ventilator

Release 0.0.0

jonny saunders et al

May 20, 2020

OVERVIEW

CHAPTER

ONE

VENT CONTROL PSEUDOCODE

Describes the procedure to operate low-cost ventilator under pressure control

1.1 Pressure control parameters

Set in GUI

- PIP: peak inhalation pressure (~20 cm H2O)
- T_insp: inspiratory time to PEEP (~0.5 sec)
- I/E: inspiratory to expiratory time ratio
- bpm: breaths per minute (15 bpm -> 1/15 sec cycle time)
- PIP_time: Target time for PIP. While lungs expand, dP/dt should be PIP/PIP_time
- flow_insp: nominal flow rate during inspiration

Set by hardware

- FiO2: fraction of inspired oxygen, set by blender
- max_flow: manual valve at output of blender
- PEEP: positive end-expiratory pressure, set by manual valve

Derived parameters

- cycle_time: 1/bpm
- t_insp: inspiratory time, controlled by cycle_time and I/E
- t_exp: expiratory time, controlled by cycle_time and I/E

Monitored variables

- Tidal volume: the volume of air entering the lung, derived from flow through t_exp
- PIP: peak inspiratory pressure, set by user in software
- Mean plateau pressure: derived from pressure sensor during inspiration cycle hold (no flow)
- PEEP: positive end-expiratory pressure, set by manual valve

Alarms

- Oxygen out of range
- High pressure (tube/airway occlusion)
- Low-pressure (disconnect)

- Temperature out of range
- Low voltage alarm (if using battery power)
- Tidal volume (expiratory) out of range

1.2 Hardware

Sensors

- O2 fraction, in inspiratory path
- Pressure, just before wye to endotrachial tube
- Flow, on expiratory path
- Temperature
- Humidity?

Actuators

- Inspiratory valve
  - Proportional or on/off
  - Must maintain low flow during expiratory cycle to maintain PEEP
- Expiratory valve
  - On/off in conjunction with PEEP valve probably OK

1.3 Pressure control loop

1. Begin inhalation
   - v1 Triggered by program every 1/bpm sec
   - v2 triggered by momentary drop in pressure when patient initiates inhalation (technically pressure-assisted control, PAC)

   1. ExpValve.close()
   2. InspValve.set(flow_insp)
2. While PSensor.read() < PIP
   1. Monitor d(PSensor.read())/dt
   2. Adjust flow rate for desired slope with controller
3. Cut flow and hold for t_insp
   1. InspValve.close()
   2. Monitor PSensor.read() and average across this time interval to report mean plateau pressure
4. Begin exhalation and hold for t_exp
   1. InspValve.set(PEEP_flow_rate) (alt: switch to parallel tube with continuous flow)
   2. ExpValve.open()
   3. integrate(FSensor.read()) for t_exhalation to determine V_tidal

5. Repeat from step 1.

CHAPTER

TWO

HARDWARE

CHAPTER

THREE

COMMON MODULE

3.1 message

Classes

`ControlSetting`(name, value, min_value, ...)	TODO: if enum is hard to use, we may just use a predefined set, e.g.
`Error`(errnum, err_str, timestamp)	
`SensorValues`([timestamp, loop_counter, ...])	**param timestamp** from time.time(). must be passed explicitly or as an entry in `vals`

class `vent.common.message.`**SensorValues**(*timestamp=None*, *loop_counter=None*, *breath_count=None*, *vals=None*, ***kwargs*)

Bases: `object`

Parameters

- **timestamp** (*float*) – from time.time(). must be passed explicitly or as an entry in `vals`
- **loop_counter** (*int*) – number of control_module loops. must be passed explicitly or as an entry in `vals`
- **breath_count** (*int*) – number of breaths taken. must be passed explicitly or as an entry in `vals`
- ****kwargs** – sensor readings, must be in `vent.values.SENSOR.keys`

Methods

`__init__`([timestamp, loop_counter, ...])	**param timestamp** from time.time(). must be passed explicitly or as an entry in `vals`
`to_dict`()	

Attributes

additional_values	Built-in immutable sequence.

additional_values = ('timestamp', 'loop_counter', 'breath_count')

__init__(*timestamp=None*, *loop_counter=None*, *breath_count=None*, *vals=None*, ***kwargs*)

Parameters

- **timestamp** (*float*) – from time.time(). must be passed explicitly or as an entry in vals
- **loop_counter** (*int*) – number of control_module loops. must be passed explicitly or as an entry in vals
- **breath_count** (*int*) – number of breaths taken. must be passed explicitly or as an entry in vals
- ****kwargs** – sensor readings, must be in vent.values.SENSOR.keys

to_dict()

class vent.common.message.**ControlSetting**(*name*, *value*, *min_value*, *max_value*, *timestamp*)

Bases: object

TODO: if enum is hard to use, we may just use a predefined set, e.g. {'PIP', 'PEEP', ...} :param name: enum belong to ValueName :param value: :param min_value: :param max_value: :param timestamp: **Methods**

__init__(name, value, min_value, max_value, ...)	TODO: if enum is hard to use, we may just use a predefined set, e.g.

__init__(*name*, *value*, *min_value*, *max_value*, *timestamp*)

TODO: if enum is hard to use, we may just use a predefined set, e.g. {'PIP', 'PEEP', ...} :param name: enum belong to ValueName :param value: :param min_value: :param max_value: :param timestamp:

class vent.common.message.**Error**(*errnum*, *err_str*, *timestamp*)

Bases: object

3.2 values

Parameterization of variables and values

Data

CONTROL	Values to control but not monitor.
LIMITS	Values that are dependent on other values:
SENSOR	Values to monitor but not control.

Classes

Value(name, units, abs_range, safe_range, ...)	Definition of a value.
ValueName	An enumeration.

class vent.common.values.**ValueName**

Bases: enum.Enum

An enumeration. **Attributes**

BREATHS_PER_MINUTE	int([x]) -> integer
FIO2	int([x]) -> integer
HUMIDITY	int([x]) -> integer
IE_RATIO	int([x]) -> integer
INSPIRATION_TIME_SEC	int([x]) -> integer
PEEP	int([x]) -> integer
PEEP_TIME	int([x]) -> integer
PIP	int([x]) -> integer
PIP_TIME	int([x]) -> integer
PRESSURE	int([x]) -> integer
TEMP	int([x]) -> integer
VTE	int([x]) -> integer

PIP = 1

PIP_TIME = 2

PEEP = 3

PEEP_TIME = 4

BREATHS_PER_MINUTE = 5

INSPIRATION_TIME_SEC = 6

IE_RATIO = 7

FIO2 = 8

TEMP = 9

HUMIDITY = 10

VTE = 11

PRESSURE = 12

class vent.common.values.**Value**(*name: str*, *units: str*, *abs_range: tuple*, *safe_range: tuple*, *decimals: int*, *control: bool*, *sensor: bool*, *default: (<class 'int'>, <class 'float'>) = None*)

Bases: object

Definition of a value.

Used by the GUI and control module to set defaults.

Parameters

- **name** (*str*) – Human-readable name of the value
- **units** (*str*) – Human-readable description of units
- **abs_range** (*tuple*) – tuple of ints or floats setting the logical limit of the value, eg. a percent between 0 and 100, (0, 100)
- **safe_range** (*tuple*) – tuple of ints or floats setting the safe ranges of the value,

  note:

  ```
  this is not the same thing as the user-set alarm values,
  though the user-set alarm values are initialized as ``safe_range``.
  ```

- **decimals** (*int*) – the number of decimals of precision used when displaying the value

Methods

`__init__`(name, units, abs_range, safe_range, ...)	Definition of a value.
`to_dict`()	

Attributes

`abs_range`
`control`
`decimals`
`default`
`name`
`safe_range`
`sensor`

__init__(*name: str*, *units: str*, *abs_range: tuple*, *safe_range: tuple*, *decimals: int*, *control: bool*, *sensor: bool*, *default: (<class 'int'>, <class 'float'>) = None*)

Definition of a value.

Used by the GUI and control module to set defaults.

Parameters

- **name** (*str*) – Human-readable name of the value
- **units** (*str*) – Human-readable description of units
- **abs_range** (*tuple*) – tuple of ints or floats setting the logical limit of the value, eg. a percent between 0 and 100, (0, 100)
- **safe_range** (*tuple*) – tuple of ints or floats setting the safe ranges of the value,

  note:

  ```
  this is not the same thing as the user-set alarm values,
  though the user-set alarm values are initialized as ``safe_
  ↪range``.
  ```

- **decimals** (*int*) – the number of decimals of precision used when displaying the value

property name

property abs_range

property safe_range

property decimals

property default

property control

property sensor

to_dict()

vent.common.values.**SENSOR = OrderedDict([(<ValueName.FIO2: 8>, <vent.common.values.Value o**

Values to monitor but not control.

Used to set alarms for out-of-bounds sensor values. These should be sent from the control module and not computed.:

```
{
    'name' (str):  Human readable name,
    'units' (str): units string, (like degrees or %),
    'abs_range' (tuple): absolute possible range of values,
    'safe_range' (tuple): range outside of which a warning will be raised,
    'decimals' (int): The number of decimals of precision this number should be␣
↪displayed with
}
```

vent.common.values.**CONTROL = OrderedDict([(<ValueName.PIP: 1>, <vent.common.values.Value ob**

Values to control but not monitor.

Sent to control module to control operation of ventilator.:

```
{
    'name' (str):  Human readable name,
    'units' (str): units string, (like degrees or %),
    'abs_range' (tuple): absolute possible range of values,
    'safe_range' (tuple): range outside of which a warning will be raised,
    'default' (int, float): the default value of the parameter,
    'decimals' (int): The number of decimals of precision this number should be␣
↪displayed with
}
```

vent.common.values.**LIMITS = {}**

Values that are dependent on other values:

```
{
    "dependent_value": (
        ['value_1', 'value_2'],
        callable_returning_boolean
    }
}
```

Where the first argument in the tuple is a list of the values that will be given as argument to the `callable_returning_boolean` which will return whether (`True`) or not (`False`) a value is allowed.

CHAPTER

FOUR

CONTROLLER MODULE

Classes

Balloon_Simulator(leak)	This is a imple physics simulator for inflating a balloon.
ControlModuleBase()	This is an abstract class for controlling simulation and hardware.
ControlModuleDevice()	
ControlModuleSimulator()	

Functions

get_control_module([sim_mode])

class vent.controller.control_module.**ControlModuleBase**

Bases: object

This is an abstract class for controlling simulation and hardware.

1. **All internal variables fall in three classes, denoted by the beginning of the variable:**
   - "COPY_varname": These are copies (see 1.) that are regularly sync'ed with internal variables.
   - "__varname": These are variables only used in the ControlModuleBase-Class
   - "_varname": These are variables used in derived classes.
2. **Internal variables should only to be accessed though the set_ and get_ functions.** These functions act on COPIES of internal variables ("__" and "_"), that are sync'd every few iterations. How often this is done is adjusted by the variable self._NUMBER_CONTROLL_LOOPS_UNTIL_UPDATE. To avoid multiple threads manipulating the same variables at the same time, every manipulation of "**COPY_**" is surrounded by a thread lock.

Methods

_ControlModuleBase__analyze_last_waveform()	This goes through the last waveform, and updates VTE, PEEP, PIP, PIP_TIME, I_PHASE, FIRST_PEEP and BPM.
_ControlModuleBase__calculate_control_signal_in()	
_ControlModuleBase__get_PID_error(ytarget, ...)	
_ControlModuleBase__test_critical_levels(...)	This test whether a variable is within bounds.
_ControlModuleBase__update_alarms()	This goes through the values obtained from the last waveform, and updates alarms.

continues on next page

Table 3 – continued from previous page

_PID_reset()	resets the PID cycle to zero
_PID_update(dt)	This instantiates the control algorithms.
_alarm_to_COPY()	
_controls_from_COPY()	
_get_control_signal_in()	This is the PID controlled signal on the inspiratory side
_get_control_signal_out()	This is the control signal (open/close) on the expiratory side
_initialize_set_to_COPY()	
_sensor_to_COPY()	
_start_mainloop()	
do_pid_control()	
do_state_control()	
get_active_alarms()	
get_control(control_setting_name)	Gets values of the COPY of the control settings.
get_logged_alarms()	
get_past_waveforms()	
get_sensors()	
get_target_waveform()	
is_running()	
set_control(control_setting)	Updates the entry of COPY contained in the control settings
start()	
stop()	

Public Methods: get_sensors(): Returns a copy of the current sensor values. get_alarms(): Returns a List of all alarms, active and logged get_active_alarms(): Returns a Dictionary of all currently active alarms. get_logged_alarms(): Returns a List of logged alarms, up to maximum lengh of self._RINGBUFFER_SIZE get_control(ControlSetting): Sets a controll-setting. Is updated at latest within self._NUMBER_CONTROLL_LOOPS_UNTIL_UPDATE get_past_waveforms(): Returns a List of waveforms of pressure and volume during at the last N breath cycles, N<self._RINGBUFFER_SIZE, AND clears this archive. get_target_waveform(): Returns a step-wise linear target waveform, as defined by the current settings. start(): Starts the main-loop of the controller stop(): Stops the main-loop of the controller

_initialize_set_to_COPY()

_alarm_to_COPY()

_sensor_to_COPY()

_controls_from_COPY()

get_sensors() → vent.common.message.SensorValues

get_active_alarms()

get_logged_alarms() → List[vent.alarm.alarm.Alarm]

set_control(*control_setting: vent.common.message.ControlSetting*)
Updates the entry of COPY contained in the control settings

get_control(*control_setting_name: vent.common.values.ValueName*) → vent.common.message.ControlSetting
Gets values of the COPY of the control settings.

