PVP

Release 0.0.0

jonny saunders et al

Aug 16, 2020

OVERVIEW

CHAPTER

ONE

PVP CONTROL PSEUDOCODE

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

1.1 Pressure control parameters

Set in GUI

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

Set by hardware

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

Derived parameters

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

Monitored variables

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

Alarms

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

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

1.2 Hardware

Sensors

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

Actuators

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

1.3 Pressure control loop

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

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

5. Repeat from step 1.

CHAPTER

TWO

HARDWARE

2.1 Mechanical Diagram

2.1.1 Sensors | Hardware

2.1.1.1 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.

2.1.1.2 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
- – U3
  - – Amphenol 1 PSI-D1-4V-MINI differential pressure sensor
  - – Analog output differential pressure sensor
  - – above
- – U2

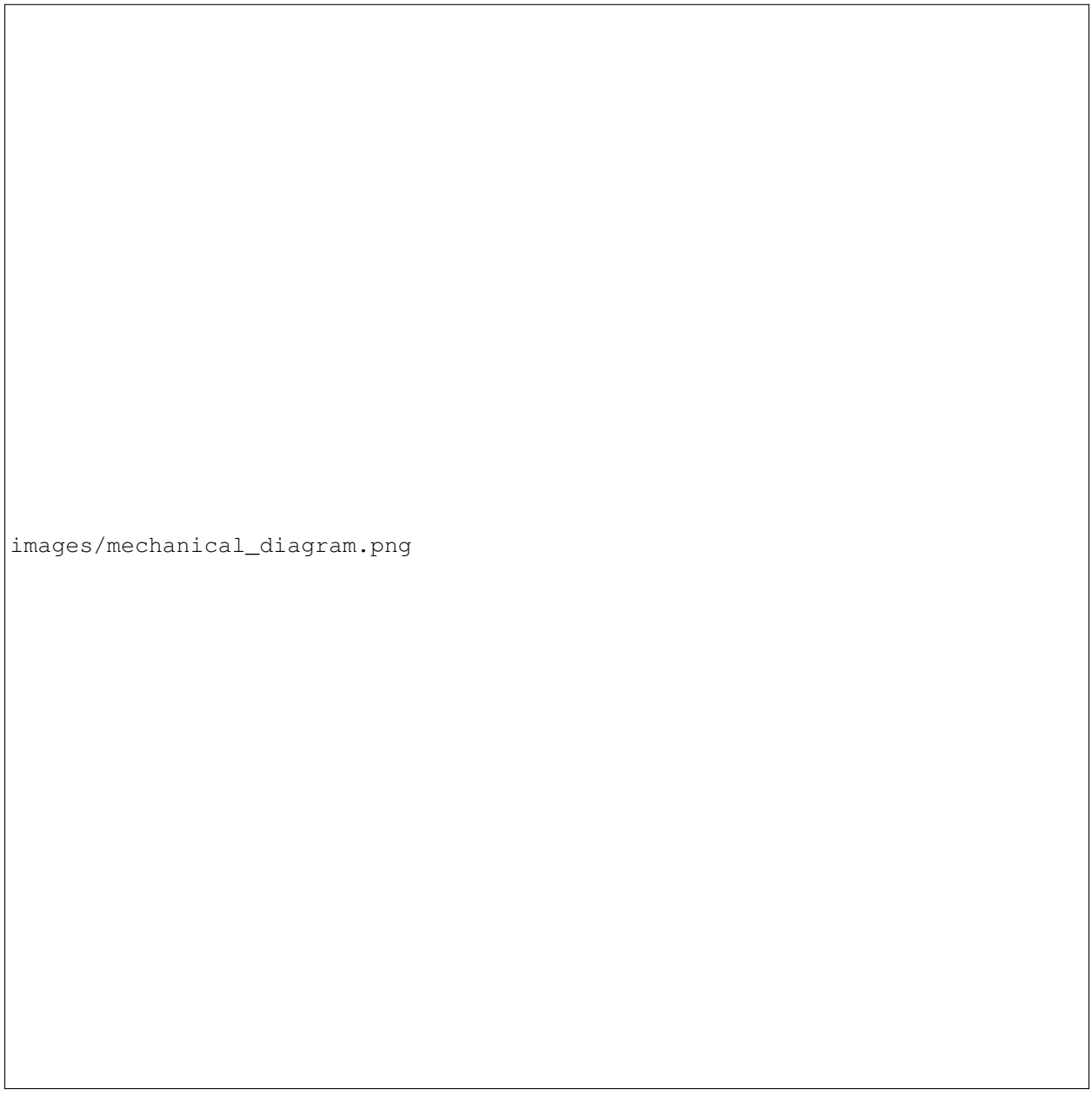

Fig. 1: Schematic diagram of main mechanical components

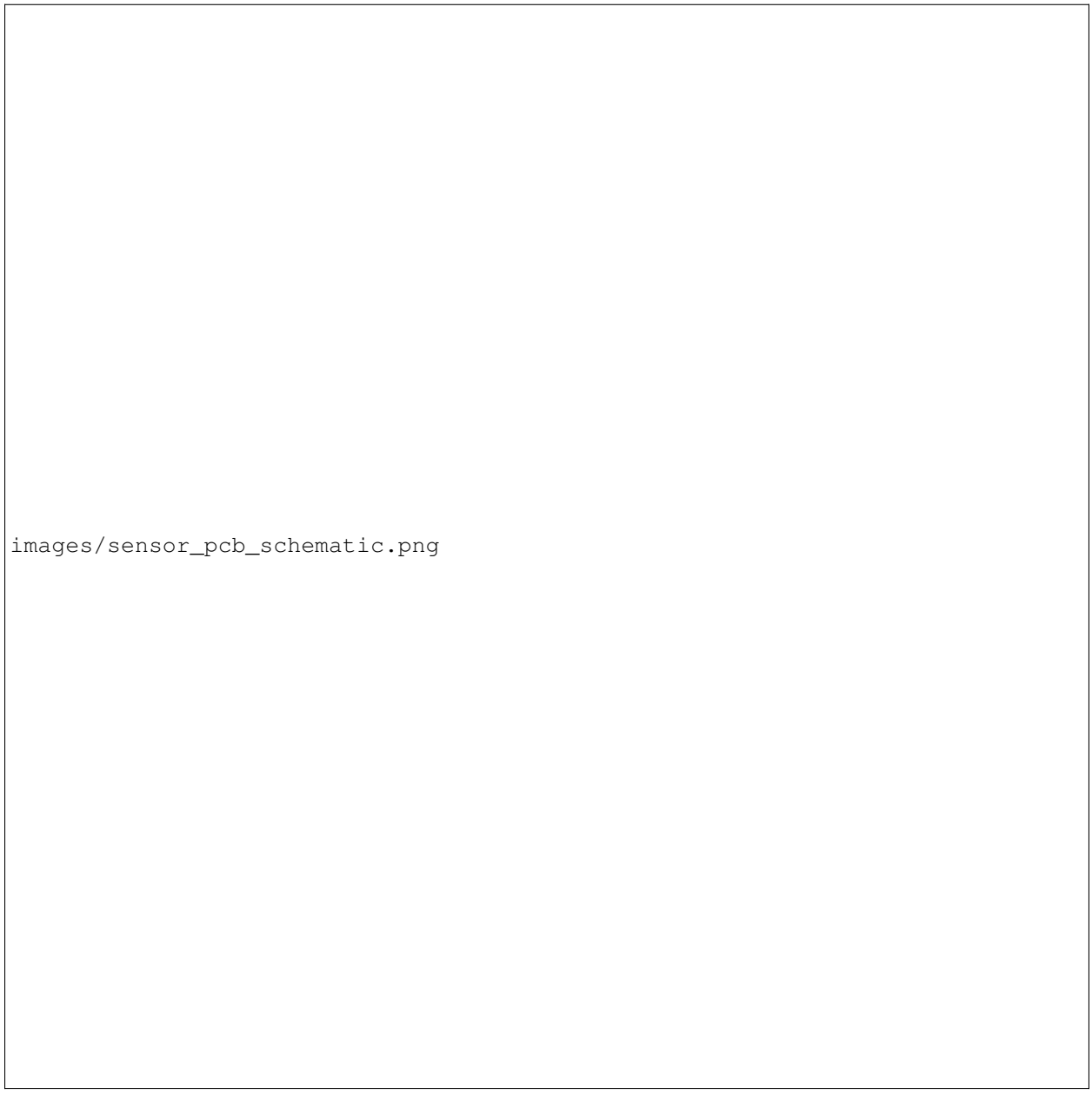

Fig. 2: Electrical schematic for sensor board

  - 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

2.1.1.3 Flow sensor

Document D-lite alternative

2.1.1.4 Pressure sensors

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

2.1.1.5 Oxygen sensor

Explanation of interface circuit and some alts

- Expiratory flow sensor (FS1)

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.)

