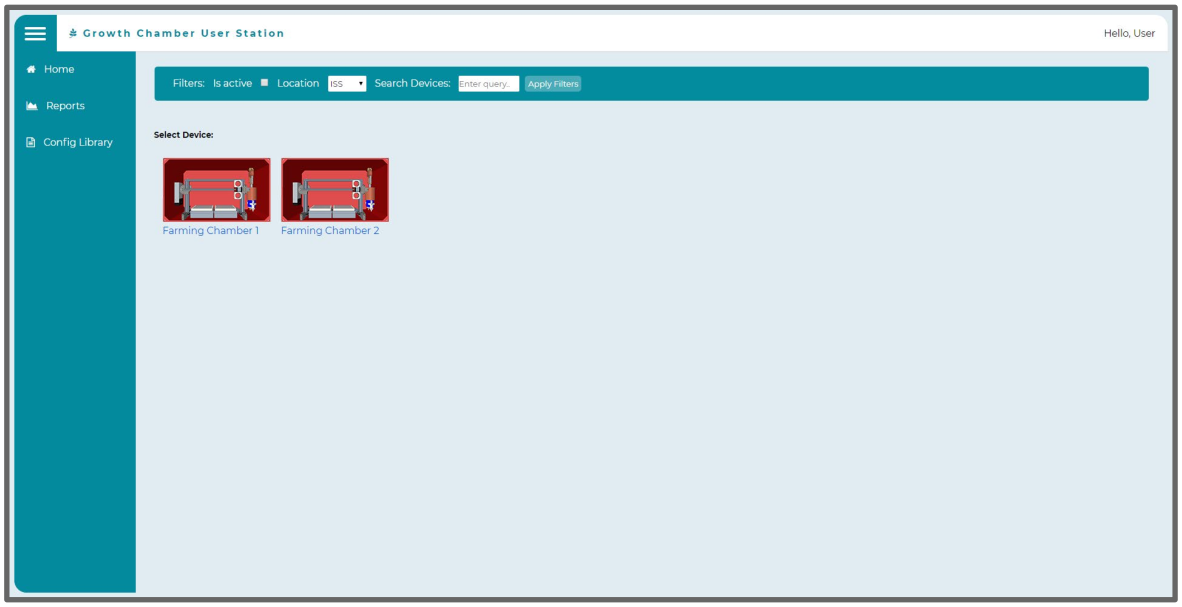
AMCF Software User Manual



Introduction

Purpose

The growth chamber software is used to autonomously schedule and operate sensors and actuators based on a premade configuration.

What is the Growth Chamber System?

The Growth Chamber

The growth chamber consists of the farming hardware, the microcontroller controlling the hardware, and the software on the microcontroller. The software uses automated scheduling to determine when and how to operate the hardware.

The User Station Application

The User Station is where the astronauts can load and edit configuration files, manually control the growth chamber, and view reports on the selected growth chamber.

Cycles and Phases

Based on the configuration, the growth chamber software will run through predefined phases of operation, such as, but not limited to, seeding, growing, harvesting, and cleanup. A cycle is one run of all phases defined in the configuration.

The Configuration File

The configuration file is a YAML formatted file that defines all the actions the automated scheduler should perform.

Astronauts - Using the User Station Application

Connecting to the Growth Chamber

Connections to the growth chamber are made by the system administrator. If you find that your device is not connected and it should be, contact your system administrator right away.

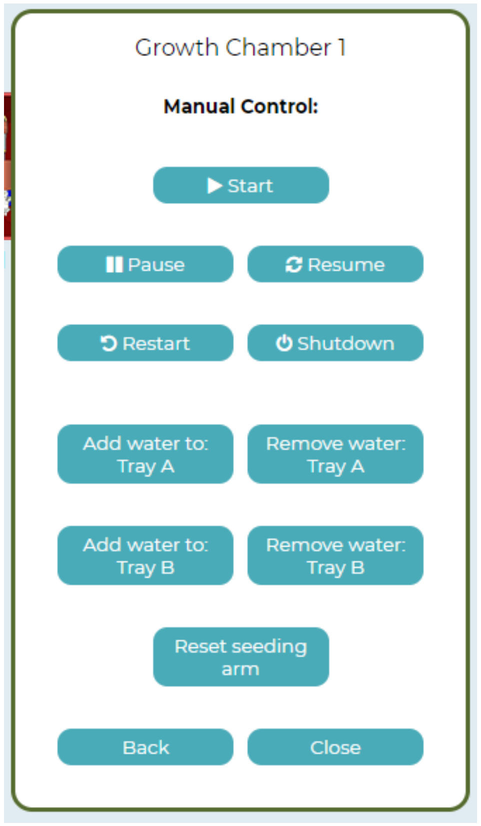
Connecting Over WiFi

The IP address of the Farming Chamber is set by the system administrator. It is static and chamber will connect automatically.

Connecting Over USB

If you need to connect to the Farming Chamber and your system administrator cannot assist you, just plug the USB cable from the computer to the Chamber and you will be connected automatically.

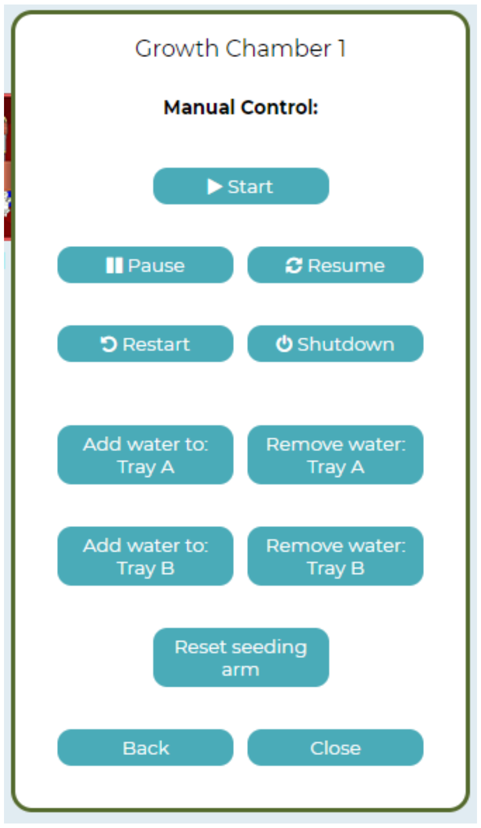
Manual Control



To navigate to the manual control box for your Growth Chamber simply select the desired chamber from the dashboard and click the manual control option on the first popup.

Adding Water

To add water to the tray first make sure the trays are inserted correctly, then navigate to the manual control box for the growth chamber you are configuring. Once at the manual control popup select ‘Add water to tray A’ or ‘Add water to tray B’.

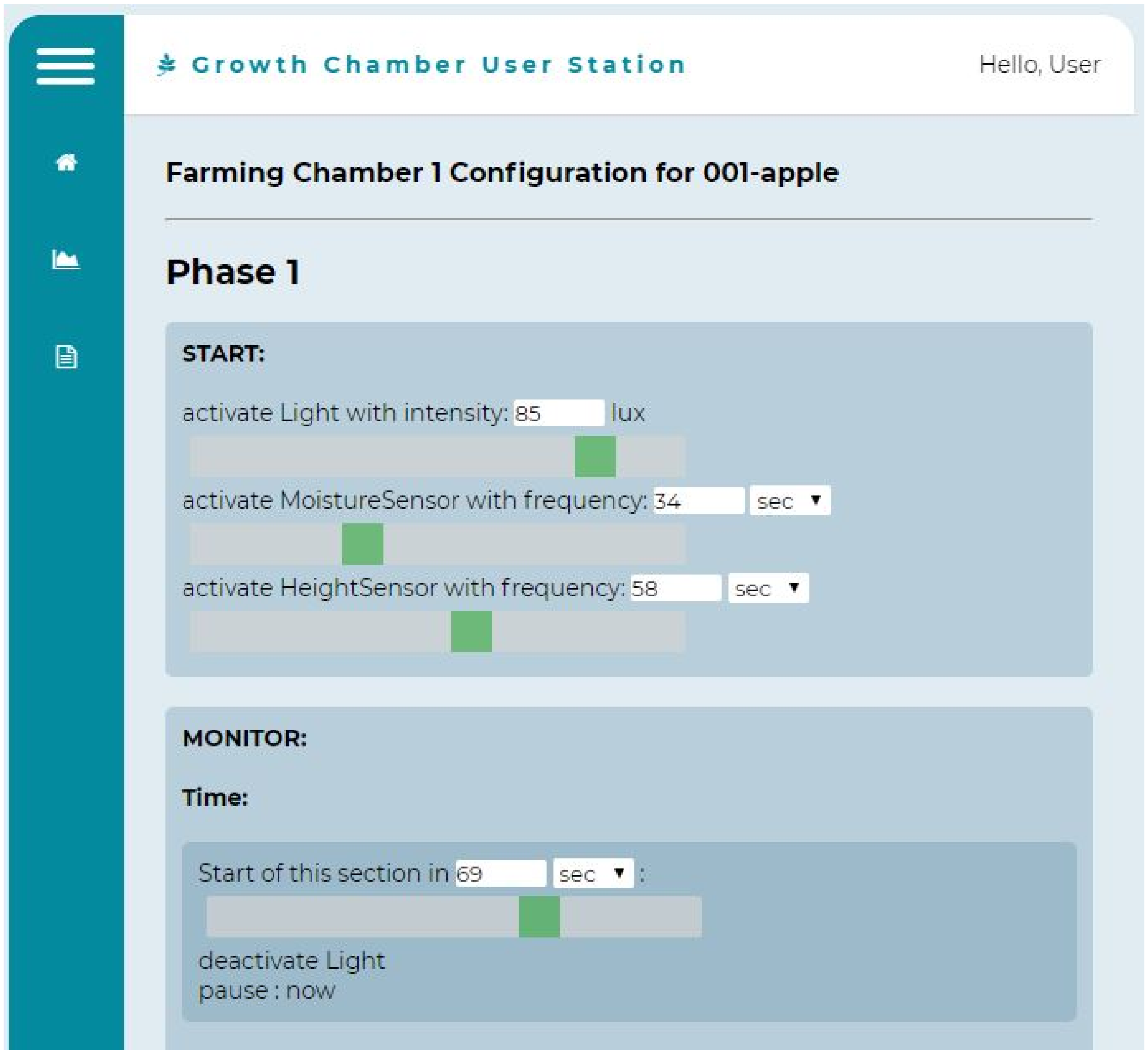


Resetting Seeding Arm

To reset the seeding arm simply navigate to the manual control box for your selected growth chamber and select ‘Reset seeding arm’. You should not need to use this function often, if you do be sure to consult your plant professional regarding the configuration running on the chamber.

