An Oxygen Engine Digital Control System (FINAL, v1.0)

(Prepared for NASA and Jacobs Engineering
JETS-SOW-PRS23-1211)

POC: Robert L. Read <read.robert@pubinv.org>
Public Invention
Austin, TX

May 11, 2023
– Robert L. Read, Geoff Mulligan, Forrest “Lee” Erickson and Lawrence Kincheloe, all of Public Invention, a US501c3 public charity

Introduction

Formal Identifications

This report is prepared under contract by Public Invention for NASA and Jacobs Engineering (**JETS-SOW-PRS23-1211**). Robert L. Read, President of Public Invention, is the point of contact for this work (email: <read.robert@pubinv.org>). Public Invention has experience in human respiration issues including PSA O2 production and gas measurement. This work is an extension of work begin under **JETS SOW: PRS22-6918** and submitted to NASA as a report entitled *Technical approaches for the next Generation of Ceramic Oxygen Generator (COG) power, data, and control systems*, also available from Public Invention upon request. Parts of this report are copied directly from that report. Public Invention worked in cooperation with American Oxygen (AmOx) and NASA in preparing this work. Much of this work is released as open source software and hardware design files as agreed with NASA, and publicly accessible repositories containing this design documents and code will be referenced as needed below, but include at least:

1. https://github.com/PubInv/NASA-COG – the main embedded software and enclosure design
2. https://github.com/PubInv/mcogserver – a data logging cloud based server

Motivation

NASA has supported the development of a Medical Ceramic Oxygen Generation (M-COG) technology. This has applications in space for producing fuel oxidants, producing therapeutic

oxygen, and scrubbing oxygen from gasses to make them unable to support combustion. Moreover, this technology could also revolutionize global health by providing a robust, rugged, almost unbreakable way to turn electric power into therapeutic oxygen in remote places. The lack of therapeutic oxygen, particularly for childhood pneumonia, contributes to seven hundred thousands[UNICEF] of deaths per year.

The COG technology was proven[NASA Live Demo] on December 8th, 2022 by NASA and AmOx to be a robust, solid state means of producing many liters per minute of extremely pure O2 from ordinary air in a laboratory setting. However, the “balance of plant”, the machinery surrounding the ceramic electrodes needed to make a practical, deployable system is still being developed. This report primarily makes recommendations for supporting electrical hardware, blowers, sensors, and software.

Strategies and Principles

Although the core technology is proven, the ability to make a field-deployable system is still being developed. Given this level of maturity, which is beyond laboratory research but not yet fully established for manufacturability, industry conventional wisdom suggests that one should design for:

1. **Modularity and Flexibility** - Because ideal deployment approaches are still being learned, one should design for modularity and flexibility to make rapid design alterations cheap and fast.
2. **Robustness** - Although per-unit and per-liter costs are becoming a design criterion, we should still design for robustness rather than minimizing costs until experience shows where costs can be minimized and volume allows the costs of the M-COG core technology to be lowered.
3. **Economy through COTS parts and standard practices** - Although the M-COG technology is a marvel with a high “gee-whiz” factor, the electronics to drive them are a little tricky but require no research and can benefit from using standard commercial off-the-shelf parts (COTS) and industry standard practices which allow widely available contractor skill sets to be applied to it.
4. **Agility and Design Flexibility Moving Toward Economy** - By designing for agile change, we can move quickly to the least expensive options, even if we do not design for the minimize cost design under fixed assumptions.

The Basic Unit of Modularity: The Oxygen Engine

Unquestionably, to obtain economies of scale we need to define a standard, replicable module that produces oxygen but allows linear scaling by simply adding modules. We call this module an “Oxygen Engine”. Due to interaction between the core technology embedded in “cell stacks”, the electronics, and the need to minimize the cost of electrical power per liter of O2 produced, the optimal configuration of an Oxygen Engine is still being discovered.

An Oxygen Engine consists of some number of cell stacks made by American Oxygen and the “balance of plant” electronic equipment needed to carefully control them. We call this control equipment the Oxygen Engine Digital Control System (OEDCS). In May 2023, a preferred embodiment of an Oxygen Engine is an AmOx “Crossflow Cylinder (CFC)” containing 1-4 stacks.

A basic design question is: how many stacks can be controlled by a single OEDCS? At the time of this writing, our example hardware should be able to control one, but has been designed to control up to six. Our goal has been to allow flexibility in terms of how many stacks, how many heaters, and indeed how many separately controlled “airways” are to be considered the Oxygen Engine (or basic unit of modularity).

Clearly, what we want is an Oxygen Engine which is cheap, robust, and power efficient. However, we also want an Oxygen Engine that can be packed in large numbers into a small space. Moreover, we would like to do this in a way that wastes as little energy as excess heat as possible. For example, because the M-COG technology works at very high temperatures of approximately 700C to 800C, it produces high-quality exhaust heat which can be used either to heat incoming air or to produce hot water, which is extremely valuable in clinical settings in Low and Middle Income Countries (LMICs). In terms of design of the individual Oxygen Engine, we always have our eye on packing dozens of them into a small space and managing heat efficiently. For example, John Graf has suggested building a hospital-scale system in a shipping container or a half-container.

Fundamentally, an individual Oxygen Engine is also the unit of:

- **Control** (it can be turned on or off independently),
- **Measurement** (its performance can be instrumented and logged),
- **Addressability** (it has an identifying number which allows dozens of hundreds of them to be organized into a single system),
- **Serviceability** (it can be individually removed and serviced.)

Geoff Mulligan has pointed out that the most elegant way to do address and control hundreds or even thousands of Oxygen Engines is to create an Ethernet-based Local Area Network of OEDCSs where each has its own unique local IP (Internet Protocol) address and can send and receive data via standard networking technologies. Any single-boad computer (SBC) should be able to do this with hardware extensions, but in particular, the Arduino DUE can do this with the addition of an inexpensive Ethernet “shield”. (Hardware extensions in the Arduino universe are called “shields”; on Raspberry Pis they are called “hats”.)

The Proposed OEDCS Design

In order to de-risk and prove out these ideas, we have constructed an initial OEDCS system based on:

1. A documented interface panel,
2. An Arduino DUE,
3. An enclosure of hardware components,
4. Hardware components such as power supplies organized into 5 electrical subsystems:
    a. The heater (110VAC) subsystem,
    b. The stack control (variable DC voltage up to 12V) subsystem,
    c. The fan control subsystem,
    d. The thermocouple measurement subsystem, and
    e. The flow sensor subsystem.
5. A control software written in C++ using the Arduino framework and deployed with the PlatformIO cross-compilation system.

