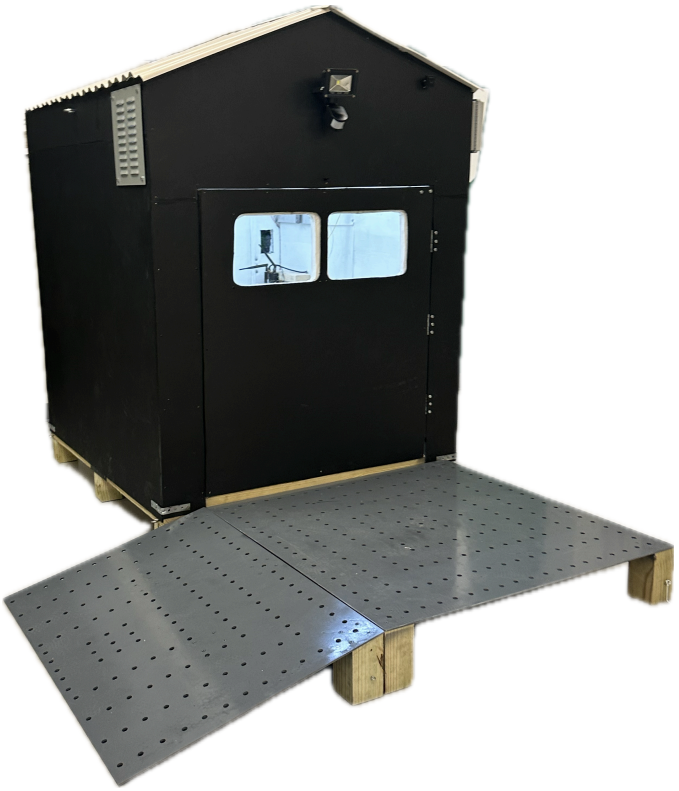
EPRI DOCK HOUSE USER AND DEVELOPER GUIDE

Design and Build of a Robot Docking House



User And Developer Manual
Release 1.0
May 2024

EPRI | ELECTRIC POWER RESEARCH INSTITUTE

TABLE OF CONTENTS

TABLE OF CONTENTS **2**
GENERAL SOFTWARE ORGANIZATION **5**
Installation 5
Quick Start Guide 5
How The Install Scripts Work 6
Git Repository Folder Organization 7
Root Directory 7
ControlCode 8
install 8
SSH 9
Touch Screen and Web HMI 10
Touch Screen Vs. Remote HMIs 10
Video Feeds Section 11
Terminal Button 11
Zoom In / Out Button 11
Table of HMI Features 12
MQTT 15
Dock House MQTT Topics 15
Flask Webserver Software Operation 19
Control Box Overview 20
Accessing The Internal Control Box 20
Control Box Internal Anatomy 21
POWER DELIVERY / E-STOP **22**
High-Level Overview 22
E-STOP Description 22
Wall Power Current Sensor 22
Power Shelf 23
AC Wall Power 24
AC Power Junction 24
Side Note on AC Fan Power 25
Battery Charger 26
The Battery Charger 27
DC Power and E-STOP 29
Steady DC Power 31
Router 33
The DC Load 34
Wall Power Script 36
Control Box Power Wiring 37
DOOR **39**
Important Notice Regarding Door Operation 39
High-Level Overview 39

Hardware Connections....40
Power to The H-Bridge / DC Motor Driver....40
Current Sensor....41
The H-Bridge / DC Motor Driver....42
The Door Motor....43
Software Operation....44
door_operation.py....44
Spot Detection....45
Spot API....45
RSSI Spot Collar....45
FAN / CLIMATE CONTROL....46
High-Level Overview....46
Hardware Connections....47
SHTC3 Indoor Temperature Sensor....47
Fan....48
Software Operation....51
climate_main.py....51
fan_operation.py....52
WEATHER STATION....53
High-Level Overview....53
Hardware Connections....54
Weather Station....54
RS-232 to USB Type B....57
Software Operation....58
Retrieving Data from The Weather Station....58
Publishing New Data....61
Cameras....62
Hardware Connections....62
Software Operation....63
LIGHTING....64
Hardware Connections....64
Outdoor Light....64
Indoor Light....65
Software Operation....66
Indoor Lighting....66
Outdoor Lighting....67
SUGGESTED UPGRADES....68
Door....68
Motor....68
Current Sensor....68
Interaction with Spot....69
Structure....69
Hand Cut Edges....69
Water Sealing....69

Cameras.....69
Website Integration.....69
Placement..... 69

GENERAL SOFTWARE ORGANIZATION

Note: All components purchased for this project and their websites including the official documentation are linked and listed in the Bill of Materials located in /documentation/Bill of Materials.xlsx of the git repository.

The general structure of the control software for this project is divided into operation Python scripts that run automatically and autonomously as .service files in a Linux environment.

Installation

This section covers how to install our software onto a Raspberry Pi controller in the control box.

Quick Start Guide

To quickly get up and running:

1. Gain Access to the Raspberry Pi via SSH or Connect a keyboard and mouse directly to the Raspberry Pi.
2. Clone our git repository somewhere on the Raspberry Pi.
3. CD into the folder the git repository was cloned.
4. Run ./install/install.sh
5. Ensure the MQTT client is running in the background and set to automatically start on startup.

Note: An explanation for step 5 will be printed to the terminal while the installAndRepairAPTInstall.sh script runs. The user must acknowledge the explanation before the script will continue.

How The Install Scripts Work

In order to speed up the time to deploy our software, a set of bash scripts in the /install/ directory were created. The table below explains what each script does.

Script File Name	**Action**
install.sh	Runs all the scripts below in order. Exits prematurely if any unexpected error occurs.
repairScripts.sh	Gives Read, Write, and Execute permissions to all .sh files in the git repository.
installAndRepairAPTInstalls.sh	Installs apps for control code to function, mainly the Mosquitto MQTT client.
installAndRepairVEnv.sh	Creates / Accesses the local Python virtual environment to install the Python modules needed for the control code to work.
installAndRepairServices.sh	Creates Linux service files to start all the Python control code scripts on startup. The Linux service files are “compiled” from template services files found in /ControlCode/service_files/. *Note: By “compiled” I mean all instances of $gitRepoDir in the service files are replaced with the action directory the git repository was cloned to. The compiled service files are stored in a “compiled” subdirectory inside the service_files subdirectory.*

If the user wants to install or repair a particular subsystem, they can bypass the install.sh script and directly run the relevant script.

Git Repository Folder Organization

The Senior Design git repository hosts, all the control code used in this project. This section will describe how the git repository is organized.

Root Directory

The table below describes what each of the directories in the root directory contains.

Directory	Description
ControlCode	This directory stores all the code that runs in the background and provides the functionality to the Dock House. More specifically all the code to control the hardware that the Raspberry Pi can directly control is in this directory.
documentation	This directory contains the important documentation for this project that may be useful to anyone who wants to further develop the dock house.
install	This directory contains a set of scripts that are used to quickly deploy the code in this repository.
Spot-API	This directory stores all the code that is used to interface with the Boston Dynamics Spot Robot's API.
SpotCollar	This directory stores all the code that is used to run an ESP32 that is directly attached to the robot that uses the dock house. This SpotCollar is an ad-hock way of getting dock house functionality to any robot, including ones that cannot communicate with the dock house in their programming.
venv	This directory contains the Python virtual environment. All Python scripts in this repository use the Python interpreter in this virtual environment. This is because all the Python modules needed are installed in this virtual environment through the install scripts.

ControlCode

The table below describes what each of the directories inside the ControlCode directory contains.

Directory	Description
bash_scripts	This directory contains all our bash scripts. The bash scripts fall into two categories. The first category is used to run Python scripts ergonomically (without the need for a rather large command line argument), some of these scripts are intended to be ran manually, and others are linked to a Linux service, and will be ran automatically on startup. The second category is all located in the *developer_service_scripts* subdirectory. They are used to quickly control all the services created by the repository. You can stop, disable, start, enable, and restart all the services associated. *Note: Enable/Disable, determines if the service will activate automatically on startup. Start/Stop, starts or stops the service and the script that is tied to it immediately.*
classes	This directory contains most of the class definitions in separate files used by the main control loop Python scripts. The main control loop Python scripts are found in the operations directory.
flask_webserver	This directory contains the Python script for the local webserver which clients (including the touchscreen interface) can connect to in order to control and view the sensor data of the dock house.
operations	This directory contains the main loop Python scripts. The start point of all the control code processes is found in each of the scripts in this directory.
service_files	This directory contains template .service files. The install scripts replace all instances of $gitRepoDir with the actual directory that the git repository was cloned to. It places these "compiled" instances of the .service file templates in a "compiled" subdirectory.

A description of what each script does can be found as a header comment in each script file at the top.

install

This subdirectory is explained in detail in the *How The Install Scripts Work* section.

SSH

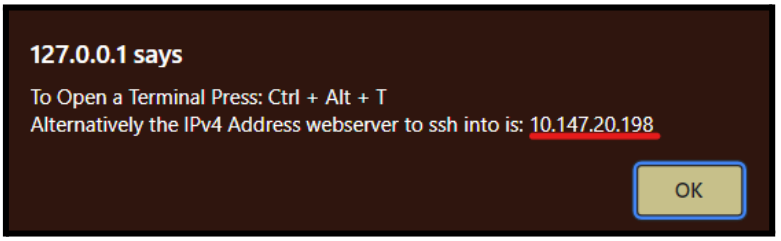
Connection Information for the microcontroller in the dock house delivered at the end of Senior Design as of Version 1.0:

Friendly Name	spotpi
User Name	eprispot
Password	314spot

Tool Recommendations: Multiple modes of connecting to the device remotely are helpful both for operation and debugging. For code updates, the VS Code Remote SSH connection is recommended. However, if the Raspberry Pi GUI is needed, using a VNC viewer (recommendation here) may be the better path.

Finding the Raspberry Pi IP Address: An IP is needed for the SSH connection and will possibly change if other devices are connected to the network first or if the router that the dock house is connected to changes.

This can be found firstly through the HMI/Physical Touchscreen. If the device is connected to WiFi a brief pop up will appear with the IP address of the device on that WiFi network. This information can also be found by touching the "Terminal" button on the HMI. This will cause a popup to appear that will provide the IPv4 Address of the webserver as seen in the picture below:



If one has access to a keyboard and mouse, one can use "Ctrl + Alt + T" to open a terminal and type "ip address" or "hostname -I". Alternatively one can access the desktop by using "alt + tab" to tab out of the kiosk, and on the top right of the screen click *WiFi* > *Connection Information*. If the device is not connected to WiFi, select a network and proceed as directed above.

Another option is using the friendly name in place of the IP address. This route is particularly stable through VNC. It is also recommended that if connection issues occur, the network of the computer you are accessing the Raspberry Pi from is connected to the same network as the Raspberry Pi.

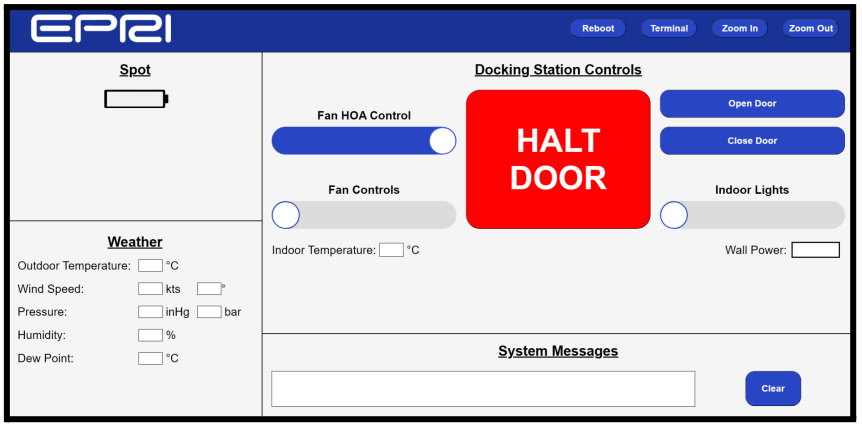
VS Code Remote Connection: It is recommended that VS Code users limit the number of extensions installed on the Raspberry Pi, as the connection and space (RAM) of the Raspberry Pi will suffer if the extensions overwhelm the memory capacity of the Raspberry Pi.

Touch Screen and Web HMI

To view sensor data and control the dock house two HMIs are available to use.

Touch Screen Vs. Remote HMIs

The first HMI is shown on the touchscreen and appears like so:



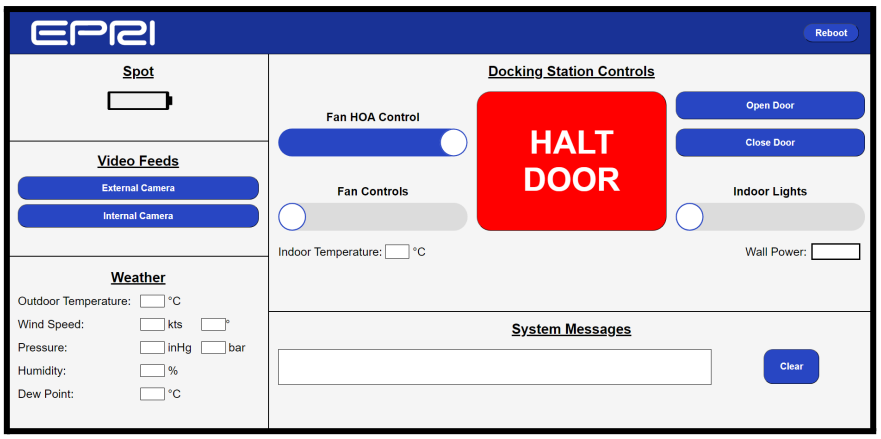
Touch Screen HMI Display

External Picture of Touch Screen HMI

The touchscreen in the pictures above is mounted in front of the control box.

The second HMI (shown below) is shown to users on the local network, by typing in the IPv4 address of the webserver into the URL of a browser.

Note: Naturally if this project is further developed, one can deploy the Flask webserver to a production server (instead of Flask's internal development server) and register the website with a proper domain name.



Remote HMI

Video Feeds Section

The first difference between the two HMIs is that the touchscreen HMI does not have access to the Video Feeds section. This is because the touchscreen HMI is a more restrictive view of the website by using Firefox's kiosk mode. Kiosk mode prevents users from going back to the previous URL and accessing the URL bar at the top.

The video feeds work by redirecting the user to a different webpage with just that camera's feed. Thus kiosk mode (touchscreen) users will not be able to navigate back to the home HMI display page. Since touchscreen users can simply look through the window to see the internal view of the dock house, the video feeds section is not necessary anyways, and is thus omitted.

Terminal Button

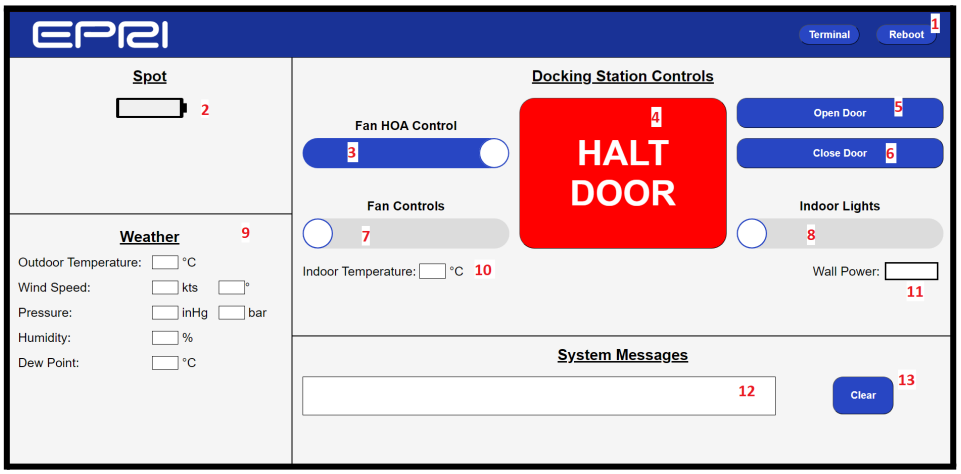
The second difference is the terminal button at the top right of the screen. This button is only present on the touchscreen HMI. The primary purpose of this button is to remind the user of two methods to gain access to a terminal on the Raspberry Pi. Firstly, the hotkey to access the terminal if they have a keyboard and mouse plugged in is "Ctrl + Alt + T". Lastly, it will provide the IPv4 Address of the webserver which should be the same one that can be used to ssh into the Raspberry Pi.

Zoom In / Out Button

Zoom in and out adjusts the root size of the website. All the other html elements scale off this root size so you can scale in and out the webpage. These buttons were made local only, because if you are accessing the website from anything other than the touchscreen, you can adjust the scale of the HMI using your browser. I don't know of a way to do the same thing with firefox in kiosk mode so I added the buttons.

Table of HMI Features

Below a picture of the touchscreen will be displayed. Each feature will be associated with a number shown in the picture. A table will follow the picture with a legend for what each number is associated with.



Number ID – Name	Description
1 – Reboot Button	This button will reboot the Raspberry Pi, by running *sudo reboot* on the Linux terminal.
2 – Spot Live Battery Graphic	This battery graphic shows Spot's current battery capacity visually and with a percentage. Spot isn't connected so it is blank in the image.
3 – Fan HOA Control Slider	This slider implements Hand Off Auto (HOA) functionality with the fan. The slider has 3 positions in the following order: Off – No Functionality Hand – Enables manual fan control, moving the Fan Controls slider (7) will affect the fan speed. Auto – The fan is controlled by the climate control code which changes the fan speed automatically based on the Indoor Temperature (10).
4 – Halt Door Button	This button stops the door in place, regardless of what it's doing.
5 – Open Door Button	This button starts opening the door. A progress bar is shown by having the button turn more and more into a darker shade of blue. 

Number ID – Name	Description
6 – Close Door Button	This button starts closing the door. A progress bar is shown by having the button turn into a darker shade of blue. The percentage of this progress bar is equal to 100 minus the percentage open for the Open Door Button (5). It should also be noted that the door's position is imprecisely calculated via a timer. Because of this, the door attempts to recalibrate itself by over-closing. The motor for the door has an internal limit switch that stops the motor from running when the door is closed past a set threshold. Using this we recalibrate the motor's current position by telling it to close for longer than it needs, to recalibrate the door's current position at 0%. All this is to say that when closing the door and viewing the close-door progress bar, an observer will note that the progress bar may have yet to be completed but still observe the door as being fully closed. This is normal and part of the door control code attempting to calibrate itself.
7 – Fan Controls Slider	This slider allows the user to manually adjust the speed of the fan. In order for this slider to work the HOA slider (3) must first be set to "hand" mode. The slider has 3 positions in the following order: Off – Turns off the fan. Slow – Sets the fan speed to slow. Fast – Sets the fan speed to fast.
8 – Indoor Lights Slider	This slider allows the user to turn on and off the internal lights. The slider only has 2 positions, "on" and "off".
9 – Weather Station Sensor Output	This section of the HMI displays sensor readings coming from the Weather Station. One thing to point out is that the wind speed section shows the direction the wind is blowing, and NOT the direction wind is hitting the weather station from. If one wishes to change this to the reciprocal, they should change it inside /ControlCode/classes/weather_class.py where the trueDirection variable is defined.
10 – Indoor Temperature Output	This displays the internal temperature of the dock house from the SHTC3 sensor. This data is used to control the fan speed if the Fan HOA control (3) is set to "auto" mode.