Configuration Editor GUI

As the astronaut you are authorized to make minor changes to numerical values and units in the chosen configuration file.


Growth Chamber User Station

Hello, User

Farming Chamber 1 Configuration for 001-apple

Phase 1

START:

activate Light with intensity: 85 lux

activate MoistureSensor with frequency: 34 sec

activate HeightSensor with frequency: 58 sec

MONITOR:

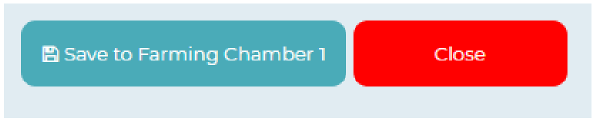
Time:

Start of this section in 69 sec :

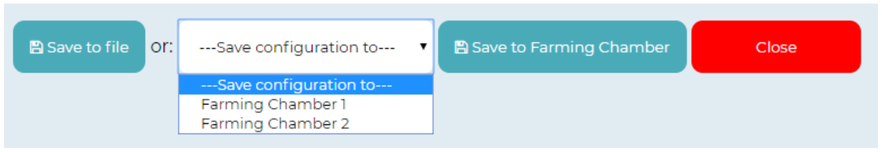
deactivate Light
pause : now

For most configuration files you will see four phases and a variety of fields to edit. To change a numerical value simply slide the slider to the desired value, to change a unit select the drop-down option at the end of the line you wish to change. You will not be able to set the values in the text boxes directly.

Once done you will need to save your configuration and record the change made.
If you are editing the config file from a Farming Chamber:



If you are editing a config file from the config library and you want to save it to a Chamber:

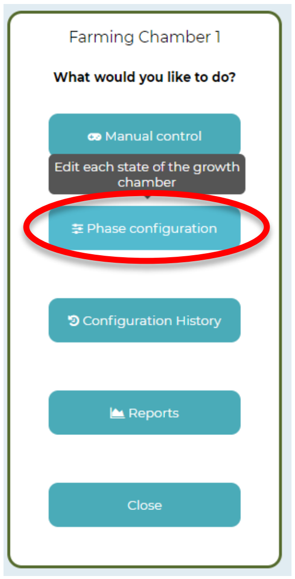


Modifying a Configuration

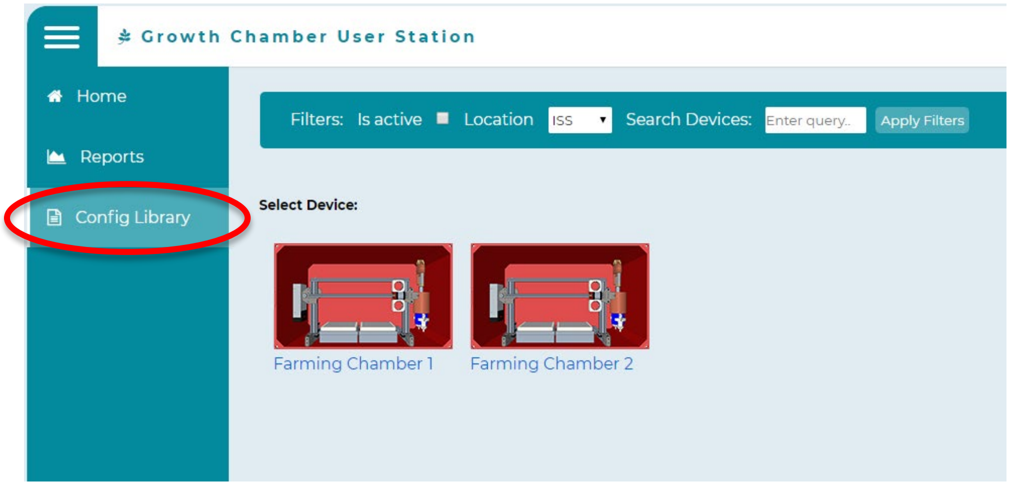
There are three ways to modify a configuration.

1. Modify a configuration that is loaded to a chamber
2. Modify a configuration in the configuration library.
3. Modify a configuration that was loaded into the chamber.

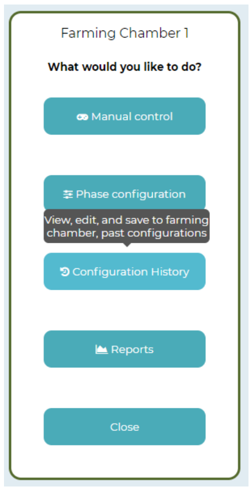
To modify a configuration that is loaded to a chamber select that chamber on the dashboard and select ‘Phase Configuration’.

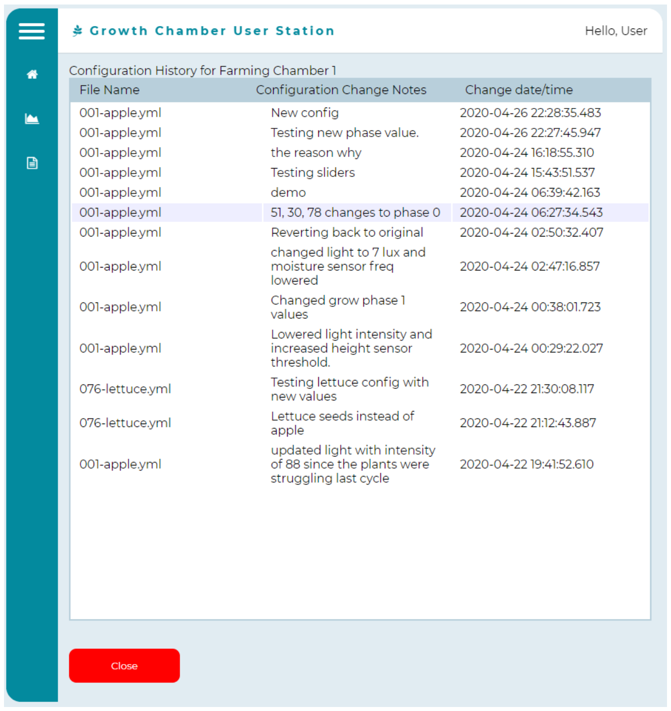


To modify a configuration from the library simply select the ‘config library’’ tab always shown on the Farming User Station.



To modify a past configuration, select the chamber then click on ‘Configuration History’ finally select the desired file.


Farming Chamber 1
What would you like to do?
Manual control
Phase configuration
View, edit, and save to farming chamber, past configurations
Configuration History
Reports
Close

