PVP

Release 0.2.0

jonny saunders et al

Oct 02, 2020
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hardware</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>PVP Hardware</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Software</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>PVP Modules</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Performance</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>ISO Standards Testing</td>
<td>7</td>
</tr>
<tr>
<td>3.2</td>
<td>Breath Detection</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>High Pressure Detection</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Hardware Overview</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Actuators</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Sensors</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Components</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Hardware Design</td>
<td>19</td>
</tr>
<tr>
<td>5.2</td>
<td>Actuator Selection</td>
<td>21</td>
</tr>
<tr>
<td>5.3</td>
<td>Sensor Selection</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>Assembly</td>
<td>23</td>
</tr>
<tr>
<td>6.1</td>
<td>SolidWorks Assembly</td>
<td>23</td>
</tr>
<tr>
<td>6.2</td>
<td>3D Printed Parts</td>
<td>23</td>
</tr>
<tr>
<td>6.3</td>
<td>Enclosure</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Electronics</td>
<td>25</td>
</tr>
<tr>
<td>7.1</td>
<td>Power and I/O</td>
<td>26</td>
</tr>
<tr>
<td>7.2</td>
<td>Sensor PCB</td>
<td>26</td>
</tr>
<tr>
<td>7.3</td>
<td>Actuator PCB</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>Bill of Materials</td>
<td>31</td>
</tr>
<tr>
<td>9</td>
<td>Software Overview</td>
<td>35</td>
</tr>
<tr>
<td>9.1</td>
<td>PVP Modules</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>GUI</td>
<td>37</td>
</tr>
<tr>
<td>10.1</td>
<td>Main GUI Module</td>
<td>37</td>
</tr>
<tr>
<td>10.2</td>
<td>GUI Widgets</td>
<td>45</td>
</tr>
<tr>
<td>10.2.1</td>
<td>Control Panel</td>
<td>45</td>
</tr>
<tr>
<td>10.2.2</td>
<td>Alarm Bar</td>
<td>50</td>
</tr>
<tr>
<td>10.2.3</td>
<td>Display</td>
<td>55</td>
</tr>
</tbody>
</table>
10.2.4 Plot .......................................................... 60
10.2.5 Components .............................................. 64
10.2.6 Dialog ....................................................... 68
10.3 GUI Stylesheets ............................................. 69
10.4 Module Overview ........................................... 70
10.5 Screenshot .................................................... 70

11 Controller ...................................................... 71
11.1 Screenshot .................................................... 72

12 common module .............................................. 83
12.1 Values ......................................................... 83
12.2 Message ....................................................... 88
12.3 Loggers ....................................................... 91
12.4 Prefs .......................................................... 95
12.5 Unit Conversion ............................................. 98
12.6 utils .......................................................... 99
12.7 fashion ......................................................... 99

13 pvp.io package ................................................ 101
13.1 Subpackages ................................................ 101
13.2 Submodules ................................................ 101
13.3 pvp.io.hal module ......................................... 101
13.4 Module contents ........................................... 103

14 Alarm ............................................................ 105
14.1 Alarm System Overview .................................. 105
14.2 Alarm Modules .............................................. 105
14.2.1 Alarm Manager .......................................... 105
14.2.2 Alarm Objects .......................................... 110
14.2.3 Alarm Rule ............................................... 112
14.2.3.1 Example ............................................. 112
14.2.3.2 Module Documentation ............................. 113
14.2.4 Alarm Condition ....................................... 114
14.3 Main Alarm Module ....................................... 122

15 coordinator module ........................................ 125
15.1 Submodules ................................................ 125
15.2 coordinator .................................................. 125
15.3 ipc ............................................................. 128
15.4 process_manager ......................................... 128

16 Requirements .................................................. 131

17 Datasheets & Manuals ..................................... 133
17.1 Manuals ...................................................... 133
17.2 Other Reference Material ................................. 133

18 Specs ............................................................ 135

19 Changelog ....................................................... 137
19.1 Version 0.0 .................................................. 137
19.1.1 v0.0.2 (April xxxth, 2020) ............................. 137
19.1.2 v0.0.1 (April 12th, 2020) ............................. 137
19.1.3 v0.0.0 (April 12th, 2020) ............................. 137
The global COVID-19 pandemic has highlighted the need for a low-cost, rapidly-deployable ventilator, for the current as well as future respiratory virus outbreaks. While safe and robust ventilation technology exists in the commercial sector, the small number of capable suppliers cannot meet the severe demands for ventilators during a pandemic. Moreover, the specialized, proprietary equipment developed by medical device manufacturers is expensive and inaccessible in low-resource areas.

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

*Read on here.* <overview>

The **People’s Ventilator Project (PVP)** is an open-source, low-cost pressure-control ventilator designed for minimal reliance on specialized medical parts to better adapt to supply chain shortages.
The device components were selected to enable a **minimalistic and relatively low-cost ventilator design, to avoid supply chain limitations, and to facilitate rapid and easy assembly**. Most parts in the PVP are not medical-specific devices, and those that are specialized components are readily available and standardized across ventilator platforms, such as standard respiratory circuits and HEPA filters. We provide complete assembly of the PVP, including 3D-printable components, as well as justifications for selecting all actuators and sensors, as guidance to those who cannot source an exact match to components used in the Bill of Materials.

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

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

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

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

2.1 PVP Modules
The completed system was tested with a standard test lung (QuickLung, IngMar Medical, Pittsburgh, PA) that allowed testing combinations of three lung compliance settings (C=5, 20, and 50 mL cm H2O) and three airway resistance settings (R=5, 20, and 50 cm H2O/L/s). The figure above shows pressure control performance for midpoint settings: C=20 mL/cm H2O, R=20 cm H2O/L/s, PIP=30 cm H2O, PEEP=5 cm H2O. PIP is reached within a 300 ms ramp period, then holds for the PIP plateau with minimal fluctuation of airway pressure for the remainder of the inspiratory cycle (blue). One the expiratory valve opens, exhalation begins and expiratory flow is measured (orange) as the airway pressure drops to PEEP and remains there for the rest of the PEEP period.

Some manual adjustment of the pressure waveforms may be warranted depending on the patient, and such adjustment is permitted through a user flow adjustment setting. This flow adjustment setting allows the user to increase the maximum flow rate during the ramp cycle to inflate lungs with higher compliance. The flow setting can be readily changed from the GUI and the control system immediately adapts to the user’s input. An example of this flow adjustment is shown in the figure above for four breath cycles. While all cycles reach PIP, the latter two have a higher mean airway pressure, which may be more desirable under certain conditions than the lower mean airway pressure of the former two.

### 3.1 ISO Standards Testing

In order to characterize the PVP’s control over a wide range of conditions, we followed FDA Emergency Use Authorization guidelines, which specify ISO 80601-2-80-2018 for a battery of pressure controlled ventilator standard tests. We tested the conditions that do not stipulate a leak, and present the results here. For each configuration the following parameters are listed: the test number (from the table below), the compliance (C, mL/cm H2O), linear resistance (R, cm H2O/L/s), respiratory frequency (f, breaths/min), peak inspiratory pressure (PIP, cm H2O), positive end-expiratory pressure (PEEP, cm H2O), and flow adjustment setting.
Fig. 1: Representative pressure control breath cycle waveforms for airway pressure and flow out. Test settings: compliance $C=20 \text{ mL/cm H}_2\text{O}$, airway resistance $R=20 \text{ cm H}_2\text{O/L/s}$, PIP=30 cm H$_2$O, PEEP=5 cm H$_2$O.
3.1. ISO Standards Testing

images/tune_waveform.png
Table 1: Standard test battery from Table 201.105 in ISO 80601-2-80-2018 for pressure controlled ventilators

<table>
<thead>
<tr>
<th>Test number</th>
<th>Intended delivered volume (mL)</th>
<th>Compliance (mL (hPa)^{-1})</th>
<th>Linear resistance (hPa(L/s)^{-1}) +/- 10%</th>
<th>Leakage (mL/min) +/- 10%</th>
<th>Ventilatory frequency (breaths/min)</th>
<th>Inspiratory time (s)</th>
<th>Pressure (hPa)</th>
<th>PEEP (hPa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>50</td>
<td>5</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>50</td>
<td>20</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>500</td>
<td>20</td>
<td>5</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>20</td>
<td>5</td>
<td>5000</td>
<td>20</td>
<td>1</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
<td>50</td>
<td>20</td>
<td>10000</td>
<td>12</td>
<td>1</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
<td>20</td>
<td>50</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>300</td>
<td>10</td>
<td>50</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>300</td>
<td>20</td>
<td>20</td>
<td>3000</td>
<td>20</td>
<td>1</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>300</td>
<td>20</td>
<td>50</td>
<td>6000</td>
<td>12</td>
<td>1</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>200</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>

These tests cover an array of conditions, and more difficult test cases involve a high airway pressure coupled with a low lung compliance (case nos. 8 and 9). Under these conditions, if the inspiratory flow rate during the ramp phase is too high, the high airway resistance will produce a transient spike in airway pressure which can greatly overshoot the PIP value. For this reason, the system uses a low initial flow setting and allows the clinician to increase the flow rate if necessary.

The PVP integrates expiratory flow to monitor the tidal volume, which is not directly set in pressure controlled ventilation, but is an important parameter. Of the test conditions in the ISO standard, four that we tested intended a nominal delivered tidal volume of 500 mL, three intended 300 mL, and one intended 200 mL. For most cases, the estimated tidal volume has a tight spread clustered within 20% of the intended value.

### 3.2 Breath Detection

A patient-initiated breath after exhalation will result in a momentary drop in PEEP. PVP may optionally detect these transient decreases to trigger a new pressure-controlled breath cycle. We tested this functionality by triggering numerous breaths out of phase with the intended inspiratory cycle, using a QuickTrigger (IngMar Medical, Pittsburgh, PA) to momentarily open the test lung during PEEP and simulate this transient drop of pressure.

### 3.3 High Pressure Detection

Above is a demonstration of the PVP’s high airway pressure alarm (HAPA). An airway blockage results in a high airway pressure (above 60 cm H2O) that the system corrects within ~500 ms. Test settings: compliance C=20 mL/cm H2O, airway resistance R=20 cm H2O/L/s, PIP=30 cm H2O, PEEP=5 cm H2O.
images/tidal_volumes.png
3.3. High Pressure Detection
CHAPTER FOUR

HARDWARE OVERVIEW

The PVP components were selected to enable a **minimalistic and relatively low-cost ventilator design**, to avoid supply chain limitations, and to facilitate rapid and easy assembly. Most parts in the PVP are not medical-specific devices, and those that are specialized components are readily available and standardized across ventilator platforms, such as standard respiratory circuits and HEPA filters. We provide complete assembly of the PVP, including 3D-printable components, as well as justifications for selecting all actuators and sensors, as guidance to those who cannot source an exact match to components used in the Bill of Materials.

4.1 Actuators

- Proportional solenoid valve
- Expiratory valve

4.2 Sensors

- Oxygen sensor
- Proximal pressure sensor
- Expiratory flow sensor
CHAPTER FIVE

COMPONENTS

The device components were selected to enable a **minimalistic and relatively low-cost ventilator design**, to avoid supply chain limitations, and to facilitate rapid and easy assembly. Most parts in our system are not medical-specific devices, and those that are specialized components are readily available and standardized across ventilator platforms, such as standard respiratory circuits and HEPA filters. Below, we provide justifications for selecting all actuators and sensors as guidance to those who cannot source an exact match to components used in the PVP.

5.1 Hardware Design

The following is a guided walk through the main hardware components that comprise the respiratory circuit, roughly following the flow of gas from the system inlet, to the patient, then out through the expiratory valve.

**Hospital gas bleder.** At the inlet to the system, we assume the presence of a commercial-off-the-shelf (COTS) gas blender. These devices mix air from U.S. standard medical air and O2 as supplied at the hospital wall at a pressure of around 50 psig. The device outlet fitting may vary, but we assume a male O2 DISS fitting (NIST standard). In field hospitals, compressed air and O2 cylinders may be utilized in conjunction with a gas blender, or a low-cost Venturi-based gas blender. We additionally assume that the oxygen concentration of gas supplied by the blender can be manually adjusted. Users will be able to monitor the oxygen concentration level in real-time on the device GUI.

**Fittings and 3D printed adapters.** Standardized fittings were selected whenever possible to ease part sourcing in the event that engineers replicating the system need to swap out a component, possibly as the result of sourcing constraints within their local geographic area. Many fittings are American national pipe thread (NPT) standard, or conform to the respiratory circuit tubing standards (15mm I.D./22 mm O.D.). To reduce system complexity and sourcing requirements of specialized adapters, a number of connectors, brackets, and manifold are provided as 3D printable parts. All 3D printed components were print-tested on multiple 3D printers, including consumer-level devices produced by MakerBot, FlashForge, and Creality3D.

**Pressure regulator.** The fixed pressure regulator near the inlet of the system functions to step down the pressure supplied to the proportional valve to a safe and consistent set level of 50 psi. It is essential to preventing the over-pressurization of the system in the event of a pressure spike, eases the real-time control task, and ensures that downstream valves are operating within the acceptable range of flow conditions.

**Proportional valve.** The proportional valve is the first of two actuated components in the system. It enables regulation of the gas flow to the patient via the PID control framework, described in a following section. A proportional valve upstream of the respiratory circuit enables the controller to modify the inspiratory time, and does not present wear limitations like pinch-valves and other analogous flow-control devices. The normally closed configuration was selected to prevent over-pressurization of the lungs in the event of system failure.

**Sensors.** The system includes an oxygen sensor for monitoring oxygen concentration of the blended gas supplied to the patient, a pressure sensor located proximally to the patient mouth along the respiratory circuit, and a spirometer, consisting of a plastic housing (D-Lite, GE Healthcare) with an attached differential pressure sensor, to measure flow. Individual sensor selection will be described in more detail in a following section. The oxygen sensor read-out is used to adjust the manual gas blender and to trigger alarm states in the event of deviations from a setpoint. The proximal
location of the primary pressure sensor was selected due to the choice of a pressure-based control strategy, specifically to ensure the most accurate pressure readings with respect to the patient’s lungs. Flow estimates from the single expiratory flow sensor are not directly used in the pressure-based control scheme, but enable the device to trigger appropriate alarm states in order to avoid deviations from the tidal volume of gas leaving the lungs during expiration. The device does not currently monitor gas temperature and humidity due to the use of an HME rather than a heated humidification system.

**Pressure relief.** A critical safety component is the pressure relief valve (alternatively called the “pressure release valve”, or “pressure safety valve”). The proportional valve is controlled to ensure that the pressure of the gas supplied to the patient never rises above a set maximum level. The relief valve acts as a backup safety mechanism and opens if the pressure exceeds a safe level, thereby dumping excess gas to atmosphere. Thus, the relief valve in this system is located between the proportional valve and the patient respiratory circuit. The pressure relief valve we source cracks at 1 psi (approx 70 cm H2O).

**Standard respiratory circuit.** The breathing circuit which connects the patient to the device is a standard respiratory circuit: the flexible, corrugated plastic tubing used in commercial ICU ventilators. Because this system assumes the use of an HME/F to maintain humidity levels of gas supplied to the patient, specialized heated tubing is not required.

**Anti-suffocation check valve.** A standard ventilator check valve (alternatively called a “one-way valve”) is used as a secondary safety component in-line between the proportional valve and the patient respiratory circuit. The check valve is oriented such that air can be pulled into the system in the event of system failure, but that air cannot flow outward through the valve. A standard respiratory circuit check valve is used because it is a low-cost, readily sourced device with low cracking pressure and sufficiently high valve flow coefficient (Cv).

**Bacterial filters.** A medical-grade electrostatic filter is placed on either end of the respiratory circuit. These function as protection against contamination of device internals and surroundings by pathogens and reduces the probability of the patient developing a hospital-acquired infection. The electrostatic filter presents low resistance to flow in the airway.

**HME.** A Heat and Moisture Exchanger (HME) is placed proximal to the patient. This is used to passively humidify and warm air inspired by the patient. HMEs are the standard solution in the absence of a heated humidifier. While we evaluated the use of an HME/F which integrates a bacteriological/viral filter, use of an HME/F increased flow resistance and compromised pressure control.

**Pressure sampling filter.** Proximal airway pressure is sampled at a pressure port near the wye adapter, and measured by a pressure sensor on the sensor PCB. To protect the sensor and internals of the ventilator, an additional 0.2 micron bacterial/viral filter is placed in-line between the proximal airway sampling port and the pressure sensor. This is also a standard approach in many commercial ventilators.

**Expiratory solenoid.** The expiratory solenoid is the second of two actuated components in the system. When this valve is open, air bypasses the lungs, thereby enabling the lungs to de-pressurize upon expiration. When the valve is closed, the lungs may inflate or hold a fixed pressure, according to the control applied to the proportional valve. The expiratory flow control components must be selected to have a sufficiently high valve flow coefficient (Cv) to prevent obstruction upon expiration. This valve is also selected to be normally open, to enable the patient to expire in the event of system failure.

**Manual PEEP valve.** The PEEP valve is a component which maintains the positive end-expiratory pressure (PEEP) of the system above atmospheric pressure to promote gas exchange to the lungs. A typical COTS PEEP valve is a spring-based relief valve which exhausts when pressure within the airway exceeds a fixed limit. This limit is manually adjusted via compression of the spring. Various low-cost alternatives to a COTS mechanical PEEP valve exist, including the use of a simple water column, in the event that PEEP valves become challenging to source. We additionally provide a 3D printable PEEP valve alternative which utilizes a thin membrane, rather than a spring, to maintain PEEP.
5.2 Actuator Selection

When planning actuator selection, it was necessary to consider the placement of the valves within the larger system. Initially, we anticipated sourcing a proportional valve to operate at very low pressures (0-50 cm H2O) and sufficiently high flow (over 120 LPM) of gas within the airway. However, a low-pressure, high-flow regime proportional valve is far more expensive than a proportional valve which operates within high-pressure (~50 psi), high-flow regimes. Thus, we designed the device such that the proportional valve would admit gas within the high-pressure regime and regulate air flow to the patient from the inspiratory airway limb. Conceivably, it is possible to control the air flow to the patient with the proportional valve alone. However, we couple this actuator with a solenoid and PEEP valve to ensure robust control during PIP (peak inspiratory pressure) and PEEP hold, and to minimize the loss of O2-blended gas to the atmosphere, particularly during PIP hold.

**Proportional valve sourcing.** Despite designing the system such that the proportional valve could be sourced for operation within a normal inlet pressure regime (approximately 50 psi), it was necessary to search for a valve with a high enough valve flow coefficient (Cv) to admit sufficient gas to the patient. We sourced an SMC PVQ31-5G-23-01N valve with stainless steel body in the normally-closed configuration. This valve has a port size of 1/8” (Rc) and has previously been used for respiratory applications. Although the manufacturer does not supply Cv estimates, we empirically determined that this valve is able to flow sufficiently for the application.

**Expiratory valve sourcing.** When sourcing the expiratory solenoid, it was necessary to choose a device with a sufficiently high valve flow coefficient (Cv) which could still actuate quickly enough to enable robust control of the gas flow. A reduced Cv in this portion of the circuit would restrict the ability of the patient to exhale. Initially, a number of control valves were sourced for their rapid switching speeds and empirically tested, as Cv estimates are often not provided by valve manufacturers. Ultimately, however, we selected a process valve in lieu of a control valve to ensure the device would flow sufficiently well, and the choice of valve did not present problems when implementing the control strategy. The SMC VXZ250HGB solenoid valve in the normally-open configuration was selected. The valve in particular was sourced partially due to its large port size (3/4” NPT). If an analogous solenoid with rapid switching speed and large Cv cannot be sourced, engineers replicating our device may consider the use of pneumatically actuated valves driven from air routed from a take-off downstream of the pressure regulator.

