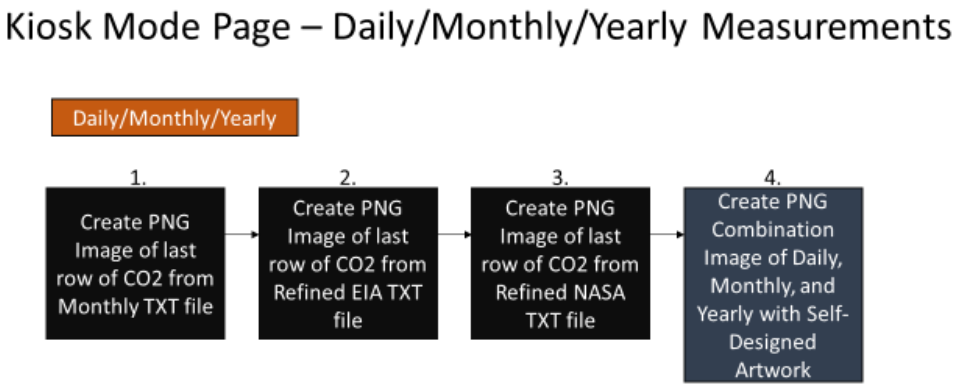


Figure 32: Diagram of Kiosk Mode Page Creation

Our project's objective is to show the viewer the local, national, and global carbon dioxide readings in a visually appealing manner. In Figure 32, a carbon dioxide value from the last row is extracted from every data file type which is converted into a numerical image. These newly created images are combined with the artwork from Senior Design 195 into a Kiosk mode PNG picture shown in Figure 33.

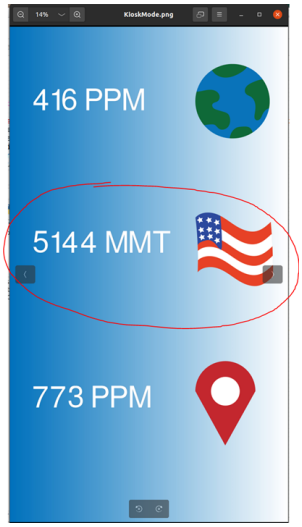


Figure 33: Kiosk Mode Page Layout for Full Screen on Standby

There are 2 different types of units due to how the readings are calculated. In Figure 33, PPM is the average measurement of CO2 content within a certain volume while MMT is the total CO2 produced within a nation. Improvement of how much CO2 is detected or generated within certain parts of the world will be discussed in the Future Works.

Web Scraping

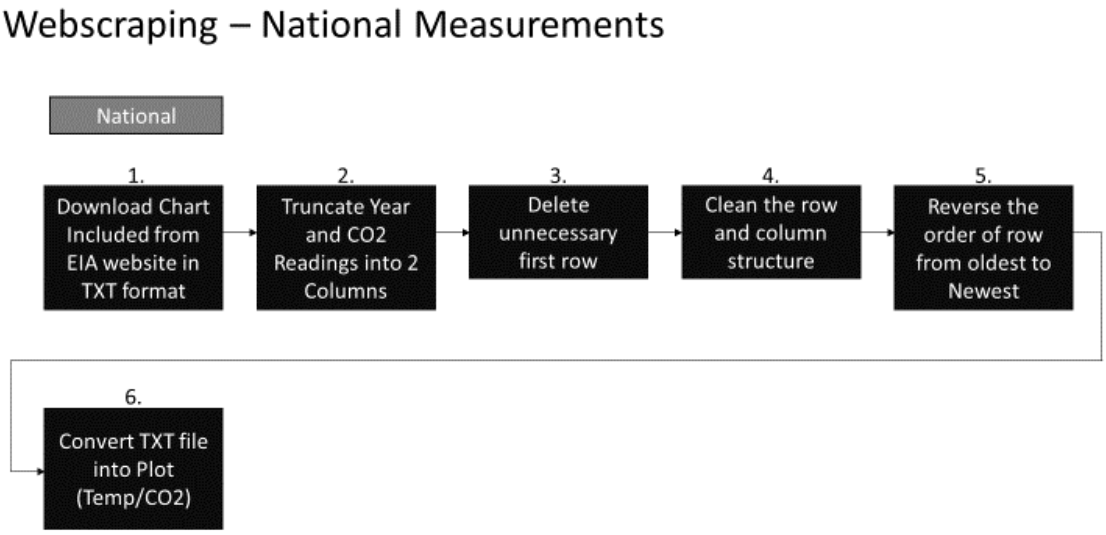


Figure 34: Flow Diagram of National Webscraping Process

A graph from EIA, Energy Information Administration, includes historical data from the 1950's up until today which is important for our project's goal. Since EIA requires an API to access their data, webscraping was the alternative choice. Figure 34 shows the multilayered process of refining the converted TXT web page into two clean columns consisting of year and carbon dioxide measurements. The final result is converted into a graph for the national page of our website as shown in Figure 35.