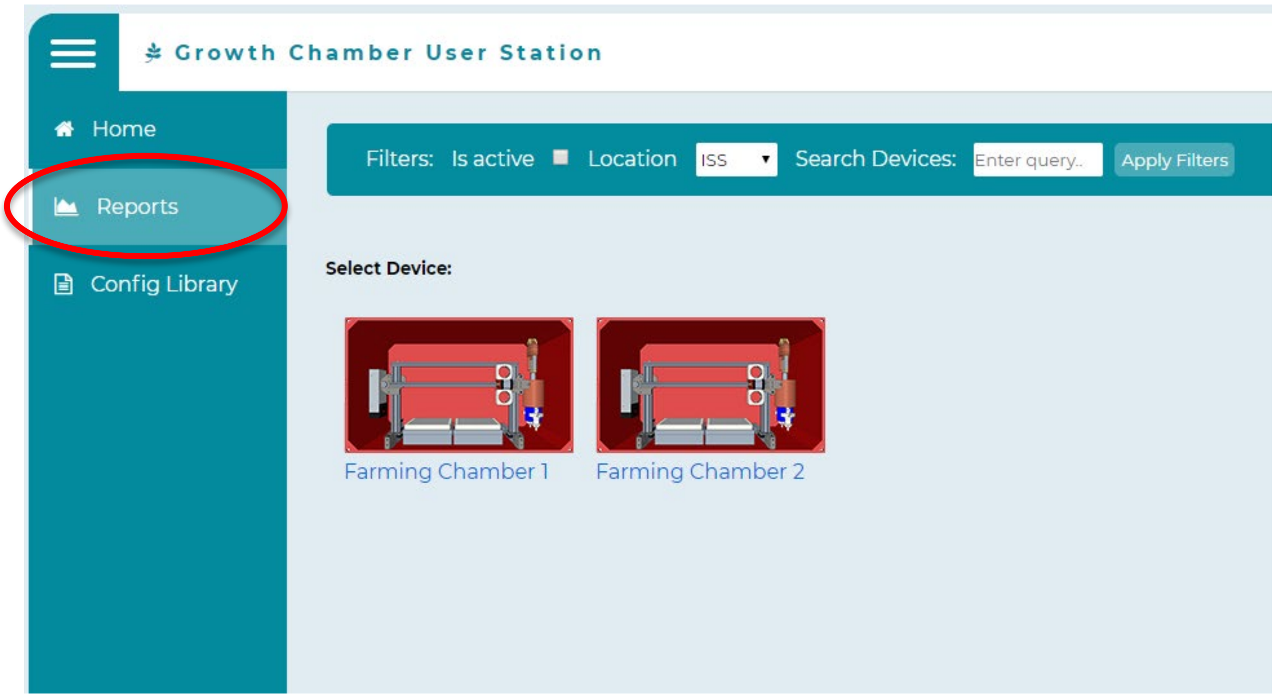


Configuration History for Farming Chamber 1

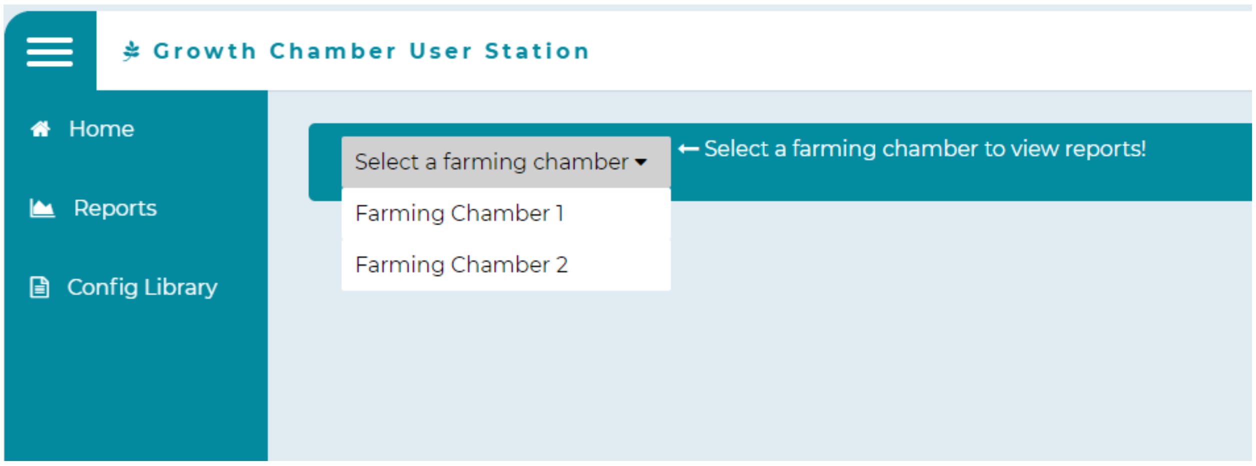
File Name	Configuration Change Notes	Change date/time
001-apple.yml	New config	2020-04-26 22:28:35.483
001-apple.yml	Testing new phase value.	2020-04-26 22:27:45.947
001-apple.yml	the reason why	2020-04-24 16:18:55.310
001-apple.yml	Testing sliders	2020-04-24 15:43:51.537
001-apple.yml	demo	2020-04-24 06:39:42.163
001-apple.yml	51, 30, 78 changes to phase 0	2020-04-24 06:27:34.543
001-apple.yml	Reverting back to original	2020-04-24 02:50:32.407
001-apple.yml	changed light to 7 lux and moisture sensor freq lowered	2020-04-24 02:47:16.857
001-apple.yml	Changed grow phase 1 values	2020-04-24 00:38:01.723
001-apple.yml	Lowered light intensity and increased height sensor threshold.	2020-04-24 00:29:22.027
076-lettuce.yml	Testing lettuce config with new values	2020-04-22 21:30:08.117
076-lettuce.yml	Lettuce seeds instead of apple	2020-04-22 21:12:43.887
001-apple.yml	updated light with intensity of 88 since the plants were struggling last cycle	2020-04-22 19:41:52.610

Reports

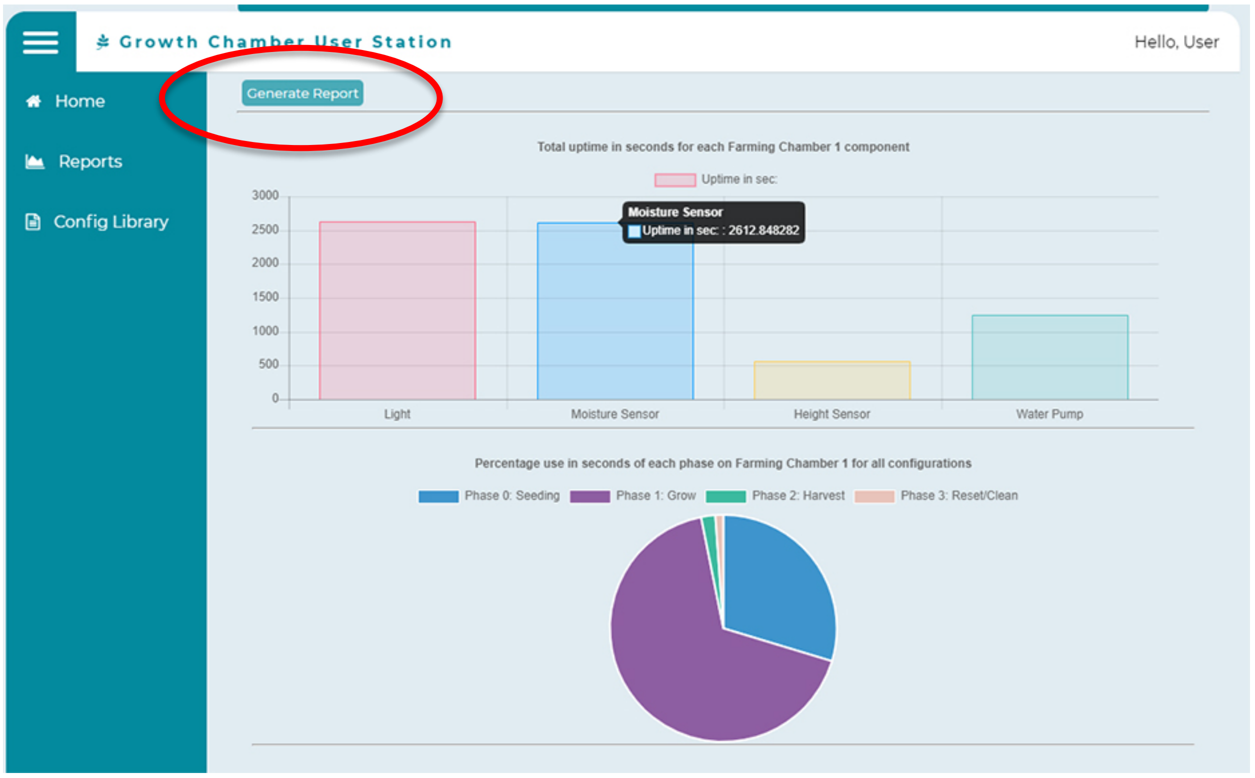
To view reports, select the ‘Reports’ tab on the left of the dashboard. The reports give various metrics on Chamber performance. Use these to determine system health.



Once on the reports page select the Chamber you wish to view



Then click generate report button:



Reports will show uptime for components of the Chamber and uptime for particular phases that have been running on the selected Chamber.

Botanists - Configuring the Autonomous Scheduler

Configuration File Design

The configuration file consists of actions that will determine the scheduling behavior of the growth chamber. Each configuration file is broken up into any number of phases. These are separated with three hyphens:

```
<phase>
---
<phase>
```

Each phase is divided into three sections: START, MONITOR, END. None of the sections require any entries, but they must be declared.

The START section consists of a list of scheduling actions, or “action sequences”, that will be executed at the beginning of the phase.

The END section also consists of an action sequence that is executed upon transition out of that phase.

The MONITOR section will consist of the sensors to monitor during execution, and action sequences corresponding to value ranges a sensor reading might fall into.

Action Sequences

Activate

This action will allow a sensor, actuator, or actuator system to be scheduled.
The syntax is as follows:

```
- - activate
  - <name>
  - <parameter>:<value>
    <parameter>:<value>
    <parameter>:<value>
    ...etc
```

If frequency is defined, the specified device will be repeatedly scheduled based on that frequency. If duration is also specified as a parameter, duration must be less than frequency.

```
- - activate
  - fan
  - frequency: 1 min
    speed: 120 rpm
    duration: 30 sec
```

If an actuator system is being activated, a mode can be specified in the actions parameters.

```
- - activate
  - water_pump
  - duration: 30 sec
    mode: drain
```

Deactivate

A sensor, actuator, or actuator system can be deactivated so it will no longer be scheduled. If an actuator system is being deactivated, it will continue running until completion of its function.

```
- - deactivate
  - fan
```

Set

If a sensor, actuator, or actuator system is activated but parameters need to be changed, the set action can be used.

```
- - set
  - fan
  - speed: 240 rpm
  - duration: 15 sec
```

Transition

To move on to execute the END sequence and move on to the next phase, the transition action can be used.

To transition after whatever is scheduled to execute in that moment, but before the system sleeps again, the “asap” keyword can be used.

```
- - transition
  - asap
```

To transition immediately, the “now” keyword is used.

```
- - transition
```

```
  - now
```

In both cases, if an actuator system is running, it will run to completion before transitioning to the next phase.

Pause

If the system should pause and wait for a resume command from the user, the pause action can be used. The “asap” and “now” keywords can be used, just like with the transition action.

```
- - pause
  - now
```

```
- - pause
  - asap
```

START and END Sections

The syntax for the START and END can be seen in the following example that may represent a seeding phase:

```
START:
    - - activate
      - water_pump
      - mode: fill
      - volume: 50 ml

    - - activate
      - fan
      - speed: 120 rpm

    - - activate
      - seeding_arm
      - mode: plant

    - - transition
      - asap

MONITOR:

END:
    - - deactivate
      - fan
```

MONITOR Section

This section is divided into subsections for each sensor being monitored. For each subsection, there are value ranges that act as conditions for an action sequence corresponding its respective range.