A depiction of the proposed (and partially actualized OEDCS) design is shown below:

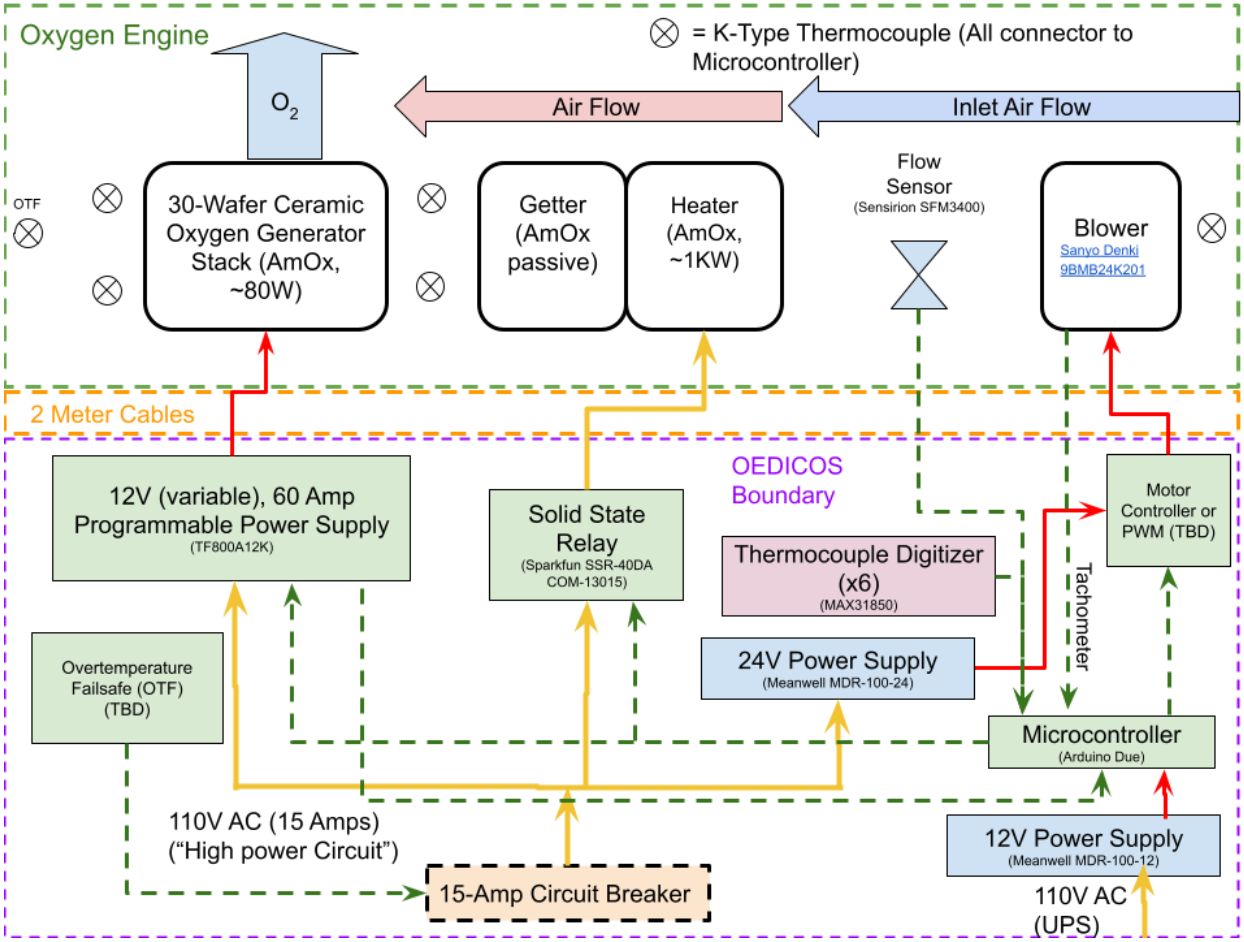

Figure 1: Process and Instrumentation Diagram

This diagram shows the system boundaries. It may be considered an informal Process and Instrumentation Diagram.

The Oxygen Engine itself is separated by a set of cables of no more than 2 meters to the OEDCS. American Oxygen currently calls the Oxygen Engine the "Cross-flow Cylinder" or CFC.

The point of making a clear distinction and documented interface between the Oxygen Engine and OEDCS is to be able to change one without having to change the other.

In this diagram, thermocouples are represented by a circle with an “X” inside. Each of these is a K-type thermocouple probe, which has long wires going back to a jack panel on the OEDCS. The yellow lines represent 110VAC power. The red lines represent DC power. The green lines are signal lines, either to or from the microcontroller. The thermocouple marked “OTF” is the Overtemperature Failsafe thermocouple, shown at the end of the air path (the hottest point, presumably.)

The OEDCS Interface Supports Modularity

The demarcation between the Oxygen Engine and OEDCS is the obvious basis of modularity which allows the oxygen producing system to change and evolve independently of the digital control system and vice versa. It is therefore of the greatest importance for supporting engineering evolution.

There are, of course, some changes on one side that will affect the other—such as a change to an electrical specification. For example, a stack might be developed that can effectively be operated at a higher voltage or amperage, which would recommend a strengthening of the power components in the OEDCS. But nonetheless, our goal is to allow improvements and redesign of the Oxygen Engine and Digital Control System independently of each other.

This interface is embodied by a physical panel right now, but it is important to understand that the panel is not the interface—the interface is the specification of the connectors and the electrical constraints they must obey.

The fundamental goal of this interface specification is to allow complete “plug and play” compatibility between the OEDCS and an Oxygen Engine. For example, if an OEDCS has been driving an Oxygen Engine for some weeks but a better OEDCS becomes available, it should be possible to simple enter the “cool down” mode, wait for the system to enter the “off” state, unplug the cables of the 5 electrical subsystems, and plug them in again to the new OEDCS and start warming up the oxygen engine again.

This specification is currently evolving, particularly around the fan controller, but as of this writing it is 2 meters cords comprising:

1. Standard 15 amp extension cords for the 110VAC heater connection
2. Standard Anderson 60 amp low-voltage connectors for the stacks themselves, with 2 meter cables and ⅜” ring lug terminals
3. K-type miniplug thermocouples
4. A DB25 connector for the fan controller (plus possibly an additional power cable) (see Appendix A for details.)
5. A DB9 connector for the flow sensor (see Appendix B for details).

These 5 subsystems are represented by 5 “ports” in an interface panel:

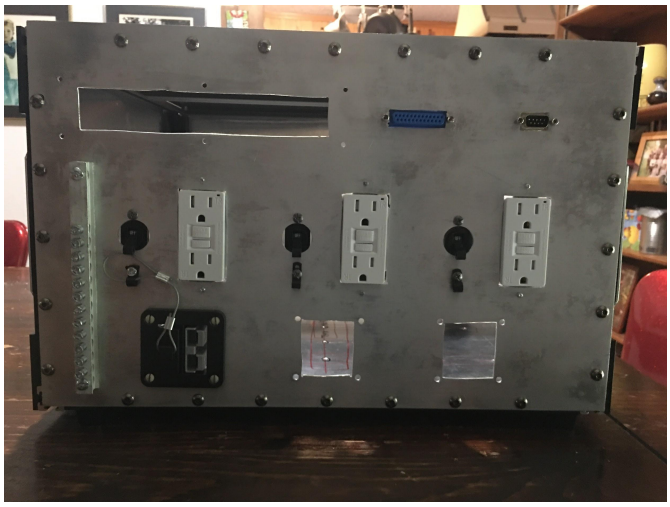

Figure 2: Empty Interface Panel

The same panel has been produced as a CAD drawing in FreeCAD, which would allow it to be produced precisely in large quantities by using a service such as Send-Cut-Send, although the panel above was cut by hand by Lee Erickson.