Number ID – Name	Description
11 – Wall Power Status	This displays 3 difference colors depending whether or not AC wall power is being supplied to the docking station. Transparent – AC Wall Power State Unknown. Red – No AC wall power being supplied (backup battery only). Green – AC wall power detected (battery is being charged).
12 – System Alert Message Box	This message box is used to display alerts to the operator as they appear. The message box does not save alerts, IE if an alert came through and the user refreshes the page, the alert will disappear. If the alerts overflow, a scrollbar will appear and when a new message comes through, the message box will automatically scroll to the latest message.
13 – Clear System Alert Messages	This button will clear the alert message box (12) of all the alerts.

MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol our scripts/programs use to communicate information between one another. When a sender wants to publish information, it specifies a topic and sends a string type message on that topic. A listener will subscribe to that topic and an interrupt can be specified for when a new message is detected.

In our project MQTT is like an API any program can interface with as long as you know the topic and correct message to send. Below we have a comprehensive list of official MQTT topics and messages used in our project.

Dock House MQTT Topics

Below is a table that lists the topic and an explanation for what information is posted on that topic.

MQTT Topic	**Descriptor/Commands Available**
alert	Any message sent here is displayed on the HMI of anyone viewing the website at that moment in time (the website doesn't save messages, see Table of HMI Features for for more information).
battery_state	Spot's current charge percentage is sent in this topic when Spot is connected.
wall_power	"Wall Power Disconnected!" and "Wall Power Reconnected!" are published on this topic if the wall power state has been changed after a set period of time. Additional "API" Messages: query_state – if "query_state" is sent in this topic, the wall power's current state ("Wall Power Disconnected!" / "Wall Power Reconnected!") will be immediately published to the topic without having to wait for the set period of time.

MQTT Topic	Descriptor/Commands Available
door	The door's current percentage open will be published to this topic (as an integer) periodically as the door opens / closes. Additional "API" Messages: open – If "open" is published on this topic, the door will start opening (until fully open). Close – If "close" is published on this topic, the door will start closing (until fully closed). stop – If "stop" is published on this topic, the door will immediately stop in place. query_state – If "query_state" is published on this topic, the door's current percent open will be immediately published to the topic without having to wait for the set period of time.
door_request	A duplicate of the 'door' topic listed above, but not as much going on. It's here to support legacy code. Future developers should refrain from using this MQTT topic.
fan_HOA	"API" Messages: ~~off – If "off" is published on this topic, the fan control code will attempt to turn off the fan. On success, "is_off" will be published to the topic as a confirmation by the control code. This setting overwrites whatever is sent in the "fan" topic~~. This functionality is not implemented and is instead just a placeholder. hand – If "hand" is published on this topic, the fan control code will attempt to set the fan to manual mode. On success, "is_hand" will be published to the topic as a confirmation by the control code. auto – If "auto" is published on this topic, the fan control code will attempt to set the fan to automatic mode. On success, "is_auto" will be published to the topic as a confirmation by the control code. query_state – If "query_state" is published on this topic, the fan control code will immediately respond with the current state ("is_off" / "is_hand" / "is_auto") on the same topic.

MQTT Topic	Descriptor/Commands Available
indoor_light	"API" Messages: power_on – If "power_on" is published on this topic, the lighting control code will attempt to turn the indoor lights on. On success, "is_on" will be published to the topic as a confirmation by the control code. power_off – If "power_off" is published on this topic, the lighting control code will attempt to turn the indoor lights off. On success, "is_off" will be published to the topic as a confirmation by the control code. toggle – If "toggle" is published on this topic, the lighting control code will attempt to toggle the indoor light's state from on to off (and vice versa). On success, "is_on" or "is_off" will be published to the topic as a confirmation by the control code. query_state – If "query_state" is published on this topic, the lighting control code will immediately respond with the current state ("is_off" / "is_on") on the same topic.
fan	*Note: the fan_HOA mode must be set to manual first for this to take effect.* "API" Messages: fast – If "fast" is published on this topic, the fan control code will attempt to set the fan speed to fast. On success, "is_fast" will be published to the topic as a confirmation by the control code. slow – If "slow" is published on this topic, the fan control code will attempt to set the fan speed to slow. On success, "is_slow" will be published to the topic as a confirmation by the control code. stop – If "stop" is published on this topic, the fan control code will attempt to stop the fan. On success, "is_off" will be published to the topic as a confirmation by the control code. query_state – If "query_state" is published on this topic, the fan control code will immediately respond with the current state ("is_off" / "is_slow" / "is_fast") on the same topic.

indoor_weather	The SHTC3 temperature sensor inside the dock house will periodically publish data in the JSON format below:
	{
	"temperature": "value",
	"relative_humidity": "value"
	}

MQTT Topic	Descriptor/Commands Available

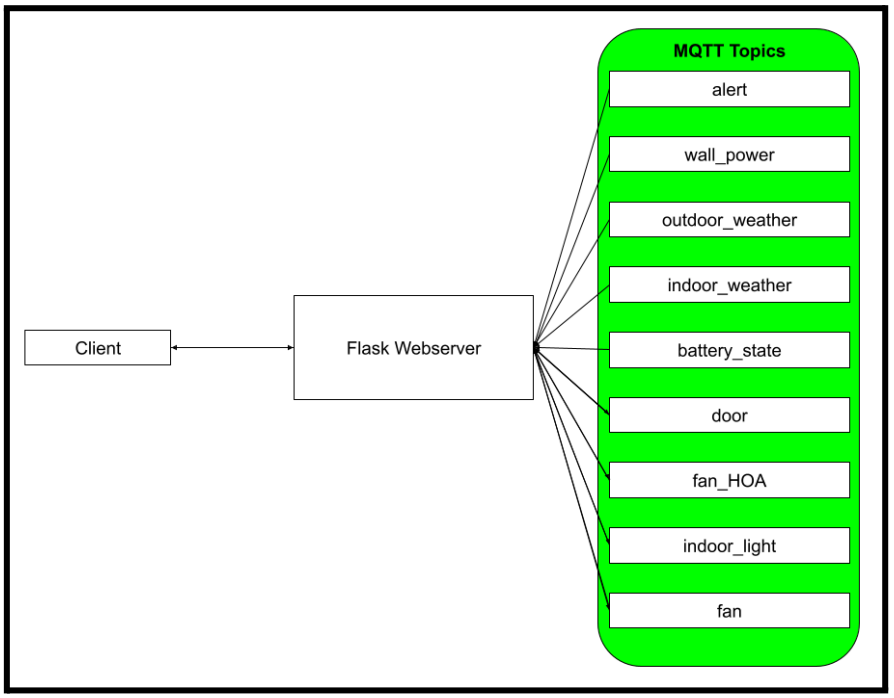
outdoor_weather	The 150WX WeatherStation will periodically publish data in the JSON format below:
	{
	"wind": {
	"speed": "value",
	"rawDirection": "value",
	"trueDirection":"value",
	"status": "value"
	},
	"heading": "value",
	"meteorological": {
	"pressureMercury": "value",
	"pressureBars": "value",
	"temperature": "value",
	"humidity": "value",
	"dewPoint": "value"
	}
	}

rssi	Sent from the Spot Collar ESP32, a negative integer representing wifi rssi mode

Flask Webserver Software Operation

Please note that this is just to give the reader a general idea of how the Python script comes together to create the webserver. It is NOT supposed to explain exactly how these subprocesses work and exactly what they are inside the script.

The flask webserver works by retrieving data from the MQTT topics listed in the diagram below and displaying their information to the client on the HMI. The flask webserver also posts data to the MQTT topics with arrows pointing towards them in the diagram below (‘door’, ‘fan_HOA’, ‘indoor_light’, and ‘fan’) based on client actions:



The flask webserver runs on multiple threads. A list of separate threads or processes is categorized into the table below with a description of each thread or process. In order for data to be shared across these threads a ‘data_handler’ object that has thread locking logic. This ‘data_handler’ object’s first-level properties (nested properties not included) are each linked to a unique MQTT topic.

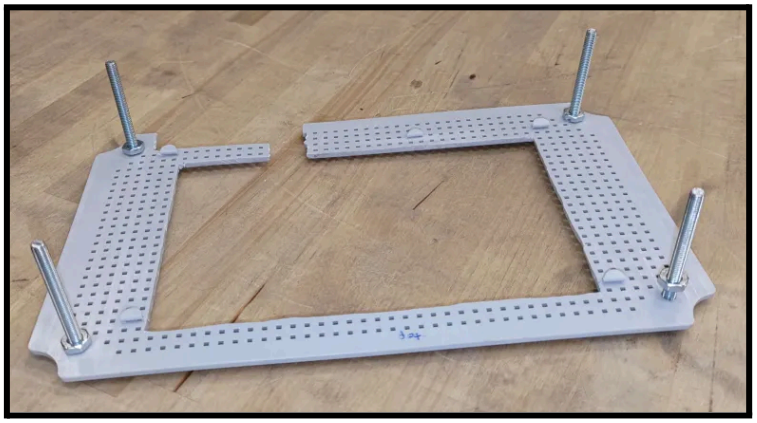
Thread or Process	**Description**
MQTT on_message	This thread constantly monitors the MQTT topics listed in the diagram above. When it detects a new message on a valid topic, the script classifies and appends the classified data into the appropriate property of the ‘data_handler’ object.
WebSocket Communication	This thread checks that the ‘data_handler’ has received initial data for each of its first-level properties. If it hasn’t, it requests data by posting ‘query_state’ on the relevant topic. It also pushes updated data from the webserver to connected clients. Lastly, it also sends the IPv4 address of the webserver to the client on request.
Camera Feed	This thread streams frames from the indoor and outdoor cameras to the client when the client opens the relevant feed.

Client HTML POSTs	This process is called when the user presses a button or moves a slider on the HMI that requests the control code to do something. It works by classifying an HTML request into an appropriate MQTT message to post on the appropriate topic.

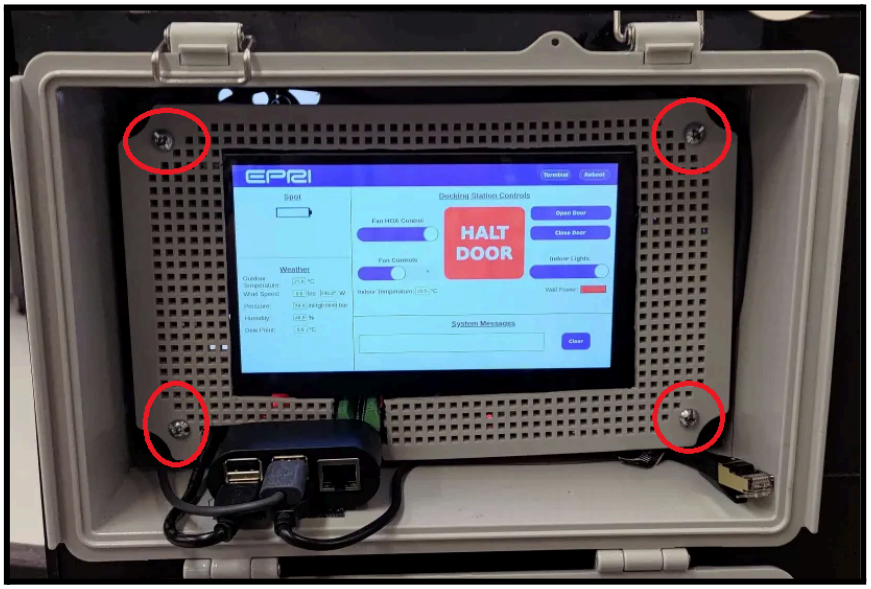
Control Box Overview

Accessing The Internal Control Box

Please note that the touchscreen display is not held in place by anything more than friction with the frame. If a user presses too hard on the touchscreen, the screen may dislodge itself from the frame. Below is a picture of the frame that the touchscreen is held in place by:



To access the internal control box first remove the 4 screws circled in red:

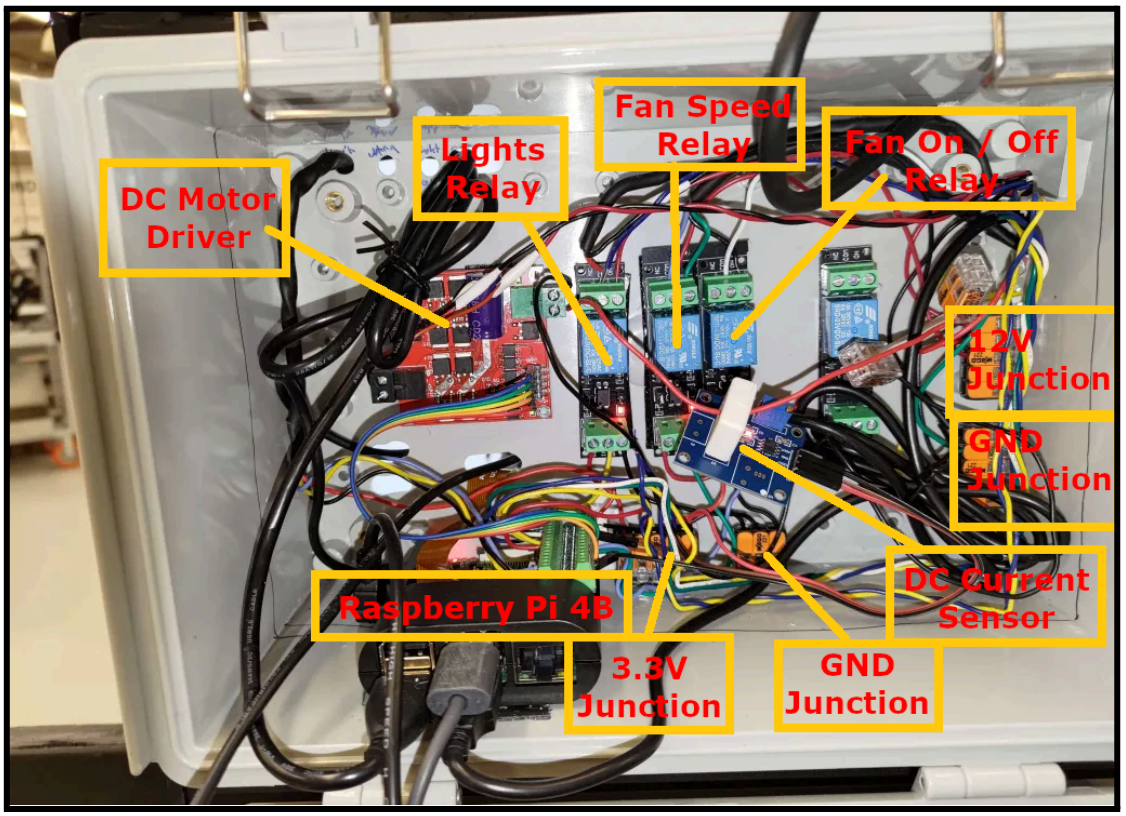


Then carefully pull and remove the frame and touchscreen. Note that the touchscreen is still connected to the Raspberry Pi via relatively short cables. The touchscreen can be directly separated from the frame it is held in place by.

To reinstall the touchscreen and frame it is recommended to pull the touchscreen through the central hole in the frame so that the cables pass through it. Then screw in the frame. After the frame is screwed in then affix the touch screen to the frame.

Control Box Internal Anatomy

Below is a picture to give the reader a quick glance at what everything in the control box is. This should hopefully allow the reader to quickly find the relevant section for that component:

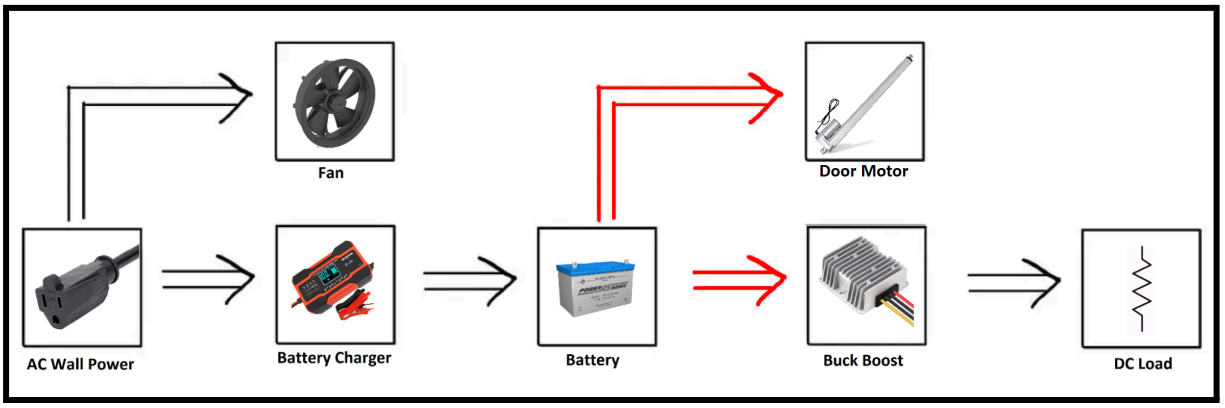


POWER DELIVERY / E-STOP

This section explains how power is distributed to where it needs to be in the dock house.

High-Level Overview

This section provides a high-level overview of how power is distributed across the dock house. Below you can see a picture that shows the order in which power flows through our dock house.



In short, wall power is used to directly power a battery charger that tries to constantly keep a battery at 100% charge. The battery then goes to a junction where the battery can directly power the door motor, and feed into a buck-boost converter which attempts to maintain a steady 12-volt output regardless of the battery's current charge (Sealed Lead Acid batteries have an output voltage that changes depending on the battery's current charge). The buck-boost is then used to power the 12-volt DC load.

E-STOP Description

The diagram above does not directly illustrate the presence of the E-STOP. Rather the E-STOP is shown as the red arrows in the picture above, indicating that the E-STOP will restrict current flow from the battery to the door motor and buck boost converter.