_get_control_signal_in()
This is the PID controlled signal on the inspiratory side

_get_control_signal_out()
This is the control signal (open/close) on the expiratory side

_PID_reset()
resets the PID cycle to zero

_PID_update(*dt*)
This instantiates the control algorithms. During the breathing cycle, it goes through the four states:

1) Rise to PIP
2) Sustain PIP pressure
3) Quick fall to PEEP
4) Sustaint PEEP pressure

Once the cycle is complete, it checks the cycle for any alarms, and starts a new one. A record of pressure/volume waveforms is kept in self.__cycle_waveform_archive

dt: Time since last update in seconds

get_past_waveforms()

get_target_waveform()

_start_mainloop()

start()

stop()

is_running()

do_pid_control()

do_state_control()

_ControlModuleBase__analyze_last_waveform()
This goes through the last waveform, and updates VTE, PEEP, PIP, PIP_TIME, I_PHASE, FIRST_PEEP and BPM.

_ControlModuleBase__calculate_control_signal_in()

_ControlModuleBase__get_PID_error(*ytarget*, *yis*, *dt*)

_ControlModuleBase__test_critical_levels(*min*, *max*, *value*, *name*)
This tests whether a variable is within bounds. If it is, and an alarm existed, then the "alarm_end_time" is set. If it is NOT, a new alarm is generated and appendede to the alarm-list. Input:

min: minimum value (e.g. 2) max: maximum value (e.g. 5) value: test value (e.g. 3) name: parameter type (e.g. "PIP", "PEEP" etc.)

_ControlModuleBase__update_alarms()
This goes through the values obtained from the last waveform, and updates alarms.

class vent.controller.control_module.**ControlModuleDevice**
Bases: vent.controller.control_module.ControlModuleBase **Methods**

_sensor_to_COPY()
_start_mainloop()

_sensor_to_COPY()

_start_mainloop()

class vent.controller.control_module.**Balloon_Simulator**(*leak*)
Bases: object

This is a imple physics simulator for inflating a balloon. For math, see https://en.wikipedia.org/wiki/Two-balloon_experiment **Methods**

OUupdate(variable, dt, mu, sigma, tau)	This is a simple function to produce an OU process.
_reset()	resets the ballon to standard parameters.
get_pressure()	
get_volume()	
set_flow_in(Qin, dt)	
set_flow_out(Qout, dt)	
update(dt)	

get_pressure()

get_volume()

set_flow_in(*Qin*, *dt*)

set_flow_out(*Qout*, *dt*)

update(*dt*)

OUupdate(*variable*, *dt*, *mu*, *sigma*, *tau*)
This is a simple function to produce an OU process. It is used as model for fluctuations in measurement variables. inputs: variable: float value at previous time step dt : timestep mu : mean sigma : noise amplitude tau : time scale returns: new_variable : value of "variable" at next time step

_reset()
resets the ballon to standard parameters.

class vent.controller.control_module.**ControlModuleSimulator**
Bases: vent.controller.control_module.ControlModuleBase **Methods**

_ControlModuleSimulator__SimulatedPropValve(x, dt)	This simulates the action of a proportional valve.
_ControlModuleSimulator__SimulatedSolenoid(x)	Depending on x, set flow to a binary value.
_sensor_to_COPY()	
_start_mainloop()	

_sensor_to_COPY()

_start_mainloop()

_ControlModuleSimulator__SimulatedPropValve(*x*, *dt*)
This simulates the action of a proportional valve. Flow-current-curve eye-balled from the datasheet of SMC PVQ31-5G-23-01N https://www.ocpneumatics.com/content/pdfs/PVQ.pdf

x: Input current [mA] dt: Time since last setting in seconds [for the LP filter]

_ControlModuleSimulator__SimulatedSolenoid(*x*)
Depending on x, set flow to a binary value. Here: flow is either 0 or 1l/sec

vent.controller.control_module.**get_control_module**(*sim_mode=False*)

CHAPTER

FIVE

COORDINATOR MODULE

5.1 Submodules

5.2 coordinator

Classes

CoordinatorBase([sim_mode])	
CoordinatorLocal([sim_mode])	**param sim_mode**
CoordinatorRemote([sim_mode])	

Functions

get_coordinator([single_process, sim_mode])	

class vent.coordinator.coordinator.**CoordinatorBase**(*sim_mode=False*)

Bases: object **Methods**

get_control(control_setting_name)	
get_sensors()	
is_running()	
set_control(control_setting)	
start()	
stop()	

get_sensors() → vent.common.message.SensorValues

set_control(*control_setting: vent.common.message.ControlSetting*)

get_control(*control_setting_name: vent.common.values.ValueName*) → vent.common.message.ControlSetting

start()

is_running() → bool

stop()

class vent.coordinator.coordinator.**CoordinatorLocal**(*sim_mode=False*)

Bases: vent.coordinator.coordinator.CoordinatorBase

Parameters **sim_mode** –

Methods

__init__([sim_mode])	**param sim_mode**
get_control(control_setting_name)	
get_sensors()	
is_running()	Test whether the whole system is running
set_control(control_setting)	
start()	Start the coordinator.
stop()	Stop the coordinator.

_is_running
.set() when thread should stop

Type threading.Event

__init__(*sim_mode=False*)

Parameters **sim_mode** –

_is_running
.set() when thread should stop
Type threading.Event

get_sensors() → vent.common.message.SensorValues

set_control(*control_setting: vent.common.message.ControlSetting*)

get_control(*control_setting_name: vent.common.values.ValueName*) → vent.common.message.ControlSetting

start()
Start the coordinator. This does a soft start (not allocating a process).

is_running() → bool
Test whether the whole system is running

stop()
Stop the coordinator. This does a soft stop (not kill a process)

class vent.coordinator.coordinator.**CoordinatorRemote**(*sim_mode=False*)
Bases: vent.coordinator.coordinator.CoordinatorBase **Methods**

get_control(control_setting_name)	
get_sensors()	
is_running()	Test whether the whole system is running
set_control(control_setting)	
start()	Start the coordinator.
stop()	Stop the coordinator.

get_sensors() → vent.common.message.SensorValues

set_control(*control_setting: vent.common.message.ControlSetting*)

get_control(*control_setting_name: vent.common.values.ValueName*) → vent.common.message.ControlSetting

start()
Start the coordinator. This does a soft start (not allocating a process).

is_running() → bool
Test whether the whole system is running

stop()
Stop the coordinator. This does a soft stop (not kill a process)

vent.coordinator.coordinator.**get_coordinator**(*single_process=False*, *sim_mode=False*) → vent.coordinator.coordinator.CoordinatorBase

5.3 ipc

5.4 process_manager

Classes

ProcessManager(sim_mode[, startCommandLine, ...])	

class vent.coordinator.process_manager.**ProcessManager**(*sim_mode*, *startCommandLine=None*, *maxHeartbeatInterval=None*)

Bases: object **Methods**

heartbeat(timestamp)	
restart_process()	
start_process()	
try_stop_process()	

start_process()

try_stop_process()

restart_process()

heartbeat(*timestamp*)

CHAPTER

SIX

GUI

6.1 Program Diagram

6.2 Design Requirements

- Display Values
  - Value name, units, absolute range, recommended range, default range
  - VTE
  - FiO2
  - Humidity
  - Temperature
- Plots
- Controlled Values
  - PIP
  - PEEP
  - Inspiratory Time
- Value Dependencies

6.3 UI Notes & Todo

- Jonny add notes from helpful RT in discord!!!
- Top status Bar
  - Start/stop button
  - Status indicator - a clock that increments with heartbeats, or some other visual indicator that things are alright
  - Status bar - most recent alarm or notification w/ means of clearing
  - Override to give 100% oxygen and silence all alarms
- API
  - Two queues, input and output. Read from socket and put directly into queue.

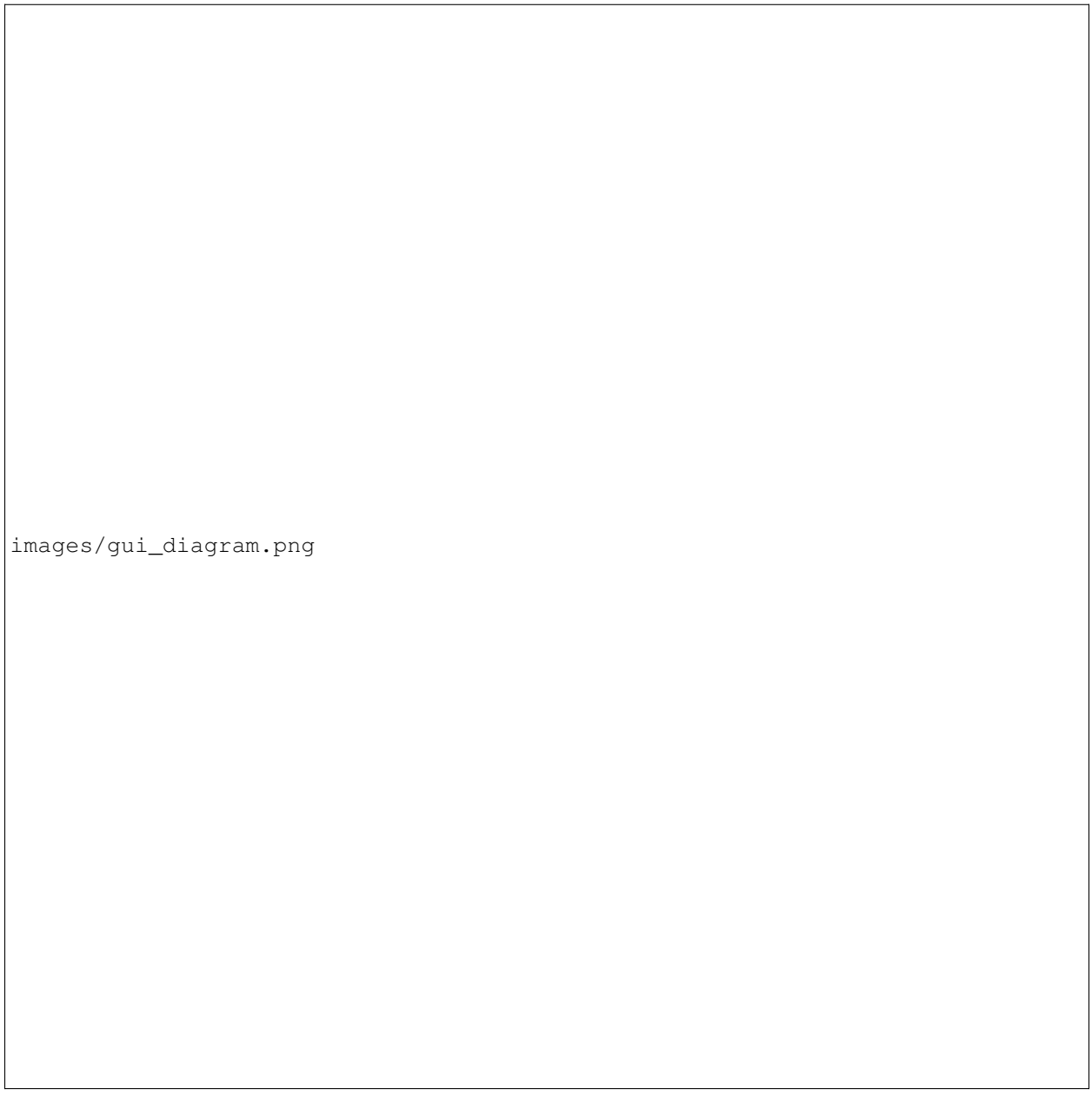

Fig. 1: Schematic diagram of major UI components and signals

  - Input, receive (timestamp, key, value) messages where key and value are names of variables and their values
  - Output, send same format
- Menus
  - Trigger some testing/calibration routine
  - Log/alarm viewer
  - Wizard to set values?
  - save/load values
- Alarms
  - Multiple levels
  - Silenced/reset
  - Logging
  - Sounds?
- General
  - Reduce space given to waveforms
  - Clearer grouping & titling for display values & controls
  - Collapsible range setting
  - Ability to declare dependencies between values
    - Limits - one value's range logically depends on another
    - Derived - one value is computed from another/others
  - Monitored values should have defaults, warning range, and absolute range
  - Two classes of monitored values – ones with limits and ones without. There seem to be lots and lots of observed values, but only some need limits. might want to make larger drawer of smaller displayed values that don't need controls
  - Save/load parameters. Autosave, and autorestore if saved <5m ago, otherwise init from defaults.
  - Implement timed updates to plots to limit resource usage
  - Make class for setting values
  - Possible plots
    - Pressure vs. flow
    - flow vs volume
    - volume vs time
- Performance
  - Cache drawText() calls in range selector by drawing to pixmap

6.4 Jonny Questions

- Which alarm sounds to use?
- Final final final breakdown on values and ranges plzzz

6.4.1 jonny todo

- use loop_counter to check on controller advancement
- implement single values list with properties 'controllable' vs not.

