pyHottop Documentation

Release 0.1.0

Brandon Dixon

Mar 16, 2018

Contents

pyHottop gives you the power to control your Hottop KN-8828b-2k+ roaster directly through python code. **This library is meant to be used within applications and should not be used by itself to conduct a roast**. Questions, comments or for support needs, please use the issues page on Github.

CHAPTER 1

Features

This library provides full control of the Hottop roaster. Built-in callback functionality allows you to build applications that decouple the processing logic from the library from the core of your application.

- **Stream Hottop readings**
  - Easy-to-use callbacks that return readings
  - Adjustable polling interval
  - Human-readable settings
  - Flexible collection of data
  - Debugging interface
- **Control the Hottop directly**
  - Heater settings
  - Fan speeds
  - Drum motor toggle
  - Cooling motor toggle
  - Solenoid (drum door) toggle
  - Chaff tray (detection) reader
- **Auto-discover roaster connection**
  - Loops over USB connections to find the proper serial

CHAPTER 2

Code Documentation

2.1 Getting Started

In order to interact with your Hottop roaster, you need to ensure your model has a USB-serial port which comes standard with the KN-8828b-2k+.

1. Install the CP210x USB driver to read from the serial port:

https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

2. Install the pyHotop module:

`pip install pyhottop` or `python setup.py install`

3. Plug your Hottop roaster into your laptop.
4. Test connectivity to the roaster by running the diagnostic utility:

```
pyhottop-test test
```

2.2 Code Documentation

2.2.1 Hottop Interface

This is the primary class that you will interact with when building applications for the Hottop roaster. This class will automatically spawn a threaded instance of the control process which will handle the core of the operations against the roaster.

class pyhottop.pyhottop.**Hottop**
Object to interact and control the hottop roaster.

Returns Hottop instance

_autodiscover_usb()
Attempt to find the serial adapter for the hottop.

This will loop over the USB serial interfaces looking for a connection that appears to match the naming convention of the Hottop roaster.

Returns string

_callback(*data*)
Processor callback to clean-up stream data.

This function provides a hook into the output stream of data from the controller processing thread. Hottop readings are saved into a local class variable for later saving. If the user has defined a callback, it will be called within this private function.

Parameters **data** (*dict*) – Information from the controller process

Returns None

_derive_charge(*config*)
Use a temperature window to identify the roast charge.

The charge will manifest as a sudden downward trend on the temperature. Once found, we save it and avoid overwriting. The charge is needed in order to derive the turning point.

Parameters **config** (*dict*) – Current snapshot of the configuration

Returns None

_derive_turning_point(*config*)
Use a temperature window to identify the roast turning point.

Turning point relies on the charge being set first. We use the rolling 5-point window to measure slope. If we show a positive trend after the charge, then the temperature has begun to turn.

Parameters **config** (*dict*) – Current snapshot of the configuration

Returns None

_init_controls()
Establish a set of base controls the user can influence.

Returns None

_logger()
Create a logger to be used between processes.

Returns Logging instance.

add_roast_event(*event*)
Add an event to the roast log.

This method should be used for registering events that may be worth tracking like first crack, second crack and the dropping of coffee. Similar to the standard reading output from the roaster, manually created events will include the current configuration reading, time and metadata passed in.

Parameters **event** (*dict*) – Details describing what happened

Returns dict

connect(*interface=None*)
Connect to the USB for the hottop.

Attempt to discover the USB port used for the Hottop and then form a connection using the serial library.

Returns bool

Raises ***SerialConnectionError*** –

drop()
Preset call to drop coffee from the roaster via thread signal.

This will set the following configuration on the roaster: - drum_motor = 0 - heater = 0 - solenoid = 1 - cooling_motor = 1 - main_fan = 10

In order to power-off the roaster after dropping coffee, it's best to use the shutdown method. It's assumed that cooling will occur for 5-10 minutes before shutting down.

Returns None

end()
End the roaster control process via thread signal.

This simply sends an exit signal to the thread, and shuts it down. In order to stop monitoring, call the *set_monitor* method with false.

