PVP

Release 0.2.0

jonny saunders et al

Aug 18, 2020

OVERVIEW

The global COVID-19 pandemic has highlighted the need for a low-cost, rapidly-deployable ventilator, for the current as well as future respiratory virus outbreaks. While safe and robust ventilation technology exists in the commercial sector, the small number of capable suppliers cannot meet the severe demands for ventilators during a pandemic.

<Statement of cost> Moreover, the specialized and proprietary equipment developed by medical device manufacturers is expensive and inaccessible in low-resource areas. Compounding the issue during an emergency, manufacturing time…

The **People's Ventilator Project (PVP)** is an open-source, low-cost pressure-control ventilator designed with minimal reliance on specialized medical parts to better adapt to supply chain shortages. The **PVP** largely follows established design conventions, most importantly active and computer-controlled inhalation, together with passive exhalation. It supports pressure-controlled ventilation, combined with standard-features like autonomous breath detection, and the suite of FDA required alarms.

<Statement of purpose>

PVP is a pressure-controlled ventilator that uses a minimal set of inexpensive, off-the-self hardware components. An inexpensive proportional valve controls inspiratory flow, and a relay valve controls expiratory flow. A gauge pressure sensor monitors airway pressure, and an inexpensive D-lite spirometer used in conjunction with a differential pressure sensor monitors expiratory flow.

PVP's components are coordinated by a Raspberry Pi 4 board, which runs the graphical user interface, administers the alarm system, monitors sensor values, and sends actuation commands to the valves. The core electrical system consists of two modular board 'hats', a sensor board and an actuator board, that stack onto the Raspberry Pi via 40-pin stackable headers. The modularity of this system enables individual boards to be revised or modified to substitute components in the case of part scarcity.

Links to system: … Mechanical overview … Electronics overview

CHAPTER

ONE

SOFTWARE

images/gui_overview_v1_1920px.png

PVP's software was developed to bring the philosophy of free and open-source software to medical devices. PVP is not only open from top to bottom, but we have developed it as a framework for **an adaptable, general-purpose, communally-developed ventilator.**

PVP's ventilation control system is fast, robust, and **written entirely in high-level Python** (3.7) – without the development and inspection bottlenecks of split computer/microprocessor systems that require users to read and write low-level hardware firmware.

All of PVP's components are **modularly designed**, allowing them to be reconfigured and expanded for new ventilation modes and hardware configurations.

We provide complete **API-level documentation** and an **automated testing suite** to give everyone the freedom to inspect, understand, and expand PVP's software framework.

1.1 PVP Modules

1.1.1 System Overview

1.1.2 Hardware Overview

1.1.2.1 Mechanical Diagram

Sensors | Hardware

Overview

The TigerVent has four main sensors: 1. oxygen sensor (O2S) 2. proximal pressure sensor (PS1) 3. expiratory pressure sensor (PS2) 4. expiratory flow sensor (FS1)

These materials interface with a modular sensor PCB that can be reconfigured for part substitution. The nominal design assumes both pressure sensors and the oxygen sensor have analog voltage outputs, and interface with the controller via I2C link with a 16-bit, 4 channel ADC (ADS1115). The expiratory flow sensor (SFM3300 or equivalent) uses a direct I2C interface, but can be replaced by a commercial spirometer and an additional differential pressure sensor.

Sensor PCB

Schematic

Bill of Materials

- - Ref
  - Part
  - Description
  - Datasheet
- - U1
  - Amphenol 1 PSI-D1-4V-MINI
  - Analog output differential pressure sensor
  - /DS-0103-Rev-A-1499253.pdf <- not sure best way to do this

images/mechanical_diagram.png

Fig. 1: Schematic diagram of main mechanical components

images/sensor_pcb_schematic.png

Fig. 2: Electrical schematic for sensor board

- - U3
  - Amphenol 1 PSI-D1-4V-MINI differential pressure sensor
  - Analog output differential pressure sensor
  - above
- - U2
  - Adafruit 4-channel ADS1115 ADC breakout
  - Supply ADC to RPi to read analog sensors
  - /adafruit-4-channel-adc-breakouts.pdf
- - U4
  - INA126 instrumentation amplifier, DIP-8
  - Instrumentation amplifier to boost oxygen sensor output
  - /ina126.pdf
- - J1
  - 01x02 2.54 mm pin header
  - Breakout for alert pin from ADS1115 ADC
  - none
- - J2
  - 02x04 2.54 mm pin header
  - Jumpers to select I2C address for ADC
  - none
- - J3
  - 40 pin RPi hat connector
  - Extends RPi GPIO pins to the board
  - (to be inserted)
- - J4
  - 01x02 2.54 mm 90 degree pin header
  - For direct connection to oxygen sensor output
  - none
- - J5
  - 01x04 2.54 mm 90 degree pin header pin header
  - For I2C connection to SFM3300 flow meter
  - none
- - J6
  - 01x03 2.54 mm 90 degree pin header pin header
  - Connector to use an additional analog output (ADS1115 input A3).
  - none

- - R1
  - 1-2.7 k resistor
  - Optional I2C pullup resistor (RPi already has 1.8k pullups)
  - none
- - R2
  - 1-2.7 k resistor
  - Connector to use an additional analog output (RPi already has 1.8k pullups).
  - none
- - R3
  - 0.1-100k resistor
  - R_G that sets gain for the INA126 instrumentation amplifier (U4). G = 5 + 80k/R_G
  - none

Flow sensor

Document D-lite alternative

Pressure sensors

Just use any other analog voltage output (0-4 V) sensor

Oxygen sensor

Explanation of interface circuit and some alts

- Expiratory flow sensor (FS1)

1.1.2.2 Flow actuators

- Actuator PCB/overview (link to PCB with BoM, schematic, layout, etc.)
- Proportional solenoid valve (V1) (link to doc with crit specs, driving circuit, part spec, datasheet, alternatives, etc.)
- Expiratory valve (V2) (link to doc with crit specs, driving circuit, part spec, datasheet, etc.)

1.1.2.3 Sensors

- Sensor PCB/overview (link to PCB with BoM, schematic, layout, etc.)
- Oxygen sensor (O2S) (link to doc with crit specs, interface circuit, part spec, datasheet, alternatives, etc.)
- Proximal pressure sensor (PS1)
- Expiratory pressure sensor (PS2)
- Expiratory flow sensor (FS1)

1.1.2.4 Safety Components

- 50 psi, high pressure relief valve (PRV1)
- Safety check valve (CV)
- 70 cmH2O patient-side pressure relief valve (PRV2)
- Filters (F1, F2)
- PEEP valve (PEEP) (include the design bifurcation in this module description)

1.1.2.5 Tubing and Adapters

- Manifold 1
- Manifold 2
- Mounting Bracket 1… etc.

1.1.2.6 Bill of Materials (need to think about what goes in this table, probably separate BoMs into tables by category, but here's a sample table)

Ref	Name	Part	Description
V1	Inspiratory on/off valve	red hat process valve	completely cut off flow if required
PRV1	High pressure relief valve	Sets to 50 psi	regulates upstream pressure to 50 psi
CV	Inspiratory check valve	valve stat here	In case of emergency power loss, allows patient to continue taking breaths from air
PRV2	Maximum pressure valve	. . .	Sets absolute maximum pressure at patient side to 53 cm H2O
F1/F2	Filters	HEPA filters?	Keeps the system's sensors from becoming contaminated
O2S	Oxygen sensor	Sensiron . . .	Checks FiO2 level
PS1/PS2	Pressure sensors	mini4v	Uses gas takeoffs to measure pressure at each desired point
FS1	Flow sensor	Sensiron flow sensor	Measures expiratory flow to calculate tidal volume
M1/M2	Manifolds	3D printed parts	Hubs to connect multiple components in one place
V3	Expiratory on/off valve	Festo Electrical Air Directional Control Valve, 3/2 flow, Normally Closed, 8 mm Push-to-Connect	Opens to initiation the expiratory cycle
PEEP	PEEP backpressure valve	PEEP valve	Sets PEEP on expiratory cycle!

1.1.3 Sensors

1.1.4 Actuators

1.1.5 Electronics

try to make a file index who knows

1.1.6 Safety

1.1.7 Enclosure

1.1.8 Software Overview

PVP runs as three independent processes:

- The *GUI* and *Coordinator* run in the first process, receive user input, display system status, and relay `ControlSetting` s to the *Controller* .
- At launch, the *Coordinator* spawns a *Controller* that runs the logic of the ventilator based on control values from the GUI.
- The *Controller* communicates with a third pigpiod process which communicates with the ventilation hardware

PVP is configured by

- The *Values* module parameterizes the different sensor and control values displayed by the GUI and used by the controller
- The *Prefs* module creates a `prefs.json` file in `~/pvp` that defines user-specific preferences.

PVP is launched like:

```
python3 -m pvp.main
```

And launch options can be displayed with the `--help` flag.

1.1.8.1 PVP Modules

1.1.9 GUI

1.1.9.1 Main GUI Module

Classes

`PVP_Gui`(coordinator, set_defaults, update_period)	The Main GUI window.

Functions

`launch_gui`(coordinator[, set_defaults, . . .])	Launch the GUI with its appropriate arguments and doing its special opening routine

class pvp.gui.main.**PVP_Gui**(*coordinator: pvp.coordinator.coordinator.CoordinatorBase, set_defaults: bool = False, update_period: float = 0.05, screenshot=False*)

The Main GUI window.

Creates 5 sets of widgets:

- A `Control_Panel` in the top left corner that controls basic system operation and settings
- A `Alarm_Bar` along the top that displays active alarms and allows them to be dismissed or muted
- A column of `Display` widgets (according to `values.DISPLAY_MONITOR`) on the left that display sensor values and control their alarm limits
- A column of `Plot` widgets (according to `values.PLOTS`) in the center that display waveforms of sensor readings
- A column of `Display` widgets (according to `values.DISPLAY_CONTROL`) that control ventilation settings

Attributes

`CONTROL`	Values to create `Display` widgets for in the Control column.
`MONITOR`	Values to create `Display` widgets for in the Sensor Monitor column.
`PLOTS`	Values to create `Plot` widgets for.
`control_width`	Relative width of the control column

continues on next page

Table 3 – continued from previous page

controls_set	Check if all controls are set
gui_closing(*args, **kwargs)	PySide2.QtCore.Signal emitted when the GUI is closing.
monitor_width	Relative width of the sensor monitor column
plot_width	Relative width of the plot column
state_changed(*args, **kwargs)	PySide2.QtCore.Signal emitted when the gui is started (True) or stopped (False)
total_width	computed from monitor_width+plot_width+control_width
update_period	The global delay between redraws of the GUI (seconds)

Methods

_screenshot()	Raise each of the alarm severities to check if they work and to take a screenshot
_set_cycle_control(value_name, new_value)	Compute the computed breath cycle control.
closeEvent(event)	Emit gui_closing and close!
handle_alarm(alarm)	Receive an Alarm from the Alarm_Manager
init_controls()	on startup, set controls in coordinator to ensure init state is synchronized
init_ui()	0. Create the UI components for the ventilator screen
init_ui_controls()	4. Create the "controls" column of widgets.Display widgets
init_ui_monitor()	2. Create the left "sensor monitor" column of widgets.Display widgets
init_ui_plots()	3. Create the Plot_Container
init_ui_signals()	5. Connect Qt signals and slots between widgets
init_ui_status_bar()	1. Create the widgets.Control_Panel and widgets.Alarm_Bar
limits_updated(control)	Receive updated alarm limits from the Alarm_Manager
load_state(state, dict])	Load GUI state and reconstitute
save_state()	Try to save GUI state to prefs['VENT_DIR"] + prefs['GUI_STATE_FN']
set_breath_detection(breath_detection)	Connected to breath_detection_button - toggles autonomous breath detection in the controller

continues on next page

Table 4 – continued from previous page

`set_control`(control_object)	Set a control in the alarm manager, coordinator, and gui
`set_pressure_units`(units)	Select whether pressure units are displayed as “cmH2O” or “hPa”
`set_value`(new_value[, value_name])	Set a control value using a value and its name.
`start`()	Click the `start_button`
`toggle_cycle_widget`(button)	Set which breath cycle control is automatically calculated
`toggle_lock`(state)	Toggle the lock state of the controls
`toggle_start`(state)	Start or stop ventilation.
`update_gui`(vals)	**param vals** Default None, but SensorValues can be passed manually – usually for debugging
`update_state`(state_type, key, val, float, int])	Update the GUI state and save it to disk with `Vent_Gui.save_state()`

Continually polls the `coordinator` with `update_gui()` to receive new `SensorValues` and dispatch them to display widgets, plot widgets, and the alarm manager

Note: Only one instance can be created at a time. Uses `set_gui_instance()` to store a reference to itself. after initialization, use *get_gui_instance* to retrieve a reference.

Parameters

- **coordinator** (`CoordinatorBase`) – Used to communicate with the `ControlModuleBase`.
- **set_defaults** (`bool`) – Whether default *Value* s should be set on initialization (default `False`) – used for testing and development, values should be set manually for each patient.
- **update_period** (`float`) – The global delay between redraws of the GUI (seconds), used by `timer`.
- **screenshot** (`bool`) – Whether alarms should be manually raised to show the different alarm severities, only used for testing and development and should never be used in a live system.

monitor
: Dictionary mapping `values.DISPLAY_MONITOR` keys to `widgets.Display` objects

Type dict

controls
: Dictionary mapping `values.DISPLAY_CONTROL` keys to `widgets.Display` objects

Type dict

plot_box
: Container for plots

Type `Plot_Box`

coordinator
: Some coordinator object that we use to communicate with the controller

Type *pvp.coordinator.coordinator.CoordinatorBase*

alarm_manager
Alarm manager instance

Type *Alarm_Manager*

timer
Timer that calls *PVP_Gui.update_gui()*

Type PySide2.QtCore.QTimer

running
whether ventilation is currently running

Type bool

locked
whether controls have been locked

Type bool

start_time
Start time as returned by time.time()

Type float

update_period
The global delay between redraws of the GUI (seconds)

Type float

logger
Logger generated by *loggers.init_logger()*

gui_closing(**args, **kwargs*) **= <PySide2.QtCore.Signal object>**
PySide2.QtCore.Signal emitted when the GUI is closing.

state_changed(**args, **kwargs*) **= <PySide2.QtCore.Signal object>**
PySide2.QtCore.Signal emitted when the gui is started (True) or stopped (False)

MONITOR = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value object>), (<ValueN
Values to create *Display* widgets for in the Sensor Monitor column. See *values.DISPLAY_MONITOR*

CONTROL = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value object>), (<ValueN
Values to create *Display* widgets for in the Control column. See *values.CONTROL*

PLOTS = OrderedDict([(<ValueName.PRESSURE: 10>, <pvp.common.values.Value object>), (<Va
Values to create *Plot* widgets for. See *values.PLOTS*

monitor_width = 3
Relative width of the sensor monitor column

plot_width = 4
Relative width of the plot column

control_width = 3
Relative width of the control column

total_width = 10
computed from monitor_width+plot_width+control_width

update_gui(*vals: pvp.common.message.SensorValues = None*)

Parameters **vals** (SensorValue) – Default None, but SensorValues can be passed manually – usually for debugging

init_ui()

0. Create the UI components for the ventilator screen

Call, in order:

- *PVP_Gui.init_ui_status_bar()*
- *PVP_Gui.init_ui_monitor()*
- *PVP_Gui.init_ui_plots()*
- *PVP_Gui.init_ui_controls()*
- *PVP_Gui.init_ui_signals()*

Create and set sizes of major layouts

init_ui_status_bar()

1. Create the widgets.Control_Panel and widgets.Alarm_Bar

and add them to the main layout

init_ui_monitor()

2. Create the left "sensor monitor" column of widgets.Display widgets

And add the logo to the bottom left corner if there's room

init_ui_plots()

3. Create the *Plot_Container*

init_ui_controls()

4. Create the "controls" column of widgets.Display widgets

init_ui_signals()

5. Connect Qt signals and slots between widgets

- Connect controls and sensor monitors to *PVP_Gui.set_value()*
- Connect control panel buttons to their respective methods

set_value(*new_value*, *value_name=None*)

Set a control value using a value and its name.

