
PVP

Release 0.2.0

jonny saunders et al

Dec 22, 2021

OVERVIEW

1	Software	3
1.1	PVP Modules	4
1.1.1	System Overview	4
1.1.1.1	Hardware	4
1.1.1.2	Software	4
1.1.2	Performance	5
1.1.2.1	ISO Standards Testing	6
1.1.2.2	Breath Detection	9
1.1.2.3	High Pressure Detection	10
1.1.3	Medical Disclaimer	10
1.1.4	Funding and Support	11
1.1.5	Hardware Overview	11
1.1.6	Components	11
1.1.6.1	Hardware Design	11
1.1.6.2	Actuator Selection	13
1.1.6.3	Sensor Selection	13
1.1.7	Assembly	14
1.1.7.1	Part 1. 3D Printed Components and Enclosure	15
1.1.7.2	Part 2. Basic Hardware Assembly	24
1.1.7.3	Part 3. Electronics Assembly	59
1.1.7.4	Part 4. Putting it all together	86
1.1.8	Electronics	96
1.1.8.1	Power and I/O	98
1.1.8.2	Sensor PCB	98
1.1.8.3	Actuator PCB	100
1.1.9	Bill of Materials	101
1.1.10	CAD	103
1.1.10.1	3D Printed Parts	103
1.1.10.2	Enclosure	105
1.1.11	Software Overview	105
1.1.12	Folder Structure	106
1.1.12.1	PVP Modules	106
1.1.13	GUI	106
1.1.13.1	Main GUI Module	106
1.1.13.2	GUI Widgets	106
1.1.13.3	GUI Stylesheets	106
1.1.13.4	Module Overview	108
1.1.13.5	Screenshot	108
1.1.14	Controller	108
1.1.14.1	Purpose of the Controller	108

1.1.14.2	Architecture of the Controller	109
1.1.15	common module	118
1.1.15.1	Values	118
1.1.15.2	Message	124
1.1.15.3	Loggers	127
1.1.15.4	Prefs	130
1.1.15.5	Unit Conversion	133
1.1.15.6	utils	134
1.1.15.7	fashion	135
1.1.16	pvp.io package	135
1.1.16.1	pvp.io.hal module	135
1.1.16.2	devices	137
1.1.17	Alarm	137
1.1.17.1	Alarm System Overview	137
1.1.17.2	Alarm Modules	137
1.1.17.3	Main Alarm Module	154
1.1.18	coordinator module	156
1.1.18.1	Submodules	156
1.1.18.2	coordinator	156
1.1.18.3	ipc	159
1.1.18.4	process_manager	160
1.1.19	Index	160
2	Medical Disclaimer	161
	Python Module Index	163
	Index	165

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.

[See our medRxiv preprint here!](#)

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

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

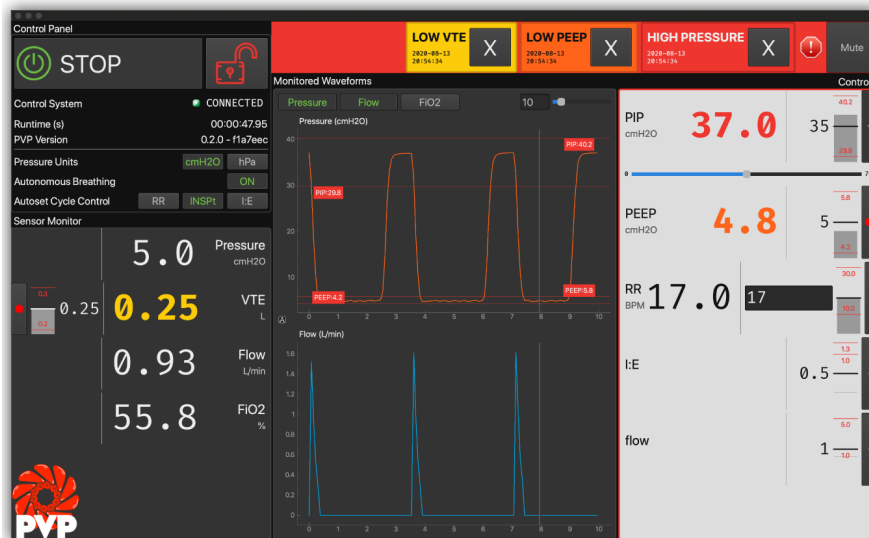
SOFTWARE

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



Multiple Control

Control ventilation and set alarm thresholds with a mouse, keyboard, or from sensor values

Monitor Limits

Sensor monitors, alarm limits, and alarm states are represented together in multiple modalities

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

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

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

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

1.1 PVP Modules

1.1.1 System Overview

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.

1.1.1.1 Hardware

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.

PVP Hardware

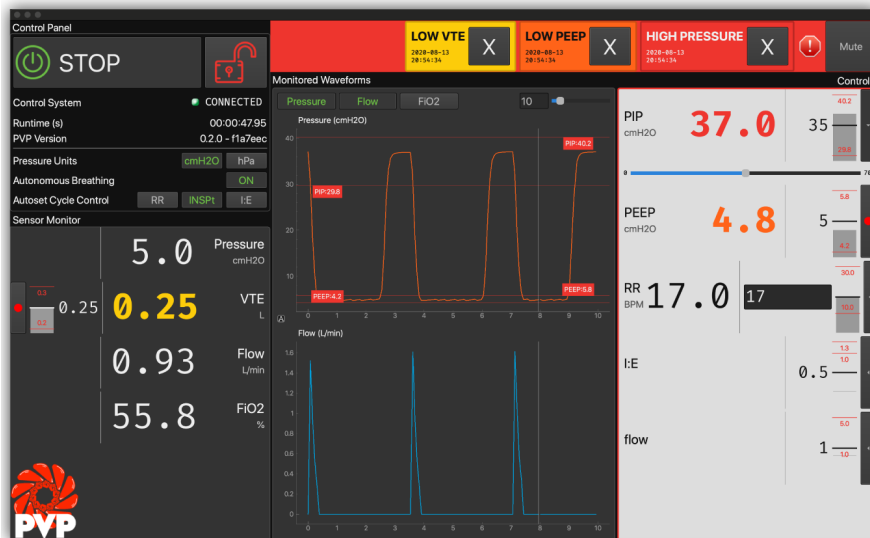
1.1.1.2 Software

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



Multiple Control

Control ventilation and set alarm thresholds with a mouse, keyboard, or from sensor values

Monitor Limits

Sensor monitors, alarm limits, and alarm states are represented together in multiple modalities

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.

PVP Modules

1.1.2 Performance

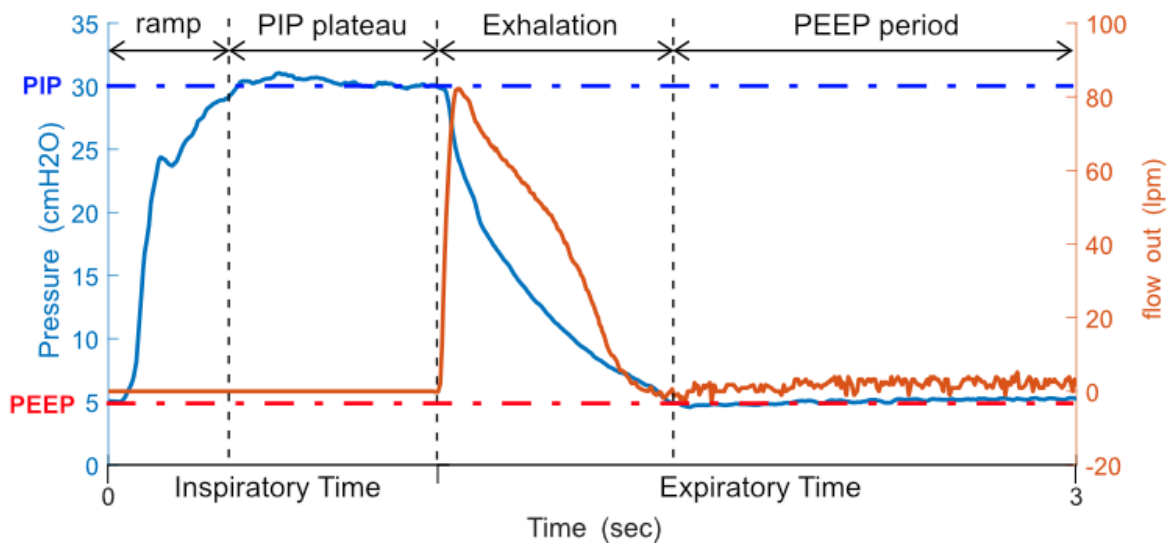


Fig. 1: Representative pressure control breath cycle waveforms for airway pressure and flow out. Test settings: compliance $C=20$ mL/cm H₂O, airway resistance $R=20$ cm H₂O/L/s, PIP=30 cm H₂O, PEEP=5 cm H₂O.

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). Once 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.

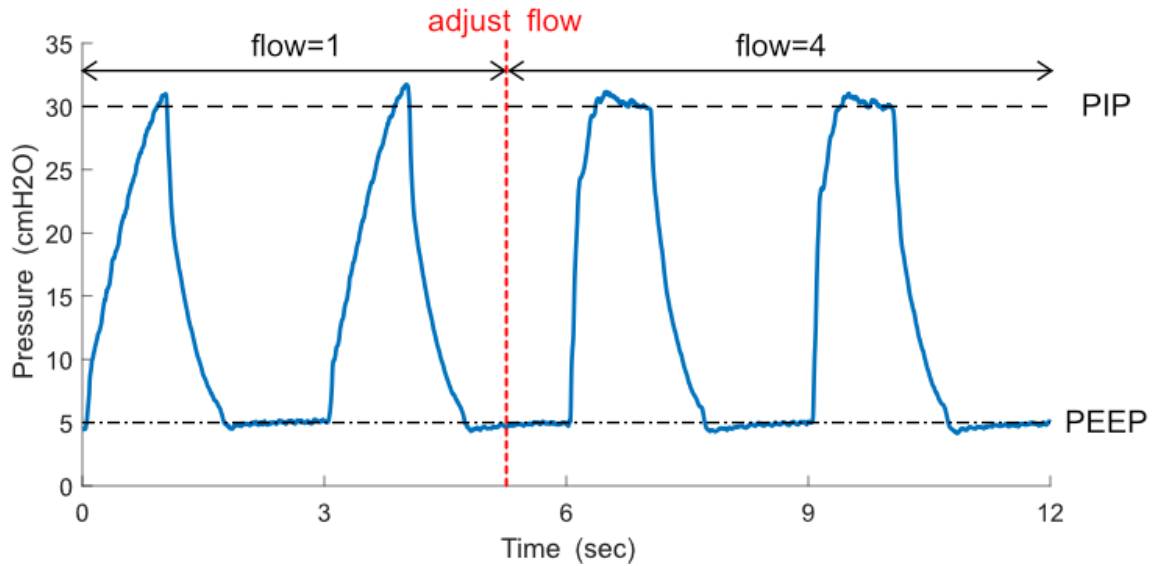


Fig. 2: Demonstration of waveform tuning via flow adjustment. If desired, the operator can increase the flow setting through the system GUI to decrease the pressure ramp time. Test settings: compliance $C=20$ mL/cm H₂O, airway resistance $R=20$ cm H₂O/L/s, PIP=30 cm H₂O, PEEP=5 cm H₂O.

1.1.2.1 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.

Table 1: Standard test battery from Table 201.105 in ISO 80601-2-80-2018 for pressure controlled ventilators

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

These tests cover an array of conditions, and more difficult test cases involve a high airway pressure coupled with a

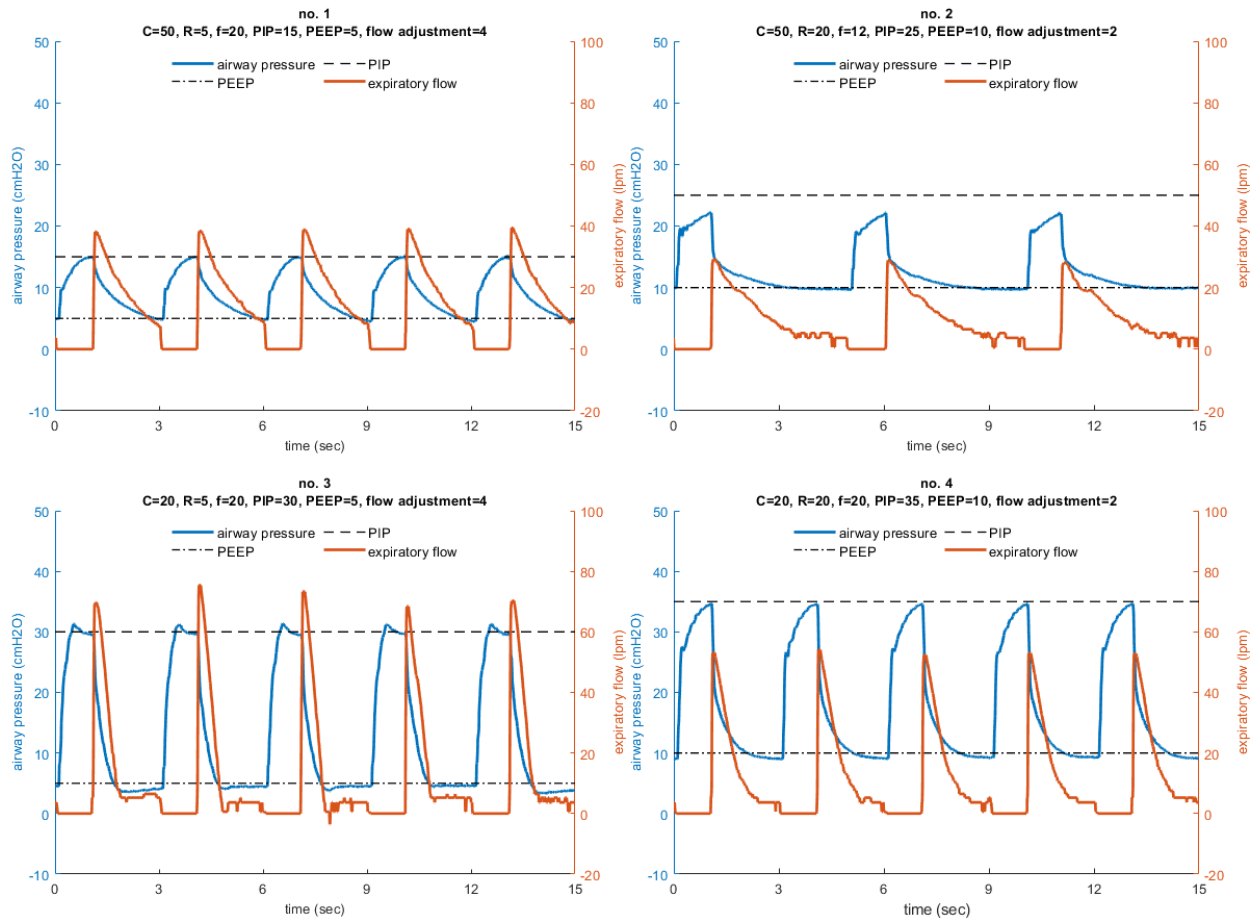


Fig. 3: Performance results of the ISO 80601-2-80-2018 pressure controlled ventilator standard tests with an intended delivered tidal volume of 500 mL. For each configuration the following parameters are listed: the test number (from table 201.105 in the ISO standard), the compliance (C, mL/cm H₂O), 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. PIP is reached in every test condition except for case 2, which is approximately 2.4 cm H₂O below the set point.

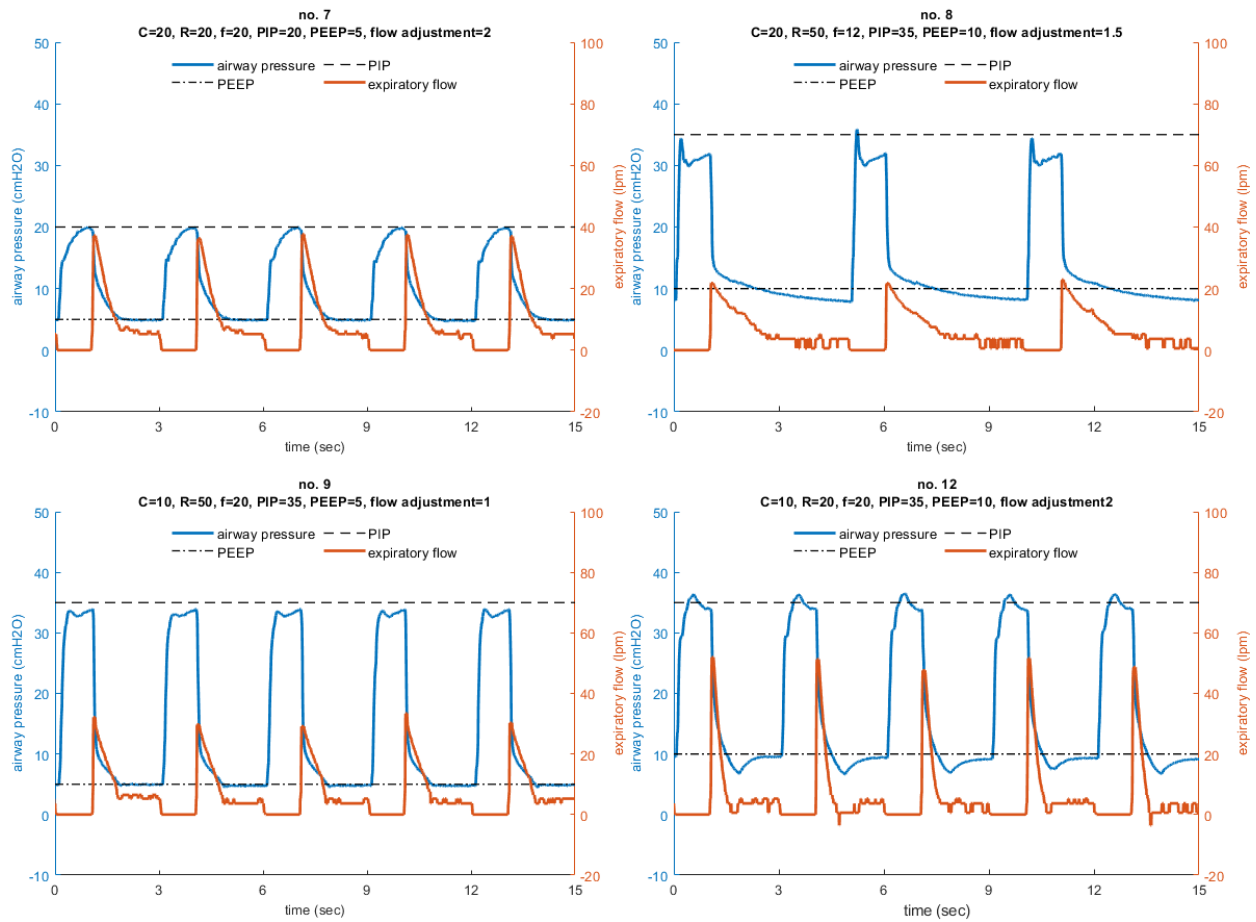


Fig. 4: Performance results of the ISO 80601-2-80-2018 pressure controlled ventilator standard tests with an intended delivered tidal volume of 300 mL. For each configuration the following parameters are listed: the test number (from table 201.105 in the ISO standard), the compliance (C, mL/cm H₂O), linear resistance (R, cm H₂O/L/s), respiratory frequency (breaths/min), peak inspiratory pressure (PIP, cm H₂O), positive end-expiratory pressure (PEEP, cm H₂O), and flow adjustment setting. PIP is reached in every test condition.

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.

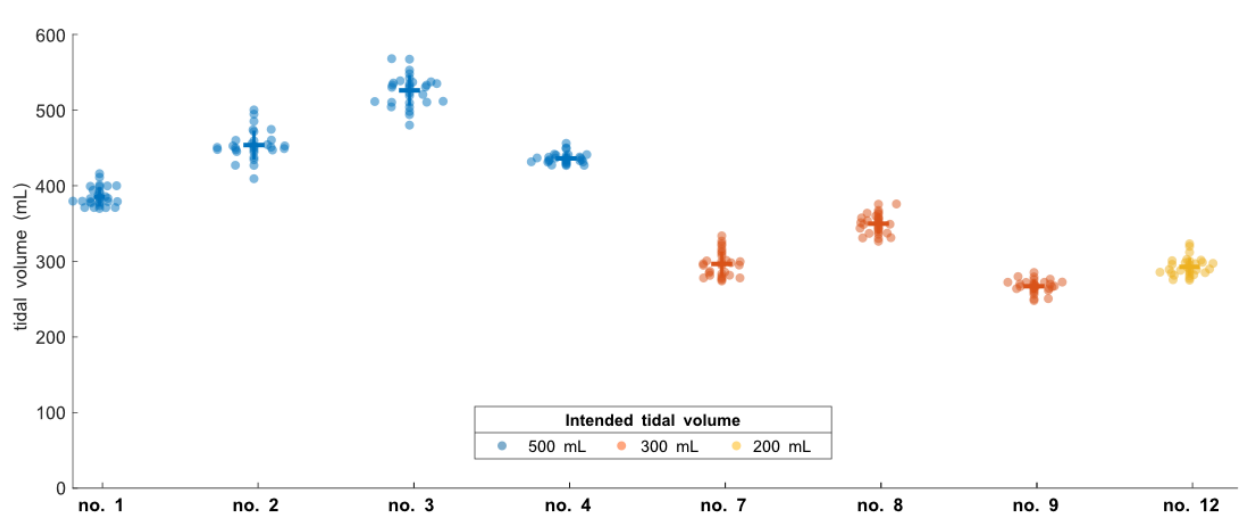


Fig. 5: Tidal volume performance for the ISO 80601-2-80-2018 pressure controlled ventilator standard tests, averaged across 30 breath cycles for each condition.

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.

1.1.2.2 Breath Detection

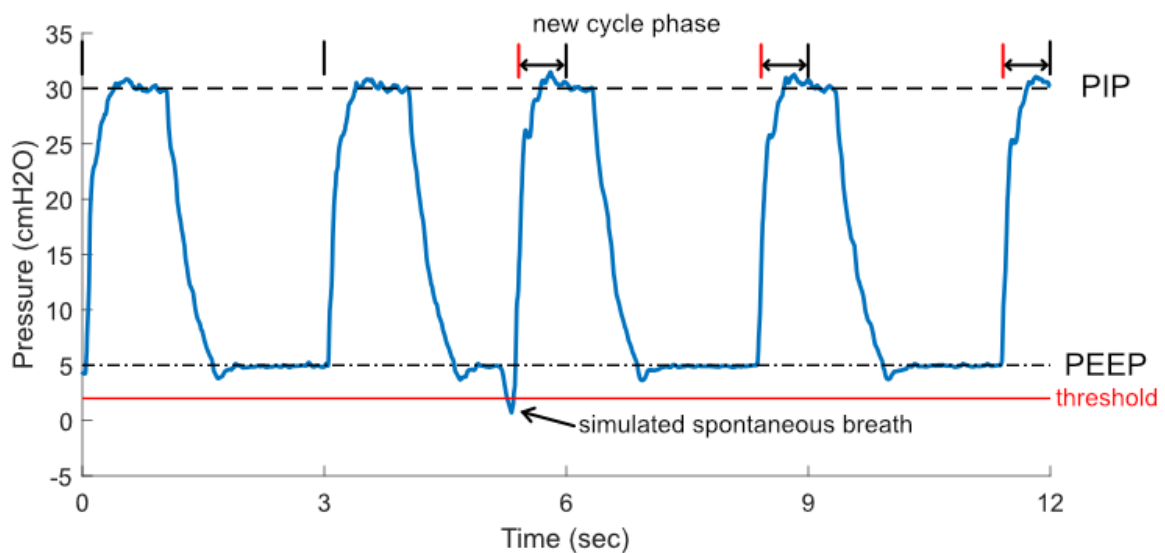


Fig. 6: Spontaneous breath detection.

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.

1.1.2.3 High Pressure Detection

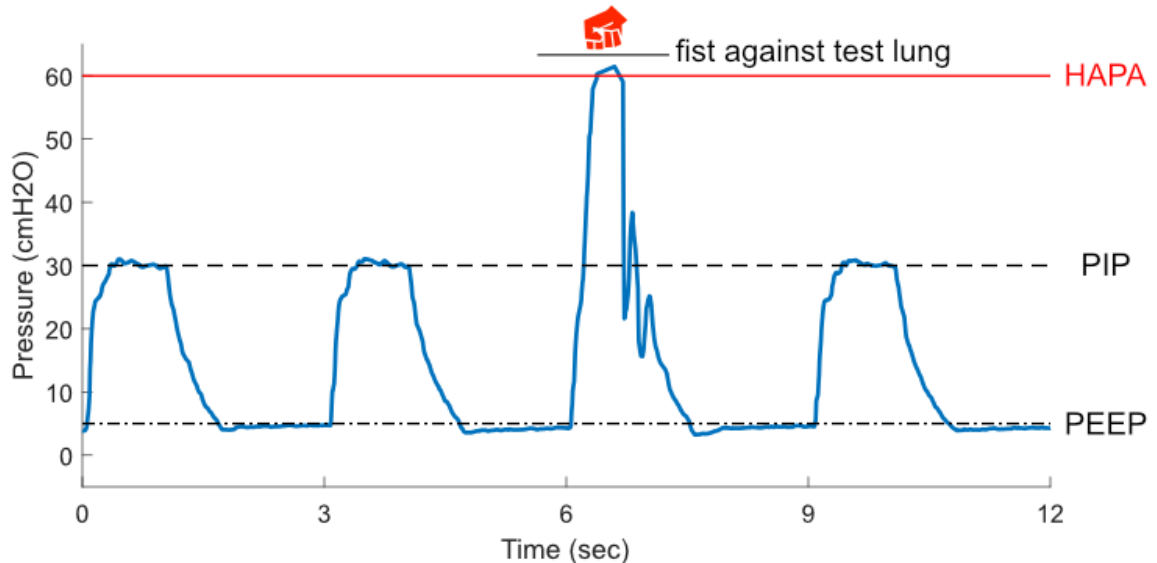


Fig. 7: High pressure alarm demonstration.

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.

1.1.3 Medical Disclaimer

PVP1 is not a regulated or clinically validated medical device. We have not yet performed testing for safety or efficacy on living organisms. All material described herein should be used at your own risk and do not represent a medical recommendation. PVP1 is currently recommended only for research purposes.

This website is not connected to, endorsed by, or representative of the view of Princeton University. Neither the authors nor Princeton University assume any liability or responsibility for any consequences, damages, or loss caused or alleged to be caused directly or indirectly for any action or inaction taken based on or made in reliance on the information or material discussed herein or linked to from this website.

PVP1 is under continuous development and the information here may not be up to date, nor is any guarantee made as such. Neither the authors nor Princeton University are liable for any damage or loss related to the accuracy, completeness or timeliness of any information described or linked to from this website.

By continuing to watch or read this, you are acknowledging and accepting this disclaimer.

1.1.4 Funding and Support

Funding and lab space for this project was provided by Princeton University in direct response to the COVID-19 pandemic. Note: See Disclaimer for further details.

We wish to thank the Trevor Day School of NYC (<https://www.trevor.org>) for their kind loan of multiple 3D printers, which greatly contributed to the rapid prototyping efforts of the team. Additional thanks are in order to recognize the indeterminate length of the loan (...no really, thanks for not asking for them back yet).

We also wish to thank Nick and Alex Winnard of the Berkshire Laser Co. in Pittsfield, MA (<https://berkshirlaserco.com/>) for generously donating their laser cutting and engraving services. We are grateful for their skills and expertise in creating the gorgeous acrylic enclosure, as displayed on our prototype device, and guidance improving the enclosure documentation.

1.1.5 Hardware Overview

Schematic diagram of main mechanical components

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.

Fig. 8: PVP hardware schematic

1.1.6 Components

Fig. 9: PVP hardware schematic

1.1.6.1 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 blender. 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.

1.1.6.2 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 (C_v) 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 C_v 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 (C_v) which could still actuate quickly enough to enable robust control of the gas flow. A reduced C_v 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 C_v 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 C_v 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.

1.1.6.3 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.

1.1.7 Assembly

PVP1 Assembly Instructions

This guide should help you assemble the ventilator from the parts found in the **Bill of Materials**:

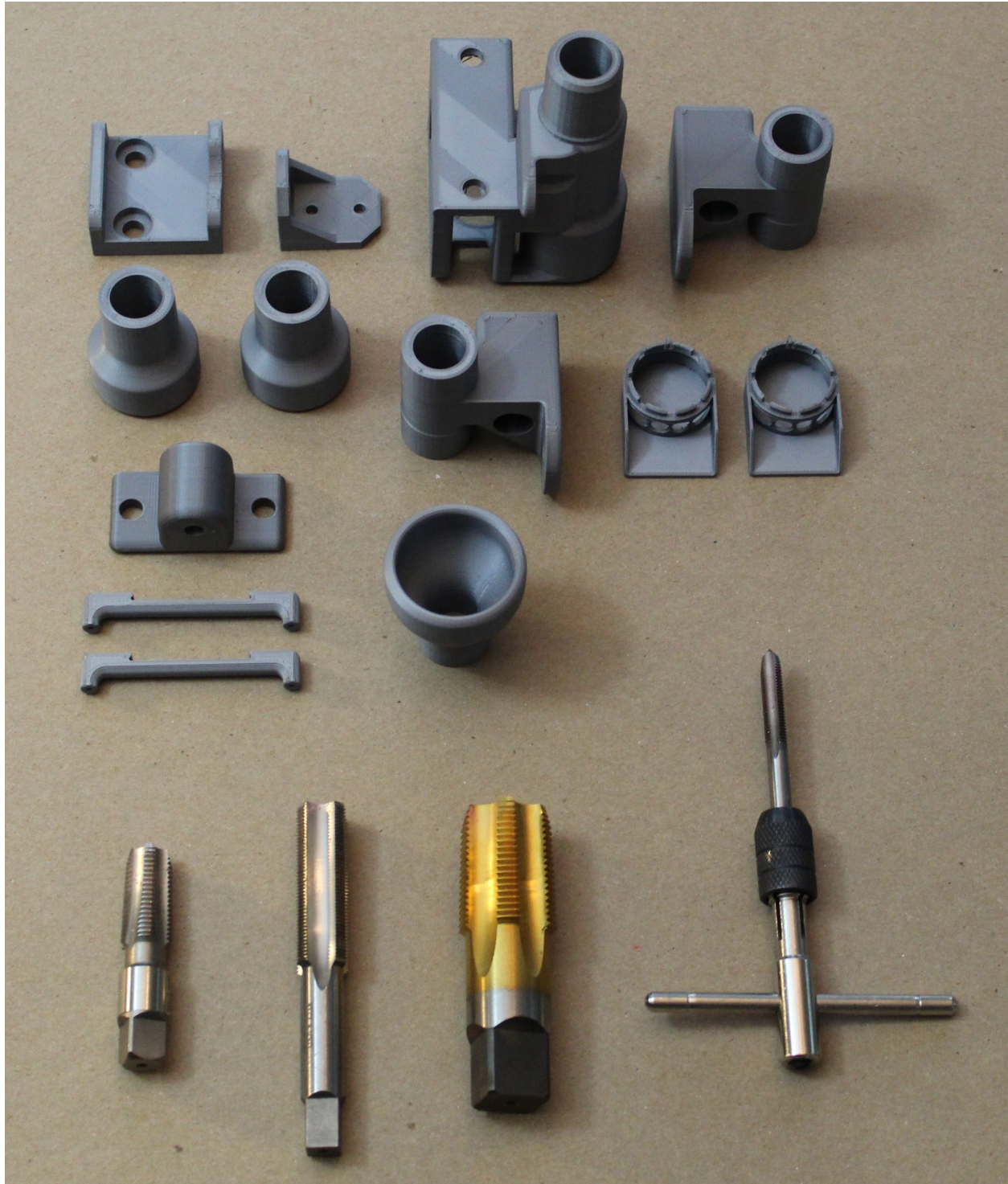
[<https://www.peoplesvent.org/en/latest/bom.html>]{.underline}

We'll first show you how to assemble the hardware, then the electronics, and finally put the two together.

The entire Solidworks Assembly, with associated part files, can be found in this Google Drive: <https://drive.google.com/drive/folders/1YrJEOmOMZtXcHABYO0hEu0etJ7LAC9Lf?usp=sharing>

The current assembly is named TLA_VENTILATOR_ASSY_V2.SLDASM.

1.1.7.1 Part 1. 3D Printed Components and Enclosure

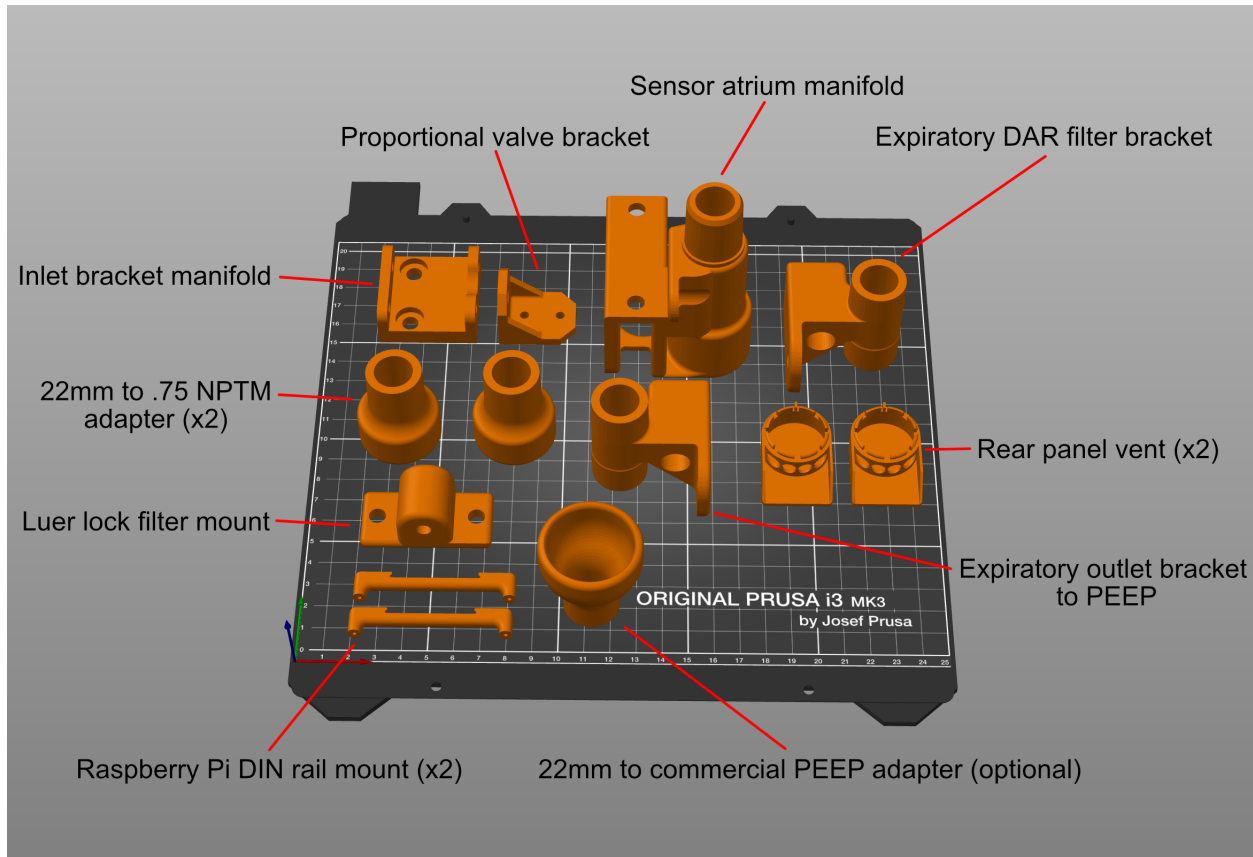


1.1 3D Printing Adapters and Brackets.

Before we can get started with assembly, you'll need to print a few parts using a standard 3D printer. (We ran our test prints on Prusa, MakerBot, FlashForge, and Creality3D printers.)

You can download all the STL files here for printing: [\[https://www.peoplesvent.org/en/latest/assembly.html#d-printed-parts\]](https://www.peoplesvent.org/en/latest/assembly.html#d-printed-parts){.underline}

Be sure to print airway components at as close to 100% infill as possible, and be mindful of printing orientation. An example printing setup is shown below. We do not recommend using supports or rafts unless you find them to be necessary, as they are challenging to remove. If there is a cylindrical channel, try to orient it vertically (such that the circle is traced along the build plate), which will improve circularity of the channel.



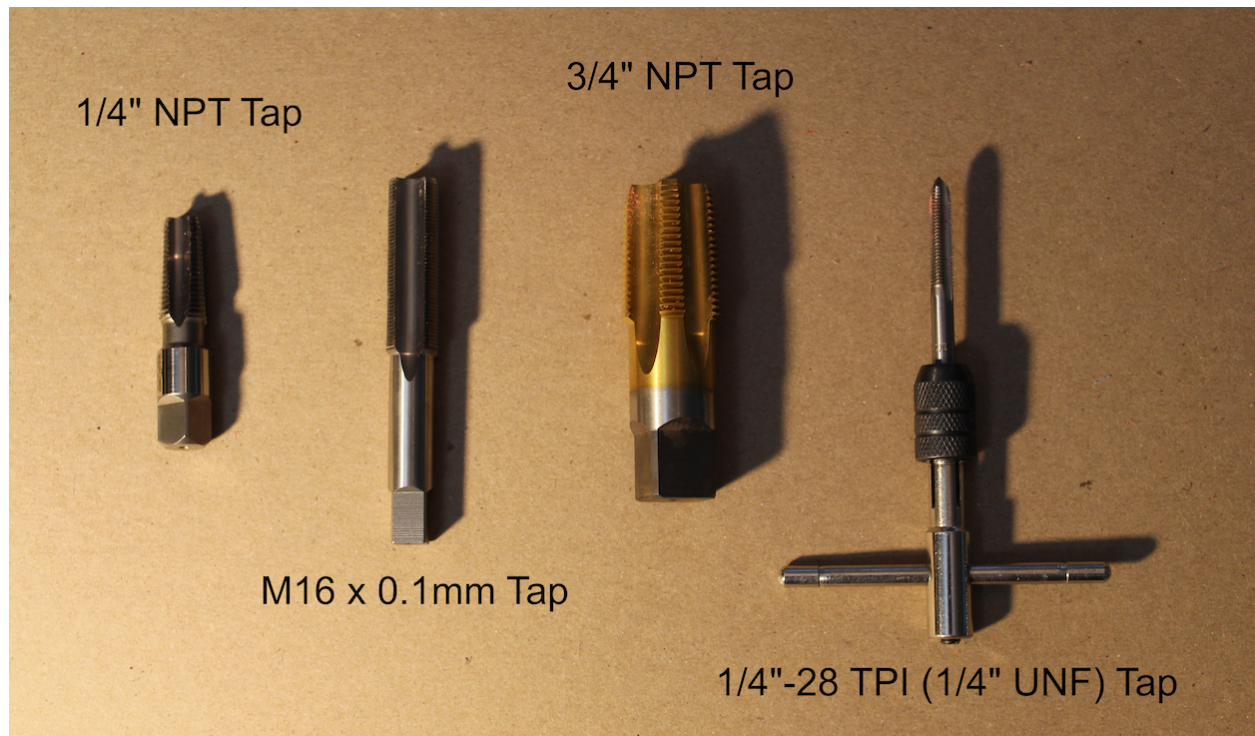
If you are using a Prusa i3, **we also provide this print setup here.**

1.2 Tapping the 3D Printed Components

Several of the 3D printed parts will need to be tapped to enable connection with other parts in the device- such as push-to-connect adapters. You will be able to tap all of these parts by hand (since the plastic cuts easily), and the **Bill of Material** contains the list of taps you will need, including specialized taps such as the M16:

<https://www.peoplesvent.org/en/latest/bom.html>

All required taps are shown below:



First, we will tap the Sensor Atrium component, which houses the oxygen sensor and “emergency breathing valve” check valve, and passes air from the inspiratory limb to the respiratory circuit (with a DAR filter in between).

Step 1. Use a 1/4” NPT tap to thread the hole as shown below (the smaller airway hole on the flat side of the Sensor Atrium), where a push-to-connect adapter will attach.



Step 2. Use the M16 tap to thread the hole on the opposite side of the Sensor Atrium, where the oxygen sensor will attach.



Next, we will tap the adapters at either side of the expiratory solenoid.

Step 3. Use the ¼" NPT tap to thread the holes on the larger end of the two "22mm to 0.75 NPTM adapter" parts.



Last, we will tap the adapter to the pressure sampling lines.

Step 4. Finally, use the ¼"-28 tap to thread the two airway holes on the "Luer lock filter mount" part. These will hold the metal luer lock adapters to the gas sampling lines for monitoring pressure.



1.3 Cutting Enclosure Pieces.

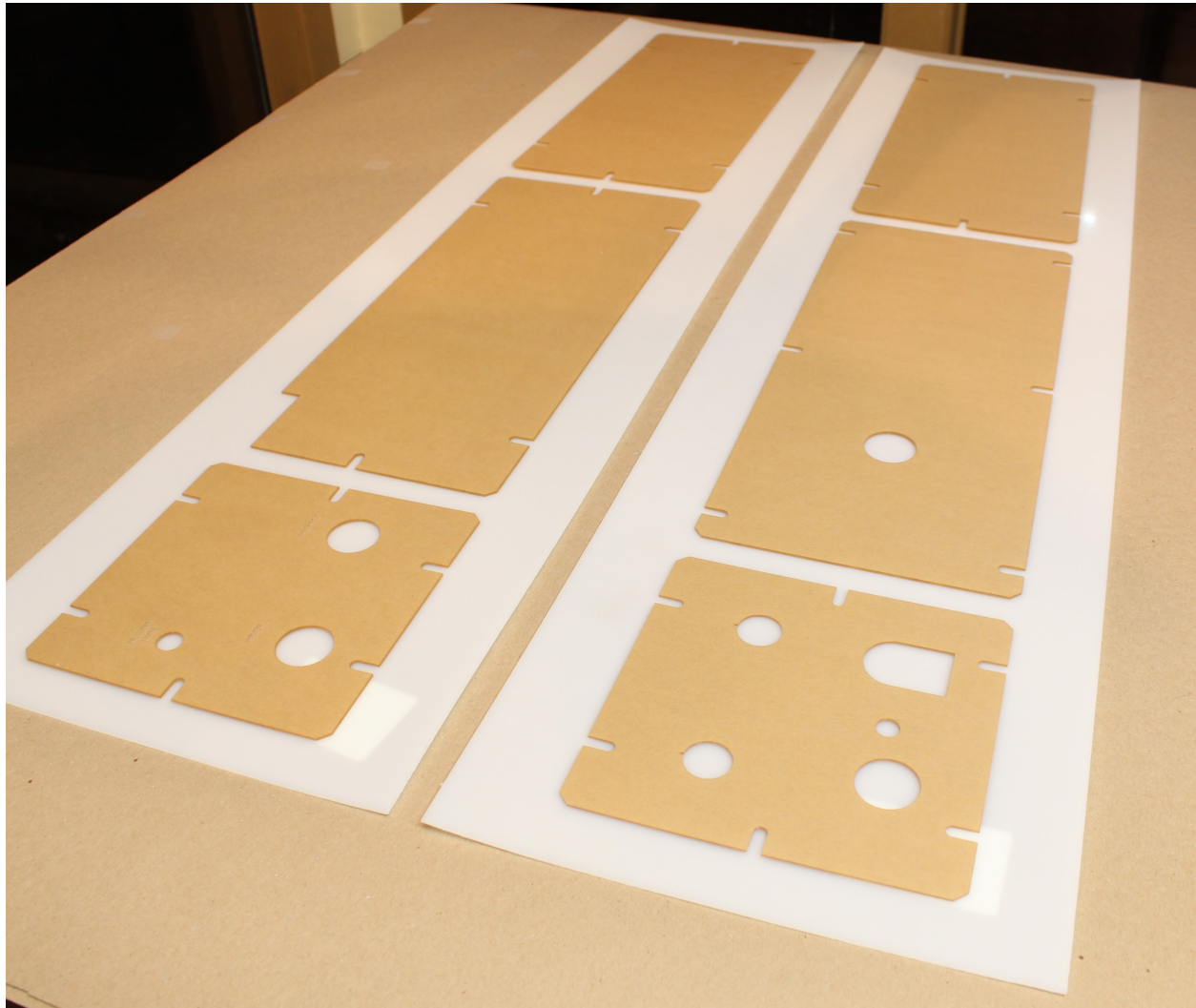
Step 1. Laser cut, or cut out by hand, the six HPDE side panels.

Cut all panels out of the 1/16" HPDE sheets. If you wish to cut these pieces using a laser cutter, we provide DXF files here, under "Enclosure": [\[https://www.peoplesvent.org/en/latest/assembly.html\]](https://www.peoplesvent.org/en/latest/assembly.html){.underline}

Cuts can also be made by hand using a sharp pair of scissors and a razor-cutter. Pieces are 17 " by 7 " (45.4025 cm by 20.0025 cm), or 7 " by 7 " (20.0025 cm by 20.0025 cm) along the outer dimensions. Inner hole placement can be determined from the documentation, also at:

[\[https://www.peoplesvent.org/en/latest/assembly.html\]](https://www.peoplesvent.org/en/latest/assembly.html){.underline}

(We cut our side panels for the demo out of acrylic for ease of visibility.)



Step 2. Insert the rubber grommet(s).

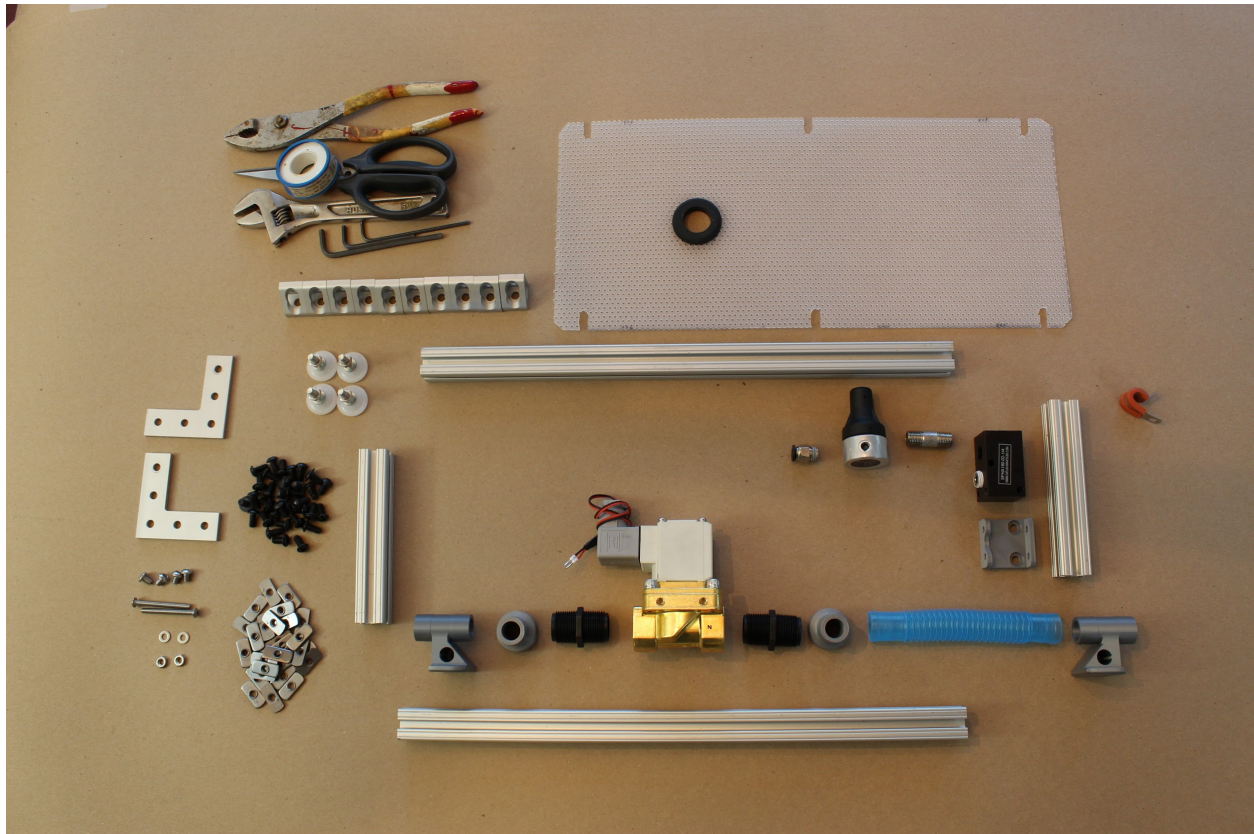
The bottom panel includes a hole for a large rubber grommet. Insert this by hand.