Returns None

get_cooling_motor()
Get the cooling motor config.

Returns None

get_current_config()
Get the current running config and state.

Returns dict

get_drum_motor()
Get the drum motor config.

Returns None

get_fan()
Get the fan config.

Returns int [0-10]

get_heater()
Get the heater config.

Returns int [0-100]

get_main_fan()
Get the main fan config.

Returns None

get_monitor()
Get the monitor config.

Returns None

get_roast()
Get the roast information.

Returns list

get_roast_properties()
Get the roast properties.

Returns dict

get_roast_time()
Get the roast time.

Returns float

get_serial_state()
Get the state of the USB connection.

Returns dict

get_simulate()
Get the simulation status.

Returns bool

get_solenoid()
Get the solenoid config.

Returns None

reset()
Reset the internal roast properties.

Returns None

set_cooling_motor(*cooling_motor*)
Set the cooling motor config.

Parameters **cooling_motor** (*bool*) – Value to set the cooling motor

Returns None

Raises InvalidInput

set_drum_motor(*drum_motor*)
Set the drum motor config.

Parameters **drum_motor** (*bool*) – Value to set the drum motor

Returns None

Raises InvalidInput

set_fan(*fan*)
Set the fan config.

Parameters **fan** (*int [0-10]*) – Value to set the fan

Returns None

Raises InvalidInput

set_heater(*heater*)
Set the heater config.

Parameters **heater** (*int [0-100]*) – Value to set the heater

Returns None

Raises InvalidInput

set_interval(*interval*)
Set the polling interval for the process thread.

Parameters **interval** (*int or float*) – How often to poll the Hottop

Returns None

Raises InvalidInput

set_main_fan(*main_fan*)
Set the main fan config.

Parameters **main_fan** (*int [0-10]*) – Value to set the main fan

Returns None

Raises InvalidInput

set_monitor(*monitor*)
Set the monitor config.

This module assumes that users will connect to the roaster and get reading information _before_ they want to begin collecting roast details. This method is critical to enabling the collection of roast information and ensuring it gets saved in memory.

Parameters **monitor** (*bool*) – Value to set the monitor

Returns None

Raises InvalidInput

set_roast_properties(*settings*)
Set the properties of the roast.

Parameters **settings** (*dict*) – General settings for the roast setup

Returns None

Raises InvalidInput

set_simulate(*status*)
Set the simulation status.

Parameters **status** (*bool*) – Value to set the simulation

Returns None

Raises InvalidInput

set_solenoid(*solenoid*)
Set the solenoid config.

Parameters **solenoid** (*bool*) – Value to set the solenoid

Returns None

Raises InvalidInput

start(*func=None*)
Start the roaster control process.

This function will kick off the processing thread for the Hottop and register any user-defined callback function. By default, it will not begin collecting any reading information or saving it. In order to do that users, must issue the monitor/record bit via *set_monitor*.

Parameters **func** (*function*) – Callback function for Hottop stream data

Returns None

2.2.2 Control Process

Due to the nature of continuously needing to poll the serial interface, a thread was required to handle interactions with the serial interface. It's possible to use the multiprocessing module to handle this work, but when using this

library inside of web server technology, multiprocessing often causes issues. Vanilla threads were used here to avoid interaction problems with co-routine or eventlet-based libraries.

class pyhottop.pyhottop.**ControlProcess**(*conn*, *config*, *q*, *logger*, *callback=None*)

Primary processor to communicate with the hottop directly.

Parameters

- **conn** (*Serial instance*) – Established serial connection to the Hottop
- **config** (*dict*) – Initial configurations settings
- **q** (*Queue instance*) – Shared queue to interact with the user interface
- **logger** (*Logging instance*) – Shared logger to keep continuity
- **callback** (*function*) – Optional callback function to stream results

Returns ControlProces instance

_generate_config()

Generate a configuration that can be sent to the Hottop roaster.

Configuration settings need to be represented inside of a byte array that is then written to the serial interface. Much of the configuration is static, but control settings are also included and pulled from the shared dictionary.