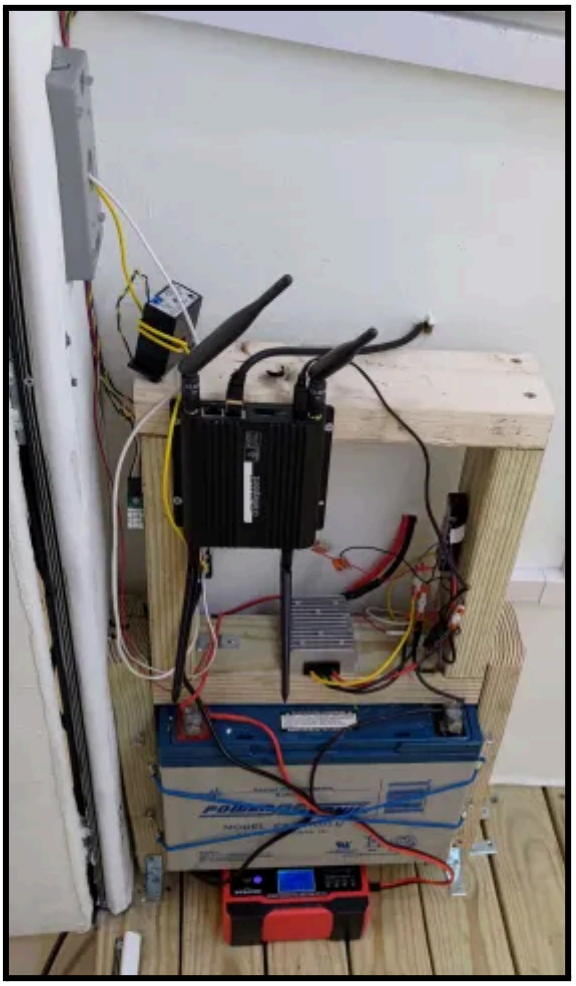
The E-STOP disconnects the battery from that junction that powers the DC motor and buck-boost converter. When the E-STOP is pressed, the DC load will not receive any power and the AC load will also be disconnected by proxy. This is because the microcontroller is connected to the DC load and turning it off will turn off the control signals to connect the fan to the AC wall power via relay.

Wall Power Current Sensor

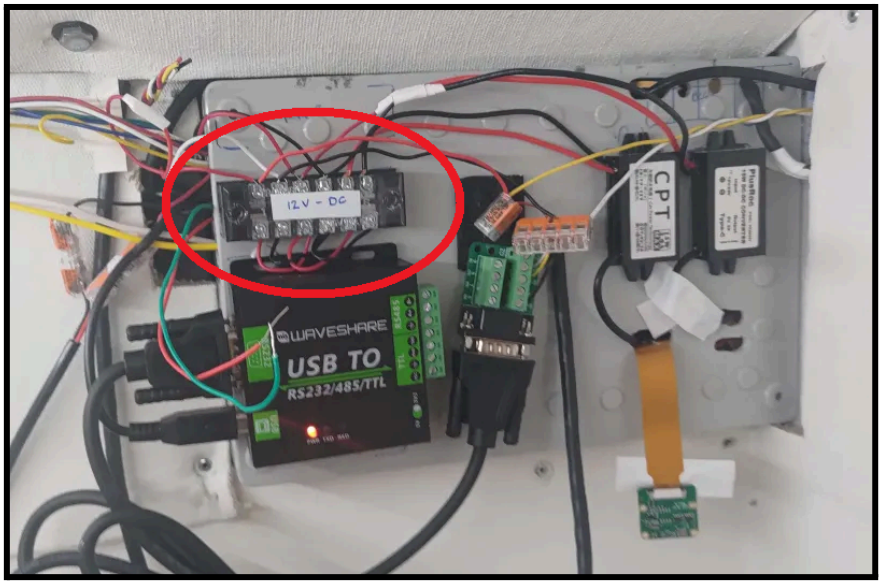
Another thing not pictured in the image above is the presence of a AC current sensor. This current sensor is a relay that closes when AC current is detected. This sends a control signal to the microcontroller to let it know that the dock house is currently only receiving power via the battery. In such a condition an alert is sent on the alert and wall_power MQTT topics with the message "Wall Power Disconnected!". When power is re-established an alert is once again sent on the alert and wall_power MQTT topics with the message "Wall Power Reconnected!".

Power Shelf

Most of the power delivery is handled on a “power shelf” pictured below:



Most of the DC load is connected to the junction circled in red behind the control box in the picture below:

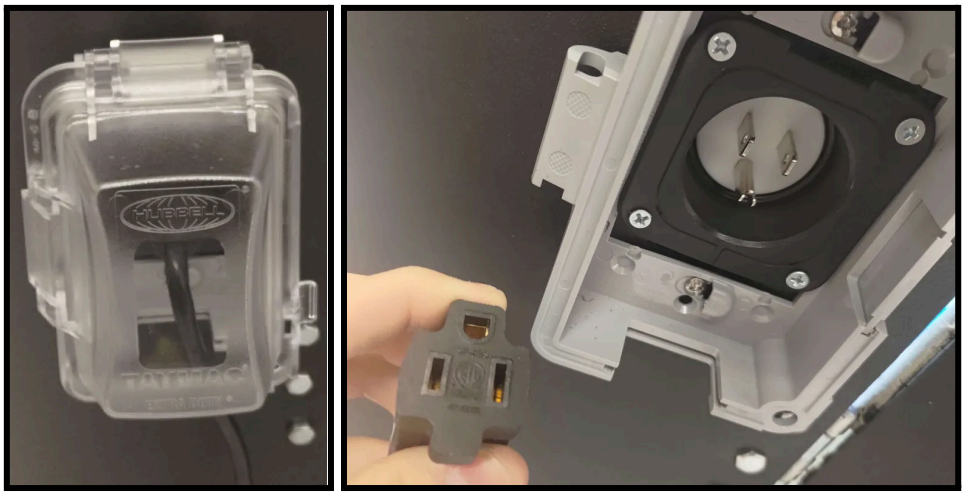


AC Wall Power

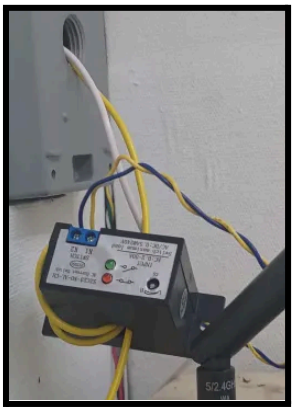
This section covers how 120 VAC wall power is delivered to the components that need it.

AC Power Junction

AC Wall power is first supplied to the dock house via a wall socket mounted outside the dock house pictured below:



The AC wall power then passes through the AC current sensor pictured below, which is powered by the mutual inductance of the AC current:



The AC current switch is tuned to close a relay as soon as AC current is detected. This is how the Raspberry Pi is able to determine if the dock house is running on the backup battery power or the AC wall power. See the Wall Power Script section for further details on the software side of things.

The AC wall power is then sent to the junction screwed on the left side of the “power shelf”. The junction is further described in the image and table below:

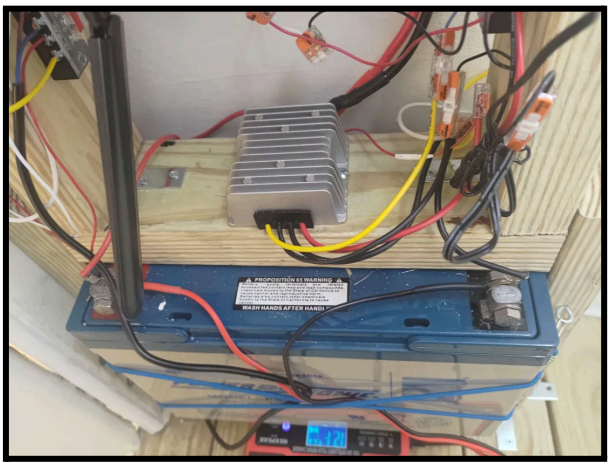
Cable	Description
1 – White	Neutral wire from the wall socket.
2 – White	AC Fan - input power. Wire routes to a relay inside the control box first.
3 – Brown	Battery Charger - input power.
4 – Blue	Battery Charger + input power.
5 – Red	AC Fan + input power. Wire routes to a relay inside the control box first.
6 – Yellow	Hot wire from the wall socket.

Side Note on AC Fan Power

Although the fan is AC-powered, a relay that is controlled by the microcontroller determines if the fan will receive power or not in order to turn it on / off. There is a 3rd wire for the fan not pictured here that when shorted with either the white (2) or red (5) wire, will set the fan speed to slow. This is also controlled by the microcontroller via relay. Both of these relays are inside the control box. See the Fan / Climate Control section for further details.

Battery Charger

The brown (3) and blue (4) wires from the table above are connected to the battery charger at the bottom of the picture below:



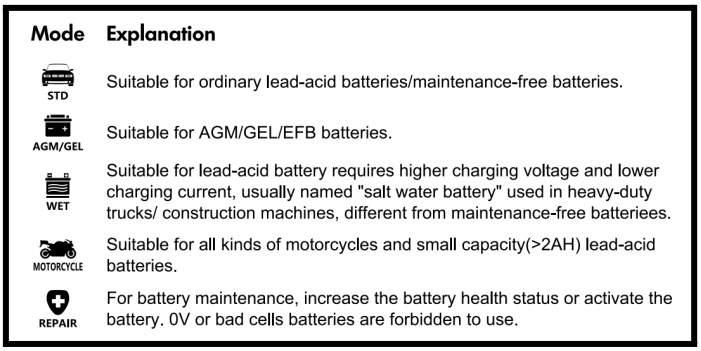
The Battery Charger

The battery charger is an automatic battery charger that is capable of sending 10 amps at 12 volts to charge the battery. This charger was selected as it outputs enough amperage to power the load even with a fully depleted battery.

There is one quirk with this charger. Although it is autonomous, if the button gets accidentally pressed the charger will exit “standard mode”, which is the only mode we want the charger to be in. The charger’s currently selected mode is shown by an LED at the bottom of the charger shown below:

In the upper image, you can see the selected mode is “AGM/GEL”. This is an incorrect mode. By pressing the mode selection button, the modes can be cycled until “STD” mode is selected. The charger should remain in this mode, even if a power outage occurs.

The charger manufacturer has the following descriptions of each mode in the image below:

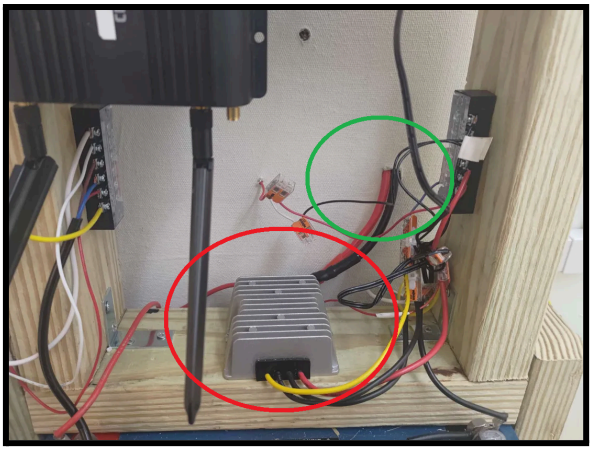


Mode	Explanation
STD	Suitable for ordinary lead-acid batteries/maintenance-free batteries.
AGM/GEL	Suitable for AGM/GEL/EFB batteries.
WET	Suitable for lead-acid battery requires higher charging voltage and lower charging current, usually named "salt water battery" used in heavy-duty trucks/ construction machines, different from maintenance-free batteriees.
MOTORCYCLE	Suitable for all kinds of motorcycles and small capacity(>2AH) lead-acid batteries.
REPAIR	For battery maintenance, increase the battery health status or activate the battery. 0V or bad cells batteries are forbidden to use.

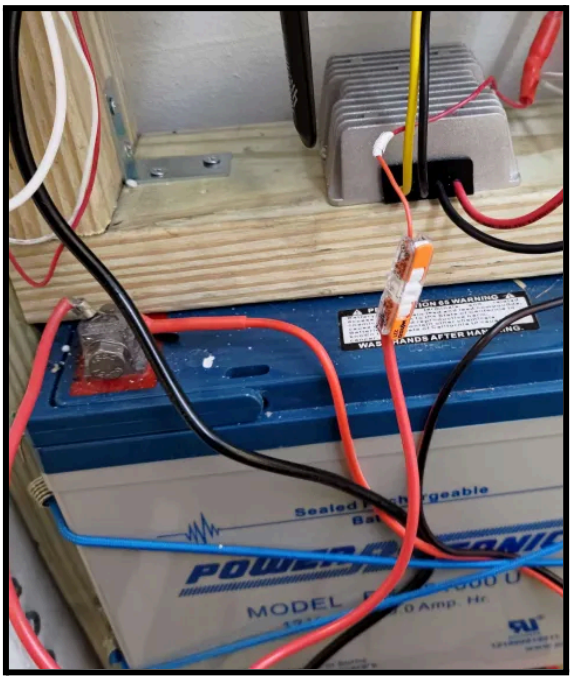
DC Power and E-STOP

The battery charger outputs a DC voltage which is connected in parallel with the battery to keep it charged while the wall power is plugged in. From there the battery is connected to a junction that feeds power to both the DC motor that operates the door as well as the buck-boost converter.

The battery is not, however, directly connected to that junction. Instead, the E-STOP is connected in series with the battery's positive terminal and that junction. Below you can see a picture with the buck-boost converter circled in red, and the cables for the E-STOP circled in green coming out of the wall:

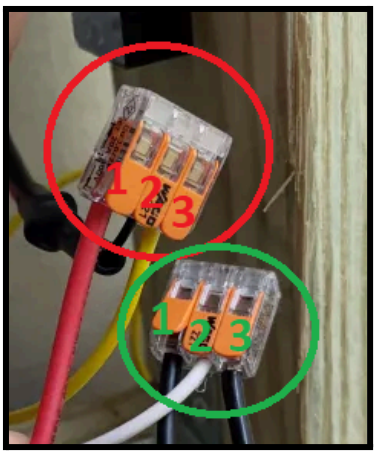


Below you can see the positive terminal of the battery feeding into the red wire of the E-STOP:



The E-STOP button itself is pictured below:

The “output” of the E-STOP is the black wire that then connects to the junction shown below:



The junction in question is the 2 WAGO three port connectors shown above. The WAGO in the red circle is the junction for the positive 12-volt DC input. The green circle is the WAGO for the ground connection of the DC input.

The table below describes what each cable connected to the red-circled WAGO junction does:

Cable	**Description**
1 – Red	Buck-Boost Converter + Input
2 – Black	Output of the E-STOP. This input gets disconnected from the battery when the E-STOP is pressed, disconnecting the battery’s positive terminal from the battery.
3 – Yellow	Connects to the + terminal of the DC motor driver (effectively an H-Bridge to control the direction of current flow through the motor). See the door section for more information.

The table below describes what each cable connected to the green-circled WAGO junction does:

Cable	Description
1 – Black	Buck-Boost Converter - Input
2 – White	Connects to the - terminal of the DC motor driver (effectively an H-Bridge to control the direction of current flow through the motor). See the door section for more information.
3 – Black	Direct connection to - battery terminal

WARNING: Pay special attention to the DC polarity of the buck-boost, mixing it up WILL destroy the buck-boost.

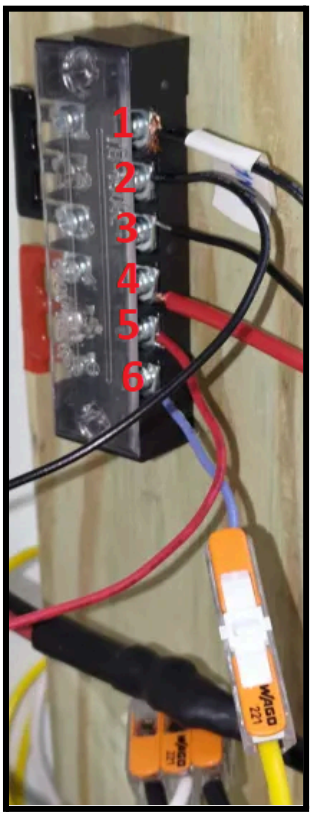
Steady DC Power

The purpose of the buck-boost converter is to take an input voltage within the range of 8 to 40 volts and stabilize the output to a steady 12 volts. Its purpose in our project is to keep the battery output stable as Sealed Lead Acid batteries have an output voltage that fluctuates above and below their rated output depending on their current charge.

The buck-boost converter pictured below has a red and black input wire circled in green below, and a yellow and black output circled in red below:

The red and black input cables are connected to the junction mentioned in the previous section, and the output of the buck-boost converter connects to the junction screwed onto the right side of the “power shelf”.