**Manual PEEP valve sourcing.** The PEEP valve is one of the few medical-specific COTS components in the device. The system configuration assumes the use of any ventilator-specific PEEP valve (Teleflex, CareFusion, etc.) coupled with an adapter to the standard 22 mm respiratory circuit tubing. In anticipation of potential supply chain limitations, as noted previously, we additionally provide the CAD models of a 3D printable PEEP valve.

5.3 Sensor Selection

We selected a minimal set of sensors with analog outputs to keep the system design sufficiently adaptable. If there were a part shortage for a specific pressure sensor, for example, any readily available pressure sensor with an analog output could be substituted into the system following a simple adjustment in calibration in the controller. Our system uses three sensors: an oxygen sensor, an airway pressure sensor, and a flow sensor with availability for a fourth addition, all interfaced with the Raspberry Pi via a 4-channel ADC (Adafruit ADS1115) through an I2C connection.

**Oxygen sensor.** We selected an electrochemical oxygen sensor (Sensironics SS-12A) designed for the range of FiO2 used for standard ventilation and in other medical devices. The cell is self-powered, generating a small DC voltage (13-16 mV) that is linearly proportional to oxygen concentration. The output signal is amplified by an instrumentation amplifier interfacing the sensor with the Raspberry Pi controller (see electronics). This sensor is a wear part with a lifespan of about 6 years under operation at ambient air; therefore under continuous ventilator operation with oxygen-enriched gas, it will need to be replaced more frequently. This part can be replaced with any other medical O2 sensor provided calibration is performed given that these parts are typically sold as raw sensors, with a 3-pin molex interface. Moreover, the sensor we specify is compatible with a range of medical O2 sensors, including the Analytical Industries PSR-11-917-M or the Puritan Bennett 4-072214-00, so we anticipate abundant sourcing options.

**Airway pressure sensor.** We selected a pressure sensor with a few key characteristics in mind: 1) the sensor had
to be compatible with the 5V supply of the Raspberry Pi, 2) the sensor’s input pressure range had conform to the range of pressures possible in our device (up to 70 cm H2O, the pressure relief valve’s cutoff), and 3) the sensor’s response time had to be sufficiently fast. We selected the amplified middle pressure sensor from Amphenol (1 PSI-D-4V), which was readily available, with a measurement range up to 70 cm H2O and an analog output voltage span of 4 V. Moreover, the decision to utilize an analog sensor is convenient for engineers replicating the design, as new analog sensors can be swapped in without extensive code and electronics modifications, as in the case of I2C devices which require modifications to hardware addresses. Other pressure sensors from this Amphenol line can be used as replacements if necessary.

**Spirometer.** Because flow measurement is essential for measuring tidal volume during pressure-controlled ventilation, medical flow sensor availability was extremely limited during the early stages of the 2020 COVID-19 pandemic, and supply is still an issue. For that reason, we looked for inexpensive, more easily sourced spirometers to use in our system. We used the GE D-Lite spirometer, which is a mass-produced part and has been used in hospitals for nearly 30 years. The D-Lite sensor is inserted in-line with the flow of gas on the expiratory limb, and two ports are used to measure the differential pressure drop resulting from flow through a narrow physical restriction. The third pressure-measurement port on the D-Lite is blocked by a male Luer cap, but this could be used as a backup pressure measurement port if desired. An Amphenol 5 INCH-D2-P4V-MINI was selected to measure the differential pressure across the two D-Lite takeoffs. As with the primary (absolute) pressure sensor, this sensor was selected to conform to the voltage range of the Raspberry Pi, operate within a small pressure range, and have a sufficiently fast response time (partially as a function of the analog-to-digital converter). Also, this analog sensor can be readily replaced with a similar analog sensor without substantial code/electronics modifications.
6.1 SolidWorks Assembly

6.2 3D Printed Parts

- Sensor atrium manifold - [.stl]
- Screen module mount (left) - [.stl]
- Screen module mount (right) - [.stl]
- etc...

6.3 Enclosure
The PVP is coordinated by a Raspberry Pi 4 board, which runs the graphical user interface, administers the alarm system, monitors sensor values, and sends actuation commands to the valves. The core electrical system consists of two modular PCB ‘hats’, a sensor PCB and an actuator PCB, that stack onto the Raspberry Pi via 40-pin stackable headers. The modularity of this system enables individual boards to be revised or modified to adapt to component substitutions if required.
7.1 Power and I/O

The main power to the systems is supplied by a DIN rail-mounted 150W 24V supply, which drives the inspiratory valve (4W) and expiratory valves (13W). This voltage is converted to 5V by a switched mode PCB-mounted regulated to power the Raspberry Pi and sensors. This power is transmitted across the PCBs through the stacked headers when required.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meanwell 24 V DC Power Supply</td>
<td>DIN Rail Power Supplies 150W 24V 5A EN55022 Class B</td>
</tr>
<tr>
<td>Raspberry Pi</td>
<td>Raspberry Pi- Model B-1 (1GB RAM)</td>
</tr>
<tr>
<td>USB-C Charger/cable</td>
<td>To power the RPi</td>
</tr>
<tr>
<td>Micro SD Card</td>
<td>SanDisk Ultra 32GB MicroSDHC UHS-I Card with Adapter</td>
</tr>
<tr>
<td>Raspberry Pi Display</td>
<td>Matrix Orbital: TFT Displays &amp; Accessories 7 in HDMI TFT G Series</td>
</tr>
<tr>
<td>HDMI for Display</td>
<td>Display cable: HDMI Cables HDMI Cbl Assbly 1M Micro to STD</td>
</tr>
<tr>
<td>Mini USB for Display</td>
<td>Display cable: USB Cables / IEEE 1394 Cables 3 ft Ext A-B Mini USB Cable</td>
</tr>
<tr>
<td>Screen mount thumb screws</td>
<td>SCREEN_MOUNT_THUMB SCREW: Brass Raised Knurled-Head Thumb Screw, 1/4''-20 Thread Size, 1/2'' Long</td>
</tr>
<tr>
<td>Cable grommet</td>
<td>USER_INTERFACE_CABLE_GROMMET: Buna-N Rubber Grommets, for 1-3/8'' Hole Diameter and 1/16'' Material Thickness, 1'' ID, pack of 10</td>
</tr>
<tr>
<td>Cable P-clip</td>
<td>USER_INTERFACE_CABLE_P-CLIP_0.375_ID_SS: Snug-Fit Vibration-Damping Loop Clamp, 304 Stainless Steel with Silicone Rubber Cushion, 3/8'' ID, pack of 10, 17/64 mounting holes</td>
</tr>
<tr>
<td>Keyboard</td>
<td>Adesso: Mini keyboard with trackball</td>
</tr>
</tbody>
</table>

7.2 Sensor PCB

The sensor board interfaces four analog output sensors with the Raspberry Pi via I2C commands to a 12-bit 4-channel ADC (Adafruit ADS1015).

1. an airway pressure sensor (Amphenol 1 PSI-D-4V-MINI)
2. a differential pressure sensor (Amphenol 5 INCH-D2-P4V-MINI) to report the expiratory flow rate through a D-Lite spirometer
3. an oxygen sensor (Sensiron SS-12A) whose 13 mV differential output signal is amplified 250-fold by an instrumentation amplifier (Texas Instruments INA126)
4. a fourth auxiliary slot for an additional analog output sensor (unused)

A set of additional header pins allows for digital output sensors (such as the Sensiron SFM3300 flow sensor) to be interfaced with the Pi directly via I2C if desired.

• Sensor PCB - [KiCad project .zip]
Fig. 2: Sensor PCB schematic
### Table 2: Sensor PCB bill of materials

<table>
<thead>
<tr>
<th>Ref</th>
<th>Part</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
<td>40-pin stackable RPi header</td>
<td>Connects board to RPi</td>
</tr>
<tr>
<td>J2</td>
<td>4-pin 0.1” header</td>
<td>I2C connector if desired</td>
</tr>
<tr>
<td>J3</td>
<td>2-pin 0.1” header</td>
<td>Connects ALRT pin from ADS1115 to RPi if needed</td>
</tr>
<tr>
<td>J4</td>
<td>3-pin 0.1” header or 3 pin fan extension cable</td>
<td>Connects board to oxygen sensor</td>
</tr>
<tr>
<td>R1</td>
<td>330 Ohm resistor</td>
<td>Sets gain for INA126</td>
</tr>
<tr>
<td>C1</td>
<td>10 uF, 25V</td>
<td>Cap for TL7660</td>
</tr>
<tr>
<td>C2</td>
<td>10 uF, 25V</td>
<td>Cap for TL7660</td>
</tr>
<tr>
<td>U1</td>
<td>TL7660, DIP8</td>
<td>Rail splitter for INA126</td>
</tr>
<tr>
<td>U2</td>
<td>INA126, DIP8</td>
<td>Instrumentation amplifier for oxygen sensor output</td>
</tr>
<tr>
<td>U3</td>
<td>Amphenol 5 INCH-D2-P4V-MINI</td>
<td>Differential pressure sensor (for flow measurement)</td>
</tr>
<tr>
<td>U4</td>
<td>Adafruit ADS1115</td>
<td>4x 12-bit ADC</td>
</tr>
<tr>
<td>U5</td>
<td>Amphenol 1 PSI-D-4V-MINI</td>
<td>Airway pressure sensor</td>
</tr>
<tr>
<td>U6</td>
<td></td>
<td>Auxiliary analog output sensor slot</td>
</tr>
</tbody>
</table>

#### 7.3 Actuator PCB

The purpose of the actuator board is twofold:

1. regulate the 24V power supply to 5V (CUI Inc PDQE15-Q24-S5-D DC-DC converter)
2. interface the Raspberry Pi with the inspiratory and expiratory valves through an array of solenoid drivers (ULN2003A Darlington transistor array)
   - [Actuator PCB - KiCad project .zip](#)

### Table 3: Actuator PCB bill of materials

<table>
<thead>
<tr>
<th>Ref</th>
<th>Part</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2</td>
<td>2-pin screw terminal, 5.08 mm pitch, PCB mount</td>
<td>Connects to 24V supply</td>
</tr>
<tr>
<td>J3</td>
<td>2-pin screw terminal, 5.08 mm pitch, PCB mount</td>
<td>Connects to on/off expiratory valve</td>
</tr>
<tr>
<td>J4</td>
<td>2-pin screw terminal, 5.08 mm pitch, PCB mount</td>
<td>Connects to inspiratory valve, driven by PWM</td>
</tr>
<tr>
<td>J5</td>
<td>40-pin stackable RPi header</td>
<td>Connects board to RPi</td>
</tr>
<tr>
<td>J6</td>
<td>2-pin 0.1” header</td>
<td>Jumper between 5V and Raspberry Pi</td>
</tr>
<tr>
<td>C1</td>
<td>100 uF, 16V</td>
<td>5V rail filter cap</td>
</tr>
<tr>
<td>C2</td>
<td>6.8 uF, 50V</td>
<td>24V rail filter cap</td>
</tr>
<tr>
<td>C3</td>
<td>6.8 uF, 50V</td>
<td>24V rail filter cap</td>
</tr>
<tr>
<td>U1</td>
<td>ULN2003A</td>
<td>Darlington BJT array to drive solenoids</td>
</tr>
<tr>
<td>U2</td>
<td>CUI PDQ15-Q24-S5-D</td>
<td>24-to-5V DC-DC converter</td>
</tr>
</tbody>
</table>
Fig. 3: Actuator PCB schematic
## BILL OF MATERIALS

---

### Bill of materials - [.csv]
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAR filters</td>
<td>DAR Electrostatic filters - Large (10x) (NEED A PRESCRIPTION TO ORDER)</td>
</tr>
<tr>
<td>Adult Respiratory Circuit w Humidifier Limb</td>
<td>Standard resp. circuit</td>
</tr>
<tr>
<td>Gas sampling line</td>
<td>Adapter / filter to connect gas sampling lines to proximal side - TOM</td>
</tr>
<tr>
<td>Luer lock connector</td>
<td>Sensit Luer Lock Connector from Grainger</td>
</tr>
<tr>
<td>Luer lock filter</td>
<td>Luer lock filter 0.2mm from Promepla</td>
</tr>
<tr>
<td>Pressure sensor (for Paw)</td>
<td>1 PSI-D-4V-MINI</td>
</tr>
<tr>
<td>Tubing connectors</td>
<td>Flexible Connector (Silicone Rubber) 22mm I.D. X 22mm I.D. (x6)</td>
</tr>
<tr>
<td>Flow sensor</td>
<td>Disposable GE D-Lite flow sensor</td>
</tr>
<tr>
<td>Luer plug</td>
<td>Plastic Quick-Turn Tube Coupling, Nylon Plastic Caps for Plugs</td>
</tr>
<tr>
<td>Differential pressure sensor for flow</td>
<td>5 INCH-D2-P4V-MINI</td>
</tr>
<tr>
<td>3/4&quot; NPT Connector (male/male)</td>
<td>Thick-Wall Polypropylene Pipe Fitting for Chemicals, Connector with Hex Body, 3/4&quot;</td>
</tr>
<tr>
<td>Exp solenoid valve</td>
<td>SMC VXZ250HGB valve, 20A, NORMALLY OPEN, C37, Port 3/4 (NPT), Orifice</td>
</tr>
<tr>
<td>Exp solenoid brackets</td>
<td>T-Slotted Framing, Silver Flush 90 Degree Angle Bracket for 1” High Rail</td>
</tr>
<tr>
<td>Framing/Structural Components</td>
<td></td>
</tr>
<tr>
<td>80/20 (x4)</td>
<td>T-Slotted Framing, Single 4-Slot Rail, Silver, 1” High x 1” Wide, Solid, 3’ Long - 4x</td>
</tr>
<tr>
<td>80/20 gusset bracket (x18)</td>
<td>Silver Gusset Bracket, 1” Long for 1” High Rail T-Slotted Framing x18</td>
</tr>
<tr>
<td>80/20 rail (x6)</td>
<td>T-Slotted Framing, Single 4-Slot Rail, Silver, 1” High x 1” Wide, Solid, 1’ Long x6</td>
</tr>
<tr>
<td>80/20 button head nuts (packs of 25, x2)</td>
<td>End-feed single nuts for T-Slotted framing: T-Slotted Framing, End-Feed Single Nuts</td>
</tr>
<tr>
<td>Button-head hex screws, pack of 50</td>
<td>BHCS_0.25-20x0.412_SS: 18-8 Stainless Steel Button Head Hex Drive Screw, 1/4&quot;</td>
</tr>
<tr>
<td>Socket head screw, pack of 100</td>
<td>SHCS_M3_X_0.5_6MM_SS: 18-8 Stainless Steel Socket Head Screw, M3 x 0.5 mm</td>
</tr>
<tr>
<td>Button head screws (long), pack of 10</td>
<td>BHCS_10-32_X_2_SS: 18-8 Stainless Steel Button Head Hex Drive Screw, 10-32 T</td>
</tr>
<tr>
<td>Button head screws (short), pack of 25</td>
<td>BHCS_M6_X_1_8MM_SS: Button Head Hex Drive Screw, Passivated 18-8 Stainless Steel</td>
</tr>
<tr>
<td>HDPE sheeting</td>
<td>Moisture-Resistant HDPE Sheet, 12&quot; x 48&quot; x 1/16&quot;</td>
</tr>
<tr>
<td>Lifting handle</td>
<td>LIFTING HANDLE: Oval Pull Handle, Unthreaded Hole, Black Aluminum, 4/9/16</td>
</tr>
<tr>
<td>Screws for lifting handle</td>
<td>SHCS_0.25-20x0.75_Gr8_ASTM_F1136, pack of 50</td>
</tr>
<tr>
<td>Leveling mount (4 feet)</td>
<td>TLA_LEVELING_MOUNT: Light Duty Leveling Mount, 1’ Long 1/4”-20 Threaded</td>
</tr>
<tr>
<td>Nuts for leveling mount</td>
<td>LN_THIN_0.25-20_STAINLESS: 18-8 Stainless Steel Thin Hex Nut, 1/4”-20 Threaded</td>
</tr>
<tr>
<td>PETG for 3d printing</td>
<td>~1 kg spool of PETG</td>
</tr>
<tr>
<td>Electronics</td>
<td></td>
</tr>
<tr>
<td>Standoffs</td>
<td>Standoffs &amp; Spacers 4.5 HEX 12.0mm ALUM</td>
</tr>
<tr>
<td>Meanwell DC Power Supply</td>
<td>DIN Rail Power Supplies 150W 24V 5A EN55022 Class B</td>
</tr>
<tr>
<td>DIN Rail</td>
<td>Aluminum DIN 3 Rail, 10mm Deep, 1m Long</td>
</tr>
<tr>
<td>DIN washers</td>
<td>W_0.25_FLAT_THICK_GR8_YELLOW_ZINC: Zinc Yellow-Chromate Plated Steel</td>
</tr>
<tr>
<td>Sensor board (excluding pressure sensors)</td>
<td>Advanced Circuits 2 layer board</td>
</tr>
<tr>
<td>2-layer PCB</td>
<td>Adafruit Data Conversion IC Development Tools ADS1015 Breakout 12-Bit ADC</td>
</tr>
<tr>
<td>Voltage splitter U1</td>
<td>TL7660, 8-DIP package</td>
</tr>
<tr>
<td>Instrumentation amplifier U2</td>
<td>INA126PA, 8-DIP package</td>
</tr>
<tr>
<td>R3, sensor</td>
<td>Raspberry pi 40 pin stacking header</td>
</tr>
<tr>
<td>4x1 header J1</td>
<td>male pin header 1x04 P2.54mm</td>
</tr>
<tr>
<td>2x1 header J3</td>
<td>male pin header 1x02 P2.54mm</td>
</tr>
<tr>
<td>Oxygen sensor cable (J4)</td>
<td>3 pin fan cable extension, 36&quot;, ribbon style</td>
</tr>
<tr>
<td>330 Ohm resistor R1</td>
<td>330 Ohm resistor, through hole, 1/4 Watt</td>
</tr>
<tr>
<td>Capacitors C1, C2</td>
<td>10 uF, 25V, electrolytic radial</td>
</tr>
<tr>
<td>Actuator board</td>
<td></td>
</tr>
</tbody>
</table>

32 Chapter 8. Bill of Materials
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-layer PCB</td>
<td>Advanced Circuits 2 layer board</td>
</tr>
<tr>
<td>5V DC/DC Converter</td>
<td>490-PDQE15-Q24-S5-D</td>
</tr>
<tr>
<td>Darlington array U1</td>
<td>ULN2003AN IC PWR RELAY 7NPN 1:1 16DIP</td>
</tr>
<tr>
<td>40 pin Pi header J1</td>
<td>Raspberry pi 40 pin stacking header</td>
</tr>
<tr>
<td>Screw terminals J2, J3, J4</td>
<td>5.08 mm pitch 2-pin screw terminal block connector, PCB mount</td>
</tr>
<tr>
<td>2x1 header J5</td>
<td>male pin header 1x02 P2.54mm</td>
</tr>
<tr>
<td>Capacitor C1</td>
<td>100 uF, 16V, electrolytic radial</td>
</tr>
<tr>
<td>Capacitors C2,C3</td>
<td>6.8 uF, 50V, electrolytic radial</td>
</tr>
<tr>
<td>Speaker</td>
<td>Logitech Z50 speaker</td>
</tr>
</tbody>
</table>

**Specialized Tools**

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M16 tap</td>
<td>HSS Tap, Bottoming Chamfer, M16 x 1.0 mm Thread (for mounting O2 Sensor)</td>
</tr>
<tr>
<td>1/4&quot; NPT tap</td>
<td>Uncoated High-Speed Steel Pipe and Conduit Thread Tap High-Speed Steel, Plug Chamfer</td>
</tr>
<tr>
<td>3/4&quot; NPT tap</td>
<td>Uncoated High-Speed Steel Pipe and Conduit Thread Tap: High-Speed Steel, Plug Chamfer</td>
</tr>
<tr>
<td>Ingmar QuickLung Test Bellows</td>
<td>(Optional) variable resistance/compliance test lung for testing only</td>
</tr>
</tbody>
</table>

**Not on the List:**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETG for 3d printing</td>
<td>~1 kg spool of PETG</td>
</tr>
<tr>
<td>Commercial PEEP valves</td>
<td>Alternative: Medium silicone film for printed PEEP valve: <a href="https://www.mcmaster.com/c/86045K58">https://www.mcmaster.com/c/86045K58</a></td>
</tr>
<tr>
<td>Adult Test Lung</td>
<td>Standard adult test lung (Many options here)</td>
</tr>
<tr>
<td>Air compressor for initial testing</td>
<td></td>
</tr>
</tbody>
</table>

---

**Total Cost**

- **Bulk Parts Cost:** $33.00
- **Unit Parts Cost:** $99.00
- **Subtotal (Bulk Parts Cost):** $33.00
- **Subtotal (Unit Parts Cost):** $99.00

---

**M16 tap**

- HSS Tap, Bottoming Chamfer, M16 x 1.0 mm Thread (for mounting O2 Sensor)
- Description: 26015A236
- Cost: $46.63

**1/4" NPT tap**

- Uncoated High-Speed Steel Pipe and Conduit Thread Tap High-Speed Steel, Plug Chamfer
- Description: 2525A173
- Cost: $24.94

**3/4" NPT tap**

- Uncoated High-Speed Steel Pipe and Conduit Thread Tap: High-Speed Steel, Plug Chamfer
- Description: 2525A176
- Cost: $70.16

**Ingmar QuickLung Test Bellows**

- (Optional) variable resistance/compliance test lung for testing only
- Description: https://www.ingmarmed.com/product/quicklung/
- Cost: $1,500.00

---

**Not on the List:**

- PETG for 3d printing: ~1 kg spool of PETG
- Adult Test Lung: Standard adult test lung (Many options here)
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SOFTWARE OVERVIEW

Modular Design
GUI components are programmatically generated, allowing for control of different hardware configurations and ventilation modes.