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)

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)

2.5 Tubing and Adapters

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

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!

CHAPTER

THREE

COMMON MODULE

3.1 values

Parameterization of variables and values

Data

CONTROL	Values to control but not monitor.
DISPLAY_CONTROL	Values that should be displayed in the GUI.
DISPLAY_MONITOR	Values that should be displayed in the GUI.
LIMITS	Values that are dependent on other values:
SENSOR	Values to monitor but not control.

Classes

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

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

Bases: enum.Enum

An enumeration. **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

Definition of a value.

Used by the GUI and control module to set defaults.

Parameters

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

  note:

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

- **decimals** (*int*) – the number of decimals of precision used when displaying the value
- **display** (*bool*) – whether the value should be displayed in the monitor. if `control == True`, automatically set to `False` because all controls have their own numerical displays
- **plot** (*bool*) – whether or not the value is plottable int he 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

Methods

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

Attributes

abs_range
control
control_type
decimals
default
display
group
name
plot
plot_limits
safe_range
sensor

__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*)

Definition of a value.

Used by the GUI and control module to set defaults.

Parameters

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

  note:

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

- **decimals** (*int*) – the number of decimals of precision used when displaying the value
- **display** (*bool*) – whether the value should be displayed in the monitor. if `control == True`, automatically set to `False` because all controls have their own numerical displays
- **plot** (*bool*) – whether or not the value is plottable int he 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

property name

property abs_range

property safe_range

property decimals

property default

property control

property sensor

property display

property control_type

property group

property plot

property plot_limits

to_dict()

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

Values to monitor but not control.

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

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

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

Values to control but not monitor.

Sent to control module to control operation of ventilator.:

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

pvp.common.values.**DISPLAY_MONITOR = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Va**

Values that should be displayed in the GUI. If a value is also a CONTROL it will always have the measured value displayed, these values are those that are sensor values that are uncontrolled and should be displayed.

pvp.common.values.**DISPLAY_CONTROL = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Va**

Values that should be displayed in the GUI. If a value is also a CONTROL it will always have the measured value displayed, these values are those that are sensor values that are uncontrolled and should be displayed.

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

Values that are dependent on other values:

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

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

3.2 loggers

Logging functionality

There are two types of loggers: a standard `logging.Logger` -based logging system for debugging and recording system events, and a `tables` - based *DataLogger* 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.
log_exception(e, tb)	# TODO: Stub exception logger. Prints exception type & traceback
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.

If a logger has already been initialized, return that.

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`

pvp.common.loggers.**log_exception**(*e*, *tb*)
TODO: Stub exception logger. Prints exception type & traceback

Parameters

- **e** – Exception to log
- **tb** – TraceBack associated with Exception e

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.

3.3 message

Classes

ControlSetting(name, value, min_value, ...)	**param name**
ControlValues(control_signal_in, ...)	Class to save control values, analogous to SensorValues.
DerivedValues(timestamp, breath_count, ...)	Class to save derived values, analogous to SensorValues.
Error(errnum, err_str, timestamp)	
SensorValues([timestamp, loop_counter, ...])	**param timestamp** from time.time(). must be passed explicitly or as an entry in `vals`

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

Bases: object

Parameters

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

Attributes

additional_values	Built-in immutable sequence.

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

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

Parameters

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

to_dict()

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

Bases: object

Class to save control values, analogous to SensorValues. 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. 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:

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

Parameters

- **name** –
- **value** –
- **min_value** –
- **max_value** –
- **timestamp** (*float*) – `time.time()`
- **range_severity** (*AlarmSeverity*) – Some control settings have multiple limits for different alarm severities, this attr, when present, specified which is being set.

Methods

__init__(name, value, min_value, max_value, ...)	**param name**

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

Parameters

- **name** –
- **value** –
- **min_value** –
- **max_value** –
- **timestamp** (*float*) – `time.time()`
- **range_severity** (*AlarmSeverity*) – Some control settings have multiple limits for different alarm severities, this attr, when present, specified which is being set.