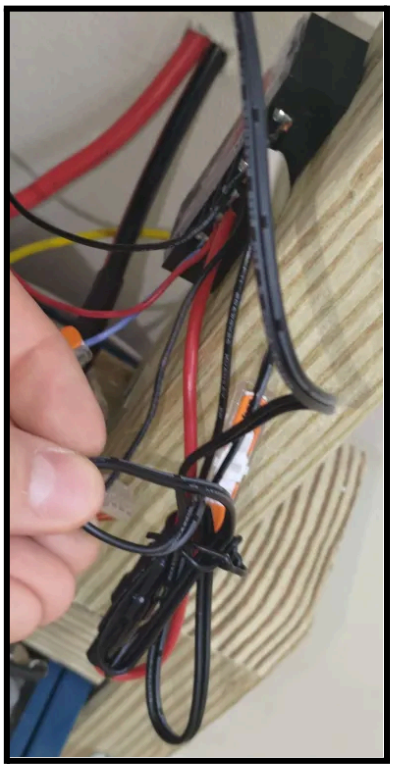
The junction screwed onto the right side of the “power shelf” is described by the image and table below:

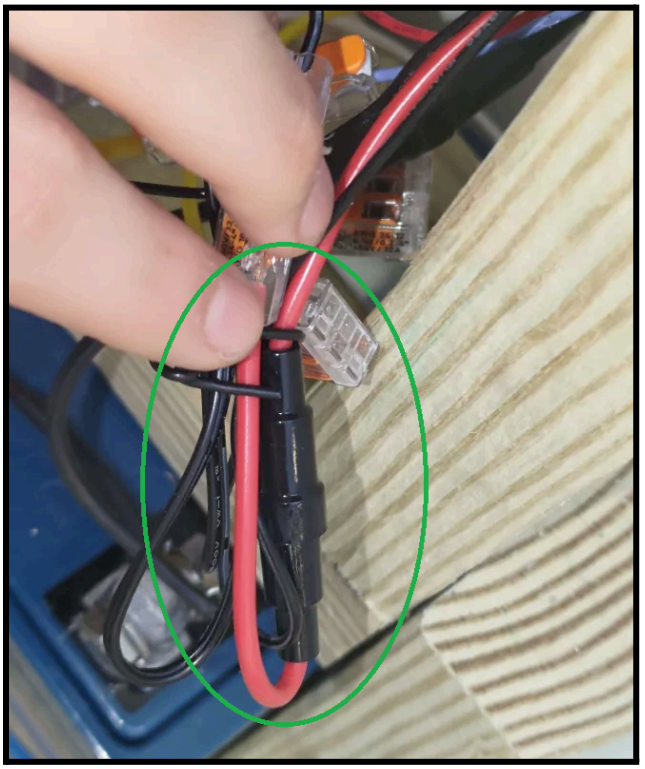


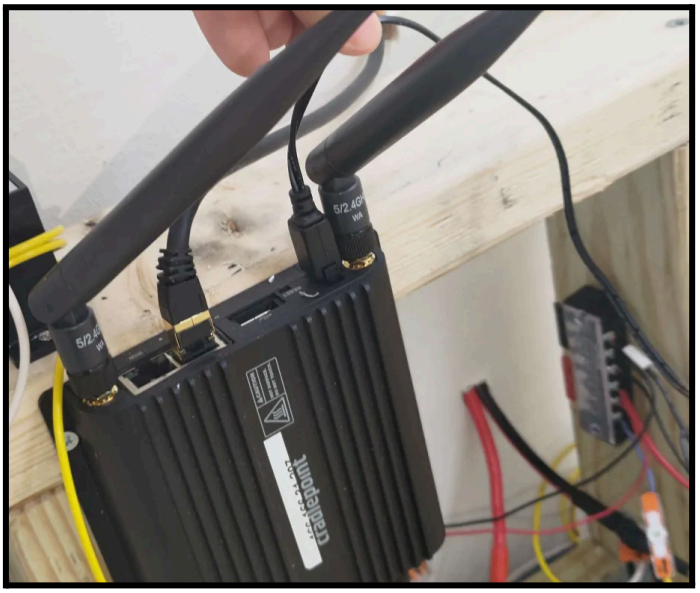
Cable	Description
1 – Black	Connects ground to the router.
2 – Black	Connects ground to the DC load by the control box.
3 – Black	Connects to the negative terminal of the output of the buck-boost converter.
4 – Red	Connects 12 volts to the router with a 3 amp fuse inline.
5 – Red	Connects 12 volts to the DC load by the control box.
6 – Blue	Connects to the 12-volt output of the buck-boost converter.

Router

The router is connected like so with the 3 Amp fuse highlighted by a green circle:

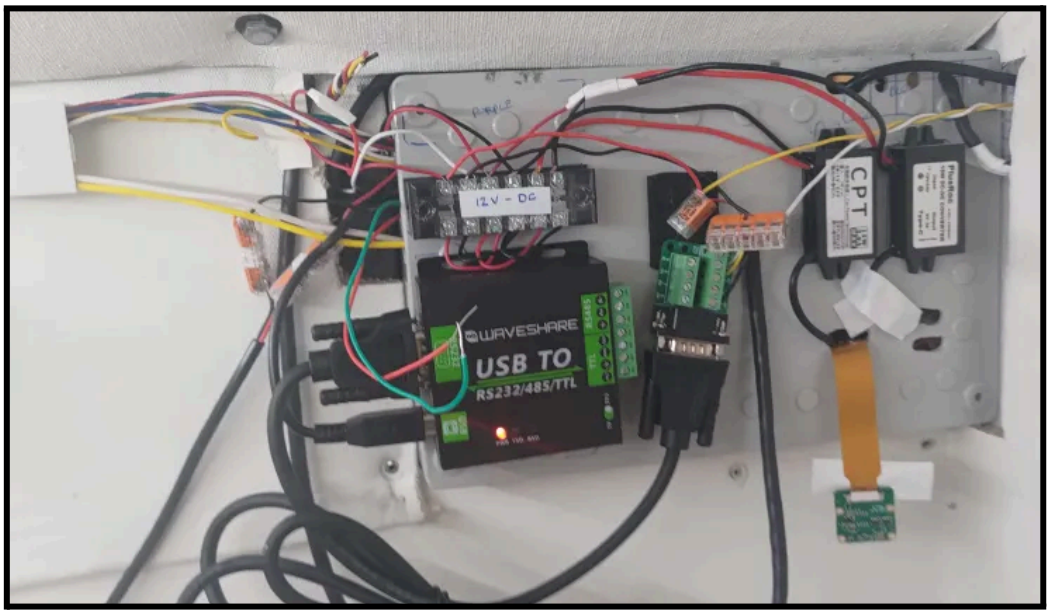




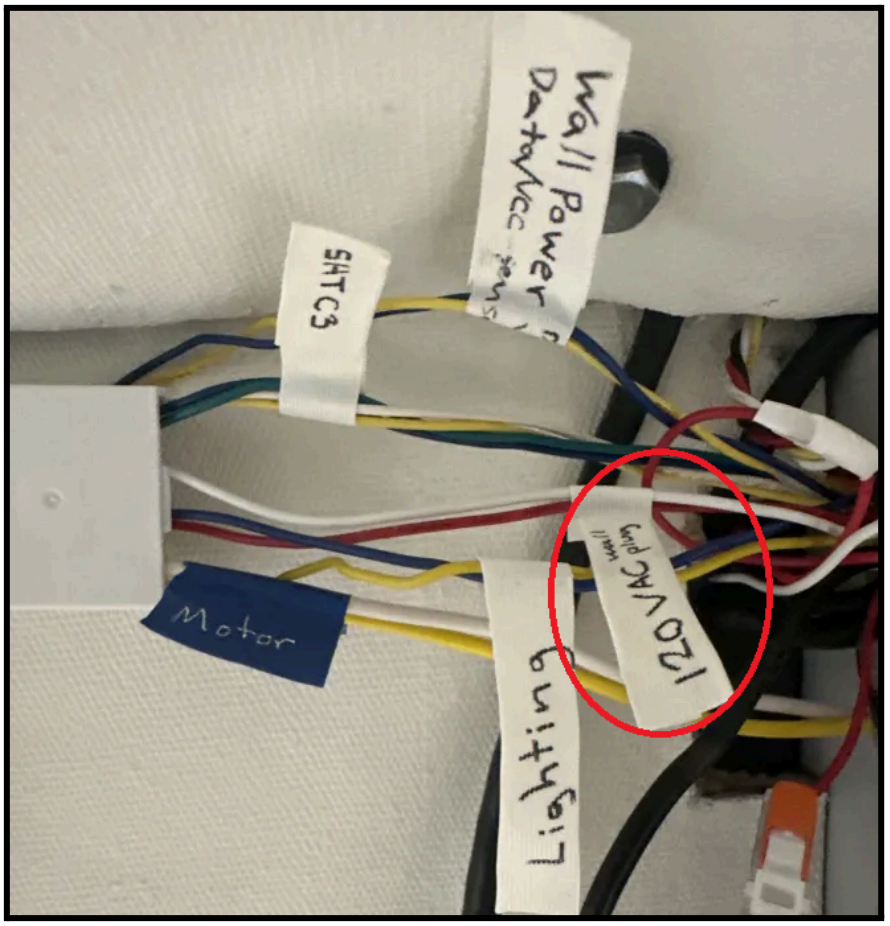


The DC Load

The 12-volt output from the Steady DC Power section is then sent to this last junction pictured below, where the 12 volts are distributed to the rest of the load at the junction labeled “12V - DC”:



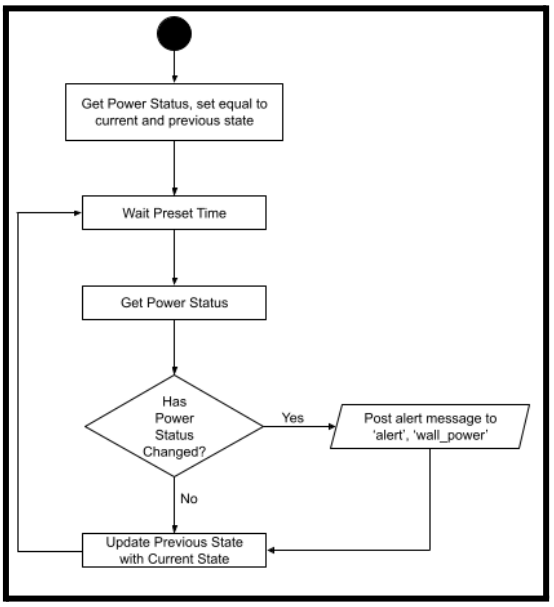
The image below shows which cables carry the input 12 volts to the junction pictured above:



From the above picture, the INCORRECTLY LABELED “120 VAC wall plug” wires actually connect the 12 volts to the aforementioned junction. “Wall Power” carries the control signal from the relay discussed in the AC Wall Power and Wall Power Script sections. The wires labeled “Motor” carry the 12 volts used to power the DC motor previously mentioned in the DC Power and E-STOP section as well as later on in the door section.

Wall Power Script

The wall power script is one of the most basic scripts in the dock house. It works using state machine logic. Below is a software flowchart describing the logic for the wall power script:



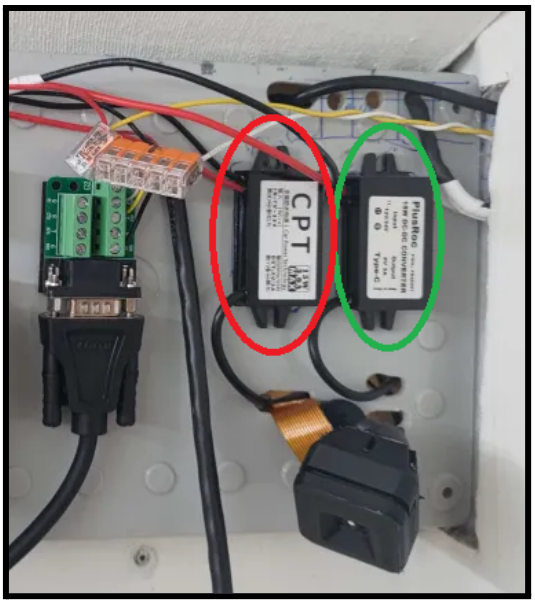
On start, it gets the current power status from GPIO 22. If GPIO 22 is high, then wall power is being supplied. If GPIO 22 is low, then no wall power is being supplied. The current power status is saved to two variables, ‘current_state’ and ‘previous_state’. The script then waits a preset time defined in the variable ‘minutes_to_wait`.

When the script awakens, it gets the power status and assigns it to ‘current_state’. It then checks if the current and previous states are the same. If they are different, the program will post the message "Wall Power Disconnected!" / "Wall Power Reconnected!" appropriately based on what the current state is. The messages are posted to the MQTT topics ‘wall_power’ and ‘alert’. The alert topic will repeat the message to the ‘System Alert Message Box’ at the bottom of the HMI, and update the ‘Wall Power Status’ (to read more about these two display components on the HMI see the Table of HMI Features section).

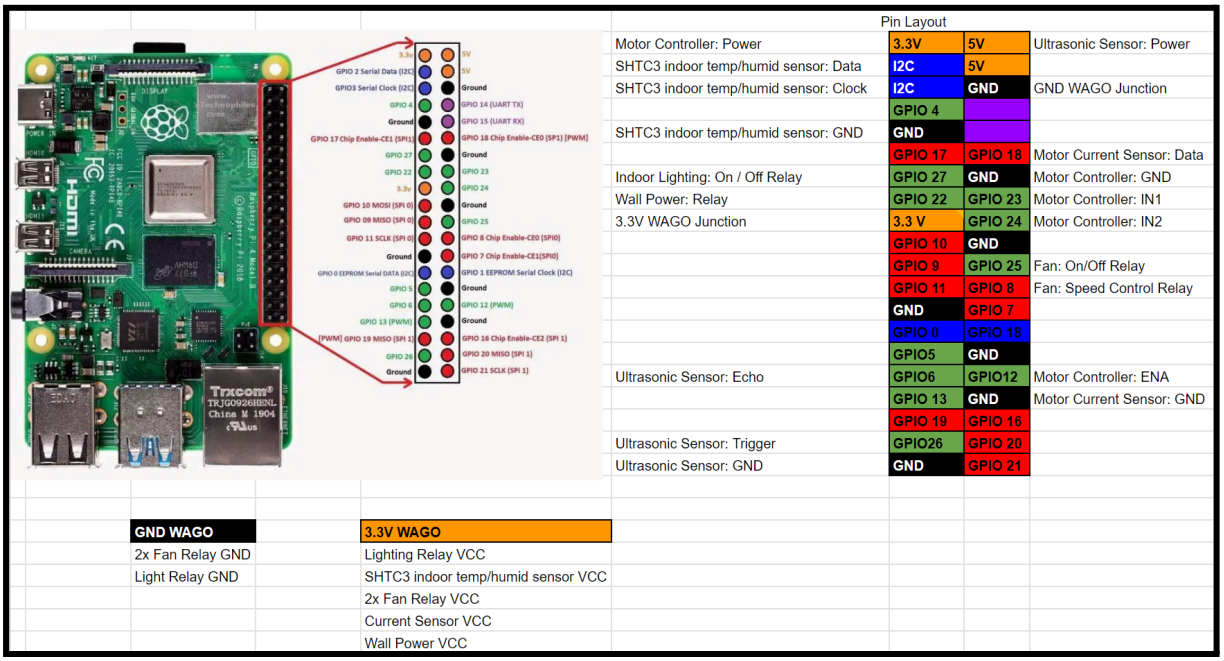
Lastly the script updates the previous state with the current state and then loops back to waiting the preset time.

Control Box Power Wiring

The Raspberry Pi and touchscreen are powered by a 12 to 5 volt USB C and USB micro connector respectively. These are found on the back of the control box pictured below. The one on the right (green circle) powers the Raspberry Pi 4B using USB C, and the one on the left (red circle) powers the touchscreen via USB micro.



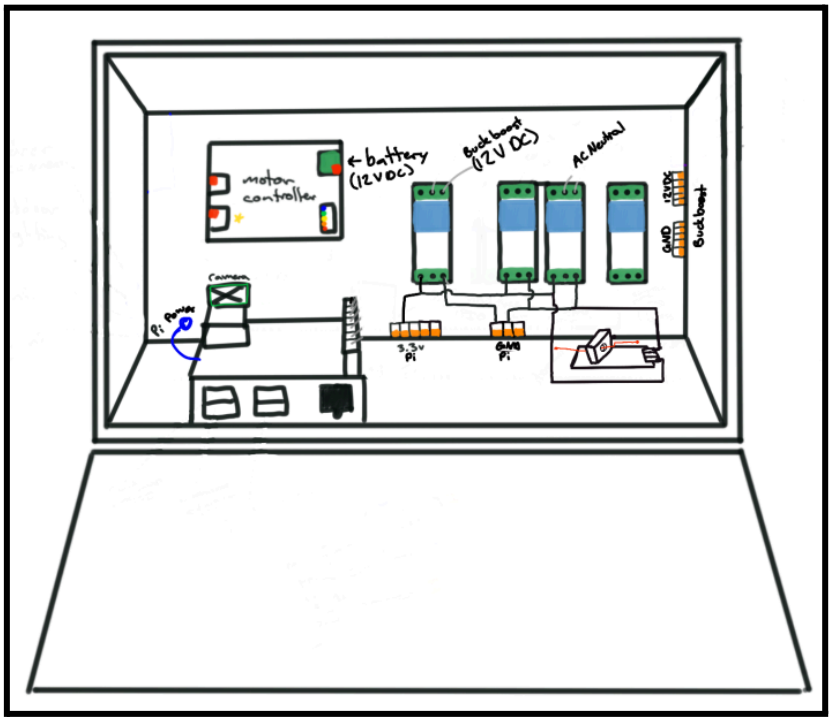
Looking at the pinout diagram below you can see that there are 2 pins on the top right that can deliver 5 volts and a pin on the top left and middle left can deliver 3.3 volts. Additionally, throughout the GPIO ports, there are pins dedicated to ground:



	Pin Layout		
Motor Controller: Power	3.3V	5V	Ultrasonic Sensor: Power
SHTC3 indoor temp/humid sensor: Data	I2C	5V	
SHTC3 indoor temp/humid sensor: Clock	I2C	GND	GND WAGO Junction
	GPIO 4		
SHTC3 indoor temp/humid sensor: GND	GND		
	GPIO 17	GPIO 18	Motor Current Sensor: Data
Indoor Lighting: On / Off Relay	GPIO 27	GND	Motor Controller: GND
Wall Power: Relay	GPIO 22	GPIO 23	Motor Controller: IN1
3.3V WAGO Junction	3.3 V	GPIO 24	Motor Controller: IN2
	GPIO 10	GND	
	GPIO 9	GPIO 25	Fan: On/Off Relay
	GPIO 11	GPIO 8	Fan: Speed Control Relay
	GND	GPIO 7	
	GPIO 0	GPIO 18	
	GPIO5	GND	
Ultrasonic Sensor: Echo	GPIO6	GPIO12	Motor Controller: ENA
	GPIO 13	GND	Motor Current Sensor: GND
	GPIO 19	GPIO 16	
Ultrasonic Sensor: Trigger	GPIO26	GPIO 20	
Ultrasonic Sensor: GND	GND	GPIO 21	

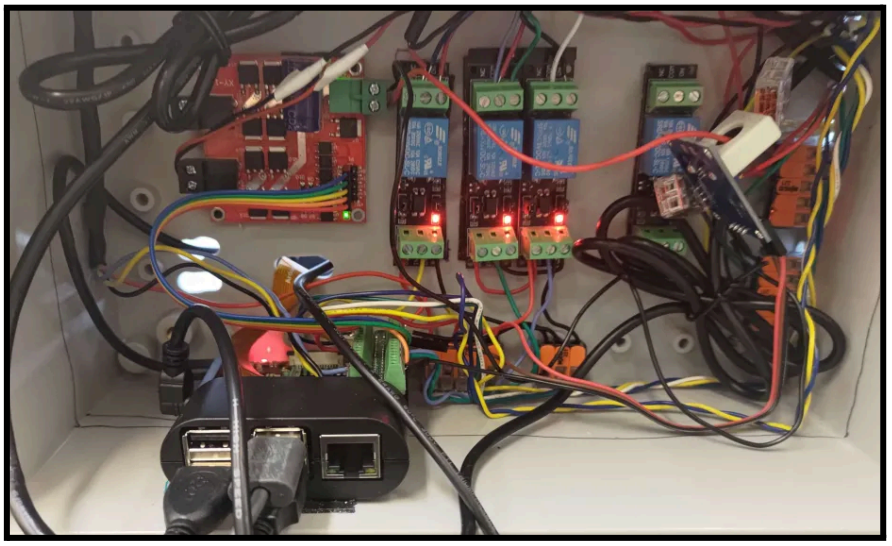
GND WAGO	3.3V WAGO
2x Fan Relay GND	Lighting Relay VCC
Light Relay GND	SHTC3 indoor temp/humid sensor VCC
	2x Fan Relay VCC
	Current Sensor VCC
	Wall Power VCC

Some of these pins are connected to WAGO junctions to distribute power across the inside of the control box. The WAGOs are found according to the diagram below:



At the bottom of the control box you will note the “3.3V Pi” which is a WAGO junction that connects the 3.3 volts from the Raspberry Pi to a bunch of components that need it inside the control box. The “GND Pi” also at the bottom similarly connects ground to many of the same components. On the right side there is the “buck-boost GND” and “buck-boost 12 VDC” that brings in 12 volts from the DC Load section.