When a sensor reads a value within a specified range, it will execute the scheduling actions specified for that range. To prevent repeated execution, it will not be able execute that action sequence again until another range has been triggered.

```
MONITOR:
    light_sensor:
        [0 lux, 20 lux]:
            - - activate
              - light
              - intensity: 15 lux
        [20 lux, 50 lux]:
            - - deactivate
              - light

    temperature_sensor:
        [0 F, 80 F]:
           - - deactivate
             - fan

        [80 F, inf F]:
           - - activate
             - fan
             - speed: 120 rpm
```

In addition to monitoring sensors, times since the phase began can also be monitored. Instead of a range, a single value is specified. When a time has been reached, it is not triggered again.

```
MONITOR:
    Time:
        [10 sec]:
            - - activate
              - water_pump
              - volume: 50 ml
        [20 sec]:
            - - transition
              - asap
```

Computer Admins - Setting up the Software

Computer Admins are also referred to as System Administrators in this document.
All software can be obtained by contacting us from our emails here:
https://killsap.github.io/automated-multi-cycle-farming-in-space-website/index.html

Installing the Growth Chamber Software

OS Requirements

A Linux operating system with Python 3 installed. This manual will assume the growth chamber microcontroller is a Beagle Bone Black.

Working Directory Requirements

After downloading the python code, the configuration file to be run must be placed in the same directory as driver.py and with the directory structure intact.

Installing Dependencies

Install python dependencies and PostgreSQL.

```
sudo ntpdate pool.ntp.org

sudo apt-get updatels

sudo apt-get install build-essential python-dev python-setuptools python-
pip python-smbus -y

sudo apt-get install postgresql libpq-dev postgresql-client postgresql-
client-common
```

Install necessary python libraries

```
pip install Adafruit_BBIO

pip install coloredlogs
pip install ruamel.yaml

pip install psycopg2-binary

pip install six

pip install pgnotify

pip install adafruit-blinka
```

Setting up the Database

Create the database user and the database, then run the service.

```
sudo service postgresql start
sudo -i -u postgres
createuser amcf -P --interactive
Enter password for new role: dbpassword
Enter it again: dbpassword
createdb amcfdb
exit
```

Initialize the database by running “create_db.py” that comes with the software.

Creating Startup Services

To allow the growth chamber software to run when the microcontroller is powered on, do the following:

Create the file “amcf.service” in some directory:

```
[Unit]
Description=AMCF

[Service]
User=root
TimeoutStartSec=0
WorkingDirectory=/opt/amcf
ExecStart=/opt/amcf/amcf_start.sh
StandardOut=syslog
StandardError=syslog
SyslogIdentifier=AMCF
Restart=never
```

```

[Install]
WantedBy=multi-user.target
```

Create the file “amcf_start.sh” in the same directory, replacing “</path/to/dir>” with the path to the directory containing “driver.py”:

```
#!/bin/bash
python </path/to/dir>/driver.py
```

Then in the same directory those files were created in, run the following commands:

```
sudo mkdir /opt/amcf
sudo cp amcf_start.sh /opt/amcf/amcf_start.sh
sudo chmod +x /opt/amcf/amcf_start.sh
sudo cp amcf.service /etc/systemd/system/amcf.service
sudo systemctl enable amcf.service
sudo systemctl start amcf.service
```

Standard out and standard error will be found in /var/log/syslog. The service will only run once when the microcontroller is powered on, but can be restarted manually with:

```
sudo systemctl start amcf.service
```

Installing the User Station Application

OS Requirements

OS: Windows, Linux
Memory: At least 2GB of Ram
.NetCore installed

Installation Procedure

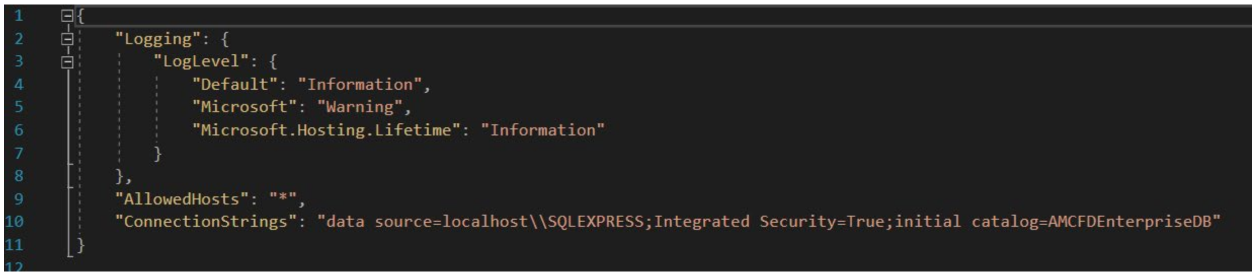
Standing up the application requires standard deployment using a server. If on windows use IIS and set the default app pool (or app pool of your choice) to the root directory of the Farming User Station. Comparable setup on Linux.

Wireless Configuration

Make sure when the static IP address is set on the Farming Chambers that they have a route from the Farming User Station server. Open all necessary ports. Refer to the IP address added in your add device script.

Database

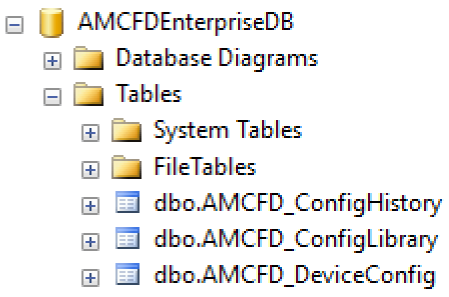
To load the database simply restore the provided .bak file and set the connection string in the appsettings.json file in the root directory of the deployment package.



```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": "data source=localhost\\SQLEXPRESS;Integrated Security=True;initial catalog=AMCFDEnterpriseDB"
}
```

You can specify user groups for LDAP here by using "AllowedClients": {"GROUP NAME": "user1,user2,user3"}

Verify that these tables restore:



```
AMCFDEnterpriseDB
    Database Diagrams
    Tables
        System Tables
        FileTables
        dbo.AMCFD_ConfigHistory
        dbo.AMCFD_ConfigLibrary
        dbo.AMCFD_DeviceConfig
```

Adding Configuration files and Farming Chambers:

To add a directory of configuration files run this script, make sure to change the directory variable string:

```
/******************************
** File: AddConfigs.sql
** Name: Add Configs
** Desc: Load all the config files (.yml) YAML files to the farming user station
** Auth: C. Millsap
** Date: 4/21/20
** Notes: Remember to use the same names for .yml and .png files so the library icons match
**        db is setup to where configfile names must be unique
**************************
** Change History
**************************
** PR   Date        Author  Description
** --   --------    -------  -----------------------------------
** 1    4/21/20     chris    creation
*******************************/
USE [AMCFDEnterpriseDB]
GO

--Get YAML files
--checking for temp table
IF OBJECT_ID('tempdb..#tempList') IS NOT NULL
```

```
BEGIN
   DROP TABLE #tempList
END

CREATE TABLE #tempList (Files VARCHAR(500))
GO

-- Set dir of config files, only .yml file in this dir
Declare @configpath varchar(8000)
set @configpath = 'c:\store' -- leave off last \


Declare @configpath_XP_CMDSHELL varchar(8000)
set @configpath_XP_CMDSHELL = 'dir '+@configpath
INSERT INTO #tempList
--enter path in ' '
EXEC MASTER..XP_CMDSHELL @configpath_XP_CMDSHELL

--delete all directories
DELETE #tempList WHERE Files LIKE '%<dir>%'

--delete all informational messages
DELETE #tempList WHERE Files LIKE ' %'

--delete the null values
DELETE #tempList WHERE Files IS NULL

--get rid of dateinfo
UPDATE #tempList SET files =RIGHT(files,(LEN(files)-20))

--get rid of leading spaces
UPDATE #tempList SET files =LTRIM(files)

-- Add to db
--checking for temp table
IF OBJECT_ID('tempdb..#newTable') IS NOT NULL
BEGIN
   DROP TABLE #newTable
END

--split data into size and filename
SELECT RIGHT(files,LEN(files) -PATINDEX('% %',files)) AS FileName, 0 as Processed into
#newTable  from  #tempList



Declare @Id int

Declare @fn_yamls varchar(MAX)

-- Loop through directory to add files
```

```
While (Select Count(*) From #newTable Where Processed = 0) > 0
Begin

   Select Top 1 @Id = Processed From #newTable Where Processed = 0 -- iterate over

   set @fn_yamls = (Select Top 1 FileName From #newTable Where Processed = 0);

         DECLARE @cmd  nvarchar(MAX)
         Declare @filedata nvarchar(MAX)
         set @cmd = 'DECLARE @FileContents varchar(MAX); SELECT
@filedata=BulkColumn FROM
OPENROWSET(BULK'''+@configpath+'\'+@fn_yamls+''',SINGLE_CLOB) x;';


         declare @params nvarchar(max)
         set @params = N'@filedata varchar(MAX) output'

         execute sp_executesql @cmd, @params, @filedata output;

         --insert into table
         INSERT INTO [dbo].[AMCFD_ConfigLibrary]
                         ([ConfigName]
                         ,[ConfigData]
                         ,[ConfigDateTimeAdded]
                         ,[ConfigDateTimeChanged])
                 VALUES
                         (@fn_yamls,@filedata,SYSDATETIME(),SYSDATETIME());

   Update #newTable Set Processed = 1 Where FileName = @fn_yamls
End

/****** Script for SelectTopNRows command from SSMS to view your additions  ******/
SELECT [ConfigName]
      ,[ConfigData]
      ,[ConfigDateTimeAdded]
      ,[ConfigDateTimeChanged]
  FROM [AMCFDEnterpriseDB].[dbo].[AMCFD_ConfigLibrary]
```