6.5 GUI Object Documentation

6.5.1 Control

6.5.2 Monitor

Classes

`Monitor`(value[, update_period, enum_name])	**param value**

class vent.gui.widgets.monitor.**Monitor**(*value*, *update_period=0.5*, *enum_name=None*)

Parameters

- **value** (Value) –
- **update_period** (*float*) – update period of monitor in s

Methods

`__init__`(value[, update_period, enum_name])	**param value**
`_limits_changed`(val)	
`check_alarm`([value])	
`init_ui`()	
`set_alarm`(alarm)	Simple wrapper to set alarm state from a qt signal
`timed_update`()	
`toggle_alarm`()	
`toggle_control`(state)	
`update_boxes`(new_values)	
`update_limits`(new_limits)	
`update_value`(new_value)	

Attributes

alarm_state

alarm(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

limits_changed(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

__init__(*value*, *update_period=0.5*, *enum_name=None*)

Parameters

- **value** (Value) –
- **update_period** (*float*) – update period of monitor in s

init_ui()

toggle_control(*state*)

update_boxes(*new_values*)

update_value(*new_value*)

update_limits(*new_limits*)

timed_update()

_limits_changed(*val*)

property alarm_state

set_alarm(*alarm*)
Simple wrapper to set alarm state from a qt signal

Parameters **alarm** (*bool*) – Whether to set as alarm state or not

toggle_alarm()

check_alarm(*value=None*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

6.5.3 Plot

Data

PLOT_FREQ	Update frequency of Plot s in Hz
PLOT_TIMER	A QTimer that updates :class:`.TimedPlotCurveItem`s

Classes

Plot(name[, buffer_size, plot_duration, ...])	When initializing PlotWidget, *parent* and *background* are passed to GraphicsWidget.__init__() and all others are passed to PlotItem.__init__().

vent.gui.widgets.plot.**PLOT_TIMER = None**
A QTimer that updates :class:`.TimedPlotCurveItem`s

vent.gui.widgets.plot.**PLOT_FREQ = 5**
Update frequency of Plot s in Hz

class vent.gui.widgets.plot.**Plot**(*name*, *buffer_size=4092*, *plot_duration=5*, *abs_range=None*, *safe_range=None*, *color=None*, *units=''*, ***kwargs*)

When initializing PlotWidget, *parent* and *background* are passed to `GraphicsWidget.__init__()` and all others are passed to `PlotItem.__init__()`. **Methods**

`_safe_limits_changed`(val)	
`set_duration`(dur)	
`set_safe_limits`(limits)	
`update_value`(new_value)	new_value: (timestamp from time.time(), value)

limits_changed(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

set_duration(*dur*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

update_value(*new_value*)

new_value: (timestamp from time.time(), value)

_safe_limits_changed(*val*)

set_safe_limits(*limits*)

6.5.4 Status Bar

Classes

`HeartBeat`([update_interval, timeout_dur])	**param update_interval** How often to do the heartbeat, in ms
`Message_Display`()	
`Power_Button`()	
`Status_Bar`()	• Start/stop button

class vent.gui.widgets.status_bar.**Status_Bar**

- Start/stop button
- **Status indicator - a clock that increments with heartbeats,** or some other visual indicator that things are alright
- Status bar - most recent alarm or notification w/ means of clearing
- Override to give 100% oxygen and silence all alarms

Methods

`init_ui`()

init_ui()

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class vent.gui.widgets.status_bar.**Message_Display**

Attributes

alarm_level

Methods

clear_message()	
draw_state([state])	
init_ui()	
make_icons()	
update_message(alarm)	**param alarm**

message_cleared(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

level_changed(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

make_icons()

init_ui()

draw_state(*state=None*)

update_message(*alarm*)

Parameters **alarm** (Alarm) –

clear_message()

property alarm_level

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class vent.gui.widgets.status_bar.**HeartBeat**(*update_interval=100*, *timeout_dur=5000*)

Parameters

- **update_interval** (*int*) – How often to do the heartbeat, in ms
- **timeout** (*int*) – how long to wait before hearing from control process

Methods

__init__([update_interval, timeout_dur])	**param update_interval** How often to do the heartbeat, in ms
_heartbeat()	Called every (update_interval) milliseconds to set the text of the timer.
beatheart(heartbeat_time)	
check_timeout()	
init_ui()	
set_indicator([state])	
start_timer([update_interval])	**param update_interval** How often (in ms) the timer should be updated.

continues on next page

Table 11 – continued from previous page

stop_timer()	you can read the sign ya punk

timeout(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

heartbeat(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

__init__(*update_interval=100*, *timeout_dur=5000*)

Parameters

- **update_interval** (*int*) – How often to do the heartbeat, in ms
- **timeout** (*int*) – how long to wait before hearing from control process

init_ui()

check_timeout()

set_indicator(*state=None*)

start_timer(*update_interval=None*)

Parameters **update_interval** (*float*) – How often (in ms) the timer should be updated.

stop_timer()
you can read the sign ya punk

beatheart(*heartbeat_time*)

_heartbeat()
Called every (update_interval) milliseconds to set the text of the timer.

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class vent.gui.widgets.status_bar.**Power_Button**
Methods

init_ui()	

init_ui()

staticMetaObject = <PySide2.QtCore.QMetaObject object>

6.5.5 Components

6.5.5.1 GUI Stylesheets

Data

MONITOR_UPDATE_INTERVAL	inter-update interval (seconds) for Monitor

Functions

set_dark_palette(app)	Apply Dark Theme to the Qt application instance.

vent.gui.styles.**MONITOR_UPDATE_INTERVAL = 0.5**
inter-update interval (seconds) for Monitor

Type (float)

vent.gui.styles.**set_dark_palette**(*app*)
Apply Dark Theme to the Qt application instance.

borrowed from https://github.com/gmarull/qtmodern/blob/master/qtmodern/styles.py

Args: app (QApplication): QApplication instance.

6.5.5.2 GUI Alarm Manager

Classes

AlarmManager()

Functions

get_alarm_manager()

vent.gui.alarm_manager.**get_alarm_manager**()

class vent.gui.alarm_manager.**AlarmManager**
Methods

monitor_alarm(alarm)	Parse a tentative alarm from a monitor – we should have already gotten an alarm from the controller, so this largely serves as a double check.
parse_message(alarm)	If an alarm doesn't have a message attr, make one for it.
update_alarms(alarms)	

new_alarm(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**

update_alarms(*alarms*)

monitor_alarm(*alarm*)
Parse a tentative alarm from a monitor – we should have already gotten an alarm from the controller, so this largely serves as a double check.

Doesn't use the Alarm class because creating a new alarm increments the counter.

Parameters **alarm** (*tuple*) – (monitor_name, monitor_value, timestamp)

parse_message(*alarm*)
If an alarm doesn't have a message attr, make one for it.

staticMetaObject = <PySide2.QtCore.QMetaObject object>

CHAPTER

SEVEN

VENT.IO PACKAGE

7.1 Subpackages

7.1.1 vent.io.devices package

7.1.1.1 Submodules

7.1.1.2 vent.io.devices.base module

Base classes & functions used throughout vent.io.devices

Classes

ADS1015([address, i2c_bus, pig])	ADS1015 16 bit, 4 Channel Analog to Digital Converter.
ADS1115([address, i2c_bus, pig])	ADS1115 16 bit, 4 Channel Analog to Digital Converter.
I2CDevice(i2c_address, i2c_bus[, pig])	A class wrapper for pigpio I2C handles.
IODeviceBase(pig)	Abstract base Class for pigpio handles (or whatever other GPIO library
SPIDevice(channel, baudrate[, pig])	A class wrapper for pigpio SPI handles.

Functions

be16_to_native(data[, signed])	Unpacks a bytes-like object respecting big-endianness of outside world and returns an int according to signed.
native16_to_be(word[, signed])	Packs an int into bytes after swapping endianness.

class vent.io.devices.base.**IODeviceBase**(*pig: vent.io.devices.base.PigpioConnection = None*)

Bases: object

Abstract base Class for pigpio handles (or whatever other GPIO library we end up using)

Note: pigpio commands return -144 if an error is encountered while attempting to communicate with the demon. TODO would be to recognize when that occurs and handle it gracefully, i.e. kill the daemon, restart it, and reopen the python interface(s)

Initializes the pigpio python bindings object if necessary, and checks that it is actually running.

Parameters **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Methods

__init__(pig)	Initializes the pigpio python bindings object if necessary, and checks that it is actually running.
_close()	Closes an I2C/SPI (or potentially Serial) connection

Attributes

handle	Pigpiod handle associated with device (only for i2c/spi)
pig	The pigpio python bindings object
pigpiod_ok	Returns True if pigpiod is running and False if not

__init__ (*pig: vent.io.devices.base.PigpioConnection = None*)
Initializes the pigpio python bindings object if necessary, and checks that it is actually running.

Parameters **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

property pig
The pigpio python bindings object

property handle
Pigpiod handle associated with device (only for i2c/spi)

property pigpiod_ok
Returns True if pigpiod is running and False if not

_close()
Closes an I2C/SPI (or potentially Serial) connection

class vent.io.devices.base.**I2CDevice**(*i2c_address*, *i2c_bus*, *pig=None*)
Bases: *vent.io.devices.base.IODeviceBase*

A class wrapper for pigpio I2C handles. Defines several methods used for reading from and writing to device registers. Defines helper classes Register and ValueField for handling the manipulation of arbitrary registers.

Note: The Raspberry Pi uses LE byte-ordering, while the outside world tends to use BE (at least, the sensors in use so far all do). Thus, bytes need to be swapped from native (LE) ordering to BE prior to being written to an i2c device, and bytes recieved need to be swapped from BE into native (LE). All methods except read_device and write_device perform this automatically. The methods read_device and write_device do NOT byteswap and return bytearrays rather than the unsigned 16-bit int used by the other read/write methods.

Initializes pigpio bindings and opens i2c connection.

Parameters

- **i2c_address** (*int*) – I2C address of the device. (e.g., *i2c_address=0x50*)
- **i2c_bus** (*int*) – The I2C bus to use. Should probably be set to 1 on Raspberry Pi.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Classes

Register(fields, values)	Describes a writable configuration register.

Methods

__init__(i2c_address, i2c_bus[, pig])	Initializes pigpio bindings and opens i2c connection.
_close()	Extends superclass method.
_open(i2c_bus, i2c_address)	Opens i2c connection given i2c bus and address.
read_device([count])	Read a specified number of bytes directly from the the device without specifying or changing the register.
read_register(register[, signed])	Read 2 bytes from the specified register and byteswap the result.
write_device(word[, signed])	Write 2 bytes to the device without specifying register.
write_register(register, word[, signed])	Write 2 bytes to the specified register.

__init__ (*i2c_address*, *i2c_bus*, *pig=None*)
Initializes pigpio bindings and opens i2c connection.

Parameters

- **i2c_address** (*int*) – I2C address of the device. (e.g., *i2c_address=0x50*)
- **i2c_bus** (*int*) – The I2C bus to use. Should probably be set to 1 on Raspberry Pi.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

_open (*i2c_bus*, *i2c_address*)
Opens i2c connection given i2c bus and address.

_close ()
Extends superclass method. Checks that pigpiod is connected and if a handle has been set - if so, closes an i2c connection.

read_device (*count=2*) → tuple
Read a specified number of bytes directly from the the device without specifying or changing the register. Does NOT perform LE/BE conversion.

Parameters **count** (*int*) – The number of bytes to read from the device.

Returns a tuple of the number of bytes read and a bytearray containing the bytes. If there was an error the number of bytes read will be less than zero (and will contain the error code).

Return type tuple

write_device (*word*, *signed=False*)
Write 2 bytes to the device without specifying register. DOES perform LE/BE conversion.

Parameters

- **word** (*int*) – The integer representation of the data to write.
- **signed** (*bool*) – Whether or not *word* is signed.

read_register (*register*, *signed=False*) → int
Read 2 bytes from the specified register and byteswap the result.

Parameters

- **register** (*int*) – The index of the register to read.
- **signed** (*bool*) – Whether or not the data to read is expected to be signed.

Returns integer representation of 16 bit register contents.

Return type int

write_register(*register*, *word*, *signed=False*)
Write 2 bytes to the specified register. Byteswaps.

Parameters

- **register** (*int*) – The index of the register to write to
- **word** (*int*) – The unsigned 16 bit integer to write to the register (must be consistent with 'signed')
- **signed** (*bool*) – Whether or not 'word' is signed

class Register(*fields*, *values*)
Bases: object

Describes a writable configuration register. Has dynamically defined attributes corresponding to the fields described by the passed arguments. Takes as arguments two tuples of equal length, the first of which names each field and the second being a tuple of tuples containing the (human readable) possible settings & values for each field.