Constructs a *message.ControlSetting* object to give to *PVP_Gui.set_control()*

Note: This method is primarily intended as a means of responding to signals from other widgets, Other cases should use *set_control()*

Parameters

- **new_value** (*float*) – A new value for some control setting
- **value_name** (*values.ValueName*) – THe ValueName for the control setting. If None, assumed to be coming from a *Display* widget that can identify itself with its objectName

set_control (*control_object: pvp.common.message.ControlSetting*)
Set a control in the alarm manager, coordinator, and gui

Also update our state with `update_state()`

Parameters **control_object** (*message.ControlSetting*) – A control setting to give to `CoordinatorBase.set_control`

handle_alarm (*alarm: pvp.alarm.alarm.Alarm*)
Receive an *Alarm* from the *Alarm_Manager*

Alarms are both raised and cleared with this method – there is no separate "clear_alarm" method because an alarm of *AlarmSeverity* of `OFF` is cleared.

Give the alarm to the *Alarm_Bar* and update the alarm *Display.alarm_state* of all widgets listed as `Alarm.cause`

Parameters **alarm** (*Alarm*) – The alarm to raise (or clear)

limits_updated (*control: pvp.common.message.ControlSetting*)
Receive updated alarm limits from the *Alarm_Manager*

When a value is set that has an *Alarm_Rule* that *Alarm_Rule.depends* on it, the alarm thresholds will be updated and handled here.

Eg. the high-pressure alarm is set to be 15% above PIP. When PIP is changed, this method will receive a *message.ControlSetting* that tells us that alarm threshold has changed.

Update the *Display* and *Plot* widgets.

If we are setting a new HAPA limit, that is also sent to the controller as it needs to respond as quickly as possible to high-pressure events.

Parameters **control** (*message.ControlSetting*) – A ControlSetting with its `max_value` or

:param `min_value` set:

start ()
Click the `start_button`

toggle_start (*state: bool*)
Start or stop ventilation.

Typically called by the `PVP_Gui.control_panel.start_button`.

Raises a dialogue to confirm ventilation start or stop

Starts or stops the controller via the coordinator

If starting, locks controls.

Parameters **state** (*bool*) – If True, start ventilation. If False, stop ventilation.

closeEvent (*event*)
Emit *gui_closing* and close!

Kill the coordinator with *CoordinatorBase.kill()*

toggle_lock (*state*)
Toggle the lock state of the controls

Typically called by `PVP_Gui.control_panel.lock_button`

Parameters **state** –

Returns:

update_state (*state_type: str, key: str, val: Union[str, float, int]*)
Update the GUI state and save it to disk with `Vent_Gui.save_state()`

Currently, just saves the state of control settings.

Parameters

- **state_type** (*str*) – What type of state to save, one of `('controls')`
- **key** (*str*) – Which of that type is being saved (eg. if 'control', 'PIP')
- **val** (*str, float, int*) – What is that item being set to?

Returns:

save_state ()
Try to save GUI state to `prefs['VENT_DIR"] + prefs['GUI_STATE_FN']`

load_state (*state: Union[str, dict]*)
Load GUI state and reconstitute

currently, just `PVP_Gui.set_value()` for all previously saved values

Parameters **state** (*str, dict*) – either a pathname to a state file or an already-loaded state dictionary

staticMetaObject = <PySide2.QtCore.QMetaObject object>

toggle_cycle_widget (*button*)
Set which breath cycle control is automatically calculated

The timing of a breath cycle can be parameterized with Respiration Rate, Inspiration Time, and Inspiratory/Expiratory ratio, but if two of these parameters are set the third is already known.

This method changes which value has its `Display` widget hidden and is automatically calculated

Parameters **button** (*PySide2.QtWidgets.QAbstractButton, values.ValueName*) – The Qt Button that invoked the method or else a ValueName

set_pressure_units (*units*)
Select whether pressure units are displayed as "cmH2O" or "hPa"

calls `Display.set_units()` on controls and plots that display pressure

Parameters **units** (*str*) – one of "cmH2O" or "hPa"

set_breath_detection (*breath_detection: bool*)
Connected to `breath_detection_button` - toggles autonomous breath detection in the controller

Parameters **breath_detection** (*bool*) – Whether the controller detects autonomous breaths and resets the breath cycle accordingly

_set_cycle_control (*value_name: str, new_value: float*)
Compute the computed breath cycle control.

We only actually have BPM and INSPt as controls, so if we're using I:E ratio we have to compute one or the other.

Computes the value and calls `set_control()` with the appropriate values:

```
# ie = inspt/expt
# inspt = ie*expt
# expt = inspt/ie
#
# cycle_time = inspt + expt
```

(continues on next page)

(continued from previous page)

```
# cycle_time = inspt + inspt/ie
# cycle_time = inspt * (1+1/ie)
# inspt = cycle_time / (1+1/ie)
```

property controls_set
Check if all controls are set

Note: Note that even when RR or INSPt are autocalculated, they are still set in their control objects, so this check is the same regardless of what is set to autocalculate

property update_period
The global delay between redraws of the GUI (seconds)

init_controls()
on startup, set controls in coordinator to ensure init state is synchronized

_screenshot()
Raise each of the alarm severities to check if they work and to take a screenshot

Warning: should never be used except for testing and development!

pvp.gui.main.**launch_gui**(*coordinator*, *set_defaults=False*, *screenshot=False*) → Tuple[PySide2.QtWidgets.QApplication, *pvp.gui.main.PVP_Gui*]
Launch the GUI with its appropriate arguments and doing its special opening routine

To launch the gui, one must:

- Create a `PySide2.QtWidgets.QApplication`
- Set the app style using `gui.styles.DARK_THEME`
- Set the app palette with *`gui.styles.set_dark_palette()`*
- Call the gui's `show` method

Parameters

- **coordinator** (*`coordinator.CoordinatorBase`*) – Coordinator used to communicate between GUI and controller
- **set_defaults** (*`bool`*) – whether default control parameters should be set on startup – only to be used for development or testing
- **screenshot** (*`bool`*) – whether alarms should be raised to take a screenshot, should never be used on a live system.

Returns The `PySide2.QtWidgets.QApplication` and *`PVP_Gui`*

Return type (tuple)

1.1.9.2 GUI Widgets

Control Panel

The Control Panel starts and stops ventilation and controls runtime options

Classes

Control_Panel()	The control panel starts and stops ventilation and controls runtime settings
HeartBeat(update_interval, timeout_dur)	Track state of connection with Controller
Lock_Button(*args, **kwargs)	Button to lock and unlock controls
Start_Button(*args, **kwargs)	Button to start and stop Ventilation, created by *Control_Panel*
StopWatch(update_interval, *args, **kwargs)	Simple widget to display ventilation time!

class pvp.gui.widgets.control_panel.**Control_Panel**

The control panel starts and stops ventilation and controls runtime settings

It creates:

- Start/stop button
- **Status indicator - a clock that increments with heartbeats,** or some other visual indicator that things are alright
- Version indicator
- Buttons to select options like cycle autoset and automatic breath detection

Methods

_pressure_units_changed(button)	Emit the str of the current pressure units
init_ui()	Initialize all graphical elements and buttons!

Attributes

cycle_autoset_changed(*args, **kwargs)	Signal emitted when a different breath cycle control value is set to be autocalculated
pressure_units_changed(*args, **kwargs)	Signal emitted when pressure units have been changed.

Args:

start_button

Button to start and stop ventilation

Type *Start_Button*

lock_button

Button used to lock controls

Type *Lock_Button*

heartbeat

Widget to keep track of communication with controller

Type *HeartBeat*

runtime
Widget used to display time since start of ventilation

Type *StopWatch*

pressure_units_changed (**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**
Signal emitted when pressure units have been changed.

Contains str of current pressure units

cycle_autoset_changed (**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**
Signal emitted when a different breath cycle control value is set to be autocalculated

init_ui ()
Initialize all graphical elements and buttons!

_pressure_units_changed (*button*)
Emit the str of the current pressure units

Parameters **button** (PySide2.QtWidgets.QPushButton) – Button that was clicked

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**Start_Button** (**args*, ***kwargs*)
Button to start and stop Ventilation, created by *Control_Panel* **Methods**

load_pixmaps()	Load pixmaps to *Start_Button.pixmaps*
set_state(state)	Set state of button

Attributes

states	Possible states of Start_Button

pixmaps
Dictionary containing pixmaps used to draw start/stop state

Type dict

states = ['OFF', 'ON', 'ALARM']
Possible states of Start_Button

load_pixmaps ()
Load pixmaps to *Start_Button.pixmaps*

set_state (*state*)
Set state of button

Should only be called by other objects (as there are checks to whether it's ok to start/stop that we shouldn't be aware of)

Parameters **state** (*str*) – one of `('OFF', 'ON', 'ALARM')`

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**Lock_Button** (**args*, ***kwargs*)
Button to lock and unlock controls

Created by *Control_Panel* **Methods**

load_pixmaps()	Load pixmaps used to display lock state to *Lock_Button.pixmaps*
set_state(state)	Set lock state of button

Attributes

states	Possible states of Lock Button

pixmaps
Dictionary containing pixmaps used to draw locked/unlocked state

Type dict

states = ['DISABLED', 'UNLOCKED', 'LOCKED']
Possible states of Lock Button

load_pixmaps()
Load pixmaps used to display lock state to *Lock_Button.pixmaps*

set_state(*state*)
Set lock state of button

Should only be called by other objects (as there are checks to whether it's ok to start/stop that we shouldn't be aware of)

Parameters **state** (*str*) – ('OFF', 'ON', 'ALARM')

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**HeartBeat**(*update_interval: int = 100, timeout_dur: int = 5000*)
Track state of connection with Controller

Check when we last had contact with controller every *HeartBeat.update_interval* ms, if longer than HeartBeat.timeout_dur then emit a timeout signal

Parameters

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

Methods

_heartbeat()	Called every (update_interval) milliseconds to set the check the status of the heartbeat.
beatheart(heartbeat_time)	Slot that receives timestamps of last contact with controller
init_ui()	Initialize labels and status indicator
set_indicator([state])	Set visual indicator
set_state(state)	Set running state
start_timer([update_interval])	Start *HeartBeat.timer* to check for contact with controller
stop_timer()	Stop timer and clear text

Attributes

heartbeat(*args, **kwargs)	Signal that requests to affirm contact with controller if no message has been received in timeout duration
timeout(*args, **kwargs)	Signal that a timeout has occurred – too long between contact with controller.

_state
whether the system is running or not

Type bool

_last_heartbeat
Timestamp of last contact with controller

Type float

start_time
Time that ventilation was started

Type float

timer
Timer that checks for last contact

Type `PySide2.QtCore.QTimer`

update_interval
How often to do the heartbeat, in ms

Type int

timeout
how long to wait before hearing from control process, in ms

Type int

timeout(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**
Signal that a timeout has occurred – too long between contact with controller.

heartbeat(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**
Signal that requests to affirm contact with controller if no message has been received in timeout duration

init_ui()
Initialize labels and status indicator

set_state(*state*)
Set running state

if just starting reset `HeartBeat._last_heartbeat`

Parameters **state** (`bool`) – Whether we are starting (True) or stopping (False)

set_indicator(*state=None*)
Set visual indicator

Parameters **state** (`'ALARM', 'OFF', 'NORMAL'`) – Current state of connection with controller

start_timer(*update_interval=None*)
Start `HeartBeat.timer` to check for contact with controller

Parameters **update_interval** (`int`) – How often (in ms) the timer should be updated. if None, use `self.update_interval`

stop_timer()
Stop timer and clear text

beatheart(*heartbeat_time*)
Slot that receives timestamps of last contact with controller

Parameters **heartbeat_time** (*float*) – timestamp of last contact with controller

_heartbeat()
Called every (update_interval) milliseconds to set the check the status of the heartbeat.

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**StopWatch**(*update_interval: float = 100, *args, **kwargs*)
Simple widget to display ventilation time!

Parameters

- **update_interval** (*float*) – update clock every n seconds
- ***args** – passed to PySide2.QtWidgets.QLabel
- ****kwargs** – passed to PySide2.QtWidgets.QLabel

Methods

`__init__`(update_interval, *args, **kwargs)	Simple widget to display ventilation time!
`_update_time`()	
`init_ui`()	
`start_timer`([update_interval])	**param update_interval** How often (in ms) the timer should be updated.
`stop_timer`()	Stop timer and reset label

__init__(*update_interval: float = 100, *args, **kwargs*)
Simple widget to display ventilation time!

Parameters

- **update_interval** (*float*) – update clock every n seconds
- ***args** – passed to PySide2.QtWidgets.QLabel
- ****kwargs** – passed to PySide2.QtWidgets.QLabel

staticMetaObject = <PySide2.QtCore.QMetaObject object>

init_ui()

start_timer(*update_interval=None*)

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

stop_timer()
Stop timer and reset label

_update_time()

Alarm Bar

The *Alarm_Bar* displays *Alarm* status with *Alarm_Card* widgets and plays alarm sounds with the *Alarm_Sound_Player*

Classes

Alarm_Bar()	Holds and manages a collection of *Alarm_Card* s and communicates
Alarm_Card(alarm)	Representation of an alarm raised by *Alarm_Manager* in GUI.
Alarm_Sound_Player(increment_delay, *args, …)	Plays alarm sounds to reflect current alarm severity and active duration with `PySide2.QtMultimedia.QSoundEffect` objects

class `pvp.gui.widgets.alarm_bar.`**`Alarm_Bar`**

Holds and manages a collection of *Alarm_Card* s and communicates requests for dismissal to the *Alarm_Manager*

The alarm bar also manages the *Alarm_Sound_Player* **Methods**

add_alarm(alarm)	Add an alarm created by the *Alarm_Manager* to the bar.
clear_alarm(alarm, alarm_type)	Remove an alarm card, update appearance and sound player to reflect current max severity
init_ui()	Initialize the UI
make_icons()	Create pixmaps from standard icons to display for different alarm types
set_icon(state)	Change the icon and bar appearance to reflect the alarm severity
update_icon()	Call *set_icon()* with highest severity in *Alarm_Bar.alarms*

Attributes

alarm_level	Current maximum *AlarmSeverity*

`alarms`

A list of active alarms

Type typing.List[*Alarm*]

`alarm_cards`

A list of active alarm cards

Type typing.List[*Alarm_Card*]

`sound_player`

Class that plays alarm sounds!

Type *Alarm_Sound_Player*

`icons`

Dictionary of pixmaps with icons for different alarm levels

Type dict

make_icons()
Create pixmaps from standard icons to display for different alarm types

Store in *Alarm_Bar.icons*

init_ui()
Initialize the UI

- Create layout
- Set icon
- Create mute button

add_alarm(*alarm:* pvp.alarm.alarm.Alarm)
Add an alarm created by the *Alarm_Manager* to the bar.

If an alarm alreaady exists with that same *AlarmType*, *Alarm_Bar.clear_alarm()*

Insert new alarm in order the prioritizes alarm severity with highest severity on right

Set alarm sound and begin playing if not already.

Parameters **alarm** (*Alarm*) – Alarm to be added

clear_alarm(*alarm:* pvp.alarm.alarm.Alarm = *None*, *alarm_type:* pvp.alarm.AlarmType = *None*)
Remove an alarm card, update appearance and sound player to reflect current max severity

Must pass one of either `alarm` or `alarm_type`

Parameters

- **alarm** (*Alarm*) – Alarm to be cleared
- **alarm_type** (*AlarmType*) – Alarm type to be cleared

update_icon()
Call *set_icon()* with highest severity in *Alarm_Bar.alarms*

set_icon(*state:* pvp.alarm.AlarmSeverity = *None*)
Change the icon and bar appearance to reflect the alarm severity

Parameters **state** (*AlarmSeverity*) – Alarm Severity to display, if None change to default display

property alarm_level
Current maximum *AlarmSeverity*

Returns *AlarmSeverity*

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.alarm_bar.**Alarm_Card**(*alarm:* pvp.alarm.alarm.Alarm)
Representation of an alarm raised by *Alarm_Manager* in GUI.

If allowed by alarm (by `latch` setting) , allows user to dismiss/silence alarm.

Otherwise request to dismiss is logged by *Alarm_Manager* and the card is dismissed when the conditions that generated the alarm are no longer met.