If you wish, you may also insert a small custom grommet in the back panel.



1.1.7.2 Part 2. Basic Hardware Assembly

2.1 Assembling the bottom frame



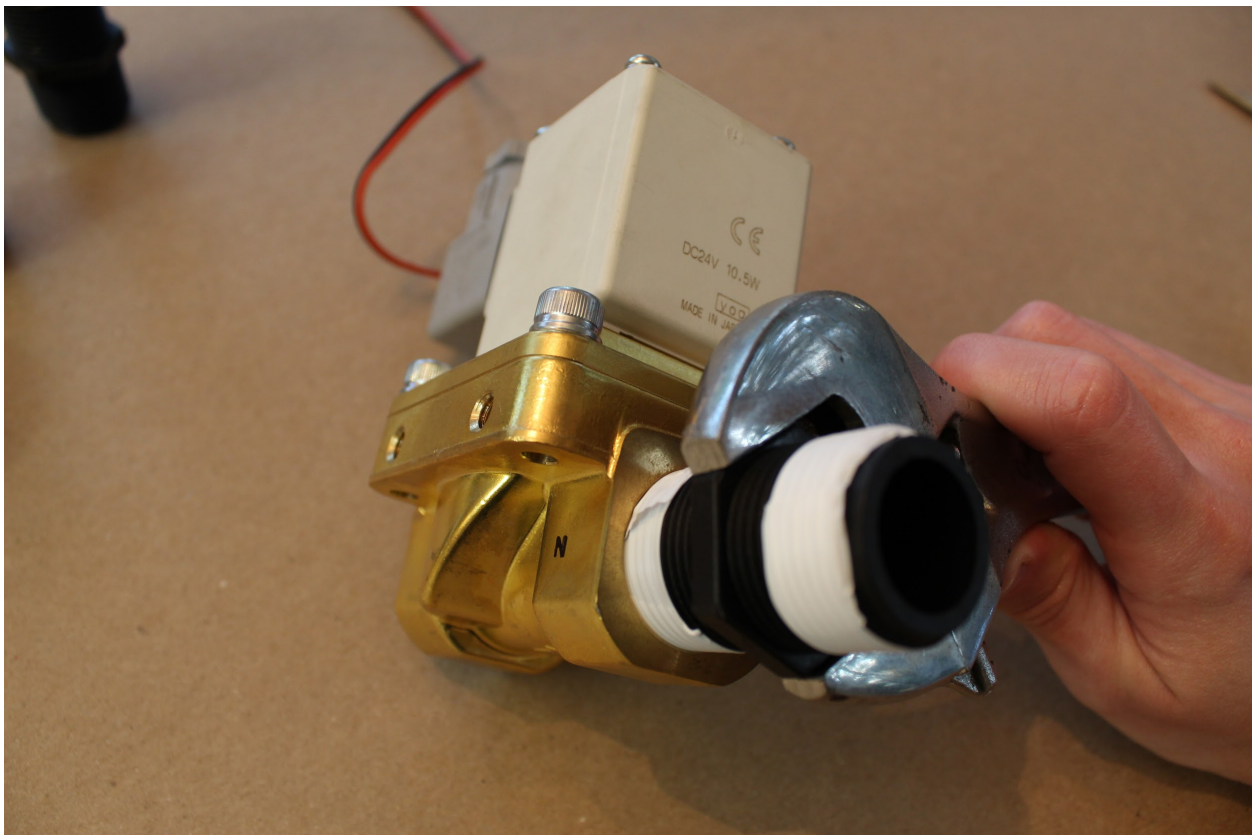
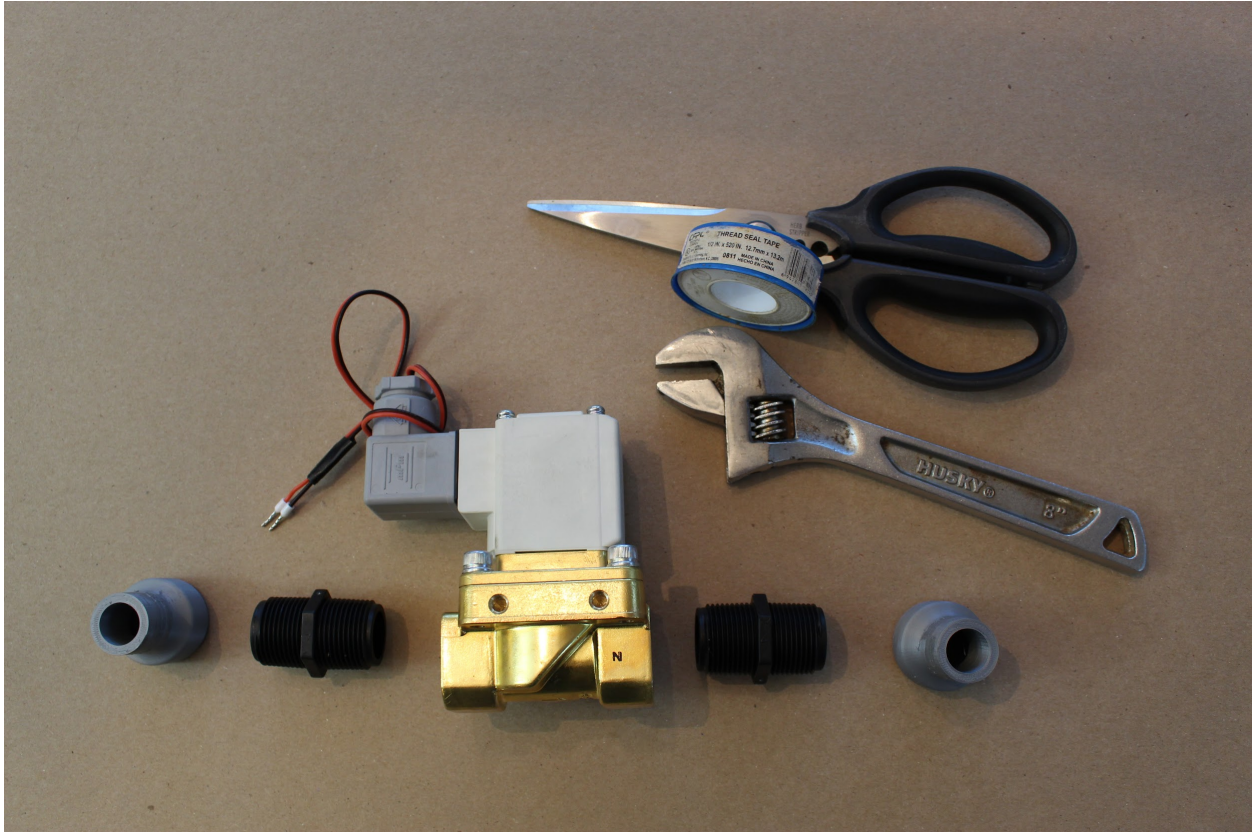
Step 1. Cut the 80/20 (“T-slotted framing”) to appropriately sized pieces: In total, you will need 4 pieces of length 17 in (45.4025 cm), and 9 pieces of length 5 in (14.9225 cm).

You can cut the aluminum 80/20 pieces by hand with a hack-saw, or using machinery such as a bandsaw. Either way, be sure to file down any rough edges afterward!



Step 2. Attach the $\frac{3}{4}$ " NPT (male-male) connectors to either side of the expiratory solenoid, and then attach the two "22mm to .75 NPTM adapter" parts to those, as shown. When attaching any airway threaded parts in this assembly process, be sure to use PTFE thread sealant tape, wrapping twice around the threads, in the direction shown (away from your body if the threads are oriented towards the right).

(Note: From now on, we will not explicitly write out the need for Teflon tape; be sure to use it whenever a threaded airway component is involved!)



Step 3. Ensure that the two side ports of the pressure regulator are blocked off, with the plugs included with the part (and Teflon tape, as always). Attach a 1/4" NPT push-to-connect adapter to the pressure regulator on the "outlet" side: check the bottom of the part to determine which side is "IN" for "inlet". The inlet manifold will have two 1/4" NPT ports on the same side: plug one of those with the manifold plug, and attach the other end to the 1/4" NPT connector. Attach the other end of this connector to the "inlet" end of the pressure regulator.

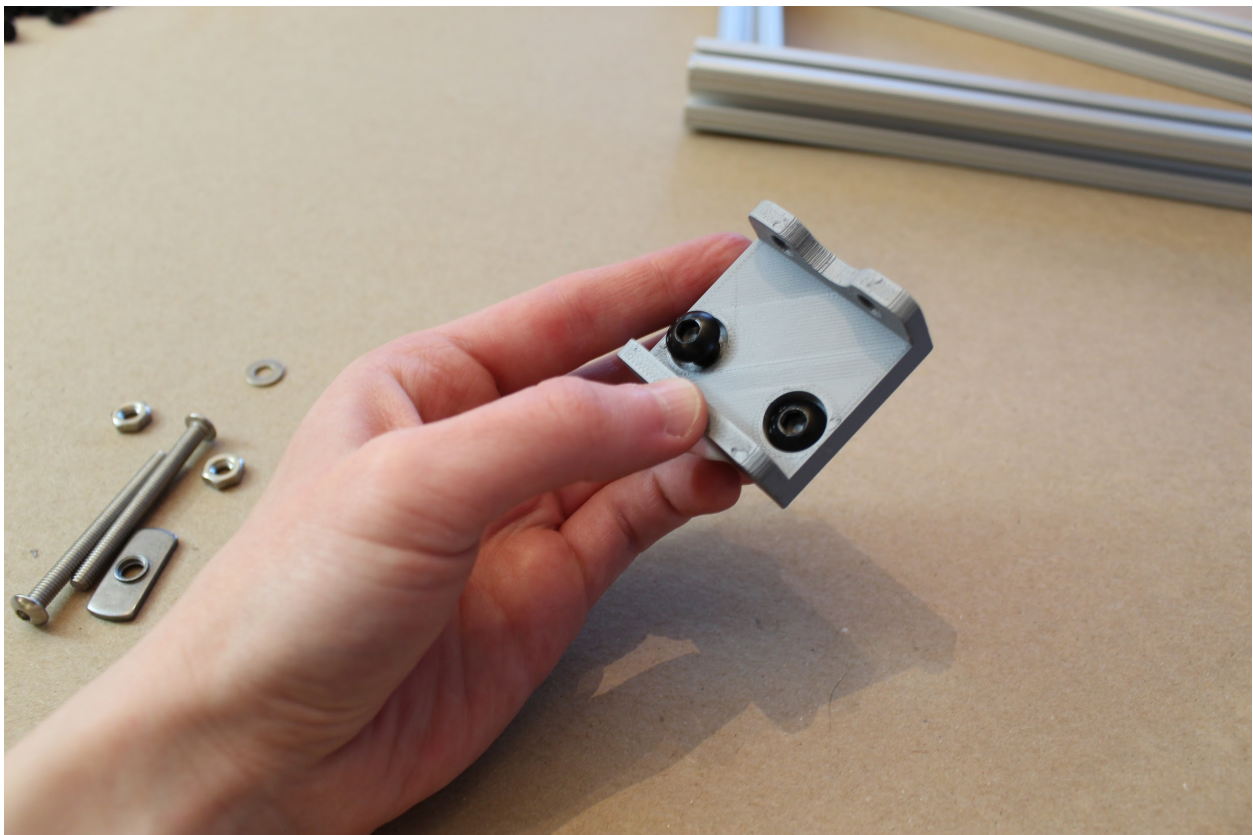
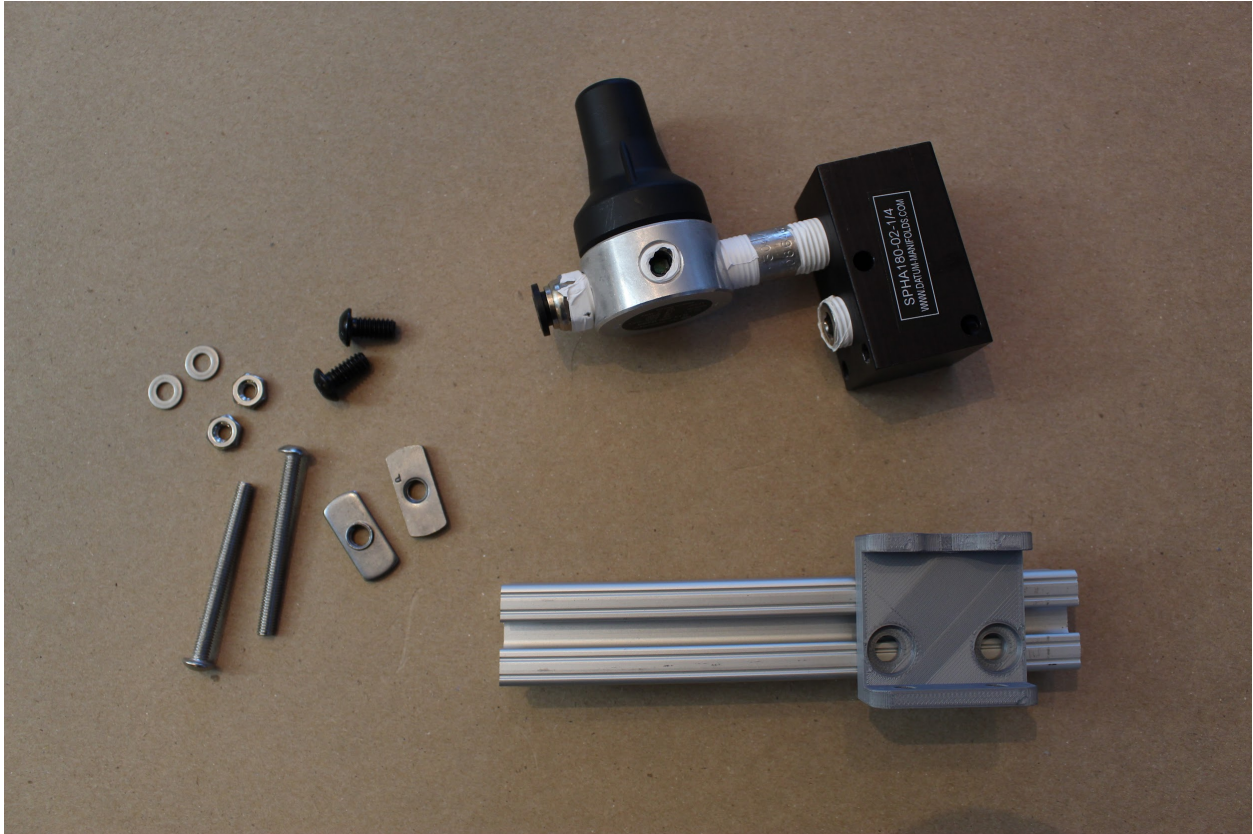


Step 4. Mounting the inlet manifold.

First, attach the 3D-printed "Inlet manifold bracket", by pushing two standard button-head screws down from the top of the piece (as shown, such that the button heads fall in the grooves), then loosely attaching a hex nut to each. Then, slide the two hex nuts into a T-slot of a short (5") 80/20 piece, and use an Allen key to secure the screws such that the printed piece is as far to the "right" as it can go (as shown), without the hex nuts extended past the length of 80/20.

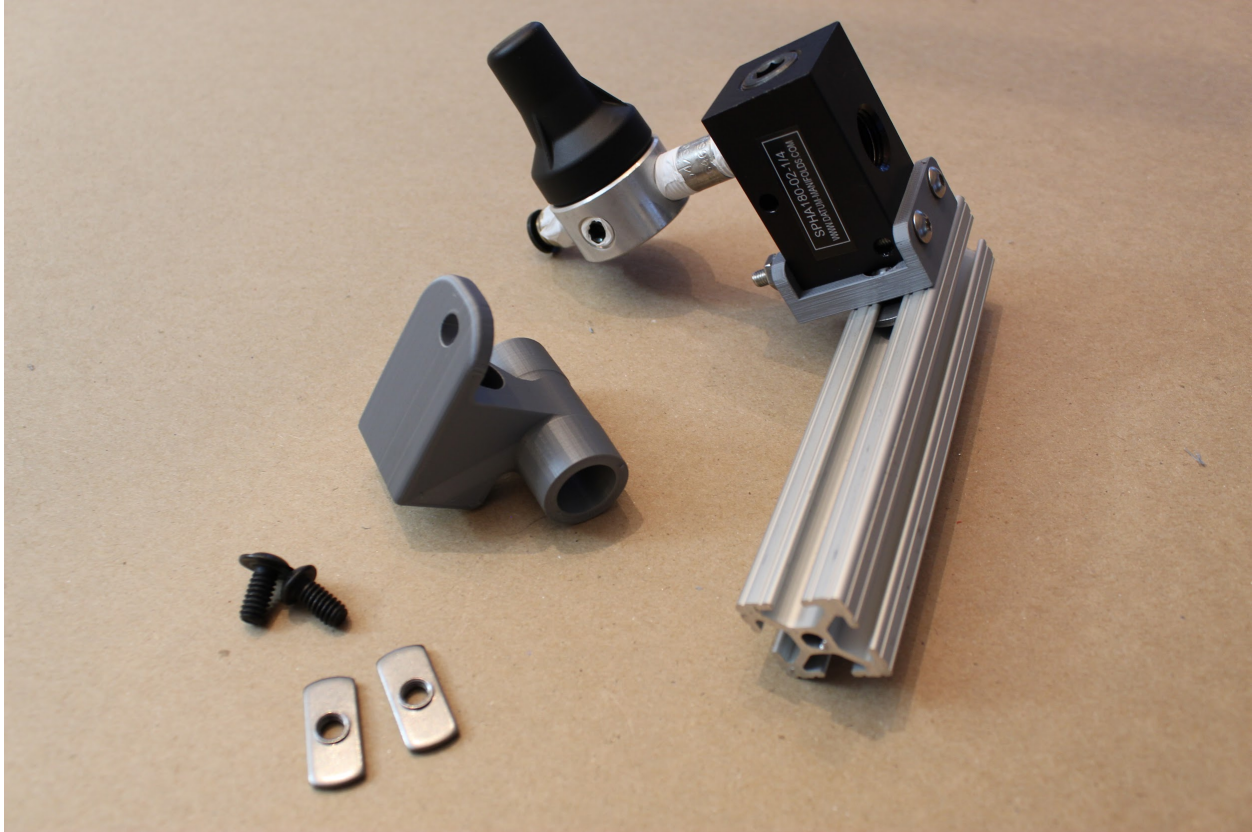
Tip: when attaching hex nuts, make sure the side with the nub is facing down (such that it touches the inner channel of the 80/20 T-slot).

Then, insert the inlet manifold into the printed piece, as shown: if the printed part is to the right of the 80/20 length, then the pressure regulator should be oriented away from you. Insert the 2" button head hex drive screws through the remaining holes in the 3D printed piece, to secure the inlet manifold, as shown (also away from you). Finally, attach a washer (W_10_NARROW_0.406OD_316_SS) and hex nut (LN_10-32_STAINLESS_18-8) to each.



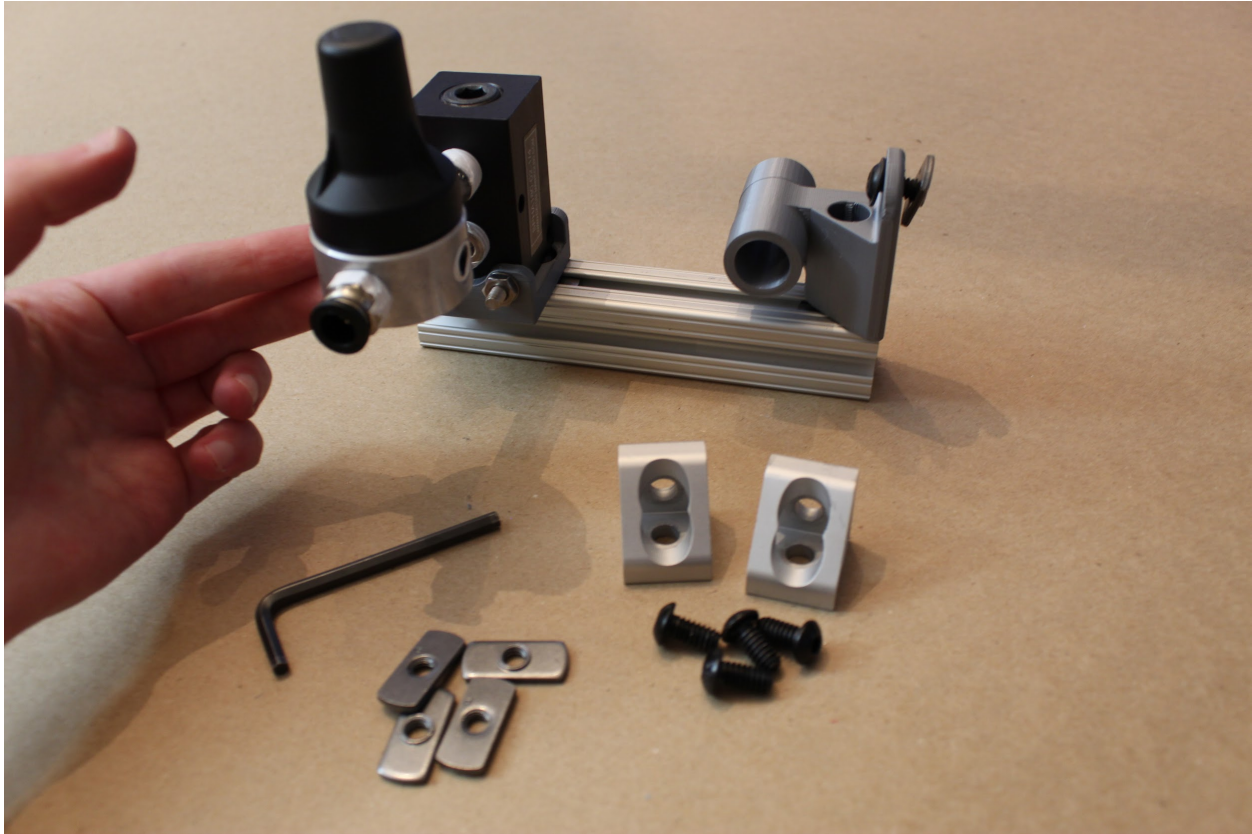
Step 5. Attach the “Expiratory outlet bracket to PEEP”.

Drop one standard button head hex screw down the deeper channel, and attach as before: loosely screw on a hex nut, slide it in the channel, and then use an Allen key to tighten the screw. This time, tighten the hex screw such that the printed part is in-line with the edge of the 80/20 piece, as shown. Then, insert a second button head hex screw, facing outward as shown, and loosely attach a hex nut (you will tighten this later).

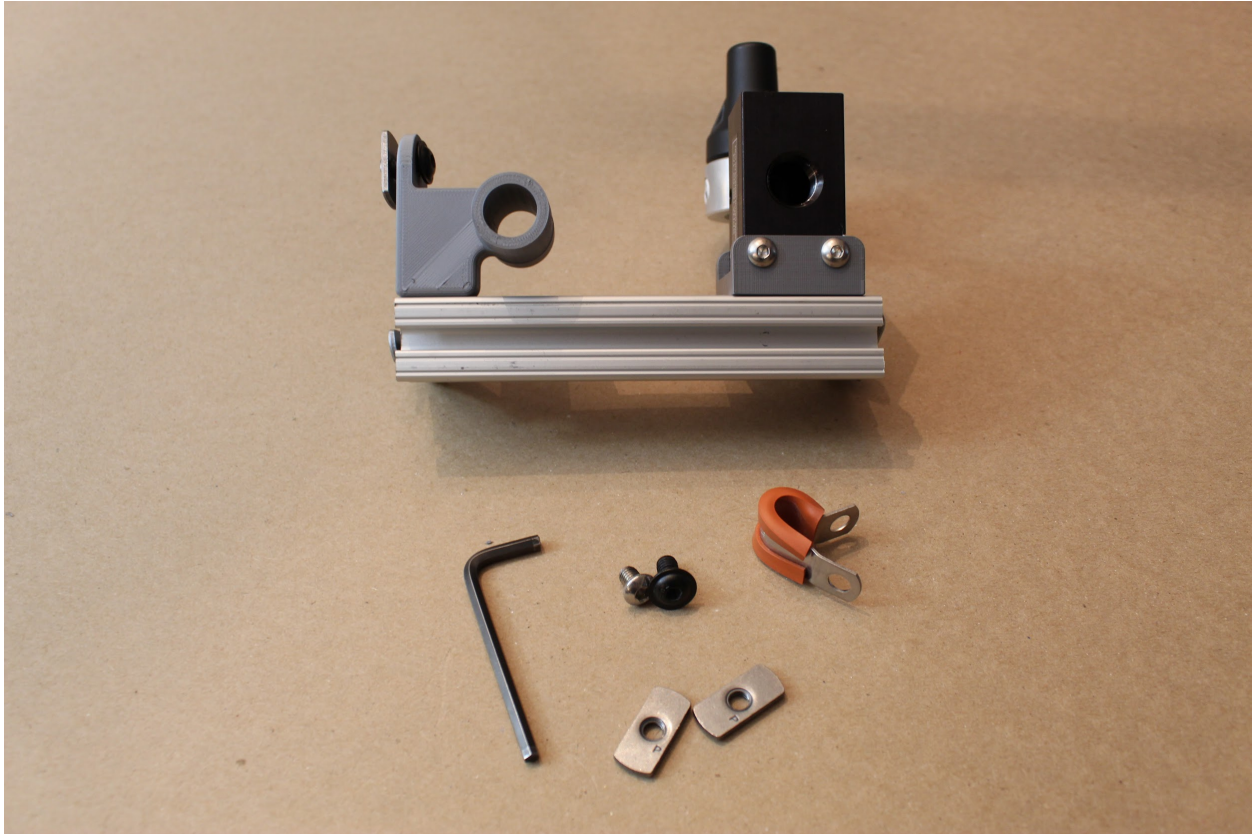


Step 6. Attach corner brackets to finish this frame piece.

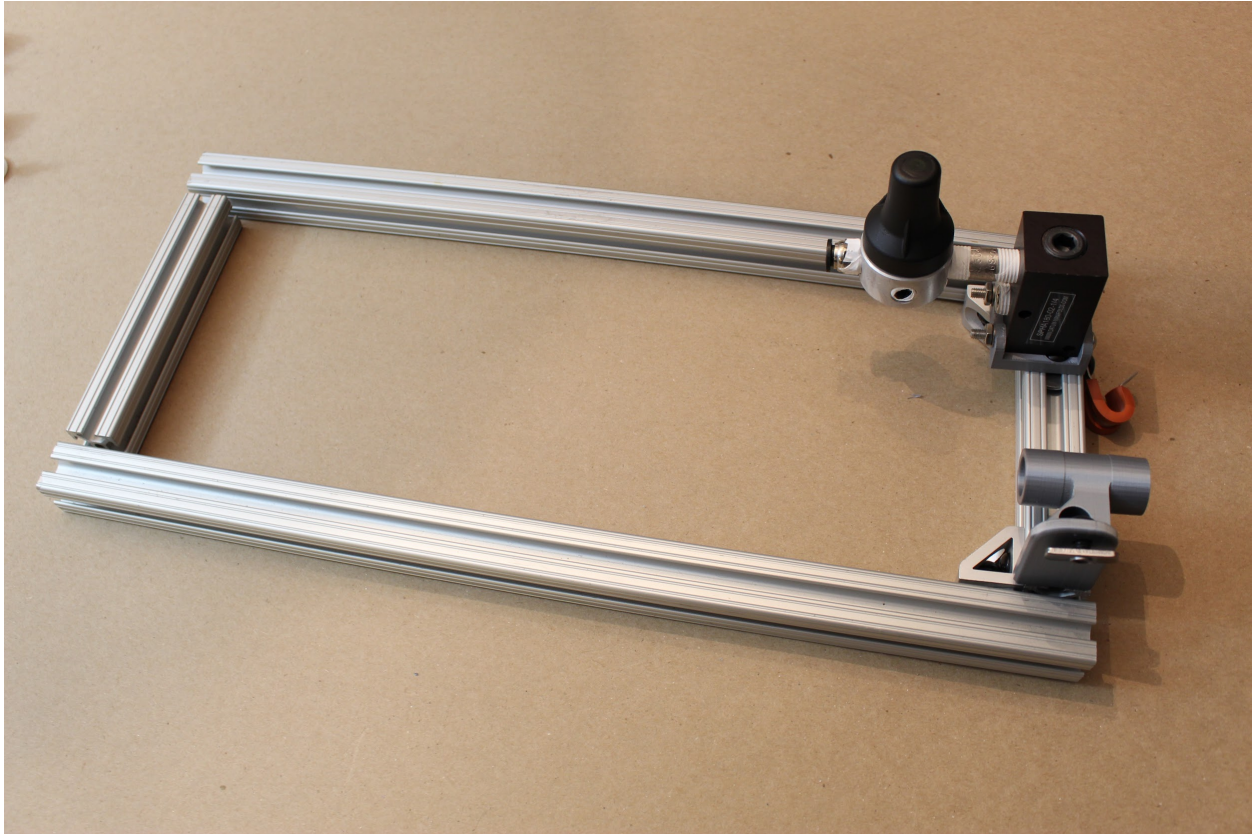
Pre-assemble the gusset (corner) brackets by inserting a hex screw into each hole, and then loosely attaching a hex nut to each, as shown. Tighten the gussets so that they align with the edge of the 80/20 piece, as shown, keeping the second hex nut on each piece loose. (You will use these to attach another 80/20 piece later.)

**Step 7. Attach cable tie and side panel screw.**

If you intend to use the cable P-clip for cable management, and to attach the HPDE side panels, attach the P-clip using a standard button head hex screw, and mount an additional hex screw for attaching the side panel later.



Progress: One piece of the bottom frame is assembled!

**Step 8. Assemble the opposite side of the frame.**

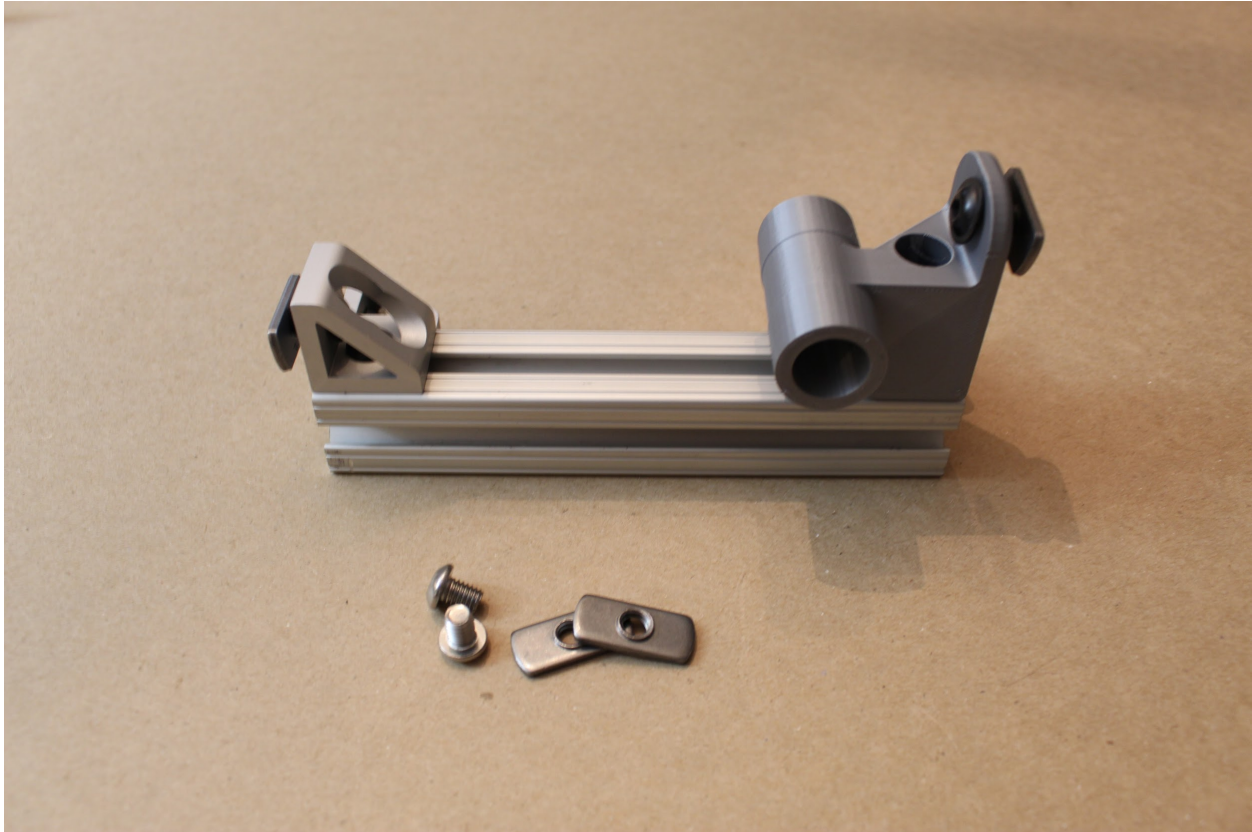
Assemble a second short (5 ") piece of 80/20 as shown: this uses the 3D printed part called the "Expiratory DAR filter bracket", which will mirror the "Expiratory outlet bracket to PEEP" part on the opposing frame leg. Also, attach three gusset (corner) brackets, as shown.

***Note:** if you intend to attach the side panels, leave off the corner bracket on the same side as the 3D printed piece. If you're not using the side panels, keep the bracket here for additional support!

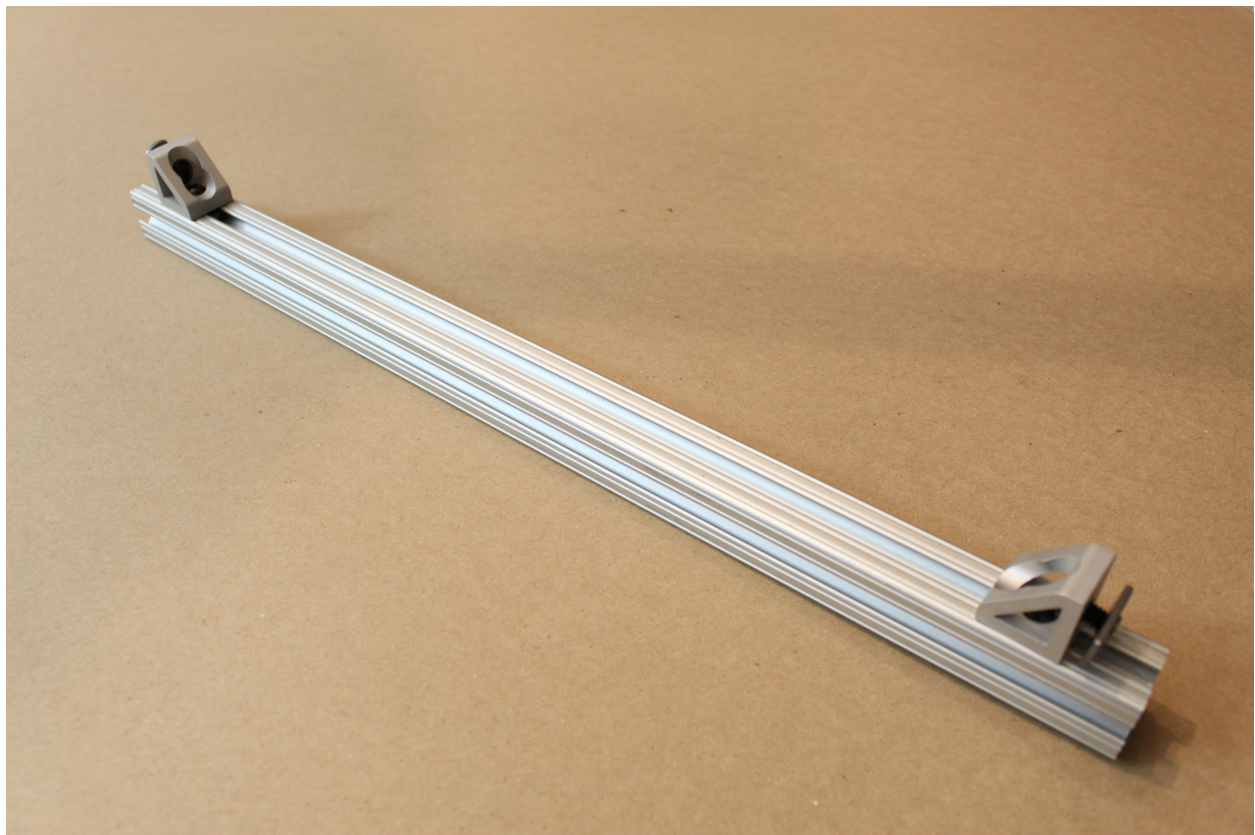
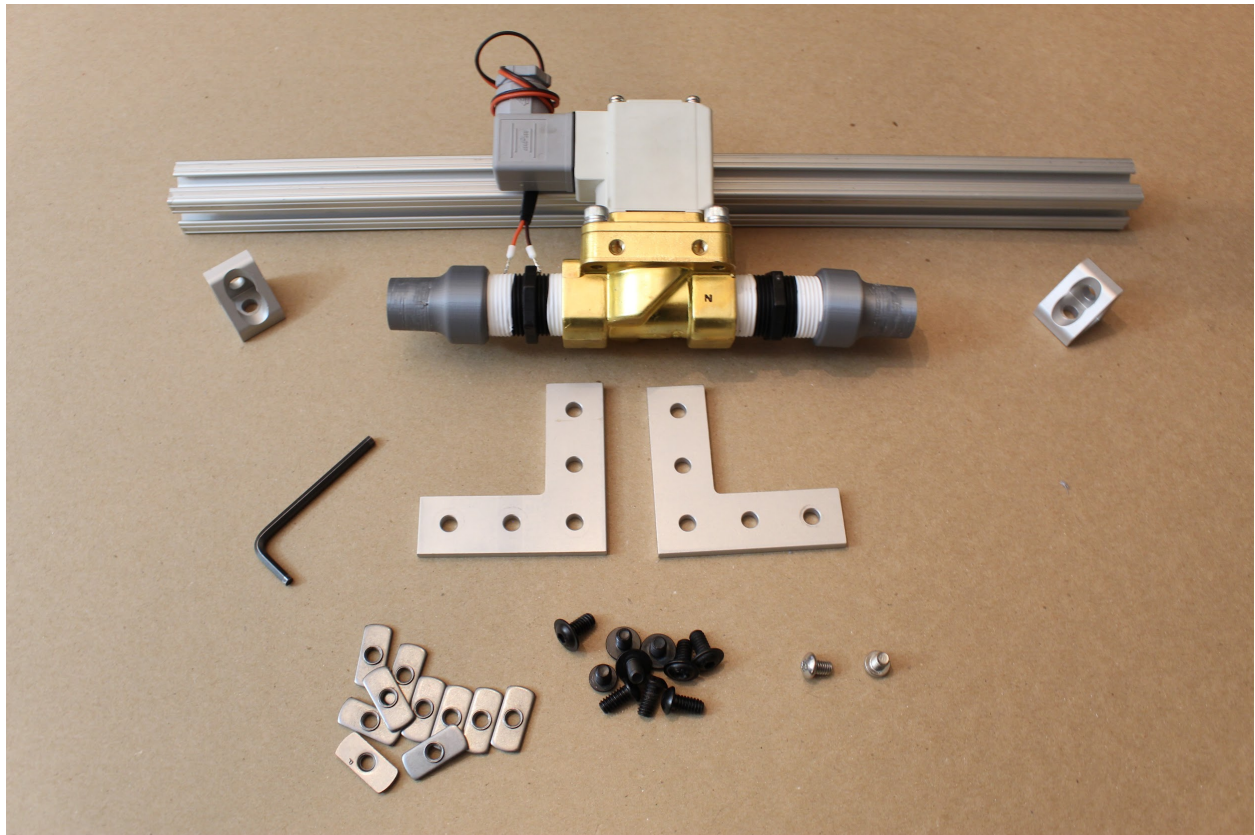


Step 9. Attach side panel screws.

As before, if you plan to attach HDPE side panels later on, attach the screws for them now, as shown.

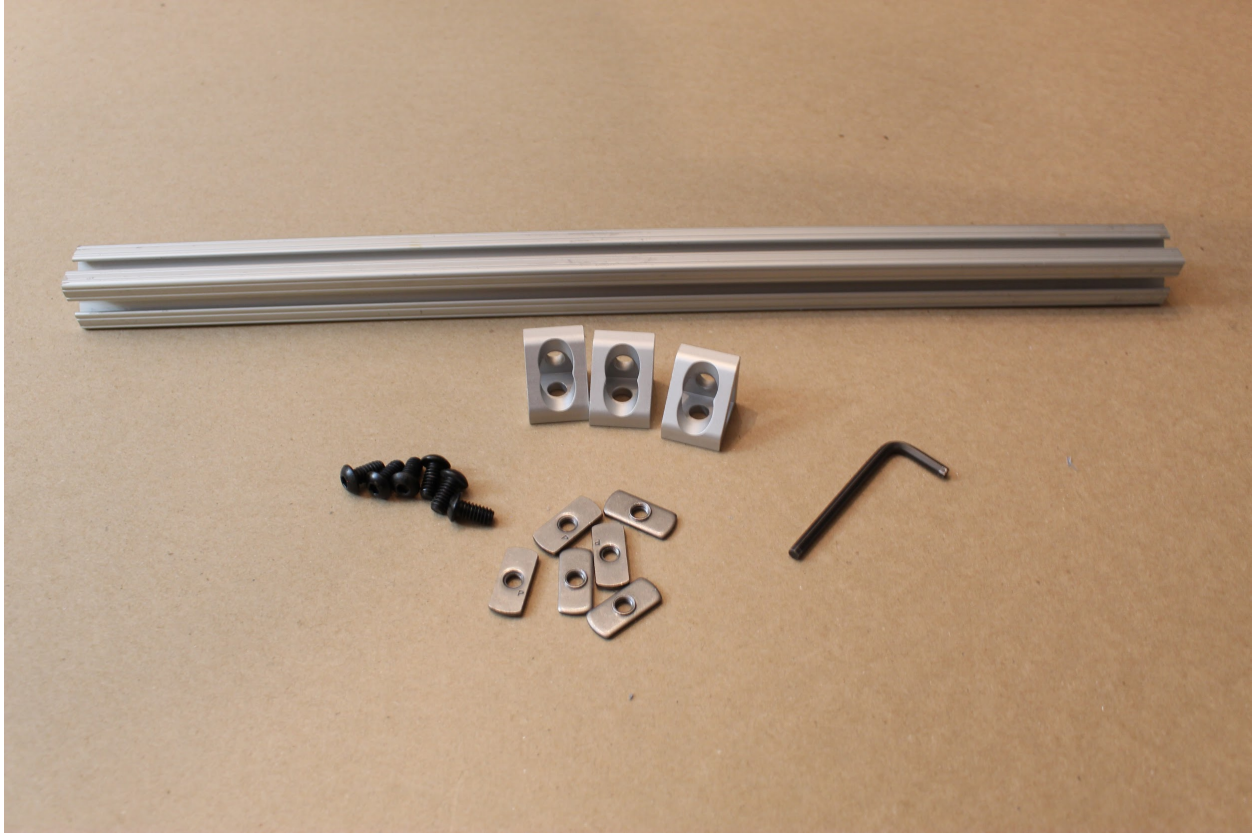
**Step 10. Attach the expiratory solenoid assembly to the frame.**

- First, assemble and attach two gusset (corner) brackets to their hex screws/nuts, then affix to a long (17 ") leg of the framing, leaving 1" from the end of the piece on each side (that is, leaving space for a vertical piece of 80/20).
- Then, attach three hex screws/nuts to the lower rows of each 90-degree angle bracket as shown; then attach the angle brackets by inserting a short (1mm long) button head screw through the topmost hole of each angle bracket in the opposite direction. These shorter hex screws will screw directly into the solenoid.
- Finally, slide the hex nuts attached to the angle brackets into the T-slot of the 80/20 piece, such that the gusset brackets and solenoid are oriented above the 80/20 piece, as shown. Loosely tighten these in place; you will adjust the position of the solenoid later.

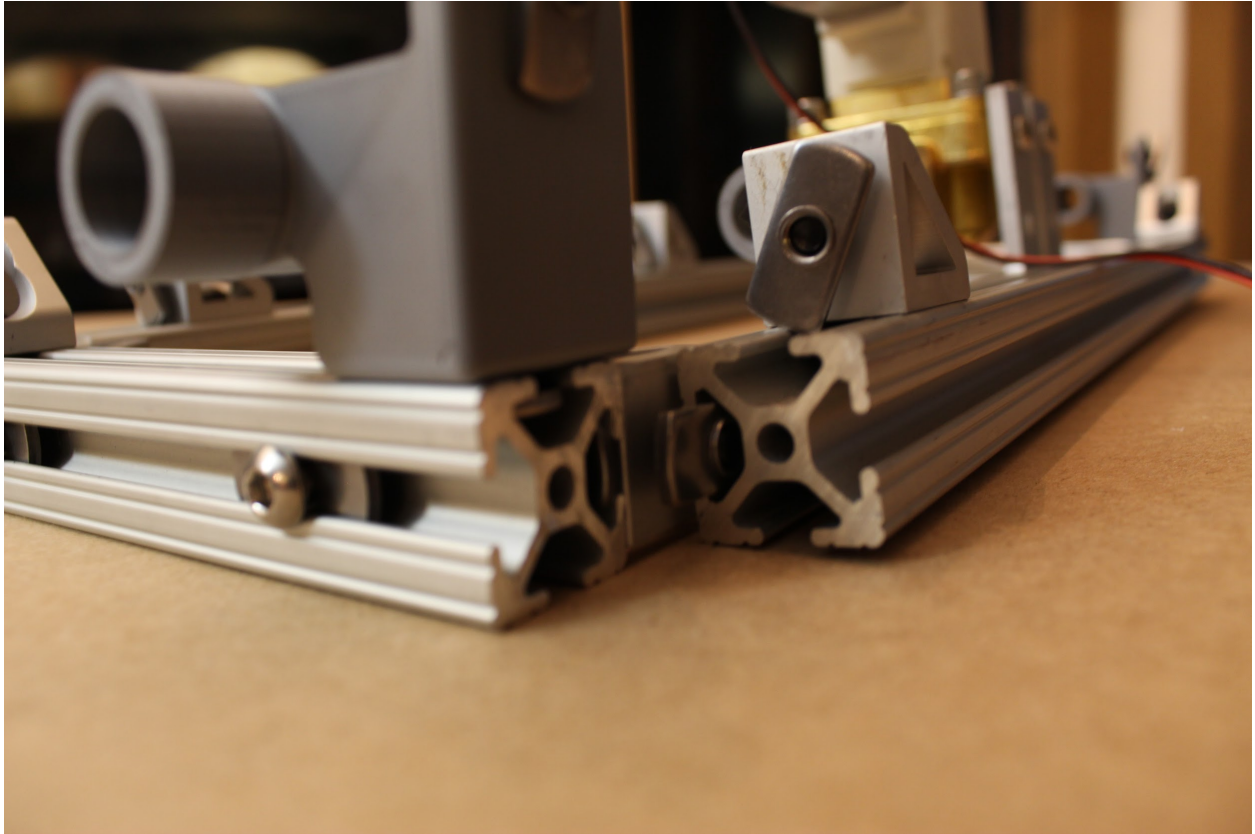


Step 11. Attach gusset (corner) brackets to the opposing 80/20 leg.