Note: The initializer reverses the fields & their values because a human reads the register, as drawn in the datasheet, from left to right - however, the fields furthest to the left are the most significant bits of the register.

Initializer which loads (dynamically defined) attributes from tuples.

Parameters

- **fields** (*tuple*) – A tuple containing the names of the register's value fields
- **values** (*tuple*) – A tuple of tuples containing the possible values for each value field. Length must match the length of fields. If there are redundant values for a field specified in the datasheet, be sure to include them. (e.g., a field takes values *A: 0b00*, *B: 0b01*, and *C: 0b10*; but the value for *0b11* is either not specified by the datasheet or is listed redundantly as *C: 0b11* -> *values* should list both the 3rd and 4th possible values as 'C' like so: ('A', 'B', 'C', 'C')

__init__(*fields*, *values*)
Initializer which loads (dynamically defined) attributes from tuples.

Parameters

- **fields** (*tuple*) – A tuple containing the names of the register's value fields
- **values** (*tuple*) – A tuple of tuples containing the possible values for each value field. Length must match the length of fields. If there are redundant values for a field specified in the datasheet, be sure to include them. (e.g., a field takes values *A: 0b00*, *B: 0b01*, and *C: 0b10*; but the value for *0b11* is either not specified by the datasheet or is listed redundantly as *C: 0b11* -> *values* should list both the 3rd and 4th possible values as 'C' like so: ('A', 'B', 'C', 'C')

unpack(*cfg*) → collections.OrderedDict
Given the contents of a register in integer form, returns a dict of fields and their current settings.

Parameters **cfg** (*int*) – An integer representing a possible configuration value for the register

pack(*cfg*, ***kwargs*) → int
Given an initial integer representation of a register and an arbitrary number of field=value settings, returns an integer representation of the register incorporating the new settings.

Parameters

- **cfg** (*int*) – An integer representing a possible configuration value for the register
- ****kwargs** – The register fields & values to patch into cfg. Takes keyword arguments of the form: *field=value*

class ValueField(*offset*, *mask*, *values*)

Bases: object

Describes a configurable value field in a writable register.

Instantiates a value field of a register given the bit offset, mask, and list of possible values.

Parameters

- **offset** (*int*) – The offset bits of the value field in the register, i.e. the distance from LSB
- **mask** (*int*) – integer representation of the value field mask (w/o offset)
- **values** (*OrderedDict*) – The possible values that the field can take.

__init__(*offset*, *mask*, *values*)

Instantiates a value field of a register given the bit offset, mask, and list of possible values.

Parameters

- **offset** (*int*) – The offset bits of the value field in the register, i.e. the distance from LSB
- **mask** (*int*) – integer representation of the value field mask (w/o offset)
- **values** (*OrderedDict*) – The possible values that the field can take.

unpack(*cfg*)

Extracts the ValueField's setting from cfg & returns the result in a human readable form.

Parameters **cfg** (*int*) – An integer representing a possible configuration value for the register

extract(*cfg*) → int

Extracts setting from passed 16-bit config & returns integer representation.

Parameters **cfg** (*int*) – An integer representing a possible configuration value for the register

pack(*value*) → int

Takes a human-readable ValueField setting and returns the corresponding bit-shifted integer.

Parameters **value** (*int*) – The integer representation of *value* bit-shifted by the ValueField's offset

Returns The integer representation of the ValueField setting according to *value*

Return type int

insert(*cfg*, *value*) → int

Validates and performs bitwise replacement with the human-readable ValueField setting and integer representation of the register configuration.

Parameters

- **cfg** (*int*) – An integer representing a possible configuration value for the register
- **value** (*object*) – The human readable representation of the desired ValueField setting. Must match a value in ValueField._values; if not, throws a ValueError

Returns

The integer representation of the Register's configuration with the value of ValueField patched according the *value*

Return type int

class vent.io.devices.base.**SPIDevice**(*channel*, *baudrate*, *pig=None*)

Bases: *vent.io.devices.base.IODeviceBase*

A class wrapper for pigpio SPI handles. Not really implemented.

Instantiates an SPIDevice on SPI *channel* with *baudrate* and, optionally, *pigpio.pi* = *pig*.

Parameters

- **channel** (*int*) – The SPI channel
- **baudrate** (*int*) – SPI baudrate

- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Methods

__init__(channel, baudrate[, pig])	Instantiates an SPIDevice on SPI *channel* with *baudrate* and, optionally, *pigpio.pi = pig*.
_close()	Extends superclass method.
_open(channel, baudrate)	Opens an SPI connection and sets the pigpiod handle.

__init__ (*channel*, *baudrate*, *pig=None*)
Instantiates an SPIDevice on SPI *channel* with *baudrate* and, optionally, *pigpio.pi = pig*.

Parameters

- **channel** (*int*) – The SPI channel
- **baudrate** (*int*) – SPI baudrate
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

_open (*channel*, *baudrate*)
Opens an SPI connection and sets the pigpiod handle.

Parameters

- **channel** (*int*) – The SPI channel
- **baudrate** (*int*) – SPI baudrate

_close ()
Extends superclass method. Checks that pigpiod is connected and if a handle has been set - if so, closes an SPI connection.

class vent.io.devices.base.**ADS1115** (*address=72*, *i2c_bus=1*, *pig=None*)
Bases: *vent.io.devices.base.I2CDevice*

ADS1115 16 bit, 4 Channel Analog to Digital Converter. Datasheet:

Attributes

USER_CONFIGURABLE_FIELDS	Note: The Conversion Register is read-only and contains a 16bit representation of the requested value (provided the conversion is ready).
_CONFIG_FIELDS	Built-in immutable sequence.
_CONFIG_VALUES	Built-in immutable sequence.
_DEFAULT_ADDRESS	int([x]) -> integer
_DEFAULT_VALUES	dict() -> new empty dictionary
_POINTER_FIELDS	Built-in immutable sequence.
_POINTER_VALUES	Config Register (R/W)
_TIMEOUT	Address Pointer Register (write-only)
cfg	Returns the contents (as a 16-bit unsigned integer) of the configuration that will be written to the config register when read_conversion() is next called.
config	Returns the Register object of the config register.

Methods

__init__([address, i2c_bus, pig])	Initializes registers: Pointer register is write only, config is R/W.
_read_conversion(**kwargs)	Backend for read_conversion.
_read_last_cfg()	Reads the config register and returns the contents as a 16-bit unsigned integer; updates internal record _last_cfg.
_ready()	Return status of ADC conversion; True indicates the conversion is complete and the results ready to be read.
print_config()	Returns the human-readable configuration for the next read.
read_conversion(**kwargs)	Returns a voltage (expressed as a float) corresponding to a channel on the ADC.

http://www.ti.com/lit/ds/symlink/ads1114.pdf?ts=1587872241912

Default Values: Default configuration for vent: 0xC3E3 Default configuration on power-up: 0x8583

Initializes registers: Pointer register is write only, config is R/W. Sets initial value of _last_cfg to what is actually on the ADS.Packs default settings into _cfg, but does not actually write to ADC - that occurs when read_conversion() is called.

Parameters

- **address** (*int*) – I2C address of the device. (e.g., *i2c_address=0x48*)
- **i2c_bus** (*int*) – The I2C bus to use. Should probably be set to 1 on Raspberry Pi.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

_DEFAULT_ADDRESS = 72

_DEFAULT_VALUES = {'DR': 860, 'MODE': 'SINGLE', 'MUX': 0, 'PGA': 4.096}

_TIMEOUT = 1

Address Pointer Register (write-only)

_POINTER_FIELDS = ('P',)

_POINTER_VALUES = (('CONVERSION', 'CONFIG', 'LO_THRESH', 'HIGH_THRESH'),)

Config Register (R/W)

_CONFIG_FIELDS = ('OS', 'MUX', 'PGA', 'MODE', 'DR', 'COMP_MODE', 'COMP_POL', 'COMP_LAT

_CONFIG_VALUES = (('NO_EFFECT', 'START_CONVERSION'), ((0, 1), (0, 3), (1, 3), (2, 3),

USER_CONFIGURABLE_FIELDS = ('MUX', 'PGA', 'MODE', 'DR')

Note: The Conversion Register is read-only and contains a 16bit representation of the requested value (provided the conversion is ready).

The Lo-thresh & Hi-thresh Registers are not Utilized here. However, their function and usage are described in the datasheet. Should you want to extend the functionality implemented here.

__init__(*address=72*, *i2c_bus=1*, *pig=None*)

Initializes registers: Pointer register is write only, config is R/W. Sets initial value of _last_cfg to what is actually on the ADS.Packs default settings into _cfg, but does not actually write to ADC - that occurs when read_conversion() is called.

Parameters

- **address** (*int*) – I2C address of the device. (e.g., *i2c_address=0x48*)
- **i2c_bus** (*int*) – The I2C bus to use. Should probably be set to 1 on Raspberry Pi.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

read_conversion(***kwargs*) → float

Returns a voltage (expressed as a float) corresponding to a channel on the ADC. The channel to read from, along with the gain, mode, and sample rate of the conversion may be may be specified as optional parameters. If read_conversion() is called with no parameters, the resulting voltage corresponds to the channel last read from and the same conversion settings.

Parameters

- **MUX** – The pin to read from in single channel mode: e.g., *0, 1, 2, 3* or, a tuple of pins over which to make a differential reading. e.g., *(0, 1), (0, 3), (1, 3), (2, 3)*
- **PGA** – The full scale voltage (FSV) corresponding to a programmable gain setting. e.g., *(6.144, 4.096, 2.048, 1.024, 0.512, 0.256, 0.256, 0.256)*
- **MODE** – Whether to set the ADC to continuous conversion mode, or operate in single-shot mode. e.g., *'CONTINUOUS', 'SINGLE'*
- **DR** – The data rate to make the conversion at; units: samples per second. e.g., *8, 16, 32, 64, 128, 250, 475, 860*

print_config() → collections.OrderedDict

Returns the human-readable configuration for the next read.

Returns an ordered dictionary of the form {field: value}, ordered from MSB -> LSB

Return type OrderedDict

property config

Returns the Register object of the config register.

Returns The Register object initialized for the ADS1115.

Return type vent.io.devices.I2CDevice.Register

property cfg

Returns the contents (as a 16-bit unsigned integer) of the configuration that will be written to the config register when read_conversion() is next called.

_read_conversion(***kwargs*) → int

Backend for read_conversion. Returns the contents of the 16-bit conversion register as an unsigned integer.

If no parameters are passed, one of two things can happen:

1) If the ADC is in single-shot (mode='SINGLE') conversion mode, _last_cfg is written to the config register; once the ADC indicates it is ready, the contents of the conversion register are read and the result is returned.
2) If the ADC is in CONTINUOUS mode, the contents of the conversion register are read immediately and returned.

If any of channel, gain, mode, or data_rate are specified as parameters, a new _cfg is packed and written to the config register; once the ADC indicates it is ready, the contents of the conversion register are read and the result is returned.

Note: In continuous mode, data can be read from the conversion register of the ADS1115 at any time and always reflects the most recently completed conversion. So says the datasheet.

Parameters ****kwargs** – see documentation of vent.io.devices.ADS1115.read_conversion

_read_last_cfg() → int
Reads the config register and returns the contents as a 16-bit unsigned integer; updates internal record _last_cfg.

_ready() → bool
Return status of ADC conversion; True indicates the conversion is complete and the results ready to be read.

class vent.io.devices.base.**ADS1015**(*address=72*, *i2c_bus=1*, *pig=None*)
Bases: *vent.io.devices.base.ADS1115*

ADS1015 16 bit, 4 Channel Analog to Digital Converter. Datasheet:

Attributes

USER_CONFIGURABLE_FIELDS	Built-in immutable sequence.
_CONFIG_FIELDS	Built-in immutable sequence.
_CONFIG_VALUES	Built-in immutable sequence.
_DEFAULT_ADDRESS	int([x]) -> integer
_DEFAULT_VALUES	Address Pointer Register (write-only)
_POINTER_FIELDS	Built-in immutable sequence.
_POINTER_VALUES	Config Register (R/W)

Methods

__init__([address, i2c_bus, pig])	See: vent.io.devices.ADS1115.__init__

http://www.ti.com/lit/ds/symlink/ads1015.pdf?&ts=1589228228921

Basically the same device as the ADS1115, except has 12 bit resolution instead of 16, and has different (faster) data rates. The difference in data rates is handled by overloading _CONFIG_VALUES. The difference in resolution is irrelevant for implementation.

See: vent.io.devices.ADS1115.__init__

_DEFAULT_ADDRESS = 72

_DEFAULT_VALUES = {'DR': 3300, 'MODE': 'SINGLE', 'MUX': 0, 'PGA': 4.096}
Address Pointer Register (write-only)

_POINTER_FIELDS = ('P',)

_POINTER_VALUES = (('CONVERSION', 'CONFIG', 'LO_THRESH', 'HIGH_THRESH'),)
Config Register (R/W)

_CONFIG_FIELDS = ('OS', 'MUX', 'PGA', 'MODE', 'DR', 'COMP_MODE', 'COMP_POL', 'COMP_LAT

_CONFIG_VALUES = (('NO_EFFECT', 'START_CONVERSION'), ((0, 1), (0, 3), (1, 3), (2, 3),

USER_CONFIGURABLE_FIELDS = ('MUX', 'PGA', 'MODE', 'DR')

__init__(*address=72*, *i2c_bus=1*, *pig=None*)
See: vent.io.devices.ADS1115.__init__

vent.io.devices.base.**be16_to_native**(*data*, *signed=False*) → int
Unpacks a bytes-like object respecting big-endianness of outside world and returns an int according to signed.