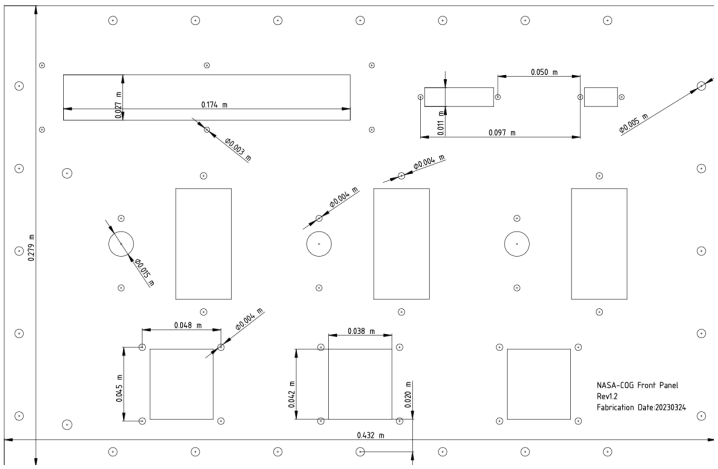

Figure 3: Example of CAD drawing

The current enclosure has ports and space to support up to 6 stacks and 6 heaters, but we have only purchased and built out support for one heater and one stack (one SSR for 110VAC and one digital DC power supply supplying up to 12V and 60 amps.)

The enclosure, of which this panel is the front, is built out of standard 80/20 (20x20) aluminum extrusions and flat plates.

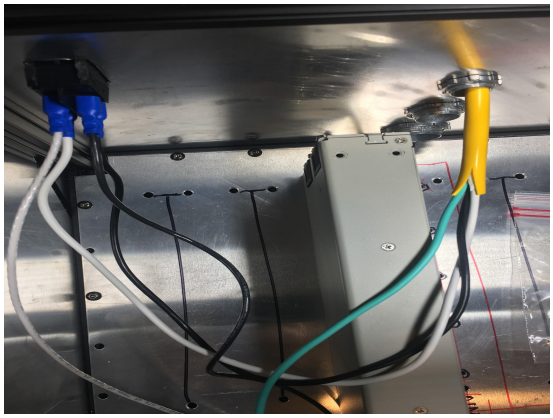

Figure 4: The Insider Rear of the case, showing the yellow 15 amp 110VAC entry cable with a gland and the back of the digital power supply.

Here are a number of views of the enclosure as it is being progressively wired:

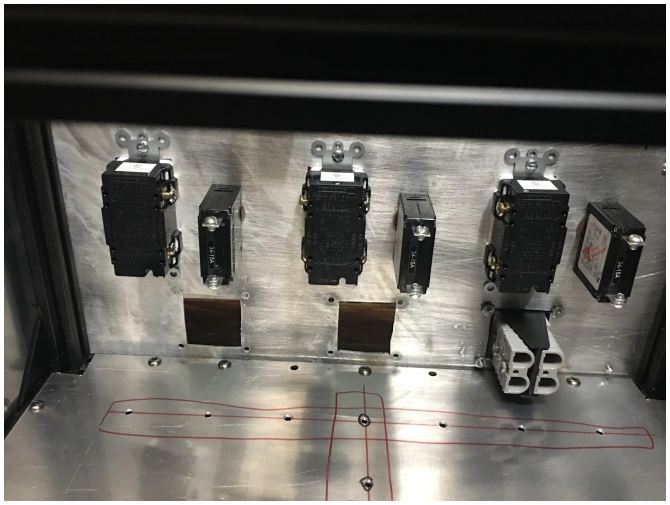

Figure 5: The Inside Front of the unpopulated case, but with standard 110VAC (unwired) receptacles for the heater power.

Each heater power is a dual receptacle. Each has beside it a GFCI with a physical circuit breaker switch on the front.

Figure 6: The Rear of the case, with the 3 entry glands (without wires), and the (lighted, but unlit) power switch.

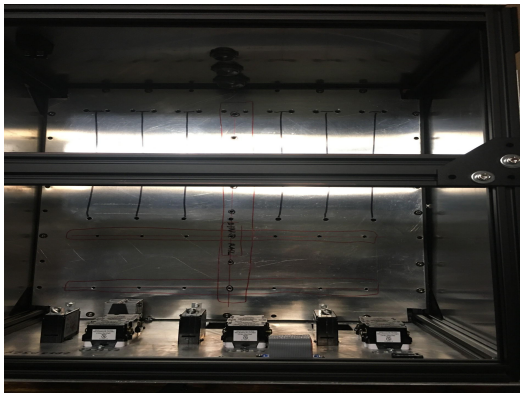

Figure 7: A top view of the unpopulated OEDCS.

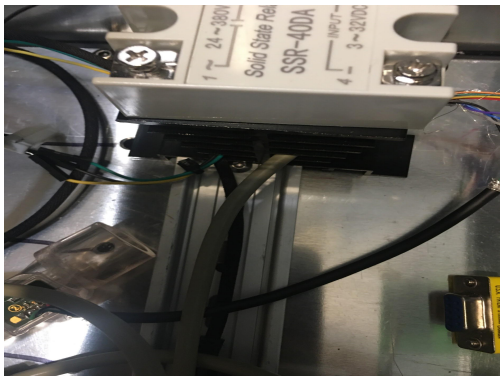

Figure 8: A view of the solid-state relay (SSR) used to digitally control the power to the heater to control temperature.

Note that by experimentation we determined that it requires a heat sink, which is the black finned object beneath it. The heat sink itself is mounted on a DIN rail.

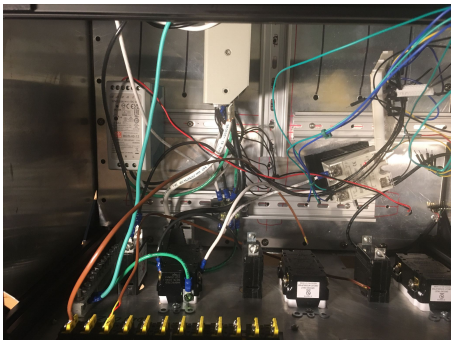

Figure 9: A top view of the partially populated OEDCS, showing the (partial) wiring, the digital power supply, the SSR, the Meanwell 12V power supply, the SL Power Systems TF800 programmable power supply, and the yellow thermocouple jack panel.

The enclosure is relatively large to accommodate the power supplies for up to six stacks. A portion of the space has standard DIN rails bolted in, which allow the mounting of a large number of small components.

Following best practices, there is a grounding bus bar, and power components are grounded to this. The input to the enclosure is a 20-amp 110VAC extension cord. The 110VAC power outputs (for the heaters) are wired with switched 15 amp circuit breakers with GFCI features.

The OEDCS In “Test Configuration”

The annotated photograph below may help to explain how the OEDCS will work in a completely “plug compatible” way with the current Cross-flow Cylinder (CFC) Oxygen Engine design.