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

Bases: object

3.4 prefs

System preferences are stored in ~/pvp/prefs.json

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*

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)	
set_pref(key, val)	

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.

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
- LOGGING_MAX_FILES : number of files to split each logger's logs across
- GUI_STATE_FN: Filename of gui control state file, relative to VENT_DIR
- BREATH_PRESSURE_DROP : pressure drop below peep that is detected as an attempt to breath.
- BREATH_DETECTION: (bool) whether the controller allows autonomous breaths (measured pressure is BREATH_PRESSURE_DROP below set PEEP)
- CONTROLLER_MAX_FLOW: If flows above that, hardware cannot be correct.
- CONTROLLER_MAX_PRESSURE: If pressure above that, hardware cannot be correct.
- CONTROLLER_MAX_STUCK_SENSOR: Max amount of time (in s) before considering a sensor stuck

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

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.

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*)

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.

3.5 unit conversion

Functions

cmH2O_to_hPa(pressure)	
hPa_to_cmH2O(pressure)	
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*)

pvp.common.unit_conversion.**hPa_to_cmH2O**(*pressure*)

pvp.common.unit_conversion.**rounded_string**(*value*, *decimals=0*)
create a rounded string of a number that doesnt have trailing .0 when decimals = 0

3.6 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")

3.7 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*)

CHAPTER

FOUR

CONTROLLER MODULE

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.

continues on next page

Table 3 – continued from previous page

_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.
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

CHAPTER

FIVE

COORDINATOR MODULE

5.1 Submodules

5.2 coordinator

Classes

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

Functions

get_coordinator([single_process, sim_mode])	

class 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*

5.3 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**()

5.4 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*)