To add a Farming Chamber set the values here:

```
/******************************
** File: AddDevice.sql
** Name: Add Device
** Desc: Add the farming chambers to be used on this deployment
** Auth: C. Millsap
** Date: 4/21/20
** Notes: Device names must be unique
**************************
** Change History
**************************
** PR   Date      Author    Description
** --   --------  -------   ----------------------------------
** 1    4/21/20   C. Millsap      creation
*******************************/

USE [AMCFDEnterpriseDB]
GO

Declare @cmd  nvarchar(MAX)
Declare @filedata nvarchar(MAX)
Declare @configpath varchar(8000)
set @configpath = 'c:\store' -- leave off last \
Declare @yamlname varchar(MAX)
set @yamlname = '003-tomato.yml' -- Enter full name
set @cmd = 'DECLARE @FileContents varchar(MAX); SELECT @filedata=BulkColumn
FROM OPENROWSET(BULK'''+@configpath+'\'+@yamlname+''',SINGLE_CLOB) x;';


declare @params nvarchar(max)
set @params = N'@filedata varchar(MAX) output'

execute sp_executesql @cmd, @params, @filedata output;
INSERT INTO [dbo].[AMCFD_DeviceConfig]
         ([DeviceID]
         ,[LastConfigName]
                  ,[ConfigData]
         ,[IsActive]
         ,[LastRun]
                  ,[IPAddress])
    VALUES                                                                    --
Initial date/time does not need to be changed
         ('Farming Chamber 2', @yamlname, @filedata, 0, '2001-01-01 01:10:10.111',
'192.168.7.3')
GO

/****** Script for SelectTopNRows command from SSMS  to view update******/
SELECT [DeviceID]
     ,[LastConfigName]
           ,[ConfigData]
     ,[IsActive]
     ,[LastRun]
           ,[IPAddress]
  FROM [AMCFDEnterpriseDB].[dbo].[AMCFD_DeviceConfig]
```

Developers - Code Documentation

All software can be obtained by contacting us from our emails here:
https://killsap.github.io/automated-multi-cycle-farming-in-space-website/index.html

Software Design on Growth Chamber

Configuration

The automated scheduler configuration is done via a YAML file that is directly loaded into a python object. See YAML documentation for more information on it's syntax. An example python object loaded from a configuration file can be found in the appendix of this document. While the system is running, it uses this object to determine what actions to take and what conditions to check.

Configuration Validation

The file "config.py" is used to validate the configuration file and in some cases convert values when needed. It will validate the structure of the configuration and check that all sensor and actuator names used in the file are actually defined in their own classes. It will also check that parameters used in actions are defined in their respective class files and that the units used match. There are predefined units attributed to sensors and certain parameters. When a new parameter or sensor is added, the respective units must be added as well. For each type of unit, there is a standard unit used in the internal representation. All units will be converted to match the standard.

Actuators

For actuators that behave like "switches", in that they are turned on and off, the actuator class is inherited. When adding a new actuator a new class is defined for each actuator. The code that does the work may be placed in the activate function or, if it can be scheduled, the optional "run" function may be added to the child class. There also must be code added to the activate and deactivate functions or run function for recording uptime. This is further explained in the API documentation later in this document. The parent classes' functions should be called from the child classes' implementation of those functions. An example class that inherits the actuator class can be found in the appendix of this document.

Actuator Systems

This class is to be inherited when there is an actuator or series of actuators that need to run in parallel to the rest of the system. This is useful for procedures or subsystems. Here there must be a “run” function defined. This is what is called to run on its own thread when the actuator system executes. If the actuator can be run in different modes, the child class should implement a function with the same identifier as the name of the mode. This will behave just like the “run” function.

Additionally, there is an end_run function that should be called at the very end of the “run” function or any mode function. There is also a “reset” function, if any mechanical elements need to be reset. An example class that inherits the actuator system class can be found in the appendix of this document.

Sensors

The sensor class includes functions “activate” and “deactivate” for allowing and disallowing scheduling of the sensor. Additionally the “enable” and “disable” functions are for starting and stopping the sensor. This is like opening and closing the shutter on a camera. The “read” function is for actually pulling the read data out. An example class inheriting the sensor class can be found in the appendix of this document.

Scheduling

The task_manager.py module contains a list of tuples that represents the schedule. These tuples consist of an operation and then a parameter. There are never two identical non-sleep tuples in the schedule.

Operation	**Parameter**	**Description**
SLEEP	Time	Let the system sleep
RUN	Actuator/Actuator System name	Call the run function of the actuator
ENABLE	Sensor name	Open the sensor’s “shutter”
DISABLE	Sensor name	Close the sensor’s “shutter”
READ	Sensor name	Pull the last read value from the sensor
TRANSITION	n/a	Transition to next phase

When a tuple is pulled from the schedule and must be rescheduled, it will reinsert it based on the frequency of the operation’s target.

Event Handler

External control of the growth chamber software, there is a module “event_handler.py” that runs on it’s own thread. It is notified whenever there is a change to the command table inside of the database. Based on the command inserted, it sets flags to signal the growth chamber software to either pause, resume, shutdown, or restart. This allows the user station application to implement manual control by interacting with the command table.

Database Design

System Status

A table that shows whether the growth chamber software is currently running.

```
CREATE table system_status (
    id INT PRIMARY KEY,
    is_running BOOLEAN NOT NULL
)
```

Subsystem Status

This table shows whether or not an actuator system is currently running.

```
CREATE TABLE subsystem_status (
    name VARCHAR(256) PRIMARY KEY,
    status INT NOT NULL)
```

Phase Metrics

This table includes the cycle set (which multicycle run an entry belongs to), the phase, and the phase’s runtime. The config being run is also included.

```
CREATE table phase_metrics (
    id SERIAL PRIMARY KEY,
    cycle_set INT NOT NULL,
    phase INT NOT NULL, phase_runtime REAL NOT NULL,
    config_name VARCHAR(256) NOT NULL
)
```

Device Metrics

This table contains the uptime and name of a sensor, actuator, or actuator system for a phase.

```
CREATE table device_metrics (
    id SERIAL PRIMARY KEY,
    metric_id INT REFERENCES phase_metrics(id),
    uptime REAL NOT NULL,
    name VARCHAR(256) NOT NULL
)
```

Commands

This table contains commands for the growth chamber software and when they were sent. The type of command is an enum. Pause is 0, resume is 1, restart is 2, shutdown is 3, start is 4.

```
CREATE table commands (
    id SERIAL PRIMARY KEY,
    type INT NOT NULL,
    recieved_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    config_name VARCHAR(256)
)
```

Command Function and Trigger

The trigger and function to notify the event handler that a command has been inserted is as follows:

```
CREATE FUNCTION notify_cmd_trigger() RETURNS trigger AS $$
BEGIN PERFORM pg_notify(
       'new_cmd'::text,
       CONCAT(NEW.id::text, CONCAT(' ', NEW.type::text))
    );
    RETURN new;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER new_command
    AFTER INSERT ON commands
    FOR EACH ROW
    EXECUTE PROCEDURE notify_cmd_trigger()
```

Actuator Class

Variables	**Description**
_logger	The logger object for the actuator class.
is_active	Boolean value that flags whether scheduling is allowed.

phase_runtime	The runtime for the current phase.
start_time	When the actuator started. For calculating runtime deltas.
name	The name of the actuator

init(self, name)

Initializes the actuator, the logger for the actuator, and sets is_active to false. Should be overridden to set child's default settings then called from child's implementation.

Parameters:

Name - the name of the actuator being initialized

activate(self)

Sets is_active to true. Allows this actuator to be scheduled. Should be overridden with actuator settings as arguments and called from child's implementation. If the child class does not have a run function and behaves like a "switch", operational code for the actuator to be "turned on" should be written in the child's implementation of this function.

deactivate(self)

Sets is_active to false. Disallows this actuator from being scheduled. If the child class behaves like a "switch", operational code for the actuator to be "turned off" should be written in the child's implementation of this function.

set_val(self)

Sets the settings of the actuator with the parameters passed to it. Should be overridden by the child class.

Actuator System Class (extends Actuator)

Variable	**Description**
running	A boolean value that shows whether the actuator is currently running.
is_running_lock	A semaphore lock for modifying the running variable.
reset_flag	A synchronized event object signaling a reset.
unpause_flag	A synchronized event object signaling an

	unpause.
mode	The name of the function to run instead of the “run” function.

reset(self)

This abstract function calls the end_tun function. Classes implementing this function should perform any necessary resetting operations and then at the end call the end_run function or the parent classes’ reset function.

run(self)

This abstract function sets the appropriate flags and sets the start time. Classes implementing this function, or any “mode” function, should call the parent class run function and then do all the work of the actuator in this function. Inside the operational loop, the implemented function should check the reset flag and unpause flag and behave appropriately. See the example actuator class in the appendix.

end_run(self)

This abstract function sets the appropriate running flags and calculates runtime and adds it to the phase_runtime. Classes implementing this function should do any work and then call the parent class end_run function.

is_running(self)

This function simply returns whether this actuator system is currently running.

Returns:
    is_running

Sensor Class

Variables	**Description**
_logger	The logger for the sensor.
is_active	Boolean flag to indicate whether the sensor may be scheduled.
is_enabled	Boolean flag to indicate whether the sensor is enabled and capturing data.

read_time	The time between enable and disable used for the sensor to capture data.
start_time	The time the sensor was activated for the current phase.
phase_runtime	Time spent reading for the current phase.
name	The name of the sensor.
frequency	How frequent the sensor should be scheduled to read.

activate(self, frequency, read_time)

Sets frequency, read_time, and is_active. Allows this sensor to be scheduled. Should be overridden with sensor settings as arguments and called from child's implementation.