The national Bash file is responsible for refreshing the process twice a year due to the government updating the information of CO2 information once every year.

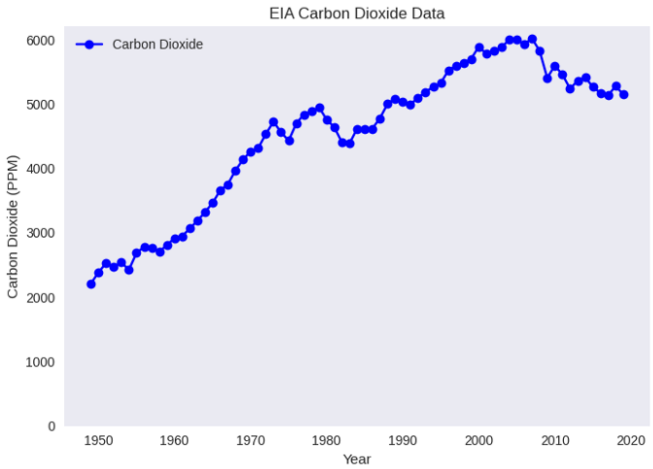


Figure 35: Automated extracted EIA Data Converted into Plot

To maintain consistency in the units of carbon dioxide, the preference is PPM, but unfortunately, national data is the result of the collection of carbon dioxide rather than the average amount of Carbon Dioxide found within the U.S., as shown in Figure 35.

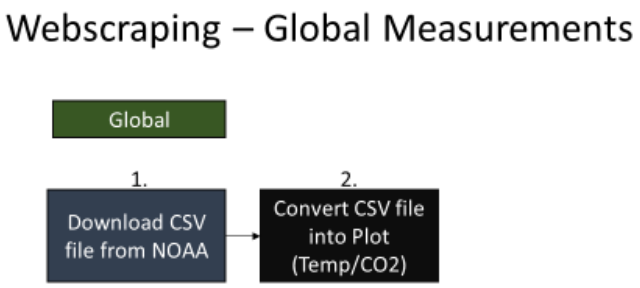


Figure 36: Flow Diagram of Global Webscraping Process

Global readings from NOAA are transferred from the site onto the main hub's directory through a bash file. The CSV file truncates the comments to acquire two strips of data found similarly on the national webscraping process, as shown in Figure 36. A plot is produced and stored onto the global page of the website as shown in Figure 37.

The global Bash file is responsible for refreshing the simple process twice a month.

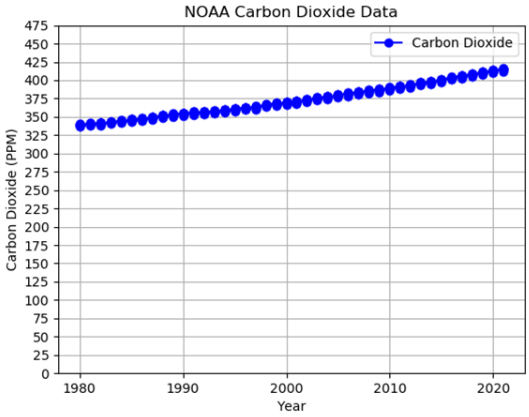


Figure 37: Automated extracted NOAA Data Converted into Plot

Apart from the local and national measurements, NOAA's global data is an important example to convey to the average passerby the importance of carbon dioxide produced yearly relevant to temperature as observed on our local sensors as shown in Figure 37.

Website Software Additions

The website is found to be the following link. Our design objective for the website is to provide a platform for displaying data collected from government sources and local data collected from sensors. Our website has a general navigation page that also allows users to gather additional information about the project. In our initial objectives we planned to include a “contact us” section, but in consideration of keeping the maximum amount of our monetary budget for the acquisition of hardware, we were unable to implement a working backend API for the email client. In future implementations we can pursue other systems for allowing users to contact the team. Without any budget limitations we can rely on Amazon Web Services’ email API to have a database with all Contact Us inquiries.

For future teams there is always the possibility of completely designing the server backend on a local device. There will be issues of obtaining domain names, and providing adequate security and data protection. A local server will also allow data processing to be implemented on scalable hardware, and save power draw for creating and storing the graphs on the main hub as we have currently.