Alarm Cards
Active alarms are unambiguous, unobtrusive, and individually controllable.

PVP is modularly designed to facilitate future adaptation to new hardware configurations and ventilation modes. APIs were designed for each of the modules to a) make them easily inspected and configured and b) make it clear to future developers how to adapt the system to their needs.

PVP runs as multiple independent processes. The GUI provides an interface to control and monitor ventilation, and the controller process handles the ventilation logic and interfaces with the hardware. Inter-process communication is mediated by a coordinator module via xml-rpc. Several ‘common’ modules facilitate system configuration and constitute the inter-process API. We designed the API around a uni-fied, configurable values module that allows the GUI and controller to be reconfigured while also ensuring system robustness and simplicity.

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

PVP is configured by...
• The Values module parameterizes the different sensor and control values displayed by the GUI and used by the controller

• The Prefs module creates a prefs.json file in ~/pvp that defines user-specific preferences.

PVP is launched like:

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

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

### 9.1 PVP Modules
10.1 Main GUI Module

Classes:

- **Alarm(alarm_type, severity, start_time, ...)** Representation of alarm status and parameters
- **AlarmSeverity(value)** An enumeration.
- **AlarmType(value)** An enumeration.
- **Alarm_Manager()** The Alarm Manager
- **ControlSetting(name, value, min_value, ...)** Message containing ventilation control parameters.
- **CoordinatorBase([sim_mode])**
- **PVP_Gui(coordinator, set_defaults, update_period)** The Main GUI window.
- **SensorValues([timestamp, loop_counter, ...])** Structured class for communicating sensor readings throughout PVP.
- **ValueName(value)** Canonical names of all values used in PVP.

Functions:

- **get_gui_instance()** Retrieve the currently running instance of the GUI
- **init_logger(module_name[, log_level, ...])** Initialize a logger for logging events.
- **launch_gui(coordinator[, set_defaults, ...])** Launch the GUI with its appropriate arguments and doing its special opening routine
- **mono_font()** Module function to return a PySide2.QtGui.QFont to use as the mono font.
- **set_gui_instance(instance)** Store the current instance of the GUI

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

The Main GUI window.

Creates 5 sets of widgets:

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

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

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

Parameters

• coordinator (CoordinatorBase) – Used to communicate with the ControlModuleBase.

• set_defaults (bool) – Whether default Values should be set on initialization (default False) – used for testing and development, values should be set manually for each patient.

• update_period (float) – The global delay between redraws of the GUI (seconds), used by timer.

• screenshot (bool) – Whether alarms should be manually raised to show the different alarm severities, only used for testing and development and should never be used in a live system.

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

Type dict

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

Type dict

plot_box
Container for plots

Type Plot_Box

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

Type pvp.coordinator.coordinator.CoordinatorBase

alarm_manager
Alarm manager instance

Type Alarm_Manager

timer
Timer that calls PVP_Gui.update_gui()

Type PySide2.QtCore.QTimer

running
whether ventilation is currently running

Type bool

locked
whether controls have been locked
Type bool

start_time
Start time as returned by time.time()

Type float

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

Type float

logger
Logger generated by loggers.init_logger()

Attributes:

CONTROL
MONITOR
PLOTS

control_width
controls_set
Check if all controls are set

gui_closing(*args, **kwargs)

monitor_width
plot_width
state_changed(*args, **kwargs)

staticMetaObject

total_width

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

Methods:

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

_set_cycle_control(value_name, new_value)
Compute the computed breath cycle control.

closeEvent(event)
Emit gui_closing and close!

handle_alarm(alarm)
Receive an Alarm from the Alarm_Manager

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

init_ui()
0. Create the UI components for the ventilator screen

init_ui_controls()
4. Create the “controls” column of widgets. Display widgets

init_ui_monitor()
2. Create the left “sensor monitor” column of widgets. Display widgets

continues on next page
Table 4 – continued from previous page

**init_ui_plots()**
3. Create the Plot_CONTAINER

**init_ui_signals()**
5. Connect Qt signals and slots between widgets

**init_ui_status_bar()**
1. Create the widgets.Control_Panel and widgets.Alarm_Bar

**limits_updated(control)**
Receive updated alarm limits from the Alarm_Manager

**load_state(state)**
Load GUI state and reconstitute

**save_state()**
Try to save GUI state to pref['VENT_DIR'] + pref['GUI_STATE_FN']

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

**set_control(control_object)**
Set a control in the alarm manager, coordinator, and gui

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

**set_value(new_value[, value_name])**
Set a control value using a value and its name.

**start()**
Click the start_button

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

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

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

**update_gui([vals])**

**update_state(state_type, key, val)**
Update the GUI state and save it to disk with Vent_Gui.save_state()

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

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

**MONITOR = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value object>), (<ValueName.PEEP: 3>, <pvp.common.values.Value object>), (<ValueName.PIP_TIME: 2>, <pvp.common.values.Value object>), (<ValueName.FLOWOUT: 11>, <pvp.common.values.Value object>), (<ValueName.FIO2: 8>, <pvp.common.values.Value object>)]))**
Values to create Display widgets for in the Sensor Monitor column. See values.DISPLAY_MONITOR

**CONTROL = OrderedDict([(<ValueName.PIP: 1>, <pvp.common.values.Value object>), (<ValueName.PEEP: 3>, <pvp.common.values.Value object>), (<ValueName.PIP_TIME: 2>, <pvp.common.values.Value object>), (<ValueName.PRESSURE: 10>, <pvp.common.values.Value object>), (<ValueName.FLOWOUT: 11>, <pvp.common.values.Value object>), (<ValueName.FIO2: 8>, <pvp.common.values.Value object>), (<ValueName.PWT: 9>, <pvp.common.values.Value object>), (<ValueName.PIP_TIME: 2>, <pvp.common.values.Value object>), (<ValueName.PRESSURE: 10>, <pvp.common.values.Value object>), (<ValueName.FLOWOUT: 11>, <pvp.common.values.Value object>), (<ValueName.FIO2: 8>, <pvp.common.values.Value object>)]))**
Values to create Display widgets for in the Control column. See values.CONTROL

**PLOTS = OrderedDict([(<ValueName.PRESSURE: 10>, <pvp.common.values.Value object>), (<ValueName.FLOWOUT: 11>, <pvp.common.values.Value object>), (<ValueName.FIO2: 8>, <pvp.common.values.Value object>), (<ValueName.FLOWIN: 12>, <pvp.common.values.Value object>), (<ValueName.PIP_TIME: 2>, <pvp.common.values.Value object>),..<ValueName.FLOWIN: 12>, <pvp.common.values.Value object>)]))**
Values to create Plot widgets for. See values.PLOTS

**monitor_width = 3**
Relative width of the sensor monitor column

**plot_width = 4**
Relative width of the plot column

\[ \text{control_width} = 3 \]

Relative width of the control column

\[ \text{total_width} = 10 \]

computed from \( \text{monitor_width} + \text{plot_width} + \text{control_width} \)

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

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

**init_ui()**

0. Create the UI components for the ventilator screen

Call, in order:

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

Create and set sizes of major layouts

**init_ui_status_bar()**

1. Create the `widgets.Control_Panel` and `widgets.Alarm_Bar` and add them to the main layout

**init_ui_monitor()**

2. Create the left “sensor monitor” column of `widgets.Display` widgets

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

**init_ui_plots()**

3. Create the `Plot_Container`

**init_ui_controls()**

4. Create the “controls” column of `widgets.Display` widgets

**init_ui_signals()**

5. Connect Qt signals and slots between widgets

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

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

Set a control value using a value and its name.

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

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

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

```python
set_control(control_object: pvp.common.message.ControlSetting)
```
Set a control in the alarm manager, coordinator, and gui
Also update our state with `update_state()`

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

```python
handle_alarm(alarm: pvp.alarm.alarm.Alarm)
```
Receive an Alarm from the Alarm_Manager
Alarms are both raised and cleared with this method – there is no separate “clear_alarm” method because an alarm of `AlarmSeverity` of OFF is cleared.
Give the alarm to the Alarm_Bar and update the alarm Display.alarm_state of all widgets listed as Alarm.cause

```python
Parameters alarm (Alarm) – The alarm to raise (or clear)
```

```python
limits_updated(control: pvp.common.message.ControlSetting)
```
Receive updated alarm limits from the Alarm_Manager
When a value is set that has an Alarm_Rule that Alarm_Rule.depends on it, the alarm thresholds will be updated and handled here.
Eg. the high-pressure alarm is set to be 15% above PIP. When PIP is changed, this method will receive a message.ControlSetting that tells us that alarm threshold has changed.
Update the Display and Plot widgets.
If we are setting a new HAPA limit, that is also sent to the controller as it needs to respond as quickly as possible to high-pressure events.

```python
Parameters control (message.ControlSetting) – A ControlSetting with its max_value or
:param min_value set:
```

```python
start()
```
Click the start_button

```python
toggle_start(state: bool)
```
Start or stop ventilation.
Typically called by the PVP_Gui.control_panel.start_button.
Raises a dialogue to confirm ventilation start or stop
Starts or stops the controller via the coordinator
If starting, locks controls.

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

```python
closeEvent(event)
```
Emit gui_closing and close!
Kill the coordinator with CoordinatorBase.kill()
**toggle_lock** (*state*)

Toggle the lock state of the controls

Typically called by `PVP_Gui.control_panel.lock_button`

**Parameters**

- **state** –

**Returns:**

**update_state** (*state_type: str, key: str, val: Union[str, float, int]*)

Update the GUI state and save it to disk with `Vent_Gui.save_state()`

Currently, just saves the state of control settings.

**Parameters**

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

**Returns:**

**save_state()**

Try to save GUI state to `prefs['VENT_DIR'] + prefs['GUI_STATE_FN']`

**load_state** (*state: Union[str, dict]*)

Load GUI state and reconstitute

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

**Parameters**

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

**staticMetaObject** = `<PySide2.QtCore.QMetaObject object>`

**toggle_cycle_widget** (*button*)

Set which breath cycle control is automatically calculated

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

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

**Parameters**

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

**set_pressure_units** (*units*)

Select whether pressure units are displayed as “cmH2O” or “hPa”

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

**Parameters**

- **units** (*str*) – one of “cmH2O” or “hPa”

**set_breath_detection** (*breath_detection: bool*)

Connected to `breath_detection_button` - toggles autonomous breath detection in the controller

**Parameters**

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

**_set_cycle_control** (*value_name: str, new_value: float*)

Compute the computed breath cycle control.

We only actually have BPM and INSPl as controls, so if we're using I:E ratio we have to compute one or the other.
Computes the value and calls `set_control()` with the appropriate values:

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

**property controls_set**
Check if all controls are set

---

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

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

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

---

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

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

To launch the gui, one must:

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

**Parameters**

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

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

**Return type** (tuple)
10.2 GUI Widgets

10.2.1 Control Panel

The Control Panel starts and stops ventilation and controls runtime options.

Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm</td>
<td>Representation of alarm status and parameters</td>
</tr>
<tr>
<td>AlarmType</td>
<td>An enumeration.</td>
</tr>
<tr>
<td>Control_Panel</td>
<td>The control panel starts and stops ventilation and controls runtime settings</td>
</tr>
<tr>
<td>HeartBeat</td>
<td>Track state of connection with Controller</td>
</tr>
<tr>
<td>Lock_Button</td>
<td>Button to lock and unlock controls</td>
</tr>
<tr>
<td>OnOffButton</td>
<td>Simple extension of toggle button with styling for clearer ‘ON’ vs ‘OFF’</td>
</tr>
<tr>
<td>QHLine</td>
<td>with respect to <a href="https://stackoverflow.com/a/51057516">https://stackoverflow.com/a/51057516</a></td>
</tr>
<tr>
<td>Start_Button</td>
<td>Button to start and stop Ventilation, created by Control_Panel</td>
</tr>
<tr>
<td>StopWatch</td>
<td>Simple widget to display ventilation time!</td>
</tr>
<tr>
<td>odict</td>
<td>alias of collections.OrderedDict</td>
</tr>
</tbody>
</table>

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_gui_instance()</td>
<td>Retrieve the currently running instance of the GUI</td>
</tr>
<tr>
<td>mono_font()</td>
<td>module function to return a PySide2.QtGui.QFont to use as the mono font.</td>
</tr>
</tbody>
</table>

```python
class pvp.gui.widgets.control_panel.Control_Panel

    The control panel starts and stops ventilation and controls runtime settings

    It creates:

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

    Args:

    **start_button**
    Button to start and stop ventilation
    Type Start_Button

    **lock_button**
    Button used to lock controls
    Type Lock_Button

    **heartbeat**
    Widget to keep track of communication with controller
    Type HeartBeat
```
### runtime

Widget used to display time since start of ventilation

**Type** *StopWatch*

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_pressure_units_changed(button)</code></td>
<td>Emit the str of the current pressure units</td>
</tr>
<tr>
<td><code>init_ui()</code></td>
<td>Initialize all graphical elements and buttons!</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cycle_autoset_changed(*args,**kwargs)</code></td>
<td>Signal emitted when a different breath cycle control value is set to be autocalculated</td>
</tr>
<tr>
<td><code>pressure_units_changed(*args,**kwargs)</code></td>
<td>Signal emitted when pressure units have been changed. Contains str of current pressure units</td>
</tr>
<tr>
<td><code>staticMetaObject</code></td>
<td></td>
</tr>
</tbody>
</table>

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

Signal emitted when pressure units have been changed.

### init_ui()

Initialize all graphical elements and buttons!

```python
init_ui()
```

### _pressure_units_changed(button)

Emit the str of the current pressure units

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>button</td>
<td>PySide2.QtWidgets.QPushButton</td>
<td>Button that was clicked</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>staticMetaObject</code></td>
<td></td>
</tr>
</tbody>
</table>

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

Button to start and stop Ventilation, created by *Control_Panel*

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load_pixmaps()</code></td>
<td>Load pixmaps to <em>Start_Button.pixmaps</em></td>
</tr>
<tr>
<td><code>set_state(state)</code></td>
<td>Set state of button</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>states</code></td>
<td>Possible states of Start_Button</td>
</tr>
</tbody>
</table>

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

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load_pixmaps()</code></td>
<td>Load pixmaps to <em>Start_Button.pixmaps</em></td>
</tr>
</tbody>
</table>
**set_state** *(state)*
Set state of button

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

**Parameters**

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

**staticMetaObject** = `<PySide2.QtCore.QMetaObject object>`

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

Created by *Control_Panel*

**pixmaps**
Dictionary containing pixmaps used to draw locked/unlocked state

**Type**
*dict*

**Methods:**

<table>
<thead>
<tr>
<th>load_pixmaps()</th>
<th>Load pixmaps used to display lock state to <em>Lock_Button.pixmaps</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>set_state(state)</td>
<td>Set lock state of button</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>states</th>
<th>staticMetaObject</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

set_state(state)
Set lock state of button

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

**Parameters**

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

**staticMetaObject** = `<PySide2.QtCore.QMetaObject object>`

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

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

**Parameters**

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

*_state_ whether the system is running or not

10.2. GUI Widgets
Type bool

_last_heartbeat
Timestamp of last contact with controller

Type float

start_time
Time that ventilation was started

Type float

timer
Timer that checks for last contact

Type PySide2.QtCore.QTimer

update_interval
How often to do the heartbeat, in ms

Type int

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

Type int

Methods:

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

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

init_ui() Initialize labels and status indicator

set_indicator([state]) Set visual indicator

set_state(state) Set running state

start_timer([update_interval]) Start HeartBeat.timer to check for contact with controller

stop_timer() Stop timer and clear text

Attributes:

heartbeat(*args, **kwargs)

staticMetaObject

timeout(*args, **kwargs)

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

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

init_ui()
Initialize labels and status indicator

set_state(state)
Set running state

if just starting reset HeartBeat._last_heartbeat
Parameters **state** (*bool*) – Whether we are starting (True) or stopping (False)

**set_indicator** (*state=None*)
Set visual indicator

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

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

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

**stop_timer** ()
Stop timer and clear text

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

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

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

```
staticMetaObject = <PySide2.QtCore.QMetaObject object>
class pvp.gui.widgets.control_panel.StopWatch(*update_interval: float = 100, *args, **kwargs*)
    Simple widget to display ventilation time!

    Parameters

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

    Methods:

    __init__([update_interval]) Simple widget to display ventilation time!
    __update_time()
    _init_ui()
    start_timer([update_interval])
    stop_timer()

    Attributes:

    staticMetaObject

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

    Parameters

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

10.2. GUI Widgets 49
• **kwargs – passed to PySide2.QtWidgets.QLabel

```
staticMetaObject = <PySide2.QtCore.QMetaObject object>
init_ui()
start_timer(update_interval=None)
```

**Parameters**

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

```
stop_timer()
_update_time()
```

### 10.2.2 Alarm Bar

The **Alarm Bar** displays **Alarm** status with **Alarm Card** widgets and plays alarm sounds with the **Alarm Sound Player**.

**Classes:**

- *Alarm(alarm_type, severity, start_time, ...)*: Representation of alarm status and parameters
- *AlarmSeverity(value)*: An enumeration.
- *AlarmType(value)*: An enumeration.
- *Alarm_Bar()*: Holds and manages a collection of **Alarm_Card**s and communicates requests for dismissal to the **Alarm_Manager**.
- *Alarm_Card(alarm)*: Representation of an alarm raised by **Alarm_Manager** in GUI.
- *Alarm_Manager()*: The Alarm Manager
- *Alarm_Sound_Player(increment_delay, *args, ...)*: Plays alarm sounds to reflect current alarm severity and active duration with PySide2.QtMultimedia.QSoundEffect objects

**Functions:**

- *init_logger(module_name[, log_level, ...])*: Initialize a logger for logging events.
- *mono_font()*: module function to return a PySide2.QtGui.QFont to use as the mono font.