Parameters

- **data** – bytes-like object. The data to be unpacked & converted
- **signed** (*bool*) – Whether or not *data* is signed

vent.io.devices.base.**native16_to_be**(*word*, *signed=False*) → bytes
Packs an int into bytes after swapping endianness.

Parameters

- **signed** (*bool*) – Whether or not *data* is signed
- **word** (*int*) – The integer representation to converted and packed into bytes

7.1.1.3 vent.io.devices.pins module

Classes

PWMOutput(pin[, initial_duty, frequency, pig])	A pin configured to output a PWM signal.
Pin(pin[, pig])	Base Class wrapping pigpio methods for interacting with GPIO pins on the raspberry pi.

class vent.io.devices.pins.**Pin**(*pin*, *pig=None*)
Bases: *vent.io.devices.base.IODeviceBase*

Base Class wrapping pigpio methods for interacting with GPIO pins on the raspberry pi. Subclasses include InputPin, OutputPin; along with any specialized pins or specific devices defined in vent.io.actuators & vent.io.sensors (note: actuators and sensors do not need to be tied to a GPIO pin and may instead be interfaced through an ADC or I2C).

This is an abstract base class. The subclasses InputPin and OutputPin extend Pin into a usable form.

Inherits attributes and methods from IODeviceBase.

Parameters

- **pin** (*int*) – The number of the pin to use
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Attributes

_PIGPIO_MODES	dict() -> new empty dictionary
mode	The currently active pigpio mode of the pin.

Methods

__init__(pin[, pig])	Inherits attributes and methods from IODeviceBase.
read()	Returns the value of the pin: usually 0 or 1 but can be overridden by subclass.
toggle()	If pin is on, turn it off.
write(value)	Sets the value of the Pin.

_PIGPIO_MODES = {'ALT0': 4, 'ALT1': 5, 'ALT2': 6, 'ALT3': 7, 'ALT4': 3, 'ALT5':

__init__(*pin*, *pig=None*)
Inherits attributes and methods from IODeviceBase.

Parameters

- **pin** (*int*) – The number of the pin to use
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

property mode
The currently active pigpio mode of the pin.

toggle()
If pin is on, turn it off. If it's off, turn it on. Do not raise a warning when pin is read in this way.

read() → int
Returns the value of the pin: usually 0 or 1 but can be overridden by subclass.

write(*value*)
Sets the value of the Pin. Usually 0 or 1 but behavior differs for some subclasses.

Parameters **value** – The value to write to the pin. Can be either *1* to turn on the pin or *0* to turn it off.

class vent.io.devices.pins.**PWMOutput**(*pin*, *initial_duty=0*, *frequency=None*, *pig=None*)
Bases: *vent.io.devices.pins.Pin*

A pin configured to output a PWM signal. Can be configured to use either a hardware-generated or software-generated signal. Overrides parent methods read() and write().

Inherits attributes from parent Pin, then sets PWM frequency & initial duty (use defaults if None)

Parameters

- **pin** (*int*) – The number of the pin to use. Hardware PWM pins are 12, 13, 18, and 19.
- **initial_duty** (*float*) – The initial duty cycle of the pin. Must be between 0 and 1.
- **frequency** (*float*) – The PWM frequency to use.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Attributes

_DEFAULT_FREQUENCY	int([x]) -> integer
_DEFAULT_SOFT_FREQ	int([x]) -> integer
_HARDWARE_PWM_PINS	Built-in immutable sequence.
duty	Returns the PWM duty cycle (pulled straight from pigpiod) mapped to the range [0-1]
frequency	Return the current PWM frequency active on the pin.
hardware_enabled	Return true if this is a hardware-enabled PWM pin; False if not.

Methods

_PWMOutput__hardware_pwm(frequency, duty)	Used for pins where hardware pwm is available.
_PWMOutput__pwm(frequency, duty)	Sets a PWM frequency and duty using either hardware or software generated PWM according to the value of *self.hardware_enabled*.

continues on next page

Table 16 – continued from previous page

_PWMOutput__software_pwm(frequency, duty)	Used for pins where hardware PWM is NOT available.
__init__(pin[, initial_duty, frequency, pig])	Inherits attributes from parent Pin, then sets PWM frequency & initial duty (use defaults if None)
_duty()	Returns the pigpio integer representation of the duty cycle.
read()	Overridden to return duty cycle instead of reading the value on the pin.
write(value)	Overridden to write duty cycle.

_DEFAULT_FREQUENCY = 20000

_DEFAULT_SOFT_FREQ = 2000

_HARDWARE_PWM_PINS = (12, 13, 18, 19)

__init__(*pin*, *initial_duty=0*, *frequency=None*, *pig=None*)
Inherits attributes from parent Pin, then sets PWM frequency & initial duty (use defaults if None)

Parameters

- **pin** (*int*) – The number of the pin to use. Hardware PWM pins are 12, 13, 18, and 19.
- **initial_duty** (*float*) – The initial duty cycle of the pin. Must be between 0 and 1.
- **frequency** (*float*) – The PWM frequency to use.
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

property hardware_enabled
Return true if this is a hardware-enabled PWM pin; False if not. The Raspberry Pi only supports hardware-generated PWM on pins 12, 13, 18, and 19, so generally *hardware_enabled* will be true if this is one of those, and false if it is not. However, *hardware_enabled* can also by dynamically set to False if for some reason pigpio is unable to start a hardware PWM (i.e. if the clock is unavailable or in use or something)

property frequency
Return the current PWM frequency active on the pin.

_duty() → int
Returns the pigpio integer representation of the duty cycle.

property duty
Returns the PWM duty cycle (pulled straight from pigpiod) mapped to the range [0-1]

read() → float
Overridden to return duty cycle instead of reading the value on the pin.

write(*value*)
Overridden to write duty cycle.

Parameters **value** (*float*) – See *PWMOutput.duty*

_PWMOutput__hardware_pwm(*frequency*, *duty*)
Used for pins where hardware pwm is available. -Tries to write a hardware pwm. result == 0 if it succeeds. -Sets hardware_enabled flag to indicate success or failure

Parameters

- **frequency** – A new PWM frequency to use.

- **duty** (*int*) – The PWM duty cycle to set. Must be between 0 and 1.

_PWMOutput__pwm(*frequency*, *duty*)
Sets a PWM frequency and duty using either hardware or software generated PWM according to the value of *self.hardware_enabled*. If hardware_enabled, starts a hardware pwm with the requested duty. If not hardware_enabled, or if there is a problem setting a hardware generated PWM, starts a software PWM.

Parameters

- **frequency** (*float*) – A new PWM frequency to use.
- **duty** (*float*) – The PWM duty cycle to set. Must be between 0 and 1.

_PWMOutput__software_pwm(*frequency*, *duty*)
Used for pins where hardware PWM is NOT available.

Parameters

- **frequency** – A new PWM frequency to use.
- **duty** (*int*) – A pigpio integer representation of duty cycle

7.1.1.4 vent.io.devices.sensors module

Classes

AnalogSensor(adc, **kwargs)	Generalized class describing an analog sensor attached to the ADS1115 analog to digital converter.
DLiteSensor(adc, **kwargs)	D-Lite flow sensor setup.
SFM3200([address, i2c_bus, pig])	I2C Inspiratory flow sensor manufactured by Sensirion AG.
Sensor()	Abstract base Class describing generalized sensors.
SimSensor([low, high, pig])	TODO Stub simulated sensor.

class vent.io.devices.sensors.**Sensor**
Bases: abc.ABC

Abstract base Class describing generalized sensors. Defines a mechanism for limited internal storage of recent observations and methods to pull that data out for external use.

Upon creation, calls update() to ensure that if get is called there will be something to return. **Attributes**

_DEFAULT_STORED_OBSERVATIONS	int([x]) -> integer
data	Returns all Locally-stored observations.
maxlen_data	Returns the number of observations kept in the Sensor's internal ndarray.

Methods

__init__()	Upon creation, calls update() to ensure that if get is called there will be something to return.
_clear()	Resets the sensors internal memory.
_convert(raw)	Converts a raw reading from a sensor in whatever format the device communicates with into a meaningful result.

continues on next page

Table 19 – continued from previous page

_raw_read()	Requests a new observation from the device and returns the raw result in whatever format/units the device communicates with.
_read()	Calls _raw_read and scales the result before returning it.
_verify(value)	Validate reading and throw exception/alarm if sensor does not appear to be working correctly.
age()	Returns the time in seconds since the last sensor update, or -1 if never updated.
get([average])	Return the most recent sensor reading, or an average of readings since last get().
reset()	Resets the sensors internal memory.
update()	Make a sensor reading, verify that it makes sense and store the result internally.

_DEFAULT_STORED_OBSERVATIONS = 128

__init__()
Upon creation, calls update() to ensure that if get is called there will be something to return.

update() → float
Make a sensor reading, verify that it makes sense and store the result internally. Returns True if reading was verified and False if something went wrong.

get(*average=False*) → float
Return the most recent sensor reading, or an average of readings since last get(). Clears internal memory so as not to have stale data.

age() → float
Returns the time in seconds since the last sensor update, or -1 if never updated.

reset()
Resets the sensors internal memory. May be overloaded by subclasses to extend functionality specific to a device.

_clear()
Resets the sensors internal memory.

property data
Returns all Locally-stored observations.

Returns An array of timestamped observations arranged oldest to newest.

Return type np.array

property maxlen_data
Returns the number of observations kept in the Sensor's internal ndarray. Once the ndarray has been filled, the sensor begins overwriting the oldest elements of the ndarray with new observations such that the size of the internal storage stays constant.

_read() → float
Calls _raw_read and scales the result before returning it.

abstract _verify(*value*)
Validate reading and throw exception/alarm if sensor does not appear to be working correctly.

abstract _convert(*raw*)
Converts a raw reading from a sensor in whatever format the device communicates with into a meaningful result.

abstract _raw_read()
Requests a new observation from the device and returns the raw result in whatever format/units the device communicates with.

_abc_impl = <_abc_data object>

class vent.io.devices.sensors.**AnalogSensor**(*adc*, ***kwargs*)
Bases: *vent.io.devices.sensors.Sensor*

Generalized class describing an analog sensor attached to the ADS1115 analog to digital converter. Inherits from the sensor base class and extends with functionality specific to analog sensors attached to the ADS1115. If instantiated without a subclass, conceptually represents a voltmeter with a normalized output.

Links analog sensor on the ADC with configuration options specified. If no options are specified, it assumes the settings currently on the ADC.

Parameters

- **adc** (*vent.io.devices.ADS1115*) – The adc object to which the AnalogSensor is attached
- ****kwargs** – *field=value* - see vent.io.devices.ADS1115 for additional documentation. Strongly suggested to specify *MUX=adc_pin* here unless you know what you're doing.

Attributes

_DEFAULT_CALIBRATION	dict() -> new empty dictionary
_DEFAULT_offset_voltage	int([x]) -> integer
_DEFAULT_output_span	int([x]) -> integer

Methods

__init__(adc, **kwargs)	Links analog sensor on the ADC with configuration options specified.
_check_and_set_attr(**kwargs)	Checks to see if arguments passed to __init__ are recognized as user configurable or calibration fields.
_convert(raw)	Scales raw voltage into the range 0 - 1.
_fill_attr()	Examines self to see if there are any fields identified as user configurable or calibration that have not been write (i.e.
_raw_read()	Builds kwargs from configured fields to pass along to adc, then calls adc.read_conversion(), which returns a raw voltage.
_read()	Returns a value in the range of 0 - 1 corresponding to a fraction of the full input range of the sensor.
_verify(value)	Checks to make sure sensor reading was indeed in [0, 1].
calibrate(**kwargs)	Sets the calibration of the sensor, either to the values contained in the passed tuple or by some routine; the current routine is pretty rudimentary and only calibrates offset voltage.

_DEFAULT_offset_voltage = 0

_DEFAULT_output_span = 5

_DEFAULT_CALIBRATION = {'conversion_factor': 1, 'offset_voltage': 0, 'output_span':

__init__(*adc*, ***kwargs*)
Links analog sensor on the ADC with configuration options specified. If no options are specified, it assumes the settings currently on the ADC.

Parameters

- **adc** (*vent.io.devices.ADS1115*) – The adc object to which the AnalogSensor is attached
- ****kwargs** – *field=value* - see vent.io.devices.ADS1115 for additional documentation. Strongly suggested to specify *MUX=adc_pin* here unless you know what you're doing.

calibrate(***kwargs*)
Sets the calibration of the sensor, either to the values contained in the passed tuple or by some routine; the current routine is pretty rudimentary and only calibrates offset voltage.