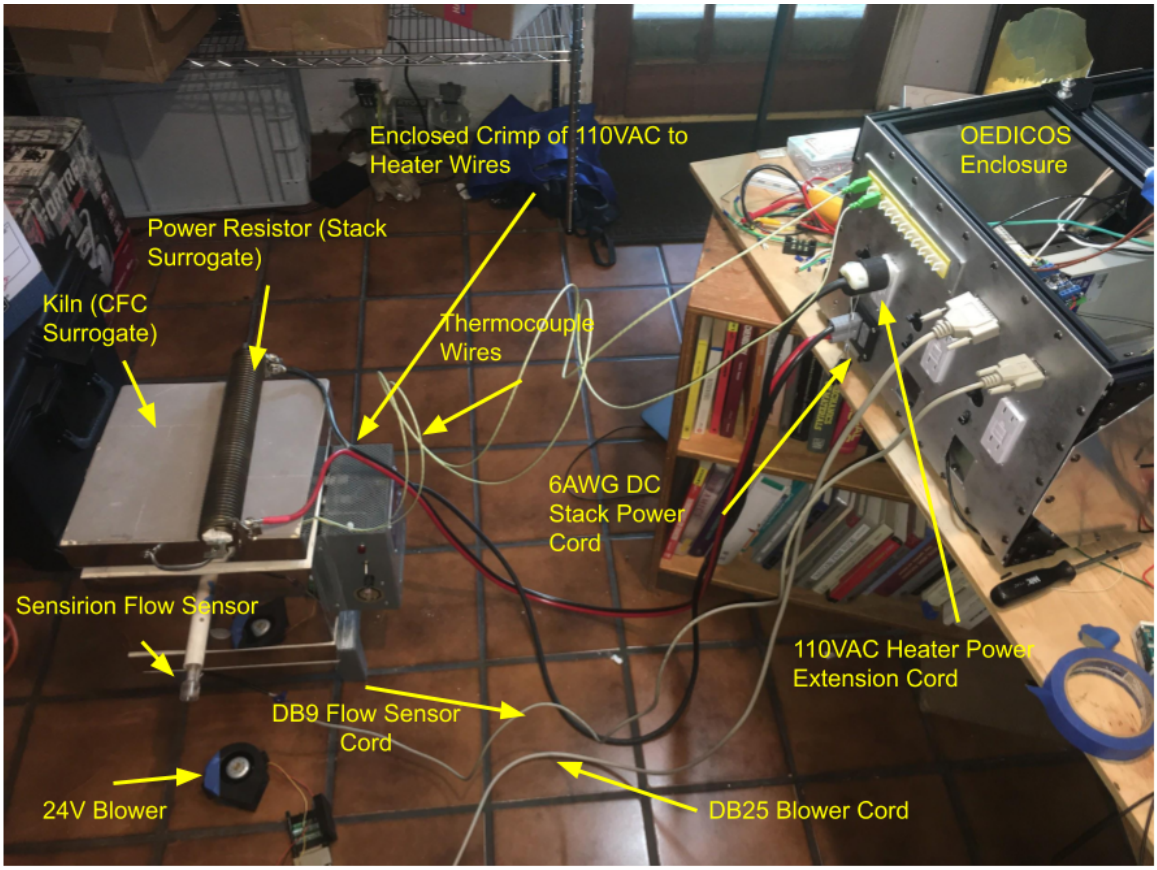

Figure 10: A labeled view of the OEDCS with a simulated Oxygen Engine.

Although a bit busy, this shows the actual equipment that we have constructed. The blower is in place (on the flow) but not mounted into the airway yet. The thermocouples are poked into the kiln through the cracked lid (there are better ports inside the perforated enclosure for this.)

At the OEDCS enclosure front panel, there are the expected 5 electrical cable connections, each of which is simply plugged in. Hopefully when ready, the CFC can be similarly plugged in. The connections are:

1. A 110VAC cord coming from the side enclosure of the kiln and plugged into the 15 amp socket.
2. An “Anderson Connector” 60-amp 6AWG red-and-black DC line is connected from a power resistor (on top of the kiln) to the connector on the Front panel.
3. Two K-type thermocouples go from the kiln to the jack panel on the OEDCS.
4. A 24V fan (not actually connected) is next to the DB25 cable which connects it to the OEDCS
5. The Flow Sensor is shown stuck in the PVC tube modeling a metallic input tube (this has not yet been constructed), showing the DB9 cable connection back to the OEDCS.

Note that the kiln came with its electrical components mounted on the side in the perforated enclosure bolted to the side. We have disconnected that thermostat, and are now using our own embedded system code to control two heaters. These cable connections are shown in greater detail below.

The Five Electrical Subsystems

The five electrical subsystems are designed to be independently testable and troubleshootable. All of them, of course, are ultimately powered by the 110VAC input to the box.

The Stack (DC) Subsystem

The Stack control subsystem uses the most expensive component in the OEDCS, a digitally controllable power supply providing up to 12V and 60 Amps. (other models can provide higher voltages). The digital power supply is an SL Power Systems TF800:
https://www.mouser.com/datasheet/2/373/TF800_Datasheet-1370067.pdf
This system is controlled by the Arduino Due via a serial (UART) interface. Like all such power supplies, it allows the voltage to be specified, but also allows an amperage limit to be specified, and will instantly decrease voltage given load to meet either the voltage or amperage limit. This is ideal, since the resistance (load) of the stack is heat dependent.

Note that the oxygen generating stack is unusual for an electrical element in that its resistance decreases with increasing temperature. Regardless of voltage or amperage, this creates a possibility of thermal run away which would probably destroy a stack. It is necessary therefore, to constantly monitor the temperature of the stack via a thermocouple and potentially adjust the amperage it receives (which heats it.)

Additionally, the amperage is in theory dependent on the O2 content of the air in contact with the stack. However, in most environments this is highly predictable and unchanging.

Luckily however the stacks are relatively heavy and have a relatively high thermal mass. Therefore adjustments can be made relatively slowly to the electrical input to the stack safely. Furthermore, since the stack acts as a capacitor, it can survive modestly abrupt changes in voltage (or ripple) well. Precisely how rapidly the amperage should be adjusted is still being investigated, but current thinking is that adjusting once per second (1Hz) is enough.

This component costs approximately $350. In the current design, each stack would require its own $350 power supply. We have designed the enclosure so that 6 such power supplies can be added to the OEDCS. Each of these at maximum power may add 80 watts into the enclosure.

Each power supply has its own internal fan; however, we have not yet installed a fan for the entire case.

There are two ways this cost could be decreased. First, if we knew how many stacks to support within each OEDCS, we could probably order a multi-channel power digital power supply that would save money overall. Secondly, if investigations demonstrate that stacks can tolerate less precise control, a much cheaper solution consisting of a PWM controlled power delivery (fed into a capacitor and inductor to smooth out the current) would be much cheaper. For example, if the stacks were completely impervious to rapid voltage changes, a simple power transistor that costs less than $5 could be used. However, one would still have to perform heat management. Therefore it is probably best to use a packaged “power supply” rather than attempting to save money by creating a simple circuit. However, by using PWM control instead of a full digital supply, considerable savings could be had.

One advantage of the current power supply is that it accurately reports the current which is drawn. This allows our software to directly compute the resistance of the stack by dividing the voltage by the amperage. Since the resistance of the stack as the temperature rises is a fundamentally interesting data point, this is useful in a system that is still being used for tuning.

The Heater (AC) Subsystem

The heater system in contrast is quite simple because heater elements are assumed to be insensitive to rapid changes in voltage. We use a simple $11 solid-state relay (SSR) to switch power on and off under digital control: https://www.sparkfun.com/products/13015. This component does require the addition of a large heat sink when sourcing an expected load. We have tested it for an hour at 830 watts and with a heat sink it remains cool enough to touch. In terms of software, this element could be controlled either as an on-off thermostat or the duty cycle could be adjusted with a PID controller. Our code currently treats it as a simple on-off system.