```python
class pvp.gui.widgets.alarm_bar.Alarm_Bar
    Holds and manages a collection of **Alarm_Card**s and communicates requests for dismissal to the **Alarm_Manager**.

    The alarm bar also manages the **Alarm_Sound_Player**

    **alarms**
    A list of active alarms
    
    Type: typing.List[Alarm]

    **alarm_cards**
    A list of active alarm cards
    
    Type: typing.List[Alarm_Card]

    **sound_player**
```
Class that plays alarm sounds!

**Type** `Alarm_Sound_Player`

**icons**
Dictionary of pixmaps with icons for different alarm levels

**Type** `dict`

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_alarm(alarm)</code></td>
<td>Add an alarm created by the <code>Alarm_Manager</code> to the bar.</td>
</tr>
<tr>
<td><code>clear_alarm(alarm, alarm_type)</code></td>
<td>Remove an alarm card, update appearance and sound player to reflect current max severity</td>
</tr>
<tr>
<td><code>init_ui()</code></td>
<td>Initialize the UI</td>
</tr>
<tr>
<td><code>make_icons()</code></td>
<td>Create pixmaps from standard icons to display for different alarm types</td>
</tr>
<tr>
<td><code>set_icon(state)</code></td>
<td>Change the icon and bar appearance to reflect the alarm severity</td>
</tr>
<tr>
<td><code>update_icon()</code></td>
<td>Call <code>set_icon(state)</code> with highest severity in <code>Alarm_Bar.alarms</code></td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alarm_level</code></td>
<td>Current maximum <code>AlarmSeverity</code></td>
</tr>
<tr>
<td><code>staticMetaObject</code></td>
<td></td>
</tr>
</tbody>
</table>

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

Store in `Alarm_Bar.icons`

**init_ui()**
Initialize the UI

- Create layout
- Set icon
- Create mute button

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

If an alarm already exists with that same `AlarmType`, `Alarm_Bar.clear_alarm()`
Insert new alarm in order the prioritizes alarm severity with highest severity on right
Set alarm sound and begin playing if not already.

**Parameters**

- `alarm (Alarm)` – Alarm to be added

**clear_alarm(alarm: pvp.alarm.Alarm = None, alarm_type: pvp.alarm.AlarmType = None)**
Remove an alarm card, update appearance and sound player to reflect current max severity
Must pass one of either `alarm` or `alarm_type`

**Parameters**

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

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

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

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

**property alarm_level**
Current maximum `AlarmSeverity`

**Returns** *
- `AlarmSeverity`

**staticMetaObject** = `<PySide2.QtCore.QMetaObject object>`

Representation of an alarm raised by `Alarm_Manager` in GUI.

If allowed by alarm (by latch setting), allows user to dismiss/silence alarm.
Otherwise request to dismiss is logged by `Alarm_Manager` and the card is dismissed when the conditions that generated the alarm are no longer met.

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

**alarm**
The represented alarm

**Type**
- `Alarm`

**severity**
The severity of the represented alarm

**Type**
- `AlarmSeverity`

**close_button**
Button that requests an alarm be dismissed

**Type** `PySide2.QtWidgets.QPushButton`

**Methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_dismiss()</code></td>
<td>Gets the <code>Alarm_Manager</code> instance and calls <code>Alarm_Manager.dismiss_alarm()</code></td>
</tr>
<tr>
<td><code>init_ui()</code></td>
<td>Initialize graphical elements</td>
</tr>
</tbody>
</table>

**Attributes:**

**staticMetaObject**

```python
init_ui()
```
Initialize graphical elements

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

**Returns:**
_dismiss()
    Gets the Alarm_Manager instance and calls Alarm_Manager.dismiss_alarm()
    Also change appearance of close button to reflect requested dismissal

staticMetaObject = <PySide2.QtCore.QMetaObject object>
class pvp.gui.widgets.alarm_bar.Alarm_Sound_Player(increment_delay: int = 10000, *args, **kwargs)
    Plays alarm sounds to reflect current alarm severity and active duration with PySide2.QtMultimedia.QSoundEffect objects
    Alarm sounds indicate severity with the number and pitch of tones in a repeating tone cluster (eg. low severity sounds have a single repeating tone, but high-severity alarms have three repeating tones)
    They indicate active duration by incrementally removing a low-pass filter and making tones have a sharper attack and decay.
    When an alarm of any severity is started the <severity_0.wav file begins playing, and a timer is started to call Alarm_Sound_Player.increment_level()

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

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

files
    list of sound file paths
    Type list

increment_delay
    Time between calling Alarm_Sound_Player.increment_level() in ms
    Type int

playing
    Whether or not a sound is playing
    Type bool

_increment_timer
    Timer that increments alarm sound level
    Type PySide2.QtCore.QTimer

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

Methods:
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>increment_level()</td>
<td>If current level is below the maximum level, increment with <code>Alarm_Sound_Player.set_sound()</code></td>
</tr>
<tr>
<td>init_audio()</td>
<td>Load audio files in <code>pvp/external/audio</code> and add to <code>Alarm_Sound_Player.idx</code></td>
</tr>
<tr>
<td>play()</td>
<td>Start sound playback</td>
</tr>
<tr>
<td>set_mute(mute)</td>
<td>Set mute state</td>
</tr>
<tr>
<td>set_sound([severity, level])</td>
<td>Set sound to be played</td>
</tr>
<tr>
<td>stop()</td>
<td>Stop sound playback</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity_map</td>
<td>mapping between string representations of severities used by filenames and <code>AlarmSeverity</code></td>
</tr>
<tr>
<td>staticMetaObject</td>
<td></td>
</tr>
</tbody>
</table>

```
severity_map = {'high': <AlarmSeverity.HIGH: 3>, 'low': <AlarmSeverity.LOW: 1>, 'medium': <AlarmSeverity.MEDIUM: 2>}
```

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

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

play()
Start sound playback
Play sound listed as `Alarm_Sound_Player._current_sound`
```

```
set_sound (severity: pvp.alarm.AlarmSeverity = None, level: int = None)
Set sound to be played
At least an `AlarmSeverity` must be provided.
```

Parameters

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

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

```
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```

```
set_mute (mute: bool)
Set mute state
```

Parameters

- `mute` (`bool`) – if True, mute. if False, unmute.
10.2.3 Display

Unified monitor & control widget
Displays sensor values, and can optionally control system settings.

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

Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlarmSeverity(value)</td>
<td>An enumeration.</td>
</tr>
<tr>
<td>ControlSetting(name, value, min_value,...)</td>
<td>Message containing ventilation control parameters.</td>
</tr>
<tr>
<td>Display(value, update_period, enum_name,...)</td>
<td>Unified widget for display of sensor values and control of ventilation parameters</td>
</tr>
<tr>
<td>DoubleSlider([decimals])</td>
<td>Slider capable of representing floats</td>
</tr>
<tr>
<td>EditableLabel([parent])</td>
<td>Editable label <a href="https://gist.github.com/mfessenden/baa2b87b8addb0b60c54a11c1da48046">https://gist.github.com/mfessenden/baa2b87b8addb0b60c54a11c1da48046</a></td>
</tr>
<tr>
<td>Limits_Plot(style, *args, **kwargs)</td>
<td>Widget to display current value in a bar graph along with alarm limits</td>
</tr>
<tr>
<td>QVLine([parent, color])</td>
<td></td>
</tr>
<tr>
<td>Value(name, units, abs_range, safe_range,...)</td>
<td>Class to parameterize how a value is used in PVP.</td>
</tr>
<tr>
<td>ValueName(value)</td>
<td>Canonical names of all values used in PVP.</td>
</tr>
</tbody>
</table>

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_logger(module_name[, log_level,...])</td>
<td>Initialize a logger for logging events.</td>
</tr>
<tr>
<td>mono_font()</td>
<td>module function to return a PySide2.QtGui.QFont to use as the mono font.</td>
</tr>
<tr>
<td>pop_dialog(message[, sub_message, modality,...])</td>
<td>Creates a dialog box to display a message.</td>
</tr>
</tbody>
</table>

```python
class pvp.gui.widgets.display.Display(value: pvp.common.values.Value, update_period: float = 0.5, enum_name: pvp.common.values.ValueName = None, button_orientation: str = 'left', control_type: Union[None, str] = None, style: str = 'dark', *args, **kwargs)
```

Unified widget for display of sensor values and control of ventilation parameters
Displayed values are updated according to Display.timed_update()

Parameters

- **value** (*Value*) – Value object to represent
- **update_period** (*float*) – Amount of time between updates of the textual display of values
- **enum_name** (*ValueName*) – Value name of object to represent
- **button_orientation** (*'left', 'right'*) – whether the controls are drawn on the 'left' or 'right'
- **control_type** (*None, 'slider', 'record'*) – type of control - either None (no control), slider (a slider can be opened to set a value), or record where recent sensor values are averaged and used to set the control value. Both types of control allow values to be input from the keyboard by clicking on the editable label
• **kwargs(*args,*) – passed on to PySide2.QtWidgets.QWidget

self.name
Unpacked from value

self.units
Unpacked from value

self.abs_range
Unpacked from value

self.safe_range
Unpacked from value

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

self.decimals
Unpacked from value

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

    Type float

self.enum_name
Value name of object to represent

    Type ValueName

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

    Type 'left', 'right'

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

    Type None, 'slider', 'record'

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

    Type 'light', 'dark'

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

    Type float

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

    Type float

Methods:
_value_changed(new_value)  “outward-directed” method to emit new changed control value when changed by this widget

init_ui() UI is initialized in several stages

init_ui_labels() Basically two methods.

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

init_ui_record()

init_ui_signals()

init_ui_slider()

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

redraw()

set_locked(locked) Set locked status of control

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

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

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

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

update_limits(control) Update the alarm range and the GUI elements corresponding to it

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

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

Attributes:

alarm_state Current visual display of alarm severity

is_set Check if value has been set for this control.

staticMetaObject

value_changed(*args, **kwargs) Signal emitted when controlled value of display object has changed.

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

Signal emitted when controlled value of display object has changed.

  Contains new value (float)

init_ui() UI is initialized in several stages

• 0: this method, get stylesheets based on self._style and call remaining initialization methods

• 1: Display.init_ui_labels() - create generic labels shared by all display objects

• 2: Display.init_ui_toggle_button() - create the toggle or record button used by controls

• 3: Display.init_ui_limits() - create a plot that displays the sensor value graphically relative to the alarm limits

• 4: Display.init_ui_slider() or Display.ini_ui_record() - depending on what type of control this is

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

- `init_ui_labels()`
- `init_ui_toggle_button()`
- `init_ui_limits()`
  - Create widgets to display sensed value alongside set value
- `init_ui_slider()`
- `init_ui_record()`
- `init_ui_layout()`
  - Basically two methods... lay objects out depending on whether we’re oriented with our button to the left or right
- `init_ui_signals()`

`toggle_control(state)`
- Toggle the appearance of the slider, if a slider
  - Parameters `state (bool)` – Whether to show or hide the slider

`toggle_record(state)`
- Toggle the recording state, if a recording control
  - Parameters `state (bool)` – Whether recording should be started or stopped. when started, start storing new sensor values in a list. when stopped, average them and emit new value.

`_value_changed(new_value: float)`
- “outward-directed” method to emit new changed control value when changed by this widget
  - Pop a confirmation dialog if values are set outside the safe range.
  - Parameters
    - `new_value (float)` – new value!
    - `emit (bool)` – whether to emit the `value_changed` signal (default True) – in the case that our value is being changed by someone other than us

`update_set_value(new_value: float)`
- Update to reflect new control value set from elsewhere (inwardly directed setter)
  - Parameters `new_value (float)` – new value to set!

`update_sensor_value(new_value: float)`
- Receive new sensor value and update display widgets
  - Parameters `new_value (float)` – new sensor value!

`update_limits(control: pvp.common.message.ControlSetting)`
- Update the alarm range and the GUI elements corresponding to it
  - Parameters `control (ControlSetting)` – control setting with min_value or max_value

`redraw()`
- Redraw all graphical elements to ensure internal model matches view
  - Typically used when changing units

`timed_update()`
- Refresh textual sensor values only periodically to prevent them from being totally unreadable from being changed too fast.
**set_units** *(units: str)*
Set pressure units to display as cmH2O or hPa.

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

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

**Note:** currently unit conversion is only supported for Pressure.

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

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

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

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

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

**Returns**
- whether we have an `Display.set_value`

**Return type**
- bool

**property alarm_state**
Current visual display of alarm severity

Change sensor value color to reflect the alarm state of that set parameter – eg. if we have a HAPA alarm, set the PIP control to display as red.

**Returns**
- `AlarmSeverity`

**staticMetaObject = <PySide2.QtCore.QMetaObject object>**

**class** pvp.gui.widgets.display.Limits_Plot *(style: str = 'light', *args, **kwargs)*

Widget to display current value in a bar graph along with alarm limits

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

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

**Type**
- float

**sensor_value**
Sensor value to compare against control, displayed as bar

**Type**
- float

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

**Methods:**

- `init_ui()`
  Create bar chart and horizontal lines to reflect
  continues on next page
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>update_value()</code></td>
<td>Set yrange to ensure that the set value is always in the center of the plot and that all the values are in range.</td>
</tr>
<tr>
<td><code>update_yrange()</code></td>
<td>Move the lines! Pass any of the represented values.</td>
</tr>
</tbody>
</table>

**Attributes:**

- `staticMetaObject`

```python
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```

**Function Definitions:**

```python
def init_ui()
    Create bar chart and horizontal lines to reflect
    • Sensor Value
    • Set Value
    • High alarm limit
    • Low alarm limit

def update_value(min: float = None, max: float = None, sensor_value: float = None, set_value: float = None)
    Move the lines! Pass any of the represented values.
    Also updates yrange to ensure that the control value is always centered in the plot

    Parameters
    • `min (float)` – new alarm minimum
    • `max (float)` – new alarm maximum
    • `sensor_value (float)` – new value for the sensor bar plot
    • `set_value (float)` – new value for the set value line

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

**10.2.4 Plot**

Widgets to plot waveforms of sensor values

The `PVP_Gui` creates a `Plot_CONTAINER` that allows the user to select

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

Plots display alarm limits as red horizontal bars

**Classes:**

- `AlarmSeverity(value)`
  An enumeration.
- `ControlSetting(name, value, min_value, ...)`
  Message containing ventilation control parameters.
- `Plot(name[, buffer_size, plot_duration, ...])`
  Waveform plot of single sensor value.

continues on next page
Table 31 – continued from previous page

| **Plot_Container**(*plot_descriptors, ...*) | Container for multiple :class:`Plot` objects |
| **SensorValues**([timestamp, loop_counter, ...]) | Structured class for communicating sensor readings throughout PVP. |
| **Value**(name, units, abs_range, safe_range, ...) | Class to parameterize how a value is used in PVP. |
| **ValueName**(value) | Canonical names of all values used in PVP. |
| **deque**(value) | `deque([iterable[, maxlen]])` \rightarrow `deque` object |

**Data:**

| **PLOT_FREQ** | Update frequency of Plot\'s in Hz |
| **PLOT_TIMER** | A QTimer that updates :class:`TimedPlotCurveItem`\'s |

**Functions:**

| get_gui_instance() | Retreive the currently running instance of the GUI |
| init_logger(module_name[, log_level, ...]) | Initialize a logger for logging events. |
| mono_font() | Module function to return a PySide2.QtGui.QFont to use as the mono font. |

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

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

class pvp.gui.widgets.plot.Plot(name, buffer_size=4092, plot_duration=10, plot_limits: tuple = None, color=\None, units=\"", **kwargs)
Waveform plot of single sensor value.
Plots values continuously, wrapping at zero rather than shifting x axis.

Parameters

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

timestamps
`deque` of timestamps
Type `collections.deque`

history
`deque` of sensor values
Type `collections.deque`
```
cycles
deque of breath cycles

Type  collections.deque

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

Attributes:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>limits_changed(*args, **kwargs)</td>
<td></td>
</tr>
<tr>
<td>staticMetaObject</td>
<td></td>
</tr>
</tbody>
</table>

Methods:

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

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

```python
set_duration(dur: float)
Set duration, or span of x axis.
Parameters:
  dur (float) – span of x axis (in seconds)
```

```python
update_value(new_value: tuple)
Update with new sensor value
Parameters:
  new_value (tuple) – (timestamp from time.time(), breath_cycle, value)
```

```python
set_safe_limits(limits: pvp.common.message.ControlSetting)
Set the position of the max and min lines for a given value
Parameters:
  limits (ControlSetting) – Controlsetting that has either a min_value or max_value defined
```

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

```python
set_units(units)
Set displayed units
Currently only implemented for Pressure, display either in cmH2O or hPa
Parameters:
  units (‘cmH2O’, ‘hPa’) – unit to display pressure as
```

```python
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```

```python
class pvp.gui.widgets.plot.Plot_Container(plot_descriptors: Dict[pvp.common.values.ValueName, pvp.common.values.Value], *args, **kwargs)
Container for multiple :class:`Plot` objects
Allows user to show/hide different plots and adjust x-span (time zoom)
```
Note: Currently, the only unfortunately hardcoded parameter in the whole GUI is the instruction to initially hide FIO2, there should be an additional parameter in Value that says whether a plot should initialize as hidden or not.

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

Parameters

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

**plots**

Dict mapping ValueName s to Plot s

Type dict

**slider**

slider used to set x span

Type PySide2.QtWidgets.QSlider

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_ui()</td>
<td>Call Plot.reset_start_time() on all plots</td>
</tr>
<tr>
<td>reset_start_time()</td>
<td>Set the current duration (span of the x axis) of all plots</td>
</tr>
<tr>
<td>set_duration(duration)</td>
<td>Try to set horizontal alarm limits on all relevant plots</td>
</tr>
<tr>
<td>set_plot_mode()</td>
<td>Toggle the visibility of a plot.</td>
</tr>
<tr>
<td>update_value(vals)</td>
<td>Try to update all plots who have new sensorvalues</td>
</tr>
</tbody>
</table>

Attributes:

**staticMetaObject**

init_ui()  

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

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

toggle_plot (state: bool)

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

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

Parameters control (ControlSetting) – with either min_value or max_value set
Returns:

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

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

**Parameters**

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

Returns:

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

*staticMetaObject = <PySide2.QtCore.QMetaObject object>*

**set_plot_mode** ()

---

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

### 10.2.5 Components

Very basic components used by other widgets.

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

**Classes:**

- **DoubleSlider**(decimals=1, *args, **kargs)**
  Slider capable of representing floats
  Ripped off from and https://stackoverflow.com/a/50300848 , Thank you!!!

- **EditableLabel**(parent)
  Editable label https://gist.github.com/mfessenden/baa2b87b8addb0b60e54a11c1da48046

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

- **OnOffButton***(state_labels, str=, toggled, ...)*
  Simple extension of toggle button with styling for clearer ‘ON’ vs ‘OFF’

- **QHLine**(parent, color)**
  with respect to https://stackoverflow.com/a/51057516

- **QVLine**(parent, color)**

**Functions:**

- **mono_font()**
  module function to return a PySide2.QtGui.QFont to use as the mono font.