Parameters:

frequency - How frequently should the sensor be scheduled.
read_time - How long the sensor should capture data for.

deactivate(self)

Sets is_active to false. Disallows this actuator from being scheduled.

enable(self)

Sets is_enabled to true. Any class implementing this function should call the parent class enable function and then run any code to begin capturing data.

disable(self)

Sets is_enabled to false. Any class implementing this function should run code to stop capturing data and then call the parent class disable function.

read(self)

Any class implementing this function should return the value of the captured data.

set_val(self)

Classes implementing this function should use it to set the settings for the sensor.

Config Module

Variables	Description
read_units	Valid units that can be read from a sensor.
op_names	Valid actions that can be listed in a configuration file's action sequences.
param_units	A dictionary with parameter types as keys corresponding to lists of valid units for those parameters.

validate_value(valid_units, value_str, logger)

Checks to see if a string represents a valid value and checks to see if the unit used is listed in valid_units.

Parameters:

valid_units - The list of units to check against.
value_str - The string to check against.
logger - The config module's logger.

Returns:

true - If the string is valid.
false - If the string is not valid.

convert_value(value_str)

Converts a string to a standard value to be used by the software.

Parameters:

value_str - either a number or a two token string representing a value with a unit

Returns:

The value that is being represented.

validate _cmd_list(cmd_list, devices, logger)

Iterates through an action sequence and validates each action by checking the target of the action and the parameters, converting values if necessary.

Parameters:

cmd_list - The list of actions tuples containing action type, target, and parameters
devices - A dictionary of sensors and actuators.

Returns:

True - If valid.
False - If not valid.

validate _range_dict(sensor_name, devices, range_dict, new_dict, logger)

Iterates through the monitor ranges for a particular sensor, validates their values, and then validates their corresponding action sequences. It then inserts the action sequences into a new range dict that uses converted range values instead of strings.

Parameters:

sensor_name - The sensor corresponding to the range dictionary being validated.
devices - A dictionary of sensors and actuators.
range_dict - The range dictionary being validated.
new_dict - The new dictionary that will use the converted value tuples as keys.
logger - The logger for the config module.

Returns:

True - If valid.
False - If not valid.

validate_config(data, sensors, actuators)

Validates the syntax and organization of the configuration data for a phase that was parsed from the configuration file. It validates the outer structure of the configuration data before validating the START action sequence. It then iterates through the MONITOR sequence checking if the sensors to be monitored exist and then validates their monitor ranges. Finally, it validates the END action sequence.
Parameters:

data - The parsed configuration file phase data.
sensors - The sensors being used in the system.
actuators - The actuators being used in the system.

Returns:

True - If valid.
False - if invalid.

load_config(path, driver)

Parses the configuration file into a python object. It then iterates through the phases in that object and validates them.

Parameters:

path - The path to the configuration file on the machine.
driver - A reference to the driver object.

Returns:

None - If the configuration data is invalid.
sys_conf - The validated configuration data object.

Driver Module

Variable	**Description**
_logger	The logger for the driver class.
eh	The event handler object.
tm	The task manager object.
sys_config	The configuration object representing the configuration file.
sleep_condition	A condition object semaphore used for accessing the is_sleeping variable.
is_sleeping	A boolean flag used to indicate that the system is sleeping.
phase_config	The current phase from sys_config.
read_ranges	A dictionary containing the previously read ranges from sensors with sensor names as keys.
actuators	A dictionary containing the actuator objects with the actuator names as keys.
sensors	A dictionary containing the sensor objects with the sensor names as keys.
threads	A dictionary containing actuator systems with the thread objects as keys.

cycle_set	The current cycle set. Increments every time the system starts or restarts.
phase_start_time	The start time for the current phase.
config_name	The name of the config file currently being used.

init(self)

Sets sets the running flag in the database, initializes the sensors, actuators, and task manager. It then loads the configuration data.

set_event_handler(self, handler)

Sets the drivers event handler.

Parameters:
    handler - The event handler.

initialize_devices(self, dir_path, device_dict)

Iterates through the actuator and sensor directories and initializes the actuator and sensor classes inside them.

take_nap(self, secs)

Puts the system to sleep. It waits on sleep_condition for “secs” number of seconds. It also records the actual sleep time in case the main thread is externally woken by the event handler.
Parameters:
    secs - Length of time the system is scheduled to sleep for.

Returns:
    Scheduled sleep time remaining.

resume_actuators(self)

Iterates through the actuators in the threads dictionary and sets their unpause_flag.

flag_actuator_reset(self)

Iterates through the actuators in the threads dictionary and sets their reset_flag.

pause_actuators(self)

Iterates through the actuators in the threads dictionary and clears their unpause_flag.

_process_cmd(self, op, param)

Processes a command from the scheduler.

On ENABLE, a READ operation is scheduled for that sensor's read time, then calls the sensor's enable function.

On READ, the sensor's read function is called and checked against its monitor ranges. If the value falls in one of the ranges, the action sequence corresponding to that range is scheduled.

On RUN, if the target is just an actuator, its run function is called. Otherwise, in the case of an actuator system, a thread is created from its run function or the appropriate mode function. That thread is then added to the threads dictionary.

On SLEEP, the system sleeps for the time specified. The function accounts for a possible Time section in the MONITOR section and possible pauses from the event handler.

On TRANSITION, the task manager's schedule is cleared and the END action sequence is scheduled.

On PAUSE, the function checks the event handlers restart_flag and its exit_flag. If neither are set, the event handler's pause_system function is called.

After the command has been processed, the function checks to see if the system has been paused and if so, it tries to pause any actuator systems running before waiting on the pause_condition.

Parameters:

  op - The enum corresponding to the operation to perform.
  param - The parameter corresponding to the operation.

run_phase(self, phase_num)

Schedules the START sequence. While there are commands in the schedule, it fetches them then executes them. If the config data indicates time must be monitored the phase runtime is calculated and if it matches one of the monitor action sequences, that sequence is scheduled.

After the command is processed, the function checks for dead threads and cleans them up, before joining any completed threads.

Once the schedule for the phase is empty, any currently enabled sensors are disabled and the phase/device metrics are updated in the database.

Parameters:
phase_num - The index of the current phase being run.

run_system(self)

Loops through each phase in the configuration data and runs it. If the restart flag is set, the configuration file is reloaded, cycle_set is incremented, and the number of cycles is reset before continuing the loop. If the exit_flag is set, the function joins the event handler thread, sets the shutdown flag and breaks the loop.

Once the loop is complete the function joins the event handler thread, sets the shutdown flag and updates the system status in the database.

service _shutdown(self, signum, frame)

This function is called if an interrupt is thrown. It updates the system_status in the database and raises a ServiceExit exception.

Parameters:
signum - The type of signal caught.

exec_sql(sql, willReturn, *args)

This function executes an SQL statement.

Parameters:
sql - A string with formaters that represents the sql statement to execute.
willReturn - A boolean value that tells whether the id of the row should be returned.
*args - A variable number of arguments to format the sql statement with.

Returns:
ident - The id of the row fetched, if the statement was a query.
-1 - If willReturn is false

main()

Initializes and starts the event handler, then waits for the start_flag to be set. Once it is set, a driver is initialized and calls its run_system function.

Event Handler Module (extends Thread)

Variables	**Description**
_logger	The logger for the event handler.
pause_condition	A condition semaphore for access to the paused variable.
paused	A boolean flag to signal the driver to pause.
restart_phase	A synchronized event object to signal the driver to restart.
exit_flag	A synchronized event object to signal the driver to begin the exit process.
start_flag	A synchronized event object to signal the driver to start.
driver	A reference to the driver object.

init(self)

Initializes all the events for flags and the conditions.

set_driver(self, driver)

Sets the reference to the driver for the event handler.

Parameters:

driver - The reference to the driver object.

run(self)

This function monitors the commands table of the database for new insertions. If there is an insertion it calls the appropriate function based on the command inserted. The type of command is an enum. Pause is 0, resume is 1, restart is 2, exit is 3, and start is 4.

wake_system(self)

Acquires the drivers sleep_condition and notifies any threads waiting on it if the driver is sleeping. It then releases the condition.

pause_system(self)

Acquires pause_condition before setting paused to true. It then releases the condition before calling wake_system.

resume_system(self)

Acquires pause_condition before setting paused to false. It then nofies any thread waiting on the pause condition before releasing it.

restart_system(self)

Wakes the system then sets restart_flag.

exit_system(self)

Wakes the system then sets the exit_flag.

Task Manager Module

Variables	**Description**
schedule	A list of operation and parameter tuples representing the schedule.
_actuators	A dictionary of actuators with their names as keys.
_sensors	A dictionary of sensors with their names as keys
_logger	The logger for the task manager class.

schedule_cmds(self, cmd_list)

Iterates through an action sequence scheduling actions based on the targets frequency or delay, if provided. Sleep operations are used between actions.

Parameters:
    cmd_list - the action sequence to schedule.

insert(self, new_cmd, wait_time)

Inserts a new command into the schedule. The schedule is iterated through and non-sleep operations are skipped. Sleep times are subtracted from wait time. If a sleep time is greater than the remaining wait time, that sleep operation is split and the command is inserted between the two new sleeps. If the end of the list has been reached place a sleep command with the remaining wait time and then place the new command.

Parameters:

new_cmd - The command to be inserted.
wait_time - The amount of time before the command should be executed.

remove(self, target_cmd)

Finds the specified command in the schedule and removes it. Merging sleep times if necessary.

Parameters:

target_cmd - The command to remove.

time_until(self, target_cmd)

Finds the amount of time until the specified command.

Parameters:

target_cmd - The command to calculate time until.

Returns:

time - The amount of time until that command
inf - Infinity, if the command is not in the schedule.

get_cmd(self)

Pops a command from the schedule.
If the command is READ, an ENABLE command is rescheduled with a wait time based on the sensor's frequency.