Assemble the gusset (corner) brackets with hex screw/nuts as before, and attach them to a long (17 ") 80/20 piece. Leave 1" of space from each of the side gussets (sufficient space for a vertical 80/20 piece), and affix the third bracket just off-center (to support a centered, vertical 80/20 piece).

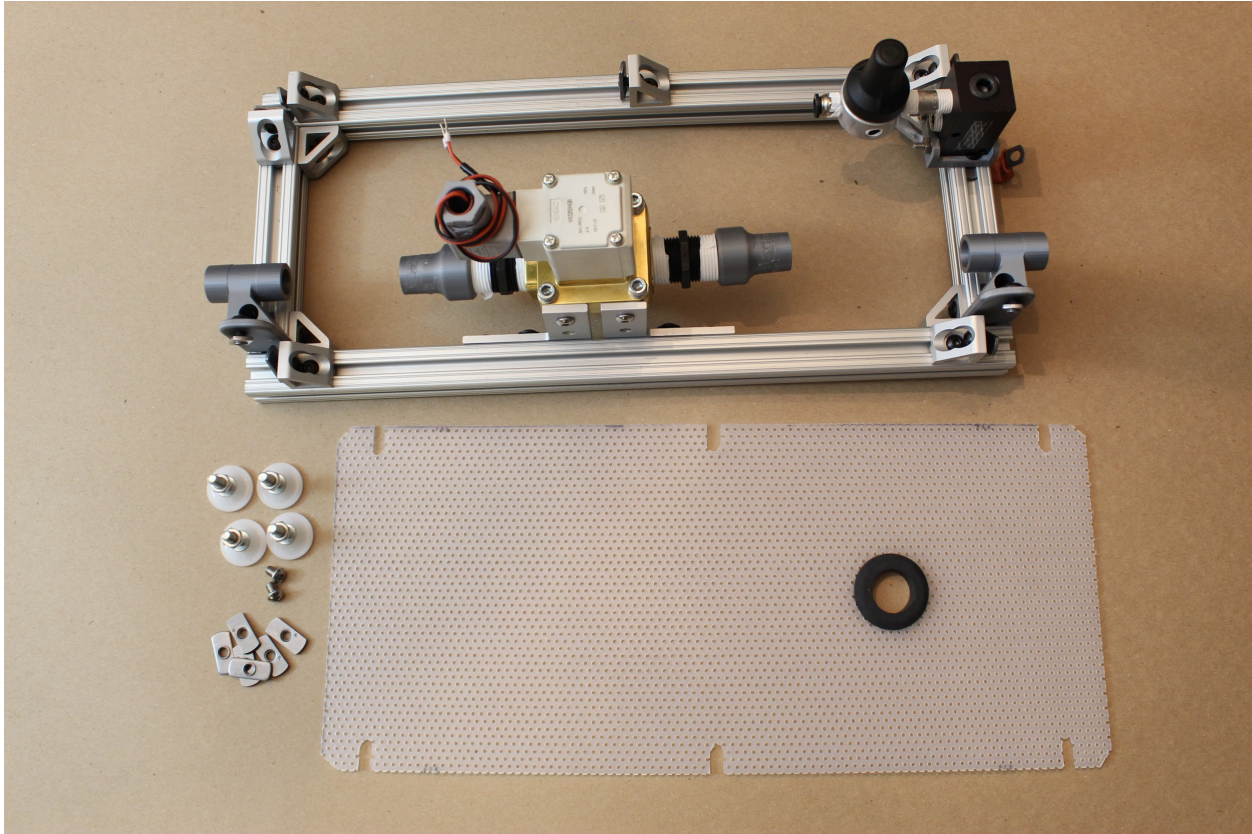
**Step 12. Assemble the bottom frame components.**

Slide the hex nuts on the shorter 80/20 pieces you have assembled into the longer 80/20 slots, and use an Allen key to tighten the screws in place, as shown. The shorter 80/20 legs should be flush with the ends of the longer 80/20 legs when you are done, and the gusset (corner) brackets will help stabilize the frame.

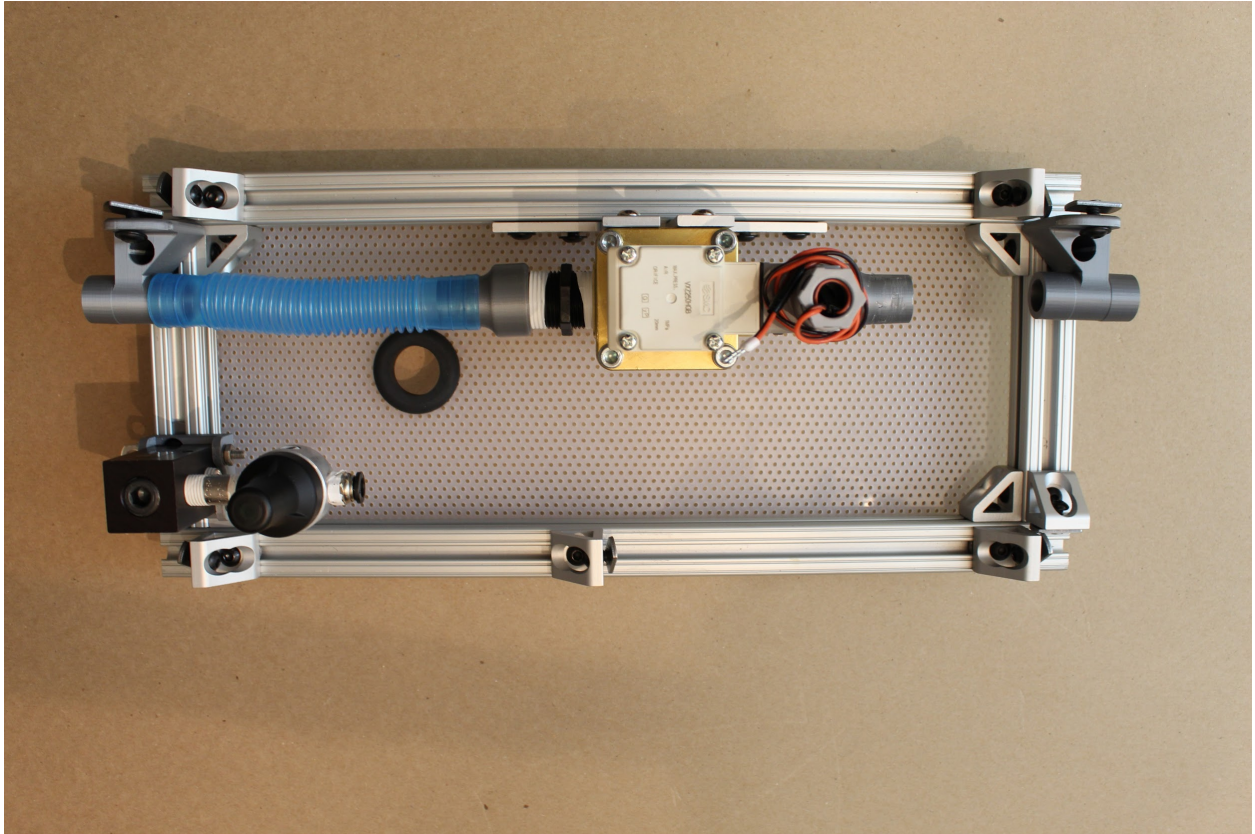


Step 13. Attaching the ventilator “feet” and bottom panel.

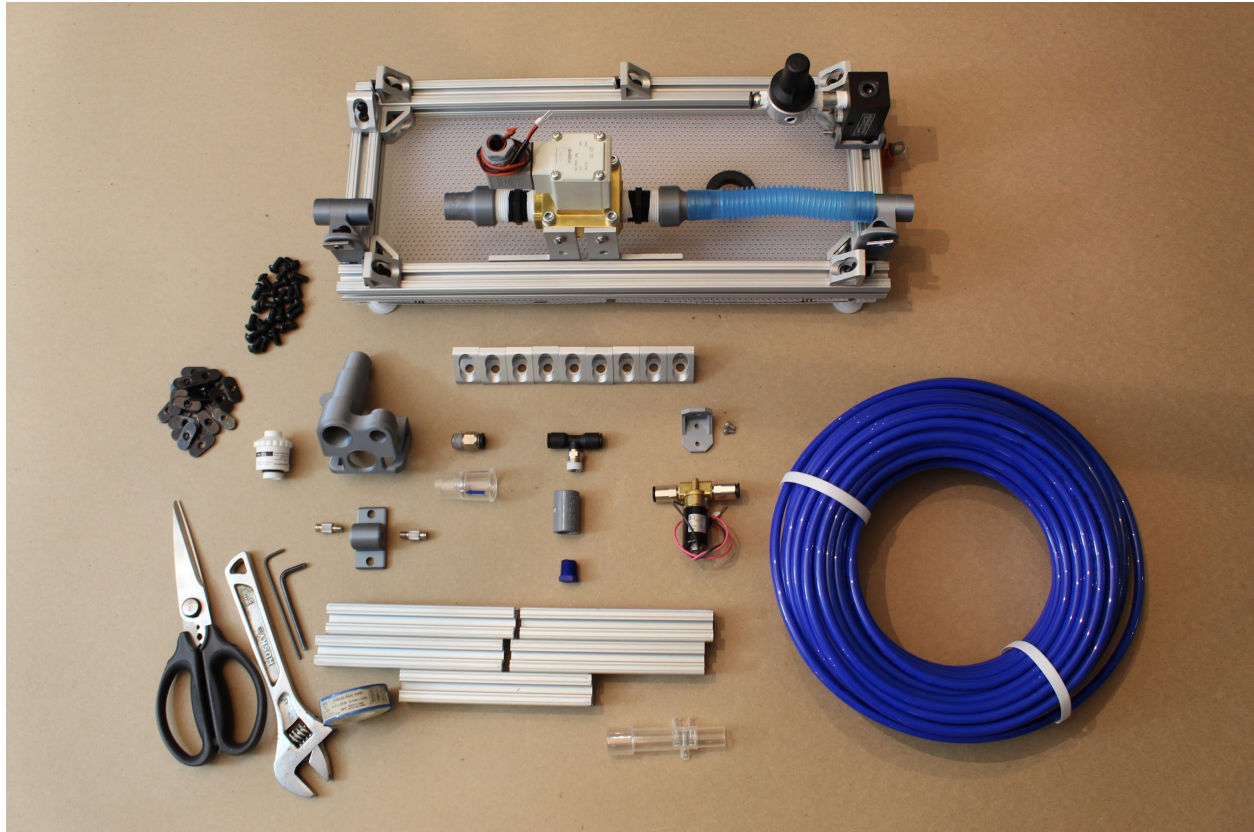
If you are attaching the lower, perforated HPDE sheet, insert the rubber grommet. Pre-assemble the shorter (1mm) button head screws to hex nuts, and the leveling mounts (feet) to hex nuts. Use these to attach the lower HPDE panel as shown, by sliding the hex nuts along the long channels on the bottom of the frame assembly; then tighten in place.

**Step 14. Attach the respiratory tube segment.**

Awesome- almost done with the lower level of the ventilator! As a last step, attach the short segment of respiratory circuit between the “Expiratory outlet bracket to PEEP” and the nearest “22mm to .75NPTM adapter”. You’ll be able to twist this on by hand.

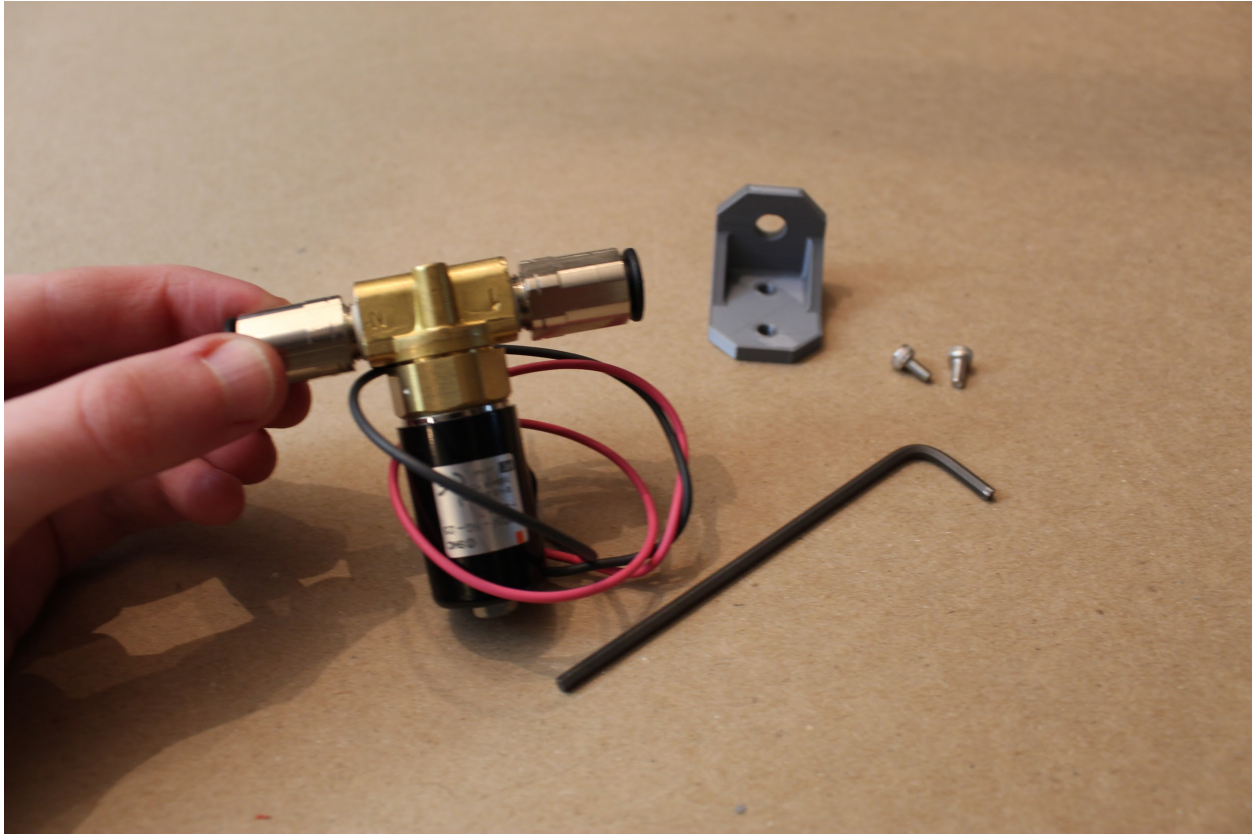


2.2 Assembling the frame sides



Step 1. Attach the proportional valve to its mount.

Use the two socket head screws (6mm long) to attach the proportional valve to the 3D printed “proportional valve bracket”. The inlet should be to the right in the orientation shown below.



Step 2. Attach the pressure relief valve to its adapters.

Assemble the T-line push-to-connect adapter, to a female-female ¼" NPT connector, and finally to the Nylon pressure release valve.



Step 3. Attach the luer lock connectors to the luer lock filter mount.

These pieces will screw in, with the luer lock portions facing outwards.



Step 4. Assemble the sensor atrium.

Attach the ¼" NPT push-to-connect adapter, check valve, and oxygen sensor into the appropriate holes in the "Sensor Atrium Manifold", as shown. Do not push the check valve into the device too far, as it may restrict air flow within the atrium.



Step 5. Attach sensor atrium to a short 80/20, and then to the device.

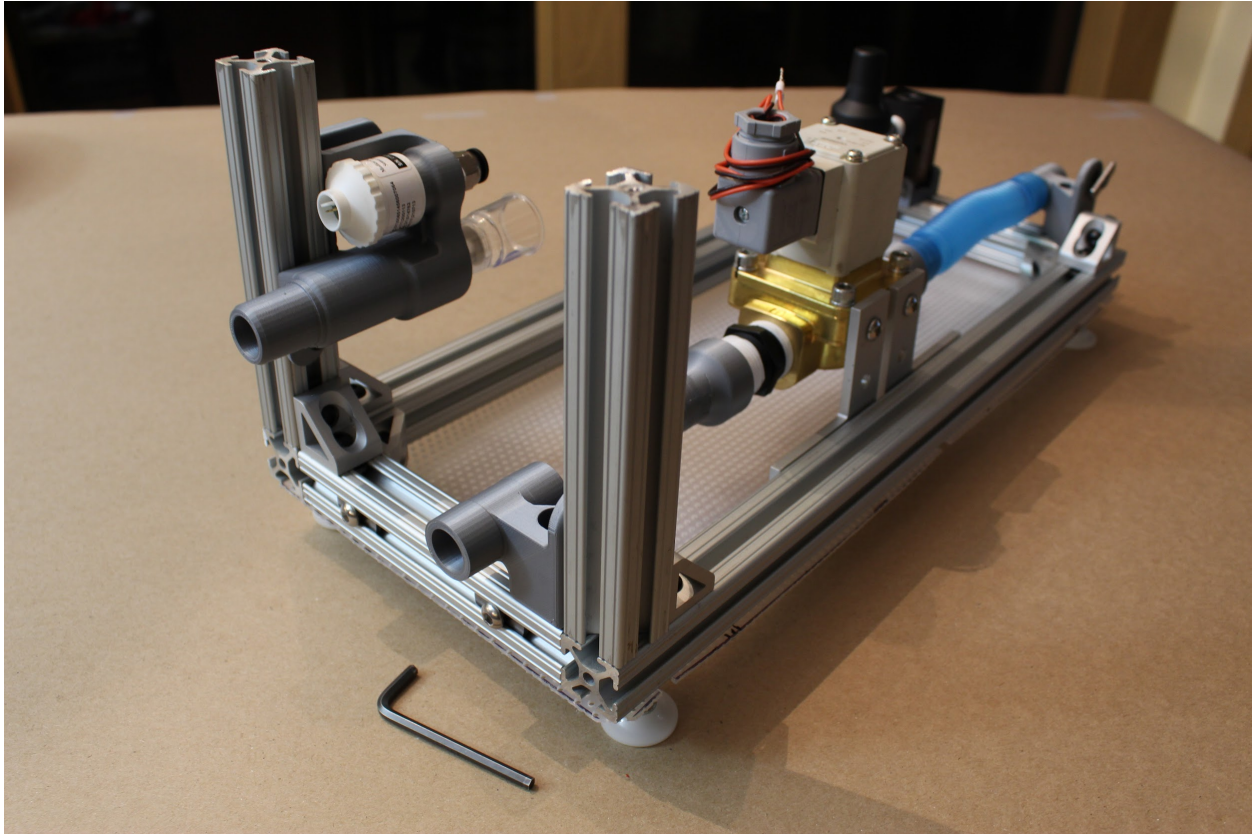
Affix button head screws/nuts to the sensor atrium, with hex nuts facing towards one another, as shown. Slide the hex nuts along T-slots on either side of a short (5 ") 80/20 piece; then insert this vertically into the device as shown. Tighten into place via the hex screws on the lower gusset (corner) brackets.

***Note:** If you are attaching the side panels, be sure to use a **SHORT** nut on the sensor atrium (the nut on the side with two, farther from the oxygen sensor), and keep the short side facing down when inserting. This will allow the sensor atrium to drop lower and align with the holes in the front panel.



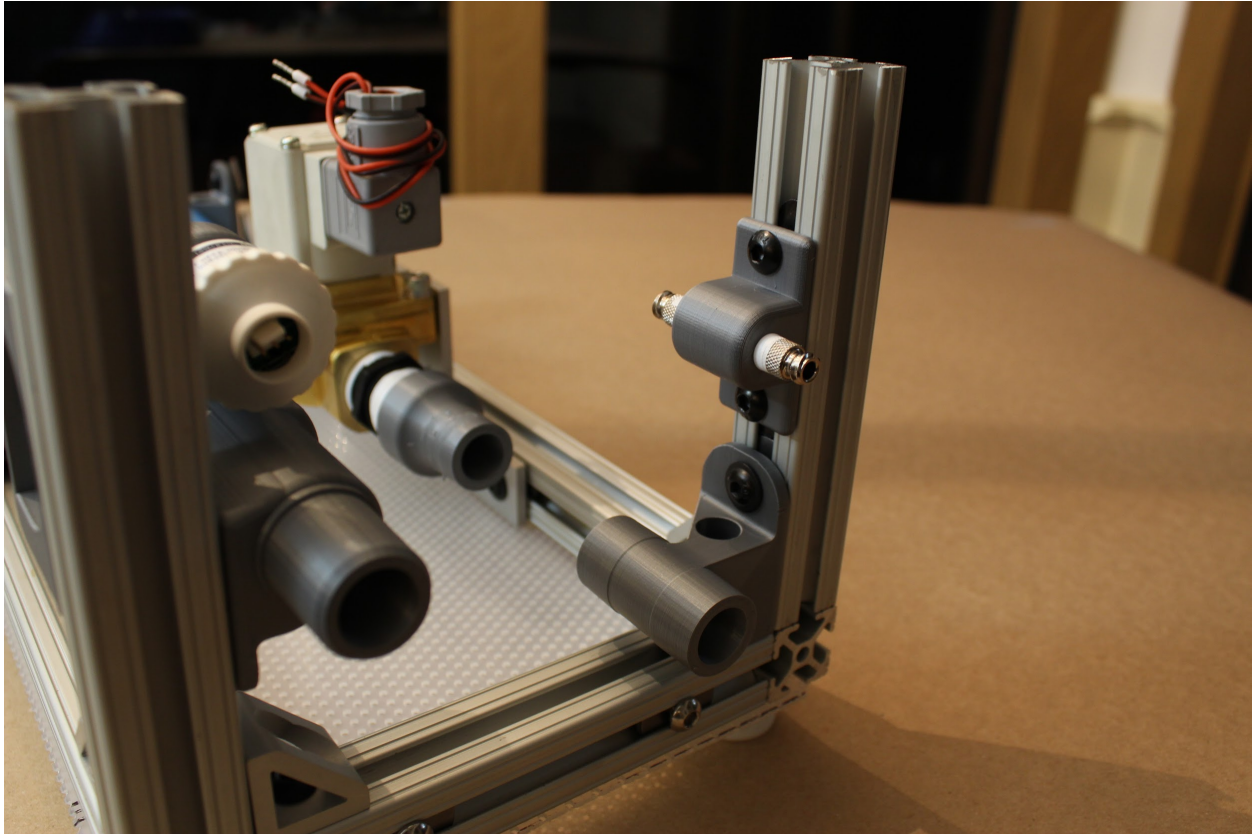
Step 6. Attach the second vertical leg, along the “Expiratory DAR filter bracket”.

As before, slide a short (5 ") 80/20 piece into the position shown, then tighten the hex screws.



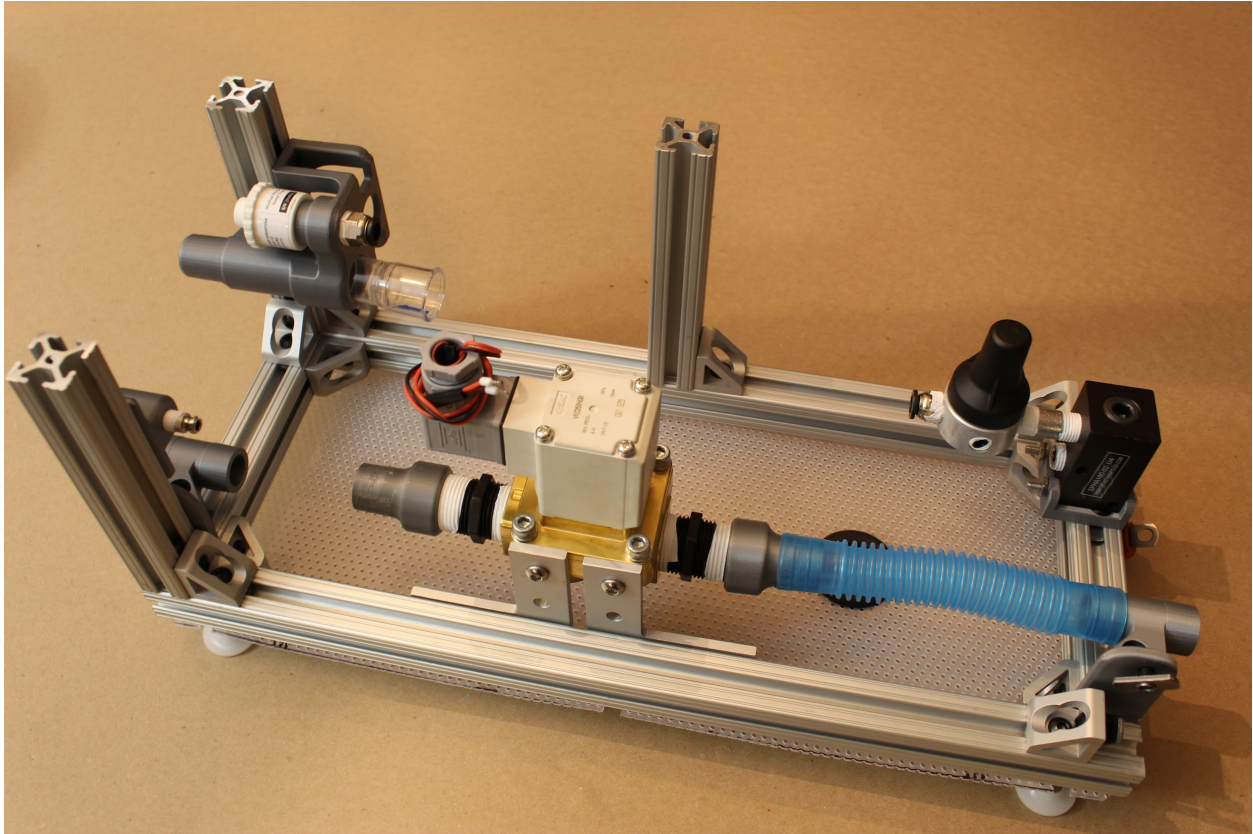
Step 7. Attach the luer lock filter bracket to the newly inserted 80/20 leg.

***Note:** If you're attaching the side panels, use another SHORT 80/20 nut in the uppermost spot on the bracket. We'll want this part to be as high as possible to match up with the holes on the front panel.



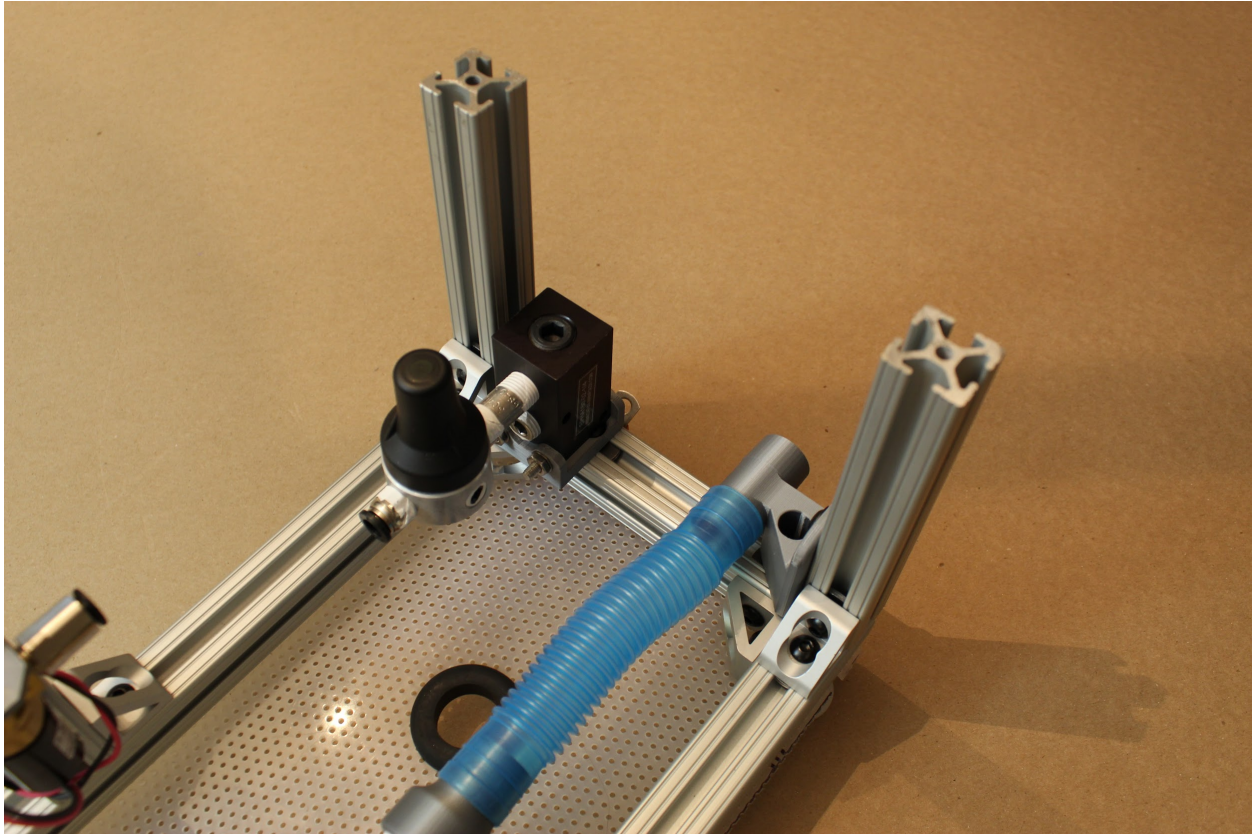
Step 8. Attach the third vertical 80/20 leg, then attach the proportional valve mount.

Insert a third, short (5 ") 80/20 piece, then tighten the hex screw to hold it in place. Use another standard hex screw/nut to attach the proportional valve mount to this piece, towards the inside of the device, such that the push-to-connects roughly align vertically with the push-to-connect on the sensor atrium.

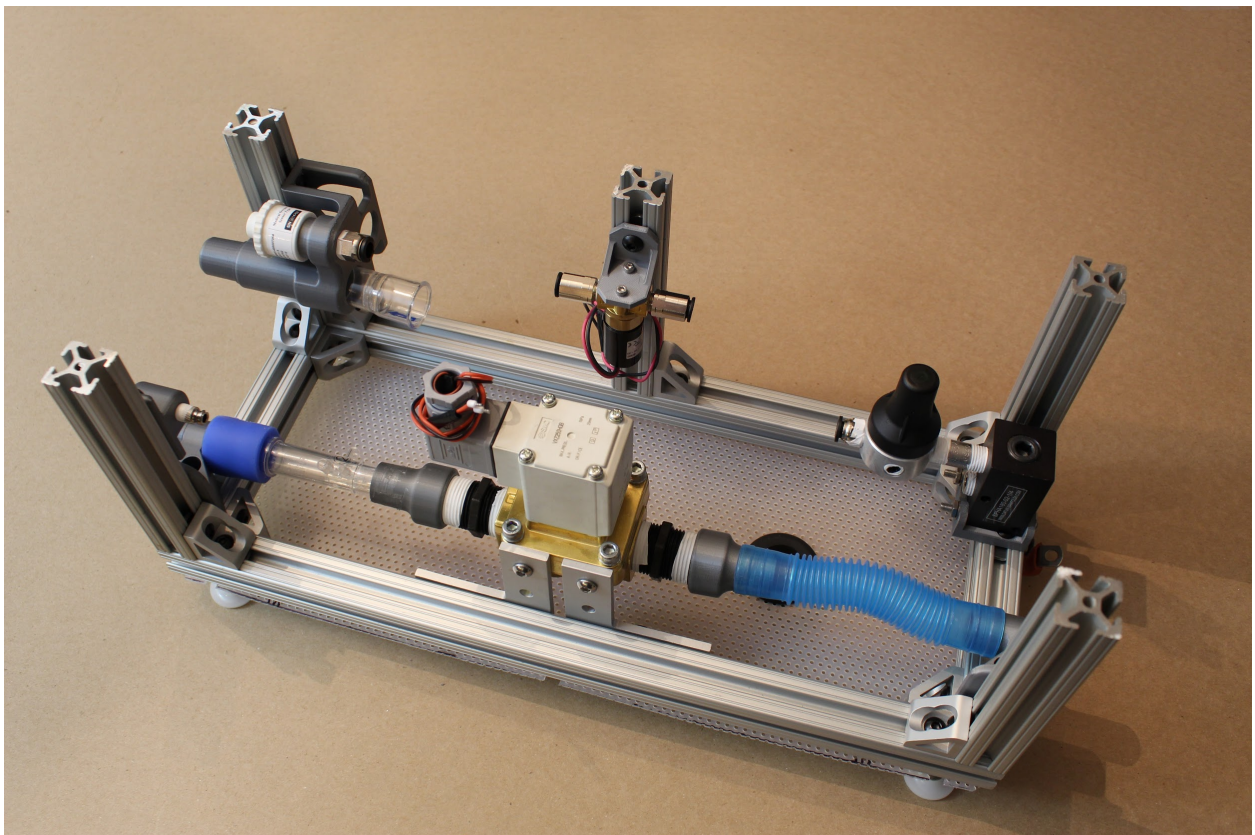
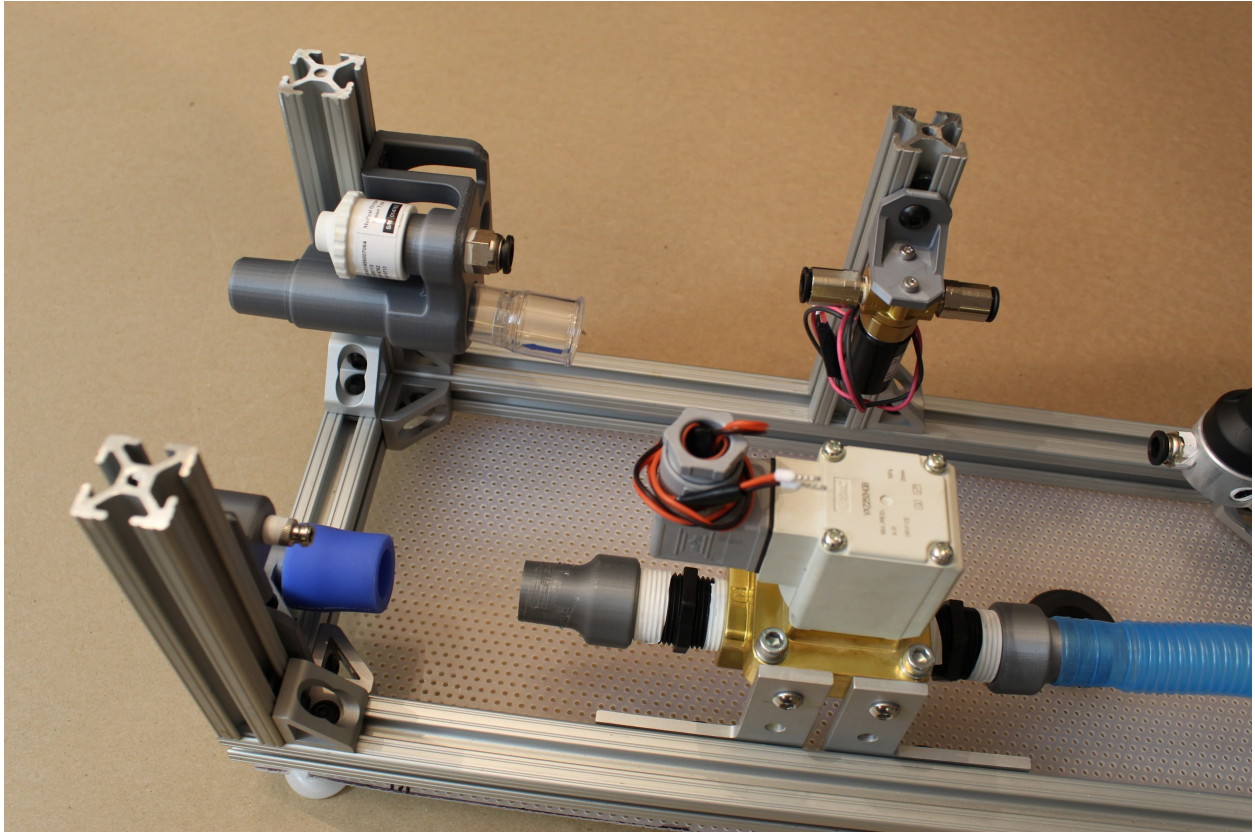


Step 9. Attach the final two vertical 80/20 pieces in the remaining corners.

***Note:** If you're attaching the side panels, now is the time to insert 80/20 nuts on the vertical 80/20 pieces as well as the long 80/20 pieces on the lower frame.

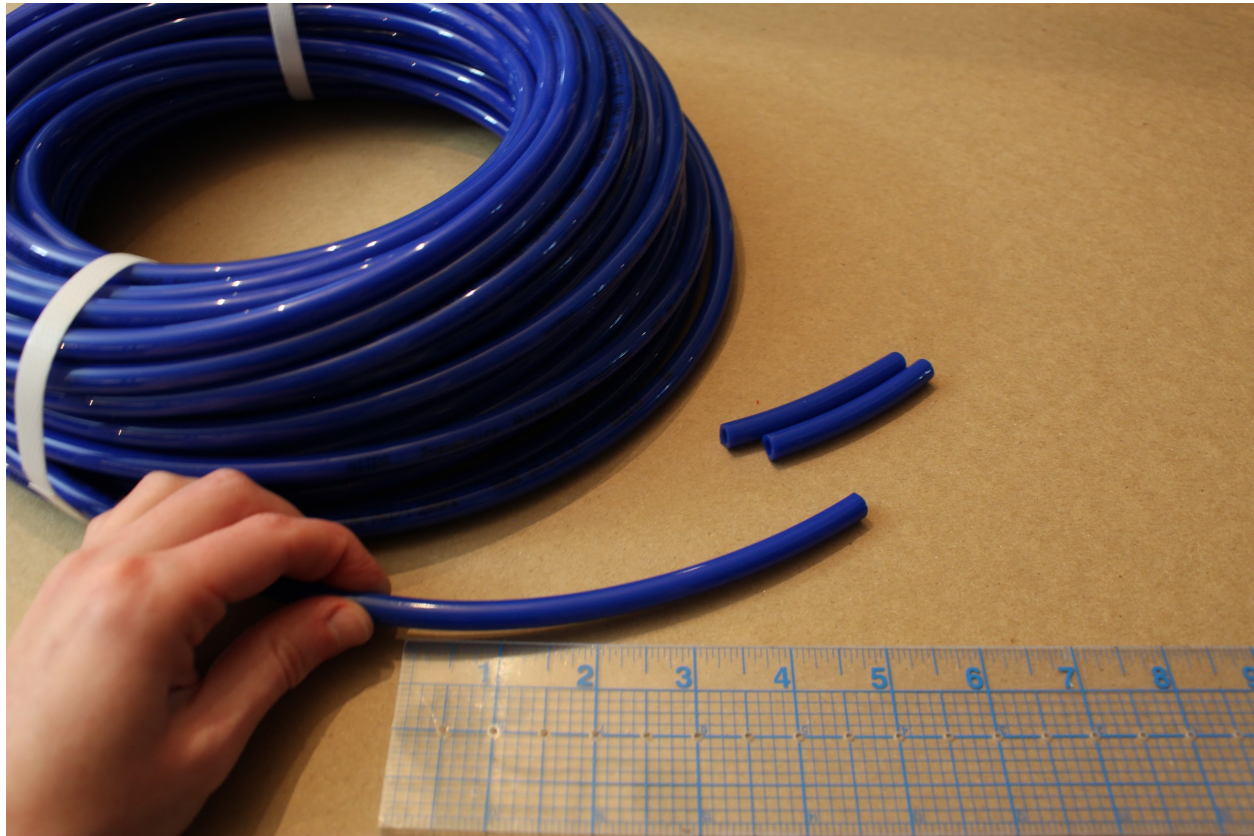
**Step 10. Attach the D-Lite.**

Attach a blue silicone connector to the “Expiratory DAR filter bracket), on the side within the device. Then, insert the D-Lite between the silicone connector and the nearest “22mm to .75 NPTM adapter”, adjusting the position of the expiratory solenoid assembly until the D-lite is firmly connected at each end. The smaller end of the D-lite should fit within the “22mm to .75 NPTM adapter”. Then, use an Allen key to tighten the hex screws on the 90 degree angle brackets to maintain the position of the solenoid.

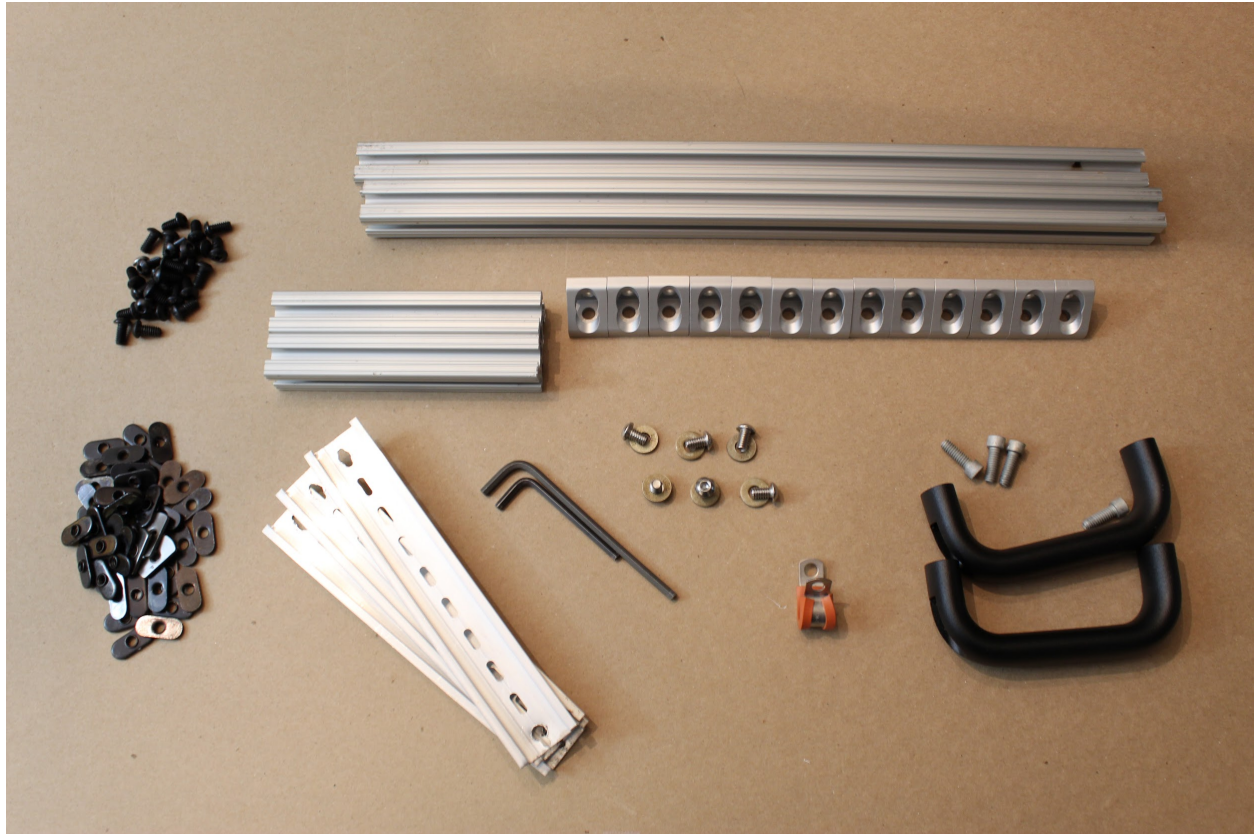


Step 11. Attach the pressure release valve and pneumatic tubing.

Cut the pneumatic tubing into three segments: two of length 2.5" (6.35cm), and one of length 5.5" (13.97cm). Insert the long piece between the push-to-connect adapters attached to the pressure regulator and proportional valve. Insert the two shorter tubes into either push-to-connect on the pressure relief valve assembly; then attach these between the sensor atrium and proportional valve push-to-connects, as shown. These tubing lengths should keep the proportional valve fairly centered.

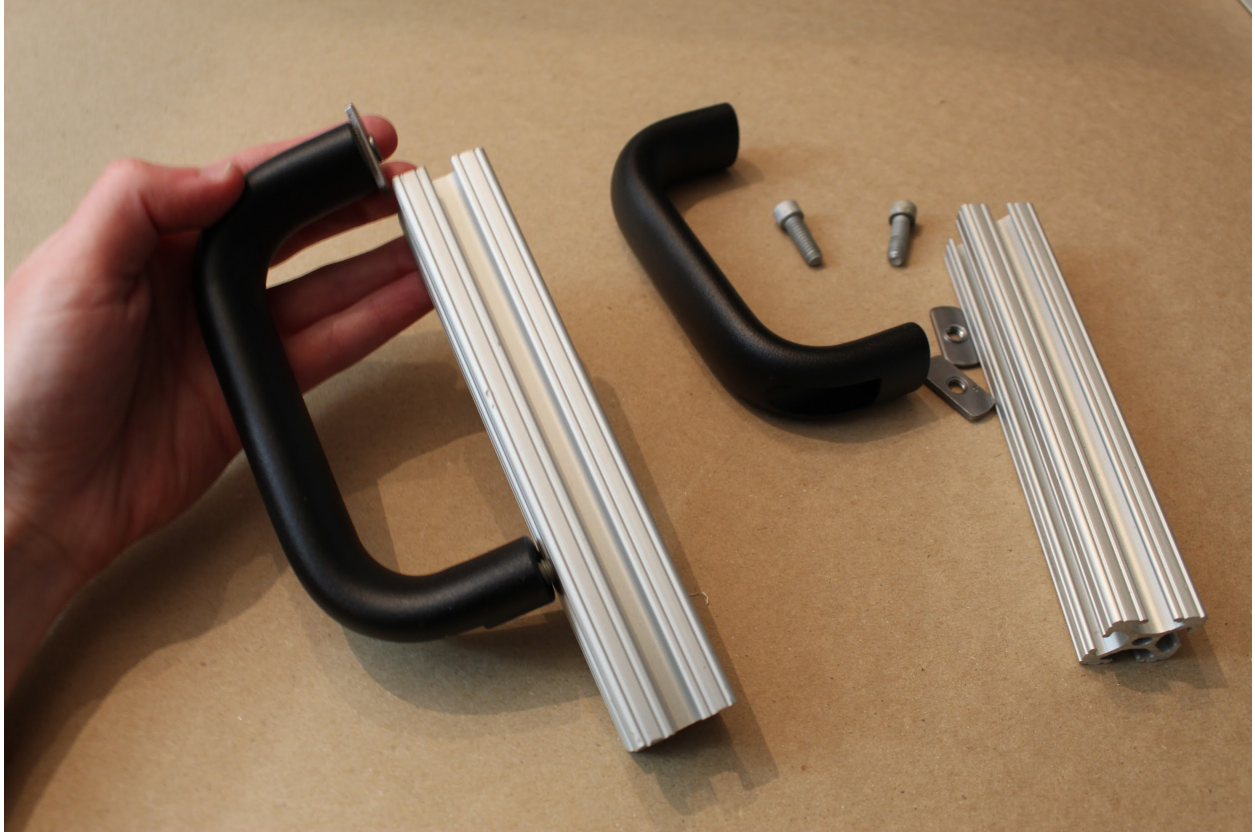


2.2 Assembling the frame top



Step 1. Attach the lifting handles.