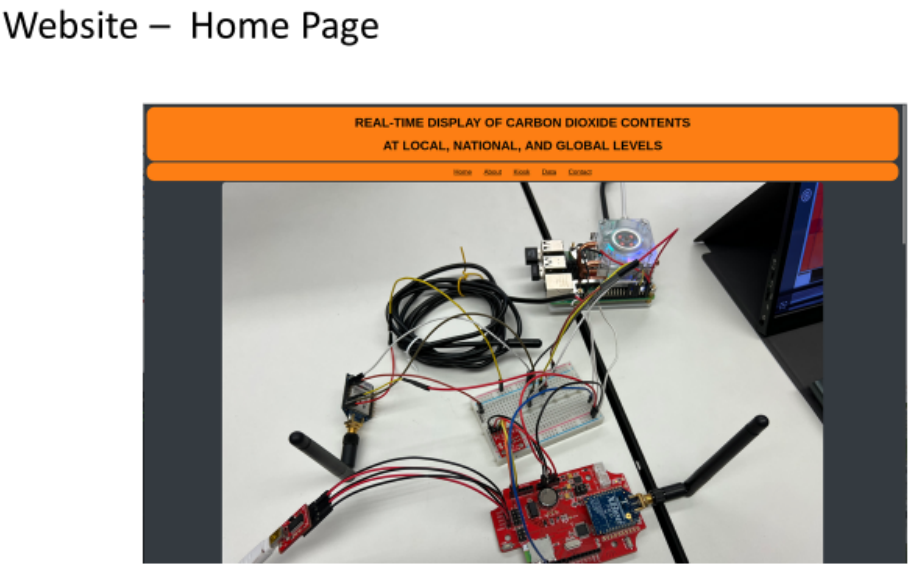


Figure 38: Home Page of the Front-end of the Website (1)

An early stage of our senior design project’s hardware is displayed on the main panel of our website which includes the main hub, and node with functioning temperature and carbon dioxide sensors, as shown in Figure 38.

Website – Home Page

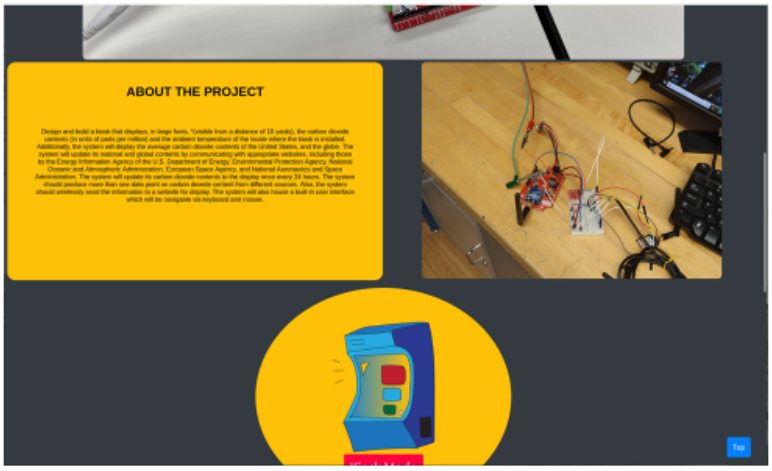


Figure 39: Home Page of the Front-end of the Website (2)

A description of the 195 project is included alongside an early stage prototype of our node, as shown in Figure 39.

Website – Home Page

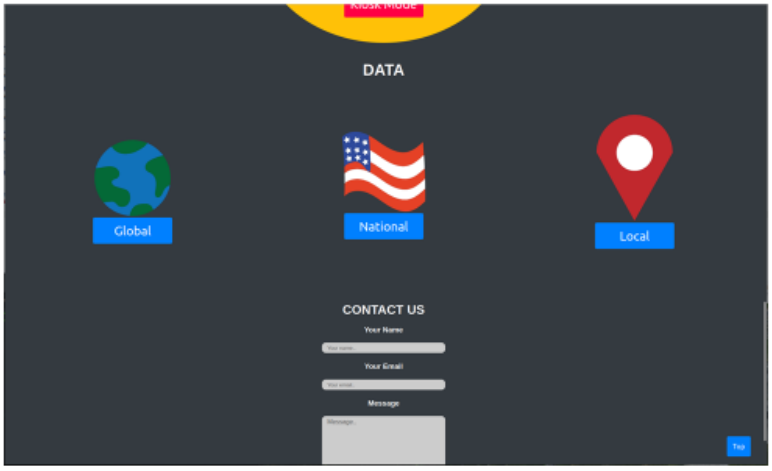


Figure 40: Home Page of the Front-end of the Website (3)

In Figure 40, user navigation between the 3 locales and kiosk mode allow for an organized portrayal of CO2 content awareness.