At present only one of these is installed. However, the heat sink is DIN rail mountable, and any number could be installed—but running many heaters would quickly draw more wattage than we can bring into the enclosure (currently 20 amps at 110 volts, or 2200 watts.) If multiple airway heaters are required, they could be run at less than full power, which would decrease the rate at which the systems warms up to operating temperature, but might be completely acceptable.

If absolutely necessary, the OEDCS could switch to 220VAC power, but this would make powering it in an office environment inconvenient in the US. However, 220VAC is critical for running multiple stacks at full power and so we are considering the integration of 110/220VAC for the next design.

The external interface uses a 15 amp grounded extension cord connection. This is extremely convenient, as the heater can simply be plugged and unplugged. However, it should be

understood that this 110VAC power is under digital control, and does NOT provide steady AC power. It cannot be used for any purpose other than the heater.

To make this system as safe and convenient as possible, it is protected by a switched circuit breaker and a ground fault circuit interrupter. At present, the first circuit breaker is wired for the entire box. However, in the future these circuit breakers could control the heaters independently.

The Thermocouple Subsystem

The thermocouple system uses a 12-thermocouple panel of K-type miniplug thermocouples. Internally, these are wired to breakout boards using the Adafruit MAX31850 digital amplifiers (https://www.adafruit.com/product/1727) . These use the “Dallas One-wire” data bus with parasitic power. We have only populated 2 such amplifiers at present (one for the air going into the stack, and one for the air coming out of the stack.)

These breakout boards are currently wired using breadboards, which are not a robust solution. When we know how many thermocouples we want in an OEDCS, we should either design our own PCB to embed that number of breakout boards, or purchase a multi-channel solution. The beauty of open source digital designs is that designing our own PCB for this purpose is a reasonable solution.

It is possibly a good idea to move the amplifiers closer to the back of the plug panel, to reduce the length of wires within the enclosure. The Thermocouple wire in particular is quite stiff and difficult to work with. However, once the thermocouples are digitally amplified, the connection is reduced to a single wire.

The Fan Controller Subsystem

Sufficient airflow is always required, and this airflow must be heated, to prevent thermal shock to the stack. The stacks themselves further heat the process air. In a sense, heating the air is a cost that argues for limiting the amount of air passing through the system. However, with heat exchangers this may be somewhat irrelevant. Furthermore, we plan to reclaim waste heat for other applications, such as hot water for use in clinics. The upshot of all of this is that at present, having careful control of the airflow is essential, although with further research and development it may be possible to treat this less precisely.

The standard way to do this is with a “4-wire DC fan” which controls speed with pulse-width modulation signal and reports the RPMs with a tachometer. The current design uses 24V fans. At present, the 24V fans are controlled by a 3.3 amp DIN rail mounted 24V power supply.

Based on assumptions from January of redundancy needed in the fans, we planned for 4 fans, which amounts to 16 wires. This creates a classic “rats nest” of wires if performed on a breadboard. We therefore designed a very simple PCB that has a mount for a DB25 connector.

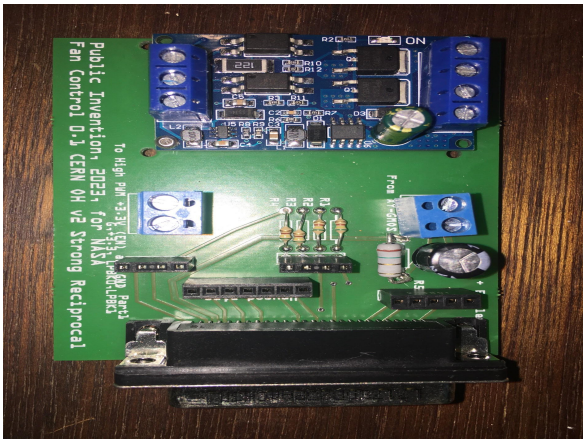

Figure 11: A Custom Fan Controller PCB.

This green board is the PCB that we designed. The blue board is a “daughterboard” which is a cheap motor controller designed to solve a problem with some of these 4-wire fans: these fans often do not turn off completely. Even with a 0% PWM duty cycle, some fans run at a minimum speed.

This board can be mounted directly onto the interface. Then a standard 2 meter DB25 cable can be screwed into this mount. The other side of this DB25 mount is connected to a screw mount breakout. Then the fans (up to 4) are directly screwed into this screw mount.

The DB25 cable individual wires can support 0.5 amps; we used four of them to support the 2 amps that we needed for the fans that we had originally designed for: https://www.delta-fan.com/Download/Spec/GFC0412DS-DF00.pdf

In order to converge to the preferred hardware of American Oxygen, we switched to this fan: The Sanyo Denki 9BMB24K201. Unfortunately, after many hours of trying, it became apparent that this fan does not work with PWM control, which is the easiest way to control it digitally. Our fan controller uses a PWM-enabled motor-controller, but it still produces a pulsed 24V voltage signal. This blower works smoothly at voltage from 7V to 24V supplied by a benchtop power supply.

Please see Appendix D for more details.

The Flow Sensor Subsystem

It is possible that the PWM control of the fans is sufficient to allow precise control of the airflow, particularly when combined with the tachometer provided by 4-wire fans.

However, such an approach will never be able to detect, simply, an obstruction in the airway that decreases flow at a given fan speed. More generally, measuring the actual flow allows us to compute the true heat flow within the system. Therefore, for so long as such calculations are useful for continuing the development of our system, it is useful to have an actual measurement of the airflow. Our current software uses a PID controller to modify the fan speed to obtain a target flow.

From previous experience, we know that the Sensirion SFM3300 flow sensor is extremely reliable up to 250 liters per minute. However, it uses a 22mm airway. At the beginning of this work 250 liters per minute was considered adequate; the current design may require more flow. If the flow is higher, we can use cheap 12V mass flow sensors used in automobiles (but they are much less accurate).

We have the same physical problem of using 2 meter cables to place the flow sensor at the beginning of the airflow next to the fan (before the air gets too hot.) The SFM3300 sensor uses I^2C, but I^2C does not in general support a 2 meter cable length. There are technical solutions to this problem which we are still investigating.

Proposed Independent Overtemperature Cutoff

John Graf and Dale Taylor have pointed out that a useful safety feature, particularly for flight operations, is to have a completely independent mechanism to cut off power completely if the temperature is over a pre-determined set-point. This requires that the “high-wattage” components be wired on a separate circuit from the low-wattage 12V power to the microcontoller (and other safety components, such as the over temperature controller.) We have created issue #79 and #80 to cover these issues.

An Overtemperature Cutoff should of course be tested independently first, and then tested in an Oxygen Engine without a stack in order to intentionally create an overtemperature condition. This is easily done with a special test script (not the normal operating code) which does not provide a check on the heater power, or which sets the setpoint for the heater higher than the overtemperature.