For reference below is an image of what the inside of the control box actually looks like:



DOOR

Important Notice Regarding Door Operation

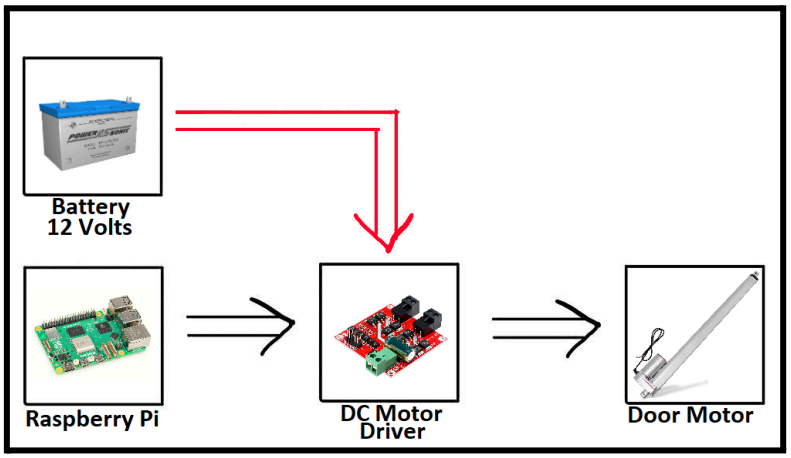
The sequence of door operation follows two modes: autonomous and manual operation. It is recommended that during automatic operation the robot is still monitored and there is a person on standby to respond to alerts especially pertaining to door collisions.

Warning: Successive door collisions that are stopped by current sensor will throw the door motor calibration off. It is recommended that the door is recalibrated after each collision by closing the door first.

In a worst-case scenario, the motor might attempt to open the door further than what the hinges will allow. In this case, the current sensor should turn off the motor. But this will wear the motor out.

High-Level Overview

This section provides a high-level overview of how the door operation works in the dock house. Below you can see a picture that shows the order in which power and control signals flow through our dock house.



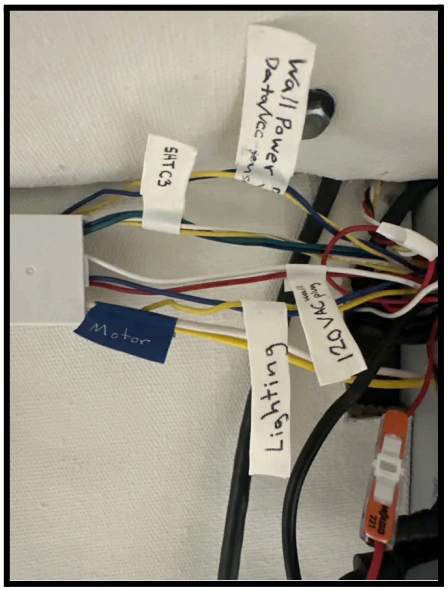
Power is supplied to the door motor from a 12-volt Sealed Lead Acid battery. The power passes through a DC Motor driver that controls the direction of current flow across the door motor. The red arrow indicates that a current sensor is connected in-between the battery and DC motor driver and is used for collision detection. A Raspberry Pi sends control signals and a PWM signal to control when to send current to the door motor and in which direction to do so.

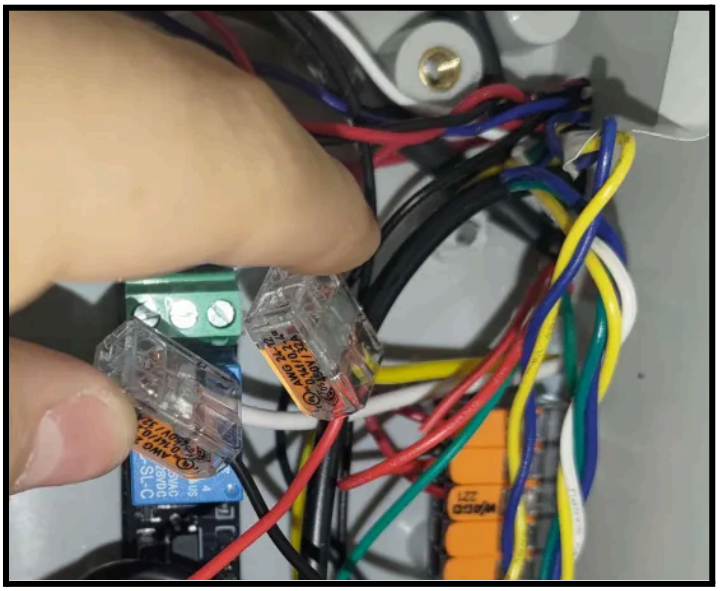
Hardware Connections

This section describes all the wired connections that the door motor uses to operate.

Power to The H-Bridge / DC Motor Driver

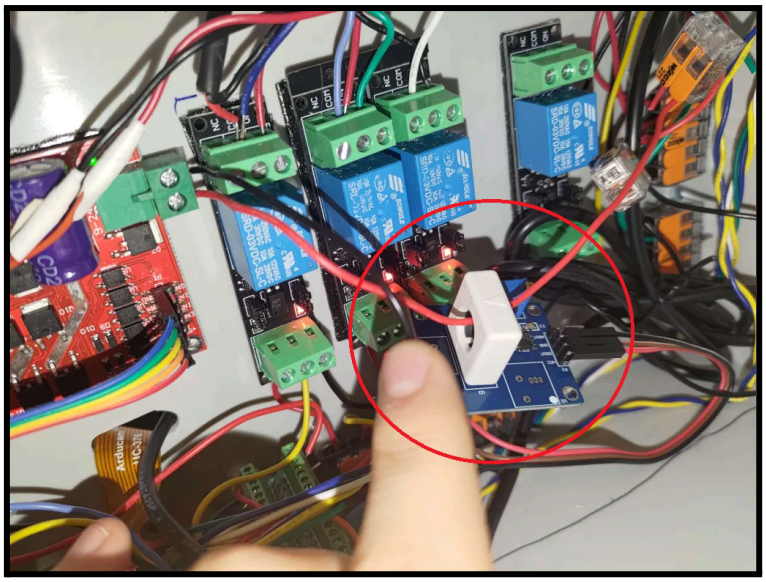
The door motor is powered by 12 volts coming in directly from the WAGO terminals connected to the battery from the DC Power and E-STOP section. These cables route directly into the control box in the images below:





Current Sensor

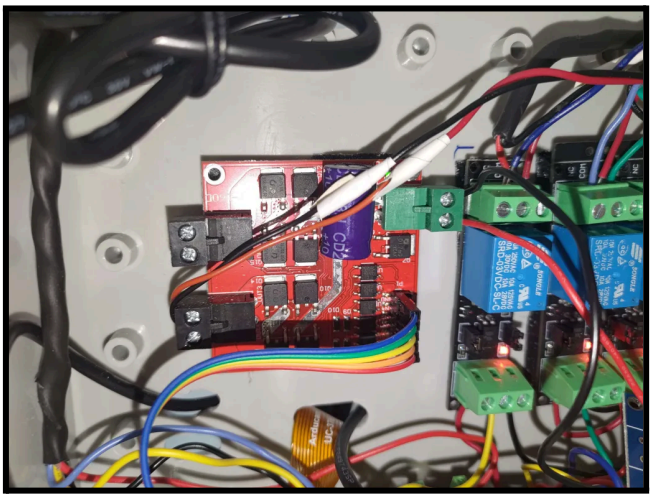
In order to detect collisions, a DC current sensor is connected to the positive 12-volt input to the DC motor driver. When the door motor encounters a block, the motor will draw more current from the battery in order to gain more torque to overcome the obstacle. This spike in current is detected by this current sensor and sent as a control signal to the microcontroller to handle the issue appropriately. Below you can see an image of the current sensor circled in red:



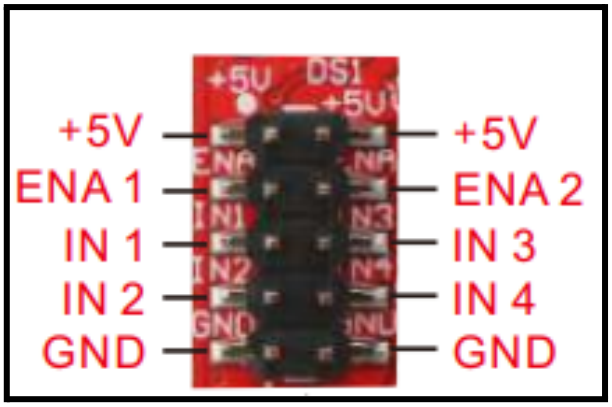
It must be noted that the Raspberry Pi 4B does not have an ADC. To work around this issue, a current sensor with digital output that will output a high logic value when the current exceeds a set threshold was selected. Turning the potentiometer on the current sensor clockwise increases the sensitivity (lowers the current threshold for the digital output pin to go high) and vice versa.

The H-Bridge / DC Motor Driver

The door motor works simply by retracting when current flows through the motor in one direction, and extending when current flows through the motor in the opposite direction. To control the direction of current flow a DC Motor Driver which is effectively an H-Bridge (shown below) is used.



The green terminal to the top right is where DC power is input, and the black terminal to the bottom left is where 12 volts is output to control the door motor. The ribbon cables connected to the bottom right terminal are used to control the DC Motor driver via control signals from the Raspberry Pi. Below is a pinout for the ribbon cable terminal in the bottom right, pulled from the manual:



Note: Where 0 is low, 1 is high level, × is any level, when floating high.

- 1 # motor interface control signal logic

IN 1	IN 2	ENA 1	OUT1、OUT2
0	0	×	brake
1	1	×	Floating
1	0	PWM	Forward to speed
0	1	PWM	Reverse speed
1	0	1	Full speed forward
0	1	1	Full speed reversal

- 2 # motor interface control signal logic

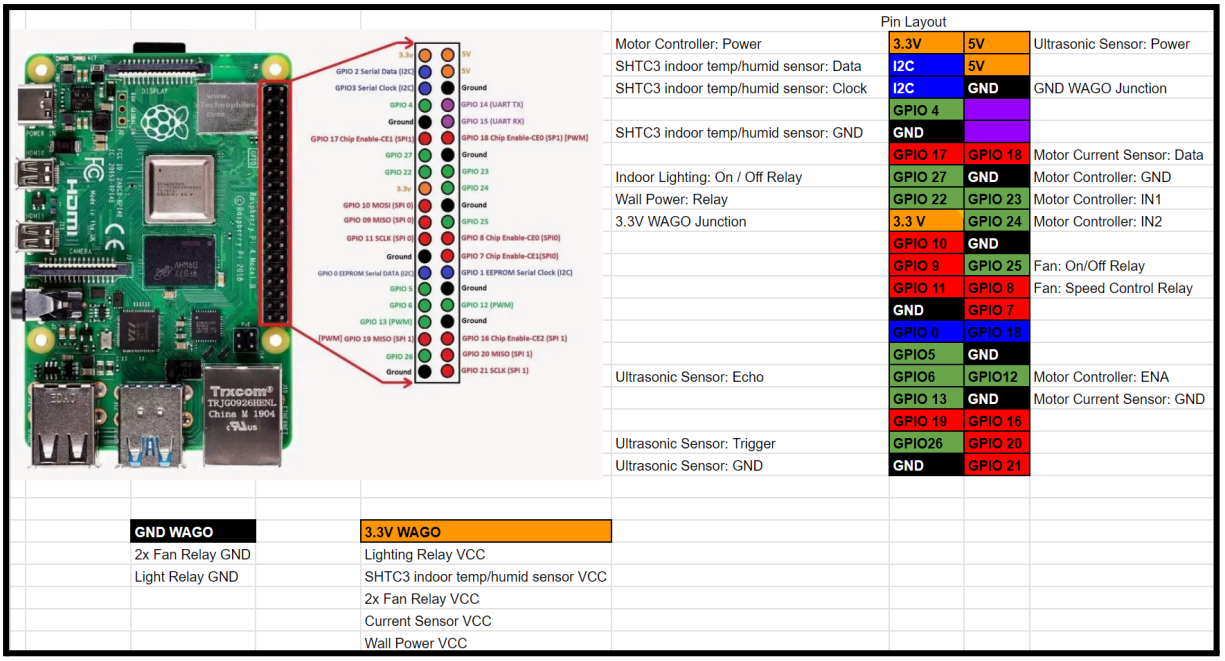
IN 3	IN 4	ENA 2	OUT3、OUT4
0	0	×	brake
1	1	×	Floating
1	0	PWM	Forward to speed
0	1	PWM	Reverse speed
1	0	1	Full speed forward
0	1	1	Full speed reversal

The manufacturer of the DC motor recommends that a PWM with a 20% duty cycle is used when operating the motor.

Although the pinout specifies 5 volts, the documentation does say that 3.3 volts will also work just as fine, and that is what is used. It is strongly recommended that developers read the documentation for the DC motor driver before making any electrical changes to avoid destroying the driver.

Note: All components purchased for this project and their websites including the official documentation are linked and listed in the Bill of Materials located in /documentation/Bill of Materials.xlsx of the git repository.

The Raspberry Pi pinout is shown below for the reader's convenience:



		Pin Layout		
	Motor Controller: Power	3.3V	5V	Ultrasonic Sensor: Power
	SHTC3 indoor temp/humid sensor: Data	I2C	5V	
	SHTC3 indoor temp/humid sensor: Clock	I2C	GND	GND WAGO Junction
		GPIO 4		
	SHTC3 indoor temp/humid sensor: GND	GND		
		GPIO 17	GPIO 18	Motor Current Sensor: Data
	Indoor Lighting: On / Off Relay	GPIO 27	GND	Motor Controller: GND
	Wall Power: Relay	GPIO 22	GPIO 23	Motor Controller: IN1
	3.3V WAGO Junction	3.3 V	GPIO 24	Motor Controller: IN2
		GPIO 10	GND	
		GPIO 9	GPIO 25	Fan: On/Off Relay
		GPIO 11	GPIO 8	Fan: Speed Control Relay
		GND	GPIO 7	
		GPIO 0	GPIO 1	
		GPIO5	GND	
	Ultrasonic Sensor: Echo	GPIO6	GPIO12	Motor Controller: ENA
		GPIO 13	GND	Motor Current Sensor: GND
		GPIO 19	GPIO 16	
	Ultrasonic Sensor: Trigger	GPIO26	GPIO 20	
	Ultrasonic Sensor: GND	GND	GPIO 21	

GND WAGO	3.3V WAGO
2x Fan Relay GND	Lighting Relay VCC
Light Relay GND	SHTC3 indoor temp/humid sensor VCC
	2x Fan Relay VCC
	Current Sensor VCC
	Wall Power VCC

The Door Motor

Pictured below are the wires connecting directly to the door motor

Software Operation

This section gives an overview of how the control code for the door motor is programmed.
The main script running the door operation is located in `/ControlCode/operations/door_operation.py` and works through messages sent to the door topic through MQTT (see the MQTT Topics section for more information).

Ideally, a flowchart describing how this software would be shown, however due to the multithreaded nature of the control code for this component the reader will unfortunately have to enjoy the wall of text below or read some of the comments in /ControlCode/operations/door_operation.py.

door_operation.py

This script directly interfaces with the motor controller / H-Bridge, and DC current sensor. The script also keeps track of the door's relative percentage open. It encodes how far open the door motor is by using a timer and relying on the motor's internal limit switch when fully closed. When the motor is fully closed, trying to close it more will do nothing. This way the door's current position can be calibrated by attempting to close it more than what is physically possible and setting the time-based percentage open to 0% on that calibration criteria.

If the door is continuously opened and closed without letting it fully close, the script's calibration will become less and less accurate. From this, there is a danger that if an uncalibrated script attempts to open the door, it may attempt to open it further than what the hinges allow and burn the motor out. The failsafe for this is the DC current sensor. This sensor is tuned to send a high logic value to GPIO 18 when the current being drawn by the motor exceeds a set threshold and vice versa. Thus if the motor attempts to push directly against the hinges due to its calibration being off, the spike in current consumption will be detected and a stop command will be sent.

The script is reactive / passive in that whenever a request is sent to the 'door' MQTT topic (see the Dock House MQTT Topics section for more details about this topic and the messages posted there), the 'door_operation' script will then perform that action. The only active thing that the script does is directly calling the function to stop the door when the current sensor detects an obstacle blocking the door.

Spot Detection

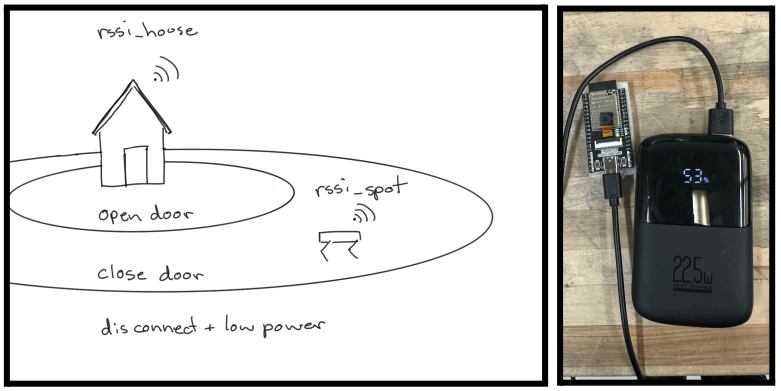
Spot detection can occur through the Spot API or the RSSI Spot Collar. This section was written by Joelle Bailey who wrote the code for the spot collar. As such the standards of this section may not match the standards of the rest of the document.

Spot API

The Spot API is a Boston Dynamics developed Python interface to the Spot robot. The first mode of operation is an attempt to connect to the Spot robot. Please refer to the spot_operation.py file for further details of how this is implemented.

RSSI Spot Collar

The RSSI Spot Collar is an ESP32 that can be secured to the Spot robot and plugged into one of the on robot power sources. This microcontroller is programmed to connect with the dock house WiFi (Version 1.0: SpotWifi) and send the received signal strength (RSSI) of that device back over MQTT to the dock house. Any changes to the dock house WiFi or IP address will need to be updated in the ESP32 code and the device will need to be reflashed. Please refer to the code base for more information and access to this code.