CHAPTER

SIX

GUI

6.1 Program Diagram

6.2 Design Requirements

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

6.3 UI Notes & Todo

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

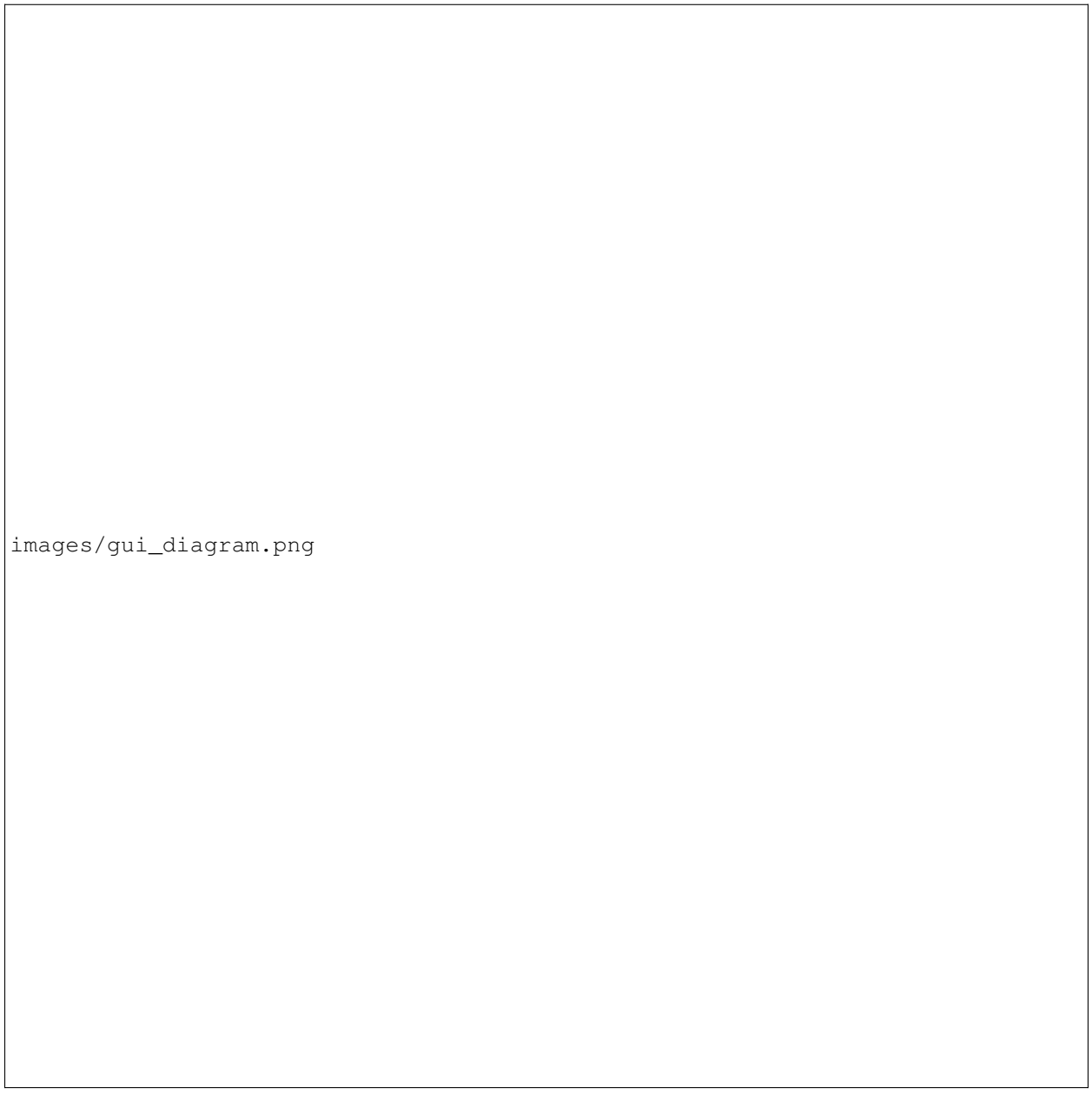

Fig. 1: Schematic diagram of major UI components and signals

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

6.4 Jonny Questions

- Which alarm sounds to use?
- Final final final breakdown on values and ranges plzzz
- RR always has to be present, can only auto calculate InspT, I:E
- make alarm dismissals all latch and snooze.

6.4.1 jonny todo

- use loop_counter to check on controller advancement
- choice between pressure/volume over time and combined P/V plot
- display flow in SLM (liters per minute)
- deque for alarm manager logged alarms
- need confirmation for start button

6.5 GUI Object Documentation

6.5.1 Display

Unified monitor & control widgets

Display sensor values, control control values, turns out it's all the same

Classes

Display(value, update_period, enum_name, ...)	**param value** Value Object to display
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*)

Parameters

- **value** (*Value*) – Value Object to display
- **update_period** (*float*) – time to wait in between updating displayed value
- **enum_name** (*ValueName*) – Value name (not in Value objects)
- **(str** (*style*) – 'left' or 'right'): whether the button should be on the left or right
- **(None, str** (*control_type*) – 'slider', 'record'): whether a slider, a button to record recent values, or `None` control should be used with this object

- **(str** – 'light', 'dark', or a QtStylesheet string): _style for the display
- ****kwargs** (**args,*) – passed to PySide2.QtWidgets.QWidget

Methods

__init__(value, update_period, enum_name, ...)	**param value** Value Object to display
_value_changed(new_value)	"outward-directed" Control value changed by components
init_ui()	
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_units(units)	Set pressure units to display as cmH2O or hPa
timed_update()	
toggle_control(state)	
toggle_record(state)	
update_limits(control)	Update the alarm range and the GUI elements corresponding to it
update_sensor_value(new_value)	
update_set_value(new_value)	inward value setting (set from elsewhere)

Attributes

alarm_state
is_set

self.**name**

self.**units**

self.**abs_range**

self.**safe_range**

self.**decimals**

self.**update_period**

self.**enum_name**

self.**orientation**

self.**control**

self.**_style**

self.**set_value**

self.**sensor_value**

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

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

__init__(*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*)

Parameters

- **value** (*Value*) – Value Object to display
- **update_period** (*float*) – time to wait in between updating displayed value
- **enum_name** (*ValueName*) – Value name (not in Value objects)
- **(str** (*style*) – 'left' or 'right'): whether the button should be on the left or right
- **(None, str** (*control_type*) – 'slider', 'record'): whether a slider, a button to record recent values, or None control should be used with this object
- **(str** – 'light', 'dark', or a QtStylesheet string): _style for the display
- ****kwargs** (**args,*) – passed to PySide2.QtWidgets.QWidget

self.**name**

self.**units**

self.**abs_range**

self.**safe_range**

self.**decimals**

self.**update_period**

self.**enum_name**

self.**orientation**

self.**control**

self.**_style**

self.**set_value**

self.**sensor_value**

init_ui()