A Note On Intrinsically Safe Thermal Fuses

Lawrence Kincheloe found a NASA brief [Eutectic Fuse] from 1967 that is somewhat relevant in describing a method of constructing a Eutectic Wire Thermal Fuse. It is not clear that there is a COTS thermal fuse that operates at the temperature (>800C) that we require for this application, but in the pursuit of achieving intrinsically safe operation, a mechanical thermal fuse helps to achieve that goal. We are hesitant to attempt to develop a thermal fusing system which is outside the scope of this project. However, it is clear that there are several potential COTS

alloys that are used industrially for brazing, such as Braze 716, which is a nickel-modified silver-copper eutectic filler metal, and are conductive enough to be placed in-line as an electrical fuse and have a precise melting point such that they could serve this purpose. In another example, one resource suggests that pure silver melts at 961C, sterling silver at 893C, and coin silver at 879C. The suggested application would be to integrate a 6AWG silver alloy wire (or thicker) at the point of maximal thermal load on the heater and the stack to provide an over temperature cutoff. This would not be a replaceable “fuse”, but a way of preventing a catastrophic thermal runaway. Precautions to contain the assumption that an over temperature event would make a mess of molten metal into a ceramic tube with the correct material selected with the correct coefficient of thermal expansion to pinch off the molten metal, would need to be developed into the stack and oxygen generator preheater. This would require rebuilding the stack and inspection of the equipment being returned to service, however it is hoped that in defining and identifying the weak point in the system, the overall stack plates could be saved.

A Note On the Thermocouple Approach

If a thermocouple is used, we need a circuit breaker that can be thrown electrically. We could monitor the over temperature thermocouple with a completely separate microcontroller. However, we would be tempted in such an application to not use a digital solution at all, but to build an analog circuit based on the very low voltage induced in a K-type thermocouple at the cutoff temperature.

Subsystem Test Procedures

Given the complexity of this system and the fact that the system is still evolving, we recommend that each subsystem have an isolated test strategy.

Because this is an electromechanical system, we suggest that each subsystem have its own software code to be uploaded to test the subsystem, and that a series of physical tests be defined in cooperation with these dedicated pieces of code.

An example of this is the file:
https://github.com/PubInv/NASA-COG/blob/develop/experimentation/TestPWM/TestPWM.ino

… was written specifically to test the ability to control the Fan subsystem and to test control. This was specifically necessitated by the recent change to the fan specification for a fan that did not allow for speed control of the fan blade motor. In test evaluation, it was determined that a different control approach was necessary to achieve control of the Fan subsystem.

Internal Wiring

Wiring the internal of the OEDCS is done by hand, and it took about 8 hours, but that was done by someone (Rob) who had little experience with it. We here recommend some practices learned.

1. The internal wiring is done with 12AWG wired.
2. Stranded wire makes it a lot easier to wire internally.
3. It is important to use consistent color coding. We use white for “hot”, black for “neutral”, and green for ground. This is a mistake and deviates from the American standard color scheme; we may have time to correct it.
4. When wiring the 12AWG clamp connections with stranded wire, it is important to strip enough insulation to make a good grip. We recommend about ½”.
5. Ideally, all wire lengths would be carefully designed and documented in a replicable way. We have not yet done this, as we have been changing the design fairly frequently.
6. Using a DIN terminal block system such as this Dinkle Terminal Block Kit is the best way to make “buses”. This is a great advantage compared to trying to add multiple connections to a single screw terminal. Here is a picture of the DIN Rail terminal blocks:

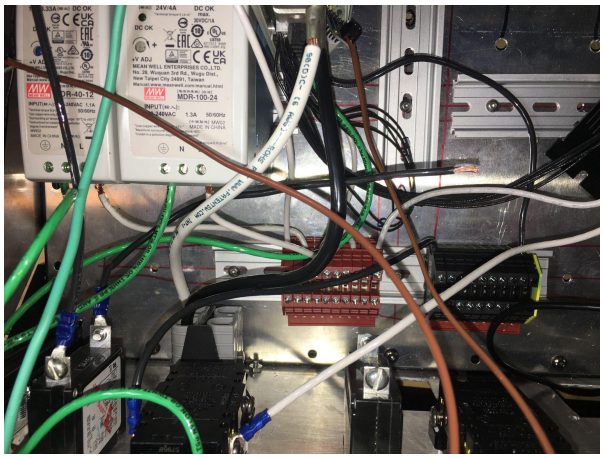

Figure 12: DIN Rail bus mounts

7. The DC power supply may provide up to 60 amps, therefore requiring 6AWG wire.
8. All Ground wires from all components are grounded to the case via a bus bar on the inside of the case along the front panel with clamp connections.

LAN-based control of many OEDCSs

As a maturing technology, we need to plan both for a large installation in a remote location without internet access, and to aim for convenience in experimental scripting.

These purposes may seem at odds with each other. However, in a large installation the most reliable way to create communication between a large number of machines is with a Local-area

Network (LAN). Although this is not technically an “internet”, the same technology (Internet Protocol) is used for both. It is therefore reasonable to use “Internet-of-Things” (IoT) approaches in both cases.

In an experimental (pre-deployment) system, the Internet may reach to a cloud service which is indeed publicly hosted by, for example, Amazon Web Services. When deployed in a remote location, the server will be implemented on the LAN only, which will be reached with the same technology, even though it may not technically be an “internet” and may not be accessible to outsiders.

IoT-based Experiment Scripting

We recommend using a scripting approach that defines an “experiment” (or script) which gives the complete instructions for running a finite or continuous operation of the Oxygen Engine.

For example a script to compare the performance of the system at 730C and 760C might instruct the oxygen engine to:

1. Make the max change in the post-stack temperature per minute at most 2.0 C.
2. Subject to that limit, use full power until the exit air reaches 400C
3. After 2 hours, until 700C is reached:
    a. increase the input air at 0.5C per minute, using
    b. No more than 60 amps in the stack
4. Then spend 1 hour at no more than 730C as a post-stack temperature
5. Then spend 1 hour at no more than 760C as a post-stack temperature
6. Then enter the standard cooldown mode (parameters defaulted in C)

Such a script is best expressed in a standard format. We recommend a (limited) use of JSON for this. Such a script in JSON looks (roughly) like:

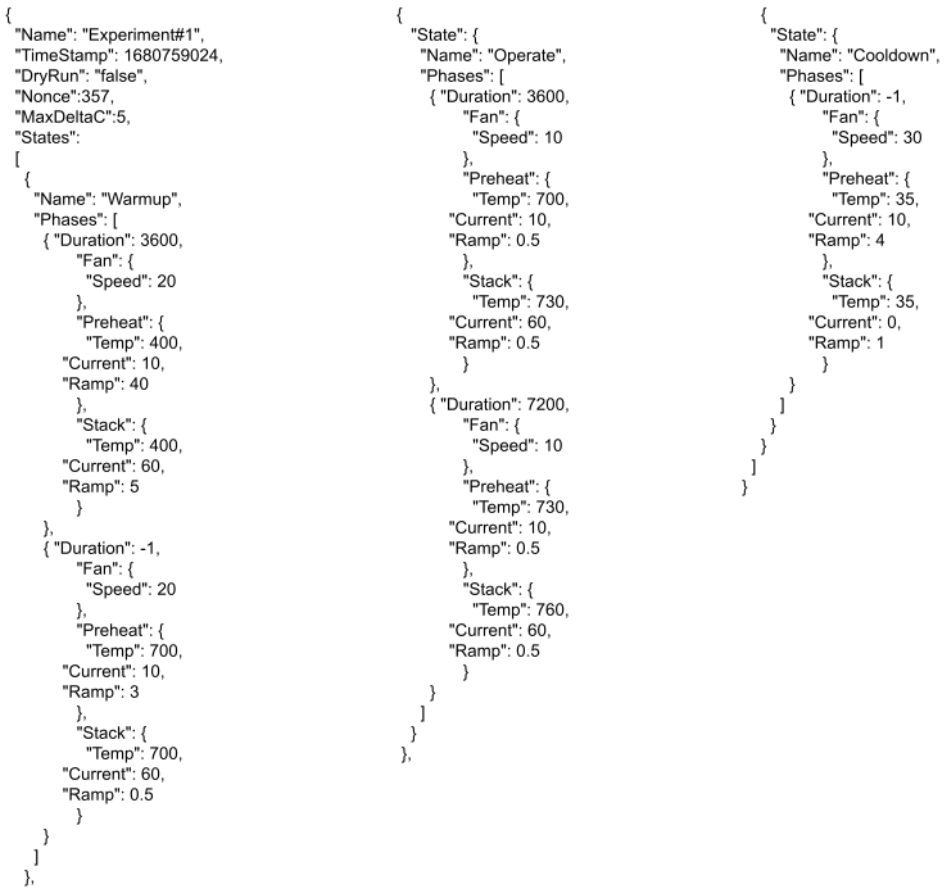

Figure 13: An Overview of a JSON control script .