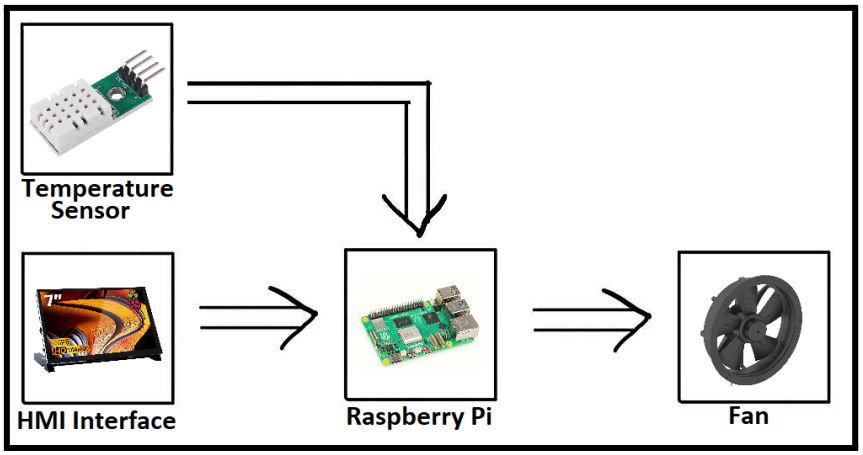


ESP32 - WROVER, selected for WiFi & Bluetooth, Battery Pack to Simulate Spot/Robot USB port

FAN / CLIMATE CONTROL

High-Level Overview

This section provides a high-level overview of how the fan and climate control works in the dock house. Below you can see a picture that shows the order in which the control signals flow through our dock house.



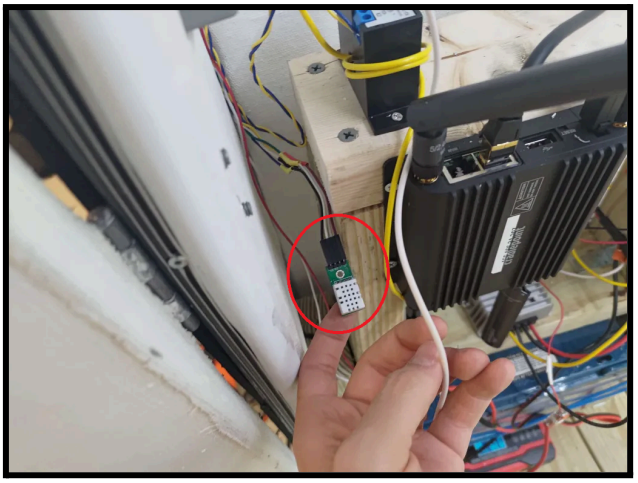
The above image shows the two ways in which the fan speed is controlled. In manual mode, a user can use the HMI interface to manually set the speed of the fan. In automatic mode, the Raspberry Pi reads the temperature sensor data and then adjusts the fan speed accordingly.

Hardware Connections

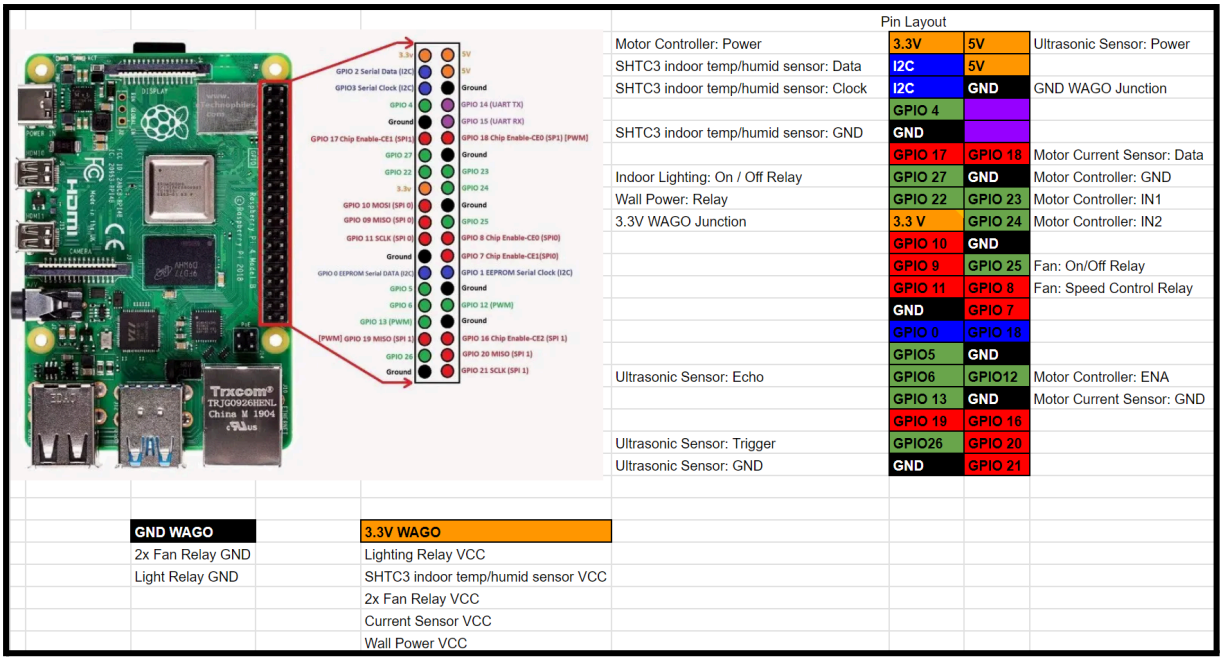
This section describes all the wired connections that the fan and temperature sensor use to operate.

SHTC3 Indoor Temperature Sensor

The SHTC3 temperature sensor can be found next to “power shelf” circled in red in the image below:



The temperature sensor communicates via I2C and its connection to the Raspberry Pi is shown in the pinout below:

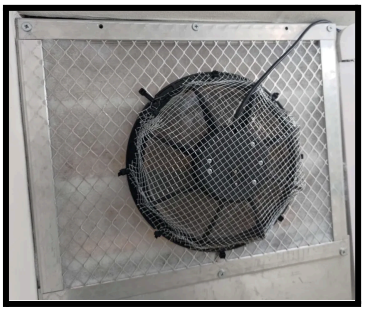


		Pin Layout		
	Motor Controller: Power	3.3V	5V	Ultrasonic Sensor: Power
	SHTC3 indoor temp/humid sensor: Data	I2C	5V	
	SHTC3 indoor temp/humid sensor: Clock	I2C	GND	GND WAGO Junction
		GPIO 4		
	SHTC3 indoor temp/humid sensor: GND	GND		
		GPIO 17	GPIO 18	Motor Current Sensor: Data
	Indoor Lighting: On / Off Relay	GPIO 27	GND	Motor Controller: GND
	Wall Power: Relay	GPIO 22	GPIO 23	Motor Controller: IN1
	3.3V WAGO Junction	3.3 V	GPIO 24	Motor Controller: IN2
		GPIO 10	GND	
		GPIO 9	GPIO 25	Fan: On/Off Relay
		GPIO 11	GPIO 8	Fan: Speed Control Relay
		GND	GPIO 7	
		GPIO 0	GPIO 18	
		GPIO5	GND	
	Ultrasonic Sensor: Echo	GPIO6	GPIO12	Motor Controller: ENA
		GPIO 13	GND	Motor Current Sensor: GND
		GPIO 19	GPIO 16	
	Ultrasonic Sensor: Trigger	GPIO26	GPIO 20	
	Ultrasonic Sensor: GND	GND	GPIO 21	

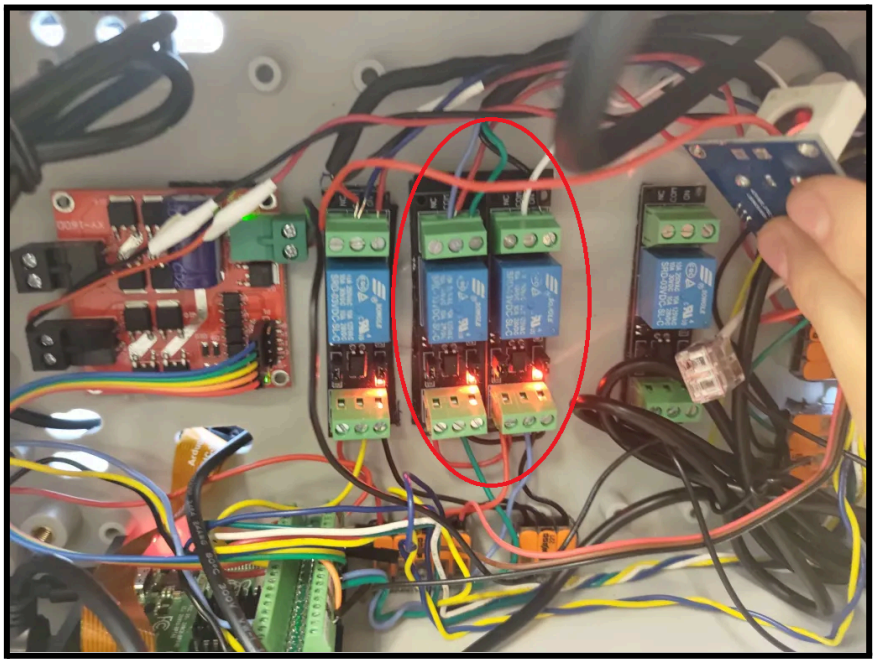
GND WAGO	3.3V WAGO
2x Fan Relay GND	Lighting Relay VCC
Light Relay GND	SHTC3 indoor temp/humid sensor VCC
	2x Fan Relay VCC
	Current Sensor VCC
	Wall Power VCC

Fan

The fan is shown below:



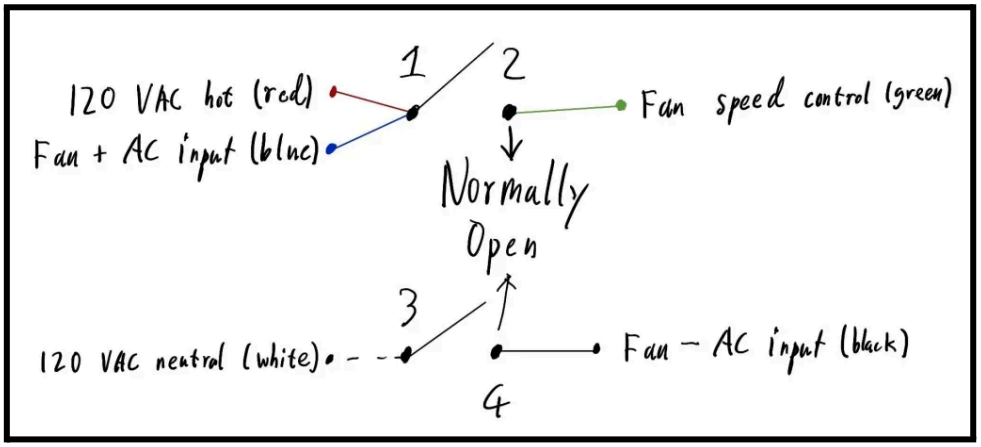
Electrically the fan has 3 wires (blue, black, and green) connected at the top of the two relays circled in red below:



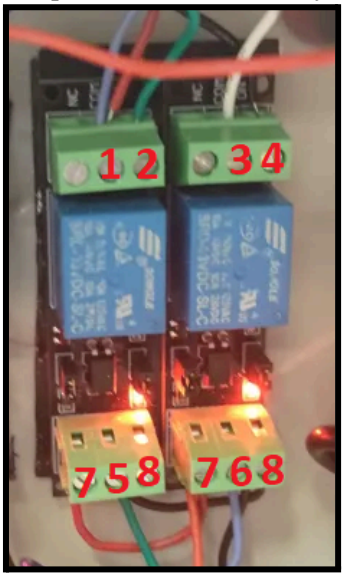
Blue and black are the positive and negative AC power terminals for the fan. Green is used to control the fan speed. When green is shorted with either blue or black, the fan speed will be set to slow.

The red and white wires going to each of the relays (also at the top), carry the 120 VAC from the wall power.

The electrical diagram below hopefully will make this electrical connection more intuitively understood (the ports of the relays are numbered the same as in the image after the electrical diagram):



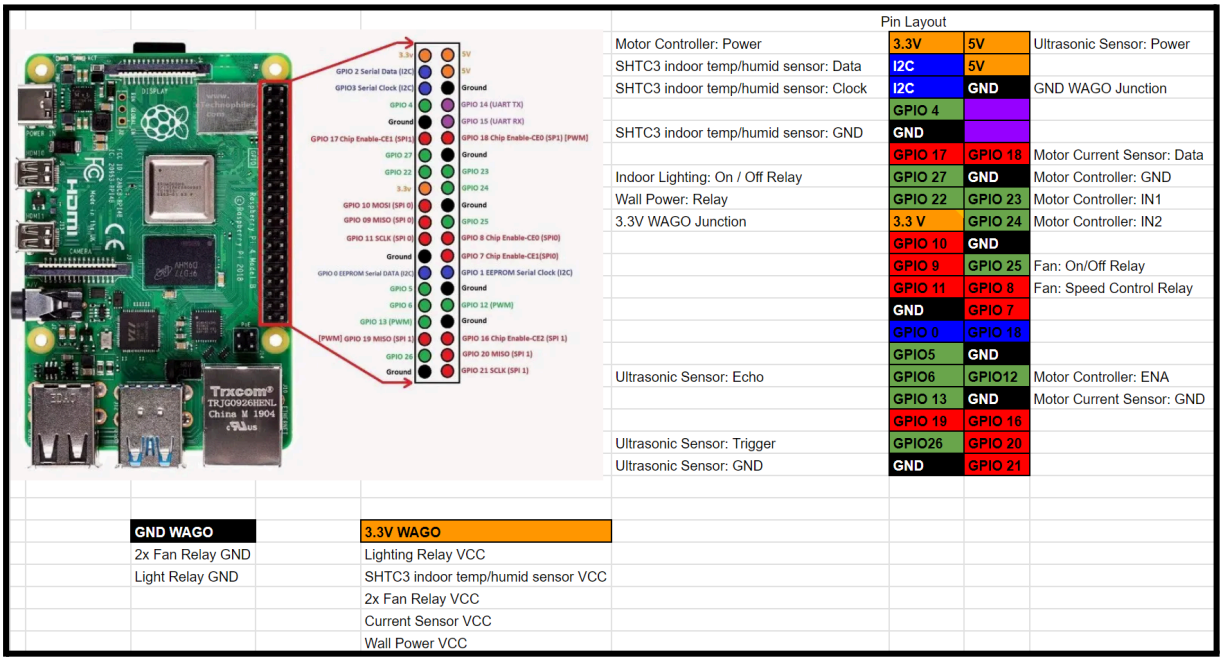
The table and picture below explains the pinout for the two relays in more detail:



Port	Description
1 – Blue + Red	The common port of relay. The positive terminal of 120 VAC input (red), and the fan's positive AC input (blue) are connected to this port.
2 – Green	The normally open port of the relay. The fan's speed control wire (green) is connected to this port. When port 5 receives a logic high signal, the relay will short common with this terminal, and the fan speed will be set to slow.
3 – White	The common port of the relay. The negative terminal of 120 VAC input (white) is connected to this port.
4 – Black	The normally open port of the relay. The fan's negative AC input wire (black) is connected to this port. When port 6 receives a logic high signal, the relay will short common with this terminal, and the fan will be turned on.
5 – Green	This port connects to GPIO 8 on the Raspberry Pi, allowing it to control the relay.

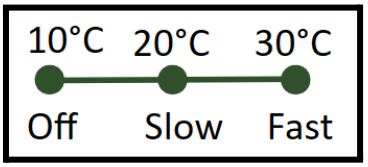
Port	Description
6 – Blue	This port connects to GPIO 25 on the Raspberry Pi, allowing it to control the relay.
7 – Red	VCC for the relay, is connected to 3.3 volts supplied by the Raspberry Pi.
8 – Black	Ground for the relay, is connected to Raspberry Pi's ground.

Below is an image of the Raspberry Pi pinout for the reader's convenience:



Software Operation

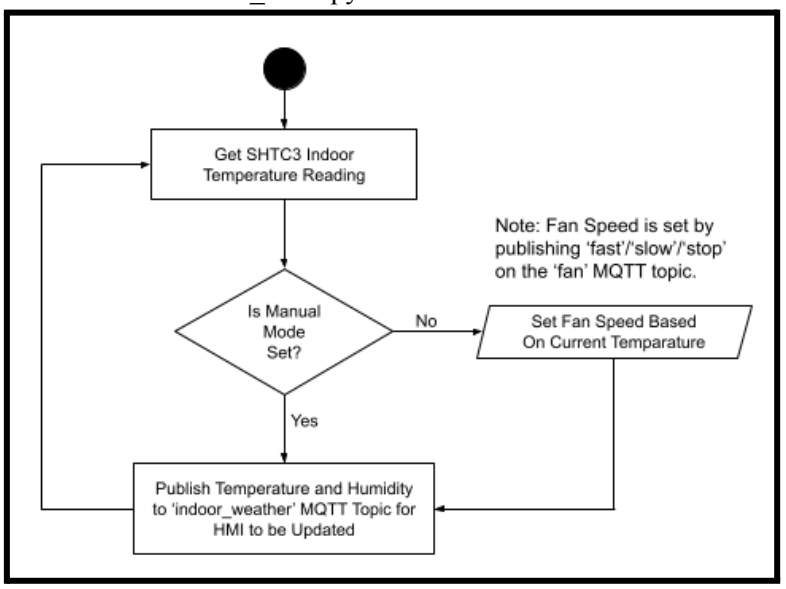
This section gives an overview of how the control code for the fan and temperature sensor is programmed. The climate control portion is handled by comparing the ambient internal temperature with certain setpoints. If the ambient temperature is less than the ‘off’ setpoint, the fan is turned off. If the temperature exceeds the slow setpoint, the fan speed is set to ‘slow’. If the temperature exceeds the ‘fast’ setpoint, the fan speed is set to fast. Below is an image illustrating these setpoints and their default configuration.



