# LightingFDDAgent – A Lighting FDD Volttron Agent

## Purpose

This agent provides a wrapper for the Lighting FDD (Fault Detection and Diagnosis) standalone application (lighting\_fdd) for integration with the Volttron system. It defines an agent class (LightingFddAgent) that acts as the point of interaction between the lighting\_fdd module and the Volttron message bus.

It defines a topic that other agents can send messages to, a format for data that a request must contain, and the corresponding format and topic for responding to any requests.

The standard use case is to create one LightingFDDAgent that will listen to all requests for lighting fault detection. When a message is received on one of the two analysis request topics the agent will perform the requested analysis and post the response to the corresponding response topic.

As a wrapper this document only contains some information about the wrapped lighting\_fdd module. This document should be enough to work with the agents but for more details about the analysis being performed consult the documentation for the lighting\_fdd module itself.

## System Requirements and Dependencies

Requires Python 2 (tested on 2.7.5)

Requires the Volttron environment: https://github.com/VOLTTRON/volttron

Requires the lighting\_fdd module. Currently this is located at <https://bitbucket.org/berkeleylab/eetd-tn-lighting/src> under the Modules folder but it is planned to be hosted in PyPi under the name lighting\_fdd.

The agent code itself introduces no additional dependencies outside of those required to run the Volttron system or the lighting\_fdd module

## Additional Considerations and Concepts

This program acts as a wrapper around the lighting\_fdd module and defines an interface for making requests for lighting FDD. It provides all Volttron-related logic for processing requests for these calculations and posting responses.

**Timeseries**: Most of the data that this application uses is in the form of a timeseries. A timeseries is a list of (timestamp, value) pairs. There is one notable exception: the weekly lighting schedule.

**Schedule**: This is the representation of when the lights are expected to be on and off. Diagnosis is made by first finding points where the system does not match the schedule and then attempting to provide context. Currently the only supported schedule type is a weekly schedule. In this schedule, unlike the timeseries, only the day of the week (Monday, Tuesday, etc…) and time of day (12:00:05) are used.

## Installation Instructions

The lighting\_fdd module must be visible to the wrapper. Volttron must be installed and running. See <https://github.com/VOLTTRON/volttron/wiki> for instructions on configuring/running Volttron and general agent usage.

If this package is installed with the rest of the Volttron package it should automatically download and install any dependencies.

If the module needs to be installed manually first download and extract the package. Open a terminal and navigate to the root of the extracted archive. There should be a file called setup.py. Run “python setup.py install”. As long as there is a connection to the internet python should install the package and all dependencies.

## Execution Instructions

See <https://github.com/VOLTTRON/volttron/wiki> for instructions on running Volttron and agents.

Topics this agent subscribes to:

* “lighting\_fdd/request”

FDD request:

* Header fields:
  + Required:
    - **requestorID**: The ID of the requesting agent. Needed to craft the response topic
  + Optional:
    - **From**: Will be used to set the “To” field in the response headers if present
* Message fields:
  + Required:
    - **implemented\_schedule**: The schedule the space should be following. Any faults will be with respect with this schedule.
    - **relay\_timeseries**: A timeseries indicating the state of the lights in the zone.
    - **override\_timeseries**: A timeseries representing the occurrences of manual overrides.
  + Optional:
    - **timezone**: The timezone of the data. If none is passed the default system timezone will be used.
    - **load\_timeseries**: If there is available load data the program can make use of it to see if the change in load matches the change in relay state further refining any diagnostic messages.
    - **expected\_load\_min\_change** (default None): If there is load it is likely the case that the load is not purely for the lighting systems or that the measurements do not go to 0 when the relay is off. This value (and the following expected\_load \_max\_change) value allow for an expected load change range. If the load of the system is roughly known then it can be entered to increase the accuracy of diagnostics made from the load\_timeseries.
    - **expected\_load\_max\_change** (default None): See expected\_load\_min\_change
    - **relay\_time\_comparison\_epsilon\_seconds**: (5 minutes): It is often the case that some of the streams get slightly out of sync or have some minor time drift. This value allows for defining a window around scheduled points so that the system change does not have to fall exactly on the scheduled time. If the system exits the window around the schedule point in the correct state but there was some point inside the window that is in the incorrect state those points are ignored.

For example assume the lights are scheduled to turn off at 18:00:00, the relay\_time\_comparison\_epsilon is 5 minutes, and the relay timeseries looks like

[(17:59:00, 1), (18:00:00, 1), (18:01:00, 1), (18:02:00, 1), (18:03:00, 0), … (18:06:00, 0)]

Then in that case there would be no reported fault. If there were no relay\_time\_comparison\_epsilon or if it were set to 1 minute there would be a fault because the relay did not turn off until 18:02 which is 2 minutes after the scheduled time.