A more complete explanation of the meaning of this script is given in Appendix E.

At the time of this writing, a server that allows this script to be read over a network has been implemented, but the execution of the script has not yet been integrated into the embedded system.

We expect when this is being used for active experiments, we will discover changes and extensions to the format and its executional implementation which will be maximally convenient.

A major limitation is that at present we do NOT imagine having this scripting being flexible and powerful enough to direct the oxygen engine when something goes wrong. For example, after a power shortage, or a limitation in the flow, one can imagine either an emergency cool-down or attempting to return to full execution. Although we may need to evolve into this, at present we assume that the logic for how to handle such problems will be programmed into the embedded C++ code and is NOT a part of the script.

We assert a basic principle of operation:

> The script continues running until a fault occurs or new instructions are received from the script service.

This may seem completely obvious; however we assert it because it is theoretically possible to have the OEDCS controlled completely “robotically”. We believe this is a bad idea because it is fragile if the network does not work very well.

IoT-based Data Logging

Complementing but distinct from the scripting system is the data logging system. The data logging for a functioning oxygen engine is expected to be steady, slowly changing, and in a word–boring. However, continuous logging (at a rate yet to be determined) allows us to develop confidence in the system.

Moreover, some things are going to go wrong. For example, variation in manufacture may result in one stack having a different resistance (in ohms) to another at the same temperature. Furthermore, the behavior of the system when something goes wrong, such as a fan breaking, is very important and informative. For example, if a fan breaks and as a result the system overheats due to a software error or some other hardware error, logging the behavior at this time will be extremely valuable for making the system more robust in the future.

Our basic recommendation is to log:

1. Heater exit temperature,
2. Stack exit temperature,
3. Stack voltage, amperage and resistance,
4. Airway flow, and
5. Fan speed

periodically, such as once every 5 seconds. Such a data record is sufficiently simple to be logged in either JSON or some slightly more compact format.

Our current code is using UDP and produces:

```
{
 "HeaterC": "349",
"HeaterV": "110",
"StackC": "350",
"StackV": "12.0",
"StackA": "30.77",
"StackOhms": "0.39",
"FlowMlPerS": "813.0",
"FanSpeed": "0.27"
}
```

The resulting data can be “dumped” or downloaded from the server by specifying which Oxygen Engine produced it, and the start and stop time of interest.

The resulting records can be analyzed by hand with Microsoft Excel (this skill is widely available.) However, it is also possible to build simple no-code dashboards using inexpensive for-pay web applications, such as Cumulocity. Alternatively, a small amount of JavaScript can be written, using free software such as d3 and Plotly. This is cheap, but requires some knowledge of JavaScript programming, and therefore a somewhat harder skill to contract for.

Convenience of Remote Reprogramming via the Raspberry Pi

When an oxygen engine is in day-to-day use, it may require configuration and control, if only to accept commands to shutdown for maintenance purposes. While we are still developing the technology, we will want to run experiments. We have in fact, prepared a system for executing experimental scripts.

However, we remain in a phase in which changes to the control code are likely to be required on a rapid, perhaps weekly basis. Although we can change the experimental script without reprogramming the control code, a bug or change in the control code cannot be accomplished this way.

Placing a Raspberry PI next to the OEDCS and connecting it with the serial port allows complete remote reprogramming and debugging of the OEDCS. This will remain a useful strategy during development.

Coordinating Many Oxygen Engines Through Wired Ethernet

Mea culpa: in previous informal conversations, I have suggested controlling many Oxygen Engines via the “serial” connections from a Raspberry Pi. I now think this is a bad idea, because operating in this way would limit the “fan out” to about 8 — that is the number of Oxygen Engines that could be controlled by one Pi would be 8. Geoff Mulligan has suggested a much better idea.

It makes more sense to use the MAC Address of each OEDCS as the “unique” name for the OEDCS. In this way, there is no limit to the number of OEDCS systems that can be controlled by a single general purpose computer running a Linux-like operating system (probably a Raspberry Pi, but it could be a Windows PC or an Apple Mac etc.) Note that this is true no matter how many (up to a handful) of stacks are controlled by a single OEDCS. It could be one, or four, or eight. This also allows for environmental, power and systems level sensors to be incorporated in parallel to the OEDCOS systems, thus allowing for expansive control over the entire production.

This does, however, require that each Arduino Due (or alternative single board computer) must have a “shield” which supports ethernet. However, Geoff Mulligan has tested this and it works quite reliably and the shields are widely available.

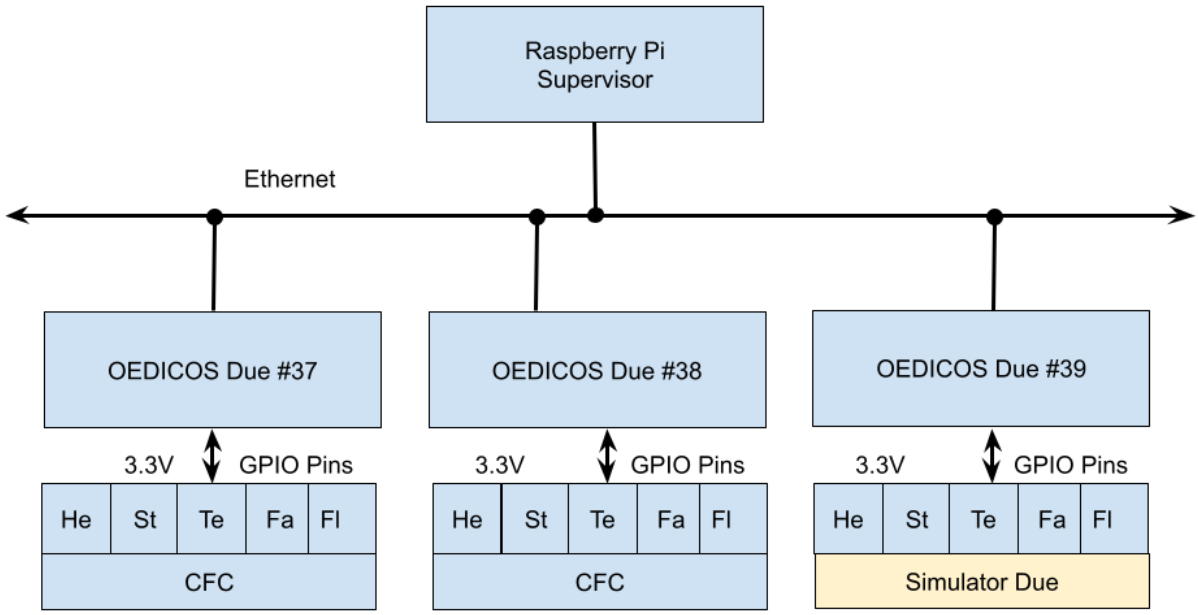

Figure 14: Connection of multiple OEDCSs via Ethenet controlled by a Raspberry Pi.