Parameters ****kwargs** – calibration_field=value, where calibration field is one of the following: 'offset_voltage', output_span' or 'conversion_factor'

_read() → float
Returns a value in the range of 0 - 1 corresponding to a fraction of the full input range of the sensor.

_verify(*value*) → bool
Checks to make sure sensor reading was indeed in [0, 1].

Parameters **value** (*float*) – Sensor reading to validate

_convert(*raw*) → float
Scales raw voltage into the range 0 - 1.

Parameters **raw** (*float*) – The raw sensor reading to convert.

_raw_read() → float
Builds kwargs from configured fields to pass along to adc, then calls adc.read_conversion(), which returns a raw voltage.

_fill_attr()
Examines self to see if there are any fields identified as user configurable or calibration that have not been write (i.e. were not passed to __init__ as **kwargs). If a field is missing, grabs the default value either from the ADC or from _DEFAULT_CALIBRATION and sets it as an attribute.

_check_and_set_attr(***kwargs*)
Checks to see if arguments passed to __init__ are recognized as user configurable or calibration fields. If so, write the value as an attribute like: self.KEY = VALUE. Keeps track of how many attributes are write in this way; if at the end there unknown arguments leftover, raises a TypeError; otherwise, calls _fill_attr() to fill in fields that were not passed as arguments.

Parameters ****kwargs** – *field=value* - see vent.io.devices.ADS1115 for additional documentation

_abc_impl = <_abc_data object>

class vent.io.devices.sensors.**DLiteSensor**(*adc*, ***kwargs*)
Bases: *vent.io.devices.sensors.AnalogSensor*

D-Lite flow sensor setup. This consists of the GE D-Lite sensor configured with vacuum lines running to an analog differential pressure sensor.

Links analog sensor on the ADC with configuration options specified. If no options are specified, it assumes the settings currently on the ADC.

Parameters

- **adc** (*vent.io.devices.ADS1115*) – The adc object to which the AnalogSensor is attached
- ****kwargs** – *field=value* - see vent.io.devices.ADS1115 for additional documentation. Strongly suggested to specify *MUX=adc_pin* here unless you know what you're doing.

Methods

_convert(raw)	Converts the raw differential voltage signal to a measurement of flow in liters-per-minute (LPM).
calibrate(**kwargs)	Do not run a calibration routine.

_convert(*raw*) → float
Converts the raw differential voltage signal to a measurement of flow in liters-per-minute (LPM).

We calibrate the D-Lite flow readings using the (pre-calibrated) Sensirion flow sensor (see SFM3200).

Parameters **raw** (*float*) – The raw sensor reading to convert.

calibrate(***kwargs*)
Do not run a calibration routine. Overrides attempt to calibrate.

_abc_impl = <_abc_data object>

class vent.io.devices.sensors.**SFM3200**(*address=64*, *i2c_bus=1*, *pig=None*)
Bases: *vent.io.devices.sensors.Sensor*, *vent.io.devices.base.I2CDevice*

I2C Inspiratory flow sensor manufactured by Sensirion AG. Range: +/- 250 SLM Datasheet:

Attributes

_DEFAULT_ADDRESS	int([x]) -> integer
_FLOW_OFFSET	int([x]) -> integer
_FLOW_SCALE_FACTOR	int([x]) -> integer

Methods

__init__([address, i2c_bus, pig])	**param address** The I2C Address of the SFM3200 (usually 0x40)
_convert(raw)	Overloaded to replace with device-specific protocol.
_raw_read()	Performs an read on the sensor, converts received bytearray, discards the last two bytes (crc values - could implement in future), and returns a signed int converted from the big endian two complement that remains.
_start()	Device specific:Sends the 'start measurement' command to the sensor.
_verify(value)	No further verification needed for this sensor.
reset()	Extended to add device specific behavior: Asks the sensor to perform a soft reset.

https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/ ... ...
5_Mass_Flow_Meters/Datasheets/Sensirion_Mass_Flow_Meters_SFM3200_Datasheet.pdf

Parameters

- **address** (*int*) – The I2C Address of the SFM3200 (usually 0x40)
- **i2c_bus** (*int*) – The I2C Bus to use (usually *1* on the Raspberry Pi)
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

_DEFAULT_ADDRESS = 64

_FLOW_OFFSET = 32768

_FLOW_SCALE_FACTOR = 120

__init__(*address=64*, *i2c_bus=1*, *pig=None*)

Parameters

- **address** (*int*) – The I2C Address of the SFM3200 (usually 0x40)
- **i2c_bus** (*int*) – The I2C Bus to use (usually *1* on the Raspberry Pi)
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

reset()

Extended to add device specific behavior: Asks the sensor to perform a soft reset. 80 ms soft reset time.

_start()

Device specific:Sends the 'start measurement' command to the sensor. Start-up time once command has been recieved is 'less than 100ms'

_verify(*value*) → bool

No further verification needed for this sensor. Onboard chip handles all that. Could throw in a CRC8 checker instead of discarding them in _convert().

Parameters **value** (*float*) – The sensor reading to verify.

_convert(*raw*) → float

Overloaded to replace with device-specific protocol. Convert raw int to a flow reading having type float with units slm. Implementation differs from parent for clarity and consistency with source material.

Source:

https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/ ... ...
5_Mass_Flow_Meters/Application_Notes/Sensirion_Mass_Flo_Meters_SFM3xxx_I2C_Functional_Description.

Parameters **raw** (*int*) – The raw value read from the SFM3200

_raw_read() → int

Performs an read on the sensor, converts received bytearray, discards the last two bytes (crc values - could implement in future), and returns a signed int converted from the big endian two complement that remains.

_abc_impl = <_abc_data object>

class vent.io.devices.sensors.**SimSensor**(*low=0*, *high=100*, *pig=None*)

Bases: *vent.io.devices.sensors.Sensor*

TODO Stub simulated sensor.

Parameters

- **low** – Lower-bound of possible sensor values

- **high** – Upper-bound of possible sensor values
- **pig** (*PigpioConnection*) – Ignored.

Methods

__init__([low, high, pig])	TODO Stub simulated sensor.
_convert(raw)	Does nothing for a simulated sensor.
_raw_read()	Initializes randomly, otherwise does a random walk-ish thing.
_verify(value)	Usually verifies sensor readings but occasionally misbehaves.

__init__ (*low=0*, *high=100*, *pig=None*)
TODO Stub simulated sensor.

Parameters

- **low** – Lower-bound of possible sensor values
- **high** – Upper-bound of possible sensor values
- **pig** (*PigpioConnection*) – Ignored.

_verify (*value*) → bool
Usually verifies sensor readings but occasionally misbehaves.

Parameters **value** (*float*) – The sensor reading to verify

_convert (*raw*) → float
Does nothing for a simulated sensor. Returns what it is passed.

Parameters **raw** (*float*) – The raw value to convert

_abc_impl = <_abc_data object>

_raw_read () → float
Initializes randomly, otherwise does a random walk-ish thing.

7.1.1.5 vent.io.devices.valves module

Classes

OnOffValve(pin[, form, pig])	An extension of vent.io.iobase.Pin which uses valve terminology for its methods.
PWMControlValve(pin[, form, frequency, . . .])	An extension of PWMOutput which incorporates linear compensation of the valve's response.
SimControlValve([pin, form, frequency, . . .])	stub: a simulated linear control valve
SimOnOffValve([pin, form, pig])	stub: a simulated on/off valve
SolenoidBase([form])	An abstract baseclass that defines methods using valve terminology.

class vent.io.devices.valves.**SolenoidBase** (*form='Normally Closed'*)
Bases: abc.ABC

An abstract baseclass that defines methods using valve terminology. Also allows configuring both normally _open and normally closed valves (called the "form" of the valve).

Parameters **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally*

Closed

Attributes

_FORMS	dict() -> new empty dictionary
form	Returns the human-readable form of the valve.
is_open	Returns True if valve is open, False if it is closed

Methods

__init__([form])	**param form** The form of the solenoid; can be either *Normally Open* or *Normally Closed*
close()	De-energizes valve if Normally Closed.
open()	Energizes valve if Normally Closed.

_FORMS = {'Normally Closed': 0, 'Normally Open': 1}

__init__(*form='Normally Closed'*)

Parameters **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*

property form
Returns the human-readable form of the valve.

abstract open()
Energizes valve if Normally Closed. De-energizes if Normally Open.

abstract close()
De-energizes valve if Normally Closed. Energizes if Normally Open.

abstract property is_open
Returns True if valve is open, False if it is closed

_abc_impl = <_abc_data object>

class vent.io.devices.valves.**OnOffValve**(*pin*, *form='Normally Closed'*, *pig=None*)
Bases: *vent.io.devices.valves.SolenoidBase*, *vent.io.devices.pins.Pin*

An extension of vent.io.iobase.Pin which uses valve terminology for its methods. Also allows configuring both normally _open and normally closed valves (called the "form" of the valve).

Parameters

- **pin** (*int*) – The number of the pin to use
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Attributes

_FORMS	dict() -> new empty dictionary

continues on next page

Table 29 – continued from previous page

is_open	Implements parent's abstractmethod; returns True if valve is open, False if it is closed

Methods

__init__(pin[, form, pig])	**param pin** The number of the pin to use
close()	De-energizes valve if Normally Closed.
open()	Energizes valve if Normally Closed.

_FORMS = {'Normally Closed': 0, 'Normally Open': 1}

__init__(*pin*, *form='Normally Closed'*, *pig=None*)

Parameters

- **pin** (*int*) – The number of the pin to use
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

open()
Energizes valve if Normally Closed. De-energizes if Normally Open.

close()
De-energizes valve if Normally Closed. Energizes if Normally Open.

property is_open
Implements parent's abstractmethod; returns True if valve is open, False if it is closed

_abc_impl = <_abc_data object>

class vent.io.devices.valves.**PWMControlValve**(*pin*, *form='Normally Closed'*, *frequency=None*, *response=None*, *pig=None*)

Bases: *vent.io.devices.valves.SolenoidBase*, *vent.io.devices.pins.PWMOutput*

An extension of PWMOutput which incorporates linear compensation of the valve's response.

Parameters

- **pin** (*int*) – The number of the pin to use
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **frequency** (*float*) – The PWM frequency to use.
- **response** (*str*) – "/path/to/response/curve/file"
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

Methods

__init__(pin[, form, frequency, response, pig])	**param pin** The number of the pin to use
_load_valve_response(response_path)	Loads and applies a response curve of the form *f(setpoint) = duty*.
close()	Implements parent's abstractmethod; fully closes the valve
inverse_response(duty_cycle[, rising])	Inverse of response.
open()	Implements parent's abstractmethod; fully opens the valve
response(setpoint[, rising])	Setpoint takes a value in the range (0,100) so as not to confuse with duty cycle, which takes a value in the range (0,1).

Attributes

is_open	Implements parent's abstractmethod; returns True if valve is open, False if it is closed
setpoint	The linearized setpoint corresponding to the current duty cycle according to the valve's response curve

__init__(*pin*, *form='Normally Closed'*, *frequency=None*, *response=None*, *pig=None*)

Parameters

- **pin** (*int*) – The number of the pin to use
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **frequency** (*float*) – The PWM frequency to use.
- **response** (*str*) – "/path/to/response/curve/file"
- **pig** (*PigpioConnection*) – pigpiod connection to use; if not specified, a new one is established

property is_open
Implements parent's abstractmethod; returns True if valve is open, False if it is closed

open()
Implements parent's abstractmethod; fully opens the valve

close()
Implements parent's abstractmethod; fully closes the valve

property setpoint
The linearized setpoint corresponding to the current duty cycle according to the valve's response curve

Returns A number between 0 and 1 representing the current flow as a proportion of maximum

Return type float

response(*setpoint*, *rising=True*)
Setpoint takes a value in the range (0,100) so as not to confuse with duty cycle, which takes a value in the range (0,1). Response curves are specific to individual valves and are to be implemented by subclasses. Different curves are calibrated to 'rising = True' (valves opening) or'rising = False' (valves closing), as different characteristic flow behavior can be observed.

Parameters

- **setpoint** (*float*) – A number between 0 and 1 representing how much to open the valve
- **rising** (*bool*) – Whether or not the requested setpoint is higher than the last (rising = True), or the opposite (Rising = False)

Returns The PWM duty cycle corresponding to the requested setpoint

Return type float

inverse_response(*duty_cycle*, *rising=True*)

Inverse of response. Given a duty cycle in the range (0,1), returns the corresponding linear setpoint in the range (0,100).

Parameters

- **duty_cycle** – The PWM duty cycle
- **rising** (*bool*) – Whether or not the requested setpoint is higher than the last (rising = True), or the opposite (Rising = False)

Returns The setpoint of the valve corresponding to *duty_cycle*

Return type float

_load_valve_response(*response_path*)

Loads and applies a response curve of the form *f(setpoint) = duty*. A response curve maps the underlying PWM duty cycle *duty* onto the normalized variable *setpoint* representing the flow through the valve as a percentage of its maximum.