Website – Home Page

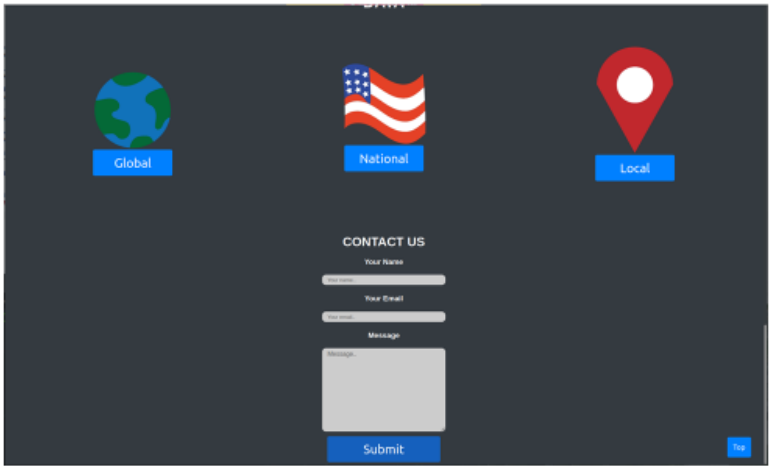


Figure 41: Home Page of the Front-end of the Website (4)

A Contact Us feature was planned at the beginning of the project but due to issues stated earlier with implementations only possible through paid services, it was not feasible due to time constraints. In Figure 41, the Contact Us section includes information that would have notified the client of information people are interested in.

Website – Global Page

Figure 42: Global Page of the Front-end of the Website

A dedicated page is assigned for the global region of measurements with only a plot image that is received from automation, as shown in Figure 42.

Website – National Page

Figure 43: National Page of the Front-end of the Website

Similar to the global version, a dedicated page is assigned for the national region of measurements with a single plot image that is received from automation, as shown in Figure 43.

Website – Local Page

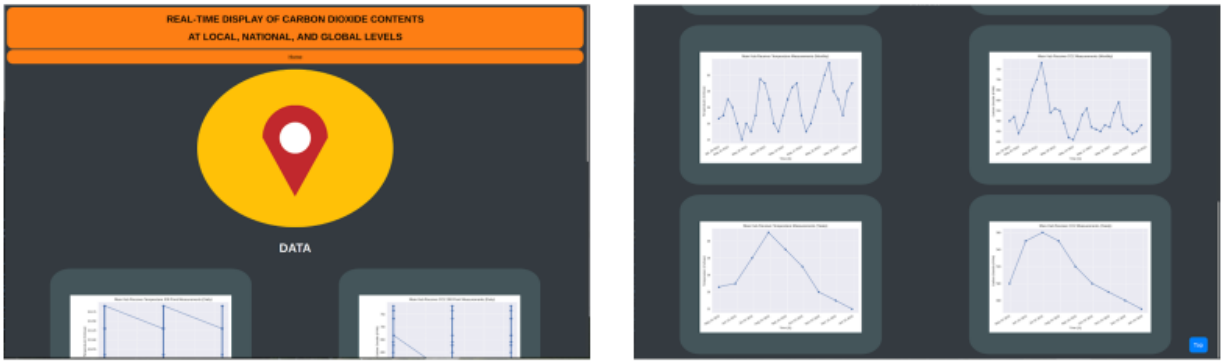


Figure 44: Local Page of the Front-end of the Website

This section, Figure 44, presents 6 figures which are the CO2 and temperature readings for daily, monthly and yearly plots. These are fake data points due to only having 2 days of recording measurements, as shown in Figure 44. Automation is tested to be successful and this would be the outcome.

Website – Kiosk Page

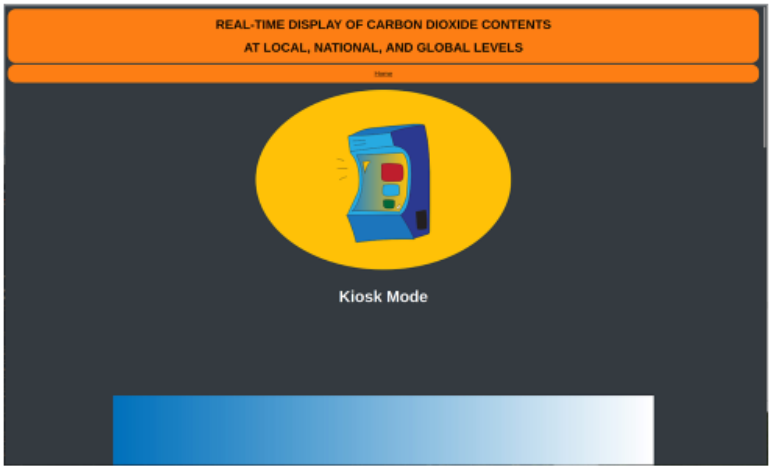


Figure 45: Kiosk Mode Page of the Front-end of the Website

A kiosk mode page is allocated for the automation of CO2 readings for all regions, as shown in Figure 45. In the future, this would be the main page when left in idle, but is considered for future works.