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_record(*state*)

_value_changed(*new_value: float*)
"outward-directed" Control value changed by components

Parameters

- **new_value** (*float*) –
- **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*)
inward value setting (set from elsewhere)

update_sensor_value(*new_value: float*)

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

timed_update()

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

Parameters **units** (*'cmH2O', 'hPa'*) –

set_locked(*locked: bool*)

property is_set

property alarm_state

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

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

init_ui()	
update_value(min, max, sensor_value, set_value)	Move the lines! Pass any of the represented values
update_yrange()	

init_ui()

staticMetaObject = <PySide2.QtCore.QMetaObject object>

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

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()

6.5.2 Control Panel

Classes

Control_Panel()	• Start/stop button
HeartBeat([update_interval, timeout_dur])	pvp.gui.widgets.control_panel.**_state**
Lock_Button(*args, **kwargs)	
Power_Button()	
Start_Button(*args, **kwargs)	
StopWatch(update_interval, *args, **kwargs)	**param update_interval** update clock every n seconds

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

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

Methods

_pressure_units_changed(button)	
add_alarm(alarm)	Wraps Alarm_Bar.add_alarm()
clear_alarm(alarm, alarm_type)	Wraps Alarm_Bar.clear_alarm()
init_ui()	

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

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

init_ui()

add_alarm(*alarm:* pvp.alarm.alarm.Alarm)
Wraps Alarm_Bar.add_alarm()

Parameters **alarm** (*Alarm*) – passed to Alarm_Bar

clear_alarm(*alarm:* pvp.alarm.alarm.Alarm = *None*, *alarm_type:* pvp.alarm.AlarmType = *None*)
Wraps Alarm_Bar.clear_alarm()

_pressure_units_changed(*button*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**Start_Button**(**args*, ***kwargs*)

Methods

load_pixmaps()	
set_state(state)	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)

Attributes

states	Built-in mutable sequence.

states = ['OFF', 'ON', 'ALARM']

load_pixmaps()

set_state(*state*)

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.**Lock_Button**(**args*, ***kwargs*)

Methods

load_pixmaps()	
set_state(state)	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)

Attributes

states	Built-in mutable sequence.

states = ['DISABLED', 'UNLOCKED', 'LOCKED']

load_pixmaps()

set_state(*state*)

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=100*, *timeout_dur=5000*)

Methods

__init__([update_interval, timeout_dur])	
	pvp.gui.widgets.control_panel.**_state**

continues on next page

Table 11 – continued from previous page

_heartbeat()	Called every (update_interval) milliseconds to set the check the status of the heartbeat.
beatheart(heartbeat_time)	
init_ui()	
set_indicator([state])	
set_state(state)	
start_timer([update_interval])	**param update_interval** How often (in ms) the timer should be updated.
stop_timer()	you can read the sign ya punk

_state
whether the system is running or not

Type bool

Parameters

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

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

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

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

_state
whether the system is running or not

Type bool

Parameters

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

init_ui()

set_state(*state*)

set_indicator(*state=None*)

start_timer(*update_interval=None*)

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

stop_timer()
you can read the sign ya punk

beatheart(*heartbeat_time*)

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

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**StopWatch**(*update_interval: float = 100*, **args*, ***kwargs*)

Parameters

- **update_interval** (*float*) – update clock every n seconds
- ***args** –
- ****kwargs** –

Methods

__init__(update_interval, *args, **kwargs)	**param update_interval** update clock every n seconds
_update_time()	
init_ui()	
start_timer([update_interval])	**param update_interval** How often (in ms) the timer should be updated.
stop_timer()	you can read the sign ya punk

__init__(*update_interval: float = 100*, **args*, ***kwargs*)

Parameters

- **update_interval** (*float*) – update clock every n seconds
- ***args** –
- ****kwargs** –

init_ui()

start_timer(*update_interval=None*)

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

stop_timer()
you can read the sign ya punk

_update_time()

staticMetaObject = <PySide2.QtCore.QMetaObject object>

class pvp.gui.widgets.control_panel.**Power_Button**

Methods

init_ui()	

init_ui()

staticMetaObject = <PySide2.QtCore.QMetaObject object>

6.5.3 Plot

Classes

Plot(name[, buffer_size, plot_duration, ...])	**param name**
Plot_Container(plot_descriptors, ...)	