Flow through a proportional valve may be nonlinear with respect to [PWM] duty cycle, if the valve itself does not include its own electronics to linearize response wrt/ input. Absent on-board compensation of response, a proportional solenoid with likely not respond [flow] at all below some minimum threshold duty cycle. Above this threshold, the proportional valve begins to open and its response resembles a sigmoid: just past the threshold there is a region where flow increases exponentially wrt/ duty cycle, this is followed by a region of pseudo-linear response that begins to taper off, eventually approaching the valve's maximum flow asymptotically as the duty cycle approaches 100% and the valve opens fully.

Parameters **response_path** – 'path/to/binary/response/file' - if response_path is None, defaults to *setpoint = duty*

_abc_impl = <_abc_data object>

class vent.io.devices.valves.**SimOnOffValve**(*pin=None*, *form='Normally Closed'*, *pig=None*)

Bases: *vent.io.devices.valves.SolenoidBase*

stub: a simulated on/off valve

Args: form (str): The form of the solenoid; can be either *Normally Open* or *Normally Closed* **Attributes**

is_open	Returns True if valve is open, False if it is closed

Methods

close()	De-energizes valve if Normally Closed.
open()	Energizes valve if Normally Closed.

open()

Energizes valve if Normally Closed. De-energizes if Normally Open.

close()
De-energizes valve if Normally Closed. Energizes if Normally Open.

property is_open
Returns True if valve is open, False if it is closed

_abc_impl = <_abc_data object>

class vent.io.devices.valves.**SimControlValve**(*pin=None*, *form='Normally Closed'*, *frequency=None*, *response=None*, *pig=None*)

Bases: *vent.io.devices.valves.SolenoidBase*

stub: a simulated linear control valve

Parameters

- **pin** (*int*) – (unused for sim)
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **frequency** (*float*) – (unused for sim)
- **response** (*str*) – (unused for sim) # TODO implement this (requires refactor)
- **pig** (*PigpioConnection*) – (unused for sim)

Methods

__init__([pin, form, frequency, response, pig])	**param pin** (unused for sim)
close()	Implements parent's abstractmethod; fully closes the valve
open()	Implements parent's abstractmethod; fully opens the valve

Attributes

is_open	Implements parent's abstractmethod; returns True if valve is open, False if it is closed
setpoint	The requested linearized set-point of the valve.

__init__(*pin=None*, *form='Normally Closed'*, *frequency=None*, *response=None*, *pig=None*)

Parameters

- **pin** (*int*) – (unused for sim)
- **form** (*str*) – The form of the solenoid; can be either *Normally Open* or *Normally Closed*
- **frequency** (*float*) – (unused for sim)
- **response** (*str*) – (unused for sim) # TODO implement this (requires refactor)
- **pig** (*PigpioConnection*) – (unused for sim)

property is_open

Implements parent's abstractmethod; returns True if valve is open, False if it is closed FIXME: Needs refactor; duplicate property to PWMControlValve.is_open

_abc_impl = <_abc_data object>

open()
Implements parent's abstractmethod; fully opens the valve FIXME: Needs refactor; duplicate method to PWMControlValve.open()

close()
Implements parent's abstractmethod; fully closes the valve FIXME: Needs refactor; duplicate method to PWMControlValve.close()

property setpoint
The requested linearized set-point of the valve.

Returns A number between 0 and 1 representing the current flow as a proportion of maximum

Return type float

7.1.1.6 Module contents

A module for ventilator hardware device drivers

7.2 Submodules

7.3 vent.io.hal module

Module for interacting with physical and/or simulated devices installed on the ventilator.

Classes

Hal([config_file])	Hardware Abstraction Layer for ventilator hardware.

class vent.io.hal.**Hal**(*config_file='vent/io/config/devices.ini'*)
Bases: object

Hardware Abstraction Layer for ventilator hardware. Defines a common API for interacting with the sensors & actuators on the ventilator. The types of devices installed on the ventilator (real or simulated) are specified in a configuration file.

Initializes HAL from config_file. For each section in config_file, imports the class <type> from module <module>, and sets attribute self.<section> = <type>(**opts), where opts is a dict containing all of the options in <section> that are not <type> or <section>. For example, upon encountering the following entry in config_file.ini:

[adc] type = ADS1115 module = devices i2c_address = 0x48 i2c_bus = 1

The Hal will:

1) **Import vent.io.devices.ADS1115 (or ADS1015) as a local variable:** class_ = getattr(import_module('.devices', 'vent.io'), 'ADS1115')
2) **Instantiate an ADS1115 object with the arguments defined in config_file and set it as an attribute:** self._adc = class_(pig=self.-pig,address=0x48,i2c_bus=1)

Note: RawConfigParser.optionxform() is overloaded here s.t. options are case sensitive (they are by default case insensitive). This is necessary due to the kwarg MUX which is so named for consistency with the config registry documentation in the ADS1115 datasheet. For example, A P4vMini pressure_sensor on pin A0 (MUX=0) of the ADC is passed arguments like:

analog_sensor = AnalogSensor(pig=self._pig, adc=self._adc, MUX=0, offset_voltage=0.25, output_span = 4.0, conversion_factor=2.54*20

)

Note: ast.literal_eval(opt) interprets integers, 0xFF, (a, b) etc. correctly. It does not interpret strings correctly, nor does it know 'adc' -> self._adc; therefore, these special cases are explicitly handled.

Methods

`__init__`([config_file])	Initializes HAL from config_file.

Attributes

`aux_pressure`	Returns the pressure from the auxiliary pressure sensor, if so equipped.
`flow_ex`	The measured flow rate expiratory side.
`flow_in`	The measured flow rate inspiratory side.
`pressure`	Returns the pressure from the primary pressure sensor.
`setpoint_ex`	The currently requested flow on the expiratory side as a proportion of the maximum.
`setpoint_in`	The currently requested flow for the inspiratory proportional control valve as a proportion of maximum.

Parameters **config_file** (*str*) – Path to the configuration file containing the definitions of specific components on the ventilator machine. (e.g., config_file = "vent/io/config/devices.ini")

__init__(*config_file='vent/io/config/devices.ini'*)

Initializes HAL from config_file. For each section in config_file, imports the class <type> from module <module>, and sets attribute self.<section> = <type>(**opts), where opts is a dict containing all of the options in <section> that are not <type> or <section>. For example, upon encountering the following entry in config_file.ini:

[adc] type = ADS1115 module = devices i2c_address = 0x48 i2c_bus = 1

The Hal will:

1) **Import vent.io.devices.ADS1115 (or ADS1015) as a local variable:** class_ = getattr(import_module('.devices', 'vent.io'), 'ADS1115')
2) **Instantiate an ADS1115 object with the arguments defined in config_file and set it as an attribute:** self._adc = class_(pig=self.-pig,address=0x48,i2c_bus=1)

Note: RawConfigParser.optionxform() is overloaded here s.t. options are case sensitive (they are by default case insensitive). This is necessary due to the kwarg MUX which is so named for consistency with the config registry documentation in the ADS1115 datasheet. For example, A P4vMini pressure_sensor on pin A0 (MUX=0) of the ADC is passed arguments like:

analog_sensor = AnalogSensor(pig=self._pig, adc=self._adc, MUX=0, offset_voltage=0.25, output_span = 4.0, conversion_factor=2.54*20

)

Note: ast.literal_eval(opt) interprets integers, 0xFF, (a, b) etc. correctly. It does not interpret strings correctly, nor does it know 'adc' -> self._adc; therefore, these special cases are explicitly handled.

Parameters **config_file** (*str*) – Path to the configuration file containing the definitions of specific components on the ventilator machine. (e.g., config_file = "vent/io/config/devices.ini")

property pressure
Returns the pressure from the primary pressure sensor.

property aux_pressure
Returns the pressure from the auxiliary pressure sensor, if so equipped. If a secondary pressure sensor is not defined, raises a RuntimeWarning.

property flow_in
The measured flow rate inspiratory side.

property flow_ex
The measured flow rate expiratory side.

property setpoint_in
The currently requested flow for the inspiratory proportional control valve as a proportion of maximum.

property setpoint_ex
The currently requested flow on the expiratory side as a proportion of the maximum.

7.4 Module contents

CHAPTER

EIGHT

ALARM

8.1 Main Alarm Module

Classes

AlarmSeverity	An enumeration.
AlarmType	An enumeration.

class vent.alarm.**AlarmType**
An enumeration. **Attributes**

HIGH_O2	int([x]) -> integer
HIGH_PEEP	int([x]) -> integer
HIGH_PRESSURE	int([x]) -> integer
HIGH_VTE	int([x]) -> integer
LEAK	int([x]) -> integer
LOW_O2	int([x]) -> integer
LOW_PEEP	int([x]) -> integer
LOW_PRESSURE	int([x]) -> integer
LOW_VTE	int([x]) -> integer
OBSTRUCTION	int([x]) -> integer

LOW_PRESSURE = 1

HIGH_PRESSURE = 2

LOW_VTE = 3

HIGH_VTE = 4

LOW_PEEP = 5

HIGH_PEEP = 6

LOW_O2 = 7

HIGH_O2 = 8

OBSTRUCTION = 9

LEAK = 10

class vent.alarm.**AlarmSeverity**
An enumeration. **Attributes**

HIGH	int([x]) -> integer
LOW	int([x]) -> integer
MEDIUM	int([x]) -> integer
OFF	int([x]) -> integer

HIGH = 3

MEDIUM = 2

LOW = 1

OFF = 0

8.2 Alarm Manager

Classes

Alarm_Manager()	
	vent.alarm.alarm_manager.**active_alarms**

class vent.alarm.alarm_manager.**Alarm_Manager**

Attributes

active_alarms	dict() -> new empty dictionary
callbacks	Built-in mutable sequence.
cleared_alarms	Built-in mutable sequence.
dependencies	dict() -> new empty dictionary
logged_alarms	Built-in mutable sequence.
pending_clears	Built-in mutable sequence.
rules	dict() -> new empty dictionary
snoozed_alarms	dict() -> new empty dictionary

Methods

add_callback(callback)	
check_rule(rule, sensor_values)	
clear_all_alarms()	
deactivate_alarm(alarm)	Mark an alarm's internal active flags and remove from active_alarms
dismiss_alarm(alarm_type, duration)	GUI or other object requests an alarm to be dismissed & deactivated
emit_alarm(alarm_type, severity)	Emit alarm (by calling all callbacks with it).
get_alarm_severity(alarm_type)	
load_rule(alarm_rule)	
load_rules()	
register_alarm(alarm)	Add alarm to registry.
register_dependency(condition, dependency)	Add dependency in a Condition object to be updated when values are changed

continues on next page

Table 6 – continued from previous page

reset()	reset all conditions, callbacks, and other stateful attributes and clear alarms
update(sensor_values)	
update_dependencies(control_setting)	Update Condition objects that update their value according to some control parameter

active_alarms
{AlarmType: Alarm}

Type dict

pending_clears
[AlarmType] list of alarms that have been requested to be cleared

Type list

callbacks
list of callables that accept *Alarm* s when they are raised/altered.

Type list

cleared_alarms
of AlarmType s, alarms that have been cleared but have not dropped back into the 'off' range to enable re-raising

Type list

snoozed_alarms
of AlarmType s : times, alarms that should not be raised because they have been silenced for a period of time

Type dict

If an Alarm_Manager already exists, when initing just return that one

_instance = None

active_alarms: Dict[vent.alarm.AlarmType, vent.alarm.alarm.Alarm] = {}

logged_alarms: List[vent.alarm.alarm.Alarm] = []

dependencies = {}

pending_clears = []

cleared_alarms = []

snoozed_alarms = {}

callbacks = []

rules = {}

load_rules()

load_rule(*alarm_rule: vent.alarm.rule.Alarm_Rule*)

update(*sensor_values: vent.common.message.SensorValues*)

check_rule(*rule: vent.alarm.rule.Alarm_Rule*, *sensor_values: vent.common.message.SensorValues*)

emit_alarm(*alarm_type: vent.alarm.AlarmType*, *severity: vent.alarm.AlarmSeverity*)
Emit alarm (by calling all callbacks with it).

Note: This method emits *and* clears alarms – a cleared alarm is emitted with `AlarmSeverity.OFF`

Parameters

- **alarm_type** (`AlarmType`) –
- **severity** (`AlarmSeverity`) –

deactivate_alarm(*alarm: (<enum 'AlarmType'>, <class 'vent.alarm.alarm.Alarm'>)*)
Mark an alarm's internal active flags and remove from `active_alarms`

Note: This does *not* alert listeners that an alarm has been cleared, for that emit an alarm with AlarmSeverity.OFF

Parameters **alarm** –

Returns:

dismiss_alarm(*alarm_type: vent.alarm.AlarmType*, *duration: float = None*)
GUI or other object requests an alarm to be dismissed & deactivated

GUI will wait until it receives an *emit_alarm* of severity == OFF to remove alarm widgets. If the alarm is not latched

If the alarm is latched, alarm_manager will not decrement alarm severity or emit *OFF* until a) the condition returns to *OFF*, and b) the user dismisses the alarm

Parameters

- **alarm_type** (`AlarmType`) – Alarm to dismiss
- **duration** (*float*) – seconds - amount of time to wait before alarm can be re-raised If a duration is provided, the alarm will not be able to be re-raised

get_alarm_severity(*alarm_type: vent.alarm.AlarmType*)

register_alarm(*alarm: vent.alarm.alarm.Alarm*)
Add alarm to registry.