Analysis

With the current state of the system and the results of hardware and software testing, most of the project specifications have been met. All the hardware for the sensor node and main hub are sufficiently powered using solar and battery. The sensors are fully operational, and their measurements can be wirelessly transmitted from the node to the main hub. The display on the main hub can be used to navigate the website through hand-based gestures. The website contains national, global, and local CO2 information plotted over daily, monthly, and yearly timelines. The automation that collects the local sensor data and uploads them to the website database is functional, as well as the automation that handles the extraction of national and global CO2 information from trusted sources.

Power Budget

The power consumption of the node and kiosk are listed in the two tables below. The listed values were attained by reviewing the datasheets for each device. Most of the values were listed as maximum possible values. To attain a more accurate value of the voltage and current required for the node and kiosk, a known DC source was used to power each device. Using both voltage and current measuring devices, a more accurate value was obtained in regards to how much power the node and kiosk are using to complete their tasks. In the tables below, the green rows are the power sources and were not used to calculate the total power consumption budget. Each table has a row showing the measured values for the kiosk and node under full load.

	Node		
Device	Voltage (V)	Current (A)	Power (W)
CO2	3.3	0.054	0.1782
Temp	3.3	0.015	0.0495
Xbee	3.3	0.215	0.7095
Seeduino	5	0.35	1.75
Battery	3.7	3	11.1
PV	5	0.5	2.5
Total	23.6	4.134	2.6872
Measured	5.499	0.084	0.461916

Table 4: Power Budget for the Sensor Node

	Kiosk		
Device	Voltage (V)	Current (A)	Power (W)
Raspberry Pi 4B	5.1	3	15.3
7" display	5	0.49	2.45
Xbee	3.3	0.215	0.7095
Battery	3.7	3	11.1
Solar Panel	7.2	1.37	9.864
Total	13.4	3.705	18.4595
Measured	5.13	1.1	5.643

Table 5: Power Budget of the Main Hub/ Kiosk

Dollar Budget

The total dollar cost shown in figure 47 financially covers the complete construction of 1 main hub and 1 sensor node.

Item	Price ($)
10Ah Battery	18.59
Charge Controller	25.59
DC/DC Converter	11.99
9W Solar Panel	79.00 (storage)
5V PV Panel	11.69
2.2Ah Battery	19.89
7 inch Display	52.99
Raspberry pi	35
CO2 Sensor	22
Temp Sensor	8.99
Seeduino	19.9
XBee Pro S1	24.95 (x2)
Total	355.44

Table 6: Dollar Budget for all components used for the Node and Main Hub

Future Work

A few key aspects of the system can be improved in future works. One of the main areas of improvement is in regards to the operation of the sensor nodes. The implementation of this project faced many time and budget constraints; only one sensor node was built for the scope of this project. This means that all wireless communication and sensor programming may only be functional for that one sensor node. The scalability of this project with its existing configurations may not even be possible. If more sensor nodes were to be added, significant parts of the microcontroller code and the flow of data across the expanded network would have to be changed.

Another aspect that can be improved also pertains to the sensor nodes. With the state of the project as it currently stands, the sensor nodes are programmed to continuously pull measurements from the attached sensors. The raspberry pi in the main hub is programmed to ignore all incoming data from the sensors except for periods when it accepts 100 measurements every 2.5 hours per day. The continuous attempts of spooling and sending data throughout a 24/7 cycle results in tremendous amounts of wasted power and measurements. Due to time constraints, a solution that alleviates this problem was not able to be implemented. In future works, a solution that can be implemented to solve this issue would be to introduce an interrupt

feature. Instead of having the seeeduino microcontroller on the sensor node send data continuously to the raspberry pi, the raspberry pi will instead send an interrupt signal to the seeeduino to request data. With interrupt functionality, the raspberry pi can then dictate when the collection of data is necessary. The seeeduino would only read and send data per the request made by the raspberry pi, which will save on overall power consumption of the sensor node.

Retrieving historical data from the website's database can be improved. Currently, the client has exactly a month to retrieve the historical data at the end of the year from the database from the command line to keep record of when the system was configured. If the code were to never refresh the historical CSV content, then if the system were to ever reboot, the file would continue to add measurements without applying a new year causing a weird plot. A solution would be to have a new script save a custom file pertaining to the end of the year which would allow the client to save endlessly without worrying at the start of a new year.