Parameters **alarm** (*Alarm*) – Alarm to represent

Methods

_dismiss()	Gets the *Alarm_Manager* instance and calls *Alarm_Manager.dismiss_alarm()*
init_ui()	Initialize graphical elements

alarm
The represented alarm

Type *Alarm*

severity
The severity of the represented alarm

Type *AlarmSeverity*

close_button
Button that requests an alarm be dismissed

Type PySide2.QtWidgets.QPushButton

init_ui()
Initialize graphical elements

- Create labels
- Set stylesheets
- Create and connect dismiss button

Returns:

_dismiss()
Gets the *Alarm_Manager* instance and calls *Alarm_Manager.dismiss_alarm()*

Also change appearance of close button to reflect requested dismissal

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.alarm_bar.**Alarm_Sound_Player**(*increment_delay: int = 10000*, **args*, ***kwargs*)

Plays alarm sounds to reflect current alarm severity and active duration with PySide2.QtMultimedia.QSoundEffect objects

Alarm sounds indicate severity with the number and pitch of tones in a repeating tone cluster (eg. low severity sounds have a single repeating tone, but high-severity alarms have three repeating tones)

They indicate active duration by incrementally removing a low-pass filter and making tones have a sharper attack and decay.

When an alarm of any severity is started the <severity_0.wav file begins playing, and a timer is started to call *Alarm_Sound_Player.increment_level()*

Parameters

- **increment_delay** (*int*) – Delay between calling *Alarm_Sound_Player.increment_level()*
- ****kwargs** (**args,*) – passed to PySide2.QtWidgets.QWidget

Methods

increment_level()	If current level is below the maximum level, increment with Alarm_Sound_Player.set_sound()
init_audio()	Load audio files in pvp/external/audio and add to Alarm_Sound_Player.idx
play()	Start sound playback
set_mute(mute)	Set mute state
set_sound(severity, level)	Set sound to be played
stop()	Stop sound playback

Attributes

severity_map	mapping between string representations of severities used by filenames and AlarmSeverity

idx
Dictionary of dictionaries allowing sounds to be accessed like `self.idx[AlarmSeverity][level]`

Type dict

files
list of sound file paths

Type list

increment_delay
Time between calling `Alarm_Sound_Player.increment_level`()` in ms

Type int

playing
Whether or not a sound is playing

Type bool

_increment_timer
Timer that increments alarm sound level

Type `PySide2.QtCore.QTimer`

_changing_track
used to ensure single sound changing call happens at a time.

Type `threading.Lock`

severity_map = {'high': <AlarmSeverity.HIGH: 3>, 'low': <AlarmSeverity.LOW: 1>, 'med
mapping between string representations of severities used by filenames and `AlarmSeverity`

init_audio()
Load audio files in `pvp/external/audio` and add to `Alarm_Sound_Player.idx`

play()
Start sound playback

Play sound listed as `Alarm_Sound_Player._current_sound`

Note: `Alarm_Sound_Player.set_sound()` must be called first.

stop()
Stop sound playback

set_sound(*severity: pvp.alarm.AlarmSeverity = None*, *level: int = None*)
Set sound to be played

At least an `AlarmSeverity` must be provided.

Parameters

- **severity** (`AlarmSeverity`) – Severity of alarm sound to play
- **level** (`int`) – level (corresponding to active duration) of sound to play

increment_level()
If current level is below the maximum level, increment with `Alarm_Sound_Player.set_sound()`
Returns:

staticMetaObject = <PySide2.QtCore.QMetaObject object>

set_mute(*mute: bool*)
Set mute state

Parameters mute (`bool`) – if True, mute. if False, unmute.

Display

Unified monitor & control widget

Displays sensor values, and can optionally control system settings.

The `PVP_Gui` instantiates display widgets according to the contents of `values.DISPLAY_CONTROL` and `values.DISPLAY_MONITOR`

Classes

`Display`(value, update_period, enum_name, . . .)	Unified widget for display of sensor values and control of ventilation parameters
`Limits_Plot`(style, *args, **kwargs)	Widget to display current value in a bar graph along with alarm limits

class pvp.gui.widgets.display.**Display**(*value: pvp.common.values.Value*, *update_period: float = 0.5*, *enum_name: pvp.common.values.ValueName = None*, *button_orientation: str = 'left'*, *control_type: Union[None, str] = None*, *style: str = 'dark'*, **args*, ***kwargs*)
Unified widget for display of sensor values and control of ventilation parameters

Displayed values are updated according to `Dispaly.timed_update()`

Parameters

- **value** (`Value`) – Value object to represent
- **update_period** (`float`) – Amount of time between updates of the textual display of values
- **enum_name** (`ValueName`) – Value name of object to represent
- **button_orientation** (`'left', 'right'`) – whether the controls are drawn on the

'left' or 'right'

- **control_type** (*None, 'slider', 'record'*) – type of control - either None (no control), slider (a slider can be opened to set a value), or record where recent sensor values are averaged and used to set the control value. Both types of control allow values to be input from the keyboard by clicking on the editable label
- **style** (*'light', 'dark'*) – whether the widget is 'dark' (light text on dark background) or 'light' (dark text on light background)
- ****kwargs** (**args,*) – passed on to PySide2.QtWidgets.QWidget

Methods

_value_changed(new_value)	"outward-directed" method to emit new changed control value when changed by this widget
init_ui()	UI is initialized in several stages
init_ui_labels()	
init_ui_layout()	Basically two methods...
init_ui_limits()	Create widgets to display sensed value alongside set value
init_ui_record()	
init_ui_signals()	
init_ui_slider()	
init_ui_toggle_button()	
redraw()	Redraw all graphical elements to ensure internal model matches view
set_locked(locked)	Set locked status of control
set_units(units)	Set pressure units to display as cmH2O or hPa.
timed_update()	Refresh textual sensor values only periodically to prevent them from being totally unreadable from being changed too fast.
toggle_control(state)	Toggle the appearance of the slider, if a slider
toggle_record(state)	Toggle the recording state, if a recording control
update_limits(control)	Update the alarm range and the GUI elements corresponding to it
update_sensor_value(new_value)	Receive new sensor value and update display widgets
update_set_value(new_value)	Update to reflect new control value set from elsewhere (inwardly directed setter)

Attributes

alarm_state	Current visual display of alarm severity
is_set	Check if value has been set for this control.
value_changed(*args, **kwargs)	Signal emitted when controlled value of display object has changed.

self.**name**
: Unpacked from value

self.**units**
: Unpacked from value

self.**abs_range**
: Unpacked from value

`self.safe_range`
Unpacked from `value`

`self.alarm_range`
initialized from `value`, but updated by alarm manager

`self.decimals`
Unpacked from `value`

`self.update_period`
Amount of time between updates of the textual display of values

Type float

`self.enum_name`
Value name of object to represent

Type `ValueName`

`self.orientation`
whether the controls are drawn on the `'left'` or `'right'`

Type 'left', 'right'

`self.control`
type of control - either `None` (no control), `slider` (a slider can be opened to set a value), or `record` where recent sensor values are averaged and used to set the control value.

Type None, 'slider', 'record'

`self._style`
whether the widget is `'dark'` (light text on dark background) or `'light'` (dark text on light background)

Type 'light', 'dark'

`self.set_value`
current set value of controlled value, if any

Type float

`self.sensor_value`
current value of displayed sensor value, if any.

Type float

value_changed(**args*, ***kwargs*) **= <PySide2.QtCore.Signal object>**
Signal emitted when controlled value of display object has changed.

Contains new value (float)

init_ui()
UI is initialized in several stages

- 0: this method, get stylesheets based on `self._style` and call remaining initialization methods
- 1: `Display.init_ui_labels()` - create generic labels shared by all display objects
- 2: `Display.init_ui_toggle_button()` - create the toggle or record button used by controls
- 3: `Display.init_ui_limits()` - create a plot that displays the sensor value graphically relative to the alarm limits
- 4: `Display.init_ui_slider()` or `Display.ini_ui_record()` - depending on what type of control this is

- 5: `Display.init_ui_layout()` since the results of the previous steps varies, do all layout at the end depending on orientation
- 6: `Display.init_ui_signals()` connect slots and signals

init_ui_labels()

init_ui_toggle_button()

init_ui_limits()
Create widgets to display sensed value alongside set value

init_ui_slider()

init_ui_record()

init_ui_layout()
Basically two methods… lay objects out depending on whether we're oriented with our button to the left or right

init_ui_signals()

toggle_control(*state*)
Toggle the appearance of the slider, if a slider

Parameters **state** (`bool`) – Whether to show or hide the slider

toggle_record(*state*)
Toggle the recording state, if a recording control

Parameters **state** (`bool`) – Whether recording should be started or stopped. when started, start storing new sensor values in a list. when stopped, average thgem and emit new value.

_value_changed(*new_value: float*)
"outward-directed" method to emit new changed control value when changed by this widget

Pop a confirmation dialog if values are set outside the safe range.

Parameters

- **new_value** (`float`) – new value!
- **emit** (`bool`) – whether to emit the *value_changed* signal (default True) – in the case that our value is being changed by someone other than us

update_set_value(*new_value: float*)
Update to reflect new control value set from elsewhere (inwardly directed setter)

Parameters **new_value** (`float`) – new value to set!

update_sensor_value(*new_value: float*)
Receive new sensor value and update display widgets

Parameters **new_value** (`float`) – new sensor value!

update_limits(*control: pvp.common.message.ControlSetting*)
Update the alarm range and the GUI elements corresponding to it

Parameters **control** (`ControlSetting`) – control setting with min_value or max_value

redraw()
Redraw all graphical elements to ensure internal model matches view

Typically used when changing units

timed_update()
Refresh textual sensor values only periodically to prevent them from being totally unreadable from being changed too fast.

set_units(*units: str*)
Set pressure units to display as cmH2O or hPa.

Uses functions from `pvp.common.unit_conversion` such that

- `self._convert_in` converts internal, canonical units to displayed units (eg. `cmH2O` is used by all other modules, so we convert it to `hPa`
- `self._convert_out` converts displayed units to send to other parts of the system

Note: currently unit conversion is only supported for Pressure.

Parameters **units** (`'cmH2O', 'hPa'`) – new units to display

set_locked(*locked: bool*)
Set locked status of control

Parameters **locked** (`bool`) – If True, disable all controlling widgets, if False, re-enable.

property is_set
Check if value has been set for this control.

Used to check if all settings have been set preflight by `PVP_Gui`

Returns whether we have an `Display.set_value`

Return type bool

property alarm_state
Current visual display of alarm severity

Change sensor value color to reflect the alarm state of that set parameter –

eg. if we have a HAPA alarm, set the PIP control to display as red.

Returns `AlarmSeverity`

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.display.**Limits_Plot**(*style: str = 'light'*, **args*, ***kwargs*)
Widget to display current value in a bar graph along with alarm limits

Parameters **style** (`'light', 'dark'`) – Whether we are being displayed in a light or dark styled `Display` widget

Methods

`init_ui`()	Create bar chart and horizontal lines to reflect
`update_value`(min, max, sensor_value, set_value)	Move the lines! Pass any of the represented values.
`update_yrange`()	Set yrange to ensure that the set value is always in the center of the plot and that all the values are in range.

set_value
Set value of control, displayed as horizontal black line always set at center of bar

Type float

sensor_value
Sensor value to compare against control, displayed as bar

Type float

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

staticMetaObject = <PySide2.QtCore.QMetaObject object>

init_ui()
Create bar chart and horizontal lines to reflect

- Sensor Value
- Set Value
- High alarm limit
- Low alarm limit

update_value(*min: float = None*, *max: float = None*, *sensor_value: float = None*, *set_value: float = None*)
Move the lines! Pass any of the represented values.

Also updates yrange to ensure that the control value is always centered in the plot

Parameters

- **min** (*float*) – new alarm minimum
- **max** (*float*) – new alarm _maximum
- **sensor_value** (*float*) – new value for the sensor bar plot
- **set_value** (*float*) – new value for the set value line

update_yrange()
Set yrange to ensure that the set value is always in the center of the plot and that all the values are in range.

Plot

Widgets to plot waveforms of sensor values

The *PVP_Gui* creates a *Plot_Container* that allows the user to select

- which plots (of those in `values.PLOT`) are displayed
- the timescale (x range) of the displayed waveforms

Plots display alarm limits as red horizontal bars

Classes

Plot(name[, buffer_size, plot_duration, . . .])	Waveform plot of single sensor value.
Plot_Container(plot_descriptors, . . .)	Container for multiple :class;`.Plot` objects

Data

PLOT_FREQ	Update frequency of *Plot* s in Hz

continues on next page

Table 26 – continued from previous page

PLOT_TIMER	A QTimer that updates :class:`.TimedPlotCurveItem`s

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

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

class pvp.gui.widgets.plot.**Plot**(*name*, *buffer_size=4092*, *plot_duration=10*, *plot_limits: tuple = None*, *color=None*, *units=''*, ***kwargs*)
Waveform plot of single sensor value.

Plots values continuously, wrapping at zero rather than shifting x axis.

Parameters

- **name** (*str*) – String version of *ValueName* used to set title
- **buffer_size** (*int*) – number of samples to store
- **plot_duration** (*float*) – default x-axis range
- **plot_limits** (*tuple*) – tuple of (ValueName)s for which to make pairs of min and max range lines
- **color** (*None, str*) – color of lines
- **units** (*str*) – Unit label to display along title
- ****kwargs** –

Methods

reset_start_time()	Reset start time – return the scrolling time bar to position 0
set_duration(dur)	Set duration, or span of x axis.
set_safe_limits(limits)	Set the position of the max and min lines for a given value
set_units(units)	Set displayed units
update_value(new_value)	Update with new sensor value

timestamps
deque of timestamps

Type collections.deque

history
deque of sensor values

Type collections.deque

cycles
deque of breath cycles

Type collections.deque

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

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

set_duration(*dur: float*)
Set duration, or span of x axis.

Parameters **dur** (*float*) – span of x axis (in seconds)

update_value(*new_value: tuple*)
Update with new sensor value

Parameters **new_value** (*tuple*) – (timestamp from time.time(), breath_cycle, value)

set_safe_limits(*limits:* pvp.common.message.ControlSetting)
Set the position of the max and min lines for a given value

Parameters **limits** (*ControlSetting*) – Controlsetting that has either a `min_value` or `max_value` defined

reset_start_time()
Reset start time – return the scrolling time bar to position 0

set_units(*units*)
Set displayed units

Currently only implemented for Pressure, display either in cmH2O or hPa

Parameters **units** (*'cmH2O', 'hPa'*) – unit to display pressure as

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.plot.**Plot_Container**(*plot_descriptors: Dict[*pvp.common.values.ValueName*,* pvp.common.values.Value*]*, **args*, ***kwargs*)
Container for multiple :class;`.Plot` objects

Allows user to show/hide different plots and adjust x-span (time zoom)

Note: Currently, the only unfortunately hardcoded parameter in the whole GUI is the instruction to initially hide FIO2, there should be an additional parameter in `Value` that says whether a plot should initialize as hidden or not

Methods

init_ui()	
reset_start_time()	Call *Plot.reset_start_time()* on all plots
set_duration(duration)	Set the current duration (span of the x axis) of all plots
set_plot_mode()	
set_safe_limits(control)	Try to set horizontal alarm limits on all relevant plots
toggle_plot(state)	Toggle the visibility of a plot.
update_value(vals)	Try to update all plots who have new sensorvalues

Todo: Currently, colors are set to alternate between orange and light blue on initialization, but they don't update when plots are shown/hidden, so the alternating can be lost and colors can look random depending on what's selcted.

Parameters **plot_descriptors** (*typing.Dict[ValueName, Value]*) – dict of *Value* items to plot

plots
Dict mapping *ValueName* s to *Plot* s

Type dict

slider
slider used to set x span

Type PySide2.QtWidgets.QSlider

init_ui()

update_value(*vals: pvp.common.message.SensorValues*)
Try to update all plots who have new sensorvalues

Parameters **vals** (*SensorValues*) – Sensor Values to update plots with

toggle_plot(*state: bool*)
Toggle the visibility of a plot.

get the name of the plot from the `sender`'s `objectName`

Parameters **state** (*bool*) – Whether the plot should be visible (True) or not (False)

set_safe_limits(*control: pvp.common.message.ControlSetting*)
Try to set horizontal alarm limits on all relevant plots

Parameters **control** (*ControlSetting*) – with either `min_value` or `max_value` set

Returns:

set_duration(*duration: float*)
Set the current duration (span of the x axis) of all plots

Also make sure that the text box and slider reflect this duration

Parameters **duration** (*float*) – new duration to set

Returns:

reset_start_time()
Call *Plot.reset_start_time()* on all plots

staticMetaObject = <PySide2.QtCore.QMetaObject object>

set_plot_mode()

Todo: switch between longitudinal timeseries and overlaid by breath cycle!!!