Use the lifting handle screws (SHCS_0.25-20x0.75_Gr8_ASTM_F1136), with standard hex nuts, to mount the lifting handles to two short (5 ") pieces of 80/20, such that the handles are centered on the pieces.



Step 2. Prepare all the remaining gusset (corner) brackets.

Attach the hex screws/nuts loosely to the 13 remaining gusset (corner) brackets, so that they can be attached readily in future steps.

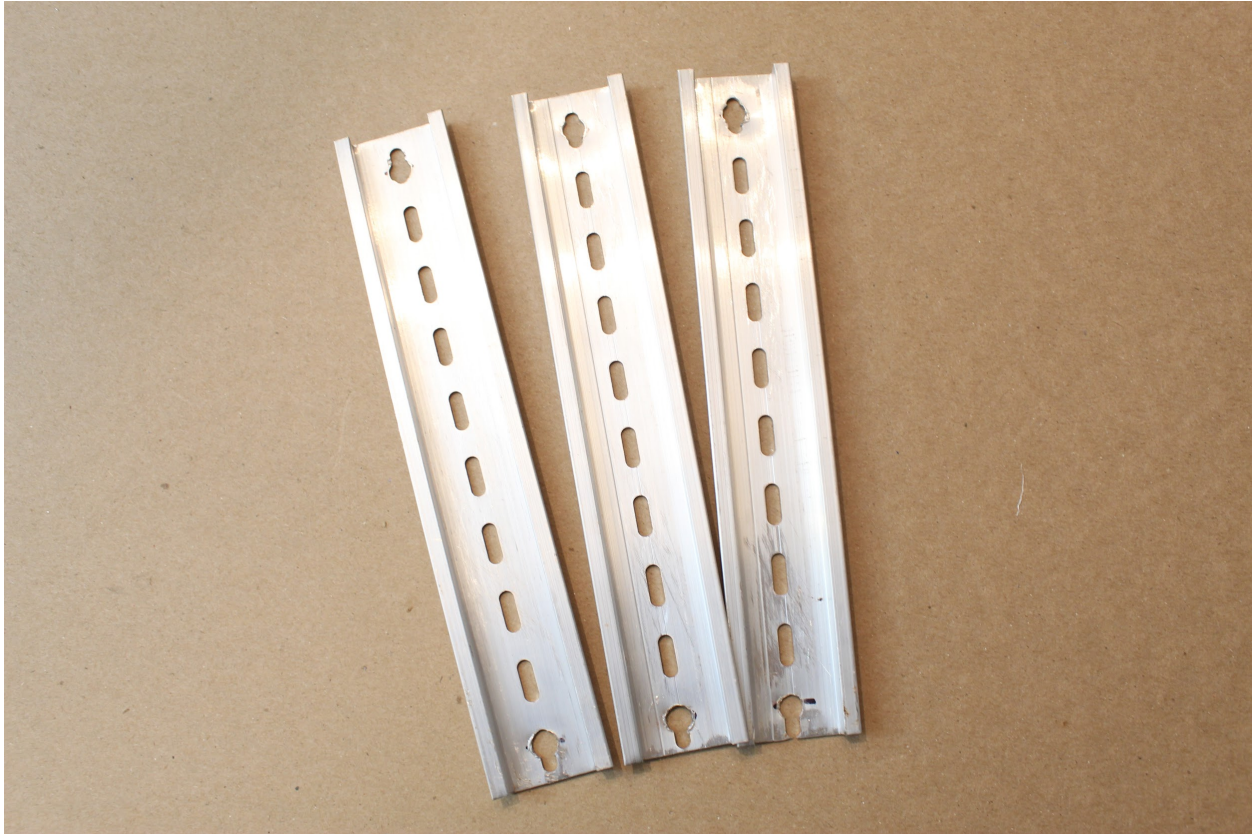


Step 3. Attach gusset (corner) brackets, as shown.



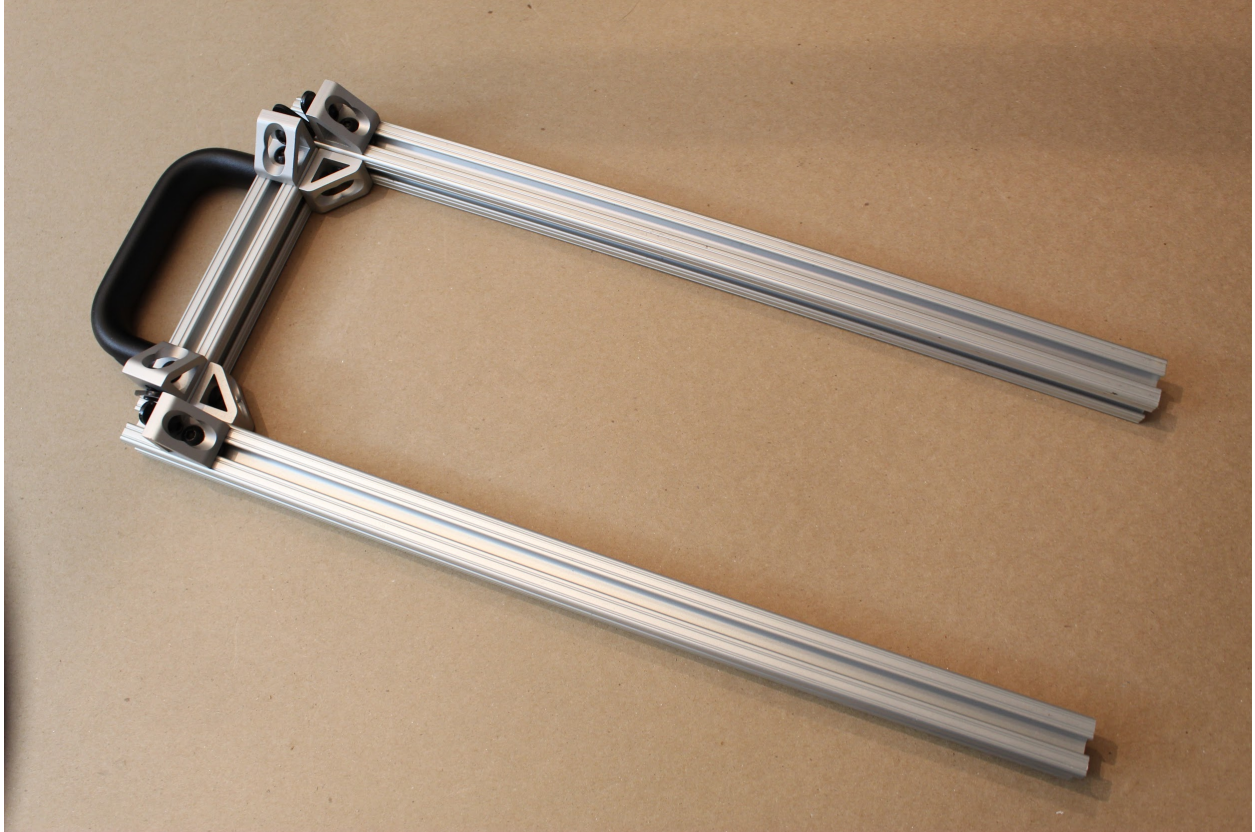
Step 4. Cut and punch DIN rail pieces.

Cut the DIN rail to 7 " (20.0025cm) lengths, punching or drilling holes ½" (1.27cm) from each edge large enough to support the standard button head hex screws.

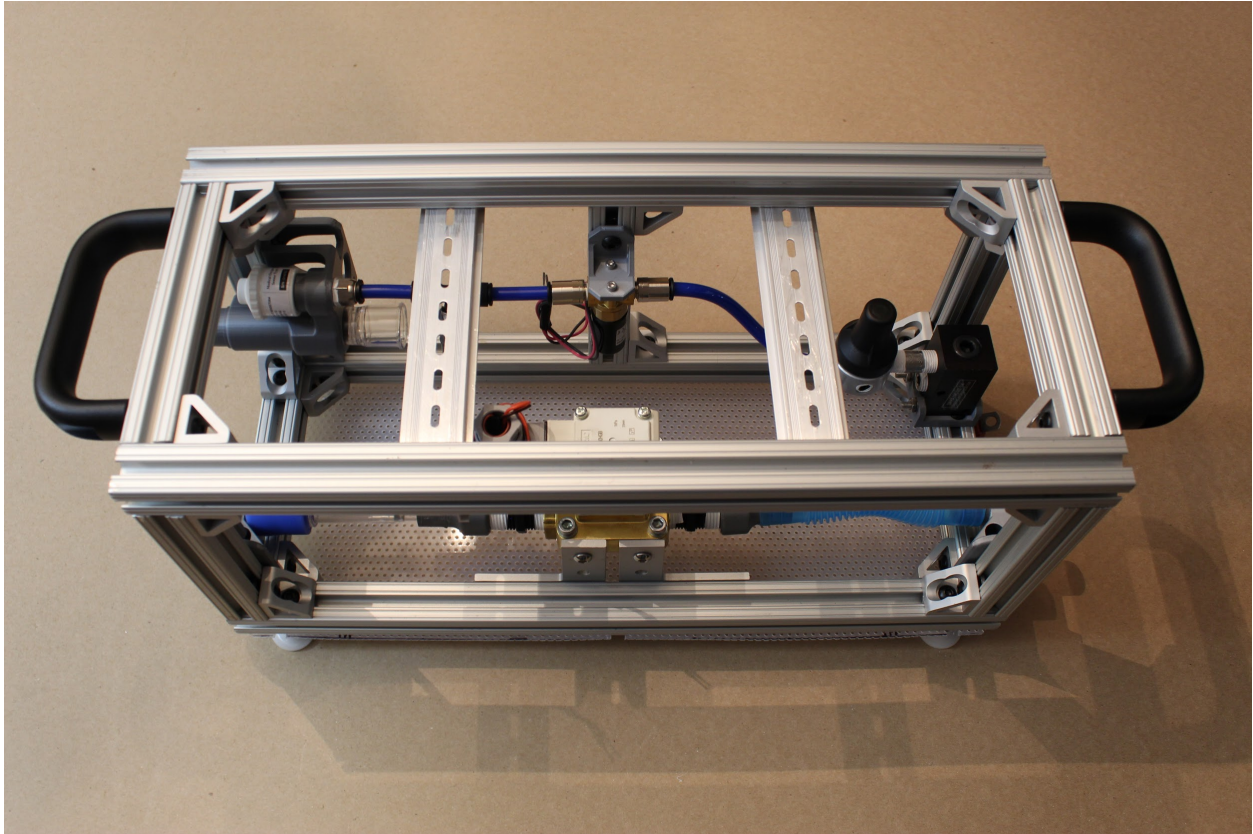


Step 5. Attach the DIN rails and assemble the rest of the top level.

- Slide two long (17 ") 80/20 pieces onto one of the shorter pieces, and screw in place. Add an additional gusset (corner) bracket to each long leg, as shown, leaving 1" of space from the end (to support a vertical piece).
- Use a short (1mm) button head hex screw, a zinc washer (W_0.25_FLAT_THICK_GR8_YELLOW_ZINC), and a standard hex nut to attach each end of the DIN rails to the frame, as shown. Slide an additional gusset (corner) bracket between the rails on one side- this will support the central vertical channel on the device.
- Once the DIN rails are roughly in place (exact positions can be adjusted later), use the remaining gusset (corner) brackets and assembled short 80/20 piece, and mirror the other side of the assembly, as shown.



Step 6. Finally, slide the top frame directly into the vertical 80/20 channels of the device, and tighten all hex screws.

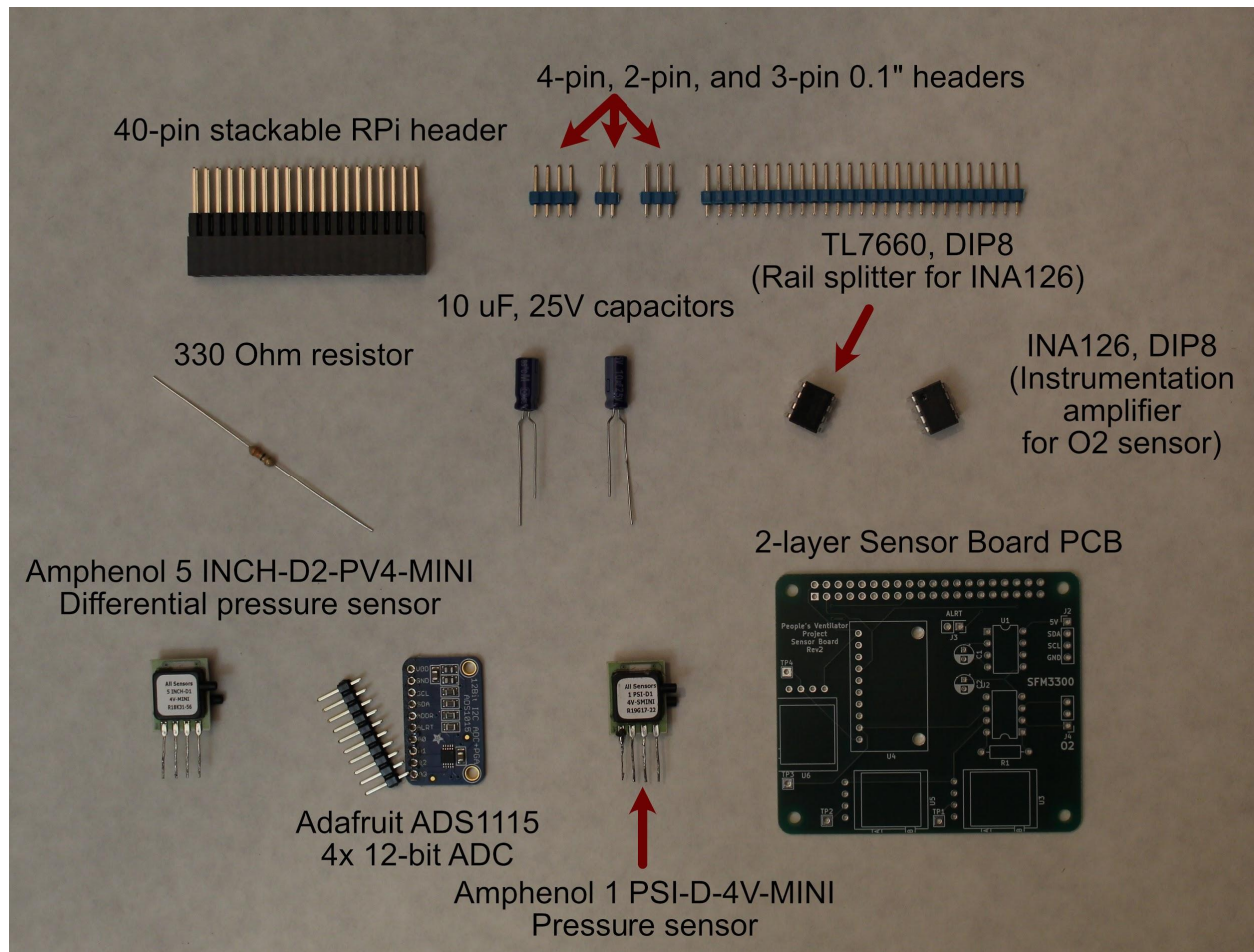


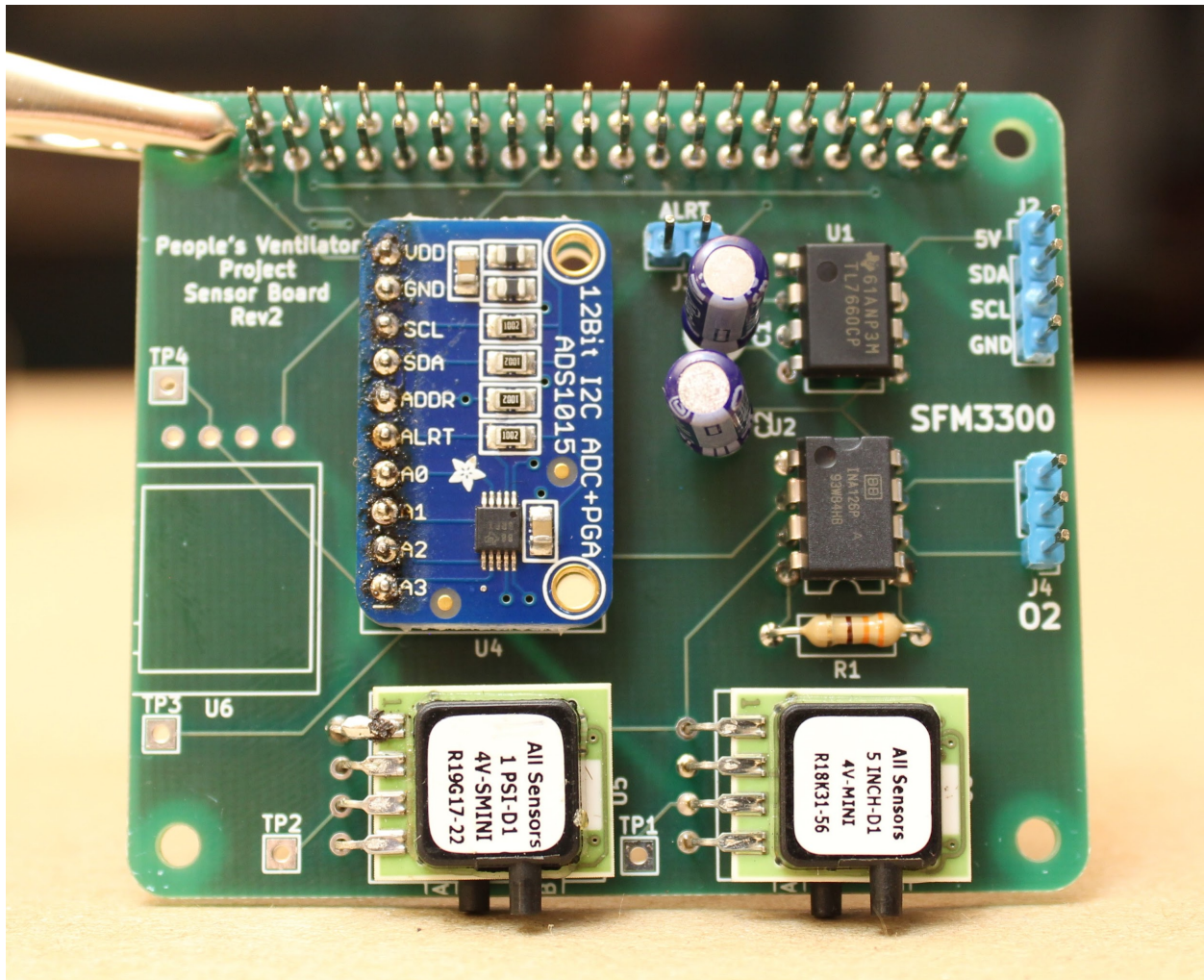
Congrats! The basic hardware assembly is complete!

1.1.7.3 Part 3. Electronics Assembly

3.1 Assembling the Sensor Board

[Image of all components laid out in order/piles, with labels]



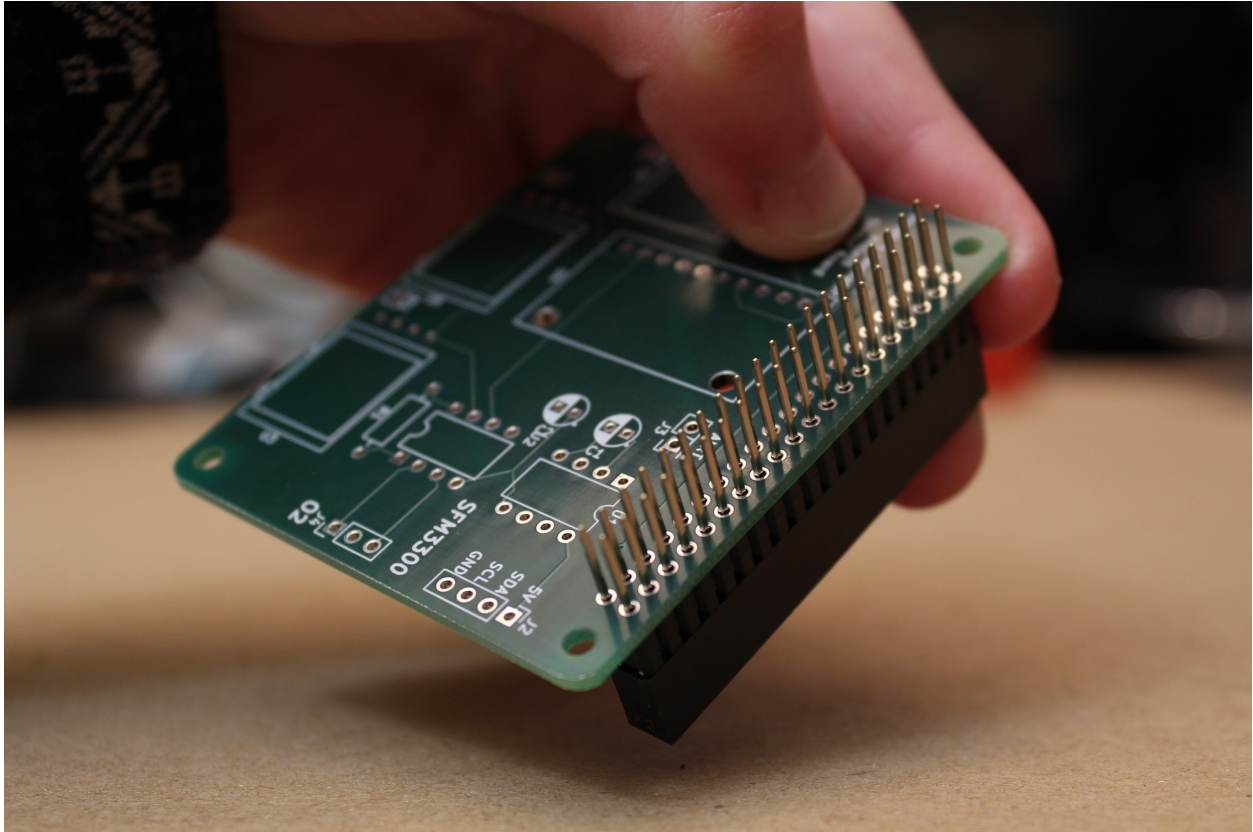


You will also need:

- A soldering iron
- Solder
- Helping hands for holding parts while soldering
- Wire cutters (for clipping off long capacitor/resistor legs)

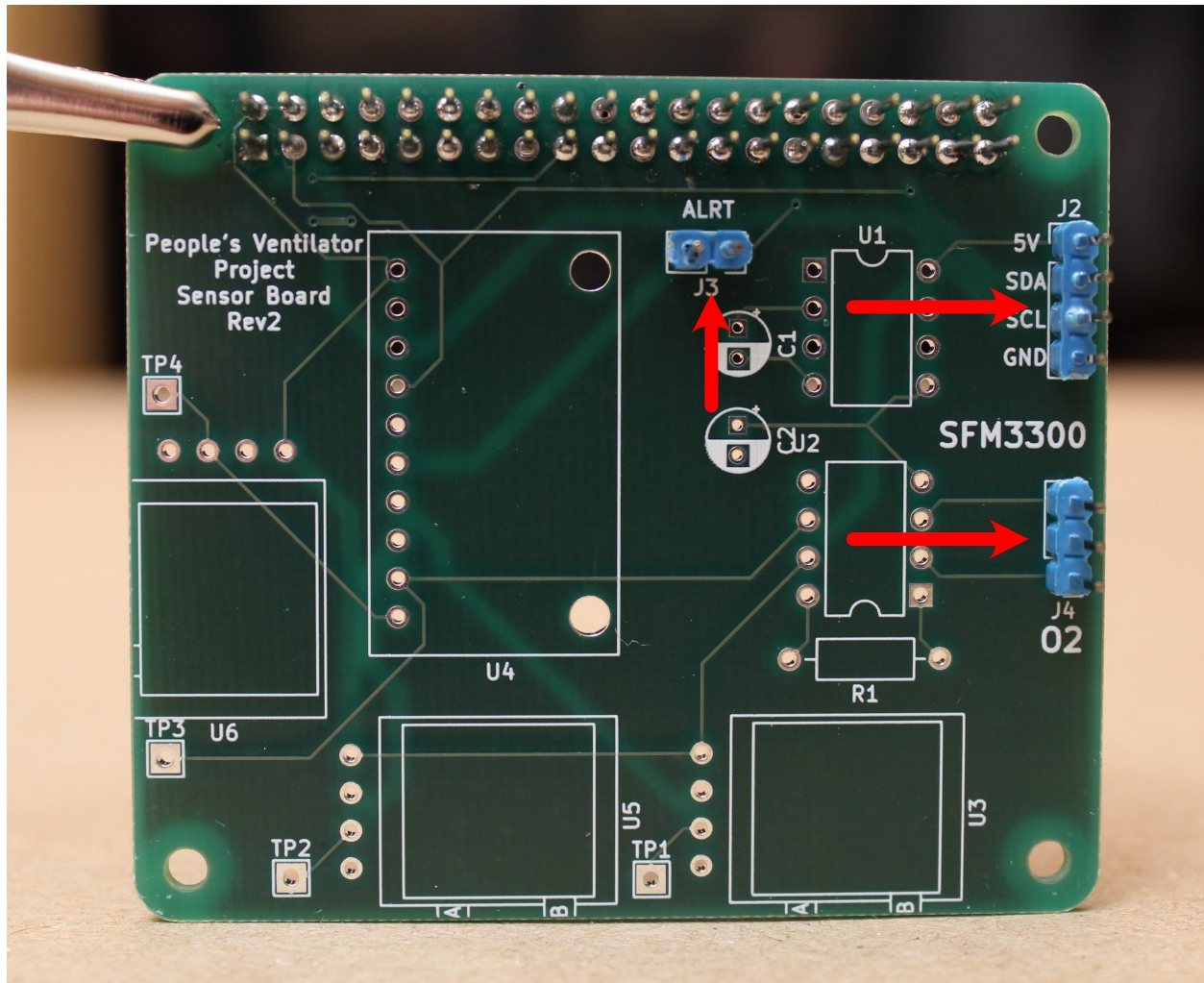
Step 1. Solder the 40-pin stackable RPi header to the Actuator PCB.

Push the pins up from the bottom of the board, as shown, and then solder into place.



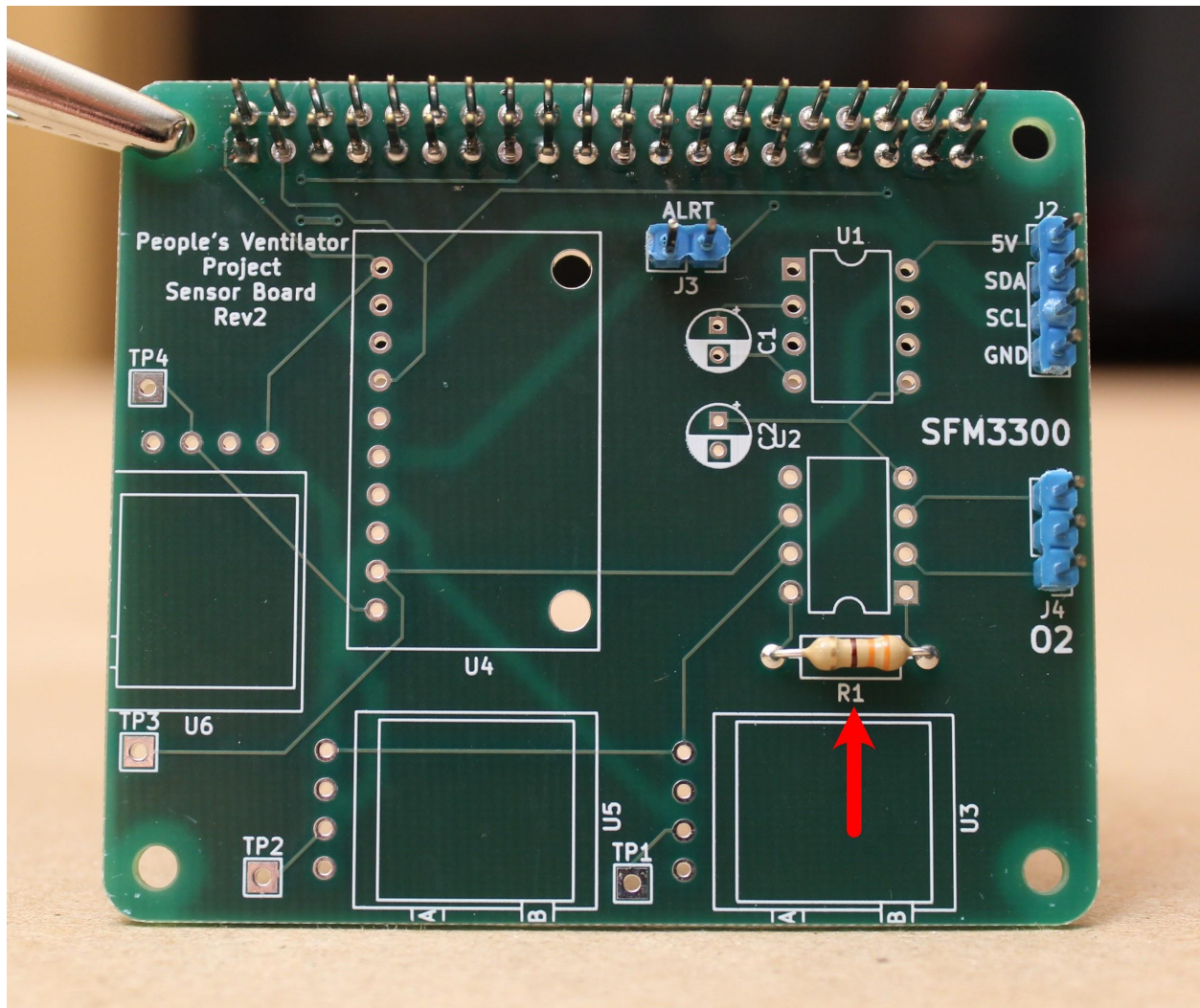
Step 2. Solder the 4-pin, 3-pin, and 2-pin 0.1" headers onto the board (positions J2, J3, J4).

Break the pins off the larger header array in units of 4, 2, and 3 using pliers. Insert the short end of the pins into the holes from the top of the board, and solder from below. This will leave the long ends of the pins sticking up vertically from the board, as shown:



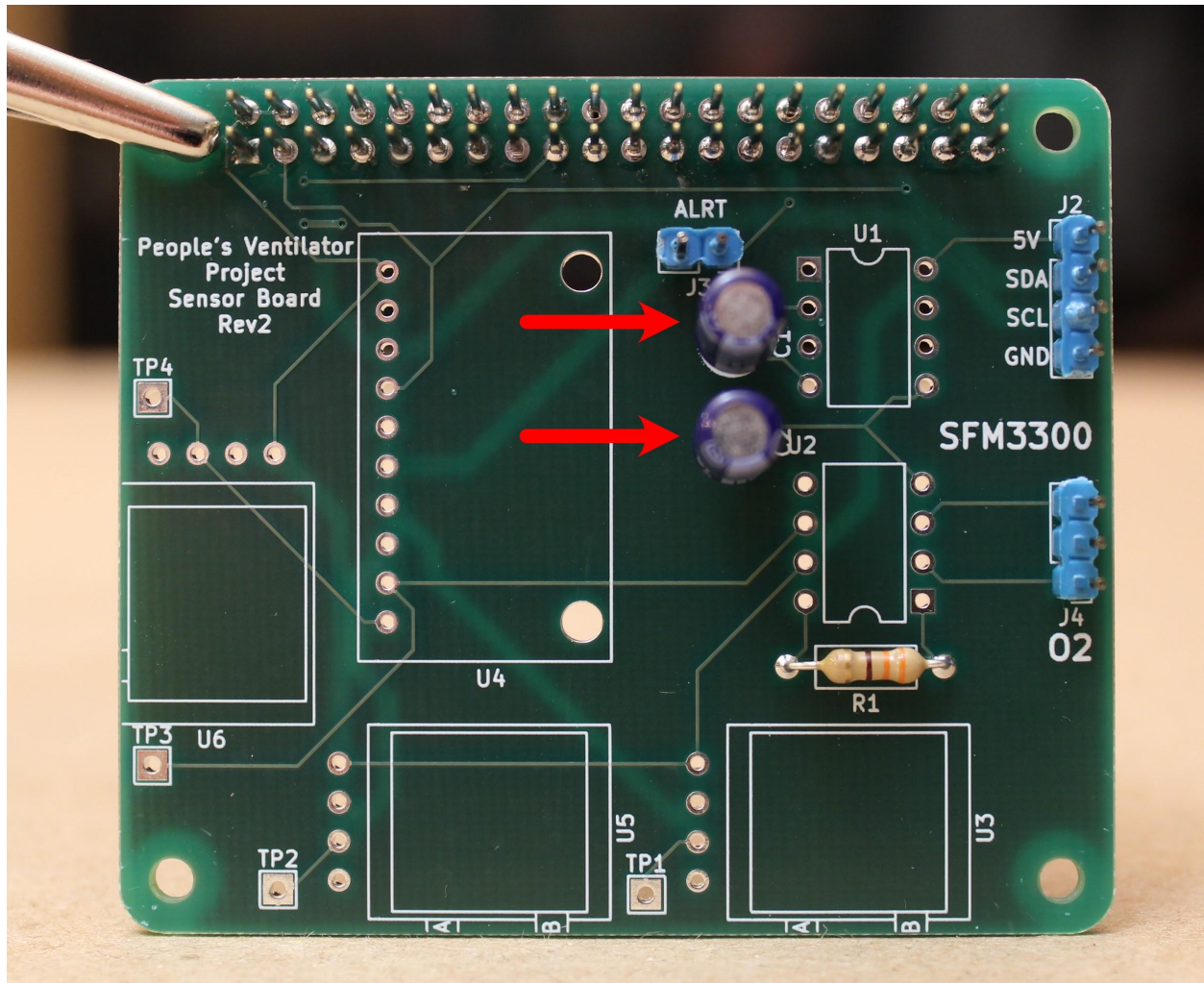
Step 3. Solder the 330 Ohm resistor onto the board (position R1).

Bend the resistor legs before inserting into the board from above. Ideally, solder the resistor in place such that the resistor is hovering just above the board. Then, snip the long legs off from the back of the board (cutting above the tiny “cone” formed by the solder) using wire cutters.



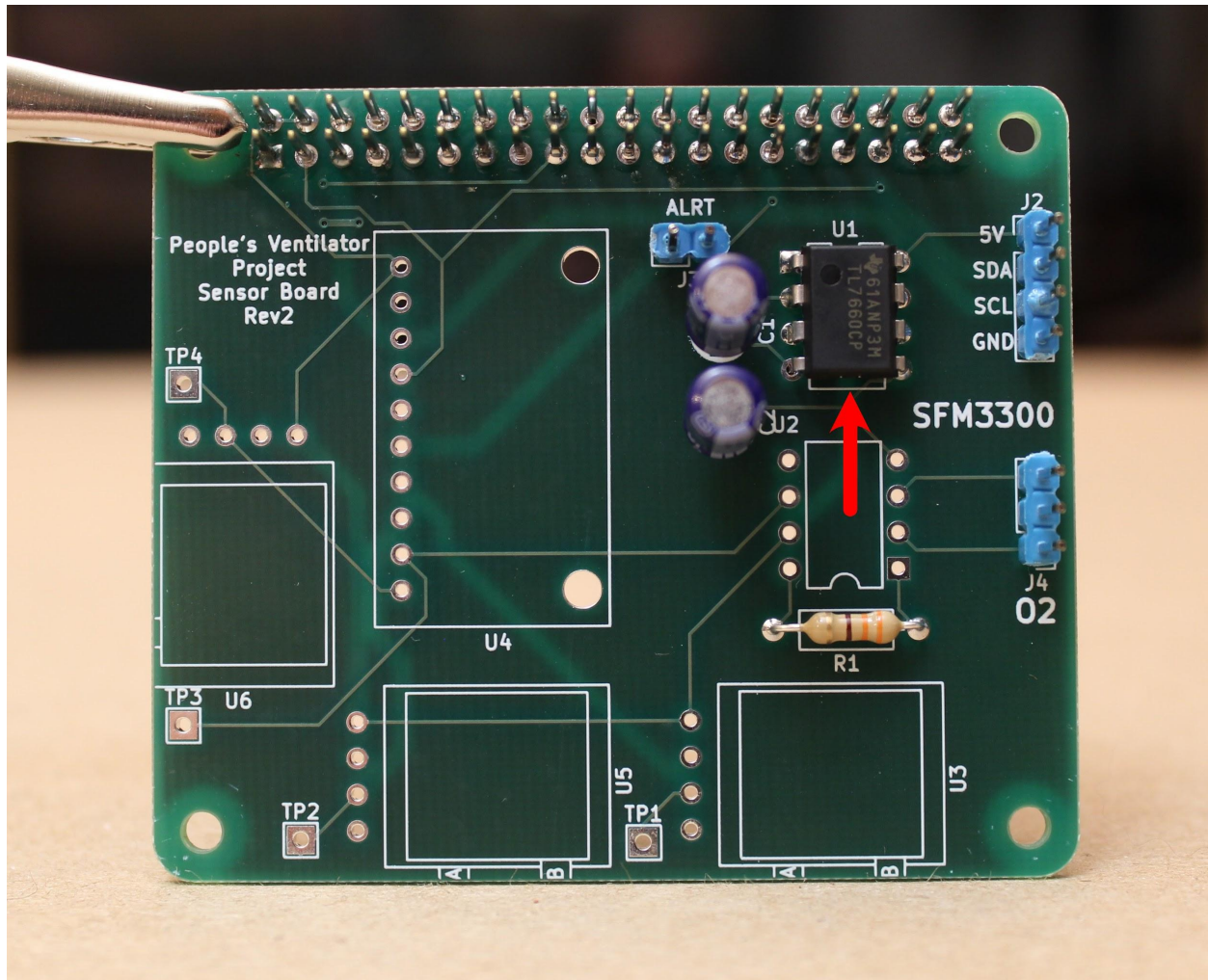
Step 4. Solder the two 10 μ F, 25V capacitors onto the board (positions C1, C2).

Insert the capacitors from the top of the board until the legs snap into place. Be sure that the longer capacitor leg is inserted into the hole corresponding to the “+” sign. After soldering, snip the long legs off from the back of the board using wire cutters.

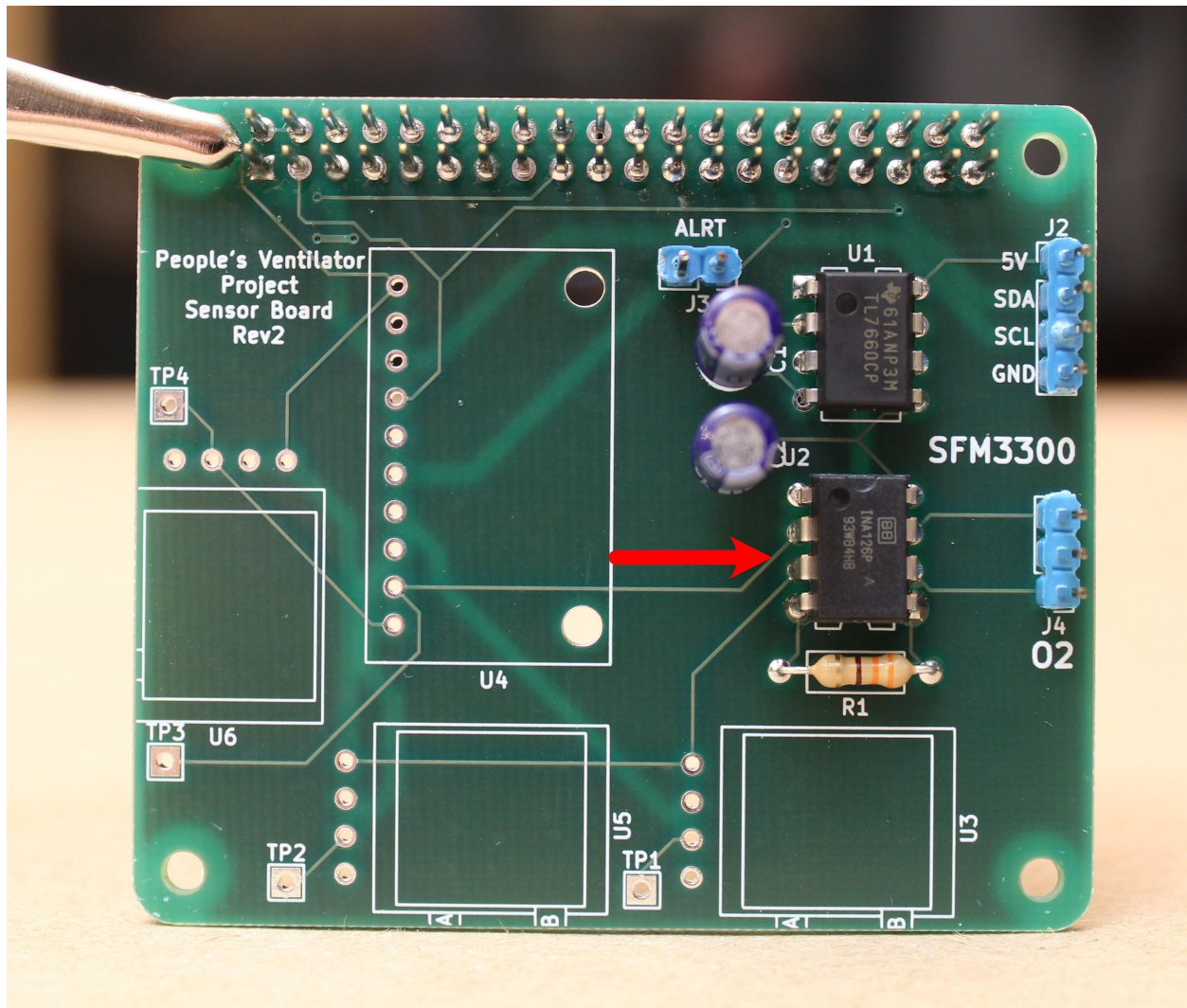


Step 5. Solder the TL7660 (the rail splitter for the INA126) onto the board (position U1).

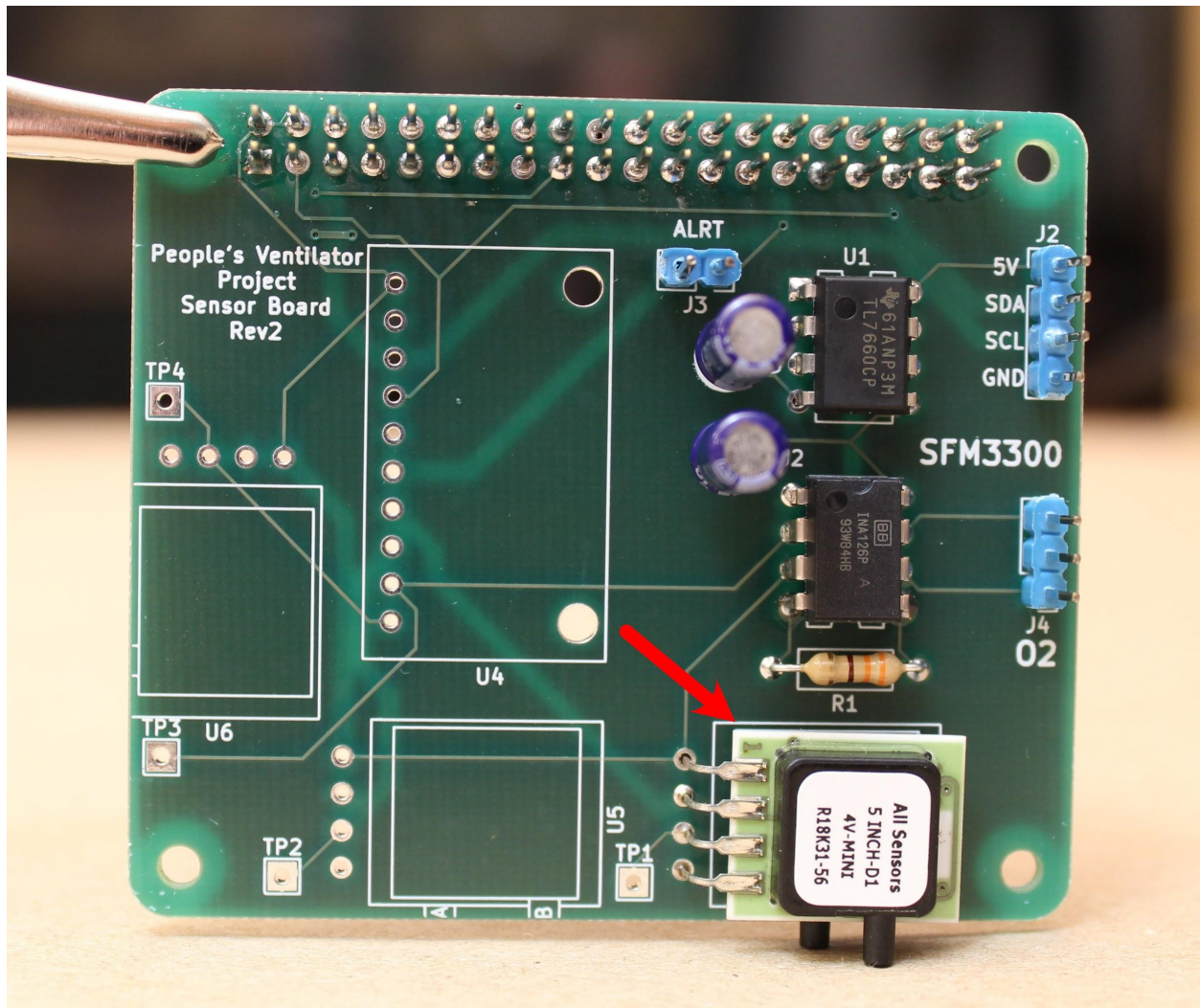
Be sure to orient the small circle on the top of the part to the indicated notch drawn on the board.