The units included in the Kiosk Mode Page and its functionality is another aspect for improvement. The average person will not understand the impact or the importance of a PPM, let alone MMTs of CO2, million metric tons of Carbon Dioxide. A group discussion led to the vital integration of CO2 awareness for everyone to comprehend which suggests the units to be converted into a material or animal with significant weight for portraying the negative results produced or measured in each of the 3 locales. Our team has a script for allowing the user to only navigate through the website only but it does not have a feature to retract to the Kiosk Mode Page in fullscreen mode when left in idle. Solving that problem requires further research on Bash code for a spring-like system for the specific image.

Conclusion

The increase in carbon dioxide levels worldwide is a cause for concern as greenhouse gasses continue to have alarming effects on the environment. There are numerous trusted sources that provide scientific observations and reports that can be freely accessed on the internet, but these sources are generally presented in a manner that is too difficult for the average person to comprehend. The system that we designed can bring awareness to the general population by providing a comprehensible and user-friendly interface that the average viewer can understand and learn the significance behind the reported numbers. This system will bring about public awareness on a global scale, and encourage people to reflect on their actions towards their environment.

Acknowledgements

We would like to acknowledge the following people for their contributions towards the completion of this project: Dr. Khoie for sponsoring this project, and Mark for his consistent help with ordering parts and troubleshooting throughout the build process of the system.

References

[1] “Climate Change: Atmospheric Carbon Dioxide | NOAA Climate.gov,” *www.climate.gov*. https://www.climate.gov/news-features/understanding-climate/climate-change-atmospheric-carbon-dioxide#:~:text=Increases%20in%20atmospheric%20carbon%20dioxide%20are%20responsible%20for

[2] Parts Per Million: Definition, Calculation & Example Video, “Parts Per Million: Definition, Calculation & Example - Video & Lesson Transcript | Study.com,” *Study.com*, 2021. https://study.com/academy/lesson/parts-per-million-definition-calculation-example.html

[3] J. McCloy, “13 Important Health & Environmental Benefits of Solar Energy,” *Green Coast*, Oct. 11, 2019. https://greencoast.org/environmental-benefits-of-solar-energy/

[4] “XBee/XBee-PRO S1 802.15.4 (Legacy) RF Modules User Guide.” Accessed: May 05, 2022. [Online]. Available: https://www.digi.com/resources/documentation/digidocs/pdfs/90000982.pdf

[5] “data-types-sizes-in-c,” *My Blog*, Apr. 22, 2016. https://www.startertutorials.com/blog/data-types-c.html/data-types-sizes-in-c (accessed May 05, 2022).

[6] ”Seeeduino Stalker V2.0 Datasheet” https://seeeddoc.github.io/Seeeduino_Stalker_v2.0/

[7] “CCS811 Ultra-Low Power Digital Gas Sensor for Monitoring Indoor Air Quality CCS811 Datasheet.” Accessed: May 06, 2022. [Online]. Available: https://www.sciosense.com/wp-content/uploads/2020/01/SC-001232-DS-2-CCS811B-Datasheet-Revision-2.pdf

[8] “DS18B20 Programmable Resolution 1-Wire Digital Thermometer Datasheet” [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

[9] ”Makerfocus Raspberry Pi Manufacturer site” https://www.makerfocus.com/products/raspberry-pi-expansion-board-ups-pack-standard-power-supply

[10] “7inch HDMI LCD (C) - Waveshare Wiki,” *www.waveshare.com*. https://www.waveshare.com/wiki/7inch_HDMI_LCD_(C) (accessed May 06, 2022).

[11] “DC-DC Power Converter Datasheet” https://www.haoyuelectronics.com/Attachment/XL6009/XL6009-DC-DC-Converter-Datasheet.pdf

[12] “DATASHEET Raspberry Pi 4 Model B Release 1,” 2019. [Online]. Available: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf

Appendix A

Code for the Automation (1), Kiosk Mode Page (2), Webscraping (3), and Website's HTML (4) & CSS (5) will be found with these links:

(1): Automation (1) **(**41pgs)

(2): Kiosk Mode Page (2) **(**6pgs)

(3): Webscraping (3) (7pgs)

(4): Website's HTML (4) (12pgs)

(5): CSS (5) (39pgs)

Appendix B

Code for finding available I2C addresses for the CCS811 CO2 sensor

```
#include <Wire.h>

void setup()
{
	Wire.begin();

	Serial.begin(9600);
	while (!Serial); // Wait for serial monitor
	Serial.println("---I2C Scanner---");
}

void loop()
{
	byte error, address;
	int nDevices;

	Serial.println("Scanning...");

	nDevices = 0;
	for(address = 1; address < 127; address++ )
	{
	Wire.beginTransmission(address);
	error = Wire.endTransmission();

	Wire.beginTransmission(address+1);

	if (error == 0 && Wire.endTransmission() != 0 ) // Special flag for SAMD
Series
	{
	Serial.print("I2C device found at address 0x");
	if (address<16)
		Serial.print("0");
	Serial.print(address,HEX);
	Serial.println("!");

	nDevices++;
	}
	else if (error==4)
	{
	Serial.print("Unknown error at address 0x");
	if (address<16)
		Serial.print("0");
	Serial.println(address,HEX);
	}
	}
	if (nDevices == 0)
	Serial.println("No I2C devices found\n");
	else
	Serial.println("done\n");

	delay(5000);           // wait 5 seconds for next scan
}
```