```
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()
```

---

continues on next page
Table 40 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_singleStep()</code></td>
</tr>
<tr>
<td><code>emitDoubleValueChanged()</code></td>
</tr>
<tr>
<td><code>maximum(self)</code></td>
</tr>
<tr>
<td><code>minimum(self)</code></td>
</tr>
<tr>
<td><code>setDecimals(decimals)</code></td>
</tr>
<tr>
<td><code>setMaximum(self, arg__1)</code></td>
</tr>
<tr>
<td><code>setMinimum(self, arg__1)</code></td>
</tr>
<tr>
<td><code>setSingleStep(self, arg__1)</code></td>
</tr>
<tr>
<td><code>setValue(self, arg__1)</code></td>
</tr>
<tr>
<td><code>singleStep(self)</code></td>
</tr>
<tr>
<td><code>value(self)</code></td>
</tr>
</tbody>
</table>

Attributes:

```python
doubleValueChanged(*args, **kwargs)
```

```python
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```

class pvp.gui.widgets.components.KeyPressHandler

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

Attributes:

```python
escapePressed(*args, **kwargs)
```

```python
returnPressed(*args, **kwargs)
```

```python
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```
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/baa2b87b8addb0b6060e54a11c1da48046

Methods:

create_signals()
escapePressedAction() Escape event handler
labelPressedEvent(event) Set editable if the left mouse button is clicked
setLabelEditableAction() Action to make the widget editable
setText(text) Standard QLabel text setter
setEditable(editable)

Attributes:

textChanged(*args, **kwargs)
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)

Chapter 10. GUI
class pvp.gui.widgets.components.QHLine(parent=None, color='#FFFFFF')
with respect to https://stackoverflow.com/a/51057516

Methods:

setColor(color)

Attributes:

staticMetaObject

setColor(color)
staticMetaObject = <PySide2.QtCore.QMetaObject object>

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

Methods:

setColor(color)

Attributes:

staticMetaObject

setColor(color)
staticMetaObject = <PySide2.QtCore.QMetaObject object>


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, toggled)

param state_labels tuple of strings to set when toggled and untoggled

set_state(state)

Attributes:

staticMetaObject

__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

```python
set_state(state: bool)
```

```python
staticMetaObject = <PySide2.QtCore.QMetaObject object>
```

### 10.2.6 Dialog

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

**Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_gui_instance()</td>
<td>Retreive the currently running instance of the GUI</td>
</tr>
<tr>
<td>init_logger(module_name[, log_level,...])</td>
<td>Initialize a logger for logging events.</td>
</tr>
<tr>
<td>pop_dialog(message[, sub_message, modality,...])</td>
<td>Creates a dialog box to display a message.</td>
</tr>
</tbody>
</table>

```python
```

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
10.3 GUI Stylesheets

Classes:

| AlarmSeverity(value) | An enumeration. |

Data:

| MONITOR_UPDATE_INTERVAL | inter-update interval (seconds) for Monitor |

Functions:

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

```python
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
```

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

The GUI also feeds the `Alarm_Manager` objects so that it can compute alarm state. The `Alarm_Manager` reciprocally updates the GUI with `Alarm` states (`PVP_Gui.handle_alarm()`) and Alarm limits (`PVP_Gui.limits_updated()`).

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

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

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

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

```bash
python3 -m pvp.main
```
10.4 Module Overview

10.5 Screenshot

images/gui_overview_v1_1920px.png
11.1 Screenshot

images/single_waveform.png
The Controller consists of one main controller object that receives sensor-data, and computes control parameters, to change valve settings. The controller receives ventilation control parameters (see set_control()), and can provide the currently active set of controls (see ???)

The Controller also feeds the Logger SensorValues objects so that it can store high-temporal resolution data.

The main polling loop of the Controller is PVP_Gui.update_gui() which queries the Hardware for new variables, that are wired up in a new SensorValues and distributes them to all listening widgets (see method documentation for more details).

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

- Display widgets in the left “sensor monitor” box from all Values in DISPLAY_MONITOR.

The Controller can be launched alone:

but was not intended to be launched alone.

add logging

Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm(alarm_type, severity, start_time,...)</td>
<td>Representation of alarm status and parameters</td>
</tr>
<tr>
<td>AlarmSeverity(value)</td>
<td>An enumeration.</td>
</tr>
<tr>
<td>AlarmType(value)</td>
<td>An enumeration.</td>
</tr>
<tr>
<td>Balloon_Simulator(peep_valve)</td>
<td>Physics simulator for inflating a balloon with an attached PEEP valve.</td>
</tr>
<tr>
<td>ControlModuleBase(save_logs, flush_every)</td>
<td>Abstract controller class for simulation/hardware.</td>
</tr>
<tr>
<td>ControlModuleDevice([save_logs,...])</td>
<td>Uses ControlModuleBase to control the hardware.</td>
</tr>
<tr>
<td>ControlModuleSimulator([simulator_dt,...])</td>
<td>Controlling Simulation.</td>
</tr>
<tr>
<td>ControlSetting(name, value, min_value,...)</td>
<td>Message containing ventilation control parameters.</td>
</tr>
<tr>
<td>ControlValues(control_signal_in,...)</td>
<td>Class to save control values, analogous to SensorValues.</td>
</tr>
<tr>
<td>DataLogger(compression_level)</td>
<td>Class for logging numerical respiration data and control settings.</td>
</tr>
<tr>
<td>DerivedValues(timestamp, breath_count,...)</td>
<td>Class to save derived values, analogous to SensorValues.</td>
</tr>
<tr>
<td>SensorValues([timestamp, loop_counter,...])</td>
<td>Structured class for communicating sensor readings throughout PVP.</td>
</tr>
<tr>
<td>ValueName(value)</td>
<td>Canonical names of all values used in PVP.</td>
</tr>
<tr>
<td>count</td>
<td>count(start=0, step=1) -&gt; count object</td>
</tr>
<tr>
<td>deque</td>
<td>deque([iterable[, maxlen]]) -&gt; deque object</td>
</tr>
</tbody>
</table>

Data:

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>The central part of internal API.</td>
</tr>
</tbody>
</table>

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_control_module([sim_mode, simulator_dt])</td>
<td>Generates control module.</td>
</tr>
</tbody>
</table>

continues on next page
Table 3 – continued from previous page

| init_logger(module_name[, log_level,...]) | Initialize a logger for logging events. |
| timeout(func) | Defines a decorator for a 50ms timeout. |

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

Abstract controller class for simulation/hardware.

1. General notes: All internal variables fall in three classes, denoted by the beginning of the variable:
   - `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 through the `set_` and `get_` functions.

   These functions act on COPIES of internal variables `__` and `_`, that are sync’ed 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]
Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_PID_update(dt)</code></td>
<td>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: <code>__SET_PIP_GAIN</code>) 2) Sustain PIP pressure 3) Quick fall to PEEP 4) Sustain 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.</td>
</tr>
<tr>
<td><code>__init__((save_logs, flush_every))</code></td>
<td>Initializes the ControlModuleBase class.</td>
</tr>
<tr>
<td><code>_control_reset()</code></td>
<td>Resets the internal controller cycle to zero, i.e. restarts the breath cycle.</td>
</tr>
<tr>
<td><code>_get_control_signal_in()</code></td>
<td>Produces the INSPIRATORY control-signal that has been calculated in <code>__calculate_control_signal_in(dt)</code></td>
</tr>
<tr>
<td><code>_get_control_signal_out()</code></td>
<td>Produces the EXPIRATORY control-signal for the different states, i.e. open/close.</td>
</tr>
<tr>
<td><code>_initialize_set_to_COPY()</code></td>
<td>Makes a copy of internal variables.</td>
</tr>
<tr>
<td><code>_sensor_to_COPY()</code></td>
<td></td>
</tr>
<tr>
<td><code>_start_mainloop()</code></td>
<td>Prototype method to start main PID loop.</td>
</tr>
<tr>
<td><code>get_alarms()</code></td>
<td>A method callable from the outside to get a copy of the alarms, that the controller checks: High airway pressure, and technical alarms.</td>
</tr>
<tr>
<td><code>get_control(control_setting_name)</code></td>
<td>A method callable from the outside to get current control settings.</td>
</tr>
<tr>
<td><code>get_heartbeat()</code></td>
<td>Returns an independent heart-beat of the controller, i.e. the internal loop counter incremented in <code>_start_mainloop</code>.</td>
</tr>
<tr>
<td><code>get_past_waveforms()</code></td>
<td>Public method to return a list of past waveforms from <code>_cycle_waveform_archive</code>. 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].</td>
</tr>
<tr>
<td><code>get_sensors()</code></td>
<td>A method callable from the outside to get a copy of sensorValues</td>
</tr>
<tr>
<td><code>interrupt()</code></td>
<td>If the controller seems stuck, this generates a new thread, and starts the main loop.</td>
</tr>
<tr>
<td><code>is_running()</code></td>
<td>Public Method to assess whether the main loop thread is running.</td>
</tr>
<tr>
<td><code>set_breath_detection(breath_detection)</code></td>
<td>A method callable from the outside to set alarms.</td>
</tr>
<tr>
<td><code>set_control(control_setting)</code></td>
<td>Method to start <code>_start_mainloop</code> as a thread.</td>
</tr>
<tr>
<td><code>stop()</code></td>
<td>Method to stop the main loop thread, and close the logfile.</td>
</tr>
</tbody>
</table>

`__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`) A method to set breath detection.

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

Parameters \( dt (\text{float}) \) – 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 occurrence.

_\_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) Sustain 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 \( dt (\text{float}) \) – 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-iterations? 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-iterations? 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 breath 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OUUpdate(variable, dt, mu, sigma, tau)</code></td>
<td>This is a simple function to produce an OU process on <code>variable</code>.</td>
</tr>
<tr>
<td><code>_reset()</code></td>
<td>Resets Balloon to default settings.</td>
</tr>
<tr>
<td><code>get_pressure()</code></td>
<td></td>
</tr>
<tr>
<td><code>get_volume()</code></td>
<td></td>
</tr>
<tr>
<td><code>set_flow_in(Qin, dt)</code></td>
<td></td>
</tr>
<tr>
<td><code>set_flow_out(Qout, dt)</code></td>
<td></td>
</tr>
<tr>
<td><code>update(dt)</code></td>
<td></td>
</tr>
</tbody>
</table>

`OUUpdate()`:

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:
__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 the simulation. Defaults to None.

Returns  Either configured for simulation, or physical device.

Return type  ControlModule-Object

Control into a breathing cycle was accomplished with a hybrid system of state and PID control. During inspiration, we actively control pressure using a PID cycle to set the inspiratory valve. Expiration was then instantiated by closing the inspiratory, and opening the expiratory valve to passively release PIP pressure as fast as possible. After reaching PEEP, we opened the inspiratory valve slightly to sustain PEEP using the aforementioned manually operated PEEP valve and to sustain a gentle flow of air through the system.

11.1. Screenshot
The Raspberry pi allowed for the primary control loop to run at speeds exceeding 320Hz, using 40% of the maximum bandwidth of the analog-to-digital converter reading the sensors.

In addition to pressure control, our software continuously monitors for autonomous breaths, high airway pressure, and general system status. Autonomous breathing was detected by transient pressure drops below PEEP. A detected breath triggered a new breath cycle. High airway pressure is defined as exceeding a certain pressure for a certain time (as to not be triggered by a cough). This triggered an alarm, and an immediate release of air to drop pressure to PEEP. The Controller also assesses whether numerical values are reasonable, and changing over time. If this is not the case, it raises a technical alarm. All alarms are collected and maintained by an intelligent alarm manager, that provides the UI with the alarms to display in order of their importance. In addition to the alarm system, the controller monitors for autonomous breath events during PEEP. We define such events by a drop below the PEEP baseline exceeding some fixed threshold. If an autonomous drop was detected, then the next breath cycle is initiated.
12.1 Values

Parameterization of variables and values used in PVP.

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

**Data:**

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL</td>
<td>Values to control but not monitor.</td>
</tr>
<tr>
<td>DISPLAY_CONTROL</td>
<td>Control values that should also have a widget created in the GUI.</td>
</tr>
<tr>
<td>DISPLAY_MONITOR</td>
<td>Those sensor values that should also have a widget created in the GUI.</td>
</tr>
<tr>
<td>PLOTS</td>
<td>Values that can be plotted</td>
</tr>
<tr>
<td>SENSOR</td>
<td>Sensor values</td>
</tr>
<tr>
<td>VALUES</td>
<td>Declaration of all values used by PVP</td>
</tr>
</tbody>
</table>

**Classes:**

```python
Enum(value)       | Generic enumeration.                                                        |
Value(name, units, abs_range, safe_range, ...) | Class to parameterize how a value is used in PVP.                           |
ValueName(value)  | Canonical names of all values used in PVP.                                  |
auto()            | Instances are replaced with an appropriate value in Enum class suites.      |
odict             | alias of collections.OrderedDict                                           |
```

```python
class pvp.common.values.ValueName(value)
```

Bases: `enum.Enum`

Canonical names of all values used in PVP.

**Attributes:**

```python
PIP
PIP_TIME
PEEP
PEEP_TIME
BREATHS_PER_MINUTE
```

continues on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>PIP</th>
<th>PIP_TIME</th>
<th>PEEP</th>
<th>PEEP_TIME</th>
<th>BREATHS_PER_MINUTE</th>
<th>INSPIRATION_TIME_SEC</th>
<th>IE_RATIO</th>
<th>FIO2</th>
<th>VTE</th>
<th>PRESSURE</th>
<th>FLOWOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

```python

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

Bases: object

Class to parameterize how a value is used in PVP.

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

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

Parameters

- **name (str)** – Human-readable name of the value
- **units (str)** – Human-readable description of units
- **abs_range (tuple)** – tuple of ints or floats setting the logical limit of the value, eg. a percent between 0 and 100, (0, 100)
- **safe_range (tuple)** – tuple of ints or floats setting the safe ranges of the value, note:
  ```
  this is not the same thing as the user-set alarm values, though the user-set alarm values are initialized as `"safe_range"`.  
  ```
- **decimals (int)** – the number of decimals of precision used when displaying the value
- **control (bool)** – Whether or not the value is used to control ventilation
• **sensor** *(bool)* – Whether or not the value is a measured sensor value

• **display** *(bool)* – whether the value should be created as a `gui.widgets.Display` widget.

• **plot** *(bool)* – whether or not the value is plottable in the center plot window

• **plot_limits** *(None, tuple(ValueName))* – If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot. eg. the PIP alarm range limits should be plotted on the Pressure plot

• **control_type** *(None, "slider", "record")* – If a control sets whether the control should use a slider or be set by recording recent sensor values.

• **group** *(None, str)* – Unused currently, but to be used to create subgroups of control & display widgets

• **default** *(None, int, float)* – Default value, if any. (Not automatically set in the GUI.)

**Methods:**

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

**Attributes:**

- **abs_range**
  - tuple of ints or floats setting the logical limit of the value, eg.

- **control**
  - Whether or not the value is used to control ventilation

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

- **decimals**
  - The number of decimals of precision used when displaying the value

- **default**
  - Default value, if any.

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

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

- **name**
  - Human readable name of value

- **plot**
  - whether or not the value is plottable in the center plot window

- **plot_limits**
  - If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot.

- **safe_range**
  - tuple of ints or floats setting the safe ranges of the value,

- **sensor**
  - Whether or not the value is a measured sensor value
__init__(name: str, units: str, abs_range: tuple, safe_range: tuple, decimals: int, control: bool, sensor: bool, display: bool, plot: bool = False, plot_limits: Union[None, Tuple[pvp.common.values.ValueName]] = None, control_type: Union[None, str] = None, group: Union[None, dict] = None, default: (<class 'int'>, <class 'float'>) = None)

Parameters

• **name** *(str)* – Human-readable name of the value

• **units** *(str)* – Human-readable description of units

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

• **safe_range** *(tuple)* – tuple of ints or floats setting the safe ranges of the value,


  note:

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

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

• **control** *(bool)* – Whether or not the value is used to control ventilation

• **sensor** *(bool)* – Whether or not the value is a measured sensor value

• **display** *(bool)* – whether the value should be created as a `gui.widgets.Display` widget.

• **plot** *(bool)* – whether or not the value is plottable in the center plot window

• **plot_limits** *(None, tuple(ValueName))* – If plottable, and the plotted value has some alarm limits for another value, plot those limits as horizontal lines in the plot. eg. the PIP alarm range limits should be plotted on the Pressure plot

• **control_type** *(None, "slider", "record")* – If a control sets whether the control should use a slider or be set by recording recent sensor values.

• **group** *(None, str)* – Unused currently, but to be used to create subgroups of control & display widgets

• **default** *(None, int, float)* – Default value, if any. (Not automatically set in the GUI.)

**property name**

Human readable name of value

**Returns** str

**property abs_range**

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

**Returns** tuple

**property safe_range**

tuple of ints or floats setting the safe ranges of the value,


  note:

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

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

Returns int

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

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

Returns bool

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

Returns bool

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

Returns bool

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

Returns None, “slider”, “record”

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

Returns None, str

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

Returns bool

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

Returns None, typing.Tuple[ValueName]

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

Returns dict

pvp.common.values.VALUES = OrderedDict([(ValueName.PIP: 1), <pvp.common.values.Value object>], ...

Declaration of all values used by PVP

pvp.common.valuesSENSOR = OrderedDict([(ValueName.PIP: 1), <pvp.common.values.Value object>], ...

Sensor values

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

pvp.common.valuesCONTROL = OrderedDict([(ValueName.PIP: 1), <pvp.common.values.Value object>], ...

Values to control but not monitor.

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

12.1. Values
pvp.common.values.DISPLAY_MONITOR = OrderedDict((<ValueName.PIP: 1>, pvp.common.values.Value), (<ValueName.PEEP: 3>, pvp.common.values.Value), ..., (<ValueName.FLOWOUT: 11>, pvp.common.values.Value), (<ValueName.FIO2: 8>, pvp.common.values.Value))

Those sensor values that should also have a widget created in the GUI

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

pvp.common.values.DISPLAY_CONTROL = OrderedDict((<ValueName.PIP: 1>, pvp.common.values.Value), (<ValueName.PEEP: 3>, pvp.common.values.Value), ..., (<ValueName.PIP_TIME: 2>, pvp.common.values.Value))

Control values that should also have a widget created in the GUI

Automatically generated as all Value objects in VALUES where control == True and display == True

pvp.common.values.PLOTS = OrderedDict((<ValueName.PRESSURE: 10>, pvp.common.values.Value), (<ValueName.FLOWOUT: 11>, pvp.common.values.Value), (<ValueName.FIO2: 8>, pvp.common.values.Value))

Values that can be plotted

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

12.2 Message

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

• SensorValues are used to communicate sensor readings between the controller, GUI, and alarm manager

• ControlSetting is used to set ventilation controls from the GUI to the controller.

Classes:

ControlSetting(name, value, min_value, ...) Message containing ventilation control parameters.

ControlValues(control_signal_in, ...) Class to save control values, analogous to SensorValues.

DerivedValues(timestamp, breath_count, ...) Class to save derived values, analogous to SensorValues.