Components

Very basic components used by other widgets.

These are relatively sparsely documented because their operation is mostly self-explanatory

Classes

DoubleSlider([decimals])	Slider capable of representing floats
EditableLabel([parent])	Editable label
KeyPressHandler	Custom key press handler

continues on next page

Table 29 – continued from previous page

OnOffButton(state_labels, str] =, toggled, . . .)	Simple extension of toggle button with styling for clearer 'ON' vs 'OFF'
QHLine([parent, color])	with respct to https://stackoverflow.com/a/51057516
QVLine([parent, color])	

class pvp.gui.widgets.components.**DoubleSlider**(*decimals=1*, **args*, ***kargs*)

Slider capable of representing floats

Ripped off from and https://stackoverflow.com/a/50300848 ,

Thank you!!! **Methods**

_maximum()
_minimum()
_singleStep()
emitDoubleValueChanged()
maximum(self)
minimum(self)
setDecimals(decimals)
setMaximum(self, arg__1)
setMinimum(self, arg__1)
setSingleStep(self, arg__1)
setValue(self, arg__1)
singleStep(self)
value(self)

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

setDecimals(*decimals*)

emitDoubleValueChanged()

value(*self*) → int

setMinimum(*self*, *arg__1: int*)

setMaximum(*self*, *arg__1: int*)

minimum(*self*) → int

_minimum()

maximum(*self*) → int

_maximum()

setSingleStep(*self*, *arg__1: int*)

singleStep(*self*) → int

_singleStep()

setValue(*self*, *arg__1: int*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.components.**KeyPressHandler**

Custom key press handler https://gist.github.com/mfessenden/baa2b87b8addb0b60e54a11c1da48046 **Methods**

`eventFilter`(self, watched, event)	

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

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

eventFilter(*self*, *watched: PySide2.QtCore.QObject*, *event: PySide2.QtCore.QEvent*) → bool

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.components.**EditableLabel**(*parent=None*, ***kwargs*)

Editable label https://gist.github.com/mfessenden/baa2b87b8addb0b60e54a11c1da48046 **Methods**

`create_signals`()	
`escapePressedAction`()	Escape event handler
`labelPressedEvent`(event)	Set editable if the left mouse button is clicked
`labelUpdatedAction`()	Indicates the widget text has been updated
`returnPressedAction`()	Return/enter event handler
`setEditable`(editable)	
`setLabelEditableAction`()	Action to make the widget editable
`setText`(text)	Standard QLabel text setter
`text`()	Standard QLabel text getter

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

create_signals()

text()

Standard QLabel text getter

setText(*text*)

Standard QLabel text setter

labelPressedEvent(*event*)

Set editable if the left mouse button is clicked

setLabelEditableAction()

Action to make the widget editable

setEditable(*editable: bool*)

labelUpdatedAction()

Indicates the widget text has been updated

returnPressedAction()

Return/enter event handler

escapePressedAction()

Escape event handler

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.components.**QHLine**(*parent=None*, *color='#FFFFFF'*)

with respct to https://stackoverflow.com/a/51057516 **Methods**

`setColor`(color)	

setColor(*color*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.components.**QVLine**(*parent=None*, *color='#FFFFFF'*)

Methods

setColor(color)	

setColor(*color*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.components.**OnOffButton**(*state_labels: Tuple[str, str] = 'ON', 'OFF'*, *toggled: bool = False*, **args*, ***kwargs*)

Simple extension of toggle button with styling for clearer 'ON' vs 'OFF'

Parameters

- **state_labels** (*tuple*) – tuple of strings to set when toggled and untoggled
- **toggled** (*bool*) – initialize the button as toggled
- ***args** – passed to QPushButton
- ****kwargs** – passed to QPushButton

Methods

__init__(state_labels, str] =, toggled, . . .)	**param state_labels** tuple of strings to set when toggled and untoggled
set_state(state)	

__init__(*state_labels: Tuple[str, str] = 'ON', 'OFF'*, *toggled: bool = False*, **args*, ***kwargs*)

Parameters

- **state_labels** (*tuple*) – tuple of strings to set when toggled and untoggled
- **toggled** (*bool*) – initialize the button as toggled
- ***args** – passed to QPushButton
- ****kwargs** – passed to QPushButton

set_state(*state: bool*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

Dialog

Function to display a dialog to the user and receive feedback!

Functions

pop_dialog(message, sub_message, modality, . . .)	Creates a dialog box to display a message.

`pvp.gui.widgets.dialog.`**`pop_dialog`**(*message: str, sub_message: str = None, modality: <class 'PySide2.QtCore.Qt.WindowModality'> = PySide2.QtCore.Qt.WindowModality.NonModal, buttons: <class 'PySide2.QtWidgets.QMessageBox.StandardButton'> = PySide2.QtWidgets.QMessageBox.StandardButton.Ok, default_button: <class 'PySide2.QtWidgets.QMessageBox.StandardButton'> = PySide2.QtWidgets.QMessageBox.StandardButton.Ok*)

Creates a dialog box to display a message.

Note: This function does *not* call *.exec_* on the dialog so that it can be managed by the caller.

Parameters

- **`message`** (*`str`*) – Message to be displayed
- **`sub_message`** (*`str`*) – Smaller message displayed below main message (InformativeText)
- **`modality`** (*`QtCore.Qt.WindowModality`*) – Modality of dialog box - QtCore.Qt.NonModal (default) is unblocking, QtCore.Qt.WindowModal is blocking
- **`buttons`** (*`QtWidgets.QMessageBox.StandardButton`*) – Buttons for the window, can be `|` ed together
- **`default_button`** (*`QtWidgets.QMessageBox.StandardButton`*) – one of `buttons` , the highlighted button

Returns QtWidgets.QMessageBox

1.1.9.3 GUI Stylesheets

Data

`MONITOR_UPDATE_INTERVAL`	(float): inter-update interval (seconds) for `Monitor`

Functions

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

`pvp.gui.styles.`**`MONITOR_UPDATE_INTERVAL = 0.5`**

inter-update interval (seconds) for `Monitor`

Type (float)

`pvp.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.

The GUI is written using PySide2 and consists of one main *`PVP_Gui`* object that instantiates a series of *GUI Widgets*. The GUI is responsible for setting ventilation control parameters and sending them to the `controller` (see *`set_control()`*), as well as receiving and displaying sensor values (*`get_sensors()`*).

The GUI also feeds the *`Alarm_Manager`* *`SensorValues`* objects so that it can compute alarm state. The

Alarm_Manager reciprocally updates the GUI with *Alarm* s (*PVP_Gui.handle_alarm()*) and Alarm limits (*PVP_Gui.limits_updated()*).

The main **polling loop** of the GUI is *PVP_Gui.update_gui()* which queries the controller for new *SensorValues* and distributes them to all listening widgets (see method documentation for more details). The rest of the GUI is event driven, usually with Qt Signals and Slots.

The GUI is **configured** by the *values* module, in particular it creates

- *Display* widgets in the left "sensor monitor" box from all *Value* s in *DISPLAY_MONITOR* ,
- *Display* widgets in the right "control" box from all *Value* s in *DISPLAY_CONTROL* , and
- *Plot* widgets in the center plot box from all *Value* s in `PLOT`

The GUI is not intended to be launched alone, as it needs an active `coordinator` to communicate with the controller process and a few prelaunch preparations (*launch_gui()*). PVP should be started like:

```
python3 -m pvp.main
```

1.1.9.4 Module Overview

1.1.9.5 Screenshot

images/gui_overview_v1_1920px.png

1.1.10 Controller

Classes

Balloon_Simulator(peep_valve)	Physics simulator for inflating a balloon with an attached PEEP valve.
ControlModuleBase(save_logs, flush_every)	Abstract controller class for simulation/hardware.
ControlModuleDevice([save_logs, ...])	Uses ControlModuleBase to control the hardware.
ControlModuleSimulator([simulator_dt, ...])	Controlling Simulation.

Functions

get_control_module([sim_mode, simulator_dt])	Generates control module.

class pvp.controller.control_module.**ControlModuleBase**(*save_logs: bool = False*, *flush_every: int = 10*)

Bases: object

Abstract controller class for simulation/hardware.

1. General notes: All internal variables fall in three classes, denoted by the beginning of the variable:

Methods

__analyze_last_waveform()	This goes through the last waveform, and updates the internal variables: VTE, PEEP, PIP, PIP_TIME, I_PHASE, FIRST_PEEP and BPM.
__calculate_control_signal_in(dt)	Calculates the PID control signal by: - Combining the the three gain parameters.
__get_PID_error(ytarget, yis, dt, RC)	Calculates the three terms for PID control.
__save_values()	Helper function to reorganize key parameters in the main PID control loop, into a *SensorValues* object, that can be stored in the logfile, using a method from the DataLogger.
__start_new_breathcycle()	Some housekeeping.
__test_for_alarms()	Implements tests that are to be executed in the main control loop: - Test for HAPA - Test for Technical Alert, making sure sensor values are plausible - Test for Technical Alert, make sure continuous in contact Currently: Alarms are time.time() of first occurance.
_PID_update(dt)	This instantiates the PID control algorithms.
__init__(save_logs, flush_every)	Initializes the ControlModuleBase class.
_control_reset()	Resets the internal controller cycle to zero, i.e.
_controls_from_COPY()	
_get_control_signal_in()	Produces the INSPIRATORY control-signal that has been calculated in *__calculate_control_signal_in(dt)*
_get_control_signal_out()	Produces the EXPIRATORY control-signal for the different states, i.e.
_initialize_set_to_COPY()	Makes a copy of internal variables.
_sensor_to_COPY()	
_start_mainloop()	Prototype method to start main PID loop.

continues on next page

Table 41 – continued from previous page

`get_alarms()`	A method callable from the outside to get a copy of the alarms, that the controller checks: High airway pressure, and technical alarms.
`get_control(control_setting_name)`	A method callable from the outside to get current control settings.
`get_heartbeat()`	Returns an independent heart-beat of the controller, i.e.
`get_past_waveforms()`	Public method to return a list of past waveforms from *__cycle_waveform_archive*.
`get_sensors()`	A method callable from the outside to get a copy of sensorValues
`interrupt()`	If the controller seems stuck, this generates a new thread, and starts the main loop.
`is_running()`	Public Method to assess whether the main loop thread is running.
`set_breath_detection(breath_detection)`	
`set_control(control_setting)`	A method callable from the outside to set alarms.
`start()`	Method to start *_start_mainloop* as a thread.
`stop()`	Method to stop the main loop thread, and close the logfile.

- *COPY_varname*: These are copies (for safe threading purposes) 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. Set and get values. 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.

Public Methods:

- *get_sensors()*: Returns a copy of the current sensor values.
- *get_alarms()*: Returns a List of all alarms, active and logged
- *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.
- *start()*: Starts the main-loop of the controller
- *stop()*: Stops the main-loop of the controller
- *set_control()*: Set the control
- *interrupt()*: Interrupt the controller, and re-spawns the thread. Used to restart a stuck controller
- *is_running()*: Returns a bool whether the main-thread is running

- *get_heartbeat()*: Returns a heartbeat, more specifically, the continuously increasing iteration-number of the main control loop.

Initializes the ControlModuleBase class.

Parameters

- **save_logs** (*bool, optional*) – Should sensor data and controls should be saved with the DataLogger? Defaults to False.
- **flush_every** (*int, optional*) – Flush and rotate logs every n breath cycles. Defaults to 10.

Raises **alert** – [description]

__init__(*save_logs: bool = False*, *flush_every: int = 10*)
Initializes the ControlModuleBase class.

Parameters

- **save_logs** (*bool, optional*) – Should sensor data and controls should be saved with the DataLogger? Defaults to False.
- **flush_every** (*int, optional*) – Flush and rotate logs every n breath cycles. Defaults to 10.

Raises **alert** – [description]

_initialize_set_to_COPY()
Makes a copy of internal variables. This is used to facilitate threading

_sensor_to_COPY()

_controls_from_COPY()

__analyze_last_waveform()

This goes through the last waveform, and updates the internal variables: VTE, PEEP, PIP, PIP_TIME, I_PHASE, FIRST_PEEP and BPM.

get_sensors() → *pvp.common.message.SensorValues*
A method callable from the outside to get a copy of sensorValues

Returns A set of current sensorvalues, handeled by the controller.

Return type *SensorValues*

get_alarms() → Union[None, Tuple[*pvp.alarm.alarm.Alarm*]]
A method callable from the outside to get a copy of the alarms, that the controller checks: High airway pressure, and technical alarms.

Returns A tuple of alarms

Return type typing.Union[None, typing.Tuple[*Alarm*]]

set_control(*control_setting: pvp.common.message.ControlSetting*)
A method callable from the outside to set alarms. This updates the entries of COPY with new control values.

Parameters **control_setting** (ControlSetting) – [description]

get_control(*control_setting_name: pvp.common.values.ValueName*) → *pvp.common.message.ControlSetting*
A method callable from the outside to get current control settings. This returns values of COPY to the outside world.

Parameters **control_setting_name** (ValueName) – The specific control asked for

Returns ControlSettings-Object that contains relevant data

Return type *ControlSetting*

set_breath_detection (*breath_detection: bool*)

__get_PID_error (*ytarget, yis, dt, RC*)

Calculates the three terms for PID control. Also takes a timestep "dt" on which the integral-term is smoothed.

Parameters

- **ytarget** (*float*) – target value of pressure
- **yis** (*float*) – current value of pressure
- **dt** (*float*) – timestep
- **RC** (*float*) – time constant for calculation of integral term.

__calculate_control_signal_in (*dt*)

Calculates the PID control signal by:

- Combining the the three gain parameters.
- And smoothing the control signal with a moving window of three frames (~10ms)

Parameters **dt** (*float*) – timestep

_get_control_signal_in ()

Produces the INSPIRATORY control-signal that has been calculated in *__calculate_control_signal_in(dt)*

Returns the numerical control signal for the inspiratory prop valve

Return type float

_get_control_signal_out ()

Produces the EXPIRATORY control-signal for the different states, i.e. open/close

Returns numerical control signal for expiratory side: open (1) close (0)

Return type float

_control_reset ()

Resets the internal controller cycle to zero, i.e. restarts the breath cycle. Used for autonomous breath detection.

__test_for_alarms ()

Implements tests that are to be executed in the main control loop:

- Test for HAPA
- Test for Technical Alert, making sure sensor values are plausible
- Test for Technical Alert, make sure continuous in contact

Currently: Alarms are time.time() of first occurance.

__start_new_breathcycle ()

Some housekeeping. This has to be executed when the next breath cycles starts:

- starts new breathcycle

- initializes newe __cycle_waveform
- analyzes last breath waveform for PIP, PEEP etc. with *__analyze_last_waveform()*
- flushes the logfile

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

1) Rise to PIP, speed is controlled by flow (variable: *__SET_PIP_GAIN*)
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 and saved

Parameters **dt** (*float*) – timesstep since last update

__save_values()
Helper function to reorganize key parameters in the main PID control loop, into a *SensorValues* object, that can be stored in the logfile, using a method from the DataLogger.

get_past_waveforms()
Public method to return a list of past waveforms from *__cycle_waveform_archive*. Note: After calling this function, archive is emptied! The format is

- Returns a list of [Nx3] waveforms, of [time, pressure, volume]
- Most recent entry is waveform_list[-1]

Returns [Nx3] waveforms, of [time, pressure, volume]

Return type list

_start_mainloop()
Prototype method to start main PID loop. Will depend on simulation or device, specified below.

start()
Method to start *_start_mainloop* as a thread.

stop()
Method to stop the main loop thread, and close the logfile.

interrupt()
If the controller seems stuck, this generates a new thread, and starts the main loop. No parameters have changed.

is_running()
Public Method to assess whether the main loop thread is running.

Returns Return true if and only if the main thread of controller is running.

Return type bool

get_heartbeat()
Returns an independent heart-beat of the controller, i.e. the internal loop counter incremented in *_start_mainloop*.