Appendix C

Code for reading basic CO2 sensor data and TVOCs from CS811

```
/* A new sensor requires at 48-burn in. Once burned in a sensor requires
  20 minutes of run in before readings are considered good. */
#include <Wire.h>

#include "SparkFunCCS811.h" //Click here to get the library:
http://librarymanager/All#SparkFun_CCS811

#define CCS811_ADDR 0x5B //Default I2C Address
//#define CCS811_ADDR 0x5A //Alternate I2C Address

CCS811 mySensor(CCS811_ADDR);

void setup()
{
  Serial.begin(9600); //115200);
  Serial.println("CCS811 Basic Example");

  Wire.begin(); //Inialize I2C Hardware

  if (mySensor.begin() == false)
  {
      Serial.print("CCS811 error. Please check wiring. Freezing...");
      while (1)
      ;
  }
}

void loop()
{
  //Check to see if data is ready with .dataAvailable()
  if (mySensor.dataAvailable())
  {
      //If so, have the sensor read and calculate the results.
      //Get them later
      mySensor.readAlgorithmResults();

      Serial.print("CO2[");
      //Returns calculated CO2 reading
      Serial.print(mySensor.getCO2());
      Serial.print("] tVOC[");
      //Returns calculated TVOC reading
      Serial.print(mySensor.getTVOC());
      Serial.print("] millis[");
      //Display the time since program start
      Serial.print(millis());
      Serial.print("]");
      Serial.println();
  }


  delay(10); //Don't spam the I2C bus
}
```

Appendix D

Code for reading temperature sensor data from DS18B20

```
// First we include the libraries
#include <OneWire.h>
#include <DallasTemperature.h>
// Data wire is plugged into pin 2 on the Arduino
#define ONE_WIRE_BUS 2
// Setup a oneWire instance to communicate with any OneWire devices
// (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

void setup(void)
{
 // start serial port
 Serial.begin(9600);
 Serial.println("Dallas Temperature IC Control Library Demo");
 // Start up the library
 sensors.begin();
}
void loop(void)
{
 // call sensors.requestTemperatures() to issue a global temperature
 // request to all devices on the bus
 Serial.print(" Requesting temperatures...");
 sensors.requestTemperatures(); // Send the command to get temperature
readings
 Serial.println("DONE");

 Serial.print("Temperature is: ");
 Serial.print(sensors.getTempCByIndex(0)); // Why "byIndex"?
   // You can have more than one DS18B20 on the same bus.
   // 0 refers to the first IC on the wire
   delay(1000);
}
```

Appendix E

Finalized code that reads CO2 and temperature data from sensors and sends the data as a string over serial communication

```
#include <OneWire.h>
#include <DallasTemperature.h>
#include <sSense-CCS811.h>
#include <SoftwareSerial.h>

#define RxD 0
#define TxD 1
#define ONE_WIRE_BUS 2
#define NODE_ID 1
#define DELAY_TIME NODE_ID

OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature tempSensors(&oneWire);
CCS811 ssenseCCS811;

String temp_str, co2_str, msg = "", packet = "", packLength = "";
unsigned int strLength = packet.length();

void setup() {
  Serial.begin(9600); // initialize serial baudrate to 9600

  tempSensors.begin(); // initialize DS811B20 Temperature Sensor

  if (!ssenseCCS811.begin(uint8_t(I2C_CCS811_ADDRESS),
uint8_t(CCS811_WAKE_PIN), driveMode_1sec))
  {
      DebugPort.println("Initialization failed.");
      while(1);
  }
}

void loop() {
  // wait based on NODE_ID ensure only one node will read at a time
  delay(1000 * DELAY_TIME);
  transmit();
}

void transmit() {
  //if (co2Sensors.dataAvailable())
  if (ssenseCCS811.checkDataAndUpdate())
  {
      // Write NODE ID to packet
      packet += String(NODE_ID);

      packet += "\t";

      // Read temp data and write to packet
      tempSensors.requestTemperatures();
      temp_str = String(tempSensors.getTempCByIndex(0));
      packet += temp_str;
```

```
        packet += "\t";

        // Read CO2 data and write to packet
        co2_str = String(ssenseCCS811.getCO2());
        packet += co2_str;

        packet += "\t";

        // Compute checksum and write to packet
        packLength = String(packet.length());
        strLength = packet.length() + packLength.length();
        packet += strLength;

        // Send packet
        Serial.println(packet);
        packet = "";
    }
    else {
        Serial.println("CO2 data not ready yet");
    }
}
```