SensorValues([timestamp, loop_counter, ...]) Structured class for communicating sensor readings throughout PVP.

odict alias of collections.OrderedDict

Functions:

copy(x) Shallow copy operation on arbitrary Python objects.

init_logger(module_name[, log_level, ...]) Initialize a logger for logging events.

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

Bases: object

Structured class for communicating sensor readings throughout PVP.

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

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

Parameters

• timestamp (float) – from time.time(). must be passed explicitly or as an entry in vals
• **loop_counter** (*int*) – number of control_module loops. must be passed explicitly or as an entry in vals

• **breath_count** (*int*) – number of breaths taken. must be passed explicitly or as an entry in vals

• **vals** (*None, dict*) – Dict of {ValueName: float} that contains current sensor readings. Can also be equivalently given as **kwargs**. if None, assumed values are being passed as kwargs, but an exception will be raised if they aren’t.

• ****kwargs – sensor readings, must be in pvp.values.SENSOR.keys

Methods:

```
__init__([timestamp, loop_counter, ...])
```

```
param timestamp from time.time(). must be passed explicitly or as an entry in vals
```

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

Attributes:

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

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

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

```
__init__ (timestamp=None, loop_counter=None, breath_count=None, vals=typing.Union[NoneType, typing.Dict[ForwardRef('ValueName'), float]], **kwargs)
```

Parameters

```
• timestamp (float) – from time.time(). must be passed explicitly or as an entry in vals
```

```
• loop_counter (int) – number of control_module loops. must be passed explicitly or as an entry in vals
```

```
• breath_count (int) – number of breaths taken. must be passed explicitly or as an entry in vals
```

```
• vals (None, dict) – Dict of {ValueName: float} that contains current sensor readings. Can also be equivalently given as kwargs. if None, assumed values are being passed as kwargs, but an exception will be raised if they aren’t.
```

```
• **kwargs – sensor readings, must be in pvp.values.SENSOR.keys
```

```
to_dict() → dict
```

```
Return a dictionary of all sensor values and additional values
```

Returns  dict

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

Bases: object
Message containing ventilation control parameters.

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

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

Parameters

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

Methods:

```python
__init__(name[, value, min_value, ...]) Message containing ventilation control parameters.
```

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

Message containing ventilation control parameters.

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

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

Parameters

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

```python
class pvp.common.message.ControlValues(control_signal_in, control_signal_out)
Bases: object
```

Class to save control values, analogous to SensorValues.

Used by the controller to save waveform data in DataLogger.store_waveform_data() and ControlModuleBase.__save_values()
Key difference: SensorValues come exclusively from the sensors, ControlValues contains controller variables, i.e. control signals and controlled signals (the flows).

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

Bases: object

Class to save derived values, analogous to SensorValues.

Used by controller to store derived values (like PIP from Pressure) in DataLogger.

store_derived_data() and in ControlModuleBase.__analyze_last_waveform()

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

12.3 Loggers

Logging functionality

There are two types of loggers:

- `loggers.init_logger()` creates a standard logging.Logger-based logging system for debugging and recording system events, and a

- `loggers.DataLogger`- a tables-based class to store continuously measured sensor values.

Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContinuousData</td>
<td>Structure for the hdf5-table for continuous waveform data; measured once per controller loop.</td>
</tr>
<tr>
<td>ControlCommand</td>
<td>Structure for the hdf5-table to store control commands.</td>
</tr>
<tr>
<td>CycleData()</td>
<td>Structure for the hdf5-table to store derived quantities from waveform measurements.</td>
</tr>
<tr>
<td>DataLogger(compression_level)</td>
<td>Class for logging numerical respiration data and control settings. Creates a hdf5 file with this general structure: / root</td>
</tr>
</tbody>
</table>

```python
datetime(year, month, day[, hour[, minute[, ...]])
```

The year, month and day arguments are required.

Data:

| LOGGERS | list of strings, which loggers have been created already. |

Functions:

```python
init_logger(module_name[, log_level, ...])
```

Initialize a logger for logging events.

continues on next page
Table 13 – continued from previous page

**update_logger_sizes()**
Adjust each logger’s `maxBytes` attribute so that the total across all loggers is `prefs.LOGGING_MAX_BYTES`.

---

```python
pvp.common.loggers._LOGGERS = ['pvp.common.prefs', 'pvp.alarm.alarm_manager']
list of strings, which loggers have been created already.

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

Initialize a logger for logging events.

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

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

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

otherwise configure and return the logger such that

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

Parameters

- **module_name** (*str*) – module name used to generate filename and name logger
- **log_level** (*int*) – one of `:var:`logging.DEBUG`, `:var:`logging.INFO`, `:var:`logging.WARNING`, or `:var:`logging.ERROR`
- **file_handler** (*bool, str*) – if True, (default), log in `<logdir>/module_name.log`. If False, don’t log to disk.

Returns

Logger 4 u 2 use

Return type `logging.Logger`

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

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

```python
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:

```
/ root  /
| waveforms (group) |
| time | pressure_data | flow_out | control_signal_in | control_signal_out | FiO2 | Cycle No. |
| controls (group) |
| 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 continuous numerical logger class.
Parameters **compression_level** (*int, optional*) – Compression level of the hdf5 file.
Defaults to 9.

Methods:

```python
__init__([compression_level])
Initialized the continuous 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. 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.

rotation_newfile()
This rotates through filenames, similar to a ring-buffer, to make sure that the program does not run out 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.
```

```python
__init__(compression_level: int = 9)
Initialized the continuous 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

12.3. Loggers
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 cur-
   rently 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 cur-
   rently 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 cur-
   rently open file. Defaults to None.
12.4 Prefs

Prefs set configurable parameters used throughout PVP.

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

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

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

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

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

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

Data:

- `_LOADED`: 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`: Locks access to `prefs_fn`.
- `_LOGGER`: A `logging.Logger` to log pref init and setting events.
- `_PREFS`: The dict created by `prefs._PREF_MANAGER` to store prefs.
- `_PREF_MANAGER`: The `multiprocessing.Manager` that stores prefs during system operation.

Classes:

- `c_bool`

Functions:

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

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

```python
pvp.common.prefs._PREFS = <DictProxy object, typeid 'dict'>
```

The dict created by `prefs._PREF_MANAGER` to store prefs.

```python
pvp.common.prefs._LOGGER = <Logger pvp.common.prefs (WARNING)>
```

A `logging.Logger` to log pref init and setting events

```python
pvp.common.prefs._LOCK = <Lock(owner=None)>
```

Locks access to `prefs_fn`

The `mp.Lock`

```python
pvp.common.prefs._DIRECTORIES = {'DATA_DIR': '/home/docs/pvp/logs', 'LOG_DIR': '/home/docs/pvp/logs', 'VENT_DIR': '/home/docs/pvp'}
```

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

```python
pvp.common.prefs._LOADED = <Synchronized wrapper for c_bool(True)>
```

A flag to indicate whether prefs have been loaded (and thus `set_pref()` should write to disk).

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

The `bool`

```python
pvp.common.prefs._DEFAULTS = {'BREATH_DETECTION': True, 'BREATH_PRESSURE_DROP': 4, 'CONTROLLER_LOOPS_UNTIL_UPDATE': 1, 'CONTROLLER_MAX_FLOW': 10, 'LOGGING_MAX_BYTES': 2*1024**3, 'LOGGING_MAX_FILES': 5, 'LOGLEVEL': 'WARNING', 'PREFS_FN': None, 'TIME_FIRST_START': None, 'TIMEOUT': 0.05, 'HEARTBEAT_TIMEOUT': 0.02, 'GUI_STATE_FN': 'gui_state.json', 'GUI_UPDATE_TIME': 0.05, 'ENABLE_DIALOGS': True, 'ENABLE_WARNINGS': True}
```

Declare all available parameters and set default values. If no default, set as None.

- **PREFS_FN**: absolute path to the prefs file
- **TIME_FIRST_START**: time when the program has been started for the first time
- **VENT_DIR**: ~/pvp - base directory for user storage
- **LOG_DIR**: ~/pvp/logs - for storage of event and alarm logs
- **DATA_DIR**: ~/pvp/data - for storage of waveform data
- **LOGGING_MAX_BYTES**: the total storage space for all loggers – each logger gets `LOGGING_MAX_BYTES/len(loggers)` space (2GB by default)
- **LOGGING_MAX_FILES**: number of files to split each logger’s logs across (default: 5)
- **LOGLEVEL**: One of ('DEBUG', 'INFO', 'WARNING', 'EXCEPTION') that sets the minimum log level that is printed and written to disk
- **TIMEOUT**: timeout used for timeout decorators on time-sensitive operations (in seconds, default 0.05)
- **HEARTBEAT_TIMEOUT**: Time between heartbeats between GUI and controller after which contact is assumed to be lost (in seconds, default 0.02)
- **GUI_STATE_FN**: Filename of gui control state file, relative to `VENT_DIR` (default: gui_state.json)
- **GUI_UPDATE_TIME**: Time between calls of `PVP_Gui.update_gui()` (in seconds, default: 0.05)
- **ENABLE_DIALOGS**: Enable all GUI dialogs – set as False when testing on virtual frame buffer that doesn’t support them (default: True and should stay that way)
- **ENABLE_WARNINGS**: Enable user warnings and value change confirmations (default: True)
- **CONTROLLER_MAX_FLOW**: Maximum flow, above which the controller considers a sensor error (default: 10)
• CONTROLLER_MAX_PRESSURE: Maximum pressure, above which the controller considers a sensor error (default: 100)
• CONTROLLER_MAX_STUCK_SENSOR: Max amount of time (in s) before considering a sensor stuck (default: 0.2)
• CONTROLLER_LOOP_UPDATE_TIME: Amount of time to sleep in between controller update times when using ControlModuleDevice (default: 0.0)
• CONTROLLER_LOOP_UPDATE_TIME_SIMULATOR: Amount of time to sleep in between controller updates when using ControlModuleSimulator (default: 0.005)
• CONTROLLER_LOOPS_UNTIL_UPDATE: Number of controller loops in between updating its externally-available COPY attributes retrieved by ControlModuleBase.get_sensor() et al
• CONTROLLER_RINGBUFFER_SIZE: Maximum number of breath cycle records to be kept in memory (default: 100)
• COUGH_DURATION: Amount of time the high-pressure alarm limit can be exceeded and considered a cough (in seconds, default: 0.1)
• BREATH_PRESSURE_DROP: Amount pressure can drop below set PEEP before being considered an autonomous breath when in breath detection mode
• BREATH_DETECTION: Whether the controller should detect autonomous breaths in order to reset ventilation cycles (default: True)

pvp.common.prefs.set_pref (key: str, val)
Sets a pref in the manager and, if prefs.LOADED is True, calls prefs.save_prefs()

Parameters

• key (str) – Name of pref key
• val – Value to set

pvp.common.prefs.get_pref (key: str = None)
Get global configuration value

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

pvp.common.prefs.load_prefs (prefs_fn: str)
Load prefs from a .json prefs file, combining (and overwriting) any existing prefs, and then saves. Called on pvp import by prefs.init()
Also initializes prefs._LOGGER

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

Parameters prefs_fn (str) – path of prefs.json

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

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

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

12.4. Prefs
pvp.common.pref.s.init()
   Initialize prefs. Called in pvp.__init__.py to ensure prefs are initialized before anything else.

## 12.5 Unit Conversion

Functions that convert between units

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

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

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

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

### Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmH2O_to_hPa</td>
<td>Convert cmH2O to hPa</td>
</tr>
<tr>
<td>hPa_to_cmH2O</td>
<td>Convert hPa to cmH2O</td>
</tr>
<tr>
<td>rounded_string</td>
<td>Create a rounded string of a number that doesn't have trailing .0 when decimals = 0</td>
</tr>
</tbody>
</table>

```python
pvp.common.unit_conversion.cmH2O_to_hPa(pressure) -> float
   Convert cmH2O to hPa
   Parameters pressure (float) - Pressure in cmH2O
   Returns Pressure in hPa (pressure*98.0665)
   Return type float

pvp.common.unit_conversion.hPa_to_cmH2O(pressure) -> float
   Convert hPa to cmH2O
   Parameters pressure (float) - Pressure in hPa
   Returns Pressure in cmH2O (pressure/98.0665)
   Return type float

pvp.common.unit_conversion.rounded_string(value[, decimals]) -> str
   Create a rounded string of a number that doesn't have trailing .0 when decimals = 0
   Parameters
       * value (float) - Value to stringify
       * decimals (int) - Number of decimal places to round to
   Returns Clean rounded string version of number
   Return type str
```
12.6 utils

Exceptions:

TimeoutException

Functions:

contextmanager(func)  @contextmanager decorator.
init_logger(module_name[, log_level,...]) Initialize a logger for logging events.
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")

12.7 fashion

Decorators for dangerous functions

Functions:

init_logger(module_name[, log_level,...]) Initialize a logger for logging events.
locked(func) Wrapper to use as decorator, handle lock logic for a @property
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)
PVP.IO PACKAGE

13.1 Subpackages

13.2 Submodules

13.3 pvp.io.hal module

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

Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hal</td>
<td>Hardware Abstraction Layer for ventilator hardware.</td>
</tr>
<tr>
<td>Sensor()</td>
<td>Abstract base Class describing generalized sensors.</td>
</tr>
</tbody>
</table>

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>import_module(name[, package])</td>
<td>Import a module.</td>
</tr>
<tr>
<td>literal_eval(node_or_string)</td>
<td>Safely evaluate an expression node or a string containing a Python expression.</td>
</tr>
</tbody>
</table>

```python
class pvp.io.hal.Hal(config_file='pvp/io/config/devices.ini')
```

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:

```python
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”)

**Methods:**

**__init__(config_file)** (str) – Initializes HAL from config_file.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aux_pressure</td>
<td>Returns the pressure from the auxiliary pressure sensor, if so equipped.</td>
</tr>
<tr>
<td>flow_ex</td>
<td>The measured flow rate expiratory side.</td>
</tr>
<tr>
<td>flow_in</td>
<td>The measured flow rate inspiratory side.</td>
</tr>
<tr>
<td>oxygen</td>
<td>Returns the oxygen concentration from the primary oxygen sensor.</td>
</tr>
<tr>
<td>pressure</td>
<td>Returns the pressure from the primary pressure sensor.</td>
</tr>
<tr>
<td>setpoint_ex</td>
<td>The currently requested flow on the expiratory side as a proportion of the maximum.</td>
</tr>
<tr>
<td>setpoint_in</td>
<td>The currently requested flow for the inspiratory proportional control valve as a proportion of maximum.</td>
</tr>
</tbody>
</table>

**__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:**
   ```python
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:**
   ```python
   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.

### 13.4 Module contents

**Classes:**

```
Hal([config_file]) Hardware Abstraction Layer for ventilator hardware.
```
14.1 Alarm System Overview

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

14.2 Alarm Modules

14.2.1 Alarm Manager

The alarm manager is responsible for checking the \texttt{Alarm_Rule}s and maintaining the \texttt{Alarm}s active in the system. Only one instance of the \texttt{Alarm_Manager} can be created at once, and if it is instantiated again, the existing object will be returned.

Classes:

- \texttt{Alarm(alarm_type, severity, start_time, \ldots)} Representation of alarm status and parameters
- \texttt{AlarmSeverity(value)} An enumeration.
- \texttt{AlarmType(value)} An enumeration.
- \texttt{Alarm_Manager} The Alarm Manager
- \texttt{Alarm_Rule(name, conditions[, latch, technical])} \texttt{name of rule}
- \texttt{Condition(depends, *args, **kwargs)} Base class for specifying alarm test conditions
- \texttt{ControlSetting(name, value, min_value, \ldots)} Message containing ventilation control parameters.
- \texttt{SensorValues([timestamp, loop_counter, \ldots])} Structured class for communicating sensor readings throughout PVP.

Functions:
**class** pvp.alarm.alarm_manager.Alarm_Manager

The Alarm Manager

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

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

**active_alarms**

*{AlarmType: Alarm}*

Type: dict

**logged_alarms**

A list of deactivated alarms.

Type: list

**dependencies**

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

Type: dict

**pending_clears**

*{AlarmType}* list of alarms that have been requested to be cleared

Type: list

**callbacks**

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

Type: list

**cleared_alarms**

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

Type: list

**snoozed_alarms**

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

Type: dict

**callbacks**

list of callables to send *Alarm* objects to

Type: list

**depends_callbacks**

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

A dict mapping `AlarmType` to `Alarm_Rule`.

Type dict

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

Attributes:

- `_instance`
- `active_alarms`
- `callbacks`
- `cleared_alarms`
- `dependencies`
- `depends_callbacks`
- `logged_alarms`
- `logger`
- `pending_clears`
- `rules`
- `snoozed_alarms`

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_callback(callback)</code></td>
<td>Assert we’re being given a callable and add it to our list of callbacks.</td>
</tr>
<tr>
<td><code>add_dependency_callback(callback)</code></td>
<td>Assert we’re being given a callable and add it to our list of dependency callbacks</td>
</tr>
<tr>
<td><code>check_rule(rule, sensor_values)</code></td>
<td><code>check()</code> the alarm rule, handle logic of raising, emitting, or lowering an alarm.</td>
</tr>
<tr>
<td><code>clear_all_alarms()</code></td>
<td>call <code>Alarm_Manager.deactivate_alarm()</code> for all active alarms.</td>
</tr>
<tr>
<td><code>deactivate_alarm(alarm)</code></td>
<td>Mark an alarm’s internal active flags and remove from <code>active_alarms</code></td>
</tr>
<tr>
<td><code>dismiss_alarm(alarm_type[, duration])</code></td>
<td>GUI or other object requests an alarm to be dismissed &amp; deactivated</td>
</tr>
<tr>
<td><code>emit_alarm(alarm_type, severity)</code></td>
<td>Emit alarm (by calling all callbacks with it).</td>
</tr>
<tr>
<td><code>get_alarm_severity(alarm_type)</code></td>
<td>Get the severity of an Alarm</td>
</tr>
<tr>
<td><code>load_rule(alarm_rule)</code></td>
<td>Add the Alarm Rule to <code>Alarm_Manager.rules</code> and register any dependencies they have with <code>Alarm_Manager.register_dependency()</code></td>
</tr>
<tr>
<td><code>load_rules()</code></td>
<td>Copy alarms from <code>alarm.ALARM_RULES</code> and call <code>Alarm_Manager.load_rule()</code> for each</td>
</tr>
<tr>
<td><code>register_alarm(alarm)</code></td>
<td>Be given an already created alarm and emit to callbacks.</td>
</tr>
<tr>
<td><code>register_dependency(condition, dependency, …)</code></td>
<td>Add dependency in a Condition object to be updated when values are changed</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>Reset all conditions, callbacks, and other stateful attributes and clear alarms</td>
</tr>
<tr>
<td><code>update(sensor_values)</code></td>
<td>Call <code>Alarm_Manager.check_rule()</code> for all rules in <code>Alarm_Manager.rules</code></td>
</tr>
<tr>
<td><code>update_dependencies(control_setting)</code></td>
<td>Update Condition objects that update their value according to some control parameter</td>
</tr>
</tbody>
</table>

14.2. Alarm Modules
_instance = None


dependencies = {}  

pending_clears = []

cleared_alarms = []

snoozed_alarms = {}  

callbacks = []

depends_callbacks = []

rules = {}  

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

load_rules()  

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

load_rule(alarm_rule: pvp.alarm.rule.Alarm_Rule)  
Add the Alarm Rule to Alarm_Manager.rules and register any dependencies they have with Alarm_Manager.register_dependency()

Parameters alarm_rule (Alarm_Rule) – Alarm rule to be loaded

update(sensor_values: pvp.common.message.SensorValues)  
Call Alarm_Manager.check_rule() for all rules in Alarm_Manager.rules

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

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

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

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

Parameters

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

Emit alarm (by calling all callbacks with it).

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

Parameters
• `alarm_type (AlarmType)` –
• `severity (AlarmSeverity)` –

**deactivate_alarm** *(alarm: (<enum 'AlarmType'>, <class 'pvp.alarm.alarm.Alarm'>))*
Mark an alarm’s internal active flags and remove from `active_alarms`
Typically called internally when an alarm is being replaced by one of the same type but a different severity.

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

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

**dismiss_alarm** *(alarm_type: pvp.alarm.AlarmType, duration: float = None)*
GUI or other object requests an alarm to be dismissed & deactivated
GUI will wait until it receives an `emit_alarm` of severity == OFF to remove alarm widgets. If the alarm is not latched
If the alarm is latched, alarm_manager will not decrement alarm severity or emit OFF until a) the condition returns to OFF, and b) the user dismisses the alarm

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

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

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

**Returns** *AlarmSeverity*

**register_alarm** *(alarm: pvp.alarm.alarm.Alarm)*
Be given an already created alarm and emit to callbacks.
Mostly used during testing for programmatically created alarms. Creating alarms outside of the Alarm_Manager is generally discouraged.

**Parameters**
- **alarm (Alarm)** –

Add dependency in a Condition object to be updated when values are changed

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

**update_dependencies** *(control_setting: pvp.common.message.ControlSetting)*
Update Condition objects that update their value according to some control parameter
Call any `transform` functions on the attribute of the control setting specified in the dependency.
Emit another `ControlSetting` describing the new max or min or the value.

**Parameters**

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

### add_callback (callback: Callable)

Assert we’re being given a callable and add it to our list of callbacks.

**Parameters**

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

### add_dependency_callback (callback: Callable)

Assert we’re being given a callable and add it to our list of dependency_callbacks

**Parameters**

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

**Returns:**

- `clear_all_alarms()`
  - call `Alarm_Manager.deactivate_alarm()` for all active alarms.

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

### 14.2.2 Alarm Objects

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

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

#### Classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Alarm</code></td>
<td>Representation of alarm status and parameters</td>
</tr>
<tr>
<td><code>AlarmSeverity</code></td>
<td>An enumeration.</td>
</tr>
<tr>
<td><code>AlarmType</code></td>
<td>An enumeration.</td>
</tr>
<tr>
<td><code>count</code></td>
<td><code>count(start=0, step=1)</code> -&gt; count object</td>
</tr>
<tr>
<td><code>datetime</code></td>
<td>The year, month and day arguments are required.</td>
</tr>
</tbody>
</table>