Returns exact value of *self._loop_counter*

Return type int

class pvp.controller.control_module.**ControlModuleDevice**(*save_logs=True*, *flush_every=10*, *config_file=None*)

Bases: *pvp.controller.control_module.ControlModuleBase*

Uses ControlModuleBase to control the hardware.

Initializes the ControlModule for the physical system. Inherits methods from ControlModuleBase

Parameters

- **save_logs** (*bool, optional*) – Should logs be kept? Defaults to True.
- **flush_every** (*int, optional*) – How often are log-files to be flushed, in units of main-loop-itertions? Defaults to 10.
- **config_file** (*str, optional*) – Path to device config file, e.g. 'pvp/io/config/dinky-devices.ini'. Defaults to None.

Methods

__init__([save_logs, flush_every, config_file])	Initializes the ControlModule for the physical system.
_get_HAL()	Get sensor values from HAL, decorated with timeout.
_sensor_to_COPY()	Copies the current measurements to`COPY_sensor_values`, so that it can be queried from the outside.
_set_HAL(valve_open_in, valve_open_out)	Set Controls with HAL, decorated with a timeout.
_start_mainloop()	This is the main loop.
set_valves_standby()	This returns valves back to normal setting (in: closed, out: open)

__init__(*save_logs=True*, *flush_every=10*, *config_file=None*)

Initializes the ControlModule for the physical system. Inherits methods from ControlModuleBase

Parameters

- **save_logs** (*bool, optional*) – Should logs be kept? Defaults to True.
- **flush_every** (*int, optional*) – How often are log-files to be flushed, in units of main-loop-itertions? Defaults to 10.
- **config_file** (*str, optional*) – Path to device config file, e.g. 'pvp/io/config/dinky-devices.ini'. Defaults to None.

_sensor_to_COPY()

Copies the current measurements to`COPY_sensor_values`, so that it can be queried from the outside.

_set_HAL(*valve_open_in*, *valve_open_out*)

Set Controls with HAL, decorated with a timeout.

As hardware communication is the speed bottleneck. this code is slightly optimized in so far as only changes are sent to hardware.

Parameters

- **valve_open_in** (*float*) – setting of the inspiratory valve; should be in range [0,100]
- **valve_open_out** (*float*) – setting of the expiratory valve; should be 1/0 (open and close)

_get_HAL()
Get sensor values from HAL, decorated with timeout. As hardware communication is the speed bottleneck. this code is slightly optimized in so far as some sensors are queried only in certain phases of the breatch cycle. This is done to run the primary PID loop as fast as possible:

- pressure is always queried
- Flow is queried only outside of inspiration
- In addition, oxygen is only read every 5 seconds.

set_valves_standby()
This returns valves back to normal setting (in: closed, out: open)

_start_mainloop()
This is the main loop. This method should be run as a thread (see the *start()* method in *ControlModuleBase*)

class pvp.controller.control_module.**Balloon_Simulator**(*peep_valve*)
Bases: object

Physics simulator for inflating a balloon with an attached PEEP valve. 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 on *variable*.
_reset()	Resets Balloon to default settings.
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 on *variable*. It is used as model for fluctuations in measurement variables.

Parameters

- **variable** (*float*) – value of a variable at previous time step
- **dt** (*float*) – timestep
- **mu** (*float)*) – mean
- **sigma** (*float*) – noise amplitude
- **tau** (*float*) – time scale

Returns value of "variable" at next time step

Return type float

_reset()
Resets Balloon to default settings.

class pvp.controller.control_module.**ControlModuleSimulator**(*simulator_dt=None*, *peep_valve_setting=5*)

Bases: *pvp.controller.control_module.ControlModuleBase*

Controlling Simulation.

Initializes the ControlModuleBase with the simple simulation (for testing/dev).

Parameters

- **simulator_dt** (*float*, *optional*) – timestep between updates. Defaults to None.
- **peep_valve_setting** (*int*, *optional*) – Simulates action of a PEEP valve. Pressure cannot fall below. Defaults to 5.

Methods

__SimulatedPropValve(x)	This simulates the action of a proportional valve.
__SimulatedSolenoid(x)	This simulates the action of a two-state Solenoid valve.
__init__([simulator_dt, peep_valve_setting])	Initializes the ControlModuleBase with the simple simulation (for testing/dev).
_sensor_to_COPY()	Make the sensor value object from current (simulated) measurements
_start_mainloop()	This is the main loop.

__init__(*simulator_dt=None*, *peep_valve_setting=5*)

Initializes the ControlModuleBase with the simple simulation (for testing/dev).

Parameters

- **simulator_dt** (*float*, *optional*) – timestep between updates. Defaults to None.
- **peep_valve_setting** (*int*, *optional*) – Simulates action of a PEEP valve. Pressure cannot fall below. Defaults to 5.

__SimulatedPropValve(*x*)

This simulates the action of a proportional valve. Flow-current-curve eye-balled from generic prop vale with logistic activation.

Parameters **x** (*float*) – A control variable [like pulse-width-duty cycle or mA]

Returns flow through the valve

Return type float

__SimulatedSolenoid(*x*)

This simulates the action of a two-state Solenoid valve.

Parameters **x** (*float*) – If x==0: valve closed; x>0: flow set to "1"

Returns current flow

Return type float

_sensor_to_COPY()

Make the sensor value object from current (simulated) measurements

_start_mainloop()

This is the main loop. This method should be run as a thread (see the *start()* method in *ControlModuleBase*)

pvp.controller.control_module.**get_control_module**(*sim_mode=False*, *simulator_dt=None*)

Generates control module.

Parameters

- **sim_mode** (*bool, optional*) – if true: returns simulation, else returns hardware. Defaults to False.
- **simulator_dt** (*float, optional*) – a timescale for thee simulation. Defaults to None.

Returns Either configured for simulation, or physical device.

Return type ControlModule-Object

1.1.11 common module

1.1.11.1 Values

Parameterization of variables and values used in PVP.

Value objects define the existence and behavior of values, including creating *Display* and *Plot* widgets in the GUI, and the contents of *SensorValues* and *ControlSetting* s used between the GUI and controller.

Data

CONTROL	Values to control but not monitor.
DISPLAY_CONTROL	Control values that should also have a widget created in the GUI
DISPLAY_MONITOR	Those sensor values that should also have a widget created in the GUI
PLOTS	Values that can be plotted
SENSOR	Sensor values
VALUES	Declaration of all values used by PVP

Classes

Value(name, units, abs_range, safe_range, . . .)	Class to parameterize how a value is used in PVP.
ValueName(value)	Canonical names of all values used in PVP.

class pvp.common.values.**ValueName**(*value*)

Bases: enum.Enum

Canonical names of all values used in PVP. **Attributes**

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

VTE = 9

PRESSURE = 10

FLOWOUT = 11

class pvp.common.values.**Value**(*name: str, units: str, abs_range: tuple, safe_range: tuple, decimals: int, control: bool, sensor: bool, display: bool, plot: bool = False, plot_limits: Union[None, Tuple[pvp.common.values.ValueName]] = None, control_type: Union[None, str] = None, group: Union[None, dict] = None, default: (<class 'int'>, <class 'float'>) = None*)

Bases: object

Class to parameterize how a value is used in PVP.

Sets whether a value is a sensor value, a control value, whether it should be plotted, and other details for the rest of the system to determine how to use it.

Values should only be declared in this file so that they are kept consistent with *ValueName* and to not leak stray values anywhere else in the program.

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
- **control** (*bool*) – Whether or not the value is used to control ventilation
- **sensor** (*bool*) – Whether or not the value is a measured sensor value
- **display** (*bool*) – whether the value should be created as a gui.widgets.Display widget.
- **plot** (*bool*) – whether or not the value is plottable in the center plot window

- **plot_limits** (*None*, *tuple*(ValueName)) – If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot. eg. the PIP alarm range limits should be plotted on the Pressure plot
- **control_type** (*None*, *"slider"*, *"record"*) – If a control sets whether the control should use a slider or be set by recording recent sensor values.
- **group** (*None*, *str*) – Unused currently, but to be used to create subgroups of control & display widgets
- **default** (*None*, *int*, *float*) – Default value, if any. (Not automatically set in the GUI.)

Methods

`__init__`(name, units, abs_range, safe_range, . . .)	**param name** Human-readable name of the value
`to_dict`()	Gather up all attributes and return as a dict!

Attributes

`abs_range`	tuple of ints or floats setting the logical limit of the value,
`control`	Whether or not the value is used to control ventilation
`control_type`	If a control sets whether the control should use a slider or be set by recording recent sensor values.
`decimals`	The number of decimals of precision used when displaying the value
`default`	Default value, if any.
`display`	Whether the value should be created as a `gui.widgets.Display` widget.
`group`	Unused currently, but to be used to create subgroups of control & display widgets
`name`	Human readable name of value
`plot`	whether or not the value is plottable in the center plot window
`plot_limits`	If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot.
`safe_range`	tuple of ints or floats setting the safe ranges of the value,
`sensor`	Whether or not the value is a measured sensor value

__init__(*name: str*, *units: str*, *abs_range: tuple*, *safe_range: tuple*, *decimals: int*, *control: bool*, *sensor: bool*, *display: bool*, *plot: bool = False*, *plot_limits: Union[None, Tuple[pvp.common.values.ValueName]] = None*, *control_type: Union[None, str] = None*, *group: Union[None, dict] = None*, *default: (<class 'int'>, <class 'float'>) = None*)

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
- **control** (*bool*) – Whether or not the value is used to control ventilation
- **sensor** (*bool*) – Whether or not the value is a measured sensor value
- **display** (*bool*) – whether the value should be created as a `gui.widgets.Display` widget.
- **plot** (*bool*) – whether or not the value is plottable in the center plot window
- **plot_limits** (*None, tuple(ValueName)*) – If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot. eg. the PIP alarm range limits should be plotted on the Pressure plot
- **control_type** (*None, "slider", "record"*) – If a control sets whether the control should use a slider or be set by recording recent sensor values.
- **group** (*None, str*) – Unused currently, but to be used to create subgroups of control & display widgets
- **default** (*None, int, float*) – Default value, if any. (Not automatically set in the GUI.)

property name
Human readable name of value

Returns str

property abs_range
tuple of ints or floats setting the logical limit of the value, eg. a percent between 0 and 100, (0, 100)

Returns tuple

property safe_range
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``.
```

Returns tuple

property decimals
The number of decimals of precision used when displaying the value

Returns int

property default
Default value, if any. (Not automatically set in the GUI.)

property control
Whether or not the value is used to control ventilation

Returns bool

property sensor
Whether or not the value is a measured sensor value

Returns bool

property display
Whether the value should be created as a `gui.widgets.Display` widget.

Returns bool

property control_type
If a control sets whether the control should use a slider or be set by recording recent sensor values.

Returns None, "slider", "record"

property group
Unused currently, but to be used to create subgroups of control & display widgets

Returns None, str

property plot
whether or not the value is plottable in the center plot window

Returns bool

property plot_limits
If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot. eg. the PIP alarm range limits should be plotted on the Pressure plot

Returns None, typing.Tuple[ValueName]

to_dict() → dict
Gather up all attributes and return as a dict!

Returns dict

`pvp.common.values.`**VALUES = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value obje**
Declaration of all values used by PVP

`pvp.common.values.`**SENSOR = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value obje**
Sensor values

Automatically generated as all *Value* objects in *VALUES* where `sensor == True`

`pvp.common.values.`**CONTROL = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value obje**
Values to control but not monitor.

Automatically generated as all *Value* objects in *VALUES* where `control == True`

`pvp.common.values.`**DISPLAY_MONITOR = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Va**
Those sensor values that should also have a widget created in the GUI

Automatically generated as all *Value* objects in *VALUES* where `sensor == True` and `display == True`

`pvp.common.values.`**DISPLAY_CONTROL = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Va**
Control values that should also have a widget created in the GUI

Automatically generated as all *Value* objects in *VALUES* where `control == True` and `display == True` pvp.common.values.**PLOTS = OrderedDict([(<ValueName.PRESSURE: 10>, <pvp.common.values.Value**

Values that can be plotted

Automatically generated as all *Value* objects in *VALUES* where `plot == True`

1.1.11.2 Message

Message objects that define the API between modules in the system.

- *SensorValues* are used to communicate sensor readings between the controller, GUI, and alarm manager
- *ControlSetting* is used to set ventilation controls from the GUI to the controller.

Classes

ControlSetting(name, value, min_value, . . .)	Message containing ventilation control parameters.
ControlValues(control_signal_in, . . .)	Class to save control values, analogous to SensorValues.
DerivedValues(timestamp, breath_count, . . .)	Class to save derived values, analogous to SensorValues.
SensorValues([timestamp, loop_counter, . . .])	Structured class for communicating sensor readings throughout PVP.

class pvp.common.message.**SensorValues**(*timestamp=None, loop_counter=None, breath_count=None, vals=typing.Union[NoneType, typing.Dict[ForwardRef('ValueName'), float]], **kwargs*)

Bases: `object`

Structured class for communicating sensor readings throughout PVP.

Should be instantiated with each of the *SensorValues.additional_values*, and values for all *ValueName* s in *values.SENSOR* by passing them in the `vals` kwarg. An `AssertionError` if an incomplete set of values is given.

Values can be accessed either via attribute name (`SensorValues.PIP`) or like a dictionary (`SensorValues['PIP']`)

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`
- **vals** (*None, dict*) – Dict of `{ValueName: float}` that contains current sensor readings. Can also be equivalently given as `kwargs` . if None, assumed values are being passed as kwargs, but an exception will be raised if they aren't.
- ****kwargs** – sensor readings, must be in `pvp.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`()	Return a dictionary of all sensor values and additional values

Attributes

`additional_values`	Additional attributes that are not `ValueName` s that are expected in each SensorValues message

additional_values = ('timestamp', 'loop_counter', 'breath_count')
Additional attributes that are not `ValueName` s that are expected in each SensorValues message

__init__(*timestamp=None, loop_counter=None, breath_count=None, vals=typing.Union[NoneType, typing.Dict[ForwardRef('ValueName'), float]], **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`
- **vals** (*None, dict*) – Dict of `{ValueName: float}` that contains current sensor readings. Can also be equivalently given as `kwargs` . if None, assumed values are being passed as kwargs, but an exception will be raised if they aren't.
- ****kwargs** – sensor readings, must be in `pvp.values.SENSOR.keys`

to_dict() → dict
Return a dictionary of all sensor values and additional values

Returns dict

class `pvp.common.message.`**ControlSetting**(*name: pvp.common.values.ValueName, value: float = None, min_value: float = None, max_value: float = None, timestamp: float = None, range_severity: AlarmSeverity = None*)

Bases: `object`

Message containing ventilation control parameters.

At least **one of** `value`, `min_value`, or `max_value` must be given (unlike `SensorValues` which requires all fields to be present) – eg. in the case where one is setting alarm thresholds without changing the actual set value

When a parameter has multiple alarm limits for different alarm severities, the severity should be passed to `range_severity`

Parameters

- **name** (*ValueName*) – Name of value being set
- **value** (*float*) – Value to set control

- **min_value** (*float*) – Value to set control minimum (typically used for alarm thresholds)
- **max_value** (*float*) – Value to set control maximum (typically used for alarm thresholds)
- **timestamp** (*float*) – `time.time()` control message was generated
- **range_severity** (*AlarmSeverity*) – Some control settings have multiple limits for different alarm severities, this attr, when present, specifies which is being set.

Methods

__init__(name, value, min_value, max_value, …)	Message containing ventilation control parameters.

__init__(*name: pvp.common.values.ValueName*, *value: float = None*, *min_value: float = None*, *max_value: float = None*, *timestamp: float = None*, *range_severity: AlarmSeverity = None*)

Message containing ventilation control parameters.

At least **one of** `value`, `min_value`, or `max_value` must be given (unlike *SensorValues* which requires all fields to be present) – eg. in the case where one is setting alarm thresholds without changing the actual set value

When a parameter has multiple alarm limits for different alarm severities, the severity should be passed to `range_severity`

Parameters

- **name** (*ValueName*) – Name of value being set
- **value** (*float*) – Value to set control
- **min_value** (*float*) – Value to set control minimum (typically used for alarm thresholds)
- **max_value** (*float*) – Value to set control maximum (typically used for alarm thresholds)
- **timestamp** (*float*) – `time.time()` control message was generated
- **range_severity** (*AlarmSeverity*) – Some control settings have multiple limits for different alarm severities, this attr, when present, specifies which is being set.

class pvp.common.message.**ControlValues**(*control_signal_in*, *control_signal_out*)

Bases: `object`

Class to save control values, analogous to SensorValues.

Used by the controller to save waveform data in *DataLogger.store_waveform_data()* and `ControlModuleBase.__save_values`()`

Key difference: SensorValues come exclusively from the sensors, ControlValues contains controller variables, i.e. control signals and controlled signals (the flows). :param control_signal_in: :param control_signal_out:

class pvp.common.message.**DerivedValues**(*timestamp*, *breath_count*, *I_phase_duration*, *pip_time*, *peep_time*, *pip*, *pip_plateau*, *peep*, *vte*)

Bases: `object`

Class to save derived values, analogous to SensorValues.

Used by controller to store derived values (like PIP from Pressure) in *DataLogger.store_derived_data()* and in `ControlModuleBase.__analyze_last_waveform`()`

Key difference: SensorValues come exclusively from the sensors, DerivedValues contain estimates of I_PHASE_DURATION, PIP_TIME, PEEP_time, PIP, PIP_PLATEAU, PEEP, and VTE. :param timestamp:

:param breath_count: :param I_phase_duration: :param pip_time: :param peep_time: :param pip: :param pip_plateau: :param peep: :param vte:

1.1.11.3 Loggers

Logging functionality

There are two types of loggers:

- `loggers.init_logger()` creates a standard `logging.Logger` -based logging system for debugging and recording system events, and a
- `loggers.DataLogger` - a `tables` - based class to store continuously measured sensor values.

Classes

`DataLogger`(compression_level)	Class for logging numerical respiration data and control settings.

Data

`_LOGGERS`	list of strings, which loggers have been created already.

Functions

`init_logger`(module_name, log_level, file_handler)	Initialize a logger for logging events.
`update_logger_sizes`()	Adjust each logger's `maxBytes` attribute so that the total across all loggers is `prefs.LOGGING_MAX_BYTES`

`pvp.common.loggers.`**`_LOGGERS = ['pvp.common.prefs', 'pvp.alarm.alarm_manager']`**

list of strings, which loggers have been created already.

`pvp.common.loggers.`**`init_logger`**(*module_name: str, log_level: int = None, file_handler: bool = True*) → logging.Logger

Initialize a logger for logging events.

To keep logs sensible, you should usually initialize the logger with the name of the module that's using it, eg:

```
logger = init_logger(__name__)
```

If a logger has already been initialized (ie. its name is in `loggers._LOGGERS`, return that.

otherwise configure and return the logger such that

- its `LOGLEVEL` is set to `prefs.LOGLEVEL`
- It formats logging messages with logger name, time, and logging level
- if a file handler is specified (default), create a `logging.RotatingFileHandler` according to params set in `prefs`

Parameters

- **`module_name`** (`str`) – module name used to generate filename and name logger

- **log_level** (*int*) – one of **:var:`logging.DEBUG`**, **:var:`logging.INFO`**, **:var:`logging.WARNING`**, or **:var:`logging.ERROR`**
- **file_handler** (*bool, str*) – if `True`, (default), log in `<logdir>/module_name.log` . if `False`, don't log to disk.

Returns Logger 4 u 2 use

Return type `logging.Logger`

`pvp.common.loggers.`**`update_logger_sizes`**`()`

Adjust each logger's `maxBytes` attribute so that the total across all loggers is `prefs.LOGGING_MAX_BYTES`

`class` `pvp.common.loggers.`**`DataLogger`**(*compression_level: int = 9*)

Bases: `object`

Class for logging numerical respiration data and control settings. Creates a hdf5 file with this general structure:

Methods

`__init__`(compression_level)	Initialized the coontinuous numerical logger class.
`_open_logfile`()	Opens the hdf5 file and generates the file structure.
`check_files`()	make sure that the file's are not getting too large.
`close_logfile`()	Flushes & closes the open hdf file.
`flush_logfile`()	This flushes the datapoints to the file.
`load_file`([filename])	This loads a hdf5 file, and returns data to the user as a dictionary with two keys: waveform_data and control_data
`log2csv`([filename])	Translates the compressed hdf5 into three csv files containing:
`log2mat`([filename])	Translates the compressed hdf5 into a matlab file containing a matlab struct.
`rotation_newfile`()	This rotates through filenames, similar to a ring-buffer, to make sure that the program does not run of of space/
`store_control_command`(control_setting)	Appends a datapoint to the event-table, derived from ControlSettings
`store_derived_data`(derived_values)	Appends a datapoint to the event-table, derived the continuous data (PIP, PEEP etc.)
`store_waveform_data`(sensor_values, ...)	Appends a datapoint to the file for continuous logging of streaming data.

/ root |— waveforms (group) | |— time | pressure_data | flow_out | control_signal_in | control_signal_out | FiO2 | Cycle No. | |— controls (group) | |— (time, controllsignal) | |— derived_quantities (group) | |— (time, Cycle No, I_PHASE_DURATION, PIP_TIME, PEEP_time, PIP, PIP_PLATEAU, PEEP, VTE) | |

Public Methods: close_logfile(): Flushes, and closes the logfile. store_waveform_data(SensorValues): Takes data from SensorValues, but DOES NOT FLUSH store_controls(): Store controls in the same file? TODO: Discuss flush_logfile(): Flush the data into the file

Initialized the coontinuous numerical logger class.

Parameters **compression_level** (*int, optional*) – Compression level of the hdf5 file. Defaults to 9.

__init__(*compression_level: int = 9*)
Initialized the coontinuous numerical logger class.

Parameters **compression_level** (*int, optional*) – Compression level of the hdf5 file. Defaults to 9.

_open_logfile()
Opens the hdf5 file and generates the file structure.

close_logfile()
Flushes & closes the open hdf file.

store_waveform_data(*sensor_values: SensorValues*, *control_values: ControlValues*)
Appends a datapoint to the file for continuous logging of streaming data. NOTE: Not flushed yet.

Parameters

- **sensor_values** (SensorValues) – SensorValues to be stored in the file.
- **control_values** (ControlValues) – ControlValues to be stored in the file

store_control_command(*control_setting: ControlSetting*)
Appends a datapoint to the event-table, derived from ControlSettings

Parameters **control_setting** (ControlSetting) – ControlSettings object, the content of which should be stored

store_derived_data(*derived_values: DerivedValues*)
Appends a datapoint to the event-table, derived the continuous data (PIP, PEEP etc.)

Parameters **derived_values** (DerivedValues) – DerivedValues object, the content of which should be stored

flush_logfile()
This flushes the datapoints to the file. To be executed every other second, e.g. at the end of breath cycle.

check_files()
make sure that the file's are not getting too large.

rotation_newfile()
This rotates through filenames, similar to a ringbuffer, to make sure that the program does not run of of space/

load_file(*filename=None*)
This loads a hdf5 file, and returns data to the user as a dictionary with two keys: waveform_data and control_data

Parameters **filename** (*str, optional*) – Path to a hdf5-file. If none is given, uses currently open file. Defaults to None.

Returns Containing the data arranged as ` {"waveform_data": waveform_data, "control_data": control_data, "derived_data": derived_data}`

Return type dictionary

log2mat(*filename=None*)
Translates the compressed hdf5 into a matlab file containing a matlab struct. This struct has the same structure as the hdf5 file, but is not compressed. Use for any file:

dl = DataLogger() dl.log2mat(filename)

The file is saved at the same path as *.mat* file, where the content is represented as matlab-structs.

Parameters **filename** (*str, optional*) – Path to a hdf5-file. If none is given, uses currently open file. Defaults to None.

log2csv (*filename=None*)

Translates the compressed hdf5 into three csv files containing:

- waveform_data (measurement once per cycle)
- derived_quantities (PEEP, PIP etc.)
- control_commands (control commands sent to the controller)

This approximates the structure contained in the hdf5 file. Use for any file:

dl = DataLogger() dl.log2csv(filename)

Parameters **filename** (*str, optional*) – Path to a hdf5-file. If none is given, uses currently open file. Defaults to None.

1.1.11.4 Prefs

Prefs set configurable parameters used throughout PVP.

See *prefs._DEFAULTS* for description of all available parameters

Prefs are stored in a .json file, by default located at `~/pvp/prefs.json` . Prefs can be manually changed by editing this file (when the system is not running, when the system is running use *prefs.set_pref()*).

When any module in pvp is first imported, the *prefs.init()* function is called that

- Makes any directories listed in *prefs._DIRECTORIES*
- Declares all prefs as their default values from *prefs._DEFAULTS* to ensure they are always defined
- Loads the existing `prefs.json` file and updates values from their defaults

Prefs can be gotten and set from anywhere in the system with *prefs.get_pref()* and *prefs.set_pref()* . Prefs are stored in a `multiprocessing.Manager` dictionary which makes these methods both thread- and process-safe. Whenever a pref is set, the `prefs.json` file is updated to reflect the new value, so preferences are durable between runtimes.

Additional `prefs` should be added by adding an entry in the *prefs._DEFAULTS* dict rather than hardcoding them elsewhere in the program.

Data

LOADED	bool: flag to indicate whether prefs have been loaded (and thus *set_pref()* should write to disk).
_DEFAULTS	Declare all available parameters and set default values.
_DIRECTORIES	Directories to ensure are created and added to prefs.
_LOCK	`mp.Lock` : Locks access to *prefs_fn*
_LOGGER	A `logging.Logger` to log pref init and setting events
_PREFS	The dict created by *prefs._PREF_MANAGER* to store prefs.
_PREF_MANAGER	The `multiprocessing.Manager` that stores prefs during system operation

Functions

get_pref(key)	Get global configuration value
init()	Initialize prefs.
load_prefs(prefs_fn)	Load prefs from a .json prefs file, combining (and overwriting) any existing prefs, and then saves.
make_dirs()	ensures _DIRECTORIES are created and added to prefs.
save_prefs(prefs_fn)	Dumps loaded prefs to `PREFS_FN`.
set_pref(key, val)	Sets a pref in the manager and, if *prefs.LOADED* is True, calls *prefs.save_prefs()*

`pvp.common.prefs.`**`_PREF_MANAGER = <multiprocessing.managers.SyncManager object>`**
The `multiprocessing.Manager` that stores prefs during system operation

`pvp.common.prefs.`**`_PREFS = <DictProxy object, typeid 'dict'>`**
The dict created by *prefs._PREF_MANAGER* to store prefs.

`pvp.common.prefs.`**`_LOGGER = <Logger pvp.common.prefs (WARNING)>`**
A `logging.Logger` to log pref init and setting events

`pvp.common.prefs.`**`_LOCK = <Lock(owner=None)>`**
Locks access to *prefs_fn*

Type `mp.Lock`

`pvp.common.prefs.`**`_DIRECTORIES = {'DATA_DIR': '/home/docs/pvp/logs', 'LOG_DIR': '/home/docs/`**
Directories to ensure are created and added to prefs.

- `VENT_DIR`: ~/pvp - base directory for user storage
- `LOG_DIR`: ~/pvp/logs - for storage of event and alarm logs
- `DATA_DIR`: ~/pvp/data - for storage of waveform data

`pvp.common.prefs.`**`LOADED = <Synchronized wrapper for c_bool(True)>`**
flag to indicate whether prefs have been loaded (and thus *set_pref()* should write to disk).

uses a `multiprocessing.Value` to be thread and process safe.

Type bool

`pvp.common.prefs.`**`_DEFAULTS = {'BREATH_DETECTION': True, 'BREATH_PRESSURE_DROP': 4, 'CONTRO`**
Declare all available parameters and set default values. If no default, set as None.

- `PREFS_FN` - absolute path to the prefs file
- `TIME_FIRST_START` - time when the program has been started for the first time
- `VENT_DIR`: ~/pvp - base directory for user storage
- `LOG_DIR`: ~/pvp/logs - for storage of event and alarm logs
- `DATA_DIR`: ~/pvp/data - for storage of waveform data
- `LOGGING_MAX_BYTES` : the **total** storage space for all loggers – each logger gets `LOGGING_MAX_BYTES/len(loggers)` space (2GB by default)
- `LOGGING_MAX_FILES` : number of files to split each logger's logs across (default: 5)
- `LOGLEVEL`: One of `('DEBUG', 'INFO', 'WARNING', 'EXCEPTION')` that sets the minimum log level that is printed and written to disk
- `TIMEOUT`: timeout used for timeout decorators on time-sensitive operations (in seconds, default 0.05)

- `HEARTBEAT_TIMEOUT`: Time between heartbeats between GUI and controller after which contact is assumed to be lost (in seconds, default 0.02)
- `GUI_STATE_FN`: Filename of gui control state file, relative to `VENT_DIR` (default: gui_state.json)
- `GUI_UPDATE_TIME`: Time between calls of `PVP_Gui.update_gui()` (in seconds, default: 0.05)
- `ENABLE_DIALOGS`: Enable all GUI dialogs – set as False when testing on virtual frame buffer that doesn't support them (default: True and should stay that way)
- `ENABLE_WARNINGS`: Enable user warnings and value change confirmations (default: True)
- `CONTROLLER_MAX_FLOW`: Maximum flow, above which the controller considers a sensor error (default: 10)
- `CONTROLLER_MAX_PRESSURE`: Maximum pressure, above which the controller considers a sensor error (default: 100)
- `CONTROLLER_MAX_STUCK_SENSOR`: Max amount of time (in s) before considering a sensor stuck (default: 0.2)
- `CONTROLLER_LOOP_UPDATE_TIME`: Amount of time to sleep in between controller update times when using `ControlModuleDevice` (default: 0.0)
- `CONTROLLER_LOOP_UPDATE_TIME_SIMULATOR`: Amount of time to sleep in between controller updates when using `ControlModuleSimulator` (default: 0.005)
- `CONTROLLER_LOOPS_UNTIL_UPDATE`: Number of controller loops in between updating its externally-available `COPY` attributes retrieved by `ControlModuleBase.get_sensor()` et al
- `CONTROLLER_RINGBUFFER_SIZE`: Maximum number of breath cycle records to be kept in memory (default: 100)
- `COUGH_DURATION`: Amount of time the high-pressure alarm limit can be exceeded and considered a cough (in seconds, default: 0.1)
- `BREATH_PRESSURE_DROP`: Amount pressure can drop below set PEEP before being considered an autonomous breath when in breath detection mode
- `BREATH_DETECTION`: Whether the controller should detect autonomous breaths in order to reset ventilation cycles (default: True)

`pvp.common.prefs.`**`set_pref`**(*key: str*, *val*)

Sets a pref in the manager and, if `prefs.LOADED` is True, calls `prefs.save_prefs()`

Parameters

- **key** (`str`) – Name of pref key
- **val** – Value to set

`pvp.common.prefs.`**`get_pref`**(*key: str = None*)

Get global configuration value

Parameters **key** (`str, None`) – get configuration value with specific `key` . if `None` , return all config values.

`pvp.common.prefs.`**`load_prefs`**(*prefs_fn: str*)

Load prefs from a .json prefs file, combining (and overwriting) any existing prefs, and then saves.

Called on pvp import by `prefs.init()`

Also initializes `prefs._LOGGER`

Note: once this function is called, *set_pref()* will update the prefs file on disk. So if *load_prefs()* is called again at any point it should not change prefs.

Parameters **prefs_fn** (*str*) – path of prefs.json

pvp.common.prefs.**save_prefs**(*prefs_fn: str = None*)
Dumps loaded prefs to PREFS_FN.

Parameters **prefs_fn** (*str*) – Location to dump prefs. if None, use existing PREFS_FN

pvp.common.prefs.**make_dirs**()
ensures _DIRECTORIES are created and added to prefs.

pvp.common.prefs.**init**()
Initialize prefs. Called in pvp.__init__.py to ensure prefs are initialized before anything else.

1.1.11.5 Unit Conversion

Functions that convert between units

Each function should accept a single float as an argument and return a single float

Used by the GUI to display values in different units. Widgets use these as

- _convert_in functions to convert units from the base unit to the displayed unit and
- _convert_out functions to convert units from the displayed unit to the base unit.

Note: Unit conversions are cosmetic – values are always kept as the base unit internally (ie. cmH2O for pressure) and all that is changed is the displayed value in the GUI.

Functions

cmH2O_to_hPa(pressure)	Convert cmH2O to hPa
hPa_to_cmH2O(pressure)	Convert hPa to cmH2O
rounded_string(value, decimals)	Create a rounded string of a number that doesnt have trailing .0 when decimals = 0

pvp.common.unit_conversion.**cmH2O_to_hPa**(*pressure: float*) → float
Convert cmH2O to hPa

Parameters **pressure** (*float*) – Pressure in cmH2O

Returns Pressure in hPa (pressure*98.0665)

Return type float

pvp.common.unit_conversion.**hPa_to_cmH2O**(*pressure: float*) → float
Convert hPa to cmH2O

Parameters **pressure** (*float*) – Pressure in hPa

Returns Pressure in cmH2O (pressure/98.0665)

Return type float

`pvp.common.unit_conversion.`**`rounded_string`**(*value: float, decimals: int = 0*) → str

Create a rounded string of a number that doesnt have trailing .0 when decimals = 0

Parameters

- **`value`** (*`float`*) – Value to stringify
- **`decimals`** (*`int`*) – Number of decimal places to round to

Returns Clean rounded string version of number

Return type str

1.1.11.6 utils

Exceptions

`TimeoutException`	

Functions

`time_limit`(seconds)	
`timeout`(func)	Defines a decorator for a 50ms timeout.

`exception` `pvp.common.utils.`**`TimeoutException`**

Bases: `Exception`

`pvp.common.utils.`**`time_limit`**(*seconds*)

`pvp.common.utils.`**`timeout`**(*func*)

Defines a decorator for a 50ms timeout. Usage/Test:

@timeout def foo(sleeptime):

time.sleep(sleeptime)

print("hello")

1.1.11.7 fashion

Decorators for dangerous functions

Functions

`locked`(func)	Wrapper to use as decorator, handle lock logic for a
`pigpio_command`(func)	

`pvp.common.fashion.`**`locked`**(*func*)

Wrapper to use as decorator, handle lock logic for a @property

Parameters **`func`** (*`callable`*) – function to wrap

`pvp.common.fashion.`**`pigpio_command`**(*func*)

1.1.12 pvp.io package

1.1.12.1 Subpackages

1.1.12.2 Submodules

1.1.12.3 pvp.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 pvp.io.hal.**Hal**(*config_file='pvp/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 pvp.io.devices.ADS1115 (or ADS1015) as a local variable:** class_ = getattr(import_module('.devices', 'pvp.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.
`oxygen`	Returns the oxygen concentration from the primary oxygen sensor.
`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 = "pvp/io/config/devices.ini")

__init__(*config_file='pvp/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 pvp.io.devices.ADS1115 (or ADS1015) as a local variable:** class_ = getattr(import_module('.devices', 'pvp.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 = "pvp/io/config/devices.ini")

property pressure
Returns the pressure from the primary pressure sensor.

property oxygen
Returns the oxygen concentration from the primary oxygen 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.

1.1.12.4 Module contents

1.1.13 Alarm

1.1.13.1 Alarm System Overview

- Alarms are represented as `Alarm` objects, which are created and managed by the `Alarm_Manager`.
- A collection of `Alarm_Rule` s define the `Condition` s for raising `Alarm` s of different `AlarmSeverity` .
- The alarm manager is continuously fed `SensorValues` objects during `PVP_Gui.update_gui()`, which it uses to `check()` each alarm rule.
- The alarm manager emits `Alarm` objects to the `PVP_Gui.handle_alarm()` method.
- The alarm manager also updates alarm thresholds set as `Condition.depends` to `PVP_Gui.limits_updated()` when control parameters are set (eg. updates the `HIGH_PRESSURE` alarm to be triggered 15% above some set `PIP`).

1.1.13.2 Alarm Modules

Alarm Manager

The alarm manager is responsible for checking the `Alarm_Rule` s and maintaining the `Alarm` s active in the system.

Only one instance of the `Alarm_Manager` can be created at once, and if it is instantiated again, the existing object will be returned.

Classes

`Alarm_Manager`()	The Alarm Manager

class pvp.alarm.alarm_manager.**Alarm_Manager**
The Alarm Manager

The alarm manager receives `SensorValues` from the GUI via `Alarm_Manager.update()` and emits `Alarm` s to methods given by `Alarm_Manager.add_callback()` . When alarm limits are updated (ie. the `Alarm_Rule` has `depends`), it emits them to methods registered with `Alarm_Manager.add_dependency_callback()` .

On initialization, the alarm manager calls `Alarm_Manager.load_rules()`, which loads all rules defined in `alarm.ALARM_RULES`. **Attributes**

`active_alarms`	dict() -> new empty dictionary
`callbacks`	Built-in mutable sequence.
`cleared_alarms`	Built-in mutable sequence.
`dependencies`	dict() -> new empty dictionary
`depends_callbacks`	Built-in mutable sequence.
`logged_alarms`	Built-in mutable sequence.
`logger`	Instances of the Logger class represent a single logging channel.
`pending_clears`	Built-in mutable sequence.
`rules`	dict() -> new empty dictionary
`snoozed_alarms`	dict() -> new empty dictionary

Methods

`add_callback`(callback)	Assert we're being given a callable and add it to our list of callbacks.
`add_dependency_callback`(callback)	Assert we're being given a callable and add it to our list of dependency_callbacks
`check_rule`(rule, sensor_values)	`check()` the alarm rule, handle logic of raising, emitting, or lowering an alarm.
`clear_all_alarms`()	call `Alarm_Manager.deactivate_alarm()` for all active 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)	Get the severity of an Alarm
`load_rule`(alarm_rule)	Add the Alarm Rule to `Alarm_Manager.rules` and register any dependencies they have with `Alarm_Manager.register_dependency()`
`load_rules`()	Copy alarms from `alarm.ALARM_RULES` and call `Alarm_Manager.load_rule()` for each
`register_alarm`(alarm)	Be given an already created alarm and emit to callbacks.
`register_dependency`(condition, dependency, ...)	Add dependency in a Condition object to be updated when values are changed
`reset`()	Reset all conditions, callbacks, and other stateful attributes and clear alarms
`update`(sensor_values)	Call `Alarm_Manager.check_rule()` for all rules in `Alarm_Manager.rules`
`update_dependencies`(control_setting)	Update Condition objects that update their value according to some control parameter

active_alarms

{`AlarmType`: `Alarm`}

Type dict

logged_alarms
A list of deactivated alarms.

Type list

dependencies
A dictionary mapping *ValueName* s to the alarm threshold dependencies they update

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

callbacks
list of callables to send *Alarm* objects to

Type list

depends_callbacks
When we *update_dependencies()*, we send back a *ControlSetting* with the new min/max

Type list

rules
A dict mapping *AlarmType* to *Alarm_Rule* .

Type dict

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

_instance = None

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

logged_alarms: List[*pvp.alarm.alarm.Alarm*] = []

dependencies = {}

pending_clears = []

cleared_alarms = []

snoozed_alarms = {}

callbacks = []

depends_callbacks = []

rules = {}

logger = <Logger pvp.alarm.alarm_manager (WARNING)>

load_rules()

Copy alarms from *alarm.ALARM_RULES* and call *Alarm_Manager.load_rule()* for each

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

Add the Alarm Rule to *Alarm_Manager.rules* and register any dependencies they have with *Alarm_Manager.register_dependency()*

Parameters **alarm_rule** (*Alarm_Rule*) – Alarm rule to be loaded

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

Call *Alarm_Manager.check_rule()* for all rules in *Alarm_Manager.rules*

Parameters **sensor_values** (*SensorValues*) – New sensor values from the GUI

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

check() the alarm rule, handle logic of raising, emitting, or lowering an alarm.

When alarms are dismissed, an *alarm.Alarm* is emitted with `AlarmSeverity.OFF` .

- If the alarm severity has increased, emit a new alarm.
- If the alarm severity has decreased and the alarm is not latched, emit a new alarm
- If the alarm severity has decreased and the alarm is latched, check if the alarm has been manually dismissed, if it has emit a new alarm.
- If a latched alarm has been manually dismissed previously and the alarm condition is now no longer met, dismiss the alarm.

Parameters

- **rule** (*Alarm_Rule*) – Alarm rule to check
- **sensor_values** (*SensorValues*) – sent by the GUI to check against alarm rule

emit_alarm(*alarm_type: pvp.alarm.AlarmType, severity: pvp.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 'pvp.alarm.alarm.Alarm'>)*)

Mark an alarm's internal active flags and remove from *active_alarms*

Typically called internally when an alarm is being replaced by one of the same type but a different severity.

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

Parameters **alarm** (*AlarmType* , *Alarm*) – Alarm to deactivate

dismiss_alarm(*alarm_type: pvp.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: pvp.alarm.AlarmType*)
Get the severity of an Alarm

Parameters **alarm_type** (`AlarmType`) – Alarm type to check

Returns `AlarmSeverity`

register_alarm(*alarm: pvp.alarm.alarm.Alarm*)
Be given an already created alarm and emit to callbacks.

Mostly used during testing for programmatically created alarms. Creating alarms outside of the Alarm_Manager is generally discouraged.

Parameters **alarm** (`Alarm`) –

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

Parameters

- **condition** (`dict`) – Condition as defined in an `Alarm_Rule`
- **dependency** (`dict`) – either a (ValueName, attribute_name) or optionally also + transformation callable
- **severity** (`AlarmSeverity`) – severity of dependency

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

Call any `transform` functions on the attribute of the control setting specified in the depencency.

Emit another `ControlSetting` describing the new max or min or the value.

Parameters **control_setting** (`ControlSetting`) – Control setting that was changed

add_callback(*callback: Callable*)
Assert we're being given a callable and add it to our list of callbacks.

Parameters **callback** (`typing.Callable`) – Callback that accepts a single argument of an `Alarm`

add_dependency_callback(*callback: Callable*)
Assert we're being given a callable and add it to our list of dependency_callbacks

Parameters **callback** (`typing.Callable`) – Callback that accepts a `ControlSetting`

Returns:

clear_all_alarms()
call *Alarm_Manager.deactivate_alarm()* for all active alarms.

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

Alarm Objects

Alarm objects represent the state and severity of active alarms, but are otherwise intentionally quite featureless.

They are created and maintained by the *Alarm_Manager* and sent to any listeners registered in `Alarm_Manager.callbacks`.

Classes

Alarm(alarm_type, severity, start_time, …)	Representation of alarm status and parameters

class pvp.alarm.alarm.**Alarm**(*alarm_type: pvp.alarm.AlarmType, severity: pvp.alarm.AlarmSeverity, start_time: float = None, latch: bool = True, cause: list = None, value=None, message=None*)
Representation of alarm status and parameters

Parameterized by a `Alarm_Rule` and managed by `Alarm_Manager`

Parameters

- **alarm_type** (*AlarmType*) – Type of alarm
- **severity** (*AlarmSeverity*) – Severity of alarm
- **start_time** (*float*) – Timestamp of alarm start, (as generated by `time.time()`
- **cause** (`ValueName`) – The *ValueName* that caused the alarm to be fired
- **value** (*int, float*) – optional - numerical value that generated the alarm
- **message** (*str*) – optional - override default text generated by `AlarmManager`

Methods

__init__(alarm_type, severity, start_time, …)	**param alarm_type** Type of alarm
deactivate()	If active, register an end time and set as `active == False`

Attributes

alarm_type	Alarm Type, property without setter to prevent change after instantiation
id_counter	`itertools.count`: used to generate unique IDs for each alarm
severity	Alarm Severity, property without setter to prevent change after instantiation

id
unique alarm ID

Type int

end_time
If None, alarm has not ended. otherwise timestamp

Type None, float

active
Whether or not the alarm is currently active

Type bool

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

Type `itertools.count`

__init__(*alarm_type: pvp.alarm.AlarmType*, *severity: pvp.alarm.AlarmSeverity*, *start_time: float = None*, *latch: bool = True*, *cause: list = None*, *value=None*, *message=None*)

Parameters

- **alarm_type** (*AlarmType*) – Type of alarm
- **severity** (*AlarmSeverity*) – Severity of alarm
- **start_time** (*float*) – Timestamp of alarm start, (as generated by `time.time()`
- **cause** (ValueName) – The *ValueName* that caused the alarm to be fired
- **value** (*int, float*) – optional - numerical value that generated the alarm
- **message** (*str*) – optional - override default text generated by `AlarmManager`

id
unique alarm ID
Type int

end_time
If None, alarm has not ended. otherwise timestamp
Type None, float

active
Whether or not the alarm is currently active
Type bool

property severity
Alarm Severity, property without setter to prevent change after instantiation

Returns *AlarmSeverity*

property alarm_type
Alarm Type, property without setter to prevent change after instantiation

Returns *AlarmType*

deactivate()
If active, register an end time and set as `active == False` Returns:

Alarm Rule

One *Alarm_Rule* is defined for each *AlarmType* in *ALARM_RULES*.

An alarm rule defines:

- The conditions for raising different severities of an alarm
- The dependencies between set values and alarm thresholds
- The behavior of the alarm, specifically whether it is `latch` ed.

Example

As an example, we'll define a `LOW_PRESSURE` alarm with escalating severity. A `LOW` severity alarm will be raised when measured `PIP` falls 10% below set `PIP`, which will escalate to a `MEDIUM` severity alarm if measured `PIP` falls 15% below set `PIP` **and** the `LOW` severity alarm has been active for at least two breath cycles.

First we define the name and behavior of the alarm:

```
Alarm_Rule(
    name = AlarmType.LOW_PRESSURE,
    latch = False,
```

In this case, `latch == False` means that the alarm will disappear (or be downgraded in severity) whenever the conditions for that alarm are no longer met. If `latch == True`, an alarm requires manual dismissal before it is downgraded or disappears.

Next we'll define a tuple of *Condition* objects for `LOW` and `MEDIUM` severity objects.

Starting with the `LOW` severity alarm:

```
conditions = (
    (
    AlarmSeverity.LOW,
    condition.ValueCondition(
        value_name=ValueName.PIP,
        limit=VALUES[ValueName.PIP]['safe_range'][0],
        mode='min',
        depends={
            'value_name': ValueName.PIP,
            'value_attr': 'value',
            'condition_attr': 'limit',
            'transform': lambda x : x-(x*0.10)
        })
    ),
    # ... continued in next block
```

Each condition is a tuple of an (*AlarmSeverity*, *Condition*). In this case, we use a *ValueCondition* which tests whether a value is above or below a set `'max'` or `'min'`, respectively. For the low severity `LOW_PRESSURE` alarm, we test if `ValueName.PIP` is below (`mode='min'`) some `limit`, which is initialized as the low-end of `PIP`'s safe range.

We also define a condition for updating the `'limit'` of the condition (`'condition_attr' : 'limit'`), from the ``ControlSetting.value` `` field whenever `PIP` is updated. Specifically, we set the `limit` to be 10% less than the set `PIP` value by 10% with a lambda function (`lambda x : x-(x*0.10)`).

Next, we define the `MEDIUM` severity alarm condition:

```
(
AlarmSeverity.MEDIUM,
condition.ValueCondition(
    value_name=ValueName.PIP,
    limit=VALUES[ValueName.PIP]['safe_range'][0],
    mode='min'
    depends={
        'value_name': ValueName.PIP,
        'value_attr': 'value',
        'condition_attr': 'limit',
        'transform': lambda x: x - (x * 0.15)
    },
) + \
condition.CycleAlarmSeverityCondition(
    alarm_type = AlarmType.LOW_PRESSURE,
    severity   = AlarmSeverity.LOW,
    n_cycles = 2
))
```

The first `ValueCondition` is the same as in the `LOW` alarm severity condition, except that it is set 15% below `PIP`.

A second *`CycleAlarmSeverityCondition`* has been added (with +) to the *`ValueCondition`* When conditions are added together, they will only return `True` (ie. trigger an alarm) if all of the conditions are met. This condition checks that the `LOW_PRESSURE` alarm has been active at a `LOW` severity for at least two cycles.

Full source for this example and all alarm rules can be found here

Module Documentation

Class to declare alarm rules

Classes

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

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

- name of rule
- conditions: ((alarm_type, (condition_1, condition_2)), . . .)
- 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

depends	Get all ValueNames whose alarm limits depend on this alarm rule
severity	Last Alarm Severity from `.check()`
value_names	Get all ValueNames specified as value_names in alarm conditions

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

Parameters **sensor_values** –

Returns:

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

reset ()

property depends
Get all ValueNames whose alarm limits depend on this alarm rule :returns: list[ValueName]

property value_names
Get all ValueNames specified as value_names in alarm conditions

Returns list[ValueName]

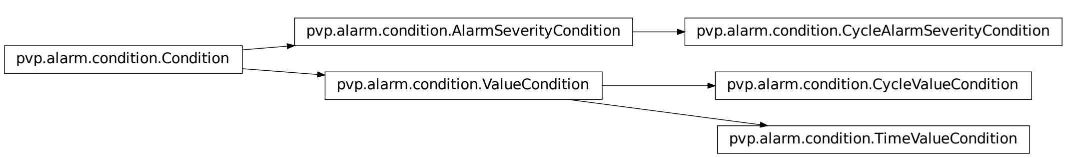
Alarm Condition

Condition objects define conditions that can raise alarms. They are used by *Alarm_Rule* s.

Each has to define a *Condition.check()* method that accepts *SensorValues* . The method should return `True` if the alarm condition is met, and `False` otherwise.

Conditions can be added (+) together to make compound conditions, and a single call to `check` will only return true if both conditions return true. If any condition in the chain returns false, evaluation is stopped and the alarm is not raised.

Conditions can



Classes

AlarmSeverityCondition(alarm_type, severity, ...)	Alarm is above or below a certain severity.
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

continues on next page

Table 76 – continued from previous page

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()

pvp.alarm.condition.**get_alarm_manager**()

class pvp.alarm.condition.**Condition**(*depends: dict = None*, **args*, ***kwargs*)

Bases: object

Base class for specifying alarm test conditions

Subclasses must define Condition.check() and Conditino.reset()

Condition objects can be added together to create compound conditions. **Methods**

__init__(depends, *args, **kwargs)	**param depends**
check(sensor_values)	Every Condition subclass needs to define this method that accepts SensorValues and returns a boolean
reset()	If a condition is stateful, need to provide some method of resetting the state

Attributes

manager	The active alarm manager, used to get status of alarms

_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
The active alarm manager, used to get status of alarms

Returns *pvp.alarm.alarm_manager.Alarm_Manager*

check (*sensor_values*) → bool
Every Condition subclass needs to define this method that accepts *SensorValues* and returns a boolean

Parameters **sensor_values** (*SensorValues*) – SensorValues used to compute alarm status

Returns bool

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

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

Bases: *pvp.alarm.condition.Condition*

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)	Check that the relevant value in SensorValues is either greater or lesser than the limit
reset()	not stateful, do nothing.

Attributes

mode	One of 'min' or 'max', defines how the incoming sensor values are compared to the set value

operator
Either the less than or greater than operators, depending on whether mode is `'min'` or `'max'`

Type callable

__init__(*value_name: pvp.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** –

operator
Either the less than or greater than operators, depending on whether mode is `'min'` or `'max'`
Type callable

property mode
One of 'min' or 'max', defines how the incoming sensor values are compared to the set value

Returns:

check(*sensor_values*)
Check that the relevant value in SensorValues is either greater or lesser than the limit

Parameters sensor_values (*SensorValues*) –

Returns bool

reset()
not stateful, do nothing.

class pvp.alarm.condition.**CycleValueCondition**(*n_cycles: int*, **args*, ***kwargs*)
Bases: *pvp.alarm.condition.ValueCondition*

Value goes out of range for a specific number of breath cycles

Parameters n_cycles (*int*) – number of cycles required

Methods

check(sensor_values)	Check if outside of range, and then check if number of breath cycles have elapsed
reset()	Reset check status and start cycle

Attributes

n_cycles	Number of cycles required

_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

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** –

operator
Either the less than or greater than operators, depending on whether mode is `'min'` or `'max'`

Type callable

property n_cycles
Number of cycles required

check(*sensor_values*) → bool
Check if outside of range, and then check if number of breath cycles have elapsed

Parameters **()** (*sensor_values*) –

Returns bool

reset()
Reset check status and start cycle

class pvp.alarm.condition.**TimeValueCondition**(*time, *args, **kwargs*)
Bases: *pvp.alarm.condition.ValueCondition*

value goes out of range for specific amount of time

> **Warning:** Not implemented!

Methods

__init__(time, *args, **kwargs)	**param time** number of seconds value must be out of range
check(sensor_values)	Check that the relevant value in SensorValues is either greater or lesser than the limit
reset()	not stateful, do nothing.

Parameters

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

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

Parameters

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

check(*sensor_values*)

Check that the relevant value in SensorValues is either greater or lesser than the limit

Parameters **sensor_values** (*SensorValues*) –

Returns bool

reset()

not stateful, do nothing.

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

Bases: *pvp.alarm.condition.Condition*

Alarm is above or below a certain severity.

Get alarm severity status from *Alarm_Manager.get_alarm_severity()* .

Parameters

- **alarm_type** (*AlarmType*) – Alarm type to check
- **severity** (*AlarmSeverity*) – Alarm severity to check against
- **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, ...)	Alarm is above or below a certain severity.
check(sensor_values)	Every Condition subclass needs to define this method that accepts *SensorValues* and returns a boolean
reset()	If a condition is stateful, need to provide some method of resetting the state

Attributes

mode	'min' returns true if the alarm is at least this value

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

Alarm is above or below a certain severity.

Get alarm severity status from `Alarm_Manager.get_alarm_severity()` .

Parameters

- **alarm_type** (`AlarmType`) – Alarm type to check
- **severity** (`AlarmSeverity`) – Alarm severity to check against
- **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

'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 <

Returns one of 'min', 'equals', or 'max'.

Return type str

check (*sensor_values*)

Every Condition subclass needs to define this method that accepts `SensorValues` and returns a boolean

Parameters **sensor_values** (`SensorValues`) – SensorValues used to compute alarm status

Returns bool

reset ()

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

class pvp.alarm.condition.**CycleAlarmSeverityCondition** (*n_cycles*, **args*, ***kwargs*)

Bases: `pvp.alarm.condition.AlarmSeverityCondition`

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)	Every Condition subclass needs to define this method that accepts `SensorValues` and returns a boolean
`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

Alarm is above or below a certain severity.

Get alarm severity status from *Alarm_Manager.get_alarm_severity()* .

Parameters

- **alarm_type** (*AlarmType*) – Alarm type to check
- **severity** (*AlarmSeverity*) – Alarm severity to check against
- **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*)
Every Condition subclass needs to define this method that accepts *SensorValues* and returns a boolean

Parameters **sensor_values** (*SensorValues*) – SensorValues used to compute alarm status

Returns bool

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

1.1.13.3 Main Alarm Module

Data

ALARM_RULES	Definitions of all *Alarm_Rule* s used by the *Alarm_Manager*

Classes

AlarmSeverity(value)	An enumeration.
AlarmType(value)	An enumeration.

class pvp.alarm.**AlarmType**(*value*)

An enumeration. **Attributes**

LOW_PRESSURE	int([x]) -> integer
HIGH_PRESSURE	int([x]) -> integer
LOW_VTE	int([x]) -> integer
HIGH_VTE	int([x]) -> integer
LOW_PEEP	int([x]) -> integer
HIGH_PEEP	int([x]) -> integer
LOW_O2	int([x]) -> integer
HIGH_O2	int([x]) -> integer
OBSTRUCTION	int([x]) -> integer
LEAK	int([x]) -> integer
SENSORS_STUCK	int([x]) -> integer
BAD_SENSOR_READINGS	int([x]) -> integer
MISSED_HEARTBEAT	int([x]) -> integer
human_name	Replace `.name` underscores with spaces

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

SENSORS_STUCK = 11

BAD_SENSOR_READINGS = 12

MISSED_HEARTBEAT = 13

property human_name

Replace `.name` underscores with spaces

class pvp.alarm.**AlarmSeverity**(*value*)

An enumeration. **Attributes**

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

HIGH = 3

MEDIUM = 2

LOW = 1

OFF = 0

TECHNICAL = -1

pvp.alarm.**ALARM_RULES = OrderedDict([(<AlarmType.LOW_PRESSURE: 1>, <pvp.alarm.rule.Alarm_Ru**

Definitions of all *Alarm_Rule* s used by the *Alarm_Manager*

See definitions here

1.1.14 coordinator module

1.1.14.1 Submodules

1.1.14.2 coordinator

Classes

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

Functions

get_coordinator([single_process, sim_mode])

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

Bases: object **Methods**

get_alarms()
get_control(control_setting_name)
get_sensors()
get_target_waveform()
is_running()
kill()
set_breath_detection(breath_detection)
set_control(control_setting)
start()
stop()

get_sensors() → *pvp.common.message.SensorValues*

get_alarms() → Union[None, Tuple[*pvp.alarm.alarm.Alarm*]]

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

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

set_breath_detection(*breath_detection: bool*)

get_target_waveform()

start()

is_running() → bool

kill()

stop()

class pvp.coordinator.coordinator.**CoordinatorLocal**(*sim_mode=False*)
Bases: *pvp.coordinator.coordinator.CoordinatorBase*

Parameters **sim_mode** –

Methods

__init__([sim_mode])	**param sim_mode**
get_alarms()	
get_control(control_setting_name)	
get_sensors()	
get_target_waveform()	
is_running()	Test whether the whole system is running
kill()	
set_breath_detection(breath_detection)	
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() → *pvp.common.message.SensorValues*

get_alarms() → Union[None, Tuple[*pvp.alarm.alarm.Alarm*]]

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

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

set_breath_detection(*breath_detection:* bool)

get_target_waveform()

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)

kill()

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

get_alarms()	
get_control(control_setting_name)	
get_sensors()	
get_target_waveform()	
is_running()	Test whether the whole system is running
kill()	Stop the coordinator and end the whole program
set_breath_detection(breath_detection)	
set_control(control_setting)	
start()	Start the coordinator.
stop()	Stop the coordinator.

get_sensors() → *pvp.common.message.SensorValues*

get_alarms() → Union[None, Tuple[*pvp.alarm.alarm.Alarm*]]

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

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

set_breath_detection(*breath_detection:* bool)

get_target_waveform()

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)

kill()
Stop the coordinator and end the whole program

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

1.1.14.3 ipc

Functions

get_alarms()	
get_control(control_setting_name)	
get_rpc_client()	
get_sensors()	
get_target_waveform()	
rpc_server_main(sim_mode, serve_event[, ...])	
set_breath_detection(breath_detection)	
set_control(control_setting)	

`pvp.coordinator.rpc.`**`get_sensors`**`()`

`pvp.coordinator.rpc.`**`get_alarms`**`()`

`pvp.coordinator.rpc.`**`set_control`**`(`*control_setting*`)`

`pvp.coordinator.rpc.`**`get_control`**`(`*control_setting_name*`)`

`pvp.coordinator.rpc.`**`set_breath_detection`**`(`*breath_detection*`)`

`pvp.coordinator.rpc.`**`get_target_waveform`**`()`

`pvp.coordinator.rpc.`**`rpc_server_main`**`(`*sim_mode, serve_event, addr='localhost', port=9533*`)`

`pvp.coordinator.rpc.`**`get_rpc_client`**`()`

1.1.14.4 process_manager

Classes

ProcessManager(sim_mode[, startCommandLine, …])	

class `pvp.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*`)`

1.1.15 Requirements

1.1.16 Datasheets & Manuals

1.1.16.1 Manuals

- Hamilton T1 Quick Guide

1.1.16.2 Other Reference Material

- Hamilton UI Simulator

1.1.17 Specs

1.1.18 Changelog

1.1.18.1 Version 0.0

v0.0.2 (April xxth, 2020)

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

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

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.

1.1.19 Contributing

1.1.20 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.

1.1.20.1 Local Build

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

Documentation will be generated into `docs`

Advertisement :)

- **pica** - high quality and fast image resize in browser.
- **babelfish** - developer friendly i18n with plurals support and easy syntax.

You will like those projects!

1.1.21 h1 Heading 8-)

1.1.21.1 h2 Heading

h3 Heading

h4 Heading

h5 Heading

h6 Heading

1.1.21.2 Horizontal Rules

1.1.21.3 Emphasis

This is bold text

This is bold text

This is italic text

This is italic text

1.1.21.4 Blockquotes

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

1.1.21.5 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.`

1.1.21.6 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

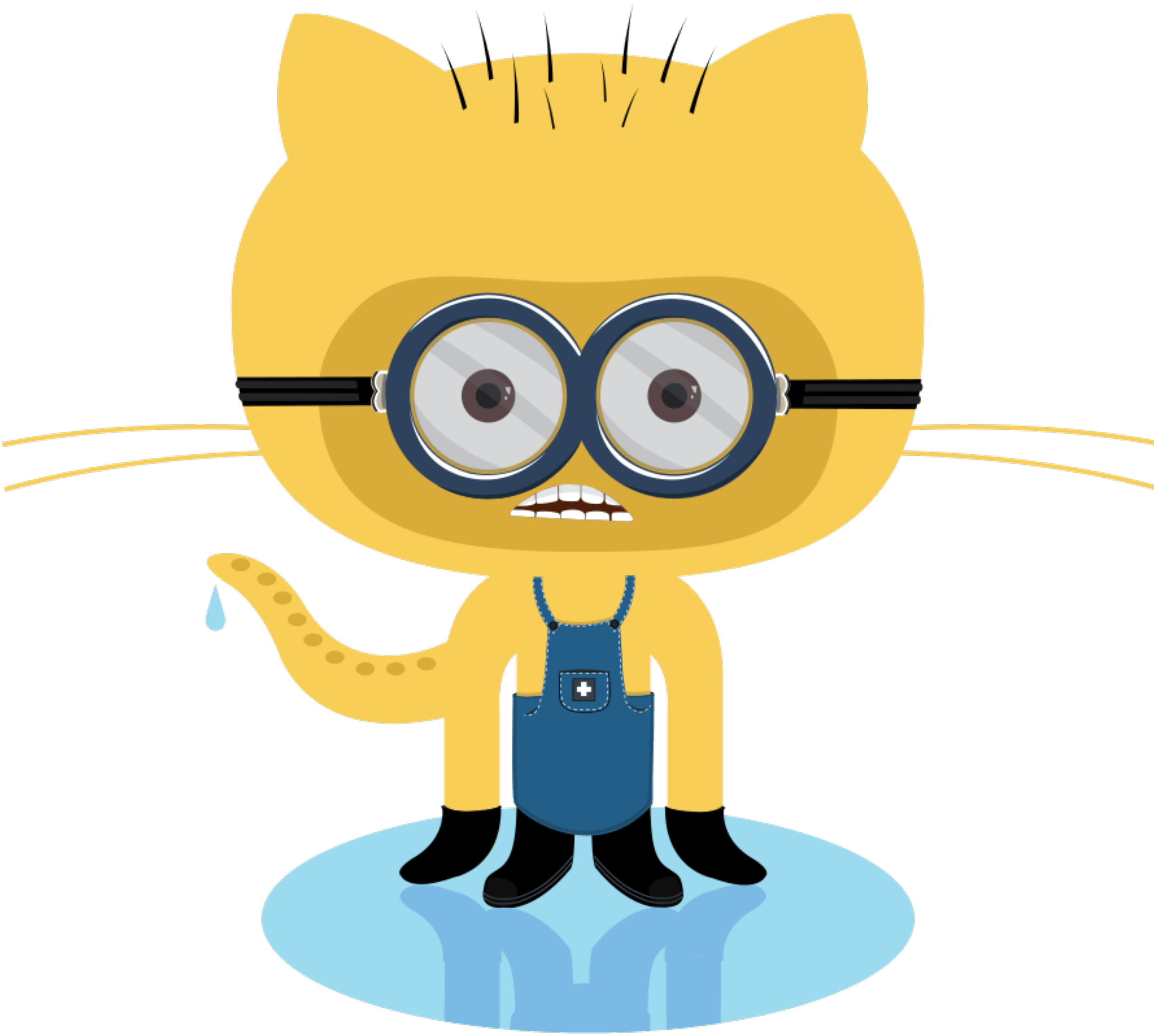
```
var foo = function (bar) {
  return bar++;
};

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

1.1.21.7 Links

link text

link with title

1.1.21.8 Images



Minion

Like links, Images also have a footnote style syntax

Alt text

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

1.1.22 Index

- genindex
- modindex

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

I

K

L

M

N

O

P

Q

R

S

T

U

V