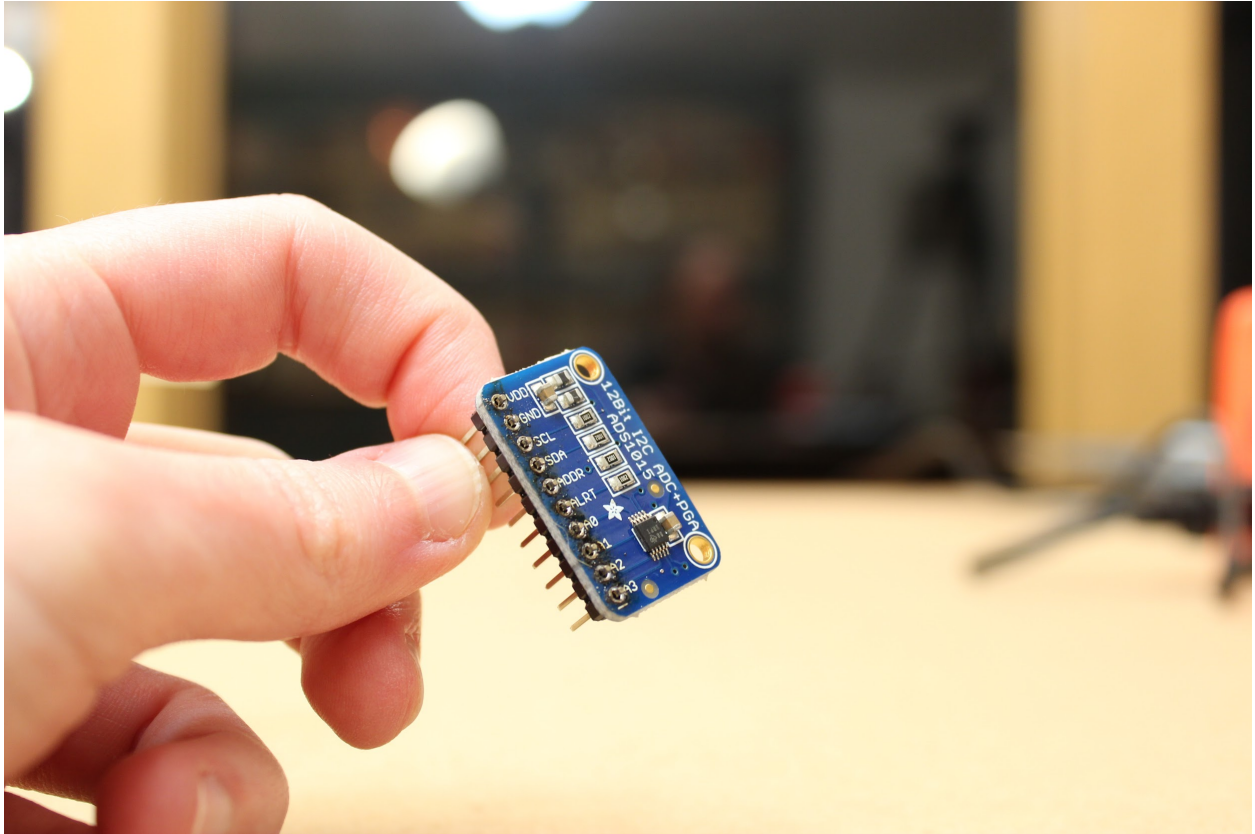
Step 6. Solder the INA126 (the instrumentation amplifier for the oxygen sensor) onto the board (position U2). As before, make sure that the notch on the part aligns with the notch drawn on the PCB.

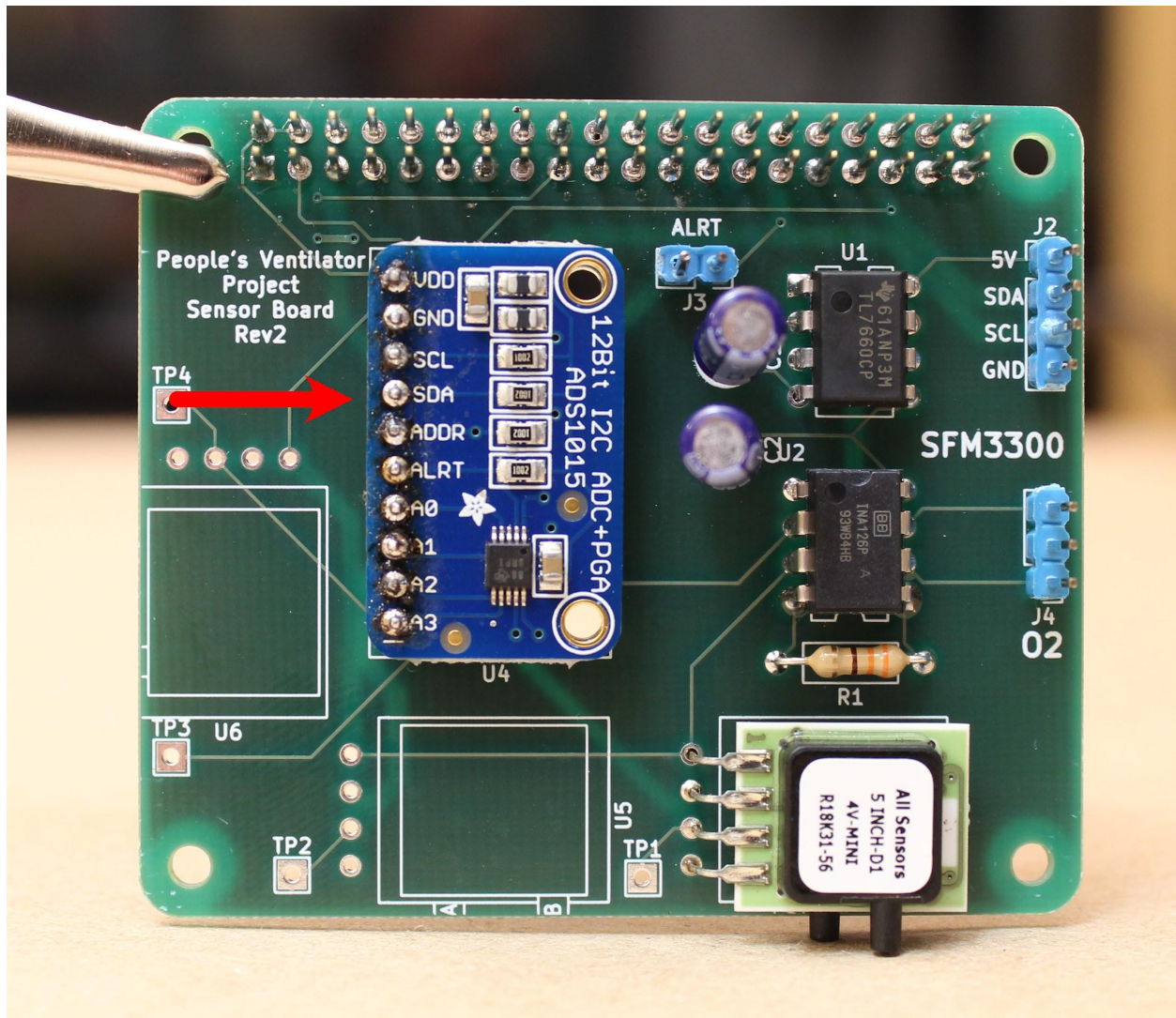


Step 7. Solder the Amphenol 5 INCH-D2-P4V-MINI (differential pressure sensor) onto the board (position U3). Bend all four pins evenly and orient the part such that the black ports face towards the outside of the board. The pressure lines will attach here.

**Step 8. Solder the Adafruit ADS1115 (12-bit ADC) to the board.**

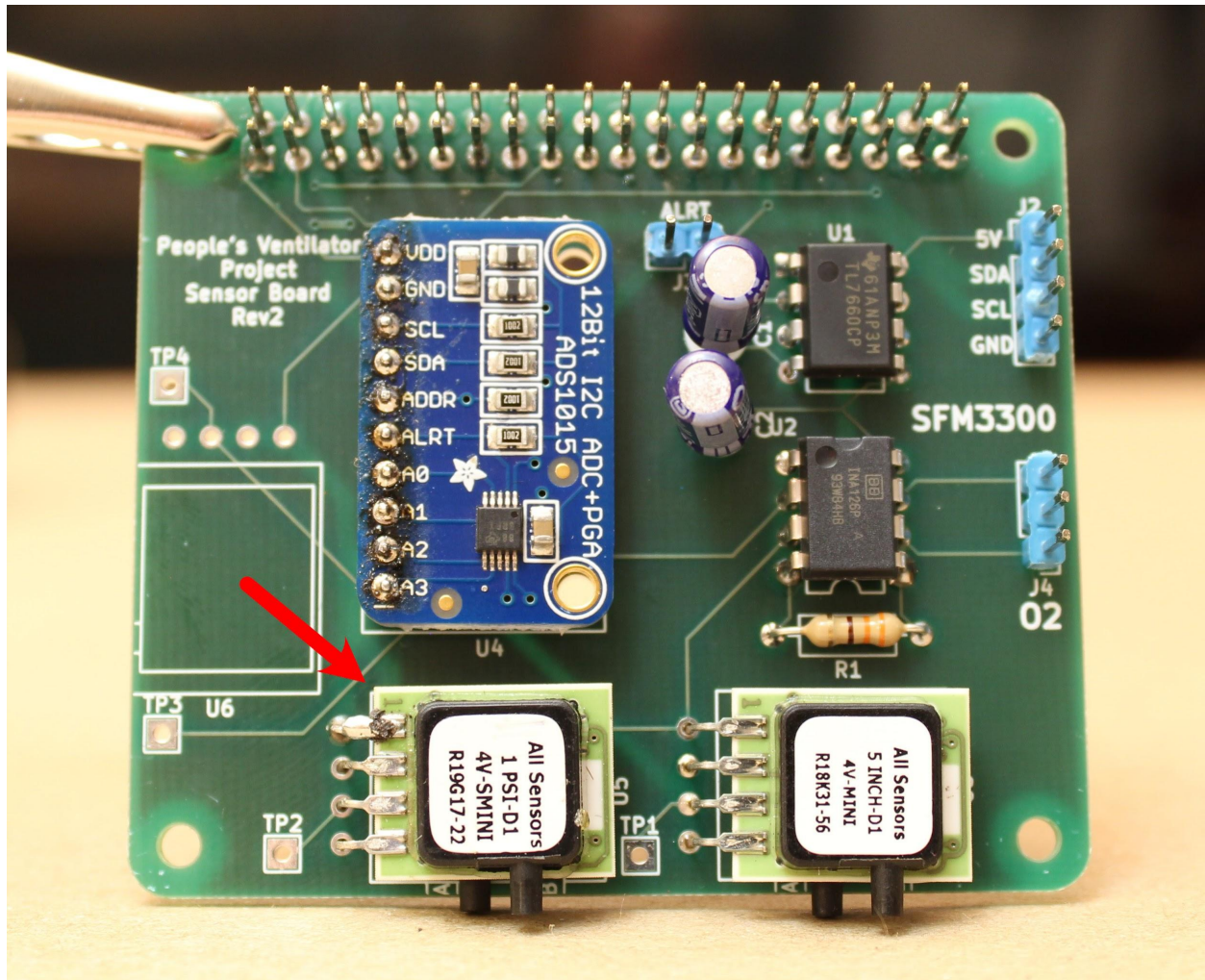
First, solder the pins to the ADC itself, with the long ends facing down. Then insert the pins through the board and solder from the back side.





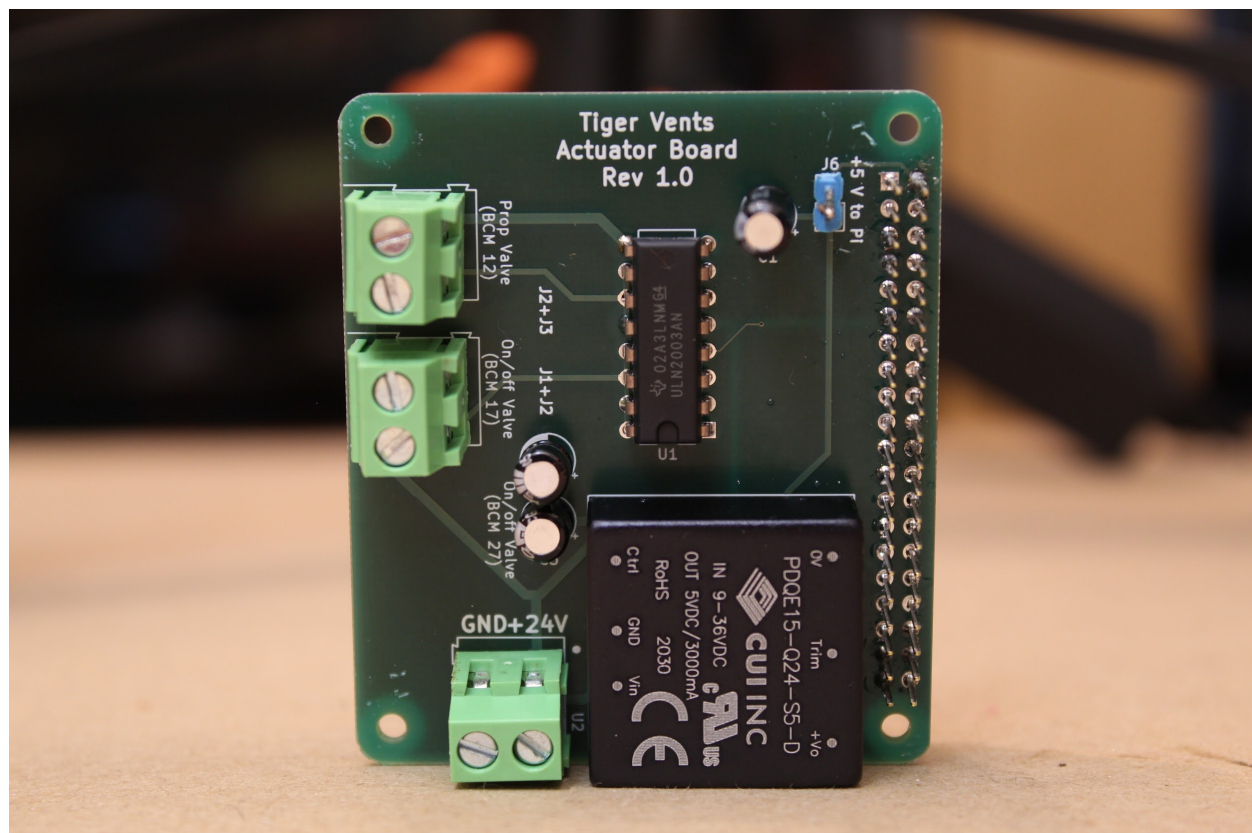
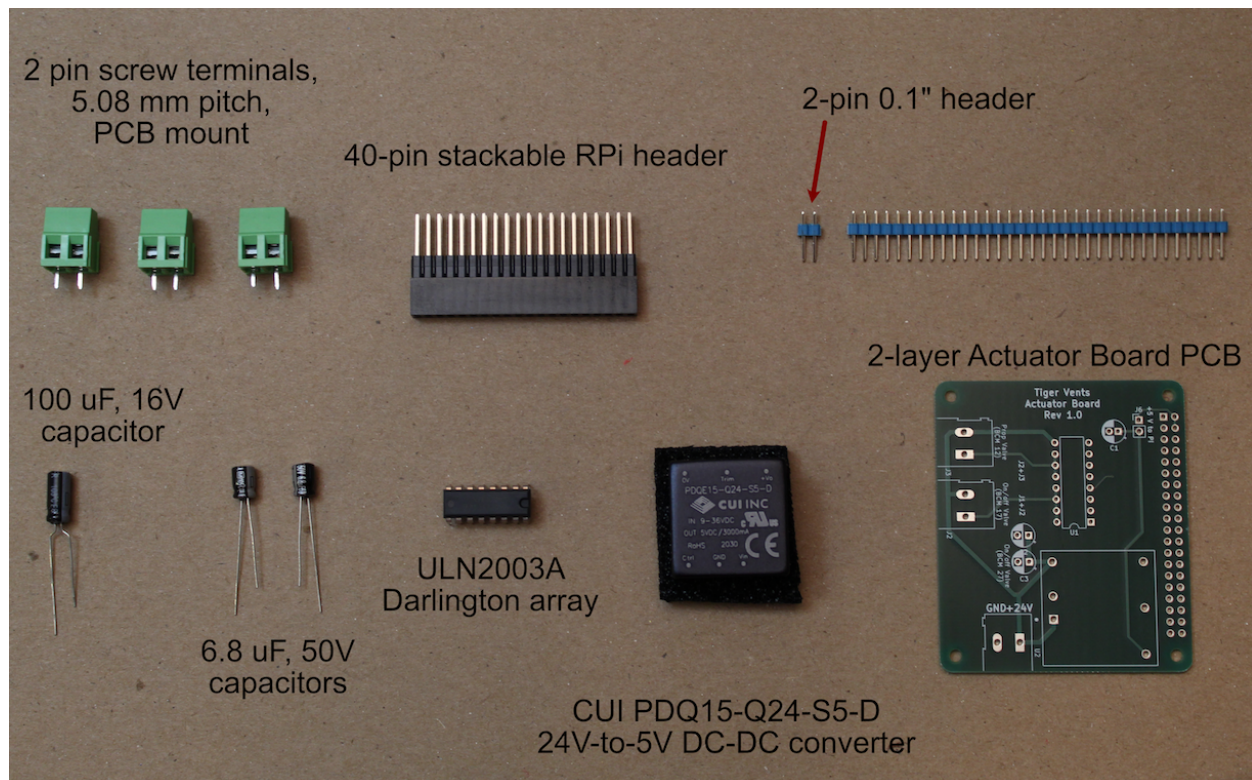
Step 9. Finally, solder the Amphenol 1 PSI-D-4V-MINI (airway pressure sensor) to the board.

As before, bend all four pins together, then insert and solder from the back side of the board. Be sure the ports are facing out: we will attach pressure lines here as well.



3.2 Assembling the Actuator Board

[Image of all components laid out in order/piles, with labels]



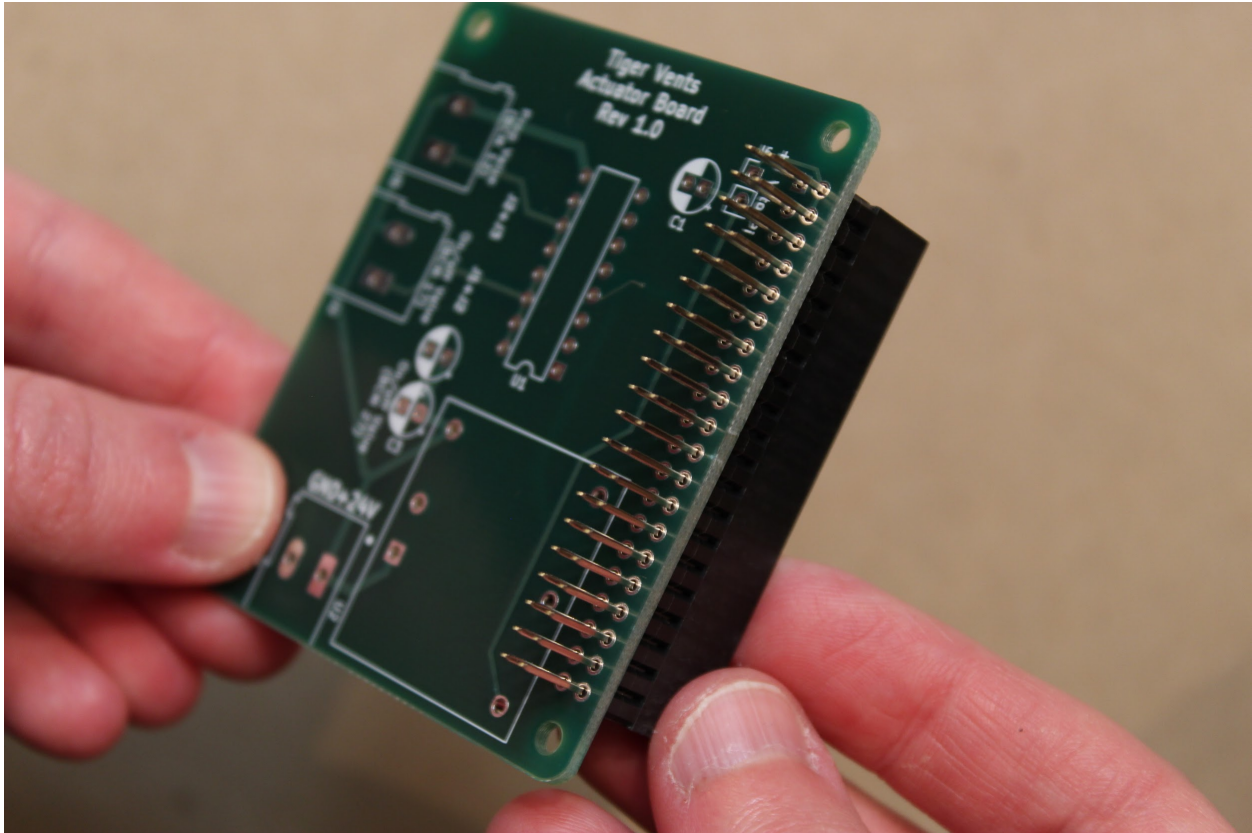
You will also need:

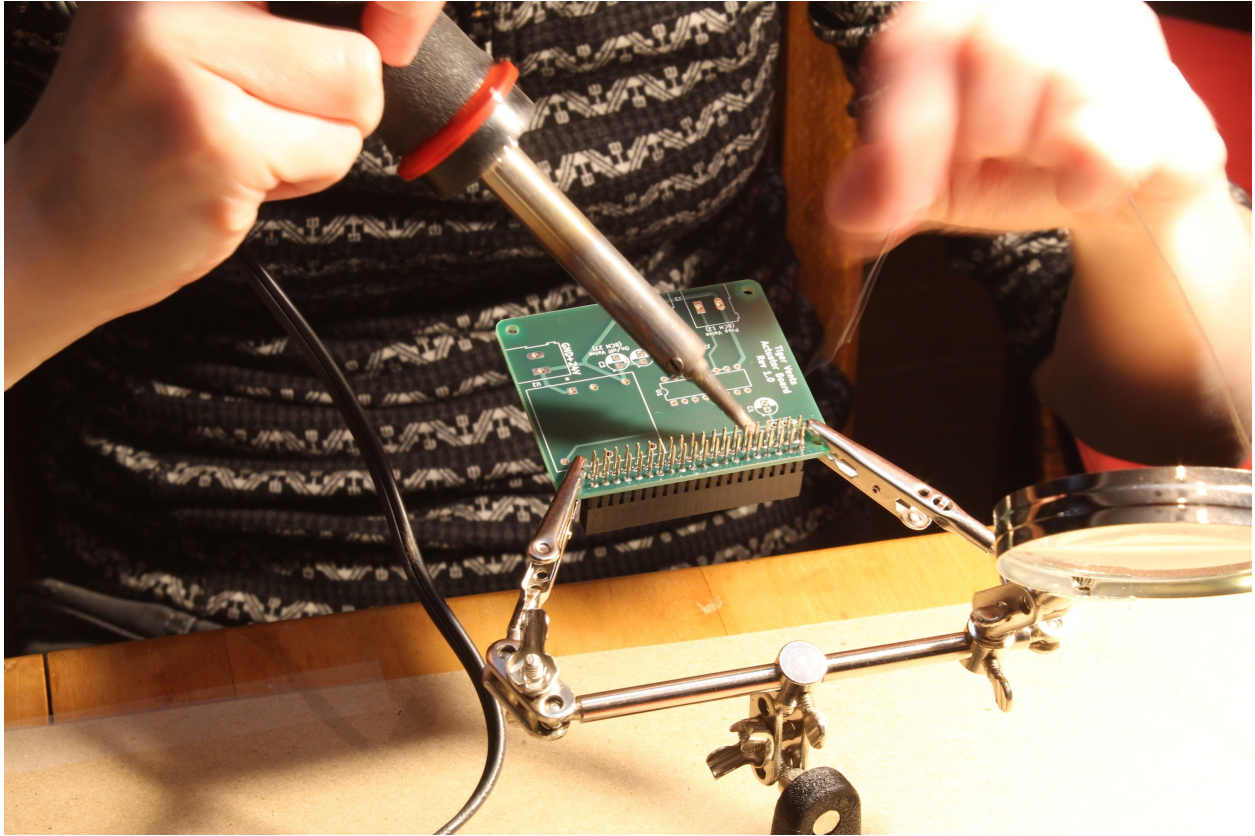
- A soldering iron

- Solder
- Helping hands for holding parts while soldering
- Wire cutters (for clipping off long capacitor legs)

Step 1. Solder the 40-pin stackable RPi header to the Actuator PCB.

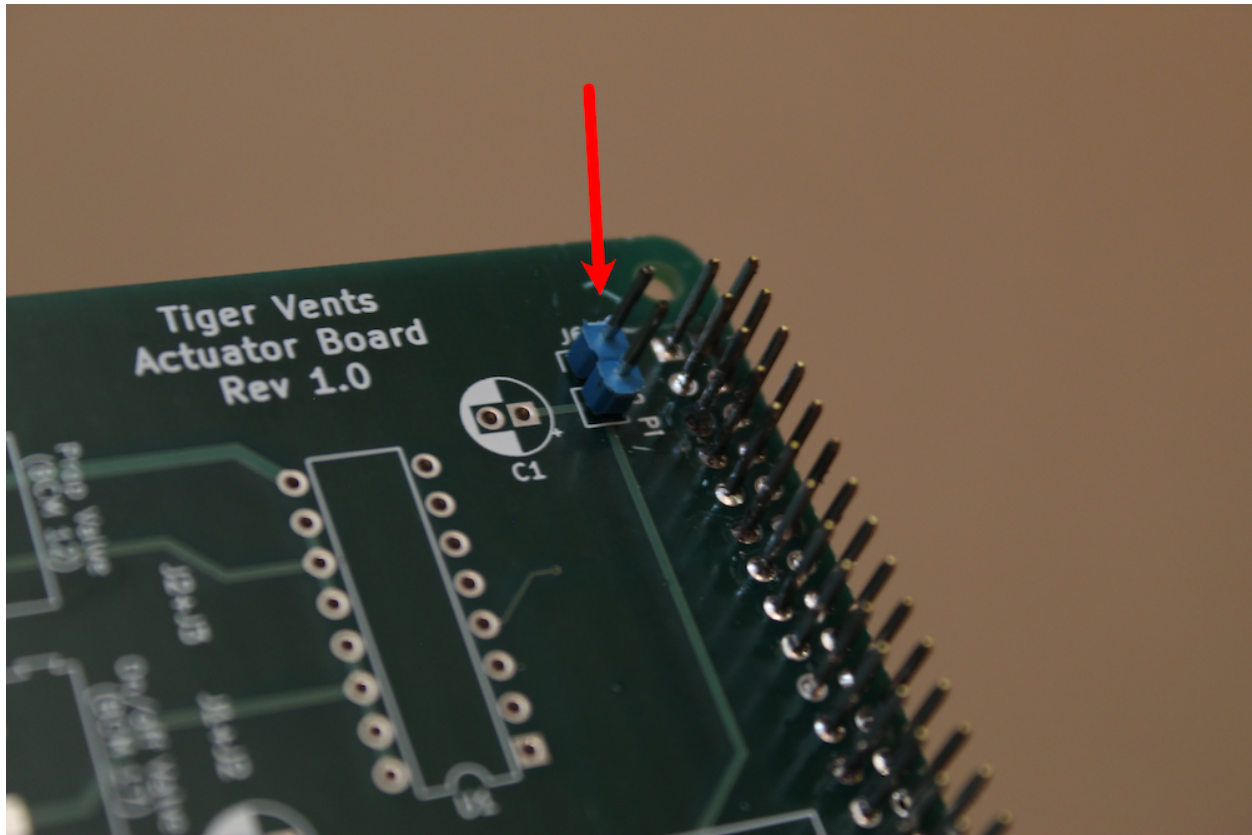
Push the pins up from the bottom of the board, as shown, and then solder into place.





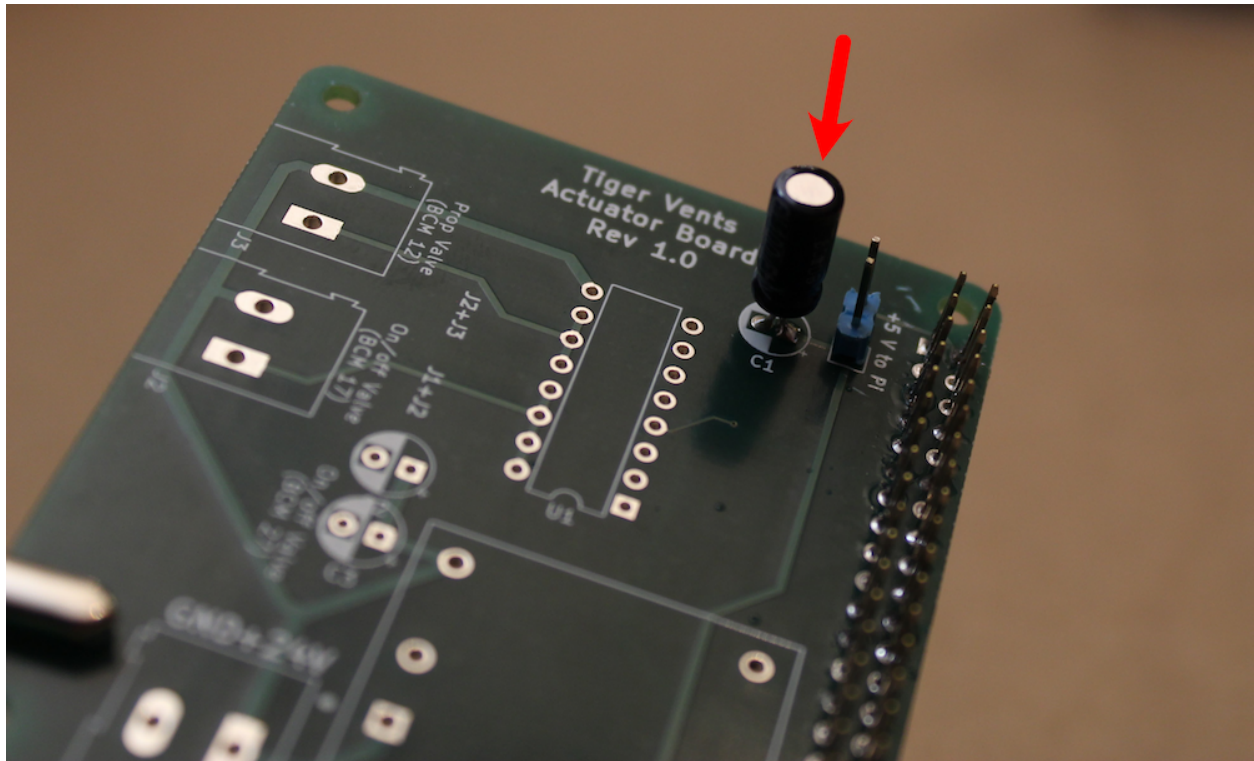
Step 2. Solder the 2-pin 0.1" header onto the board (position J5).

Insert the short end of the pins into the holes from the top of the board, and solder from below. This will leave the long ends of the pins sticking up vertically from the board, as shown:

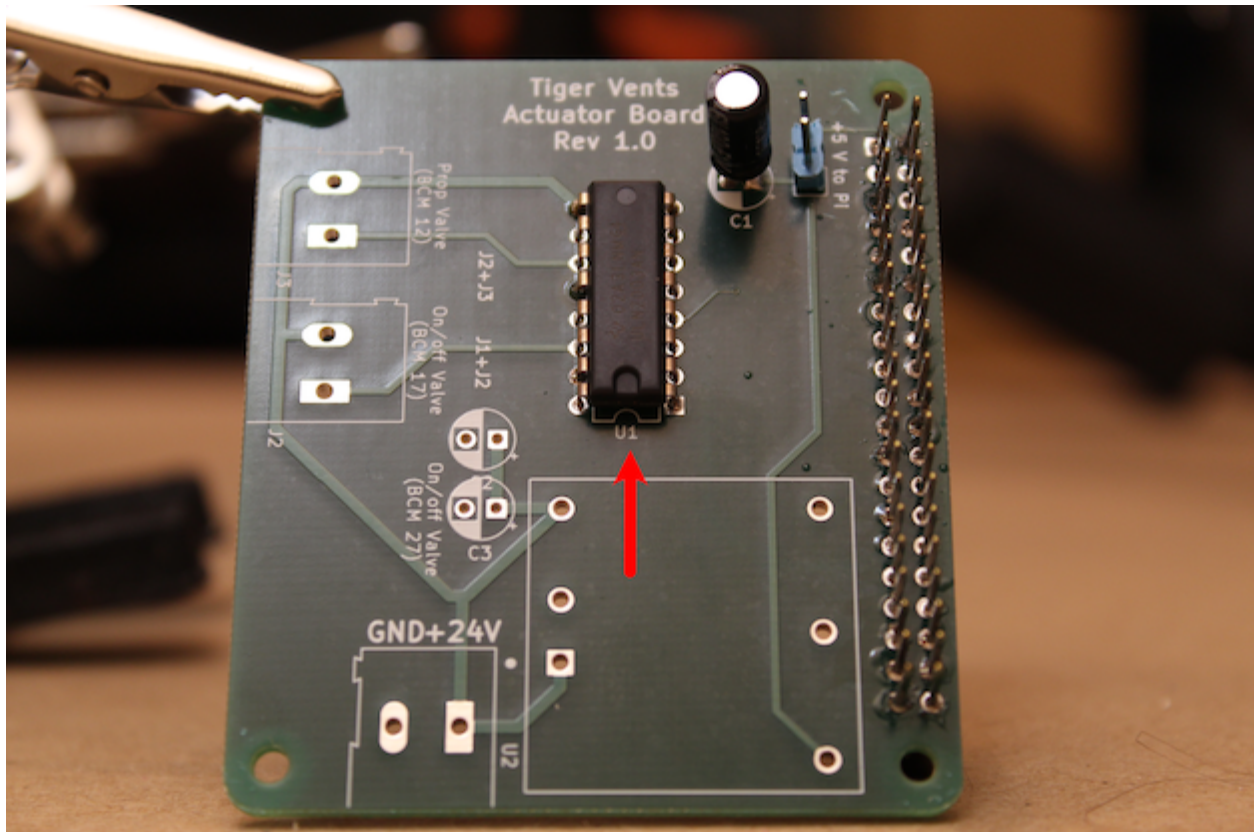


Step 3. Solder the 100 μ F, 16V capacitor onto the board (position C1).

Insert the legs of the capacitor into the slots from the top of the board. The longer leg should be inserted into the side marked “+”. Once the capacitor is soldered in place, use wire cutters to clip off the long legs from the back.

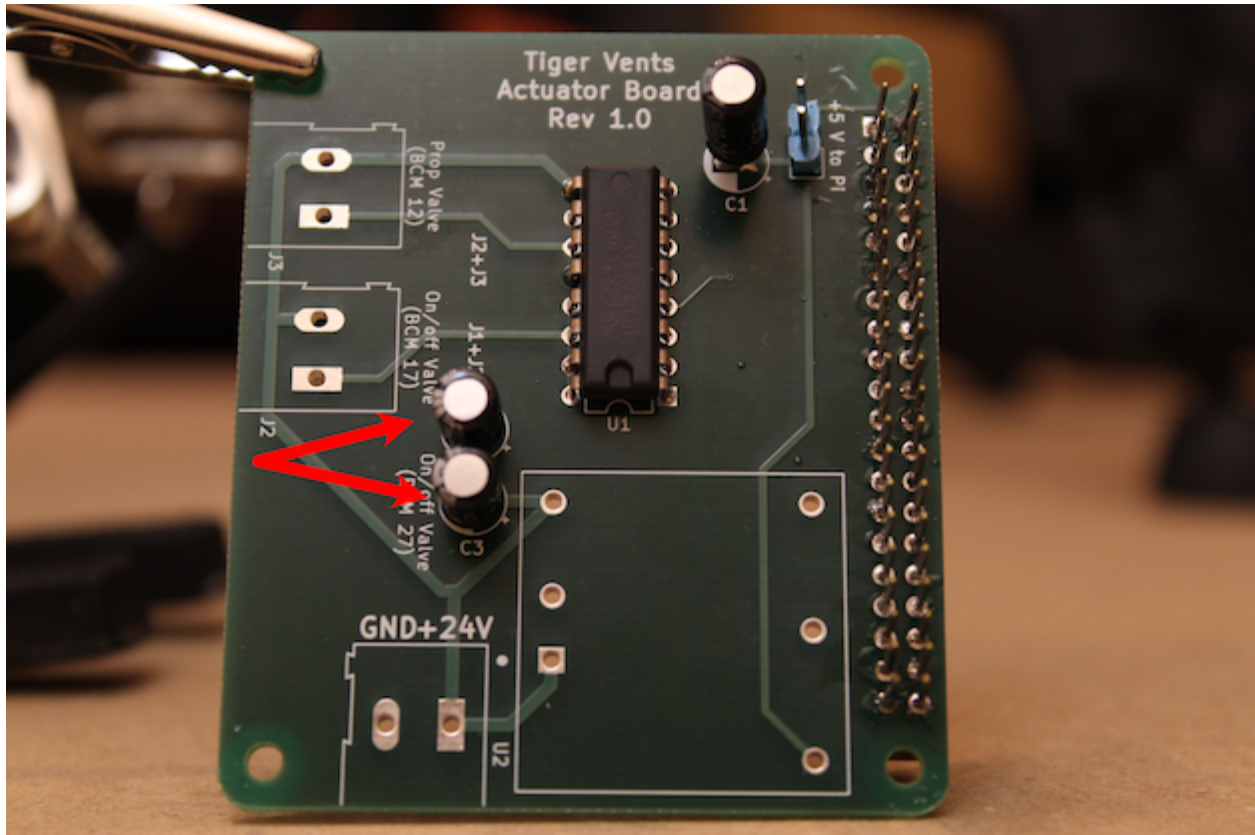


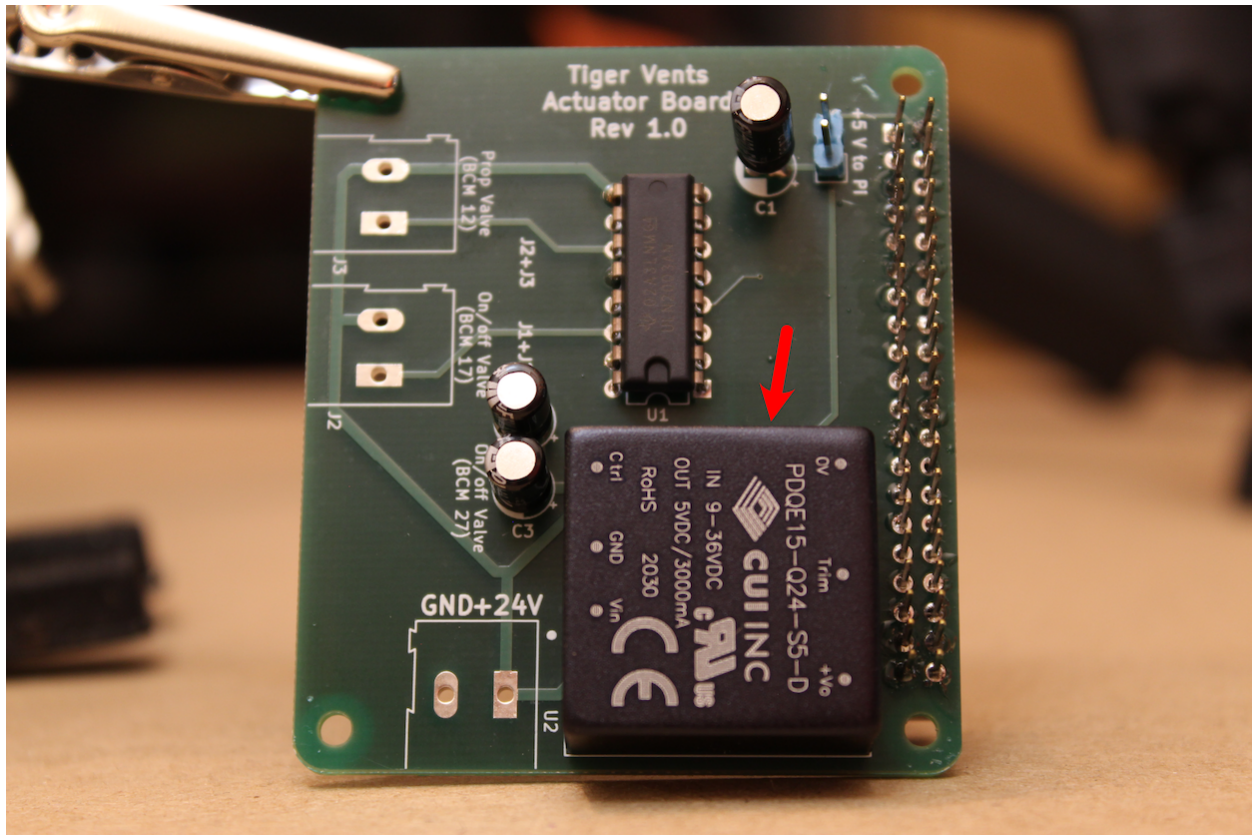
Step 4. Solder the ULN2003A (Darlington array) into place (position U1).
Be sure to match the notched end of the part to the notch indicated on the board.



Step 5. Solder the two 6.8 μ F, 50V capacitors onto the board (positions C2, C3).

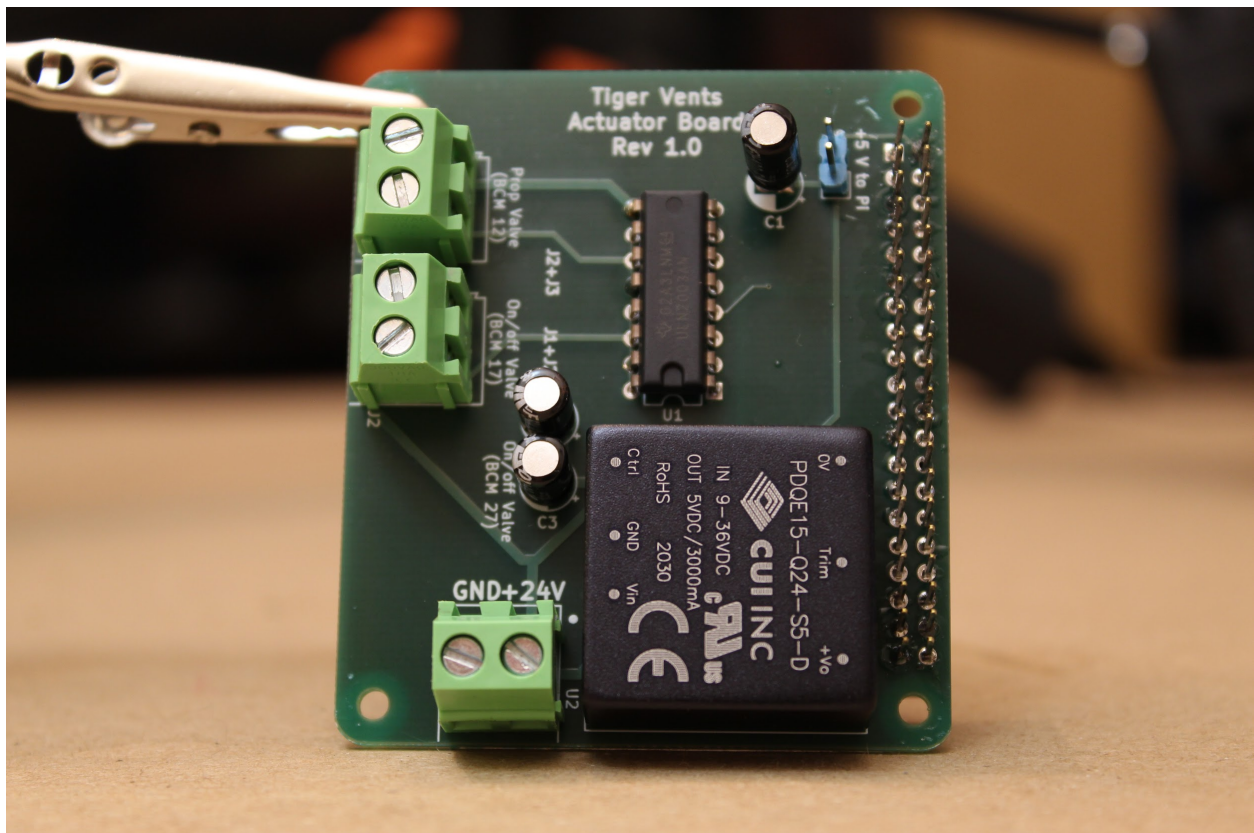
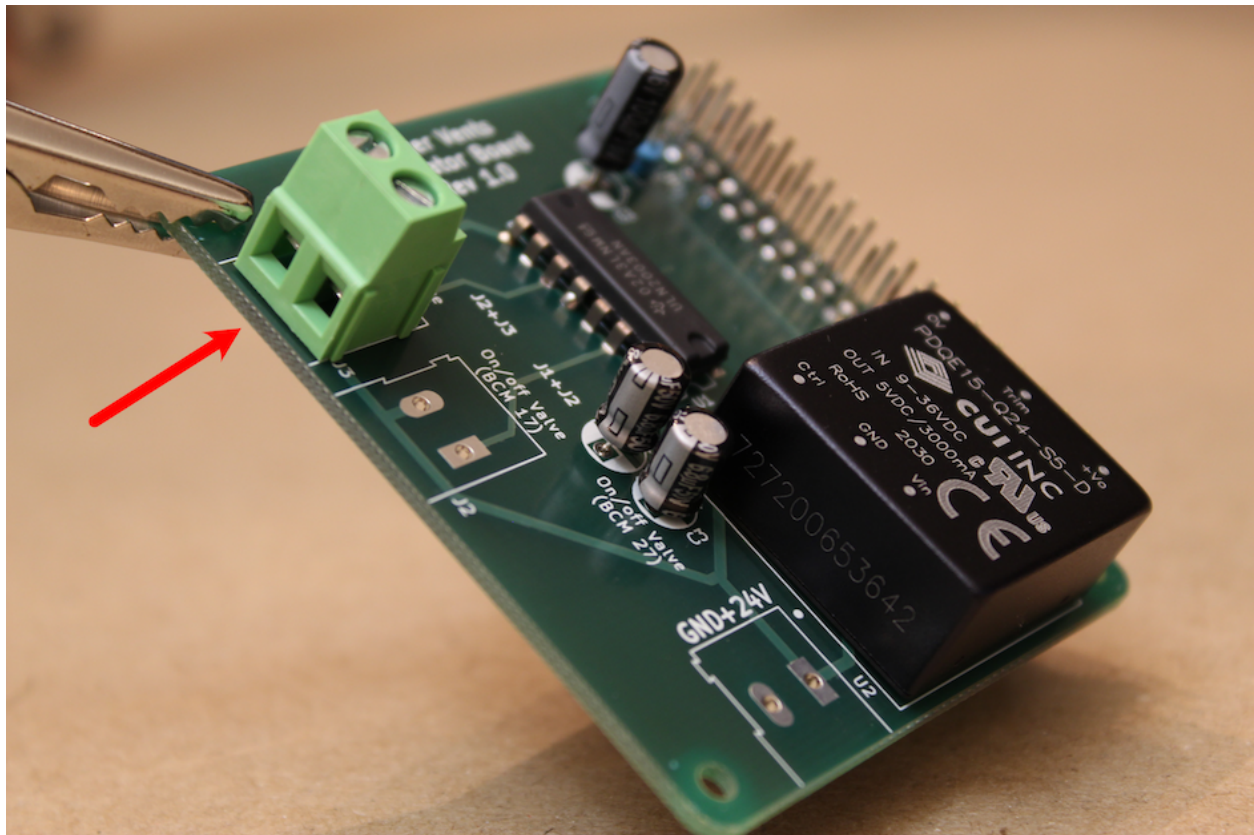
As before, ensure that the longer capacitor legs are inserted through the side marked with the “+”. Once the parts are soldered in, snip the long legs from the back of the board.