```python

Representation of alarm status and parameters

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

#### Parameters

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

**id**

unique alarm ID
PVP, Release 0.2.0

Type int

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

active
Whether or not the alarm is currently active
Type bool

Methods:

__init__(alarm_type, severity[, start_time, ...])

param alarm_type Type of alarm

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

Returns:

Attributes:

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

id_counter
used to generate unique IDs for each alarm

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

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

Type itertools.count

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

Parameters

• alarm_type (AlarmType) – Type of alarm

• severity (AlarmSeverity) – Severity of alarm

• start_time (float) – Timestamp of alarm start, (as generated by time.time())

• cause (ValueName) – The ValueName that caused the alarm to be fired

• value (int, float) – optional - numerical value that generated the alarm

• message (str) – optional - override default text generated by AlarmManager

id
unique alarm ID
Type int

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

active
Whether or not the alarm is currently active
Type bool
property severity
Alarm Severity, property without setter to prevent change after instantiation

Returns AlarmSeverity

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

Returns AlarmType

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

14.2.3 Alarm Rule

One Alarm_Rule is defined for each AlarmType in ALARM_RULES.

An alarm rule defines:

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

14.2.3.1 Example

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

First we define the name and behavior of the alarm:

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

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

Next we’ll define a tuple of Condition objects for LOW and MEDIUM severity objects.

Starting with the LOW severity alarm:

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

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

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

Next, we define the MEDIUM severity alarm condition:

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

The first ValueCondition is the same as in the LOW alarm severity condition, except that it is set 15% below PIP. A second CycleAlarmSeverityCondition has been added (with +) to the ValueCondition When conditions are added together, they will only return True (i.e. trigger an alarm) if all of the conditions are met. This condition checks that the LOW_PRESSURE alarm has been active at a LOW severity for at least two cycles.

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

### 14.2.3.2 Module Documentation

Class to declare alarm rules

**Classes:**

- **AlarmSeverity(value)**: An enumeration.
- **AlarmType(value)**: An enumeration.
- **Alarm_Rule(name, conditions[, latch, technical])**:
  - name of rule

- **ValueName(value)**: Canonical names of all values used in PVP.

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

### 14.2. Alarm Modules

...
• name of rule
• conditions: ((alarm_type, (condition_1, condition_2)), . . .)
• latch (bool): if True, alarm severity cannot be decremented until user manually dismisses
• silencing/overriding rules

Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check(sensor_values)</td>
<td>Check all of our conditions.</td>
</tr>
<tr>
<td>reset()</td>
<td></td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>depends</td>
<td>Get all ValueNames whose alarm limits depend on this alarm rule :returns: list[ValueName]</td>
</tr>
<tr>
<td>severity</td>
<td>Last Alarm Severity from .check() :returns: AlarmSeverity</td>
</tr>
<tr>
<td>value_names</td>
<td>Get all ValueNames specified as value_names in alarm conditions</td>
</tr>
</tbody>
</table>

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]

### 14.2.4 Alarm Condition

Condition objects define conditions that can raise alarms. They are used by Alarm_Rules.

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

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

Conditions can
Classes:

- **AlarmSeverity(value)**: An enumeration.
- **AlarmSeverityCondition(alarm_type, severity, ...)**: Alarm is above or below a certain severity.
- **AlarmType(value)**: An enumeration.
- **Condition(depending, *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.
- **SensorValues([timestamp, loop_counter, ...])**: Structured class for communicating sensor readings throughout PVP.
- **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.
- **ValueName(value)**: Canonical names of all values used in PVP.

Functions:

- **_get_alarm_manager()**

```python
def _get_alarm_manager()
```

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

```text
Bases: object

Base class for specifying alarm test conditions

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

Condition objects can be added together to create compound conditions.

_type_ 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:

```python
{'value_name':ValueName, 'value_attr':attr in ControlMessage, 'condition_attr',
    optional: transformation: callable)
```

that declare what values are needed to update
Methods:

```python
__init__(depends)
```

**param depends**

```python
check(sensor_values)
```

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

```python
reset()
```

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

Attributes:

```python
manager
```

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

```python
__init__(depends=\texttt{dict = None}, *args, **kwargs)
```

**Parameters**

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

  ```python
  
  \{
  'value\_name':\texttt{ValueName},
  'value\_attr': \texttt{attr in ControlMessage},
  'condition\_attr',
  \texttt{optional: transformation: callable)
  \}
  
  that declare what values are needed to update
  
  *args –

  **kwargs –

- **property manager**

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

  **Returns** `pvp.alarm.alarm_manager.Alarm_Manager`

  ```python
  check(sensor_values) \rightarrow \text{bool}
  
  Every Condition subclass needs to define this method that accepts `SensorValues` and returns a boolean.

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

  **Returns** `bool`

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

  ```python
  class pvp.alarm.condition.ValueCondition
  
  \text{value\_name: pvp.common.values.ValueName,}
  \text{limit: (\texttt{<class 'int'>, <class 'float'>}), mode: str,}
  \text{*args, **kwargs}
  
  **Bases:** `pvp.alarm.condition.Condition`

  Value is greater or lesser than some max/min

  **Parameters**

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

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

**Type** callable

**Methods:**

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

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

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

**Attributes:**

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

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

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

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

**Type** callable

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

**Returns:**

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

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

**Returns** bool

**reset** ()
not stateful, do nothing.
class pvp.alarm.condition.CycleValueCondition(n_cycles: int, *args, **kwargs)
    Bases: pvp.alarm.condition.ValueCondition
    Value goes out of range for a specific number of breath cycles

    Parameters
    n_cycles (int) – number of cycles required

    _start_cycle
    The breath cycle where the
    Type int

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

    Parameters

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

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

    Methods:

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

    reset() Reset check status and start cycle

    Attributes:

    n_cycles Number of cycles required

    property n_cycles
    Number of cycles required

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

    Parameters () (sensor_values)–

    Returns bool

    reset ()
    Reset check status and start cycle

class pvp.alarm.condition.TimeValueCondition(time, *args, **kwargs)
    Bases: pvp.alarm.condition.ValueCondition
    value goes out of range for specific amount of time
Warning: Not implemented!

Parameters

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

Methods:

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

param time number of seconds value must be out of range
```

```python
check(sensor_values)

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

reset()

not stateful, do nothing.
```

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

Parameters

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

check (sensor_values)

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

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

Returns bool

reset ()

not stateful, do nothing.
```

```python

Bases: pvp.alarm.condition.Condition

Alarm is above or below a certain severity.

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

Parameters

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Alarm is above or below a certain severity.</td>
</tr>
<tr>
<td>check</td>
<td>Every Condition subclass needs to define this method that accepts <code>SensorValues</code> and returns a boolean.</td>
</tr>
<tr>
<td>reset</td>
<td>If a condition is stateful, need to provide some method of resetting the state.</td>
</tr>
</tbody>
</table>

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>‘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’.</td>
</tr>
</tbody>
</table>

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

Alarm is above or below a certain severity.

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

**Parameters**

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

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

```python
*args –
**kwargs –
```

**property mode**

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

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

**Returns** one of ‘min’, ‘equals’, or ‘max’.

**Return type** `str`

```python
check(sensor_values)
```

Every Condition subclass needs to define this method that accepts `SensorValues` and returns a boolean.
Parameters `sensor_values (SensorValues)` – SensorValues used to compute alarm status

Returns bool

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

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

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

```python
_start_cycle
The breath cycle where the
    Type int

_mid_check
whether a value has left the acceptable range and we are counting consecutive breath cycles
    Type bool
```
Alarm is above or below a certain severity.
Get alarm severity status from `Alarm_Manager.get_alarm_severity()`.

Parameters
- `alarm_type (AlarmType)` – Alarm type to check
- `severity (AlarmSeverity)` – Alarm severity to check against
- `mode (str)` – one of ‘min’, ‘equals’, or ‘max’. ‘min’ returns true if the alarm is at least this value (note the difference from ValueCondition which returns true if the alarm is less than..) and vice versa for ‘max’.

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

- *args –
- **kwargs –

Methods:

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

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

**Attributes:**

- `n_cycles`

- `property n_cycles`
check (sensor_values)
Every Condition subclass needs to define this method that accepts SensorValues and returns a boolean

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

Returns bool

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

14.3 Main Alarm Module

Data:

<table>
<thead>
<tr>
<th>ALARM_RULES</th>
<th>Definitions of all Alarm_Rule s used by the Alarm_Manager</th>
</tr>
</thead>
</table>

Classes:

Alarm(alarm_type, severity, start_time,...) Representation of alarm status and parameters
AlarmSeverity(value) An enumeration.
AlarmType(value) An enumeration.
Alarm_Manager() The Alarm Manager
Alarm_Rule(name, conditions[, latch, technical])

• name of rule

Enum(value) Generic enumeration.
IntEnum(value) Enum where members are also (and must be) ints
ValueName(value) Canonical names of all values used in PVP.
auto() Instances are replaced with an appropriate value in Enum class suites.
odict alias of collections.OrderedDict

class pvp.alarm.AlarmType(value) An enumeration.

Attributes:

LOW_PRESSURE
HIGH_PRESSURE
LOW_VTE
HIGH_VTE
LOW_PEEP
HIGH_PEEP
LOW_O2
HIGH_O2
OBSTRUCTION
LEAK
SENSORS_STUCK
BAD_SENSOR_READINGS

continues on next page
Table 26 – continued from previous page

<table>
<thead>
<tr>
<th>MISSED_HEARTBEAT</th>
<th>human_name</th>
<th>Replace .name underscores with spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW_PRESSURE</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>HIGH_PRESSURE</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOW_VTE</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>HIGH_VTE</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>LOW_PEEP</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>HIGH_PEEP</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>LOW_O2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>HIGH_O2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OBSTRUCTION</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>LEAK</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SENSORS_STUCK</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>BAD_SENSOR_READINGS</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>MISSED_HEARTBEAT</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

property human_name
Replace .name underscores with spaces

class pvp.alarm.AlarmSeverity(value)
An enumeration.

Attributes:

<table>
<thead>
<tr>
<th>HIGH</th>
<th>MEDIUM</th>
<th>LOW</th>
<th>OFF</th>
<th>TECHNICAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH = 3</td>
<td>MEDIUM = 2</td>
<td>LOW = 1</td>
<td>OFF = 0</td>
<td>TECHNICAL = -1</td>
</tr>
</tbody>
</table>

pvp.alarm.ALARM_RULES = OrderedDict([(AlarmType.LOW_PRESSURE: 1), ...
Definitions of all Alarm_Rules used by the Alarm_Manager

See definitions here
COORDINATOR MODULE

15.1 Submodules

15.2 coordinator

Classes:

- Alarm(alarm_type, severity, start_time, ...) Representation of alarm status and parameters
- ControlSetting(name, value, min_value, ...) Message containing ventilation control parameters.
- CoordinatorBase([sim_mode])
- CoordinatorLocal([sim_mode])
  - param sim_mode
- CoordinatorRemote([sim_mode])
- ProcessManager(sim_mode, startCommandLine, ...
- SensorValues([timestamp, loop_counter, ...]) Structured class for communicating sensor readings throughout PVP.
- ValueName(value) Canonical names of all values used in PVP.

Data:

- Dict The central part of internal API.
- List The central part of internal API.

Functions:

- get_coordinator([single_process, sim_mode])
- get_rpc_client()
- init_logger(module_name[, log_level, ...]) Initialize a logger for logging events.

```python
class pvp.coordinator.coordinator.CoordinatorBase(sim_mode=False)
  Bases: object
  Methods:
    get_alarms()
```

continues on next page
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) →
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 –
    .is_running when thread should stop
    Type threading.Event

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.

__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

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

```python
kill()
```
Stop the coordinator and end the whole program

```python
pvp.coordinator.coordinator.get_coordinator (single_process=False, sim_mode=False) -> pvp.coordinator.coordinator.CoordinatorBase
```

## 15.3 ipc

### Classes:

```python
SimpleXMLRPCServer(addr[, requestHandler, ...])   # Simple XML-RPC server.
```

### Functions:

- `get_alarms()`
- `get_control(control_setting_name)`
- `get_rpc_client()`
- `get_sensors()`
- `get_target_waveform()`
- `init_logger(module_name[, log_level, ...])`  # Initialize a logger for logging events.
- `rpc_server_main(sim_mode, serve_event[, ...])`
- `set_breath_detection(breath_detection)`
- `set_control(control_setting)`

```python
pvp.coordinator.rpc.get_sensors()  # Method
pvp.coordinator.rpc.get_alarms()  # Method
pvp.coordinator.rpc.set_control(control_setting)  # Method
pvp.coordinator.rpc.get_control(control_setting_name)  # Method
pvp.coordinator.rpc.set_breath_detection(breath_detection)  # Method
pvp.coordinator.rpc.get_target_waveform()  # Method
pvp.coordinator.rpc.rpc_server_main(sim_mode, serve_event, addr='localhost', port=9533)  # Method
pvp.coordinator.rpc.get_rpc_client()  # Method
```

## 15.4 process_manager

### Classes:

```python
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

SIXTEEN

REQUIREMENTS
17.1 Manuals

- Hamilton T1 Quick Guide

17.2 Other Reference Material

- Hamilton UI Simulator
19.1 Version 0.0

19.1.1 v0.0.2 (April xxth, 2020)

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

19.1.2 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

19.1.3 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.
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.

### 21.1 Local Build

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

Documentation will be generated into `docs`.

---

**Advertisement :)**

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

You will like those projects!
22.1 h2 Heading

22.1.1 h3 Heading

22.1.1.1 h4 Heading

h5 Heading

h6 Heading

22.2 Horizontal Rules

22.3 Emphasis

This is bold text

This is italic text

22.4 Blockquotes

Blockquotes can also be nested...

...by using additional greater-than signs right next to each other...

...or with spaces between arrows.
22.5 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.

22.6 Code

Inline code

Indented code

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

Block code “fences”