A user can set the fan into manual mode in the HMI. When in this mode, the fan speed is set manually by the user, using a slider on the HMI (to read more about these two display components on the HMI see the Table of HMI Features section). The control code to accomplish this is split between the two files /ControlCode/operations/fan_operation.py, and /ControlCode/operations/climate_main.py.

climate_main.py

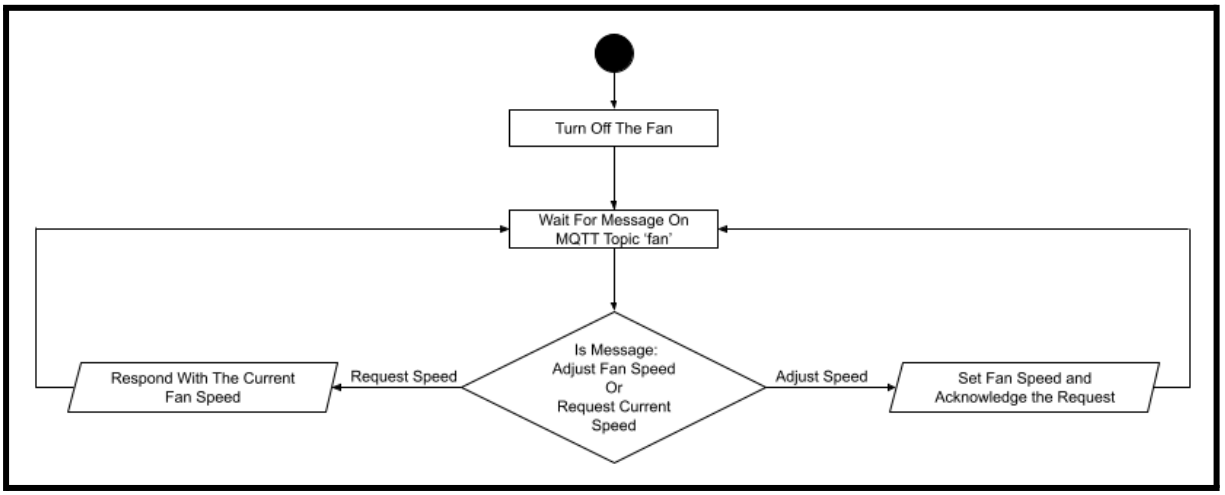
A flowchart that describes how climate_main.py works is shown below:



The script first reads in the current temperature sensor reading and then adjusts the fan speed if manual mode is not set. Manual mode can be set by an MQTT ISR for the topic ‘fan_HOA’ (to read more about this MQTT topic and what is sent see the Dock House MQTT Topics section). Then regardless of whether manual mode is set or not, the script then publishes the current temperature and humidity to the ‘indoor_weather’ MQTT topic. The script then loops back to read the current temperature.

fan_operation.py

A flowchart that describes how fan_operation.py works is shown below:

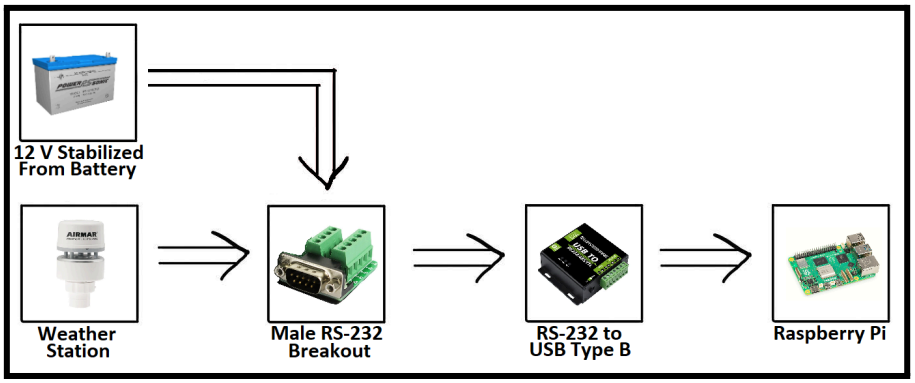


On start, the fan is turned off. The script then waits for a new message on the 'fan' MQTT topic. The Script then classifies the message into either a command to adjust the speed or a request for the current speed. If the message is a command, the script applies the command and responds with 'is_off'/'is_slow'/'is_fast' to confirm that the command has been applied. If the message is a request, the script replies with 'is_off'/'is_slow'/'is_fast' depending on the current speed of the fan.

WEATHER STATION

High-Level Overview

This section provides a high-level overview of how the weather station works in the dock house. Below you can see a picture that shows the order in which the control signals flow through our dock house.



Weather data from the weather station is sent via the RS-232 protocol. The weather station isn't directly wired to a female RS-232 connector, so it is first wired through to a male RS-232 breakout board that is then connected to a female RS-232 connector. At the breakout board, 12 volts is supplied to power the weather station.

The female RS-232 cable then connects the weather station to an isolated RS-232 to USB Type B converter. It's isolated to prevent a ground loop effect as the data is technically being sent to the ground respective to the Raspberry Pi and not the 12-volt power rail (although in practice it doesn't matter because the Raspberry Pi is powered by the same 12 volts).

The USB type B then connects to a USB type A port on the Raspberry Pi.

Hardware Connections

This section describes all the wired connections that the weather station uses to operate.

Weather Station

An image of the weather station mounted on the dock house is shown below:

The weather station's pinout and the RS-232 standard from their documentation are shown below:

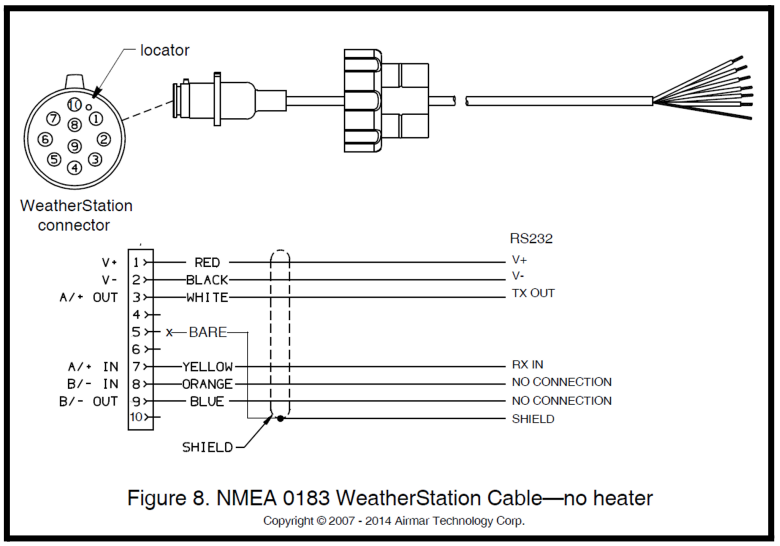
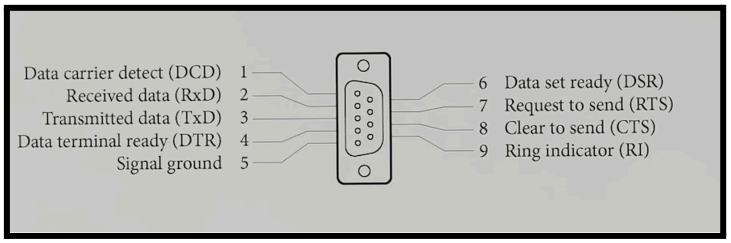
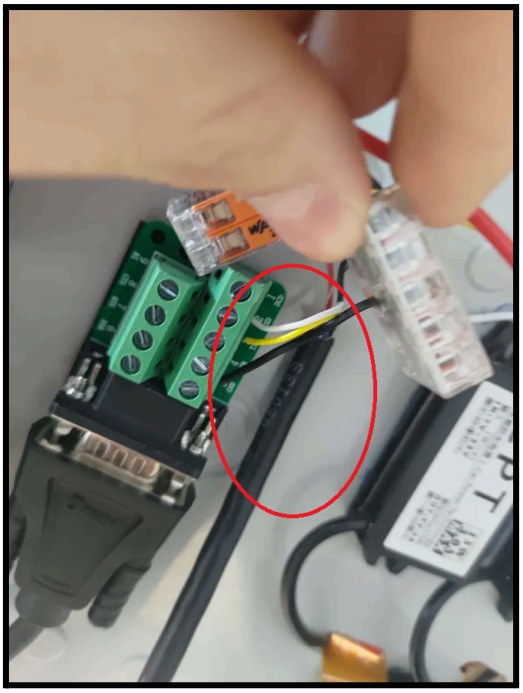


Figure 8. NMEA 0183 WeatherStation Cable—no heater

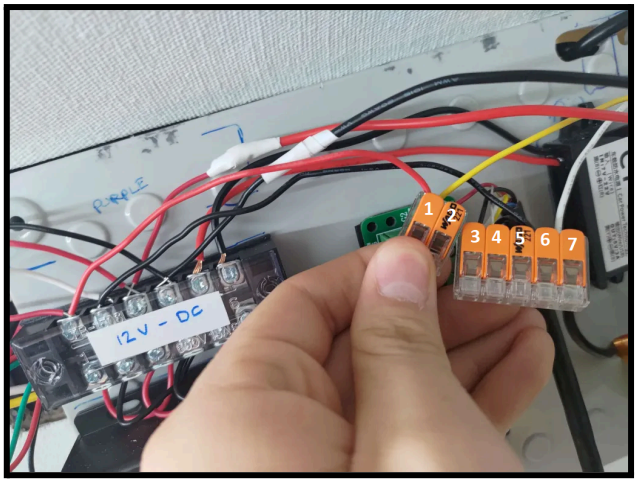


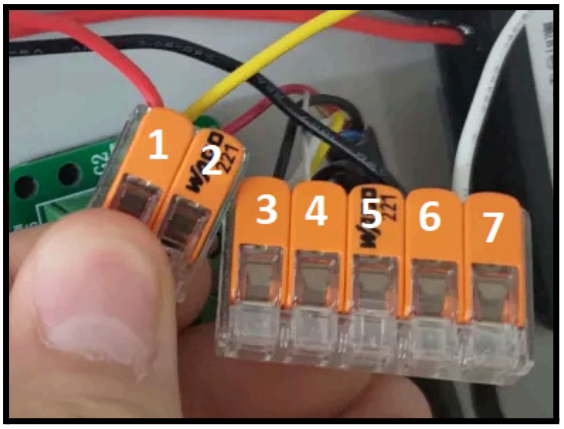
Electrically, ‘V+’ is connected to 12 volts from the battery. ‘V-’ is connected to ground from the battery and ‘GND’ on the breakout board for ‘Signal ground’. ‘TX OUT’ is connected to ‘RxD’. ‘RX IN’ is connected to ‘TxD’. Shield is connected to ground from the battery.

The wire circled in red is the cable coming out of the weather station:



From the image above you will also note a white wire going into ‘RXD’, this comes directly from the weather station cable and is ‘TX OUT’. A yellow wire going into ‘TXD’, this comes directly from the weather station cable and is ‘RX IN’. A black wire going into ‘GND’, this comes from a 12 volt WAGO junction which is further described in the images and table below:

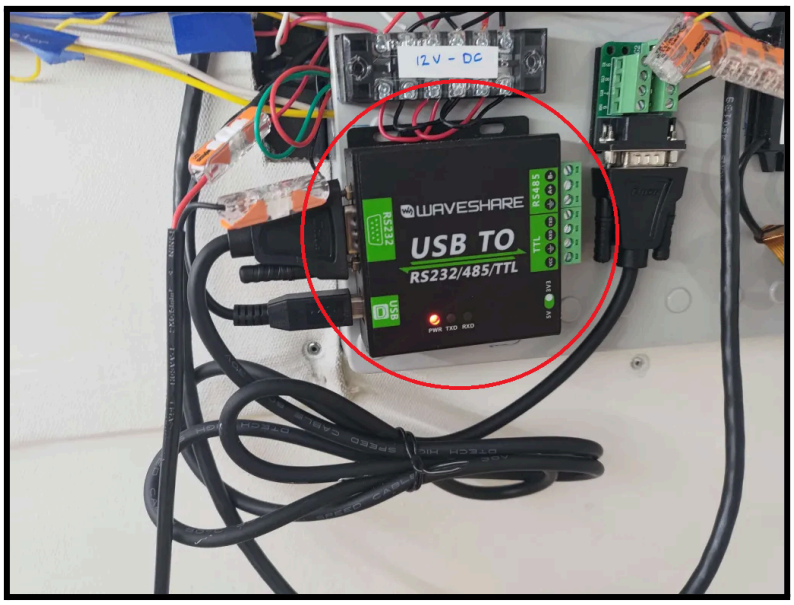




Port	Description
1 – Red + Yellow	The red cable connects to the 12-volt rail on the back of the control box. The yellow cable connects to the + input to the outdoor light.
2 – Red	The red cable connects to ‘V+’ on the weather station cable.
3 – Black	The black cable connects to ‘V-’ on the weather station cable.
4 – Bare	The bare cable connects to the shield of the weather station cable.
5 – Black	The black cable connects to ‘GND’ on the RS-232 male breakout board.
6 – Black	The black cable connects to the ground on the 12-volt rail.
7 – White	The white cable connects to the ground input to the outdoor light.

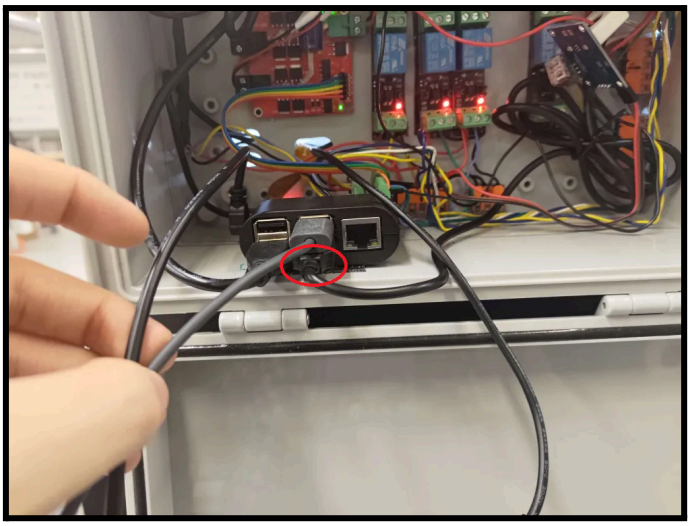
RS-232 to USB Type B

Pictured below and circled in red is the isolated RS-232 to USB Type B converter:



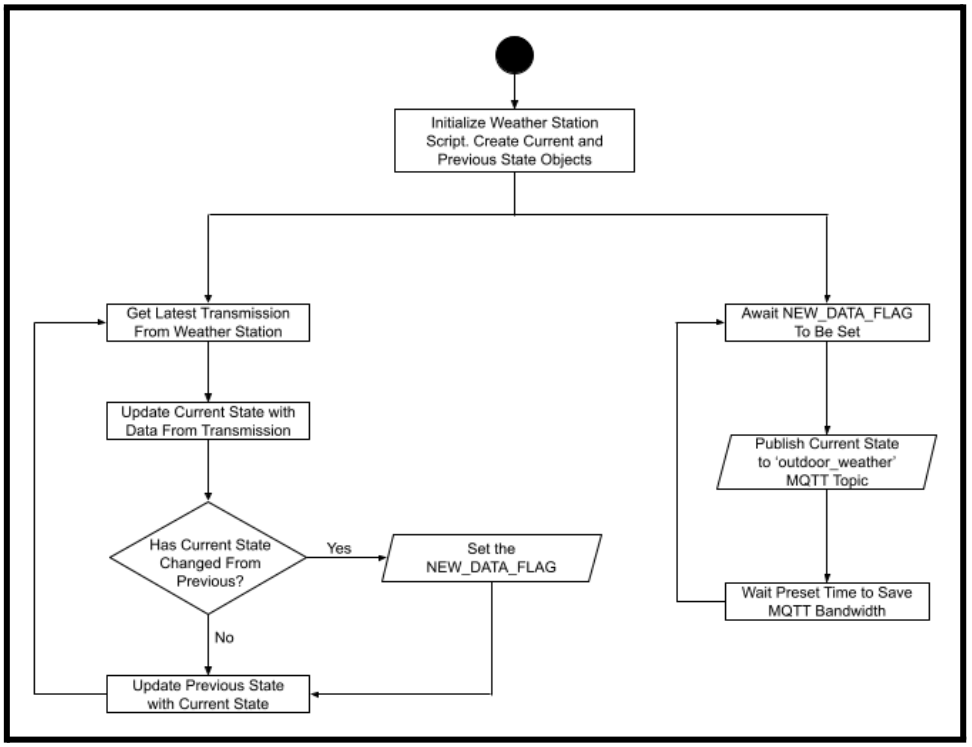
The converter is isolated so that the power source for the weather station and Raspberry Pi need not be the same (to avoid the ground loop effect). Currently this does not matter as the Raspberry Pi and Weather Station share the same power source (the 12-volt rail).

A USB Type B to USB Type A cable then connects the converter to the Raspberry Pi directly and the connection is circled in red below:



Software Operation

This section gives an overview of how the control code for the weather station is programmed. Below is a flowchart providing an overview on how the weather station control code works:



At start, the script initializes a weather station class. This initialization includes opening up the serial port connection with the weather station, creating a current and previous state dictionary, creating the MQTT client, and compiling the regular expressions for the NMEA sentences. The script then creates two asynchronous tasks that run in parallel.

Retrieving Data from The Weather Station

The first task scans the data sent by the weather station on the serial port for the following NMEA 0183 sentences using regular expressions. The official documentation for these NMEA sentences will be shown on the following page.

Once found, the data is parsed into the current state dictionary and compared with the previous state. If there is a difference between the two states, the NEW_DATA_FLAG is set. This lets the publisher task know that there is something for it to publish. After that, the task updates the previous state with the current state and loops back to the top of the task.

$WIMWV

$PAMTC,EN: MWVR	**Default Interval (m:ss.00)** 0:0.50	**Enabled By Default:** Yes
$PAMTC,EN: MWVT	**Default Interval (m:ss.00)** 0:1.00	**Enabled By Default:** Yes

Summary

NMEA 0183 standard Wind Speed and Angle, in relation to the vessel's bow/centerline.

Syntax

`$WIMWV,<1>,<2>,<3>,<4>,<5>*hh<CR><LF>`

Fields

<1> Wind angle, 0.0 to 359.9 degrees, in relation to the vessel's bow/centerline, to the nearest 0.1 degree. If the data for this field is not valid, the field will be blank.

<2> Reference:
R = Relative (apparent wind, as felt when standing on the moving ship)
T = Theoretical (calculated actual wind, as though the vessel were stationary)

<3> Wind speed, to the nearest tenth of a unit. If the data for this field is not valid, the field will be blank.

<4> Wind speed units:
K = km/hr
M = m/s
N = knots
S = statute miles/hr

In the WX Series WeatherStation Sensor, this field always contains "N" (knots).

<5> Status:
A = data valid; V = data invalid

Notes

Depending on the contents of the Reference field (field <2>), this sentence provides either relative (apparent) wind or theoretical (true) wind data, both in relation to the bow of the vessel. As it is conceivable that both of these forms could be useful simultaneously, the WX Series WeatherStation Sensor may output this sentence twice, once in each form.