**Step 6. Solder the 24-to-5 V DC-DC converter (CUI PDQ15-Q24-S5-D) onto the board, as shown.**



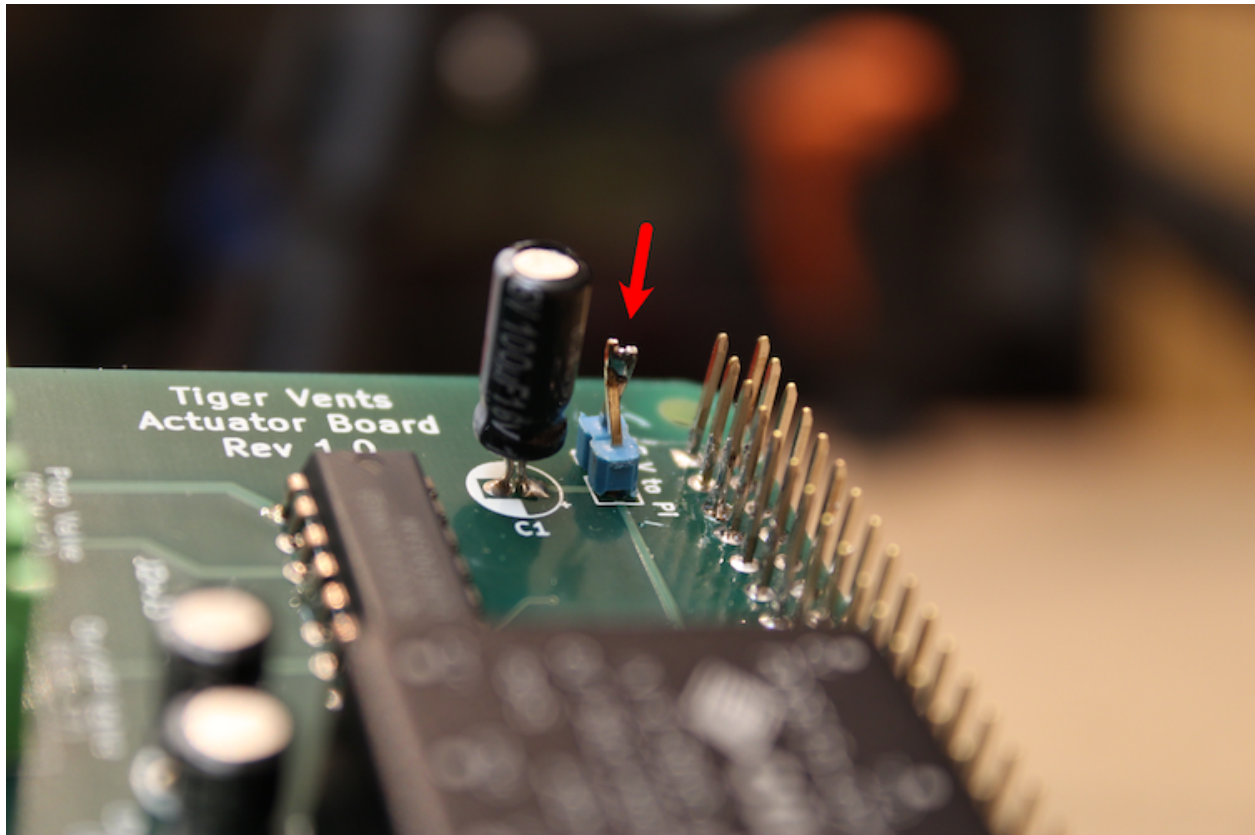
Step 7. Solder the 3 2-pin screw terminals (5.08mm pitch) to the board, as shown.

Be sure to orient all three such that wires can be screwed into the terminals from the outside of the board (as shown).

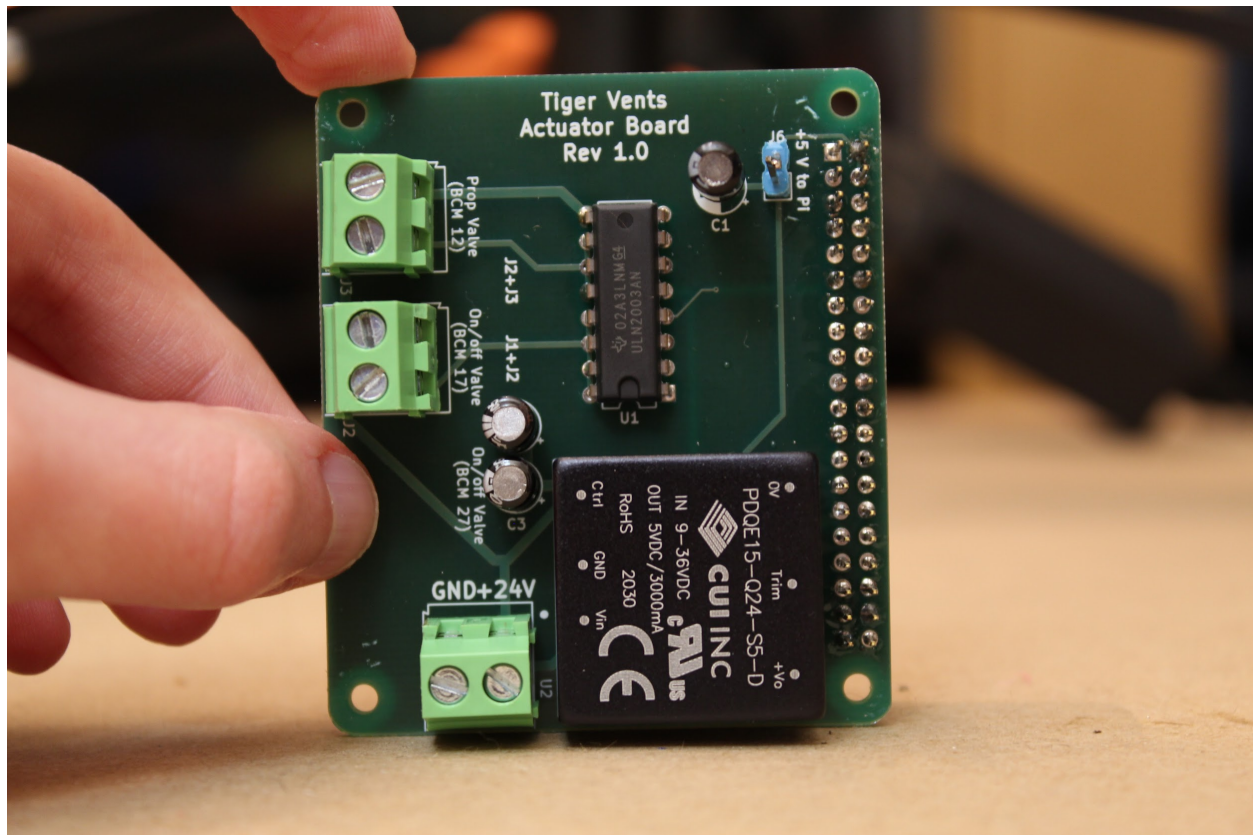


Step 8. Finally, jumper the 2-pin header.

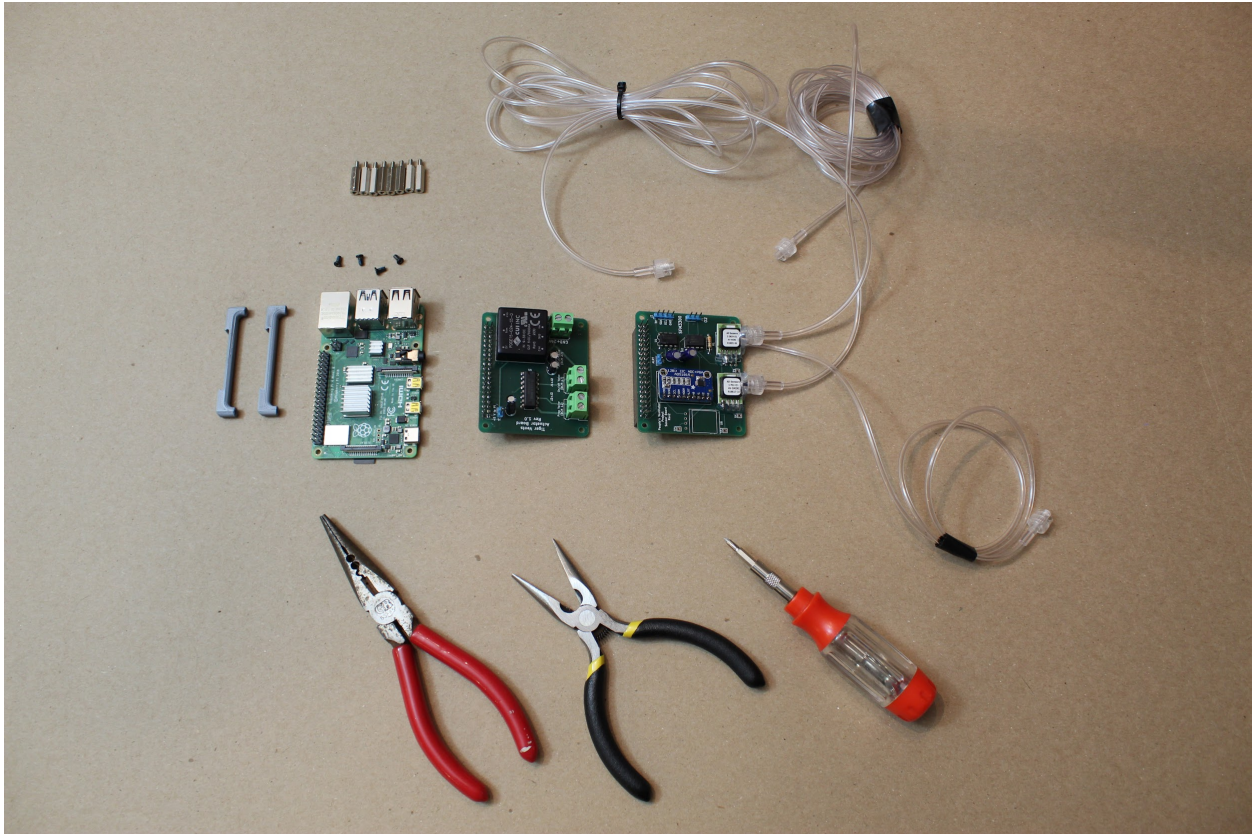
You can use a pin jumper (“shunt”) if you have one, or crush the pins together with a pair of pliers, and then solder the two pins together. We display the latter method, below:



Step 9. The actuator board is ready to go!

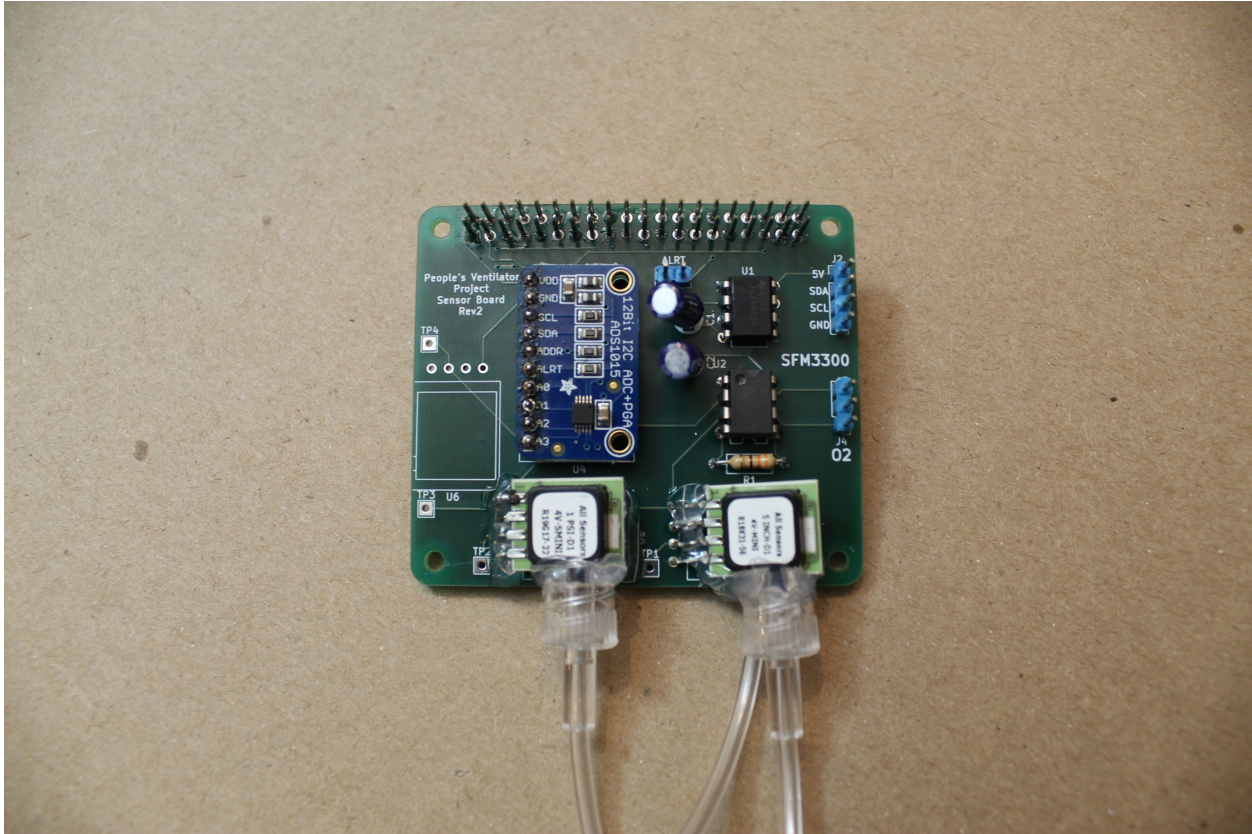


3.3 Assembling the PCB-RPi stack



Step 1. Attach three gas sampling lines to the port on the two pressure sensors.

The differential pressure sensor has two ports. If you have an airway-safe glue, feel free to glue these in place (not required).



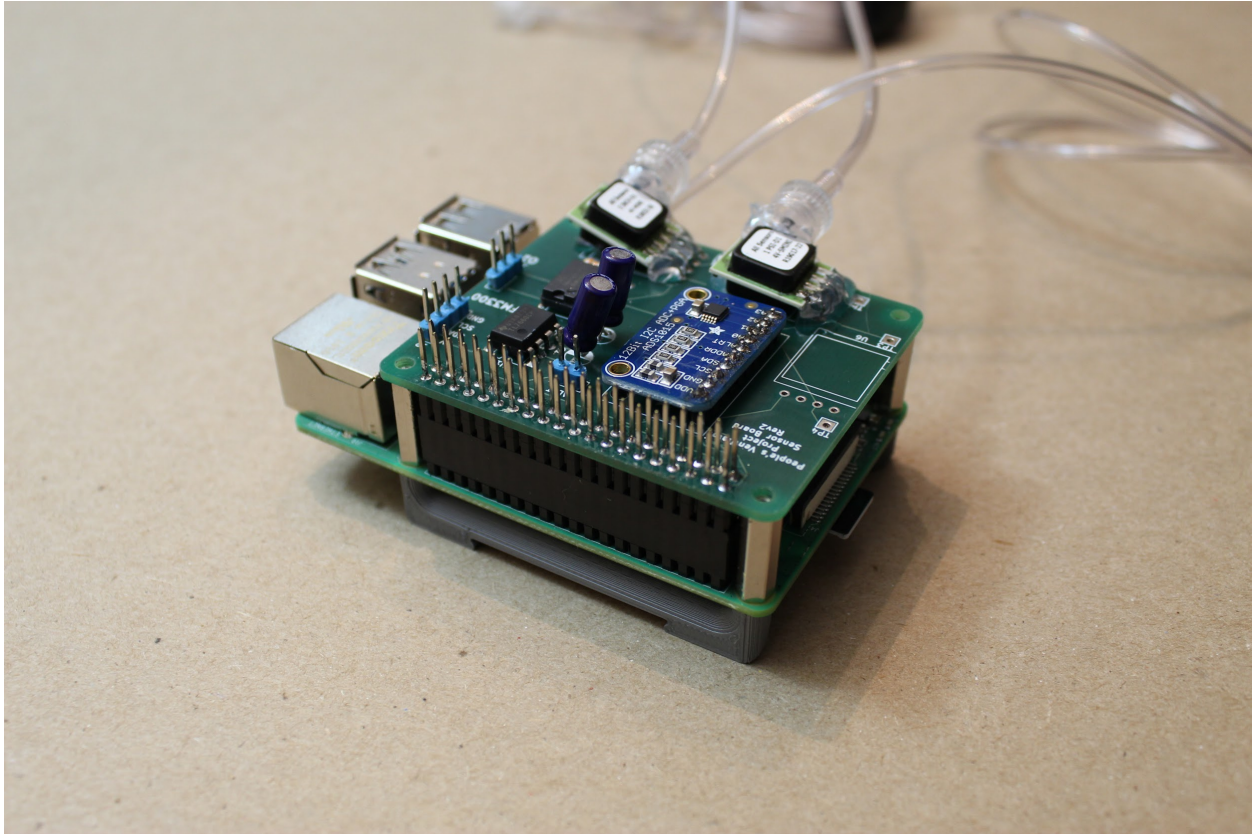
Step 2. Begin stacking the board by attaching the DIN rail mounts.

You can pre-“tap” the 3D printed DIN rail mounts using the small screws that come with the Raspberry Pi. Then, screw four of the 16mm standoffs through the Raspberry Pi from above, and into the DIN rail mounts, as shown.

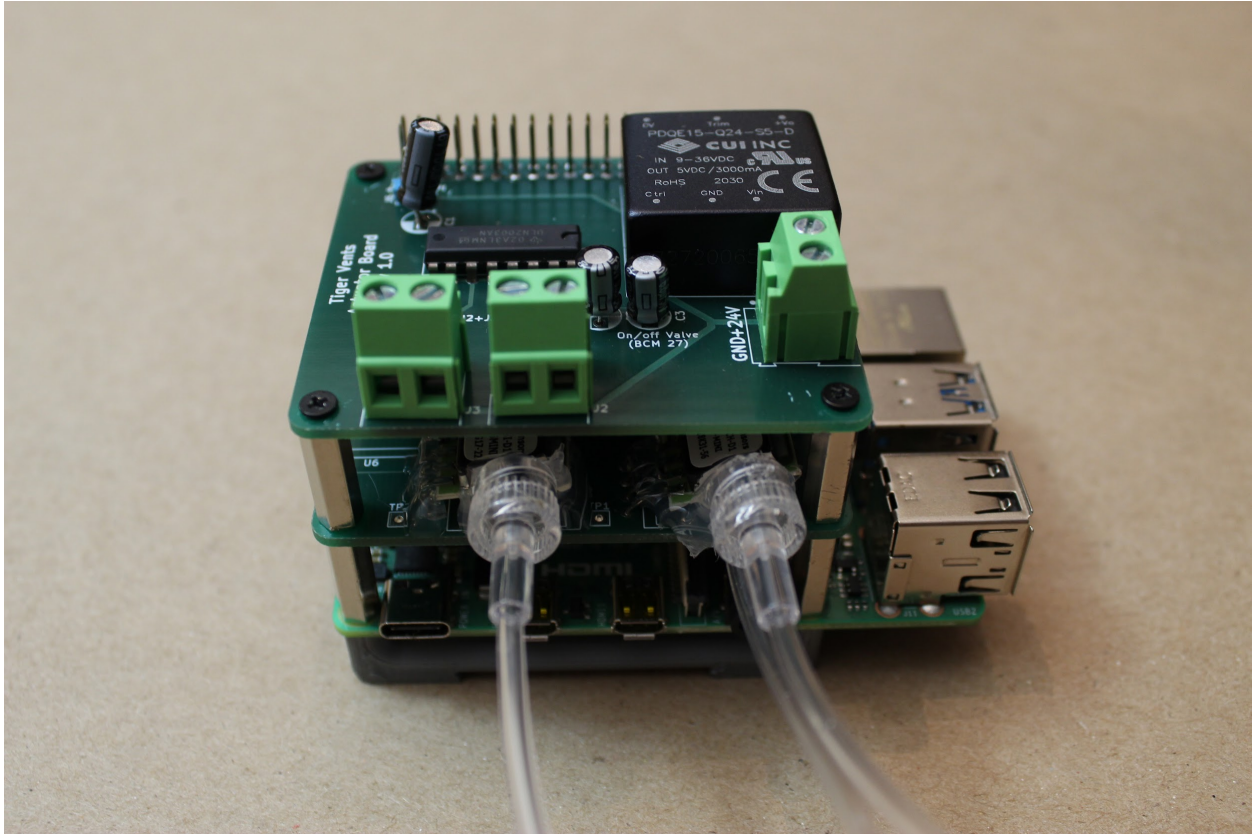


Step 3. Next, add the sensor board.

Connect the sensor board via the stackable headers, then use another four 16mm standoffs to attach the board by screwing these into the original four standoffs, using pliers as needed.

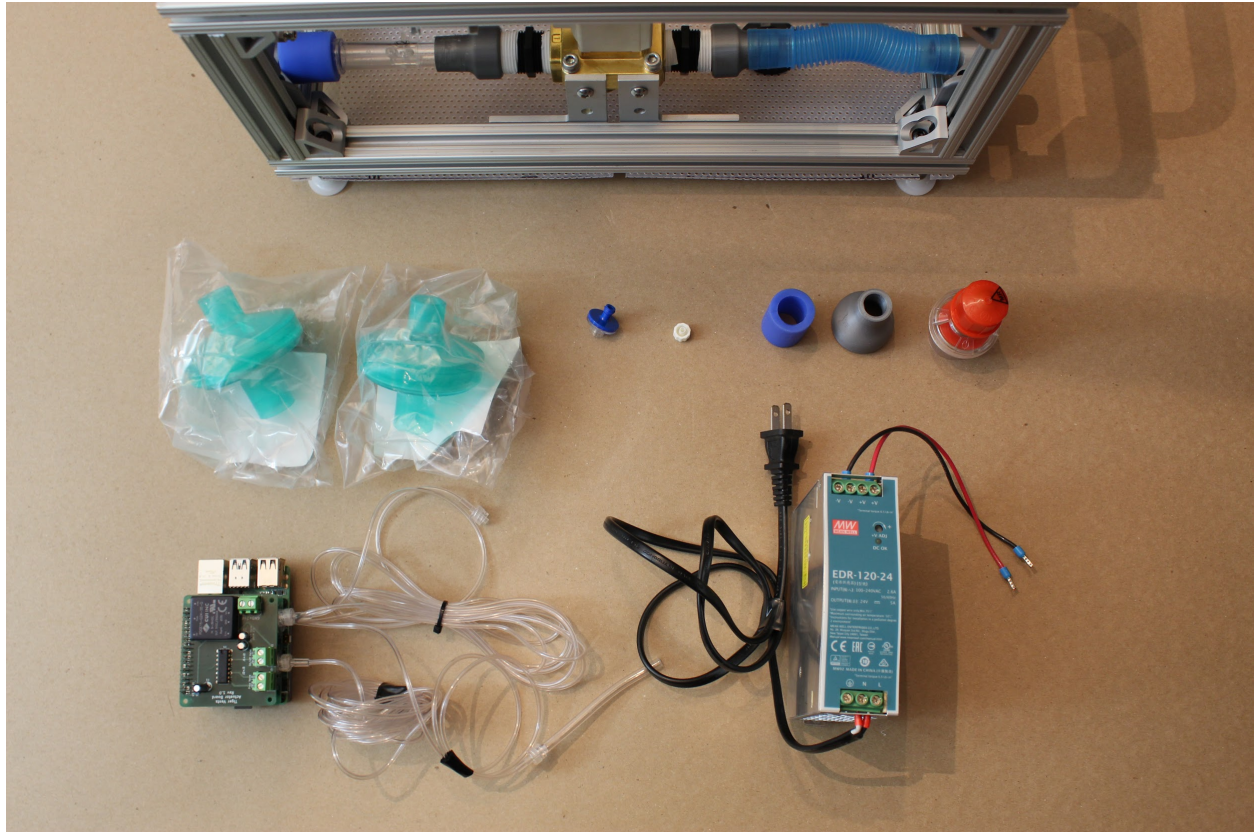
**Step 4. Add the actuator board.**

Connect the actuator board via the stackable headers, then use the four screws that came with your Raspberry Pi to attach the board, as shown.



1.1.7.4 Part 4. Putting it all together

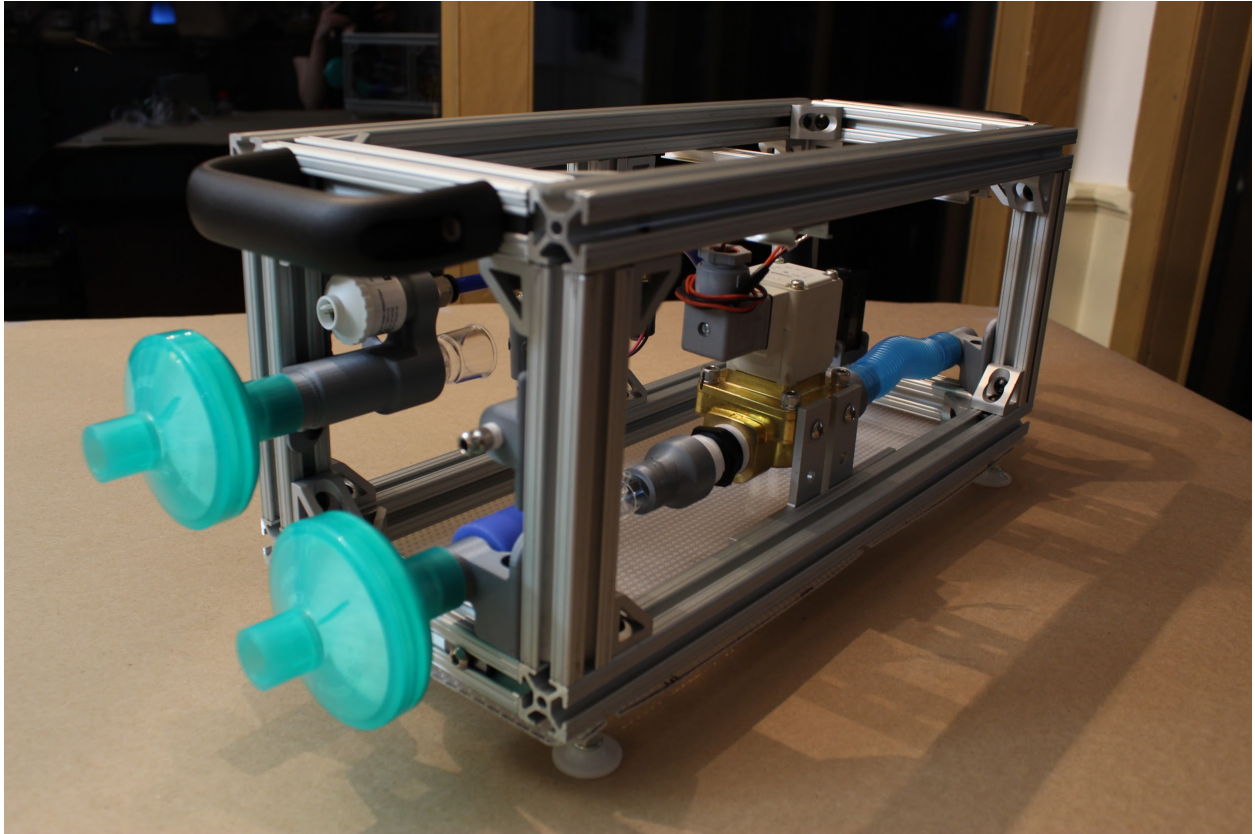
4.1 Wrapping it up.



(Optional step: Attach the side panels using the shorter 80/20 hex screws, as you go.)

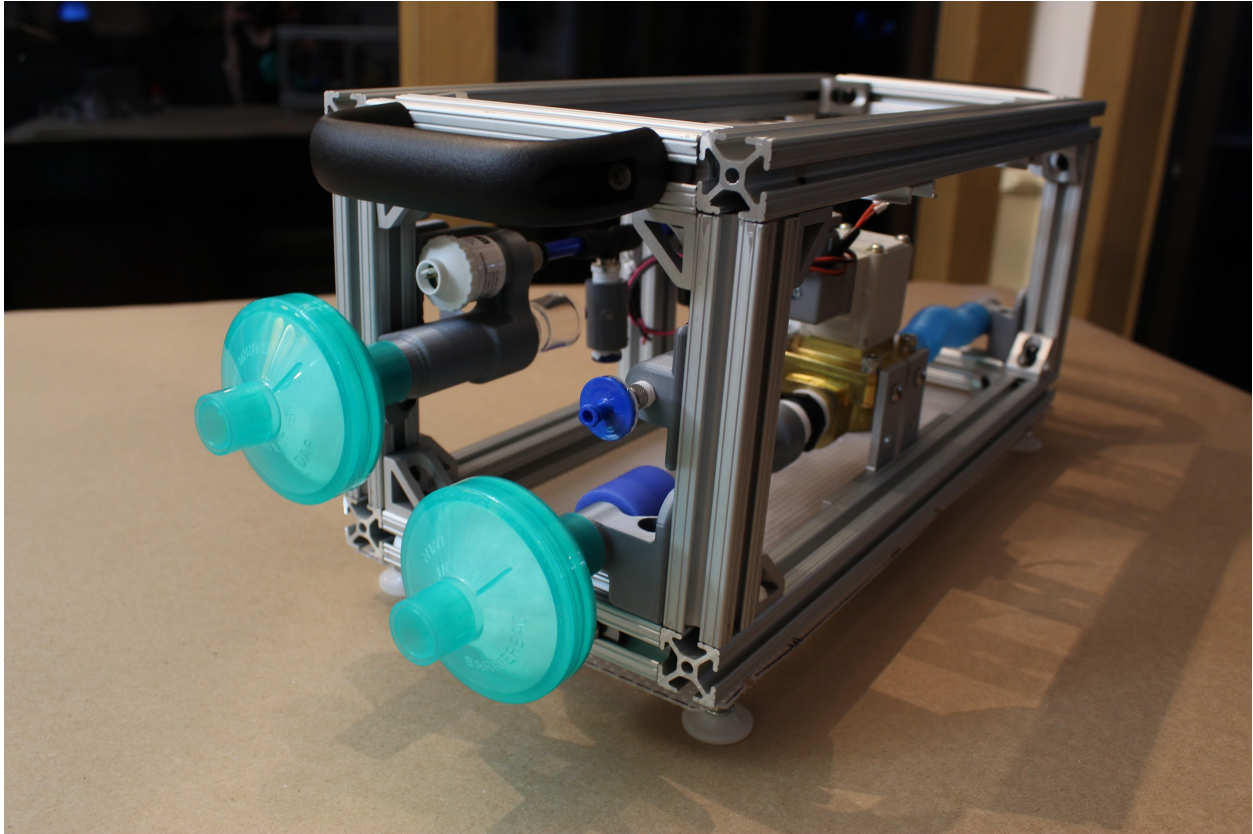
Step 1. Attach the DAR filters.

Insert these by hand to the appropriate ports on the front of the device, as shown: these will attach to the sensor atrium and the expiratory DAR filter bracket. The DAR filters indicate which end should face the patient.

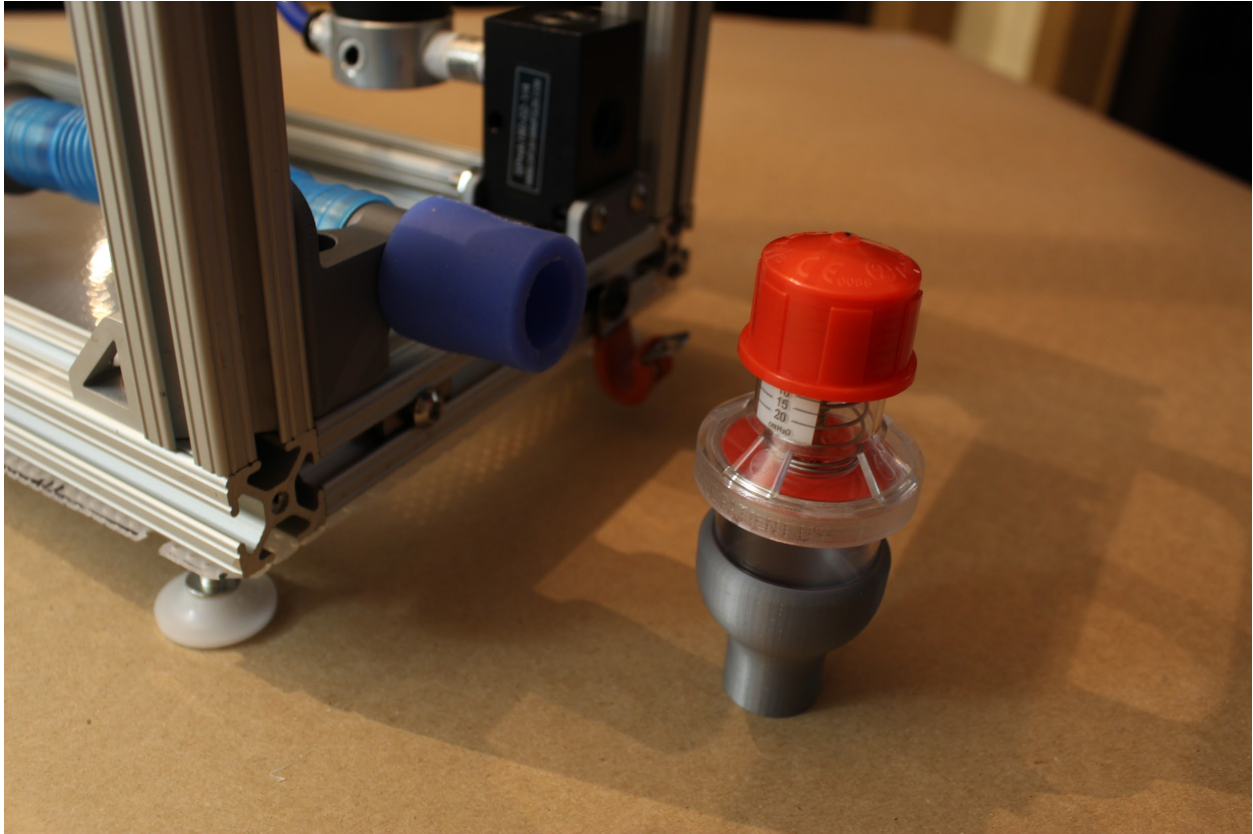


Step 2. Attach the luer lock filter.

This can be twisted on by hand to the luer lock adapter at the front of the device.

**Step 3. Attach the PEEP valve.**

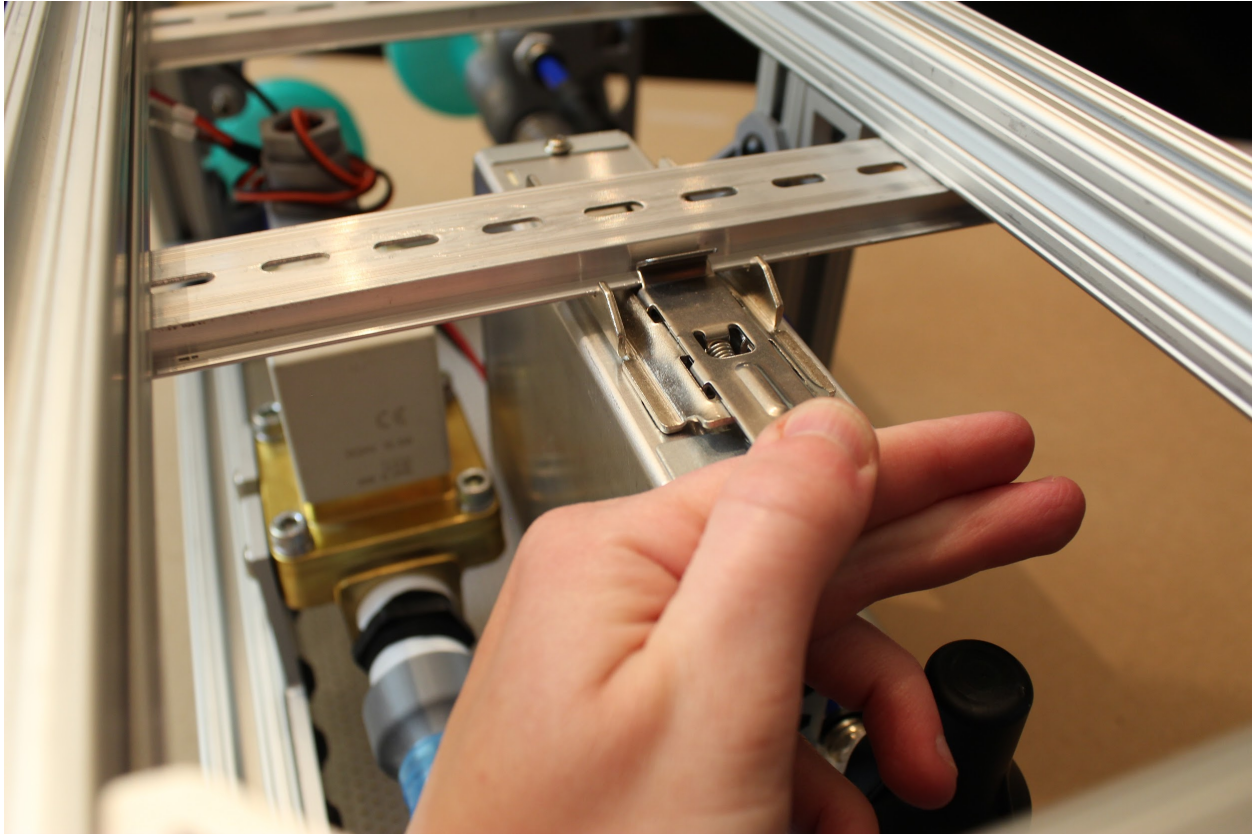
If you're using a commercial PEEP valve, begin by attaching the blue silicone connector. Then, insert the PEEP valve into the 3D printed "22mm to commercial PEEP adapter"- then plug this assembly into the silicone connector.



Step 4. Attach the power supply.

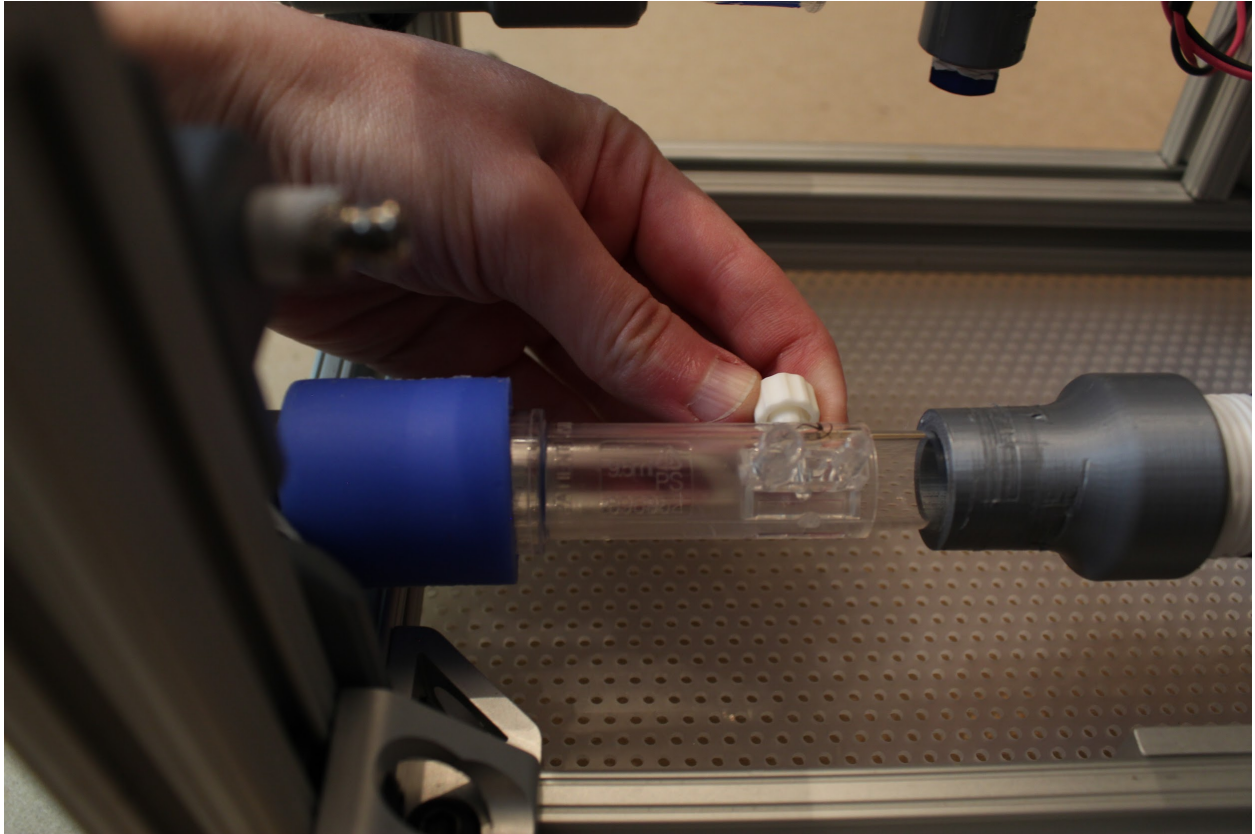
Pull the tab on the Meanwell power supply to attach it to the DIN rail closest to the back of the device. It may be necessary to adjust the position of this DIN rail to ensure that the power supply is not touching the expiratory solenoid or inlet manifold/pressure regulator. Feel free to reposition as needed!

***Note:** if you wish to attach a rear HPDE panel, we recommend inserting the power cable to the power supply through the grommet of the rear panel before insertion of the power supply into the device. Do this now! You can also attach the “Rear Panel Vent”s to the rear panel; they will snap right in.



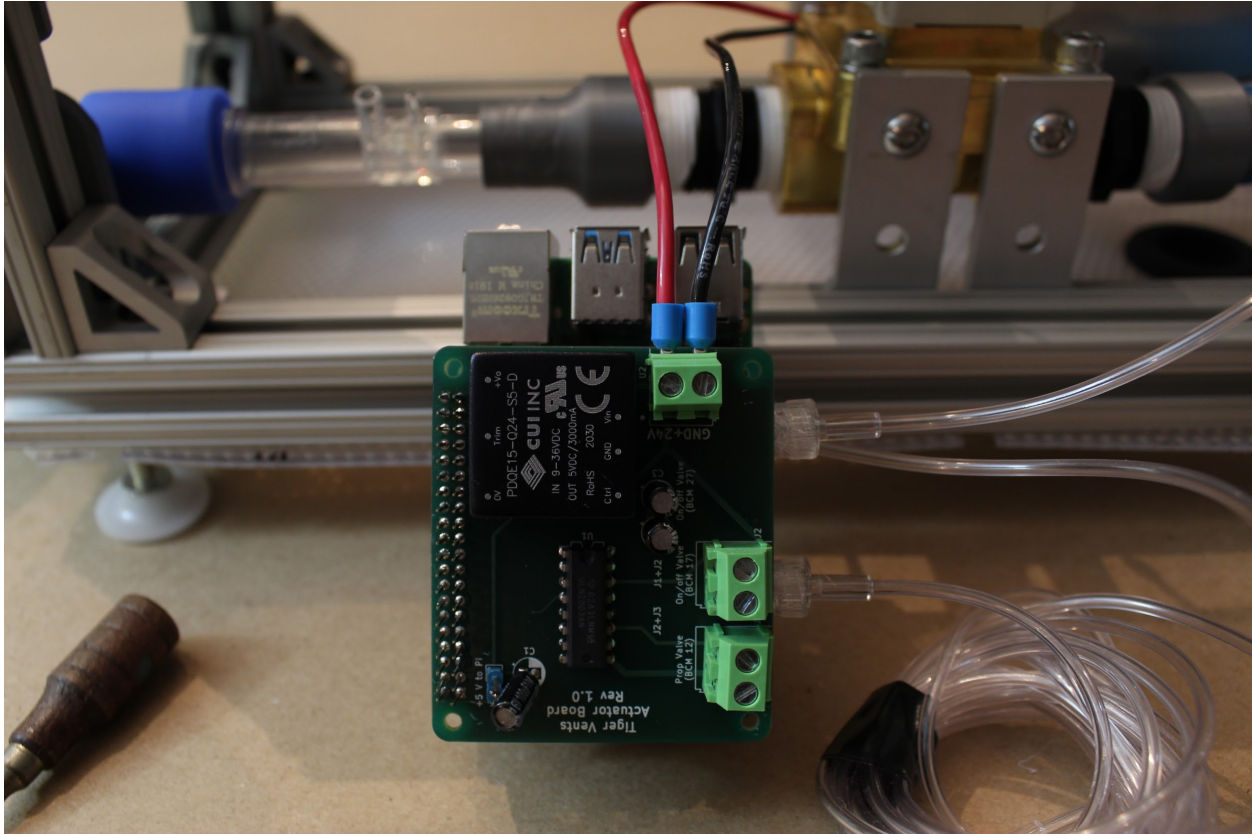
Step 5. Attach the luer lock plug to the D-Lite.

The third port on the D-Lite (physically removed from the other two) is not needed; plug that now with a luer lock plug.

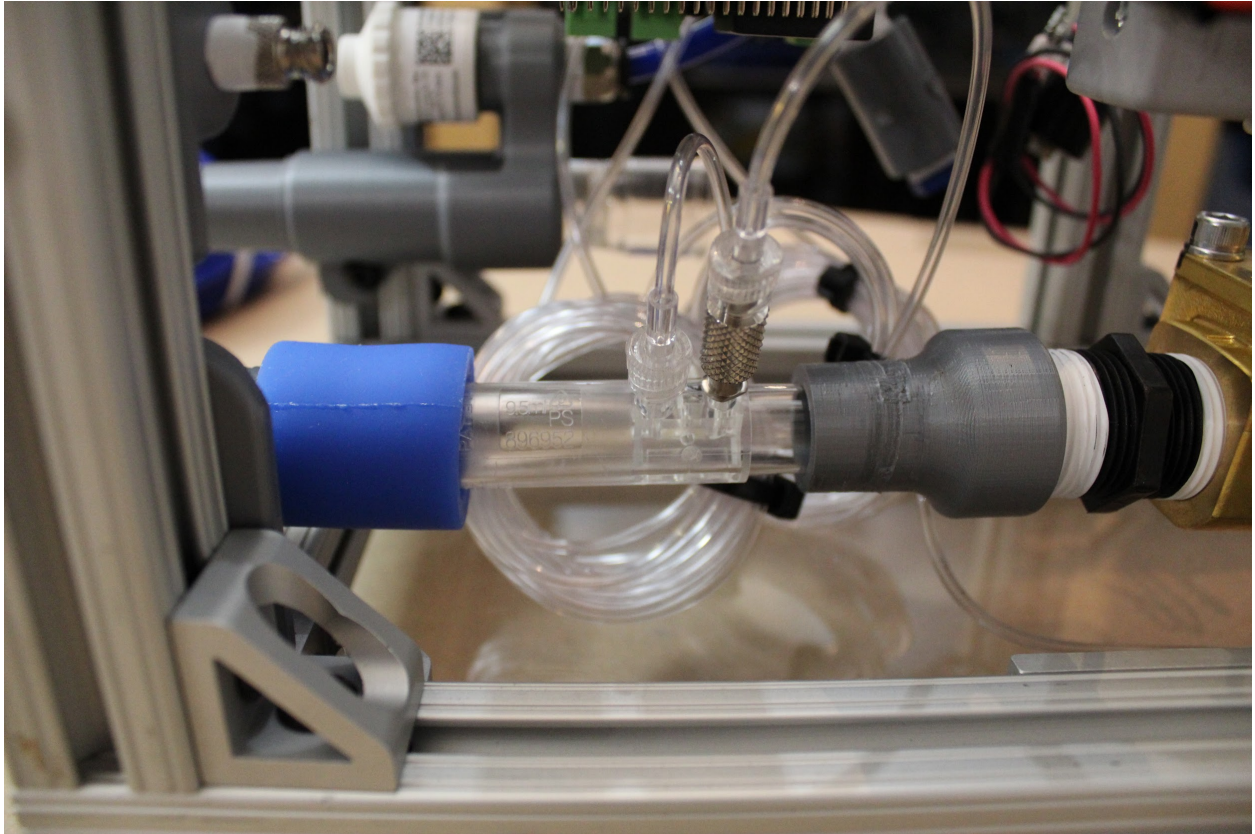


Step 6. Attach the power and ground wires to the PCB stack. Then wire the proportional valve and solenoid.