Parameters **alarm** (`Alarm`) –

register_dependency(*condition: vent.alarm.condition.Condition*, *dependency: dict*)
Add dependency in a Condition object to be updated when values are changed

Parameters

- **condition** –
- **dependency** (*dict*) – either a (ValueName, attribute_name) or optionally also + transformation callable

update_dependencies(*control_setting: vent.common.message.ControlSetting*)
Update Condition objects that update their value according to some control parameter

Parameters **control_setting** (`ControlSetting`) –

Returns:

add_callback(*callback: Callable*)

clear_all_alarms()

reset()
reset all conditions, callbacks, and other stateful attributes and clear alarms

8.3 Alarm

Classes

Alarm(alarm_type, severity, start_time[, ...])	Class used by the program to control and coordinate alarms.

class vent.alarm.alarm.**Alarm**(*alarm_type: vent.alarm.AlarmType*, *severity: vent.alarm.AlarmSeverity*, *start_time: float = None*, *latch=True*, *persistent=True*, *value=None*, *message=None*, *managed=False*)
Class used by the program to control and coordinate alarms.

Parameterized by a Alarm_Rule and managed by Alarm_Manager

Parameters

- **alarm_type** –
- **severity** –
- **start_time** –
- **value** (*int, float*) – optional - numerical value that generated the alarm
- **message** (*str*) – optional - override default text generated by AlarmManager
- **managed** (*bool*) – if created by the alarm_manager, don't register

Methods

__init__(alarm_type, severity, start_time[, ...])	**param alarm_type**
deactivate()	

Attributes

alarm_type	
id_counter	used to generate unique IDs for each alarm
manager	have ta do it this janky way to avoid circular imports
severity	

id_counter = count(0)
used to generate unique IDs for each alarm

Type itertools.count

__init__(*alarm_type: vent.alarm.AlarmType*, *severity: vent.alarm.AlarmSeverity*, *start_time: float = None*, *latch=True*, *persistent=True*, *value=None*, *message=None*, *managed=False*)

Parameters

- **alarm_type** –

- **severity** –
- **start_time** –
- **value** (*int, float*) – optional - numerical value that generated the alarm
- **message** (*str*) – optional - override default text generated by `AlarmManager`
- **managed** (*bool*) – if created by the alarm_manager, don't register

property manager
have ta do it this janky way to avoid circular imports

property severity

property alarm_type

deactivate()

8.4 Condition

Classes

`AlarmSeverityCondition`(alarm_type, severity, ...)	**param alarm_type**

`Condition`(depends, *args, **kwargs)	Base class for specifying alarm test conditions
`CycleAlarmSeverityCondition`(n_cycles, *args, ...)	alarm goes out of range for a specific number of breath cycles
`CycleValueCondition`(n_cycles, *args, **kwargs)	value goes out of range for a specific number of breath cycles
`TimeValueCondition`(time, *args, **kwargs)	value goes out of range for specific amount of time
`ValueCondition`(value_name, limit, mode, ...)	value is greater or lesser than some max/min

Functions

`get_alarm_manager`()

vent.alarm.condition.**get_alarm_manager**()

class vent.alarm.condition.**Condition**(*depends: dict = None*, **args*, ***kwargs*)
Base class for specifying alarm test conditions

Need to be able to condition alarms based on * value ranges * value ranges & durations * levels of other alarms
Methods

`__init__`(depends, *args, **kwargs)	**param depends**

`check`(sensor_values)	
`reset`()	If a condition is stateful, need to provide some method of resetting the state

Attributes

manager

manager
alarm manager, used to get status of alarms

Type vent.alarm.alarm_manager.Alarm_Manager

_child
if another condition is added to this one, store a reference to it

Type Condition

Parameters

- **depends** (*list, dict*) – a list of, or a single dict:

```
{'value_name':ValueName,
'value_attr': attr in ControlMessage,
 'condition_attr',
 optional: transformation: callable)
that declare what values are needed to update
```

- ***args** –
- ****kwargs** –

__init__(*depends: dict = None*, **args*, ***kwargs*)

Parameters

- **depends** (*list, dict*) – a list of, or a single dict:

```
{'value_name':ValueName,
'value_attr': attr in ControlMessage,
 'condition_attr',
 optional: transformation: callable)
that declare what values are needed to update
```

- ***args** –
- ****kwargs** –

property manager

check(*sensor_values*)

reset()
If a condition is stateful, need to provide some method of resetting the state

class vent.alarm.condition.**ValueCondition**(*value_name: vent.common.values.ValueName*, *limit: (<class 'int'>, <class 'float'>)*, *mode: str*, **args*, ***kwargs*)

value is greater or lesser than some max/min

Parameters

- **value_name** (*ValueName*) – Which value to check
- **limit** (*int, float*) – value to check against
- **mode** (*'min', 'max'*) – whether the limit is a minimum or maximum
- ***args** –

- ****kwargs** –

Methods

`__init__`(value_name, limit, mode, *args, ...)	**param value_name** Which value to check

`check`(sensor_values)	
`reset`()	not stateful, do nothing.

Attributes

`mode`

__init__ (*value_name: vent.common.values.ValueName*, *limit: (<class 'int'>, <class 'float'>)*, *mode: str*, **args*, ***kwargs*)

Parameters

- **value_name** (*ValueName*) – Which value to check
- **limit** (*int, float*) – value to check against
- **mode** (*'min', 'max'*) – whether the limit is a minimum or maximum
- ***args** –
- ****kwargs** –

property mode

check (*sensor_values*)

reset ()
not stateful, do nothing.

class vent.alarm.condition.**CycleValueCondition** (*n_cycles*, **args*, ***kwargs*)
value goes out of range for a specific number of breath cycles **Methods**

`check`(sensor_values)	
`reset`()	not stateful, do nothing.

Attributes

`n_cycles`

_start_cycle
The breath cycle where the

Type int

_mid_check
whether a value has left the acceptable range and we are counting consecutive breath cycles

Type bool

Args: value_name (ValueName): Which value to check limit (int, float): value to check against mode ('min', 'max'): whether the limit is a minimum or maximum *args: **kwargs:

property n_cycles

check(*sensor_values*)

reset()
not stateful, do nothing.

class vent.alarm.condition.**TimeValueCondition**(*time*, **args*, ***kwargs*)
value goes out of range for specific amount of time

Parameters

- **time** (*float*) – number of seconds value must be out of range
- ***args** –
- ****kwargs** –

Methods

__init__(time, *args, **kwargs)	**param time** number of seconds value must be out of range
check(sensor_values)	
reset()	not stateful, do nothing.

__init__(*time*, **args*, ***kwargs*)

Parameters

- **time** (*float*) – number of seconds value must be out of range
- ***args** –
- ****kwargs** –

check(*sensor_values*)

reset()
not stateful, do nothing.

class vent.alarm.condition.**AlarmSeverityCondition**(*alarm_type: vent.alarm.AlarmType*, *severity: vent.alarm.AlarmSeverity*, *mode: str = 'min'*, **args*, ***kwargs*)

Parameters

- **alarm_type** –
- **severity** –
- **mode** (*str*) – one of ‘min’, ‘equals’, or ‘max’. ‘min’ returns true if the alarm is at least this value (note the difference from ValueCondition which returns true if the alarm is less than..) and vice versa for ‘max’.

  Note: ’min’ and ‘max’ use >= and <= rather than > and <

- ***args** –
- ****kwargs** –

Methods

`__init__`(alarm_type, severity, mode, *args, ...)	**param alarm_type**
`check`(sensor_values)	
`reset`()	If a condition is stateful, need to provide some method of resetting the state

Attributes

`mode`

__init__ (*alarm_type: vent.alarm.AlarmType*, *severity: vent.alarm.AlarmSeverity*, *mode: str = 'min'*, **args*, ***kwargs*)

Parameters

- **alarm_type** –
- **severity** –
- **mode** (*str*) – one of 'min', 'equals', or 'max'. 'min' returns true if the alarm is at least this value (note the difference from ValueCondition which returns true if the alarm is less than..) and vice versa for 'max'.

  Note: 'min' and 'max' use >= and <= rather than > and <
- ***args** –
- ****kwargs** –

property mode

check (*sensor_values*)

reset ()
If a condition is stateful, need to provide some method of resetting the state

class vent.alarm.condition.**CycleAlarmSeverityCondition** (*n_cycles*, **args*, ***kwargs*)
alarm goes out of range for a specific number of breath cycles

Todo: note that this is exactly the same as CycleValueCondition. Need to do the multiple inheritance thing

Methods

`check`(sensor_values)	
`reset`()	If a condition is stateful, need to provide some method of resetting the state

Attributes

`n_cycles`

_start_cycle
The breath cycle where the

Type int

_mid_check
whether a value has left the acceptable range and we are counting consecutive breath cycles

Type bool

Args: alarm_type: severity: mode (str): one of 'min', 'equals', or 'max'.

'min' returns true if the alarm is at least this value (note the difference from ValueCondition which returns true if the alarm is less than..) and vice versa for 'max'.

Note: 'min' and 'max' use >= and <= rather than > and <

*args: **kwargs:

property n_cycles

check(*sensor_values*)

reset()
If a condition is stateful, need to provide some method of resetting the state

8.5 Alarm Rule

Class to declare alarm rules

Classes

Alarm_Rule(name, conditions[, latch, ...])	• name of rule

class vent.alarm.rule.**Alarm_Rule**(*name: vent.alarm.AlarmType*, *conditions*, *latch=True*, *persistent=True*, *technical=False*)

- name of rule
- conditions: ((alarm_type, (condition_1, condition_2)), ...)
- persistent (bool): if True, alarm will not be visually dismissed until alarm conditions are no longer true
- latch (bool): if True, alarm severity cannot be decremented until user manually dismisses
- silencing/overriding rules

Methods

check(sensor_values)	Check all of our conditions.
reset()	

Attributes

`severity`	Last Alarm Severity from `.check()`

check(*sensor_values*)
Check all of our `conditions` .

Parameters **sensor_values** –

Returns:

property severity
Last Alarm Severity from `.check()` :returns: `AlarmSeverity`

reset()

CHAPTER

NINE

REQUIREMENTS

CHAPTER

TEN

DATASHEETS & MANUALS

10.1 Manuals

- Hamilton T1 Quick Guide

10.2 Other Reference Material

- Hamilton UI Simulator

CHAPTER

ELEVEN

SPECS

CHAPTER

TWELVE

CHANGELOG

12.1 Version 0.0

12.1.1 v0.0.2 (April xxth, 2020)

- Refactored gui into a module, splitting `widgets`, `styles`, and `defaults`.

12.1.2 v0.0.1 (April 12th, 2020)

- Added changelog
- Moved requirements for building docs to *requirements_docs.txt* so regular program reqs are a bit lighter.
- added autosummaries
- added additional resources & documentation files, with examples for adding external files like pdfs

12.1.3 v0.0.0 (April 12th, 2020)

Example of a changelog entry!!!

- We fixed this
- and this
- and this

Warning: but we didn't do this thing

Todo: and we still have to do this other thing.

CHAPTER

THIRTEEN

BUILDING THE DOCS

A very brief summary...

- Docs are configured to be built from `_docs` into `docs`.
- The main page is `index.rst` which links to the existing modules
- To add a new page, you can create a new `.rst` file if you are writing with Restructuredtext , or a *.md* file if you are writing with markdown.

13.1 Local Build

- `pip install -r requirements.txt`
- `cd _docs`
- `make html`

Documentation will be generated into `docs`

CHAPTER

FOURTEEN

H1 HEADING 8-)

14.1 h2 Heading

14.1.1 h3 Heading

14.1.1.1 h4 Heading

h5 Heading

h6 Heading

14.2 Horizontal Rules

14.3 Emphasis

This is bold text

This is bold text

This is italic text

This is italic text

14.4 Blockquotes

> Blockquotes can also be nested...
>
> > ...by using additional greater-than signs right next to each other...
> >
> > > ...or with spaces between arrows.

14.5 Lists

Unordered

- Create a list by starting a line with +, −, or ⋆
- Sub-lists are made by indenting 2 spaces:
  - Marker character change forces new list start:
    - Ac tristique libero volutpat at
    - Facilisis in pretium nisl aliquet
    - Nulla volutpat aliquam velit
- Very easy!

Ordered

1. Lorem ipsum dolor sit amet
2. Consectetur adipiscing elit
3. Integer molestie lorem at massa
4. You can use sequential numbers...
5. ...or keep all the numbers as `1.`

14.6 Code

Inline `code`

Indented code

```
// Some comments
line 1 of code
line 2 of code
line 3 of code
```

Block code "fences"

```
Sample text here...
```

Syntax highlighting

```
var foo = function (bar) {
  return bar++;
};

console.log(foo(5));
```

14.7 Links

link text

link with title

14.8 Images

Minion

Like links, Images also have a footnote style syntax

Alt text

With a reference later in the document defining the URL location:

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

V

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W