$WIMDA

$PAMTC,EN: MDA	Default Interval (m:ss.00) 0:1.00	Enabled By Default: Yes

Summary

NMEA 0183 standard Meteorological Composite.

Syntax

```
$WIMDA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,
<12>,<13>,<14>,<15>,<16>,<17>,<18>,<19>,<20>*hh
<CR><LF>
```

Fields

<1>	Barometric pressure, inches of mercury, to the nearest 0.01 inch
<2>	I = inches of mercury
<3>	Barometric pressure, bars, to the nearest .001 bar
<4>	B = bars
<5>	Air temperature, degrees C, to the nearest 0.1 degree C
<6>	C = degrees C
<7>	Water temperature, degrees C (this field left blank by the WX Series WeatherStation Sensor)
<8>	C = degrees C (this field left blank by the WX Series WeatherStation Sensor)
<9>	Relative humidity, percent, to the nearest 0.1 percent (this field left blank if the relative humidity sensor is not available in the WX Series WeatherStation Sensor, see Notes)
<10>	Absolute humidity, percent (this field left blank by the WX Series WeatherStation Sensor)
<11>	Dew point, degrees C, to the nearest 0.1 degree C (this field left blank if the relative humidity sensor is not available in the WX Series WeatherStation Sensor, see Notes)
<12>	C = degrees C
<13>	Wind direction, degrees True, to the nearest 0.1 degree
<14>	T = true
<15>	Wind direction, degrees Magnetic, to the nearest 0.1 degree
<16>	M = magnetic
<17>	Wind speed, knots, to the nearest 0.1 knot

<18> N = knots

<19> Wind speed, meters per second, to the nearest 0.1 m/s

<20> M = meters per second

Notes

This WX Series WeatherStation Sensor has a relative humidity sensor.

The barometric pressure provided in this sentence is the *altimeter setting*, which is the barometric pressure corrected for altitude above sea level. See the transmitted $YXXDR(A) sentence, and the received $PAMTC,ALT command, for further information.

NMEA 0183 TRANSMITTED SENTENCE

$HCHDT

$PAMTC,EN: HDT	**Default Interval (m:ss.00)** 0:0.50	**Enabled By Default:** No

Summary

NMEA 0183 standard Heading relative to True North

Syntax

$HCHDT,<1>,<2>*hh<CR><LF>

Fields

<1> Heading relative to True North

<2> T = True

Notes

The data in field <1> is only provided if both magnetic compass heading and magnetic variation values are available.

Publishing New Data

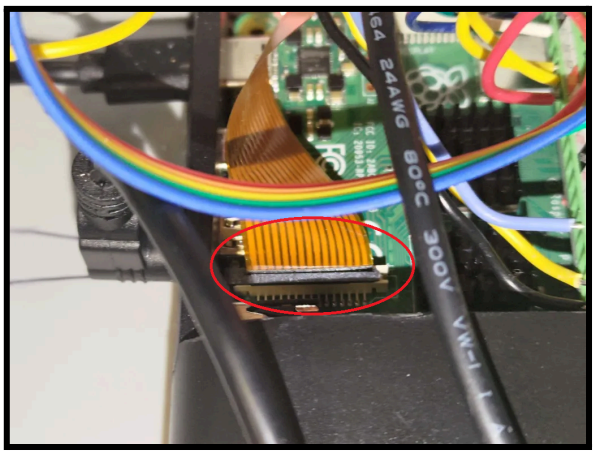
The publisher task works more simply. It waits for the NEW_DATA_FLAG to be set. When the flag is set, the publisher sends the data in the current state dictionary to the 'outdoor_weather' MQTT topic for the HMI to display. The task also unsets the flag for itself to mark the new data as published.

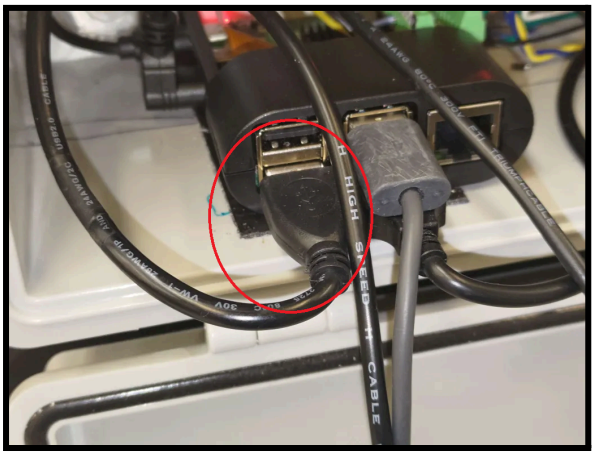
The task then waits a preset amount of time so that the MQTT topic isn't completely flooded with new information. The script then loops back up to wait for the NEW_DATA_FLAG to be set again.

Cameras

Hardware Connections

The cameras in the dock house are connected directly to the Raspberry Pi. One camera is connected via usb, and another is connected by an orange ribbon cable. These two connections are shown below and circled in red:





Below the two cameras (outdoor and indoor) are pictured:

Software Operation

The basic control code for the cameras is rather simplistic, and can be found within the /ControlCode/flask_webserver/flask_webserver.py script. The issue is that the simplistic implementation leaves a lot to be desired. One of the major issues is that when viewing the camera, the camera appears to only work for 5 seconds before freezing and never recovering. This may potentially be caused by a memory leak where the client is continuously sent new frames but never told to delete them.

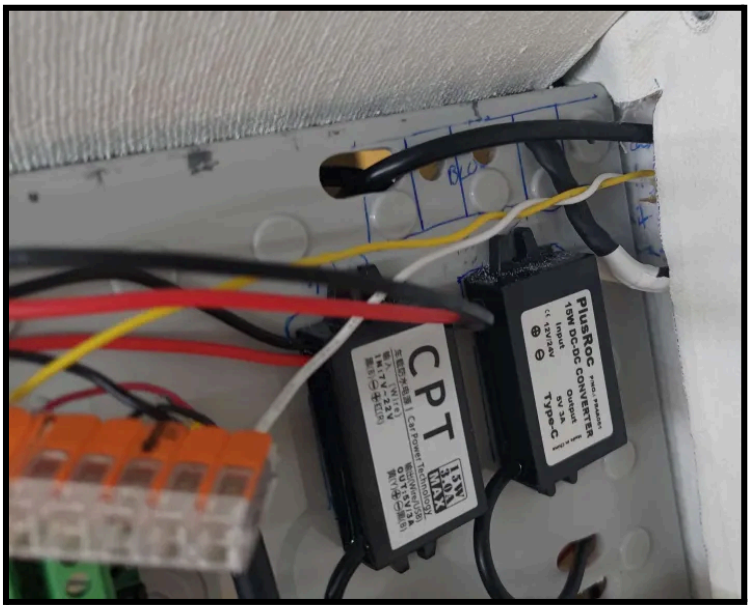
In its current implementation when a client clicks the button to view on of the cameras, the client is redirected to a subpage where the webserver continuously sends an HTML POST request to the client with the next frame. This is very slow, and clearly buggy. An initial attempt to replace this with a WebSocket implementation was made, but it didn't work.

LIGHTING

Hardware Connections

Outdoor Light

The outdoor light for the dock house is directly connected to a 12-volt rail at the back of the control panel as shown in the images below:



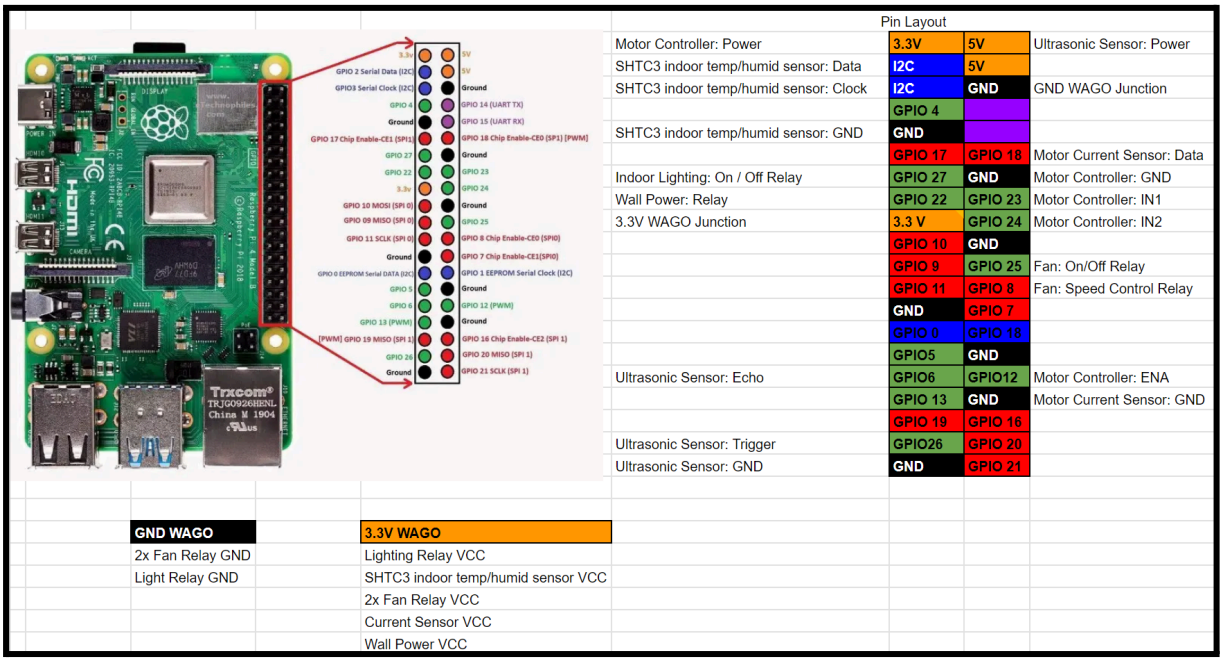
For a description of all the wires connected to the 12V and GND WAGO junction see the Weather Station section.

Indoor Light

The indoor light is also connected to the 12-volt junction, but it first goes through a relay inside the control box. From memory, I believe the indoor lighting uses the 12-volt and GND WAGO junctions inside the control box on the right. Below is an image and table describing the pinout for the indoor lighting relay inside the control box:

Port	Description
1 – Red + Blue	The red wire connects to the 12-volt input for the indoor lighting. From memory, I recall that the blue cable is not connected to anything (sorry).
2 – Red	The red wire connects to the 12-volt WAGO junction inside the box.
3 – Yellow	This port connects to GPIO 27 on the Raspberry Pi, allowing it to control the relay.
4 – Red	VCC for the relay, is connected to 3.3 volts supplied by the Raspberry Pi.
5 – Black	Ground for the relay, is connected to Raspberry Pi's ground.

Below is an image of the Raspberry Pi pinout for the reader's convenience:



		Pin Layout		
	Motor Controller: Power	3.3V	5V	Ultrasonic Sensor: Power
	SHTC3 indoor temp/humid sensor: Data	I2C	5V	
	SHTC3 indoor temp/humid sensor: Clock	I2C	GND	GND WAGO Junction
		GPIO 4		
	SHTC3 indoor temp/humid sensor: GND	GND		
		GPIO 17	GPIO 18	Motor Current Sensor: Data
	Indoor Lighting: On / Off Relay	GPIO 27	GND	Motor Controller: GND
	Wall Power: Relay	GPIO 22	GPIO 23	Motor Controller: IN1
	3.3V WAGO Junction	3.3 V	GPIO 24	Motor Controller: IN2
		GPIO 10	GND	
		GPIO 9	GPIO 25	Fan: On/Off Relay
		GPIO 11	GPIO 8	Fan: Speed Control Relay
		GND	GPIO 7	
		GPIO 0	GPIO 18	
		GPIO5	GND	
	Ultrasonic Sensor: Echo	GPIO6	GPIO12	Motor Controller: ENA
		GPIO 13	GND	Motor Current Sensor: GND
		GPIO 19	GPIO 16	
	Ultrasonic Sensor: Trigger	GPIO26	GPIO 20	
	Ultrasonic Sensor: GND	GND	GPIO 21	

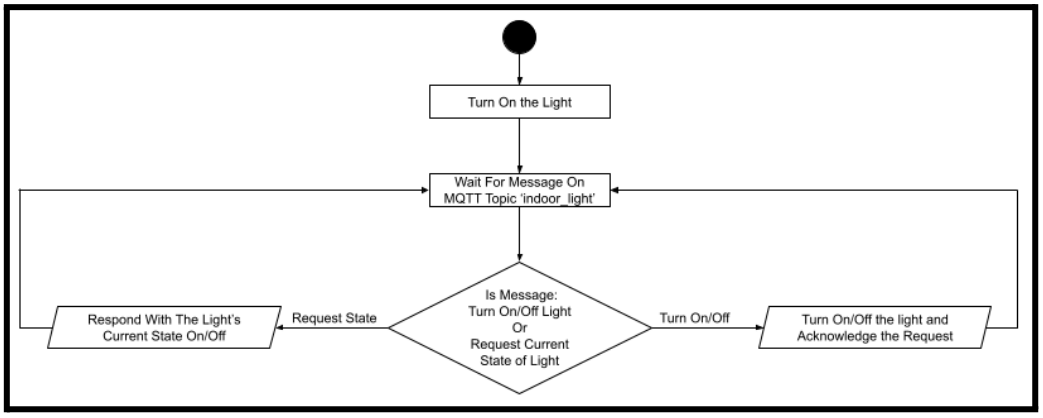
GND WAGO	3.3V WAGO
2x Fan Relay GND	Lighting Relay VCC
Light Relay GND	SHTC3 indoor temp/humid sensor VCC
	2x Fan Relay VCC
	Current Sensor VCC
	Wall Power VCC

Software Operation

This section gives an overview of how the control code for the lighting is programmed.

Indoor Lighting

The flowchart below describes how the indoor light is programmed:



On start, the GPIO port is initialized and the light is turned on. Then the program goes into an idle state where it waits for a message to be posted on the 'indoor_light' MQTT topic (for more information about this MQTT topic see the Dock House MQTT Topics section).

Once a message is detected, the message is classified into either a command to turn on / off the light or a request for the light's current state. If the message is a command, the script will first turn on / off the light appropriately and then respond on the same topic with 'is_on' or 'is_off'. If the message is a request, the script will respond on the same topic with 'is_on' or 'is_off' appropriately.

Outdoor Lighting

This system operates in isolation from the general control system. Details on its operation and how to adjust its sensitivity can be found in the product documentation here:

SUGGESTED UPGRADES

Our final design is definitely imperfect and has many flaws. Below is a list of suggested major improvements that should be made to the dock house.

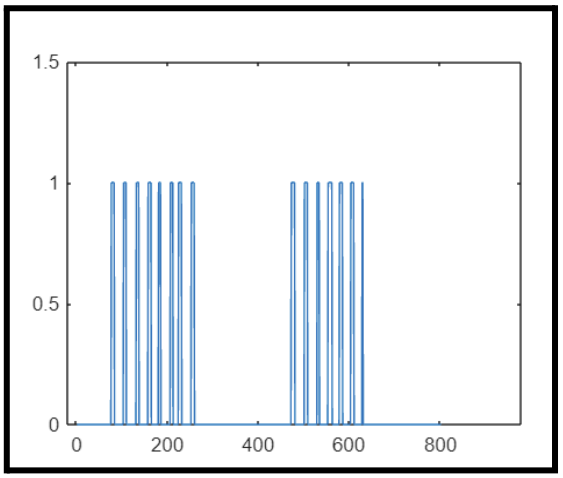
Door

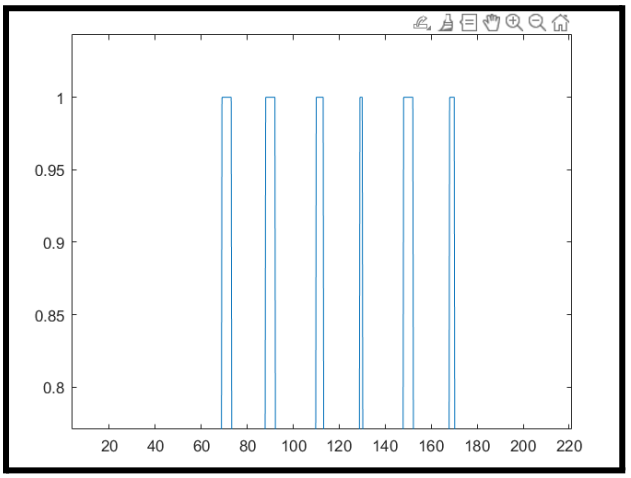
Motor

The linear actuator for the door does not open at a satisfactory speed. Replacing the motor with a faster alternative would be ideal.

Current Sensor

The current sensor is kind of glitchy. The expected output would be that the digital pin is set high when the current exceeds the set threshold. In testing what actually happens is the current sensor spikes as the door opens. The graph below illustrates the output of the digital pin as the door opens over time:





The Senior Design team did not spend too much time trying to debug the current sensor, so before looking into purchasing a different one, it would be wise to play around with the current sensor using a multimeter.

The way our door obstruction control code works is by counting the number of spikes from the current sensor's digital pin. When the spikes exceed a set value, the door is stopped. This works for the most part however there are a couple of issues.

Firstly with the way it's programmed the door has quite a delay between being obstructed and stopping. Lastly, there seems to be a bug where sometimes the door will just stop suddenly with no obstruction.

Interaction with Spot

The dock house has very minimal interaction with spot. Spot doesn't directly communicate with the dock house and rather alternatives like the spot collar is used. Also during expo, we could not get spot's current battery capacity to show up on the HMI.

Structure

Hand Cut Edges

Much of the structure for the dock house was hand-cut. Edges are thus very uneven which makes the structure not look super aesthetically pleasing.

Water Sealing

There are many gaps that can be seen from where the doors are installed. This makes me hesitant to call the dock house waterproof. I'd suggest that more thought be put into creating seals for the doors of the dock house and that the dock house go through testing against inclement weather before it actually be put to use.

Cameras

Website Integration

The code to grab frames from the camera feed and send them to clients accessing the HMI is very buggy. It seems to work for 5 seconds or so before crashing. This might be because the client is never told to discard the old frames it grabbed leading to a memory leak, I'm unsure. An attempt to fix this by getting the frames to be sent via WebSockets instead of HTML POST requests, but time ran out and we couldn't get it to work.

I suggest someone look into the code to get the camera feed to work and make it more reliable.

Placement

The indoor camera placement is obstructed by the linear actuator running across the center of the frame. It's hard to move the camera into a better spot because the ribbon cable that connects to the internal camera is very small. Figuring out a solution to this would also help a lot.