Appendix F

Code for receiving sensor measurements, error checking, and storing of data locally on the Raspberry Pi 4B

```
#!/usr/bin/env python

import time
import serial
import re

ser = serial.Serial(port='/dev/ttyUSB0',
      baudrate=9600,
      parity=serial.PARITY_NONE,
      stopbits=serial.STOPBITS_ONE,
      bytesize=serial.EIGHTBITS,
      timeout=1
) # open serial port

NODE_MAX = 1 # number of nodes in network

# run indefinitely
while 1:
      # Node data request loop, take n measurements
      for i in range(50): # Node ID loop, increases with more sensor nodes
      for k in range(NODE_MAX):
            # print current NODE ID
            print("Working with [NODE " + str(k+1) + "]")
            # read data from serial
            raw_data = ser.readline().strip()
            # print data from serial
            print("Incoming Data Type: " + str(type(raw_data)))
            # print data type of serial
            print("Data Received: " + str(raw_data))
            # convert serial data to string
            data = str(raw_data,'UTF-8')
            # print new data type
            print("Converting Data to: " + str(type(data)))
            # parse string, saves elements into a list
            post = re.split(r'\t+',data)
            # print data type of list
            print("Converted Data: " + str(post))
            # check length of list, must have 4 elements in list
            if len(post) >= 4:
            # print third element, checksum value
            print("Expected Packet Length: " + post[3])
            # print length of string
            print("Actual Packet Length: " + str(len(data)))
            # packet is valid if checksum = length of string
            if len(data) == int(post[3]):
                  # print attempting to save data to file
                  print("Saving Packet to File...")
                  # grab the current time
                  t = time.localtime(time.time())
                  # check if NODE ID from list = NODE ID in loop
                  if int(post[0]) == (k+1):
```

```
                # initialize file name based on NODE ID
                f_name = "node_%d"%(k+1)
                # open file to append packet and time data
                file = open(f_name,"a+")
                file.write(data + "\t" + str(t.tm_year) + "\t" + \
                str(t.tm_mon) + "\t" + str(t.tm_mday) + "\t" + \
                str(t.tm_hour) + "\t" + str(t.tm_min) + "\t" + \
                str(t.tm_sec) + "\n")
                # close file
                file.close()
                # print time of packet saved
                print("Packet Saved on: ",time.ctime(time.time()))
        # print error, unexpected NODE ID, may cause problems with
        timing other nodes
        else:
                print("Save Unsuccessful: NODE ID Mismatch, Expected
                [NODE %d] Instead"%int(post[0]))
# print error, checksum from packet did not match length of packet
else:
        print("Checksum Error: Packet Invalid [NODE " + str(k+1) +
        "]")
# print error, incomplete packet, missing data
else:
print("Error: Incomplete Packet [NODE " + str(k+1) + "]")
# reset variables
print("\n")
raw_data = ""
data = ""
post = ""
time.sleep(5)
```

Appendix G

Bash script that downloads necessary dependencies to collect and process sensor data; also provides website credentials to access the website (Run on Raspberry Pi 4B with Sudo/root permissions)

```
#!/bin/bash

# File to add and setup all packages and dependencies on Raspberry Pi4
apt-get update -y
apt-get upgrade -y

apt install python3-venv python3-pip python3-tk -y
apt install build-essential zlib1g-dev libncurses5-dev libgdbm-dev libnss3-dev
libssl-dev libreadline-dev libffi-dev libsqlite3-dev wget libbz2-dev
wget https://www.python.org/ftp/python/3.8.0/Python-3.8.0.tgz
tar -xf Python-3.8.0.tgz
cd Python-3.8.0
./configure #--enable-optimizations #Performs dependency checks/Makes build
process slow
make -j 4
make altinstall
python3.8 --version

apt-get install python3.8-dev -y

pip install numpy scipy matplotlib
pip install pandas
pip install html2text
Pip install PySerial

python3 -m pip install --upgrade Pillow

wget https://awscli.amazonaws.com/awscli-exe-linux-aarch64.zip
unzip awscli-exe-linux-aarch64.zip -d /home/pi/aws
chmod 755 /home/pi/aws/aws/install
cd /home/pi/aws/aws
./install

#credential for 2021 kiosk team
aws --profile default configure set aws_access_key_id "AKIAR6QTEY44RXATHBJP"
aws --profile default configure set aws_secret_access_key
"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"
aws --profile default configure set region us-east-2

aws s3 ls s3://co2-monitoring
#done
echo "done"
```