Returns Byte array of the prepared configuration.

_read_settings(*retry=True*)

Read the information from the Hottop.

Read the settings from the serial interface and convert them into a human-readable format that can be shared back to the end-user. Reading from the serial interface will occasionally produce strange results or blank reads, so a retry process has been built into the function as a recursive check.

Returns dict

_send_config()

Send configuration data to the hottop.

Returns bool

Raises Generic exceptions if an error is identified.

_valid_config(*settings*)

Scan through the returned settings to ensure they appear sane.

There are time when the returned buffer has the proper information, but the reading is inaccurate. When this happens, temperatures will swing or system values will be set to improper values.

Parameters **settings** (*dict*) – Configuration derived from the buffer

Returns bool

_validate_checksum(*buffer*)

Validate the buffer response against the checksum.

When reading the serial interface, data will come back in a raw format with an included checksum process.

Returns bool

_wake_up()

Wake the machine up to avoid race conditions.

When first interacting with the Hottop, the machine may not wake up right away which can put our reader into a death loop. This wake up routine ensures we prime the roaster with some data before starting our main loops to read/write data.

Returns None

drop()
Register a drop event to begin the cool-down process.

Returns None

run()
Run the core loop of reading and writing configurations.

This is where all the roaster magic occurs. On the initial run, we prime the roaster with some data to wake it up. Once awoke, we check our shared queue to identify if the user has passed any updated configuration. Once checked, start to read and write to the Hottop roaster as long as the exit signal has not been set. All steps are repeated after waiting for a specific time interval.

There are also specialized routines built into this function that are controlled via events. These events are unique to the roasting process and pre-configure the system with a configuration, so the user doesn't need to do it themselves.

Returns None

shutdown()
Register a shutdown event to stop interacting with the Hottop.

Returns None

2.3 Exceptions

class pyhottop.pyhottop.**InvalidInput**
Exception to capture invalid input commands.

class pyhottop.pyhottop.**SerialConnectionError**
Exception to capture serial connection issues.

2.4 Changelog

Running list of changes to the library.

2.4.1 2017-03-15

- Bugfix: Capture error when validating byte sequence

2.4.2 2017-03-07

- Change: Removed non-python3 dict method
- Bugfix: Error in valid config checking

2.4.3 2017-02-24

- Change: Added logic to add event code to find a valid configuration before saving

2.4.4 2017-02-10

- Change: Added logic to turning point logic to avoid setting too soon

2.4.5 2017-12-20

- Feature: Added a mock service to simulate a roast without being connected to a machine

2.4.6 2017-12-10

- Bugfix: Removed the reset on start as it clears any properties set by the user

2.4.7 2017-12-06

- Change: Keep the drum on by default to avoid any stalls

2.4.8 2017-12-03

- Change: Wrap the buffer read and pull from cache if it continues to fail
- Change: Adjusted lower bound temperature to 50
- Feature: Reset all the roast settings when starting a roast

2.4.9 2017-12-02

- Bugfix: Called the proper logging object on buffer measurement
- Change: Added raw buffer responses to the event log
- Feature: Added a validate routine to the buffer read to account for inaccurate responses from the roaster
- Feature: Automatically derive charge and turning point events based on temperature data

2.4.10 2017-12-01

- Bugfix: Turned drum motor on when doing a cool-down to push beans out

2.4.11 2017-11-29

- Bugfix: Replaced existing extenal_temp with environment_temp
- Bugfix: Fixed issue with buffer retry loop where it was not being called
- Change: Adjusted default interval to 1 second to avoid buffer issues
- Change: Toggle serial connection if having trouble reading buffer

2.4.12 2017-11-28

- Change: Adjusted duration to be of format MM:SS instead of total seconds
- Change: Return roast state when toggling monitoring

2.4.13 2017-11-24

- Feature: several new methods for getting additional roast details
- Change: Refactored code related to tracking roast properties and timing
- Change: Updated documentation within the code
- Bugfix: when running with python3 due to queue library

CHAPTER 3

License

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

E

G

H

I

R

S