If the command is RUN and the commands target has a frequency parameter, the RUN command is rescheduled.

Returns:

The command popped from the front of the list.

Software Farming User Station

The GUI for the Farming Chamber is built in. NetCore and utilizes the MVC structure for the code.
All software and static files can be obtained by contacting us on our website:
https://killsap.github.io/automated-multi-cycle-farming-in-space-website/index.html

Frameworks used: bootstrap.bundle.min.js, bootstrap-grid.cs, jquery.min.js

Classes

ConfigFile

This class is the model representation of the config files. When a configuration file is deserialized it is parsed into this structure.

ReportData

This class is the model view for the report metrics pulled from a selected Farming Chamber

Controllers

Controllers are the classes which contain the functions that operate directly with the views

EditDevice

This is the controller for any page where a config file is edited. The functions here connect to the farming chamber over Wi-Fi

Home

This controller is where the dashboard is populated, and selections made to the menu are handled.

Views

AddDevice

Currently not implemented, this feature can be added so that non-technical users can updated the Farming Chamber list.

Config

This is the view for displaying the configuration library. The library is built dynamically at runtime so any updates will be show on refresh.

ConfigHistory

This is the view for the config file history/version control for any Farming Chamber. The list is built in JavaScript and can be found in tablelist.js under the js folder in src.

Connect

Not implemented but this is where the non-technical users can connect their farming chambers and verify connections settings.

Edit

This view builds the config file selected to be edited dynamically. Each deserialized config file part is traversed and populated to this page. Only the numerical fields are available to the astronaut to edit.

Index

This is the main view where the dashboard and popups are displayed. Not to be confused with layout where the menu is populated to run over other views.

Reports

This view fully renders when a Farming Chamber is selected. The model for this view connects to the database.

SharedViews

Layout is the only shared view. This is where the main menu is created and given to the other views, so it does not need to be rendered on each page.

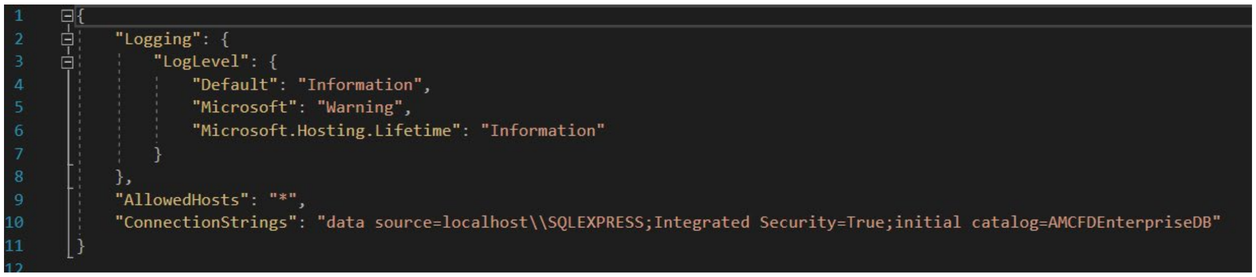
Settings

appsettings.json

This is where the system computer administrator can add a database connection string and restrict user access vie LDAP.

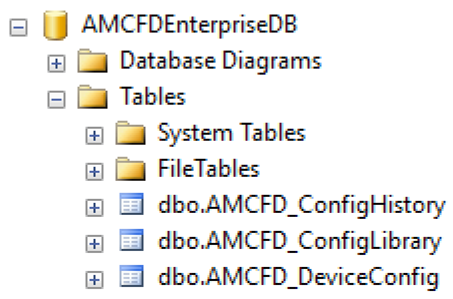
Database

To load the database simply restore the provided .bak file and set the connection string in the appsettings.json file in the root directory of the deployment package.

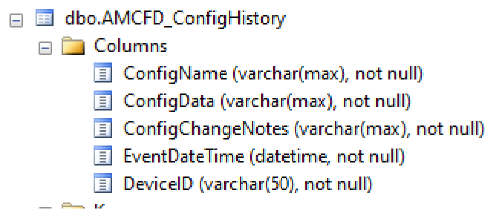


```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*",
    "ConnectionStrings": "data source=localhost\\SQLEXPRESS;Integrated Security=True;initial catalog=AMCFDEnterpriseDB"
}
```

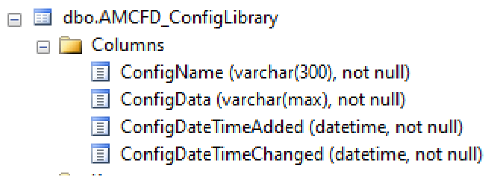
You can specify user groups for LDAP here by using “AllowedClients”: {“GROUP NAME”: “user1,user2,user3”}
Verify that these tables restore:



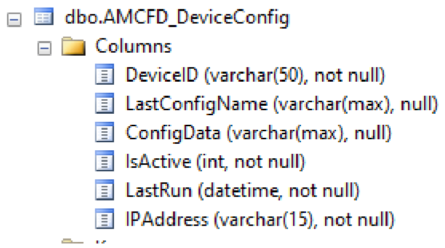
```
AMCFDEnterpriseDB
    Database Diagrams
    Tables
        System Tables
        FileTables
        dbo.AMCFD_ConfigHistory
        dbo.AMCFD_ConfigLibrary
        dbo.AMCFD_DeviceConfig
```



```
dbo.AMCFD_ConfigHistory
    Columns
        ConfigName (varchar(max), not null)
        ConfigData (varchar(max), not null)
        ConfigChangeNotes (varchar(max), not null)
        EventDateTime (datetime, not null)
        DeviceID (varchar(50), not null)
```

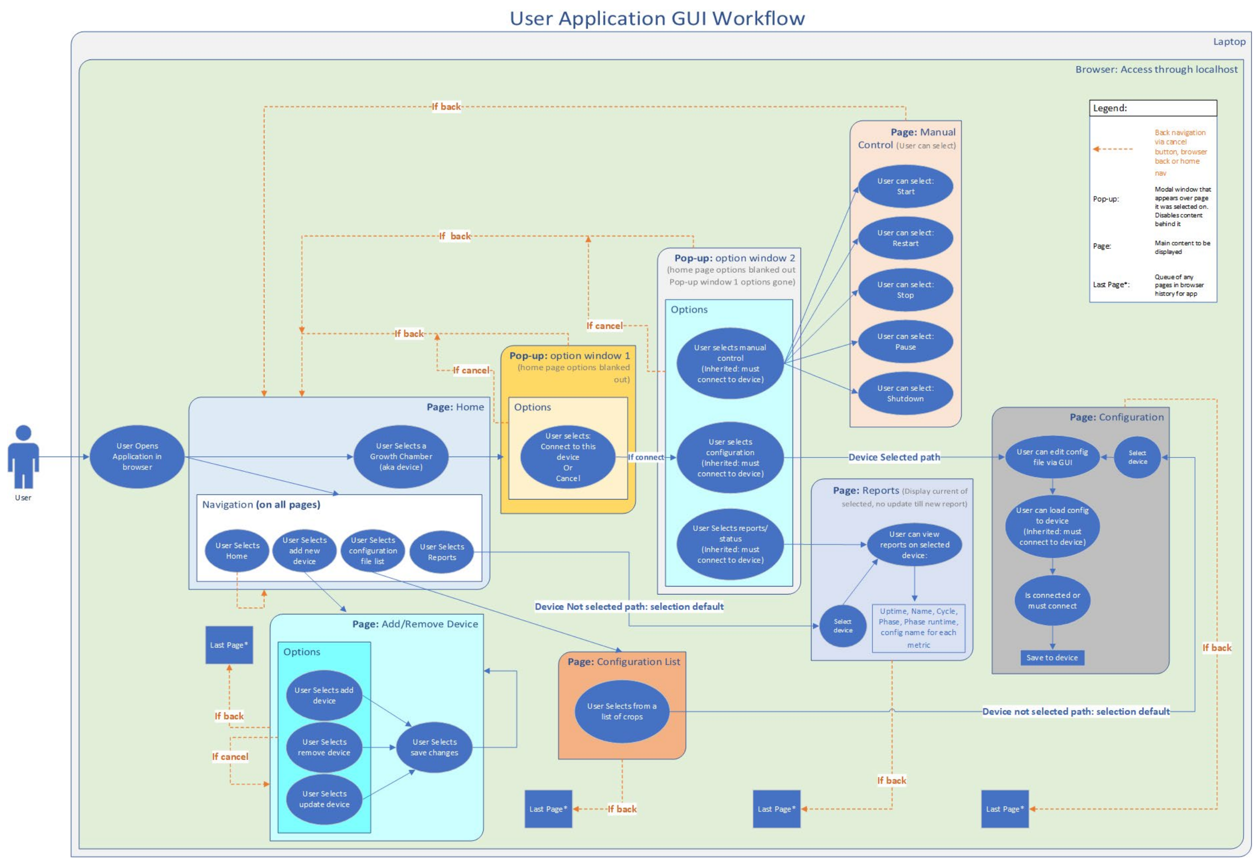


```
dbo.AMCFD_ConfigLibrary
    Columns
        ConfigName (varchar(300), not null)
        ConfigData (varchar(max), not null)
        ConfigDateTimeAdded (datetime, not null)
        ConfigDateTimeChanged (datetime, not null)
```



```
dbo.AMCFD_DeviceConfig
    Columns
        DeviceID (varchar(50), not null)
        LastConfigName (varchar(max), null)
        ConfigData (varchar(max), null)
        IsActive (int, not null)
        LastRun (datetime, not null)
        IPAddress (varchar(15), not null)
```

Full diagram of application flow:


User Application GUI Workflow
Laptop
Browser: Access through localhost
Legend:
Back navigation via cancel button, browser back or home nav
Pop-up:
Modal window that appears over page it was selected on. Disables content behind it
Page:
Main content to be displayed
Last Page*:
Queue of any pages in browser history for app
User
User Opens Application in browser
Page: Home
User Selects a Growth Chamber (aka device)
Navigation (on all pages)
User Selects Home
User Selects add new device
User Selects configuration file list
User Selects Reports
Pop-up: option window 1 (home page options blanked out)
Options
User selects: Connect to this device Or Cancel
If connect
If cancel
If back
Pop-up: option window 2 (home page options blanked out Pop-up window 1 options gone)
Options
User selects manual control (Inherited: must connect to device)
User selects configuration (Inherited: must connect to device)
User Selects reports/ status (Inherited: must connect to device)
Page: Manual Control (User can select)
User can select: Start
User can select: Restart
User can select: Stop
User can select: Pause
User can select: Shutdown
Device Selected path
Page: Configuration
User can edit config file via GUI
Select device
User can load config to device (Inherited: must connect to device)
Is connected or must connect
Save to device
Page: Reports (Display current of selected, no update till new report)
User can view reports on selected device:
Select device
Uptime, Name, Cycle, Phase, Phase runtime, config name for each metric
Device Not selected path: selection default
Page: Add/Remove Device
Options
User Selects add device
User Selects remove device
User Selects update device
User Selects save changes
Last Page*
If back
If cancel
Page: Configuration List
User Selects from a list of crops
Device not selected path: selection default
Last Page*
If back

Appendix

Configuration Example

```
START: # Action Sequence
  - - activate          # Operation
      - Light           # Target
      - intensity: 5.3 lux  # Param: Value

  - - activate
      - MoistureSensor
      - frequency: 5.2 sec

  - - activate
      - HeightSensor
      - frequency: 10 sec
```

```
MONITOR:
  Time:
      [10 sec]:
      - - deactivate
      - Light

      - - pause
      - now

      [30 sec]:
      - - transition
      - asap

  HeightSensor:
      [0 cm, 1 cm]: # First element is inclusive second is exclusive
      - - set
      - HeightSensor
      - frequency: 10 sec

      [1 cm, 2 cm]:
      - - set
      - HeightSensor
      - frequency: 5 sec

      [2 cm, inf cm]:
      - - transition
      - asap

  MoistureSensor:
      [0 hum, 2 hum]:
      - - activate
      - WaterPump
      - frequency: 5 sec

      - - set
      - MoistureSensor
      - frequency: 1 sec

      [2 hum, 3 hum]:
      - - set
      - MoistureSensor
      - frequency: 1 sec

      - - deactivate
      - WaterPump

END:
  - - deactivate
      - Light

  - - deactivate
      - MoistureSensor

  - - deactivate
      - HeightSensor

  - - deactivate
      - WaterPump
---
START: # Action Sequence
  - - activate          # Operation
      - Light           # Target
      - intensity: 5.4567 lux  # Param: Value

  - - activate
      - WaterPump
      - mode: pump

  - - transition
      - asap

MONITOR:

END:
  - - deactivate
      - Light
```

Code Examples

Sensor Example

```
from .sensor import Sensor
import time

class MoistureSensor(Sensor):
      sim_farm = None

      def __init__(self, name, sim_farm = None):
      Sensor.__init__(self, name)
      self.sim_farm = sim_farm

      def activate(self, frequency = 0, read_time = 1):
      super().activate(frequency, read_time)
      self._logger.info("Activated MoistureSensor with frequency
{}".format(self.frequency))

      # Sets is_active to false
      def deactivate(self):
      super().deactivate()
      self._logger.info("Deactivated MoistureSensor")

      def enable(self):
      super().enable()
      self.start_time = time.time()
      # Put hardware code here...
      self._logger.info("MoistureSensor enabled")

      def read(self):
      val = None

      # Remove this on hardware implementation
      if self.sim_farm:
            val = self.sim_farm.get_state_moisture_sensor()

      self._logger.info("Read {} from MoistureSensor".format(val))
      return val

      def disable(self):
      if self.start_time != None:
            self.phase_runtime += time.time() - self.start_time
      super().disable()
      # Put hardware code here...
      self._logger.info("MoistureSensor disabled")

      # Sets a variable number of this sensors params
      def set_val(self, frequency = None):
      if frequency:
            self.frequency = frequency
            self._logger.info("Set MoistureSensor frequency to
{}".format(frequency))
```

Actuator Example

Uses a simulation module to simulate hardware code.

```
from .actuator import Actuator
import math
import time

class Light(Actuator):
      intensity = 0
      sim_farm = None

      def __init__(self, name, sim_farm = None):
      Actuator.__init__(self, name)
      self.sim_farm = sim_farm

      def activate(self, intensity = 1):
      super().activate()
      if self.is_active:
            self.start_time = time.time()
      self.intensity = intensity
      self._logger.info("Activated Light with intensity
{}".format(self.intensity))
      self.sim_farm.set_state_light(True, intensity) # Replace with
hardware code

      def deactivate(self):
      if self.is_active and self.start_time != None:
            self.phase_runtime += time.time() - self.start_time
      super().deactivate()
      self.start_time = None
      self._logger.info("Deactivated Light")
      self.sim_farm.set_state_light(False) # Replace with hardware code

      def set_val(self, intensity = None):
      if intensity:
            self.intensity = intensity
            self._logger.info("Set Light intensity to
{}".format(self.intensity))

            if self.is_active:
                  self.sim_farm.set_state_light(True, intensity) # Replace
with hardware code
```

Actuator System Example

Uses a simulation module to simulate hardware code.

```
from .actuator_system import ActuatorSystem
import math
import time

class WaterPump(ActuatorSystem):
      duration = 0
      frequency = 0
      _sim_farm = None

      def __init__(self, name, sim_farm):
      ActuatorSystem.__init__(self, name)
      self._sim_farm = sim_farm

      def activate(self, frequency = 0, duration = 0, mode = "run"):
      super().activate()
      self.frequency = frequency
      self.duration = duration
      self.mode = mode
      self._logger.info("Activated WaterPump with frequency {}, duration
{}, mode {}".format(self.frequency, self.duration, self.mode))

      def deactivate(self):
      super().deactivate()
      self._logger.info("Deactivated WaterPump")
      def set_val(self, frequency = None, duration = None):
      if frequency:
            self.frequency = frequency

      if duration:
            self.duration = self.duration

      self._logger.info("Set WaterPump frequency to {} and duration to
{}".format(self.frequency, self.duration))

      def run(self):
      self._logger.info("Running WaterPump with duration
{}".format(self.duration))
      super().run() # Required


      self._sim_farm.set_state_water_pump() # Simulate operating loop,
remove with hardware implementation

      time_left = self.duration
      while True:
            # Put hardware code here
            # Check if has been paused
            if not self.unpause_flag.isSet():
                  self._logger.info("Pausing waterpump with remaining
duration {}".format(time_left))
                  self.unpause_flag.wait()
                  self._logger.info("Resuming waterpump with remaining
duration {}".format(time_left))
            if self.reset_flag.isSet():
                  self._logger.debug("Resetting waterpump with remaining
duration {}".format(time_left))
                  self.reset()
                  return
            time.sleep(0.05)
            time_left = time_left - 0.05
            if time_left <= 0:
                  break

      self._sim_farm.set_state_water_pump()
      self.end_run() # Required
      def pump(self):
      self._logger.info("Running WaterPump with duration {} and mode
{}".format(self.duration, self.mode))
      super().run() # Required

      # Simulate operating loop
      self._sim_farm.set_state_water_pump()
      time_left = self.duration
      while True:
            # Put hardware code here...
            # Check if has been paused
            if not self.unpause_flag.isSet():
                  self._logger.info("Pausing waterpump with remaining
duration {}".format(time_left))
                  self.unpause_flag.wait()
                  self._logger.info("Resuming waterpump with remaining
duration {}".format(time_left))
            if self.reset_flag.isSet():
                  self._logger.debug("Reseting waterpump with remaining
duration {}".format(time_left))
                  self.reset()
                  return
            time.sleep(0.05)
            time_left = time_left - 0.05
            if time_left <= 0:
                  break

      self._sim_farm.set_state_water_pump()
      self.end_run() # Required

      def end_run(self):
      self._logger.info("Finished WaterPump with duration
{}".format(self.duration))
      # Put hardware code here if needed
      super().end_run() # call after

      def reset(self):
      self._logger.info("Resetting water pump")
      # Put hardware code here if needed
      super().reset() # call after
```

Example Internal Data Structure for Config File Phase

```
{'START':
      [['activate', 'light', {'intensity':'5 lux', 'duration':'14 days'}],
      ['activate', 'moistureSensor', {'frequency': '6 hrs'}],
      ['activate', 'heightSensor', {'frequency': '3 days'}]],

 'MONITOR': {
      'heightSensor': {
            ('1 cm','2 cm'):
                  [['set', 'heightSensor', {'frequency': '2 days'}]],

            ('2 cm', '3 cm'):
                  [['set', 'heightSensor', {'frequency': '1 days'}]],

            ('3 cm', 'inf cm'):
                  [['transition']]
      },

      'moistureSensor': {
            ('0 hum', '2 hum'):
                  [['activate', 'waterPump', {'duration': '10 sec'}],
                  ['set', 'moistureSensor', {'frequency': '6 hrs'}]],

            ('2 hum', '3 hum'):
                  [['set', 'moistureSensor', {'frequency': '1 days'}]],

            ('3 hum', 'inf hum'):
                  [['transition']]
      }
  },

 'END':
      [[{'deactivate': 'light'}, None],
      [{'deactivate': 'moistureSensor'}, None],
      [{'deactivate': 'heightSensor'}, None]]
}
```