* + - **intended\_schedule**: It may be the case that the implemented schedule is read from the system while there is some other representation of what the intended schedule is (a design document or the like). In case they differ in a meaningful way the application can provide information about the difference as well as locate faults with respect to the intended schedule as well as the one that is actually implemented. This can be useful for distinguishing between hardware faults and system misconfigurations.
    - **override\_timeout\_seconds**: The programmed length of the override timeout. The assumption is that once an override has occurred the system will remain in the override state (absent any other factors) for the length of the override\_timeout.
    - **compress\_overrides\_to\_change\_of\_state** (default True): In our system the override data stream was polled rather than interrupt-driven so it was necessary to take the data from that stream and only use the change-of-value points. However other systems that may have interrupt-driven overrides. To accommodate this set this value to False. Then each point in the override stream will be treated as a separate override event.
* Response Topic: “lighting\_fdd/response/{requester\_ID}”
  + requestor\_ID is the value in the initial request’s “requestorID” header field.
* Response Headers:
  + **To**: The value in the “From” header field from the initial request if it exists, “Unknown” otherwise.
  + **From** : This will always be “lightingAnalysisAgent”
* Response Message: A dictionary with the following fields. If there are any faults or suggested changes those fields will have values. If there are no faults or suggested changes the values will be None.
  + faults\_with\_indended\_schedule
  + faults\_with\_intended\_schedule
  + suggested\_schedule\_changes
  + suggested\_override\_timeout\_changes.

### Input Formats

* **Timeseries**: A list of (timestamp, value) pairs. See each individual type below for what the values represent. All timestamps used in a timeseries **except** those used in the weekly schedule can be expressed in any of the following forms:
  + Seconds since Jan 1, 1970
  + Milliseconds since Jan 1, 1970
  + A string that is parsable by the dateutil.parser.parse function ([https://labix.org/python-dateutil](https://labix.org/python-dateutil#head-c0e81a473b647dfa787dc11e8c69557ec2c3ecd2))
* **Schedule**: The times used in the schedule are similar to the timeseries with the addition of allowing the timestamps to represented as a string of the form “%A %H:%M:%S” which reads as “Day of the week 24-hour-clock:minutes:seconds”

In the schedule the values represent the state of the system and can be a value in {0,1} where 0 is off and 1 is on.

It is important to note that since the schedule is cyclical all that really needs to be defined is the change-of-state points. For a very simple schedule that schedules the lights to turn on at 8:00am Monday and off at 6:00pm Friday (leaving the lights on all hours during the week and turning them off for the weekend) could look like [(“Monday 08:00:00”, 1), (“Friday 18:00:00”, 0)].

* **relay\_timeseries**: There is no support for partial dimming so the values in this timeseries must be either 0 or 1 where 0 means the lights are off and 1 means they are on.
* **override\_timeseries**: Like the relay\_timeseries the values here must also be 0 or 1 where 0 is an override to off and 1 is an override to on. If no overrides occur during the monitoring period pass either None or an empty list.
* **load\_timeseries**: The values represent the electrical load. Units do not matter.
* **timezone**: A string that can be parsed by pytz.timezone (<http://pytz.sourceforge.net/>).
* **relay\_time\_comparison\_epsilon\_seconds**: Convertible to a float or None.
* **expected\_load\_min\_change**: Convertible to a float or None.
* **expected\_load\_max\_change**: Convertible to a float or None.
* **override\_timeout\_seconds:** Convertible to a float or None.
* **compress\_overrides\_to\_change\_of\_state**: Convertible to True or False.

## Output

**For a valid request**: A dictionary with the following fields. If there are any faults or suggested changes those fields will have values. If there are no faults or suggested changes the values will be None.

* **faults\_with\_implemented\_schedule**: Any cases where the system is not following implemented schedule that is not excused by a manual override. Each fault will have:
  + Fault description – A message that describes the fault and, if possible, provides some indication of likely locations for the fault.
  + Estimated start time – The time the fault appears in the stream. Note that this may not correspond to the actual time the fault starts, it is just the first time that it is seen in the data.
  + Estimated stop time – The time when the system re-enters a non-faulty state. Note that depending on the nature of the fault the fault may still exist (e.g. if the error is the system does not turn off when scheduled the end time will be the next time the system is scheduled to be on) or the fault may truly be over (e.g. if there was a glitch in recording the data due to a power outage)
* **faults\_with\_intended\_schedule**: Same as faults\_with\_implemented\_schedule but using the intended schedule. Can be used to differentiate issues due to misconfigured schedule from actual hardware faults.
* **suggested\_schedule\_changes**: Schedule changes can either be suggestions to lengthen the schedule because the space is still in use or to reduce the scheduled on periods because the space is empty for some of the scheduled on times.
* **suggested\_override\_timeout\_changes**: Override timeout changes can either be suggestions to lengthen the time that a manual override turns the lights on in cases where occupants need more time or suggestions to reduce the time if occupants do not use the entire override period.

**For an invalid request**: Posts a message to the same topic as the successful message but where the message content is a dictionary with one entry called “error” where the value is the error message.

### Output Formats

Times: Formatted as Year-Month-day 24 hours:minute:second timezone correction. In strftime notation: “%Y-%m-%d %H:%M:%S%z”

For example “5pm PST on September 17, 2014” would be returned as “2014-09-17 17:00:00-0800”