Use a small flathead screwdriver to attach the power and ground from the power supply to the PCB stack where indicated. Then do the same for the proportional valve wires and the expiratory solenoid wires: the board indicates which pair goes where!

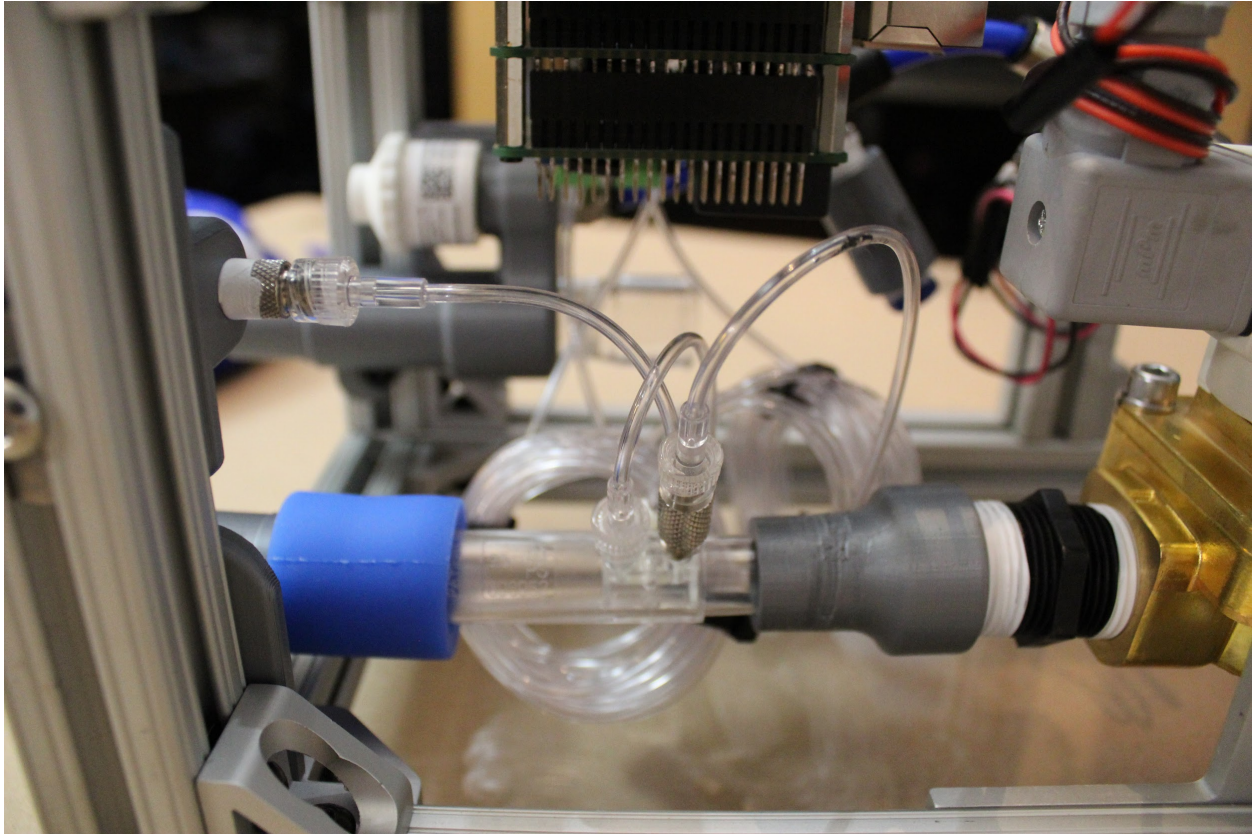
**Step 7. Attach the gas sampling lines to the D-Lite.**

Use the luer lock connector on the smaller of the two ports, then connect the two gas sampling lines from the differential pressure sensor (from the same sensor on the Sensor Board).



Step 8. Attach the gas sampling line to the luer lock filter mount adapter.

The third gas sampling line runs from the second pressure sensor; attach this to the inner luer lock adapter on the luer lock filter mount.



Step 9. Plug in any required cables to connect the monitor and keyboard.

Run the cables outside of the box through the rubber grommets; wire ties can assist with cable management.

Congratulations! You're done!



1.1.8 Electronics

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.

We outsourced our PCB fabrication to Advanced Circuits, based out of Aurora, CO (for \$33 each): <https://www.4pcb.com/pcb-prototype-2-4-layer-boards-specials.html> If you would like to do the same, you can send them the Gerber .zip files we have provided directly.

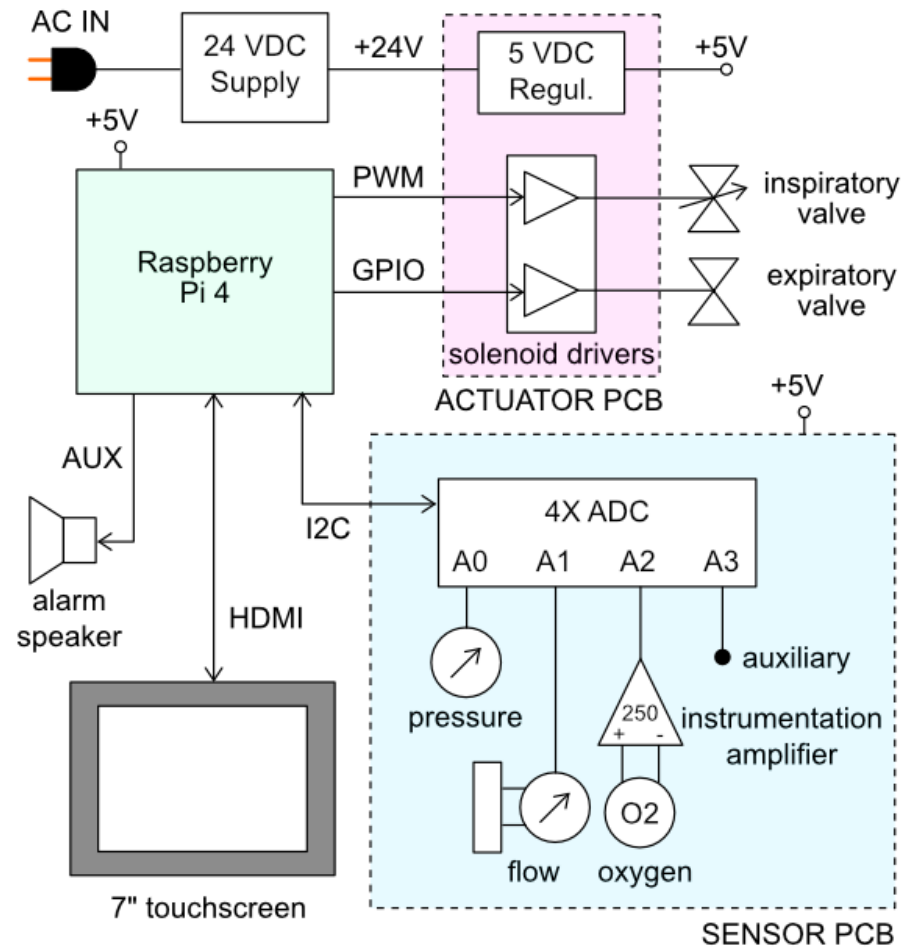


Fig. 10: PVP block diagram for main electrical components

1.1.8.1 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 2: Power and I/O bill of materials

Part	Description
Meanwell 24 V DC Power Supply	DIN Rail Power Supplies 150W 24V 5A EN55022 Class B
Raspberry Pi	Raspberry Pi- Model B-1 (1GB RAM)
USB-C Charger/cable	To power the RPi
Micro SD Card	SanDisk Ultra 32GB MicroSDHC UHS-I Card with Adapter
Raspberry Pi Display	Matrix Orbital: TFT Displays & Accessories 7 in HDMI TFT G Series
HDMI for Display	Display cable: HDMI Cables HDMI Cbl Assbly 1M Micro to STD
Mini USB for Display	Display cable: USB Cables / IEEE 1394 Cables 3 ft Ext A-B Mini USB Cable
Screen mount thumb screws	SCREEN_MOUNT_THUMB_SCREW: Brass Raised Knurled-Head Thumb Screw, 1/4"-20 Thread Size, 1/2" Long
Cable grommet	USER_INTERFACE_CABLE_GROMMET: Buna-N Rubber Grommets, for 1-3/8" Hole Diameter and 1/16" Material Thickness, 1" ID, pack of 10
Cable P-clip	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
Keyboard	Adesso: Mini keyboard with trackball

1.1.8.2 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]

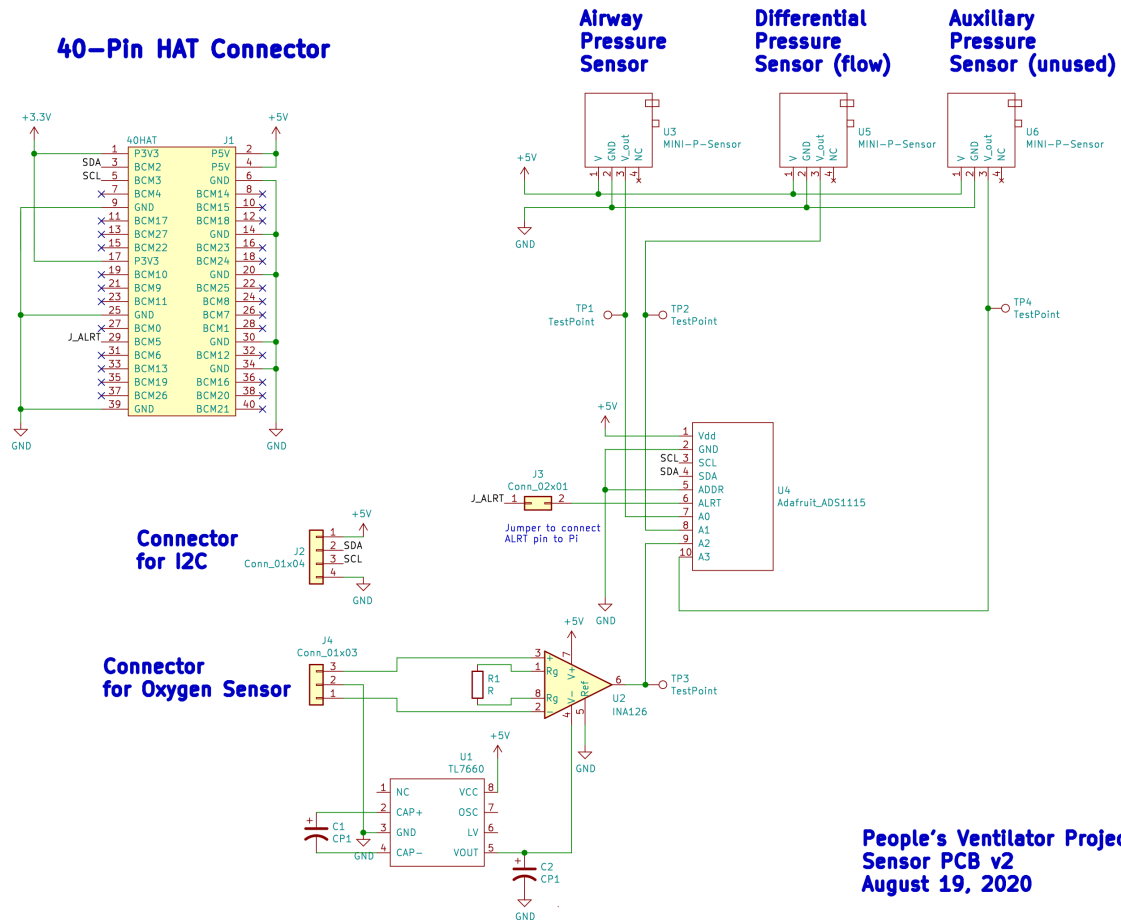


Fig. 11: Sensor PCB schematic

Table 3: Sensor PCB bill of materials

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

1.1.8.3 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]

40-Pin HAT Connector

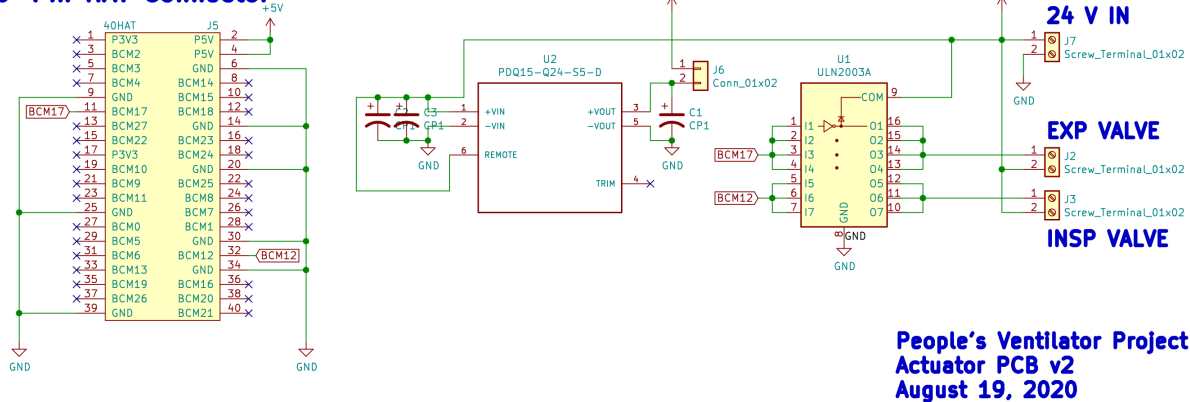


Fig. 12: Actuator PCB schematic

Table 4: Actuator PCB bill of materials

Ref	Part	Purpose
J2	2-pin screw terminal, 5.08 mm pitch, PCB mount	Connects to 24V supply
J3	2-pin screw terminal, 5.08 mm pitch, PCB mount	Connects to on/off expiratory valve
J4	2-pin screw terminal, 5.08 mm pitch, PCB mount	Connects to inspiratory valve, driven by PWM
J5	40-pin stackable RPi header	Connects board to RPi
J6	2-pin 0.1" header	Jumper between 5V and Raspberry Pi
C1	100 uF, 16V	5V rail filter cap
C2	6.8 uF, 50V	24V rail filter cap
C3	6.8 uF, 50V	24V rail filter cap
U1	ULN2003A	Darlington BJT array to drive solenoids
U2	CUI PDQ15-Q24-S5-D	24-to-5V DC-DC converter

1.1.9 Bill of Materials

- Bill of materials - [.csv]

Component	Description
I/O	
Raspberry Pi	Raspberry Pi- Model B-1 (1GB RAM)
USB-C Charger/cable	To power the RPi
Micro SD Card	SanDisk Ultra 32GB MicroSDHC UHS-I Card with Adapter
7" Display	Lilliput 779GL-70NP/C/T - 7" HDMI Capacitive Touchscreen monitor
Screen Mount	Lilliput Monitor Stand
HDMI for Display	Display cable: HDMI Cables HDMI Cbl Assbly 1M Micro to STD
Mini USB for Display	Display cable: USB Cables / IEEE 1394 Cables 3 ft Ext A-B Mini USB Cable
Cable grommet	USER_INTERFACE_CABLE_GROMMET: Buna-N Rubber Grommets, for 1-3/8"
Cable P-clip	USER_INTERFACE_CABLE_P-CLIP_0.375_ID_SS: Snug-Fit Vibration-Damping
Keyboard	Adesso: Mini keyboard with trackball
	AC Power Cable
Power strip	Mini power strip - 3 outlet power strip with 3 ft cord
Airway Components	
Oxygen DISS to 3/8" NPT Adapter	CUSTOM_ID_RUBBER_GROMMET_FOR_0.5_HOLE_0.0625_MATL: Cut-to-S
Inlet manifold	(Not specified in assembly)
1/4" NPT Connector (male/male)	1023N150_MANIFOLD_0.375_NPT_IN_0.25NPT_OUT : Straight-Flow Rectangu
Teflon tape	PIPE_STUB_0.25_NPT_SS_W_SEALANT: Standard-Wall 304/304L Stainless Stee
Manifold plug 1/4" NPT	PTFE thread sealant tape (1/2")
Hex nuts	MANIFOLD_PLUG_0.25_NPT_SS: High-Pressure 347 Stainless Steel Pipe Fitting
Manifold washers	LN_10-32_STAINLESS_18-8: 18-8 Stainless Steel Hex Nut, 10-32 Thread Size, pa
Fixed pressure regulator	W_10_NARROW_0.406OD_316_SS: 18-8 Stainless Steel Washer for Soft Material
Push-to-connect: 1/4" NPT	Fixed-Pressure Compressed Air Regulator (with two plugs) , 50 psi
	PUSH_CONNECT_8MM_0.25_NPTM: 1/4" NPT to push-to-connect 8mm tubing

Component	Description
Pneumatic Tubing	Polyurethane tubing: Firm Polyurethane Rubber Tubing for Air and Water, 5 mm ID
Push-to-connect: 1/8" BSPT	PUSH_CONNECT_FITTING_8MM_TO_0.125_BPST: Push-to-Connect Tube Fitting
Insp proportional valve	SMC PVQ31-5G-23-01N-H valve, proportional, PVQ PROPORTIONAL VALVE
Push-to-connect: Inline T-adapter to 1/4" NPT	PUSH_CONNECT_T_8MM_0.25_NPT: Push-to-Connect Tube Fitting for Air, Inline
1/4" NPT Connector (female/female)	NPTF_0.25_X_0.25_NPTF_COUPLER: CPVC Pipe Fitting for Hot Water, Straight
Pressure release valve	Compact Nylon Pressure-Relief Valve for Air, 1 psi, 1/4 NPT Male
Oxygen sensor	O2 Sensor, SS-12A (or equivalents, e.g. Analytical Industries PSR-11-917-M)
Emergency breathing (check) valve	Teleflex Medical One Way Valve
DAR filters	DAR Electrostatic filters - Large (10x) (NEED A PRESCRIPTION TO ORDER)
Adult Respiratory Circuit w Humidifier Limb	Standard resp. circuit
	Adapter / filter to connect gas sampling lines to proximal side - TOM
Gas sampling line	Gas Sampling Line McKesson 16-GSL Each/1 (x3)
Luer lock connector	Sensit Luer Lock Connector from Grainger
Luer lock filter	Luer lock filter 0.2um from Promepla
Pressure sensor (for Paw)	1 PSI-D-4V-MINI
Tubing connectors	Flexible Connector (Silicone Rubber) 22mm I.D. X 22mm I.D. (x6)
Flow sensor	Disposable GE D-Lite flow sensor
Luer plug	Plastic Quick-Turn Tube Coupling, Nylon Plastic Caps for Plugs
Luer lock connector for D-Lite	Quick Turn Tube Coupling for Air - 316 Stainless Steel Socket Connector
Differential pressure sensor for flow	5 INCH-D2-P4V-MINI
3/4" NPT Connector (male/male)	Thick-Wall Polypropylene Pipe Fitting for Chemicals, Connector with Hex Body, 3/4"
Exp solenoid valve	SMC VXZ250HGB valve, 20A, NORMALLY OPEN, C37, Port 3/4 (NPT), Orifice
Exp solenoid brackets	T-Slotted Framing, Silver Flush 90 Degree Angle Bracket for 1" High Rail
Framing/Structural Components	
80/20 (x4)	T-Slotted Framing, Single 4-Slot Rail, Silver, 1" High x 1" Wide, Solid, 3' Long - 4x
80/20 gusset bracket (x18)	Silver Gusset Bracket, 1" Long for 1" High Rail T-Slotted Framing x18
80/20 rail (x6)	T-Slotted Framing, Single 4-Slot Rail, Silver, 1" High x 1" Wide, Solid, 1' Long x6
80/20 button head nuts (packs of 25, x2)	End-feed single nuts for T-Slotted framing: T-Slotted Framing, End-Feed Single Nut
Button-head hex screws, pack of 50	BHCS_0.25-20x0.412_SS: 18-8 Stainless Steel Button Head Hex Drive Screw, 1/4"
Socket head screw, pack of 100	SHCS_M3_X_0.5_6MM_SS: 18-8 Stainless Steel Socket Head Screw, M3 x 0.5 mm
Button head screws (long), pack of 10	BHCS_10-32_X_2_SS: 18-8 Stainless Steel Button Head Hex Drive Screw, 10-32 T
Button head screws (short), pack of 25	BHCS_M6_X_1_8MM_SS: Button Head Hex Drive Screw, Passivated 18-8 Stainless
HDPE sheeting	Moisture-Resistant HDPE Sheet, 12" x 48" x 1/16"
Lifting handle	LIFTING_HANDLE: Oval Pull Handle, Unthreaded Hole, Black Aluminum, 4-9/16"
Screws for lifting handle	SHCS_0.25-20x0.75_Gr8_ASTM_F1136, pack of 50
Leveling mount (4 feet)	TLA_LEVELING_MOUNT: Light Duty Leveling Mount, 1" Long 1/4"-20 Threaded
Nuts for leveling mount	LN_THIN_0.25-20_STAINLESS: 18-8 Stainless Steel Thin Hex Nut, 1/4"-20 Threa
PETG for 3d printing	~1 kg spool of PETG
Electronics	
Standoffs	Standoffs & Spacers M2.5 x 16mm HEX M/F BRASS 4.9mm A/F
Meanwell DC Power Supply	DIN Rail Power Supplies 150W 24V 5A EN55022 Class B
DIN Rail	Aluminum DIN 3 Rail, 10mm Deep, 1m Long
DIN washers	W_0.25_FLAT_THICK_GR8_YELLOW_ZINC: Zinc Yellow-Chromate Plated Steel
Sensor board (excluding pressure sensors)	
2-layer PCB	Advanced Circuits 2 layer board
12-bit ADC	Adafruit Data Conversion IC Development Tools ADS1015 Breakout 12-Bit ADC
Voltage splitter U1	TL7660, 8-DIP package

Component	Description
Instrumentation amplifier U2	INA126PA, 8-DIP package
R3, sensor	
40 pin Pi header J1	Raspberry pi 40 pin stacking header
4x1 header J2	male pin header 1x04 P2.54mm
2x1 header J3	male pin header 1x02 P2.54mm
Oxygen sensor cable (J4)	3 pin fan cable extension, 36", ribbon style
330 Ohm resistor R1	330 Ohm resistor, through hole, 1/4 Watt
Capacitors C1,C2	10 uF, 25V, electrolytic radial
Actuator board	
2-layer PCB	Advanced Circuits 2 layer board
5V DC/DC Converter	490-PDQE15-Q24-S5-D
Darlington array U1	ULN2003AN IC PWR RELAY 7NPN 1:1 16DIP
40 pin Pi header J1	Raspberry pi 40 pin stacking header
Screw terminals J2, J3, J4	5.08 mm pitch 2-pin screw terminal block connector, PCB mount
2x1 header J5	male pin header 1x02 P2.54mm
Capacitor C1	100 uF, 16V, electrolytic radial
Capacitors C2,C3	6.8 uF, 50V, electrolytic radial
Speaker	Logitech Z50 speaker
Specialized Tools	
M16 tap	HSS Tap, Bottoming Chamfer, M16 x 1.0 mm Thread (for mounting O2 Sensor)
1/4" NPT tap	Uncoated High-Speed Steel Pipe and Conduit Thread Tap High-Speed Steel, Plug C
3/4" NPT tap	Uncoated High-Speed Steel Pipe and Conduit Thread Tap: High-Speed Steel, Plug C
1/4"-28 tap	1/4"-28 TPI (1/4" UNF) for filter mount
Ingmar QuickLung Test Bellows	(Optional) variable resistance/compliance test lung for testing only
Not on the List:	
PETG for 3d printing	~1 kg spool of PETG
Commercial PEEP valves	Alternative: Medium silicone film for printed PEEP valve: https://www.mcmaster.co
Adult Test Lung	Standard adult test lung (Many options here)
Air compressor for initial testing	

1.1.10 CAD

1.1.10.1 3D Printed Parts

Individual 3D printed parts may be downloaded here:

- Inlet manifold bracket - [.stl]
- Proportional valve bracket - [.stl]
- Sensor atrium manifold - [.stl]
- Expiratory DAR filter bracket - [.stl]
- 22mm to Ø.75 NPTM adapter (x2) - [.stl]
- Expiratory outlet bracket to PEEP - [.stl]
- Luer lock filter mount - [.stl]

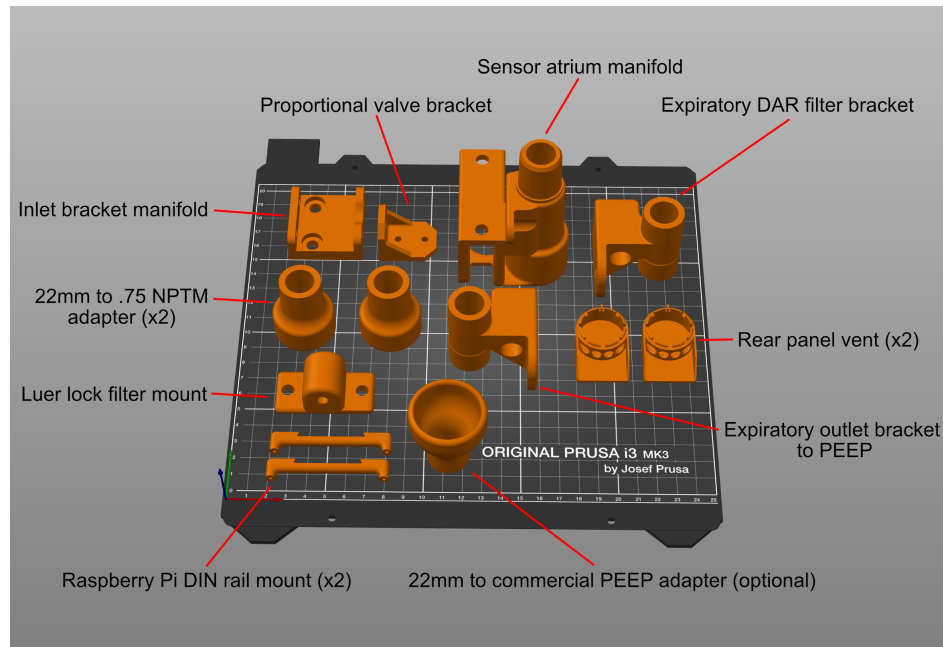


Fig. 13: Sample Prusa project with 3D printed components

- Raspberry Pi DIN rail mount (x2) - [.stl]
- Rear panel vent (x2) - [.stl]
- 22mm to commercial PEEP adapter - [.stl]

Optional RPi DIN rail mount alternative, if your printer cannot handle the feature resolution on the original DIN rail mount:

- ALTERNATIVE Raspberry Pi DIN rail mount (x2) - [.stl]

Download all parts:

- All components - [.zip]

Printing tips: Be sure to maintain high infill for airway components: ideally, use 4 wall-layers (vertical layers), minimum 40% infill, and 4 layers on both the top and bottom. Parts should be leak-tested prior to installation, for instance, with a stopper and water bath. Most of our test prints were performed using PLA, and without any supports or rafts, since these are challenging to remove later. Supports should not be necessary provided the parts are oriented mindfully on the build plate. Also, try to keep cylindrical components oriented vertically (so that the circle is traced on the build plate); this will improve circularity of the chamber.

For PRUSA users, we provide an example project, demonstrating part orientation:

- Sample Prusa project - [.3mf]

1.1.10.2 Enclosure

The side, top, and bottom panels are made out of 1/16" HPDE sheeting. Laser cut, or cut by hand, two of the "SIDE_IP_PANEL", and one of each of the rest! We use a perforated HPDE for the bottom panel only; the rest are solid HPDE.

- Bottom panel (perforated) - [.dxf]
- Front panel - [.dxf]
- Rear panel - [.dxf]
- Side panel (x2) - [.dxf]
- Top panel - [.dxf]

Download all .DXF files:

- All DXF files - [.zip]

1.1.11 Software Overview

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 allow 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 *ControlSettings* to the *Controller*.
- At launch, the *Coordinator* spawns a *Controller* that runs the logic of the ventilator based on control values from the GUI.
- The *Controller* communicates with a third *pigpiod* process which communicates with the ventilation hardware.

PVP is configured by

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

PVP is launched like:

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

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

1.1.12 Folder Structure

The repository is organized as follows:

- *pvp/assets/* contains technical information like CAD drawings, circuit diagrams.
- *pvp/data/* contains information for calibrating sensors; data and information.
- *pvp/_docs* and *pvp/docs* is raw, and built documentation.
- *pvp/tests* contains automated tests for all software modules.
- *pvp/sandbox* is experimental code, that is not necessary to operate pvp. Place your toy programs here.
- *pvp/pvp* is the main code. It contains individual files for all modules of PVP1. Binary files like audio/graphics are deposited with the respective module, and not collected in a central site.

1.1.12.1 PVP Modules

1.1.13 GUI

1.1.13.1 Main GUI Module

1.1.13.2 GUI Widgets

Control Panel

The Control Panel starts and stops ventilation and controls runtime options

Alarm Bar

Display

Plot

Components

Dialog

1.1.13.3 GUI Stylesheets

Data:

<i>MONITOR_UPDATE_INTERVAL</i>	inter-update interval (seconds) for Monitor
--------------------------------	---

Functions:

<i>set_dark_palette</i> (app)	Apply Dark Theme to the Qt application instance.
-------------------------------	--

```
pvp.gui.styles.MONITOR_UPDATE_INTERVAL = 0.5  
inter-update interval (seconds) for Monitor
```

Type (float)

`pvp.gui.styles.set_dark_palette(app)`

Apply Dark Theme to the Qt application instance.

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

Args: app (QApplication): QApplication instance.

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

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

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

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

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

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

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

1.1.13.4 Module Overview

1.1.13.5 Screenshot

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



Multiple Control

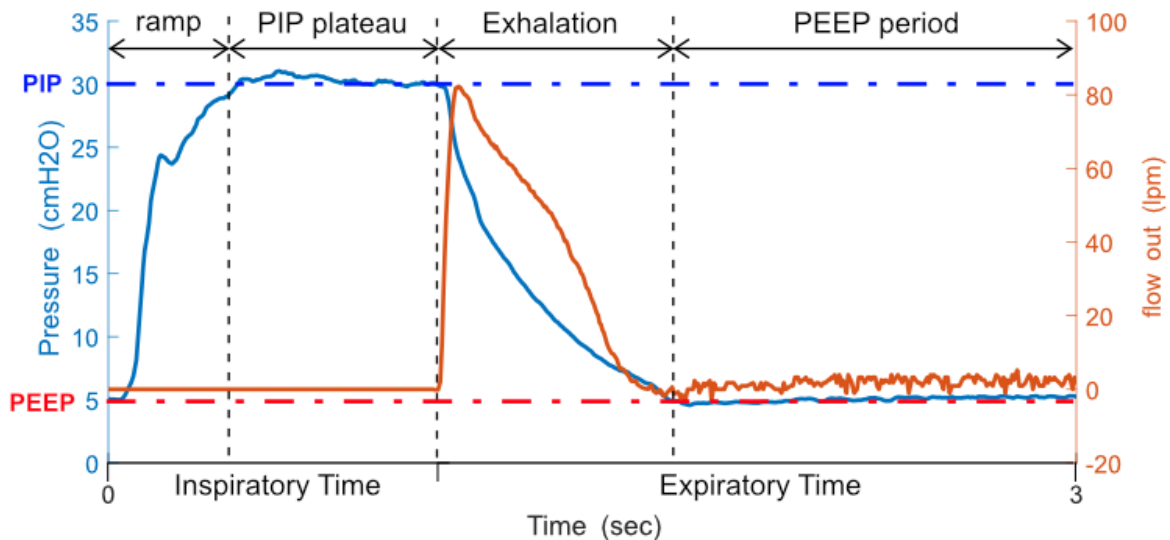
Control ventilation and set alarm thresholds with a mouse, keyboard, or from sensor values

Monitor Limits

Sensor monitors, alarm limits, and alarm states are represented together in multiple modalities

1.1.14 Controller

1.1.14.1 Purpose of the Controller



Shown above is a typical respiratory waveform (without averaging) as produced with PVP1. Blue is the recorded pressure, orange is the flow out of the system. Note that airflow (and also oxygen concentration) are only measured during expiration, so that the main control-loop during inspiration runs as fast as possible, and is not slowed down

by communication delays. Pressure is recorded continuously. Empirically, the Raspberry pi allowed for the primary control loop to run at speeds of ~5ms per loop, which was considerably faster than all hardware delays (i.e. the time it takes for a mechanical, physical valve to open or close; see main manuscript).

The purpose of the controller is to produce a breath waveform, as the one shown above. More specifically, it's job is to reach a certain target-pressure (PIP), and to hold that pressure for a certain amount of time. These numbers are provided by the user via the UI. Exhalation is passive, and PEEP pressure is mechanically controlled with a spring-valve.

Conceptually, the controller is written as a hybrid system of state and PID control. During inspiration, it actively controls pressure with a simple [PID controller](#). That means that during inspiration, it measures the deviation of the pressure-value from the pressure-target-value, and depending on that distance (and its integral and derivative), it adjusts the opening of 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, the controller opens the inspiratory valve slightly to sustain a small flow during PEEP, using the aforementioned manually operated PEEP-valve. We found, empirically, that a combination of proportional and integral term worked best across different physical lung settings.

The controller was also built to allow the user to adjust flow through the system. This is done by a linear correction of the proportional-term. With this adjustment, the user can manipulate the rise-time of the pressure waveform.

In addition to this core function, the controller module 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 event triggered an alarm, and an immediate release of air to drop to a safe pressure and not to exceed PIP. Both of these functionalities are fast, and respond, at the latest, within few hundred milliseconds. The controller also assesses whether numerical values and sensor readings 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.

The final functionality of the control module is the estimation of VTE (VTE stands for exhaled tidal volume), which is the volume of air that made it in- and out of the lung. We estimate this number by integrating the expiratory flow during expiration, and subtracting the baseline flow used to sustain PEEP (details in the accompanying manuscript):

1.1.14.2 Architecture of the Controller

In terms of software components, the Controller consists of one main `controller` class, that is instantiated in its own thread. This object receives sensor-data from HAL, and computes control parameters, to change the mechanical position of valves. The Controller also receives ventilation control parameters (see `set_control()`). All exchanged variables are mutex'd.

The Controller also feeds the `Logger` a continuous stream of [SensorValues](#) objects so as to store high-temporal resolution data, including the control signals.

The main **control loop** is `pvp.controller._start_mainloop()` which queries the Hardware for new variables, and performs a PID update using `.pvp.controller._PID_update()`.

The Controller is **configured** by the [values](#) module,

The Controller can be launched alone, but was not intended to be launched alone. The alarm functionality requires the UI.

Classes:

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

Functions:

<code>get_control_module(sim_mode, simulator_dt)</code>	Generates control module.
---	---------------------------

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

Bases: `object`

Abstract controller class for simulation/hardware.

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

- *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 be accessed though the **set_** and **get_** functions.

These functions act on COPIES of internal variables (`__` and `_`), that are sync'd every few iterations. How often this is done is adjusted by the variable `self._NUMBER_CONTROLL_LOOPS_UNTIL_UPDATE`. To avoid multiple threads manipulating the same variables at the same time, every manipulation of *COPY_* is surrounded by a thread lock.

Public Methods:

- `get_sensors()`: Returns a copy of the current sensor values.
- `get_alarms()`: Returns a List of all alarms, active and logged
- `get_control(ControlSetting)`: Sets a controll-setting. Is updated at latest within `self._NUMBER_CONTROLL_LOOPS_UNTIL_UPDATE`
- `get_past_waveforms()`: Returns a List of waveforms of pressure and volume during at the last N breath cycles, `N < self._RINGBUFFER_SIZE`, AND clears this archive.
- `start()`: Starts the main-loop of the controller
- `stop()`: Stops the main-loop of the controller
- `set_control()`: Set the control
- `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:

<code>__init__([save_logs, flush_every])</code>	Initializes the ControlModuleBase class.
<code>_initialize_set_to_COPY()</code>	Makes a copy of internal variables.
<code>_sensor_to_COPY()</code>	
<code>_controls_from_COPY()</code>	
<code>get_sensors()</code>	A method callable from the outside to get a copy of sensorValues
<code>get_alarms()</code>	A method callable from the outside to get a copy of the alarms, that the controller checks: High airway pressure, and technical alarms.
<code>set_control(control_setting)</code>	A method callable from the outside to set alarms.
<code>get_control(control_setting_name)</code>	A method callable from the outside to get current control settings.
<code>set_breath_detection(breath_detection)</code>	
<code>get_breath_detection()</code>	Return current state of autonomous breath detection
<code>_get_control_signal_in()</code>	Produces the INSPIRATORY control-signal that has been calculated in <code>_calculate_control_signal_in(dt)</code>
<code>_get_control_signal_out()</code>	Produces the EXPIRATORY control-signal for the different states, i.e. open/close.
<code>_control_reset()</code>	Resets the internal controller cycle to zero, i.e. restarts the breath cycle.
<code>_PID_update(dt)</code>	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.
<code>get_past_waveforms()</code>	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].
<code>_start_mainloop()</code>	Prototype method to start main PID loop.
<code>start()</code>	Method to start <code>_start_mainloop</code> as a thread.
<code>stop()</code>	Method to stop the main loop thread, and close the logfile.
<code>is_running()</code>	Public Method to assess whether the main loop thread is running.
<code>get_heartbeat()</code>	Returns an independent heart-beat of the controller, i.e. the internal loop counter incremented in <code>_start_mainloop</code> .

`__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()

__comptest(*phase*, *ls*, *selector*)

Helper function to identify the index the first occurrence of a number in *list* exceeding *threshold*, and returns *phase[idx]*

Parameters

- **phase** (*array*) – a list of numbers
- **list** (*array*) – array of bools with same length as *phase*
- **selector** (*string*) – ‘first’ or ‘last’ whichever is wanted

Returns *phase[idx]* where *idx* is first, or last point where numbers in *list* exceed *threshold*

Return type *float*

__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*, handed by the controller.

Return type *SensorValues*

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

A method callable from the outside to get a copy of the alarms, that the controller checks: High airway pressure, and technical alarms.

Returns A tuple of alarms

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

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

A method callable from the outside to set alarms. This updates the entries of *COPY* with new control values.

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

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

A method callable from the outside to get current control settings. This returns values of *COPY* to the outside world.

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

Returns *ControlSettings-Object* that contains relevant data

Return type *ControlSetting*

set_breath_detection(*breath_detection: bool*)

get_breath_detection() → *bool*

Return current state of autonomous breath detection

Returns *bool*

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

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

Parameters

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

__calculate_control_signal_in(*dt*)

Calculates the PID control signal by:

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

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

_get_control_signal_in()

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

Returns the numerical control signal for the inspiratory prop valve

Return type *float*

_get_control_signal_out()

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

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

Return type *float*

_control_reset()

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

__test_for_alarms()

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

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

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

__start_new_breathcycle()

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

- starts new breathcycle
- initializes new `__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` (*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.

`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:

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

```
__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.

```
__get_hal(**kwargs)
```

```
_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 breach

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:

`get_pressure()`

`set_flow_in(Qin, dt)`

`set_flow_out(Qout, dt)`

`update(dt)`

<code>OUupdate(variable, dt, mu, sigma, tau)</code>	This is a simple function to produce an OU process on <i>variable</i> .
---	---

<code>_reset()</code>	Resets Balloon to default settings.
-----------------------	-------------------------------------

get_pressure()

set_flow_in(Qin, dt)

set_flow_out(Qout, dt)

update(dt)

OUupdate(variable, dt, mu, sigma, tau)

This is a simple function to produce an OU process on *variable*. It is used as model for fluctuations in measurement variables.

Parameters

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

Returns value of “variable” at next time step

Return type `float`

_reset()

Resets Balloon to default settings.

```
class pvp.controller.control_module.ControlModuleSimulator(save_logs: bool = False,
                                                           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

- **save_logs** (*bool*, *optional*) – should logs be saved? (Useful for testing)
- **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:

<code>__init__</code> ([save_logs, simulator_dt, ...])	Initializes the ControlModuleBase with the simple simulation (for testing/dev).
<code>_sensor_to_COPY</code> ()	Make the sensor value object from current (simulated) measurements
<code>_start_mainloop</code> ()	This is the main loop.

```
__init__(save_logs: bool = False, simulator_dt=None, peep_valve_setting=5)
```

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

Parameters

- **save_logs** (*bool*, *optional*) – should logs be saved? (Useful for testing)
- **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

1.1.15 common module

1.1.15.1 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.

Classes:

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

Data:

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

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

Bases: `enum.Enum`

Canonical names of all values used in PVP.

Attributes:

PIP

PIP_TIME

PEEP

PEEP_TIME

continues on next page

Table 16 – continued from previous page

BREATHS_PER_MINUTE

INSPIRATION_TIME_SEC

IE_RATIO

FIO2

VTE

PRESSURE

FLOWOUT

PIP = 1**PIP_TIME = 2****PEEP = 3****PEEP_TIME = 4****BREATHS_PER_MINUTE = 5****INSPIRATION_TIME_SEC = 6****IE_RATIO = 7****FIO2 = 8****VTE = 9****PRESSURE = 10****FLOWOUT = 11**

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

Bases: `object`

Class to parameterize how a value is used in PVP.

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

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

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

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

note:

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

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

Methods:

`__init__`(name, units, abs_range, safe_range, ...)

param name Human-readable name of the value

`to_dict`()

Gather up all attributes and return as a dict!

Attributes:

<code>name</code>	Human readable name of value
<code>abs_range</code>	tuple of ints or floats setting the logical limit of the value, eg.
<code>safe_range</code>	tuple of ints or floats setting the safe ranges of the value,
<code>decimals</code>	The number of decimals of precision used when displaying the value
<code>default</code>	Default value, if any.
<code>control</code>	Whether or not the value is used to control ventilation
<code>sensor</code>	Whether or not the value is a measured sensor value
<code>display</code>	Whether the value should be created as a <code>gui.widgets.Display</code> widget.
<code>control_type</code>	If a control sets whether the control should use a slider or be set by recording recent sensor values.
<code>group</code>	Unused currently, but to be used to create subgroups of control & display widgets
<code>plot</code>	whether or not the value is plottable in the center plot window

continues on next page

Table 18 – continued from previous page

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

```
__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: *str*

Human readable name of value

Returns *str*

property abs_range: *tuple*

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*

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: `int`

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: `bool`

Whether or not the value is used to control ventilation

Returns bool

property sensor: `bool`

Whether or not the value is a measured sensor value

Returns bool

property display

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

Returns bool

property control_type

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

Returns None, “slider”, “record”

property group

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

Returns None, str

property plot

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

Returns bool

property plot_limits

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

Returns None, `typing.Tuple[ValueName]`

to_dict() → `dict`

Gather up all attributes and return as a dict!

Returns dict

```
pvp.common.values.VALUES = OrderedDict([(ValueName.PIP: 1, <pvp.common.values.Value object>),
(ValueName.PEEP: 3, <pvp.common.values.Value object>),
(ValueName.BREATHS_PER_MINUTE: 5, <pvp.common.values.Value object>),
(ValueName.INSPIRATION_TIME_SEC: 6, <pvp.common.values.Value object>),
(ValueName.IE_RATIO: 7, <pvp.common.values.Value object>), (ValueName.PIP_TIME: 2,
<pvp.common.values.Value object>), (ValueName.PEEP_TIME: 4, <pvp.common.values.Value object>),
(ValueName.PRESSURE: 10, <pvp.common.values.Value object>), (ValueName.VTE: 9,
<pvp.common.values.Value object>), (ValueName.FLOWOUT: 11, <pvp.common.values.Value object>),
(ValueName.FIO2: 8, <pvp.common.values.Value object>)])
```

Declaration of all values used by PVP

```
pvp.common.values.SENSOR = OrderedDict([(ValueName.PIP: 1, <pvp.common.values.Value object>),
(ValueName.PEEP: 3, <pvp.common.values.Value object>),
(ValueName.BREATHS_PER_MINUTE: 5, <pvp.common.values.Value object>),
(ValueName.INSPIRATION_TIME_SEC: 6, <pvp.common.values.Value object>),
(ValueName.PRESSURE: 10, <pvp.common.values.Value object>), (ValueName.VTE: 9,
<pvp.common.values.Value object>), (ValueName.FLOWOUT: 11, <pvp.common.values.Value object>),
(ValueName.FIO2: 8, <pvp.common.values.Value object>)])
```

Sensor values

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

```
pvp.common.values.CONTROL = OrderedDict([(ValueName.PIP: 1, <pvp.common.values.Value object>),
(ValueName.PEEP: 3, <pvp.common.values.Value object>),
(ValueName.BREATHS_PER_MINUTE: 5, <pvp.common.values.Value object>),
(ValueName.INSPIRATION_TIME_SEC: 6, <pvp.common.values.Value object>),
(ValueName.IE_RATIO: 7, <pvp.common.values.Value object>), (ValueName.PIP_TIME: 2,
<pvp.common.values.Value object>), (ValueName.PEEP_TIME: 4, <pvp.common.values.Value object>)])
```

Values to control but not monitor.