Data

PLOT_FREQ	Update frequency of *Plot* s in Hz
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*, *abs_range=None*, *plot_limits: tuple = None*, *color=None*, *units=''*, ***kwargs*)

Parameters

- **name** –
- **buffer_size** –
- **plot_duration** –
- **abs_range** –
- **plot_limits** (*tuple*) – tuple of (ValueName)s for which to make pairs of min and max range lines
- **color** –
- **units** –
- ****kwargs** –

Methods

__init__(name[, buffer_size, plot_duration, ...])	**param name**
_safe_limits_changed(val)	
reset_start_time()	
set_duration(dur)	
set_safe_limits(limits)	
set_units(units)	
update_value(new_value)	new_value (tuple): (timestamp from time.time(), breath_cycle, value)

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

__init__(*name*, *buffer_size=4092*, *plot_duration=10*, *abs_range=None*, *plot_limits: tuple = None*, *color=None*, *units=''*, ***kwargs*)

Parameters

- **name** –
- **buffer_size** –
- **plot_duration** –
- **abs_range** –
- **plot_limits** (*tuple*) – tuple of (ValueName)s for which to make pairs of min and max range lines
- **color** –
- **units** –
- ****kwargs** –

set_duration(*dur*)

update_value(*new_value: tuple*)
new_value (tuple): (timestamp from time.time(), breath_cycle, value)

_safe_limits_changed(*val*)

set_safe_limits(*limits:* pvp.common.message.ControlSetting)

reset_start_time()

set_units(*units*)

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*)

Methods

init_ui()	
reset_start_time()	
set_duration(duration)	
set_plot_mode()	
set_safe_limits(control)	
toggle_plot(state)	
update_value(vals)	

init_ui()

update_value(*vals:* pvp.common.message.SensorValues)

toggle_plot(*state: bool*)

set_safe_limits(*control:* pvp.common.message.ControlSetting)

set_duration(*duration: float*)

staticMetaObject = <PySide2.QtCore.QMetaObject object>

reset_start_time()

set_plot_mode()

6.5.4 Alarm Bar

6.5.5 Components

Classes

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

6.5.6 Dialog

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

6.5.6.1 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.

CHAPTER

SEVEN

PVP.IO PACKAGE

7.1 Subpackages

7.2 Submodules

7.3 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.

7.4 Module contents

CHAPTER

EIGHT

ALARM

8.1 Main Alarm Module

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	

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

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

8.2 Alarm Manager

Classes

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

class pvp.alarm.alarm_manager.**Alarm_Manager**

Attributes

active_alarms	dict() -> new empty dictionary
callbacks	Built-in mutable sequence.
cleared_alarms	Built-in mutable sequence.
dependencies	dict() -> new empty dictionary
depends_callbacks	When we *update_dependencies()*, we send back a *ControlSetting* with the new min/max
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)	
add_dependency_callback(callback)	
check_rule(rule, sensor_values)	
clear_all_alarms()	
deactivate_alarm(alarm)	Mark an alarm's internal active flags and remove from *active_alarms*
dismiss_alarm(alarm_type, duration)	GUI or other object requests an alarm to be dismissed & deactivated
emit_alarm(alarm_type, severity)	Emit alarm (by calling all callbacks with it).
get_alarm_severity(alarm_type)	
load_rule(alarm_rule)	
load_rules()	
register_alarm(alarm)	Add alarm to registry.
register_dependency(condition, dependency, ...)	Add dependency in a Condition object to be updated when values are changed
reset()	reset all conditions, callbacks, and other stateful attributes and clear alarms
update(sensor_values)	
update_dependencies(control_setting)	Update Condition objects that update their value according to some control parameter

active_alarms
{*AlarmType*: *Alarm*}

Type dict

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

Type list

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

Type list

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

Type list

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

Type dict

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

```
_instance = None
active_alarms:  Dict[pvp.alarm.AlarmType, pvp.alarm.alarm.Alarm] = {}
logged_alarms:  List[pvp.alarm.alarm.Alarm] = []
dependencies = {}
pending_clears = []
```

cleared_alarms = []

snoozed_alarms = {}

callbacks = []