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

Syntax highlighting

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

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

link text

link with title
22.8 Images

Minion
Like links, Images also have a footnote style syntax

Alt text

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

- genindex
- modindex
p
pvp.alarm, 122
pvp.alarm.alarm, 110
pvp.alarm.alarm_manager, 105
pvp.alarm.condition, 115
pvp.alarm.rule, 113
pvp.common.fashion, 99
pvp.common.loggers, 91
pvp.common.message, 88
pvp.common.prefs, 95
pvp.common.unit_conversion, 98
pvp.common.utils, 99
pvp.common.values, 83
pvp.controller.control_module, 73
pvp.coordinator.coordinator, 125
pvp.coordinator.process_manager, 128
pvp.coordinator.rpc, 128
pvp.gui.main, 37
pvp.gui.styles, 69
pvp.gui.widgets.alarm_bar, 50
pvp.gui.widgets.components, 64
pvp.gui.widgets.control_panel, 45
pvp.gui.widgets.dialog, 68
pvp.gui.widgets.display, 55
pvp.gui.widgets.plot, 60
pvp.io, 103
pvp.io.hal, 101
## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleBase</td>
<td>77</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleDevice</td>
<td>79</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.coordinator.coordinator.CoordinatorBase</td>
<td>126</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.controller.control_module.ControlModuleSimulator</td>
<td>81</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.components.OnOffButton</td>
<td>67</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.gui.widgets.control_panel.StopWatch</td>
<td>49</td>
</tr>
<tr>
<td>__BACK_CMD()</td>
<td>Method of pvp.io.hal.Hal</td>
<td>102</td>
</tr>
</tbody>
</table>
_initialize_set_to_COPY()
  (pvp.controller.control_module.ControlModuleBase.style
method), 76
_instance(pvp.alarm.alarm_manager.Alarm_Manager _update_time()
  (pvp.gui.widgets.control_panel.StopWatch
method), 50
_is_running(pvp.coordinator.coordinator.CoordinatorLocal.value_changed() (pvp.gui.widgets.display.Display
method), 58
_last_heartbeat(pvp.gui.widgets.control_panel.HeartBeat
attribute), 48
_maximum() (pvp.gui.widgets.components.DoubleSlider
method), 65
_mid_check (pvp.alarm.condition.CycleAlarmSeverityCondition
attribute), 121
_minimum() (pvp.gui.widgets.components.DoubleSlider
method), 65
_open_logfile() (pvp.common.loggers.DataLogger
method), 93
_pressure_units_changed()
  (pvp.gui.widgets.control_panel.Control_Panel
method), 46
_reset() (pvp.controller.control_module.Balloon_Simulator
method), 80
_screenshot () (pvp.main.PVP_Gui method), 44
_sensor_to_COPY()
  (pvp.controller.control_module.ControlModuleBase
method), 76
_sensor_to_COPY()
  (pvp.controller.control_module.ControlModuleDevice
method), 79
_sensor_to_COPY()
  (pvp.controller.control_module.ControlModuleSimulator
method), 81
_set_HAL() (pvp.controller.control_module.ControlModuleDevice
method), 79
_set_cycle_control()
  (pvp.main.PVP_Gui attribute), 38
_singleStep() (pvp.widgets.components.DoubleSlider
method), 65
_start_cycle
  (pvp.alarm.condition.CycleAlarmSeverityCondition
attribute), 121
_start_cycle
  (pvp.alarm.condition.CycleValueCondition
attribute), 118
_start_mainloop()
  (pvp.controller.control_module.ControlModuleBase
method), 78
_start_mainloop()
  (pvp.controller.control_module.ControlModuleDevice
method), 79
_start_mainloop()
  (pvp.controller.control_module.ControlModuleSimulator
method), 81
_state
  (pvp.gui.widgets.control_panel.HeartBeat at-
tribute), 47
| B | aux_pressure() | (pvp.io.health.Hal property), 103 |
| BAD_SENSOR_READINGS | (pvp.alarm.AlarmType attribute), 123 |
| Balloon_Simulator | | (class in pvp.controller.control_module), 79 |
| beatheart() | | (class in pvp.controller.control_module), 49 |
| BREATHS_PER_MINUTE | (pvp.common.values.ValueName attribute), 84 |
| C | callbacks | (pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108 |
| check() | (pvp.alarm.condition.AlarmSeverityCondition method), 120 |
| check() | (pvp.alarm.condition.Condition method), 116 |
| check() | (pvp.alarm.condition.CycleAlarmSeverityCondition method), 121 |
| check() | (pvp.alarm.condition.CycleValueCondition method), 118 |
| check() | (pvp.alarm.condition.TimeValueCondition method), 119 |
| check() | (pvp.alarm.condition.ValueCondition method), 117 |
| check_files() | (pvp.common.loggers.DataLogger method), 94 |
| check_rule() | (pvp.alarm.alarm_manager.Alarm_Manager method), 108 |
| clear_alarm() | (pvp.gui.widgets.alarm_bar.Alarm_Bar method), 51 |
| clear_all_alarms() | (pvp.alarm.alarm_manager.Alarm_Manager method), 110 |
| cleared_alarms | (pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108 |
| close_button | (pvp.gui.widgets.alarm_bar.Alarm_Card attribute), 52 |
| close_logfile() | (pvp.common.loggers.DataLogger method), 93 |
| closeEvent() | (pvp.gui.main.PVP_Gui method), 42 |
| cmH2O_to_hPa() | (in module pvp.common.unit_conversion), 98 |
| Condition | (class in pvp.alarm.condition), 115 |
| CONTROL | (in module pvp.common.values), 87 |
| CONTROL | (pvp.gui.main.PVP_Gui attribute), 40 |
| control | (pvp.gui.widgets.display.Display.self attribute), 56 |
| control() | (pvp.common.values.Value property), 87 |
| Control_Panel | | (class in pvp.gui.widgets.control_panel), 45 |
| control_type() | (pvp.common.values.Value property), 87 |
| control_width | (pvp.gui.main.PVP_Gui attribute), 41 |
| ControlModuleBase | (class in pvp.controller.control_module), 74 |
| ControlModuleDevice | (class in pvp.controller.control_module), 78 |
| ControlModuleSimulator | (class in pvp.controller.control_module), 80 |
| controls | (pvp.gui.main.PVP_Gui attribute), 38 |
| controls_set() | (pvp.gui.main.PVP_Gui property), 44 |
| ControlSetting | (class in pvp.common.message), 89 |
| ControlValues | (class in pvp.common.message), 90 |
| coordinator | (pvp.gui.main.PVP_Gui attribute), 38 |
| CoordinatorBase | (class in pvp.coordinator.coordinator), 125 |
| CoordinatorLocal | (class in pvp.coordinator.coordinator), 126 |
| CoordinatorRemote | (class in pvp.coordinator.coordinator), 127 |
| create_signals() | (pvp.gui.widgets.components.EditableLabel method), 66 |
| cycle_autoset_changed | (pvp.gui.widgets.control_panel.Control_Panel attribute), 46 |
| CycleAlarmSeverityCondition | (class in pvp.alarm.condition), 121 |
| cycles | (pvp.gui.widgets.plot.Plot attribute), 61 |
| CycleValueCondition | (class in pvp.alarm.condition), 117 |
| D | DataLogger | (class in pvp.common.loggers), 92 |
| deactivate() | (pvp.alarm.alarm.Alarm method), 112 |
| deactivate_alarm() | (pvp.alarm.alarm_manager.Alarm_Manager method), 109 |
| decimals | (pvp.gui.widgets.display.Display.self attribute), 56 |
| decimals() | (pvp.common.values.Value property), 87 |
| default() | (pvp.common.values.Value property), 87 |
| dependencies | (pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108 |
| depends() | (pvp.alarm.rule.Alarm_Rule property), 114 |
| depends_callbacks | (pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108 |
| DerivedValues | (class in pvp.common.message), 91 |
| dismiss_alarm() | (pvp.alarm.alarm_manager.Alarm_Manager method), 109 |
| Display | (class in pvp.gui.widgets.display), 55 |
| display() | (pvp.common.values.Value property), 87 |
heartbeat (pvp.gui.widgets.control_panel.Control_Panel.init_ui_controls () (pvp.gui.main.PVP_Gui method), 41
 heartbeat (pvp.gui.widgets.control_panel.HeartBeat attribute), 48
 heartbeat () (pvp.coordinator.process_manager.ProcessManager method), 129
 HIGH (pvp.alarm.AlarmSeverity attribute), 123
 HIGH_O2 (pvp.alarm.AlarmType attribute), 123
 HIGH_PEEP (pvp.alarm.AlarmType attribute), 123
 HIGH_PRESSURE (pvp.alarm.AlarmType attribute), 123
 history (pvp.gui.widgets.plot.Plot attribute), 61
 hPa_to_cmH2O () (in module pvp.common.unit_conversion), 98
 human_name () (pvp.alarm.AlarmType property), 123
 icons (pvp.gui.widgets.alarm_bar.Alarm_Bar attribute), 51
 id (pvp.alarm.alarm.Alarm attribute), 110, 111
 id_counter (pvp.alarm.alarm.Alarm attribute), 111
 idx (pvp.gui.widgets.alarm_bar.Alarm_Sound_Player attribute), 53
 IE_RATIO (pvp.common.values.ValueName attribute), 84
 increment_delay (pvp.gui.widgets.alarm_bar.Alarm_Sound_Player attribute), 53
 increment_level () (pvp.gui.widgets.alarm_bar.Alarm_Sound_Player method), 54
 init () (in module pvp.common.prefs), 97
 init_audio () (pvp.gui.widgets.alarm_bar.Alarm_Sound_Player method), 54
 init_controls () (pvp.gui.main.PVP_Gui method), 44
 init_logger () (in module pvp.common.loggers), 92
 init_ui () (pvp.gui.main.PVP_Gui method), 41
 init_ui () (pvp.gui.widgets.alarm_bar.Alarm_Bar method), 51
 init_ui () (pvp.gui.widgets.alarm_bar.Alarm_Card method), 52
 init_ui () (pvp.gui.widgets.control_panel.Control_Panel method), 46
 init_ui () (pvp.gui.widgets.control_panel.HeartBeat method), 48
 init_ui () (pvp.gui.widgets.control_panel.StopWatch method), 50
 init_ui () (pvp.gui.widgets.display.Display method), 57
 init_ui () (pvp.gui.widgets.display.Limits_Plot method), 60
 init_ui () (pvp.gui.widgets.plot.Plot method), 63
 init_ui_labels () (pvp.gui.widgets.display.Display method), 58
 init_ui_layout () (pvp.gui.widgets.display.Display method), 58
 init_ui_limits () (pvp.gui.widgets.display.Display method), 58
 init_ui_monitor () (pvp.gui.main.PVP_Gui method), 41
 init_ui_plots () (pvp.gui.main.PVP_Gui method), 41
 init_ui_record () (pvp.gui.widgets.display.Display method), 58
 init_ui_signals () (pvp.gui.widgets.display.Display method), 58
 init_ui_slider () (pvp.gui.widgets.display.Display method), 58
 init_ui_status_bar () (pvp.gui.main.PVP_Gui method), 41
 init_ui_toggle_button () (pvp.gui.widgets.display.Display method), 58
 INSPIRATION_TIME_SEC (pvp.common.values.ValueName attribute), 84
 interrupt () (pvp.coordinator.coordinator.ControlModuleBase method), 78
 is_running () (pvp.coordinator.coordinator.ControlModuleBase method), 78
 is_running () (pvp.coordinator.coordinator.CoordinatorLocal method), 126
 is_running () (pvp.coordinator.coordinator.CoordinatorRemote method), 127
 is_set () (pvp.gui.widgets.display.Display property), 59
 INTERRUPT

KeyPressHandler (class in pvp.gui.widgets.components), 65
 kill () (pvp.coordinator.coordinator.CoordinatorRemote method), 128

Index 157
(pvp.gui.widgets.components.EditableLabel method), 66
labelUpdatedAction()
(pvp.gui.widgets.components.EditableLabel method), 66
launch_gui() (in module pvp.gui.main), 44
LEAK (pvp.alarm.AlarmType attribute), 123
limits_changed (pvp.gui.widgets.plot.Plot attribute), 62
Limits_Plot (class in pvp.gui.widgets.display), 59
limits_updated()
(pvp.gui.main.PVP_Gui method), 42
load_file()
(pvp.common.loggers.DataLogger method), 94
load_pixmaps()
(pvp.gui.widgets.control_panel.Lock_Button method), 47
load_pixmaps()
(pvp.gui.widgets.control_panel.Start_Button method), 46
load_prefs() (in module pvp.common_prefs), 97
load_rule()
(pvp.alarm.alarm_manager.Alarm_Manager method), 108
load_rules()
(pvp.alarm.alarm_manager.Alarm_Manager method), 108
load_state()
(pvp.gui.main.PVP_Gui method), 43
LOADED (in module pvp.common_prefs), 96
Lock_Button (class in pvp.gui.widgets.control_panel), 47
lock_button
(pvp.gui.widgets.control_panel.Control_Panel attribute), 45
locked
(pvp.gui.main.PVP_Gui attribute), 38
locked()
(in module pvp.common_fashion), 99
log2csv()
(pvp.common.loggers.DataLogger method), 94
log2mat()
(pvp.common.loggers.DataLogger method), 94
logged_alarms
(pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108
logger
(pvp.alarm.alarm_manager.Alarm_Manager attribute), 108
logger
(pvp.gui.main.PVP_Gui attribute), 39
LOW (pvp.alarm.AlarmSeverity attribute), 123
LOW_02 (pvp.alarm.AlarmType attribute), 123
LOW_PEEP (pvp.alarm.AlarmType attribute), 123
LOW_PRESSURE (pvp.alarm.AlarmType attribute), 123
LOW_VTE (pvp.alarm.AlarmType attribute), 123
M
make_dirs() (in module pvp.common_prefs), 97
make_icons()
(pvp.gui.widgets.alarm_bar.Alarm_Bar method), 51
manager()
(pvp.alarm.condition.Condition property), 116
maximum()
(pvp.gui.widgets.components.DoubleSlider method), 65
MEDIUM (pvp.alarm.AlarmSeverity attribute), 123
minimum()
(pvp.gui.widgets.components.DoubleSlider method), 65
MISSED_HEARTBEAT (pvp.alarm.AlarmType attribute), 123
mode()
(pvp.alarm.condition.AlarmSeverityCondition property), 120
mode()
(pvp.alarm.condition.ValueCondition property), 117
module
pvp.alarm, 122
pvp.alarm.alarm, 110
pvp.alarm.alarm_manager, 105
pvp.alarm.condition, 115
pvp.common.fashion, 99
pvp.common.loggers, 91
pvp.common.message, 88
pvp.commonprefs, 95
pvp.common.unit_conversion, 98
pvp.common.utils, 99
pvp.common.values, 83
pvp.controller.control_module, 73
pvp.coordinator.coordinator, 125
pvp.coordinator.process_manager, 128
pvp.coordinator.rpc, 128
pvp.gui.main, 37
pvp.gui.styles, 69
pvp.gui.widgets.alarm_bar, 50
pvp.gui.widgets.components, 64
pvp.gui.widgets.control_panel, 45
pvp.gui.widgets.dialog, 68
pvp.gui.widgets.display, 55
pvp.gui.widgets.plot, 60
pvp.io, 103
pvp.io_hal, 101
N
n_cycles()
(pvp.alarm.condition.CycleAlarmSeverityCondition property), 121
n_cycles()
(pvp.alarm.condition.CycleValueCondition property), 118
name
(pvp.gui.widgets.display.Display.self attribute), 56
name()
(pvp.common.values.Value property), 86
O
OBSTRUCTION (pvp.alarm.AlarmType attribute), 123
OFF (pvp.alarm.AlarmSeverity attribute), 123
Q
QHLine (class in pvp.gui.widgets.components), 66
QVLine (class in pvp.gui.widgets.components), 67
R
redraw() (pvp.gui.widgets.display.Display method), 58
register_alarm() (pvp.alarm.alarm_manager.Alarm_Manager method), 109
register_dependency()
(pvp.alarm.alarm_manager.Alarm_Manager method), 109
reset() (pvp.alarm.condition.AlarmSeverityCondition method), 116
reset() (pvp.alarm.condition.Condition method), 116
reset() (pvp.alarm.condition.CycleAlarmSeverityCondition method), 122
reset() (pvp.alarm.condition.CycleValueCondition method), 118
reset() (pvp.alarm.condition.TimeValueCondition method), 119
reset() (pvp.alarm.condition.ValueCondition method), 117
reset() (pvp.arm.rule.Alarm_Rule method), 114
reset_start_time() (pvp.gui.widgets.plot.Plot method), 62
reset_start_time() (pvp.gui.widgets.plot.Plot_Container method), 64
restart_process() (pvp.coordinator.process_manager.ProcessManager method), 129
returnPressed (pvp.gui.widgets.components.KeyPressHandler attribute), 66
returnPressedAction()
(pvp.gui.widgets.components.EditableLabel method), 66
rotation_newfile() (pvp.common.loggers.DataLogger method), 94
rounded_string() (in module pvp.common.unit_conversion), 98
rpc_server_main() (in module pvp.coordinator.rpc), 128
rules (pvp.alarm.alarm_manager.Alarm_Manager attribute), 106, 108
running (pvp.gui.main.PVP_Gui attribute), 38
runtime (pvp.gui.widgets.control_panel.Control_Panel attribute), 45
S
safe_range() (pvp.common.values.Value property), 86
save_prefs() (in module pvp.common.prefs), 97
save_state() (pvp.gui.main.PVP_Gui method), 43
SENSOR (in module pvp.common.values), 87
sensor() (pvp.common.values.Value property), 87
sensor_value (pvp.gui.widgets.display.Display.self attribute), 56
sensor_value (pvp.gui.widgets.display.Limits_Plot attribute), 59
SENSORS_STUCK (pvp.alarm.AlarmType attribute), 123
SensorValues (class in pvp.common.message), 88
set_breath_detection() (in module pvp.coordinator.rpc), 128
set_breath_detection() (pvp.controller.control_module.ControlModuleBase method), 76
set_breath_detection() (pvp.controller.coordinator.CoordinatorBase method), 126
set_breath_detection() (pvp.controller.coordinator.CoordinatorLocal method), 127
set_breath_detection() (pvp.controller.coordinator.CoordinatorRemote method), 127
set_breath_detection() (pvp.gui.main.PVP_Gui method), 43
set_control() (in module pvp.coordinator.rpc), 128
set_control() (pvp.controller.control_module.ControlModuleBase method), 76
set_control() (pvp.coordinator.coordinator.CoordinatorLocal method), 127
set_control() (pvp.coordinator.coordinator.CoordinatorRemote method), 127
set_duration() (pvp.gui.widgets.plot.Plot method), 62
set_duration() (pvp.gui.widgets.plot.Plot_Container method), 64
set_icon() (pvp.gui.widgets.alarm_bar.Alarm_Bar method), 52
set_indicator() (pvp.gui.widgets.control_panel.HeartBeat method), 49
set_locked() (pvp.gui.widgets.display.Display method), 59
set_mute() (pvp.gui.widgets.alarm_bar.Alarm_Sound_Player method), 52
staticMetaObject (pvp.gui.widgets.components.EditableLabel attribute), 66
staticMetaObject (pvp.gui.widgets.components.KeyPressHandler attribute), 66
staticMetaObject (pvp.gui.widgets.components.OnOffButton attribute), 68
staticMetaObject (pvp.gui.widgets.components.QHLine attribute), 67
staticMetaObject (pvp.gui.widgets.components.QVLine attribute), 67
staticMetaObject (pvp.gui.widgets.control_panel.Control_Panel attribute), 46
staticMetaObject (pvp.gui.widgets.control_panel.HeartBeat attribute), 49
staticMetaObject (pvp.gui.widgets.control_panel.Lock_Button attribute), 47
staticMetaObject (pvp.gui.widgets.control_panel.Start_Button attribute), 47
staticMetaObject (pvp.gui.widgets.control_panel.StopWatch attribute), 50
staticMetaObject (pvp.gui.widgets.display.Display attribute), 59
staticMetaObject (pvp.gui.widgets.display.Limits_Plot attribute), 60
staticMetaObject (pvp.gui.widgets.display.Limits_Plot attribute), 62
staticMetaObject (pvp.gui.widgets.plot.Plot attribute), 64
staticMetaObject (pvp.gui.widgets.plot.Plot_Container attribute), 64
StopWatch (class in pvp.gui.widgets.control_panel), 49
StopWatch (class in pvp.gui.widgets.control_panel), 49
store_control_command () (pvp.common.loggers.DataLogger method), 93
store_control_data () (pvp.common.loggers.DataLogger method), 93
store_waveform_data () (pvp.common.loggers.DataLogger method), 93

TECHNICAL (pvp.alarm.AlarmSeverity attribute), 123
text () (pvp.gui.widgets.components.EditableLabel method), 66

U
units (pvp.gui.widgets.display.Display.self attribute), 56
update () (pvp.alarm.alarm_manager.Alarm_Manager method), 108
update () (pvp.controller.control_module.Balloon_Simulator method), 80
update_dependencies () (pvp.alarm.alarm_manager.Alarm_Manager method), 109
update_gui () (pvp.gui.main.PVP_Gui method), 41
update_icon () (pvp.gui.widgets.alarm_bar.Alarm_Bar method), 52
update_interval (pvp.gui.widgets.control_panel.HeartBeat attribute), 48
update_limits () (pvp.gui.widgets.display.Display method), 58
update_logger_sizes () (in module pvp.common.loggers), 92
update_period (pvp.gui.main.PVP_Gui attribute), 39
update_period (pvp.gui.widgets.display.Display.self attribute), 56
update_period() (pvp.gui.main.PVP_Gui property), 44
update_sensor_value() (pvp.gui.widgets.display.Display method), 58
update_set_value() (pvp.gui.widgets.display.Display method), 58
update_state() (pvp.gui.main.PVP_Gui method), 43
update_value() (pvp.gui.widgets.display.Limits_Plot method), 60
update_value() (pvp.gui.widgets.plot.Plot method), 62
update_value() (pvp.gui.widgets.plot.Plot_Container method), 63
update_yrange() (pvp.gui.widgets.display.Limits_Plot method), 60

V
Value (class in pvp.common.values), 84
value() (pvp.gui.widgets.components.DoubleSlider method), 65
value_changed (pvp.gui.widgets.display.Display attribute), 57
value_names() (pvp.alarm.rule.Alarm_Rule property), 114
ValueCondition (class in pvp.alarm.condition), 116
ValueName (class in pvp.common.values), 83
VALUES (in module pvp.common.values), 87
VTE (pvp.common.values.ValueName attribute), 84