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

```
pvp.common.values.DISPLAY_MONITOR = OrderedDict([(ValueName.PIP: 1,
<pvp.common.values.Value object>), (ValueName.PEEP: 3, <pvp.common.values.Value object>),
(ValueName.BREATHS_PER_MINUTE: 5, <pvp.common.values.Value object>),
(ValueName.INSPIRATION_TIME_SEC: 6, <pvp.common.values.Value object>),
(ValueName.PRESSURE: 10, <pvp.common.values.Value object>), (ValueName.VTE: 9,
<pvp.common.values.Value object>), (ValueName.FLOWOUT: 11, <pvp.common.values.Value object>),
(ValueName.FIO2: 8, <pvp.common.values.Value object>)])
```

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 object>), (ValueName.PEEP: 3, <pvp.common.values.Value object>),
(ValueName.BREATHS_PER_MINUTE: 5, <pvp.common.values.Value object>),
(ValueName.INSPIRATION_TIME_SEC: 6, <pvp.common.values.Value object>),
(ValueName.IE_RATIO: 7, <pvp.common.values.Value object>), (ValueName.PIP_TIME: 2,
<pvp.common.values.Value object>)])
```

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 object>), (ValueName.FLOWOUT: 11, <pvp.common.values.Value object>),
(ValueName.FIO2: 8, <pvp.common.values.Value object>)])
```

Values that can be plotted

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

1.1.15.2 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:

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

```
class pvp.common.message.SensorValues(timestamp=None, loop_counter=None, breath_count=None,
                                       vals=typing.Optional[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`

Attributes:

<i>additional_values</i>	Additional attributes that are not <i>ValueName</i> s that are expected in each <i>SensorValues</i> message
--------------------------	---

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
```

```
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.Optional[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:

<code>__init__(name[, value, min_value, ...])</code>	Message containing ventilation control parameters.
--	--

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

Message containing ventilation control parameters.

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

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

Parameters

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

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

Bases: `object`

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

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

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

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

Bases: `object`

Class to save derived values, analogous to `SensorValues`.

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

Key difference: `SensorValues` come exclusively from the sensors, `DerivedValues` contain estimates of `I_PHASE_DURATION`, `PIP_TIME`, `PEEP_time`, `PIP`, `PIP_PLATEAU`, `PEEP`, and `VTE`. :param *timestamp*: :param *breath_count*: :param *I_phase_duration*: :param *pip_time*: :param *peep_time*: :param *pip*: :param *pip_plateau*: :param *peep*: :param *vte*:

1.1.15.3 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.

Data:

<code>__LOGGERS</code>	list of strings, which loggers have been created already.
------------------------	---

Functions:

<code>init_logger(module_name[, log_level, ...])</code>	Initialize a logger for logging events.
<code>update_logger_sizes()</code>	Adjust each logger's <code>maxBytes</code> attribute so that the total across all loggers is <code>prefs.LOGGING_MAX_BYTES</code>

Classes:

<code>DataLogger([compression_level])</code>	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) --- (time, controllsignal) --- derived_quantities (group) --- (time, Cycle No, I_PHASE_DURATION, PIP_TIME, PEEP_time, PIP, PIP_PLATEAU, PEEP, VTE) --- program_information (group) --- (version & githash).
--	---

```
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: Optional[int] = None, file_handler: bool = True) → logging.Logger
```

Initialize a logger for logging events.

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

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

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

otherwise configure and return the logger such that

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

Parameters

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

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

Returns Logger 4 u 2 use

Return type `logging.Logger`

`pvp.common.loggers.update_logger_sizes()`

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

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

Bases: `object`

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

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

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

Initialized the coontinuous numerical logger class.

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

Methods:

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

continues on next page

Table 26 – continued from previous page

<code>log2mat([filename])</code>	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: <code>dl = DataLogger() dl.log2mat(filename)</code> The file is saved at the same path as <code>.mat</code> file, where the content is represented as matlab-structs.
<code>log2csv([filename])</code>	Translates the compressed hdf5 into three csv files containing:

`__init__(compression_level: int = 9)`

Initialized the coontinuous numerical logger class.

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

`_open_logfile()`

Opens the hdf5 file and generates the file structure.

`close_logfile()`

Flushes & closes the open hdf file.

`store_program_data()`

Appends program metadata to the logfile: githash and version

`store_waveform_data(sensor_values: SensorValues, control_values: ControlValues)`

Appends a datapoint to the file for continuous logging of streaming data. NOTE: Not flushed yet.

Parameters

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

`store_control_command(control_setting: ControlSetting)`

Appends a datapoint to the event-table, derived from ControlSettings

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

`store_derived_data(derived_values: DerivedValues)`

Appends a datapoint to the event-table, derived the continuous data (PIP, PEEP etc.)

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

`flush_logfile()`

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

`check_files()`

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

`rotation_newfile()`

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

`load_file(filename=None)`

This loads a hdf5 file, and returns data to the user as a dictionary with two keys: `waveform_data` and `control_data`

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

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

Return type dictionary

log2mat(*filename=None*)

Translates the compressed hdf5 into a matlab file containing a matlab struct. This struct has the same structure as the hdf5 file, but is not compressed. Use for any file:

```
dl = DataLogger() dl.log2mat(filename)
```

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

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

log2csv(*filename=None*)

Translates the compressed hdf5 into three csv files containing:

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

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

```
dl = DataLogger() dl.log2csv(filename)
```

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

1.1.15.4 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:

[*_DIRECTORIES*](#)

Directories to ensure are created and added to prefs.

continues on next page

Table 27 – continued from previous page

<code>_DEFAULTS</code>	Declare all available parameters and set default values.
------------------------	--

Functions:

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

`pvp.common.prefs._PREF_MANAGER = <multiprocessing.managers.SyncManager object>`

The multiprocessing.Manager that stores prefs during system operation

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

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

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

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

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

Locks access to `prefs_fn`

Type `mp.Lock`

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

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

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

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

Type `bool`

`pvp.common.prefs._DEFAULTS = {'BREATH_DETECTION': True, 'BREATH_PRESSURE_DROP': 4, 'CONTROLLER_LOOPS_UNTIL_UPDATE': 1, 'CONTROLLER_LOOP_UPDATE_TIME': 0.0, 'CONTROLLER_LOOP_UPDATE_TIME_SIMULATOR': 0.005, 'CONTROLLER_MAX_FLOW': 10, 'CONTROLLER_MAX_PRESSURE': 100, 'CONTROLLER_MAX_STUCK_SENSOR': 5, 'CONTROLLER_RINGBUFFER_SIZE': 100, 'COUGH_DURATION': 0.1, 'ENABLE_DIALOGS': True, 'ENABLE_WARNINGS': True, 'GUI_STATE_FN': 'gui_state.json', 'GUI_UPDATE_TIME': 0.05, 'HEARTBEAT_TIMEOUT': 0.02, 'LOGGING_MAX_BYTES': 2147483648, 'LOGGING_MAX_FILES': 5, 'LOGLEVEL': 'WARNING', 'OXYGEN_READ_FREQUENCY': 2, 'PREFS_FN': None, 'TIMEOUT': 0.05, 'TIME_FIRST_START': None}`

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

`pvplib.common.prefs.get_pref(key: Optional[str] = None)`

Get global configuration value

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

`pvplib.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 pvplib 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

`pvplib.common.prefs.save_prefs(prefs_fn: Optional[str] = None)`

Dumps loaded prefs to PREFERENCES_FN.

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

`pvplib.common.prefs.make_dirs()`

ensures `_DIRECTORIES` are created and added to prefs.

`pvplib.common.prefs.init()`

Initialize prefs. Called in `pvplib.__init__.py` to ensure prefs are initialized before anything else.

1.1.15.5 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:

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

`pvplib.common.unit_conversion.cmH2O_to_hPa(pressure: float) → float`

Convert cmH2O to hPa

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

Returns Pressure in hPa (pressure / 1.0197162129779)

Return type `float`

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

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

Returns Pressure in cmH2O (`pressure * 1.0197162129779`)

Return type `float`

`pvp.common.unit_conversion.rounded_string(value: float, decimals: int = 0) → str`
Create a rounded string of a number that doesnt have trailing .0 when decimals = 0

Parameters

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

Returns Clean rounded string version of number

Return type `str`

1.1.15.6 utils

Exceptions:

`TimeoutException`

Functions:

`time_limit(seconds)`

<code>timeout(func)</code>	Defines a decorator for a 50ms timeout.
----------------------------	---

<code>get_version()</code>	Returns PVP version, and if available githash, as a string.
----------------------------	---

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")`

`pvp.common.utils.get_version()`

Returns PVP version, and if available githash, as a string.

1.1.15.7 fashion

Decorators for dangerous functions

Functions:

`pigpio_command(func)`

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

1.1.16 pvp.io package

1.1.16.1 pvp.io.hal module

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

Classes:

<code>Hal([config_file])</code>	Hardware Abstraction Layer for ventilator hardware.
---------------------------------	---

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

Bases: `object`

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

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

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

The Hal will:

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

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

```
analog_sensor = AnalogSensor( pig=self._pig, adc=self._adc, MUX=0, offset_voltage=0.25, out-
    put_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:

<code>__init__([config_file])</code>	Initializes HAL from <code>config_file</code> .
--------------------------------------	---

Attributes:

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

`__init__(config_file='pvp/io/config/devices.ini')`

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

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

The Hal will:

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

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

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

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

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

property pressure: float

Returns the pressure from the primary pressure sensor.

property oxygen: float

Returns the oxygen concentration from the primary oxygen sensor.

property aux_pressure: float

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: float

The measured flow rate inspiratory side.

property flow_ex: float

The measured flow rate expiratory side.

property setpoint_in: float

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

property setpoint_ex: float

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

1.1.16.2 devices

A module for ventilator hardware device drivers

1.1.17 Alarm

1.1.17.1 Alarm System Overview

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

1.1.17.2 Alarm Modules

Alarm Manager

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

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

Classes:

<i>Alarm_Manager()</i>	The Alarm Manager
------------------------	-------------------

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

The Alarm Manager

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

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

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/alterd.

Type list

cleared_alarms

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

Type list

snoozed_alarms

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

Type dict

callbacks

list of callables to send *Alarm* objects to

Type list

depends_callbacks

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

Type list

rules

A dict mapping *AlarmType* to *Alarm_Rule* .

Type dict

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

Attributes:

<i>_instance</i>
<i>active_alarms</i>
<i>logged_alarms</i>
<i>dependencies</i>
<i>pending_clears</i>
<i>cleared_alarms</i>
<i>snoozed_alarms</i>
<i>callbacks</i>
<i>depends_callbacks</i>
<i>rules</i>
<i>logger</i>

Methods:

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

continues on next page

Table 38 – continued from previous page

<code>clear_all_alarms()</code>	call <code>Alarm_Manager.deactivate_alarm()</code> for all active alarms.
<code>reset()</code>	Reset all conditions, callbacks, and other stateful attributes and clear alarms

`_instance = None`

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

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

`dependencies = {}`

`pending_clears = []`

`cleared_alarms = []`

`snoozed_alarms = {}`

`callbacks = []`

`depends_callbacks = []`

`rules: Dict[pvp.alarm.AlarmType, pvp.alarm.rule.Alarm_Rule] = {}`

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

`load_rules()`

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

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

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

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

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

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

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

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

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

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

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

Parameters

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

emit_alarm(*alarm_type*: [pvp.alarm.AlarmType](#), *severity*: [pvp.alarm.AlarmSeverity](#))

Emit alarm (by calling all callbacks with it).

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

Parameters

- **alarm_type** ([AlarmType](#)) –
- **severity** ([AlarmSeverity](#)) –

deactivate_alarm(*alarm*: (<enum 'AlarmType'>, <class 'pvp.alarm.alarm.Alarm'>))

Mark an alarm's internal active flags and remove from [active_alarms](#)

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

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

Parameters **alarm** ([AlarmType](#) , [Alarm](#)) – Alarm to deactivate

dismiss_alarm(*alarm_type*: [pvp.alarm.AlarmType](#), *duration*: *Optional[float] = None*)

GUI or other object requests an alarm to be dismissed & deactivated

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

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

Parameters

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

get_alarm_severity(*alarm_type*: [pvp.alarm.AlarmType](#))

Get the severity of an Alarm

Parameters **alarm_type** ([AlarmType](#)) – Alarm type to check

Returns [AlarmSeverity](#)

register_alarm(*alarm*: [pvp.alarm.alarm.Alarm](#))

Be given an already created alarm and emit to callbacks.

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

Parameters **alarm** ([Alarm](#)) –

register_dependency(*condition*: [pvp.alarm.condition.Condition](#), *dependency*: *dict*, *severity*: [pvp.alarm.AlarmSeverity](#))

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

Parameters

- **condition** (*dict*) – Condition as defined in an [Alarm_Rule](#)

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

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

Update Condition objects that update their value according to some control parameter

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

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:

<i>Alarm</i> (<i>alarm_type</i> , <i>severity</i> [, <i>start_time</i> , ...])	Representation of alarm status and parameters
---	---

```
class pvp.alarm.alarm.Alarm(alarm_type: pvp.alarm.AlarmType, severity: pvp.alarm.AlarmSeverity,  
                           start_time: Optional[float] = None, latch: bool = True, cause: Optional[list] =  
                           None, value=None, message=None)
```

Representation of alarm status and parameters

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

Parameters

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

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

id

unique alarm ID

Type *int*

end_time

If None, alarm has not ended. otherwise timestamp

Type *None, float*

active

Whether or not the alarm is currently active

Type *bool*

Attributes:

<i>id_counter</i>	used to generate unique IDs for each alarm
<i>severity</i>	Alarm Severity, property without setter to prevent change after instantiation
<i>alarm_type</i>	Alarm Type, property without setter to prevent change after instantiation

Methods:

<i>__init__</i> (alarm_type, severity[, start_time, ...])	param alarm_type Type of alarm
<i>deactivate</i> ()	If active, register an end time and set as active == False Returns:

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: *Optional[float]* = None, latch: *bool* = True, cause: *Optional[list]* = None, value=None, message=None)

Parameters

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

id

unique alarm ID

Type *int*

end_time

If None, alarm has not ended. otherwise timestamp

Type `None, float`

active

Whether or not the alarm is currently active

Type `bool`

id: `int`

property severity: `pvp.alarm.AlarmSeverity`

Alarm Severity, property without setter to prevent change after instantiation

Returns `AlarmSeverity`

property alarm_type: `pvp.alarm.AlarmType`

Alarm Type, property without setter to prevent change after instantiation

Returns `AlarmType`

deactivate()

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

Alarm Rule

One `AlarmRule` is defined for each `AlarmType` in `ALARM_RULES`.

An alarm rule defines:

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

Example

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

First we define the name and behavior of the alarm:

```
AlarmRule(  
    name = AlarmType.LOW_PRESSURE,  
    latch = False,  
)
```

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

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

Starting with the `LOW` severity alarm:

```
conditions = (  
    (  
        AlarmSeverity.LOW,  
        condition.ValueCondition(  
            value_name=ValueName.PIP,  
            limit=VALUES[ValueName.PIP]['safe_range'][0],  
        )  
    )  
)
```

(continues on next page)

(continued from previous page)

```

    mode='min',
    depends={
        'value_name': ValueName.PIP,
        'value_attr': 'value',
        'condition_attr': 'limit',
        'transform': lambda x : x-(x*0.10)
    })
),
# ... continued in next block

```

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

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

Next, we define the MEDIUM severity alarm condition:

```

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

```

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

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

Full source for this example and all alarm rules can be found [here](#)

Module Documentation

Class to declare alarm rules

Classes:

Alarm_Rule(name, conditions[, latch, technical])

- name of rule
-

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

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

Methods:

<i>check</i> (sensor_values)	Check all of our conditions .
<i>reset</i> ()	

Attributes:

<i>severity</i>	Last Alarm Severity from .check() :returns: <i>AlarmSeverity</i>
<i>depends</i>	Get all ValueNames whose alarm limits depend on this alarm rule :returns: list[ValueName]
<i>value_names</i>	Get all ValueNames specified as value_names in alarm conditions

check(*sensor_values*)

Check all of our conditions .

Parameters *sensor_values* –

Returns:

property severity

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

reset()

property depends: List[*pvp.common.values.ValueName*]

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

property value_names: List[*pvp.common.values.ValueName*]

Get all ValueNames specified as value_names in alarm conditions

Returns list[ValueName]

Alarm Condition

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

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

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

Conditions can

Functions:

get_alarm_manager()

Classes:

<i>Condition</i> ([depends])	Base class for specifying alarm test conditions
<i>ValueCondition</i> (value_name, limit, mode, ...)	Value is greater or lesser than some max/min
<i>CycleValueCondition</i> (n_cycles, *args, **kwargs)	Value goes out of range for a specific number of breath cycles
<i>TimeValueCondition</i> (time, *args, **kwargs)	value goes out of range for specific amount of time
<i>AlarmSeverityCondition</i> (alarm_type, severity)	Alarm is above or below a certain severity.
<i>CycleAlarmSeverityCondition</i> (n_cycles, *args, ...)	alarm goes out of range for a specific number of breath cycles

`pvp.alarm.condition.get_alarm_manager()`

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

Bases: `object`

Base class for specifying alarm test conditions

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

Condition objects can be added together to create compound conditions.

_child

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

Type *Condition*

Parameters

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

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

- ***args** –
- ****kwargs** –

Methods:

<code>__init__([depends])</code>	param depends
<code>check(sensor_values)</code>	Every Condition subclass needs to define this method that accepts <i>SensorValues</i> and returns a boolean
<code>reset()</code>	If a condition is stateful, need to provide some method of resetting the state

Attributes:

<code>manager</code>	The active alarm manager, used to get status of alarms
----------------------	--

`__init__(depends: Optional[dict] = None, *args, **kwargs)`

Parameters

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

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

- ***args** –
- ****kwargs** –

property manager

The active alarm manager, used to get status of alarms

Returns *pvp.alarm.alarm_manager.Alarm_Manager*

check(*sensor_values*) → bool

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

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

Returns bool

reset()

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

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

Bases: *pvp.alarm.condition.Condition*

Value is greater or lesser than some max/min

Parameters

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

- ****kwargs** –

operator

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

Type callable

Methods:

<code>__init__(value_name, limit, mode, *args, ...)</code>	param value_name Which value to check
<code>check(sensor_values)</code>	Check that the relevant value in SensorValues is either greater or lesser than the limit
<code>reset()</code>	not stateful, do nothing.

Attributes:

<code>mode</code>	One of 'min' or 'max', defines how the incoming sensor values are compared to the set value
-------------------	---

```
__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

Attributes:

<i>n_cycles</i>	Number of cycles required
-----------------	---------------------------

Methods:

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

property `n_cycles`: *int*

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:

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

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

Parameters

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

check(*sensor_values*)

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

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

Returns bool

reset()

not stateful, do nothing.

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

Bases: *pvp.alarm.condition.Condition*

Alarm is above or below a certain severity.

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

Parameters

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

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

- ***args** –
- ****kwargs** –

Methods:

<code>__init__(alarm_type, severity[, mode])</code>	Alarm is above or below a certain severity.
<code>check([sensor_values])</code>	Every Condition subclass needs to define this method that accepts <i>SensorValues</i> and returns a boolean
<code>reset()</code>	If a condition is stateful, need to provide some method of resetting the state

Attributes:

<code>mode</code>	'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'.
-------------------	---

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

Alarm is above or below a certain severity.

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

Parameters

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

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

- ***args** –
- ****kwargs** –

property mode: *str*

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

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

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

Return type *str*

check(*sensor_values=None*)

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

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

Returns bool

reset()

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

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

Bases: [pvp.alarm.condition.AlarmSeverityCondition](#)

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

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

_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** –

Attributes:

[n_cycles](#)

Methods:

check (<i>sensor_values</i>)	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

property [n_cycles](#)

check(*sensor_values*)

Every Condition subclass needs to define this method that accepts [SensorValues](#) and returns a boolean

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

Returns `bool`

reset()

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

1.1.17.3 Main Alarm Module**Classes:**

<i>AlarmType</i> (value)	An enumeration.
<i>AlarmSeverity</i> (value)	An enumeration.

Data:

<i>ALARM_RULES</i>	Definitions of all <i>AlarmRule</i> s used by the <i>Alarm_Manager</i>
--------------------	--

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

An enumeration.

Attributes:

<i>LOW_PRESSURE</i>	
<i>HIGH_PRESSURE</i>	
<i>LOW_VTE</i>	
<i>HIGH_VTE</i>	
<i>LOW_PEEP</i>	
<i>HIGH_PEEP</i>	
<i>LOW_O2</i>	
<i>HIGH_O2</i>	
<i>OBSTRUCTION</i>	
<i>LEAK</i>	
<i>SENSORS_STUCK</i>	
<i>BAD_SENSOR_READINGS</i>	
<i>MISSED_HEARTBEAT</i>	
<i>human_name</i>	Replace .name underscores with spaces

LOW_PRESSURE = 1

HIGH_PRESSURE = 2


```

LOW_VTE = 3
HIGH_VTE = 4
LOW_PEEP = 5
HIGH_PEEP = 6
LOW_O2 = 7
HIGH_O2 = 8
OBSTRUCTION = 9
LEAK = 10
SENSORS_STUCK = 11
BAD_SENSOR_READINGS = 12
MISSED_HEARTBEAT = 13
property human_name: str
    Replace .name underscores with spaces
class pvp.alarm.AlarmSeverity(value)
    An enumeration.

    Attributes:

```

HIGH

MEDIUM

LOW

OFF

TECHNICAL

```

HIGH = 3
MEDIUM = 2
LOW = 1
OFF = 0
TECHNICAL = -1

pvp.alarm.ALARM_RULES = OrderedDict([(AlarmType.LOW_PRESSURE: 1,
<pvp.alarm.rule.Alarm_Rule object>), (AlarmType.HIGH_PRESSURE: 2,
<pvp.alarm.rule.Alarm_Rule object>), (AlarmType.LOW_VTE: 3, <pvp.alarm.rule.Alarm_Rule
object>), (AlarmType.HIGH_VTE: 4, <pvp.alarm.rule.Alarm_Rule object>),
(<AlarmType.LOW_PEEP: 5>, <pvp.alarm.rule.Alarm_Rule object>), (<AlarmType.HIGH_PEEP: 6>,
<pvp.alarm.rule.Alarm_Rule object>), (<AlarmType.LOW_O2: 7>, <pvp.alarm.rule.Alarm_Rule
object>), (<AlarmType.HIGH_O2: 8>, <pvp.alarm.rule.Alarm_Rule object>)])
    Definitions of all Alarm_Rule s used by the Alarm_Manager
    See definitions here

```

1.1.18 coordinator module

The coordinator provides an interface between the process threads, and facilitates inter-process communication. It is a wrapper around xml-rpc, which allowed us to use defined data-structures such as *SensorValues*.

1.1.18.1 Submodules

1.1.18.2 coordinator

Classes:

`CoordinatorBase([sim_mode])`

`CoordinatorLocal([sim_mode])`

param sim_mode

`CoordinatorRemote([sim_mode])`

Functions:

`get_coordinator([single_process, sim_mode])`

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

Bases: `object`

Methods:

`get_sensors()`

`get_alarms()`

`set_control(control_setting)`

`get_control(control_setting_name)`

`set_breath_detection(breath_detection)`

`get_breath_detection()`

`start()`

`is_running()`

`kill()`

`stop()`

get_sensors() \rightarrow `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_breath_detection() → bool
start()
is_running() → bool
kill()
stop()

```

```

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

```

Parameters **sim_mode** –

_is_running
 .set() when thread should stop

Type `threading.Event`

Methods:

```
__init__([sim_mode])
```

param **sim_mode**

```
get_sensors()
```

```
get_alarms()
```

```
set_control(control_setting)
```

```
get_control(control_setting_name)
```

```
set_breath_detection(breath_detection)
```

```
get_breath_detection()
```

```
start()
```

Start the coordinator.

```
is_running()
```

Test whether the whole system is running

```
stop()
```

Stop the coordinator.

```
kill()
```

```
__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_breath_detection() → bool
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:

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

```
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_breath_detection() → bool
start()
    Start the coordinator. This does a soft start (not allocating a process).
```


is_running() → *bool*

Test whether the whole system is running

stop()

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

kill()

Stop the coordinator and end the whole program

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

1.1.18.3 ipc

Functions:

get_sensors()

get_alarms()

set_control(control_setting)

get_control(control_setting_name)

set_breath_detection(breath_detection)

get_breath_detection()

rpc_server_main(sim_mode, serve_event[, ...])

get_rpc_client()

`pvp.coordinator.rpc.get_sensors()`

`pvp.coordinator.rpc.get_alarms()`

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

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

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

`pvp.coordinator.rpc.get_breath_detection()`

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

`pvp.coordinator.rpc.get_rpc_client()`

1.1.18.4 process_manager

Classes:

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

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

Bases: `object`

Methods:

start_process()

try_stop_process()

restart_process()

heartbeat(timestamp)

start_process()

try_stop_process()

restart_process()

heartbeat(*timestamp*)

1.1.19 Index

- `genindex`
- `modindex`

MEDICAL DISCLAIMER

PVP1 is not a regulated or clinically validated medical device. We have not yet performed testing for safety or efficacy on living organisms. All material described herein should be used at your own risk and do not represent a medical recommendation. PVP1 is currently recommended only for research purposes.

This website is not connected to, endorsed by, or representative of the view of Princeton University. Neither the authors nor Princeton University assume any liability or responsibility for any consequences, damages, or loss caused or alleged to be caused directly or indirectly for any action or inaction taken based on or made in reliance on the information or material discussed herein or linked to from this website.

PVP1 is under continuous development and the information here may not be up to date, nor is any guarantee made as such. Neither the authors nor Princeton University are liable for any damage or loss related to the accuracy, completeness or timeliness of any information described or linked to from this website.

By continuing to watch or read this, you are acknowledging and accepting this disclaimer.

PYTHON MODULE INDEX

p

- `pvp.alarm`, [154](#)
- `pvp.alarm.alarm`, [142](#)
- `pvp.alarm.alarm_manager`, [137](#)
- `pvp.alarm.condition`, [147](#)
- `pvp.alarm.rule`, [146](#)
- `pvp.common.fashion`, [135](#)
- `pvp.common.loggers`, [127](#)
- `pvp.common.message`, [124](#)
- `pvp.common.prefs`, [130](#)
- `pvp.common.unit_conversion`, [133](#)
- `pvp.common.utils`, [134](#)
- `pvp.common.values`, [118](#)
- `pvp.controller.control_module`, [109](#)
- `pvp.coordinator.coordinator`, [156](#)
- `pvp.coordinator.process_manager`, [160](#)
- `pvp.coordinator.rpc`, [159](#)
- `pvp.gui.styles`, [106](#)
- `pvp.io`, [135](#)
- `pvp.io.devices`, [137](#)
- `pvp.io.hal`, [135](#)

Symbols

[_DEFAULTS](#) (in module `pvp.common.prefs`), 131
[_DIRECTORIES](#) (in module `pvp.common.prefs`), 131
[_LOCK](#) (in module `pvp.common.prefs`), 131
[_LOGGER](#) (in module `pvp.common.prefs`), 131
[_LOGGERS](#) (in module `pvp.common.loggers`), 127
[_PID_update\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 114
[_PREFS](#) (in module `pvp.common.prefs`), 131
[_PREF_MANAGER](#) (in module `pvp.common.prefs`), 131
[__SimulatedPropValve\(\)](#)
 (`pvp.controller.control_module.ControlModuleSimulator` method), 117
[__SimulatedSolenoid\(\)](#)
 (`pvp.controller.control_module.ControlModuleSimulator` method), 117
[__analyze_last_waveform\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 112
[__calculate_control_signal_in\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 113
[__comp_test\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 112
[__get_PID_error\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 113
[__get_hal\(\)](#) (`pvp.controller.control_module.ControlModuleDevice` method), 115
[__init__\(\)](#) (`pvp.alarm.alarm.Alarm` method), 143
[__init__\(\)](#) (`pvp.alarm.condition.AlarmSeverityCondition` method), 152
[__init__\(\)](#) (`pvp.alarm.condition.Condition` method), 148
[__init__\(\)](#) (`pvp.alarm.condition.TimeValueCondition` method), 151
[__init__\(\)](#) (`pvp.alarm.condition.ValueCondition` method), 149
[__init__\(\)](#) (`pvp.common.loggers.DataLogger` method), 129
[__init__\(\)](#) (`pvp.common.message.ControlSetting` method), 126
[__init__\(\)](#) (`pvp.common.message.SensorValues` method), 125
[__init__\(\)](#) (`pvp.common.values.Value` method), 121
[__init__\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 111
[__init__\(\)](#) (`pvp.controller.control_module.ControlModuleDevice` method), 115
[__init__\(\)](#) (`pvp.controller.control_module.ControlModuleSimulator` method), 117
[__init__\(\)](#) (`pvp.coordinator.coordinator.CoordinatorLocal` method), 157
[__init__\(\)](#) (`pvp.io.hal.Hal` method), 136
[_save_values\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 114
[__start_new_breathcycle\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 113
[__test_for_alarms\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 113
[_child](#) (`pvp.alarm.condition.Condition` attribute), 147
[_control_reset\(\)](#) (`pvp.controller.control_module.ControlModuleBase` method), 113
[_controls_from_COPY\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 112
[_get_HAL\(\)](#) (`pvp.controller.control_module.ControlModuleDevice` method), 115
[_get_control_signal_in\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 113
[_get_control_signal_out\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 113
[_initialize_set_to_COPY\(\)](#)
 (`pvp.controller.control_module.ControlModuleBase` method), 112
[_instance](#) (`pvp.alarm.alarm_manager.Alarm_Manager` attribute), 140
[_is_running](#) (`pvp.coordinator.coordinator.CoordinatorLocal` attribute), 157
[_mid_check](#) (`pvp.alarm.condition.CycleAlarmSeverityCondition` attribute), 153

- `_mid_check` (*pvp.alarm.condition.CycleValueCondition* attribute), 150
- `_open_logfile()` (*pvp.common.loggers.DataLogger* method), 129
- `_reset()` (*pvp.controller.control_module.Balloon_Simulator* method), 116
- `_sensor_to_COPY()` (*pvp.controller.control_module.ControlModuleBase* method), 112
- `_sensor_to_COPY()` (*pvp.controller.control_module.ControlModuleDevice* method), 115
- `_sensor_to_COPY()` (*pvp.controller.control_module.ControlModuleSimulator* method), 117
- `_set_HAL()` (*pvp.controller.control_module.ControlModuleDevice* method), 115
- `_start_cycle` (*pvp.alarm.condition.CycleAlarmSeverityCondition* attribute), 153
- `_start_cycle` (*pvp.alarm.condition.CycleValueCondition* attribute), 150
- `_start_mainloop()` (*pvp.controller.control_module.ControlModuleBase* method), 114
- `_start_mainloop()` (*pvp.controller.control_module.ControlModuleDevice* method), 116
- `_start_mainloop()` (*pvp.controller.control_module.ControlModuleSimulator* method), 117
- A**
- `abs_range` (*pvp.common.values.Value* property), 121
- `active` (*pvp.alarm.alarm.Alarm* attribute), 143, 144
- `active_alarms` (*pvp.alarm.alarm_manager.Alarm_Manager* attribute), 138, 140
- `add_callback()` (*pvp.alarm.alarm_manager.Alarm_Manager* method), 142
- `add_dependency_callback()` (*pvp.alarm.alarm_manager.Alarm_Manager* method), 142
- `additional_values` (*pvp.common.message.SensorValues* attribute), 125
- `Alarm` (class in *pvp.alarm.alarm*), 142
- `Alarm_Manager` (class in *pvp.alarm.alarm_manager*), 137
- `Alarm_Rule` (class in *pvp.alarm.rule*), 146
- `ALARM_RULES` (in module *pvp.alarm*), 155
- `alarm_type` (*pvp.alarm.alarm.Alarm* property), 144
- `AlarmSeverity` (class in *pvp.alarm*), 155
- `AlarmSeverityCondition` (class in *pvp.alarm.condition*), 151
- `AlarmType` (class in *pvp.alarm*), 154
- `aux_pressure` (*pvp.io.hal.Hal* property), 137
- B**
- `BAD_SENSOR_READINGS` (*pvp.alarm.AlarmType* attribute), 155
- `Balloon_Simulator` (class in *pvp.controller.control_module*), 116
- `BREATHS_PER_MINUTE` (*pvp.common.values.ValueName* attribute), 119
- C**
- `callbacks` (*pvp.alarm.alarm_manager.Alarm_Manager* attribute), 138, 140
- `check()` (*pvp.alarm.condition.AlarmSeverityCondition* method), 152
- `check()` (*pvp.alarm.condition.Condition* method), 148
- `check()` (*pvp.alarm.condition.CycleAlarmSeverityCondition* method), 153
- `check()` (*pvp.alarm.condition.CycleValueCondition* method), 150
- `check()` (*pvp.alarm.condition.TimeValueCondition* method), 151
- `check()` (*pvp.alarm.condition.ValueCondition* method), 149
- `check()` (*pvp.alarm.rule.Alarm_Rule* method), 146
- `check_rule()` (*pvp.common.loggers.DataLogger* method), 129
- `clear_alarms()` (*pvp.alarm.alarm_manager.Alarm_Manager* method), 142
- `cleared_alarms` (*pvp.alarm.alarm_manager.Alarm_Manager* attribute), 138, 140
- `close_logfile()` (*pvp.common.loggers.DataLogger* method), 129
- `cmH2O_to_hPa()` (in module *pvp.common.unit_conversion*), 133
- `Condition` (class in *pvp.alarm.condition*), 147
- `CONTROL` (in module *pvp.common.values*), 123
- `control` (*pvp.common.values.Value* property), 122
- `control_type` (*pvp.common.values.Value* property), 122
- `ControlModuleBase` (class in *pvp.controller.control_module*), 110
- `ControlModuleDevice` (class in *pvp.controller.control_module*), 114
- `ControlModuleSimulator` (class in *pvp.controller.control_module*), 116
- `ControlSetting` (class in *pvp.common.message*), 125
- `ControlValues` (class in *pvp.common.message*), 126
- `CoordinatorBase` (class in *pvp.coordinator.coordinator*), 156
- `CoordinatorLocal` (class in *pvp.coordinator.coordinator*), 157
- `CoordinatorRemote` (class in *pvp.coordinator.coordinator*), 158
- `CycleAlarmSeverityCondition` (class in *pvp.alarm.condition*), 152
- `CycleValueCondition` (class in *pvp.alarm.condition*), 149

D

`DataLogger` (class in `pvp.common.loggers`), 128

`deactivate()` (`pvp.alarm.alarm.Alarm` method), 144

`deactivate_alarm()` (`pvp.alarm.alarm_manager.Alarm_Manager` method), 141

`decimals` (`pvp.common.values.Value` property), 122

`default` (`pvp.common.values.Value` property), 122

`dependencies` (`pvp.alarm.alarm_manager.Alarm_Manager` attribute), 138, 140

`depends` (`pvp.alarm.rule.Alarm_Rule` property), 146

`depends_callbacks` (`pvp.alarm.alarm_manager.Alarm_Manager` attribute), 138, 140

`DerivedValues` (class in `pvp.common.message`), 126

`dismiss_alarm()` (`pvp.alarm.alarm_manager.Alarm_Manager` method), 141

`display` (`pvp.common.values.Value` property), 122

`DISPLAY_CONTROL` (in module `pvp.common.values`), 123

`DISPLAY_MONITOR` (in module `pvp.common.values`), 123

E

`emit_alarm()` (`pvp.alarm.alarm_manager.Alarm_Manager` method), 140

`end_time` (`pvp.alarm.alarm.Alarm` attribute), 143

F

`FI02` (`pvp.common.values.ValueName` attribute), 119

`flow_ex` (`pvp.io.hal.Hal` property), 137

`flow_in` (`pvp.io.hal.Hal` property), 137

`FLOWOUT` (`pvp.common.values.ValueName` attribute), 119

`flush_logfile()` (`pvp.common.loggers.DataLogger` method), 129

G

`get_alarm_manager()` (in module `pvp.alarm.condition`), 147

`get_alarm_severity()` (`pvp.alarm.alarm_manager.Alarm_Manager` method), 141

`get_alarms()` (in module `pvp.coordinator.rpc`), 159

`get_alarms()` (`pvp.controller.control_module.ControlModuleBase` method), 112

`get_alarms()` (`pvp.coordinator.coordinator.CoordinatorBase` method), 156

`get_alarms()` (`pvp.coordinator.coordinator.CoordinatorLocal` method), 158

`get_alarms()` (`pvp.coordinator.coordinator.CoordinatorRemote` method), 158

`get_breath_detection()` (in module `pvp.coordinator.rpc`), 159

`get_breath_detection()` (`pvp.controller.control_module.ControlModuleBase` method), 113

`get_breath_detection()` (`pvp.coordinator.coordinator.CoordinatorBase` method), 157

`get_breath_detection()` (`pvp.coordinator.coordinator.CoordinatorLocal` method), 158

`get_breath_detection()` (`pvp.coordinator.coordinator.CoordinatorRemote` method), 158

`get_control()` (in module `pvp.coordinator.rpc`), 159

`get_control()` (`pvp.controller.control_module.ControlModuleBase` method), 112

`get_control()` (`pvp.coordinator.coordinator.CoordinatorBase` method), 157

`get_control()` (`pvp.coordinator.coordinator.CoordinatorLocal` method), 158

`get_control()` (`pvp.coordinator.coordinator.CoordinatorRemote` method), 158

`get_control_module()` (in module `pvp.controller.control_module`), 117

`get_coordinator()` (in module `pvp.coordinator.coordinator`), 159

`get_heartbeat()` (`pvp.controller.control_module.ControlModuleBase` method), 114

`get_past_waveforms()` (`pvp.controller.control_module.ControlModuleBase` method), 114

`get_pref()` (in module `pvp.common.prefs`), 132

`get_pressure()` (`pvp.controller.control_module.Balloon_Simulator` method), 116

`get_rpc_client()` (in module `pvp.coordinator.rpc`), 159

`get_sensors()` (in module `pvp.coordinator.rpc`), 159

`get_sensors()` (`pvp.controller.control_module.ControlModuleBase` method), 112

`get_sensors()` (`pvp.coordinator.coordinator.CoordinatorBase` method), 156

`get_sensors()` (`pvp.coordinator.coordinator.CoordinatorLocal` method), 157

`get_sensors()` (`pvp.coordinator.coordinator.CoordinatorRemote` method), 158

`get_version()` (in module `pvp.common.utils`), 134

`group` (`pvp.common.values.Value` property), 122

`hal`

`hal`

`Hal` (class in `pvp.io.hal`), 135

`heartbeat()` (`pvp.coordinator.process_manager.ProcessManager` method), 160

`HIGH` (`pvp.alarm.AlarmSeverity` attribute), 155

`HIGH_02` (`pvp.alarm.AlarmType` attribute), 155

`HIGH_PEEP` (`pvp.alarm.AlarmType` attribute), 155

`HIGH_PRESSURE` (`pvp.alarm.AlarmType` attribute), 154

`HIGH_VTE` (`pvp.alarm.AlarmType` attribute), 155

hPa_to_cmH2O() (in module *pvp.common.unit_conversion*), 134
 human_name (*pvp.alarm.AlarmType* property), 155

I

id (*pvp.alarm.alarm.Alarm* attribute), 143, 144
 id_counter (*pvp.alarm.alarm.Alarm* attribute), 143
 IE_RATIO (*pvp.common.values.ValueName* attribute), 119
 init() (in module *pvp.common.prefs*), 133
 init_logger() (in module *pvp.common.loggers*), 127
 INSPIRATION_TIME_SEC (*pvp.common.values.ValueName* attribute), 119
 is_running() (*pvp.controller.control_module.ControlModuleBase* method), 114
 is_running() (*pvp.coordinator.coordinator.CoordinatorBase* method), 157
 is_running() (*pvp.coordinator.coordinator.CoordinatorLocal* method), 158
 is_running() (*pvp.coordinator.coordinator.CoordinatorRemote* method), 158

K

kill() (*pvp.coordinator.coordinator.CoordinatorBase* method), 157
 kill() (*pvp.coordinator.coordinator.CoordinatorLocal* method), 158
 kill() (*pvp.coordinator.coordinator.CoordinatorRemote* method), 159

L

LEAK (*pvp.alarm.AlarmType* attribute), 155
 load_file() (*pvp.common.loggers.DataLogger* method), 129
 load_prefs() (in module *pvp.common.prefs*), 133
 load_rule() (*pvp.alarm.alarm_manager.Alarm_Manager* method), 140
 load_rules() (*pvp.alarm.alarm_manager.Alarm_Manager* method), 140
 LOADED (in module *pvp.common.prefs*), 131
 log2csv() (*pvp.common.loggers.DataLogger* method), 130
 log2mat() (*pvp.common.loggers.DataLogger* method), 130
 logged_alarms (*pvp.alarm.alarm_manager.Alarm_Manager* attribute), 138, 140
 logger (*pvp.alarm.alarm_manager.Alarm_Manager* attribute), 140
 LOW (*pvp.alarm.AlarmSeverity* attribute), 155
 LOW_O2 (*pvp.alarm.AlarmType* attribute), 155
 LOW_PEEP (*pvp.alarm.AlarmType* attribute), 155
 LOW_PRESSURE (*pvp.alarm.AlarmType* attribute), 154
 LOW_VTE (*pvp.alarm.AlarmType* attribute), 154

M

make_dirs() (in module *pvp.common.prefs*), 133
 manager (*pvp.alarm.condition.Condition* property), 148
 MEDIUM (*pvp.alarm.AlarmSeverity* attribute), 155
 MISSED_HEARTBEAT (*pvp.alarm.AlarmType* attribute), 155
 mode (*pvp.alarm.condition.AlarmSeverityCondition* property), 152
 mode (*pvp.alarm.condition.ValueCondition* property), 149
 module
 pvp.alarm, 154
 pvp.alarm.alarm, 142
 pvp.alarm.alarm_manager, 137
 pvp.alarm.condition, 147
 pvp.alarm.rule, 146
 pvp.common.fashion, 135
 pvp.common.loggers, 127
 pvp.common.message, 124
 pvp.common.prefs, 130
 pvp.common.unit_conversion, 133
 pvp.common.utils, 134
 pvp.common.values, 118
 pvp.controller.control_module, 109
 pvp.coordinator.coordinator, 156
 pvp.coordinator.process_manager, 160
 pvp.coordinator.rpc, 159
 pvp.gui.styles, 106
 pvp.io, 135
 pvp.io.devices, 137
 pvp.io.hal, 135
 MONITOR_UPDATE_INTERVAL (in module *pvp.gui.styles*), 106

N

n_cycles (*pvp.alarm.condition.CycleAlarmSeverityCondition* property), 153
 n_cycles (*pvp.alarm.condition.CycleValueCondition* property), 150
 name (*pvp.common.values.Value* property), 121

O

OBSTRUCTION (*pvp.alarm.AlarmType* attribute), 155
 OFF (*pvp.alarm.AlarmSeverity* attribute), 155
 operator (*pvp.alarm.condition.CycleValueCondition* attribute), 150
 operator (*pvp.alarm.condition.ValueCondition* attribute), 149
 OUpdate() (*pvp.controller.control_module.Balloon_Simulator* method), 116
 oxygen (*pvp.io.hal.Hal* property), 137

P

PEEP (*pvp.common.values.ValueName* attribute), 119

PEEP_TIME (*pvplib.common.values.ValueName* attribute), 119
 pending_clears (*pvplib.alarm.alarm_manager.Alarm_Manager* module, 135 attribute), 138, 140
 pigpio_command() (in module *pvplib.common.fashion*), 135
 PIP (*pvplib.common.values.ValueName* attribute), 119
 PIP_TIME (*pvplib.common.values.ValueName* attribute), 119
 plot (*pvplib.common.values.Value* property), 122
 plot_limits (*pvplib.common.values.Value* property), 122
 PLOTS (in module *pvplib.common.values*), 123
 PRESSURE (*pvplib.common.values.ValueName* attribute), 119
 pressure (*pvplib.io.hal.Hal* property), 136
 ProcessManager (class in *pvplib.coordinator.process_manager*), 160
 pvplib.alarm module, 154
 pvplib.alarm.alarm module, 142
 pvplib.alarm.alarm_manager module, 137
 pvplib.alarm.condition module, 147
 pvplib.alarm.rule module, 146
 pvplib.common.fashion module, 135
 pvplib.common.loggers module, 127
 pvplib.common.message module, 124
 pvplib.common.prefs module, 130
 pvplib.common.unit_conversion module, 133
 pvplib.common.utils module, 134
 pvplib.common.values module, 118
 pvplib.controller.control_module module, 109
 pvplib.coordinator.coordinator module, 156
 pvplib.coordinator.process_manager module, 160
 pvplib.coordinator.rpc module, 159
 pvplib.gui.styles module, 106
 pvplib.io module, 135
 pvplib.io.devices

R

register_alarm() (*pvplib.alarm.alarm_manager.Alarm_Manager* method), 141
 register_dependency() (*pvplib.alarm.alarm_manager.Alarm_Manager* method), 141
 reset() (*pvplib.alarm.alarm_manager.Alarm_Manager* method), 142
 reset() (*pvplib.alarm.condition.AlarmSeverityCondition* method), 152
 reset() (*pvplib.alarm.condition.Condition* method), 148
 reset() (*pvplib.alarm.condition.CycleAlarmSeverityCondition* method), 153
 reset() (*pvplib.alarm.condition.CycleValueCondition* method), 150
 reset() (*pvplib.alarm.condition.TimeValueCondition* method), 151
 reset() (*pvplib.alarm.condition.ValueCondition* method), 149
 reset() (*pvplib.alarm.rule.Alarm_Rule* method), 146
 restart_process() (*pvplib.coordinator.process_manager.ProcessManager* method), 160
 rotation_newfile() (*pvplib.common.loggers.DataLogger* method), 129
 rounded_string() (in module *pvplib.common.unit_conversion*), 134
 rpc_server_main() (in module *pvplib.coordinator.rpc*), 159
 rules (*pvplib.alarm.alarm_manager.Alarm_Manager* attribute), 138, 140

S

safe_range (*pvplib.common.values.Value* property), 121
 save_prefs() (in module *pvplib.common.prefs*), 133
 SENSOR (in module *pvplib.common.values*), 123
 sensor (*pvplib.common.values.Value* property), 122
 SENSORS_STUCK (*pvplib.alarm.AlarmType* attribute), 155
 SensorValues (class in *pvplib.common.message*), 124
 set_breath_detection() (in module *pvplib.coordinator.rpc*), 159
 set_breath_detection() (*pvplib.controller.control_module.ControlModuleBase* method), 112
 set_breath_detection() (*pvplib.coordinator.coordinator.CoordinatorBase* method), 157
 set_breath_detection() (*pvplib.coordinator.coordinator.CoordinatorLocal* method), 158

set_breath_detection()
 (*pvp.coordinator.coordinator.CoordinatorRemote*
 method), 158

set_control() (*in module pvp.coordinator.rpc*), 159

set_control() (*pvp.controller.control_module.ControlModuleBase*
 method), 112

set_control() (*pvp.coordinator.coordinator.CoordinatorBase*
 method), 157

set_control() (*pvp.coordinator.coordinator.CoordinatorLocal*
 method), 158

set_control() (*pvp.coordinator.coordinator.CoordinatorRemote*
 method), 158

set_dark_palette() (*in module pvp.gui.styles*), 107

set_flow_in() (*pvp.controller.control_module.Balloon_Simulator*
 method), 116

set_flow_out() (*pvp.controller.control_module.Balloon_Simulator*
 method), 116

set_pref() (*in module pvp.common.prefs*), 132

set_valves_standby()
 (*pvp.controller.control_module.ControlModuleDevice*
 method), 116

setpoint_ex (*pvp.io.hal.Hal* property), 137

setpoint_in (*pvp.io.hal.Hal* property), 137

severity (*pvp.alarm.alarm.Alarm* property), 144

severity (*pvp.alarm.rule.Alarm_Rule* property), 146

snoozed_alarms (*pvp.alarm.alarm_manager.Alarm_Manager*
 attribute), 138, 140

start() (*pvp.controller.control_module.ControlModuleBase*
 method), 114

start() (*pvp.coordinator.coordinator.CoordinatorBase*
 method), 157

start() (*pvp.coordinator.coordinator.CoordinatorLocal*
 method), 158

start() (*pvp.coordinator.coordinator.CoordinatorRemote*
 method), 158

start_process() (*pvp.coordinator.process_manager.ProcessManager*
 method), 160

stop() (*pvp.controller.control_module.ControlModuleBase*
 method), 114

stop() (*pvp.coordinator.coordinator.CoordinatorBase*
 method), 157

stop() (*pvp.coordinator.coordinator.CoordinatorLocal*
 method), 158

stop() (*pvp.coordinator.coordinator.CoordinatorRemote*
 method), 159

store_control_command()
 (*pvp.common.loggers.DataLogger* *method*),
 129

store_derived_data()
 (*pvp.common.loggers.DataLogger* *method*),
 129

store_program_data()
 (*pvp.common.loggers.DataLogger* *method*),
 129

store_waveform_data()
 (*pvp.common.loggers.DataLogger* *method*),
 129

T

TECHNICAL (*pvp.alarm.AlarmSeverity* attribute), 155

time_limit() (*in module pvp.common.utils*), 134

timeout() (*in module pvp.common.utils*), 134

TimeoutException, 134

TimeValueCondition (class *in pvp.alarm.condition*),
 150

to_dict() (*pvp.common.message.SensorValues*
 method), 125

to_dict() (*pvp.common.values.Value* *method*), 122

try_stop_process() (*pvp.coordinator.process_manager.ProcessManager*
 method), 160

U

update() (*pvp.alarm.alarm_manager.Alarm_Manager*
 method), 140

update() (*pvp.controller.control_module.Balloon_Simulator*
 method), 116

update_dependencies()
 (*pvp.alarm.alarm_manager.Alarm_Manager*
 method), 142

update_logger_sizes() (*in module*
 pvp.common.loggers), 128

V

Value (class *in pvp.common.values*), 119

value_names (*pvp.alarm.rule.Alarm_Rule* property),
 146

ValueCondition (class *in pvp.alarm.condition*), 148

ValueName (class *in pvp.common.values*), 118

VALUES (*in module pvp.common.values*), 122

VIE (*pvp.common.values.ValueName* attribute), 119