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

rules = {}

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

load_rules()

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

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

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

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*

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

Parameters **alarm** –

Returns:

dismiss_alarm(*alarm_type:* 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)

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

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** –
- **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

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

Returns:

add_callback(*callback: Callable*)

add_dependency_callback(*callback: Callable*)

clear_all_alarms()

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

8.3 Alarm

Classes

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

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

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

__init__(alarm_type, severity, start_time, ...)	pvp.alarm.alarm.**id**
deactivate()	

Attributes

alarm_type

continues on next page

Table 9 – continued from previous page

id_counter	itertools.count: used to generate unique IDs for each alarm
severity	

id
unique alarm ID

Type int

Parameters

- **alarm_type** –
- **severity** –
- **start_time** –
- **cause** (ValueName) – The value 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_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*, *persistent: bool = True*, *cause: list = None*, *value=None*, *message=None*)

id
unique alarm ID

Type int

Parameters

- **alarm_type** –
- **severity** –
- **start_time** –
- **cause** (ValueName) – The value 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

property severity

property alarm_type

deactivate()

8.4 Condition

Classes

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

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

Functions

get_alarm_manager()

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

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

Base class for specifying alarm test conditions

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

Methods

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

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

Attributes

manager

manager

alarm manager, used to get status of alarms

Type *pvp.alarm.alarm_manager.Alarm_Manager*

_child

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

Type *Condition*

Parameters

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

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

- ***args** –
- ****kwargs** –

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

Parameters

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

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

- ***args** –
- ****kwargs** –

property manager

check(*sensor_values*)

reset()

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

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

value is greater or lesser than some max/min

Parameters

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

Methods

__init__(value_name, limit, mode, *args, ...)	**param value_name** Which value to check
check(sensor_values)	
reset()	not stateful, do nothing.

Attributes

mode

__init__(*value_name: 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** –

property mode

check(*sensor_values*)

reset()
not stateful, do nothing.

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

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

Attributes

n_cycles

_start_cycle
The breath cycle where the

Type int

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

Type bool

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

property n_cycles

check(*sensor_values*)

reset()
not stateful, do nothing.

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

Parameters

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

- ****kwargs** –

Methods

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

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

Parameters

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

check (*sensor_values*)

reset ()
not stateful, do nothing.

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

Parameters

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

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

- ***args** –
- ****kwargs** –

Methods

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

Attributes

mode

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

Parameters

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

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

- ***args** –
- ****kwargs** –

property mode

check(*sensor_values*)

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

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

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

Methods

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

Attributes

n_cycles

_start_cycle
The breath cycle where the

Type int

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

Type bool

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

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

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

*args: **kwargs:

property n_cycles

check(*sensor_values*)

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

8.5 Alarm Rule

Class to declare alarm rules

Classes

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

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

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

Methods

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

Attributes

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]

CHAPTER

NINE

REQUIREMENTS

CHAPTER

TEN

DATASHEETS & MANUALS

10.1 Manuals

- Hamilton T1 Quick Guide

10.2 Other Reference Material

- Hamilton UI Simulator

CHAPTER

ELEVEN

SPECS

CHAPTER

TWELVE

CHANGELOG

12.1 Version 0.0

12.1.1 v0.0.2 (April xxth, 2020)

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

12.1.2 v0.0.1 (April 12th, 2020)

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

12.1.3 v0.0.0 (April 12th, 2020)

Example of a changelog entry!!!

- We fixed this
- and this
- and this

Warning: but we didn't do this thing

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

CHAPTER

THIRTEEN

BUILDING THE DOCS

A very brief summary...

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

13.1 Local Build

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

Documentation will be generated into `docs`

CHAPTER

FOURTEEN

H1 HEADING 8-)

14.1 h2 Heading

14.1.1 h3 Heading

14.1.1.1 h4 Heading

h5 Heading

h6 Heading

14.2 Horizontal Rules

14.3 Emphasis

This is bold text

This is bold text

This is italic text

This is italic text

14.4 Blockquotes

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

14.5 Lists

Unordered

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

Ordered

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

14.6 Code

Inline `code`

Indented code

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

Block code "fences"

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

Syntax highlighting

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

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

14.7 Links

link text

link with title

14.8 Images

Minion

Like links, Images also have a footnote style syntax

Alt text

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

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V