In this diagram, The 5 subsystems of the Oxygen Engine (a CFC in some cases) are abbreviated:

- He - Heater
- St - Stack
- Te - Thermocouples
- Fa - Fan/Blower
- Fl - Flow sensor

In one instance, we have shown a yellow hardware simulator (implemented with an Arduino Due.) The creation of Simulator is a long-tem strategy discussed in further detail below that will allow better control.

An Initial Pull-only Solution

We are currently imagining a "pull only" system. The OEDCS pulls data from a server based on its own client MAC address (built in to some ethernet devices, constructed in code when not available.) The server uses the fact that there has been a request from a MAC address as

sufficient for the server to list the device in a set that it is managing. If the server fails to respond for a long time, such as 20 minutes, then the OEDCS enters a fault condition. The server, right, now is running in the cloud, but will eventually be on a Raspberry Pi on the local area network so that our entire system is independent of outside internet connectivity.

There are other ways to do it. For example, if we had a need for fast emergency shutdown of all the systems in a cluster at once, we would have to do push notifications. We will design such a system when we have multiple OEDCSs coordinated by a single server.

Associating MAC addresses with Physical Systems

A device that connects to a local-area network (LAN) should have its own unique MAC address. However, the SEED Studio Ethernet shield we are currently using does not have a hardware-defined MAC address. At present, we assign a unique MAC address in software by uploading it into the firmware with PlatformIO.

When we have a large number of systems on one LAN, we will need to number physical OEDCS systems and associate them with MAC addresses. This can be done by running a simple Arduino script that prints at the MAC address, writing it down, and labeling the hardware appropriately.

Embedded System Programming

We recommend an industry-standard approach to controlling the electronics needed by an oxygen engine via an “embedded system” consisting of C++ computer program running on a single board computer. One could control the system with LabView; but this is a specialized skill and somewhat expensive. One can control the system with no digital control at all, essentially by adjusting knobs on partially analog power supplies to control fan speeds, voltages and current limits. However, this approach does not scale to simultaneously controlling many oxygen engines and requires an operator on-site to make these adjustments over the course of many hours during the experiment.

We therefore recommend using a single board computer programmed in C++. (There is a trend to allow such computers to be programmed in Python, but this remains somewhat unusual and perhaps 5-10 years away from being ready to replace C++ as a preferred approach.) The disadvantage of this approach is that modifying the OEDCS requires a skilled C++ programmer. Although this is a common skill, it is not cheap. Nonetheless on balance it seems the best approach.

This report has already suggested ways in which the program can be made flexible by "scripting"—that is, by making scripts which define simple parameters and their changes over time, we can "script" a large number of experiments without having to change the underlying C++ code.

Once C++ running on a simple microcontroller has been thoroughly tested and documented, it can function without changes and obsolescence for a long time—perhaps a decade. However, if the system is actively being used and expanded, it is likely that improvements and enhancements to the core OEDCS code will be desirable—but of course in that case there is enough economic or other incentive to warrant paying a programmer to make these changes.

Although there are many Single Board Computers that could be used for this purpose, we attempt to use the most common and "plain vanilla" off-the-shelf system that we can. The lowest common denominator is the Arduino UNO/Nano. However, although fast enough, these computers have memories too small to hold all of the code that we have already written.

In the Arduino Family, the next obvious steps up are the Due and Mega, both of which use the same physical footprint (with about 50 General Purpose I/O pins (GPIO pins)). The Mega is a 5-volt computer, and its processor is somewhat weaker than the Due, which is a 3.3V computer. At the time of this writing, sensors and digital breakout boards are about even split between the 5V world and the 3.3V world, so, sadly, no matter what family you pick you will likely have to use "level shifting" hardware to automatically shift a signal either to or from 5V from or to a 3.3V signal. We in fact do this now, to drive our flow Sensirion Flow sensor, which is an inherently 5V device, from the Due.

We chose the Due over the Mega because it has a slightly more powerful (faster, larger memory) computer and some of our sensors are 3.3V sensors (such as the thermocouples amplifiers.)

Why not using only Raspberry Pis Instead of Dues?

The Raspberry Pi has about 25 GPIO pins, and VERY powerful computers capable of running Linux. We consider using a Raspberry Pi as the core computer of the OEDCS an acceptable solution. For example, we have reviewed Tom's "Birch" Raspberry Pi code, and it is a well-written and workable approach. Our code could be recompiled for that solution. However, the Pi has some disadvantages:

- As of May 2023, there is a worldwide shortage of Raspberry Pis. They are not available without paying about triple the price before the shortage. There are, however, "knock-offs" available.
- The power of the Raspberry Pi is probably a disadvantage for running an OEDCS, because it:
    - Is not a "real time" system, but a multitasking operating system which does not inherently promise to service any need for computation in a timely manner,

    - Tends to require periodic upgrades of the many software components which they maintain, and
    - Has a relatively large “attack surface” from a cyber security point of view.

Nonetheless, we plan to rely on the Raspberry Pi architecture to control a large number of OEDCSs. That is, with a single Pi we can control a few hundred Oxygen Engines, without the need to purchase a hundred Pis.

Arduino and PlatformIO

The Arduino framework is perhaps the most broadly supported microcontroller system, although other frameworks are widely used. Confusingly, some chipsets can also be programmed with the Arduino framework! Nonetheless, our use of PlatformIO gives us further flexibility in terms of cross-compilation to other chip families, frameworks and platforms.

This would minimize, but not eliminate, the expense of porting the system. For example, the RISC-V chipset is becoming interesting, because it is a completely open-source chipset. PlatformIO already supports some RISC-V chips.

The Firmware - A Superloop (Simple Control Loop)

On any microcontroller, the programmatic tasks can be organized in several ways, which can be categorized as either:

1. A real-time operating system (RTOS), which is multi-threaded and provides sophisticated control or task priorities, or
2. A super-loop system (aka “simple control loop”) which has only a single thread or control, in which the programmers must assure each programming task completes in a timely and deterministic manner.

Given the speed of modern processors used in Arduino micro-controllers and the relatively slow processing requirements of the COG systems, the “super-loop” approach is much simpler, more reliable, and easier to verify. The burden of assuring each subtasks finishes in a finite, predictable amount of time is expected to be easy given the anticipated tasks for the COG system and the time available. The super-loop architecture is well-understood in the industry (https://en.wikipedia.org/wiki/Embedded_system#Simple_control_loop) (in this reference it is called “Simple Control Loop”.)

The Arduino Due is probably fast enough to read all of the sensors and adjust all voltages and currents within 100 ms. (Some sensors may take up to 10 ms to read, such as pressure sensors.) However, we expect this to be unnecessarily fast, and suspect that reading sensors once every 5 seconds should be adequate to respond to control this system, since essentially everything in this system changes slowly.

A block diagram of the software components showing the Simple Control Loop is:

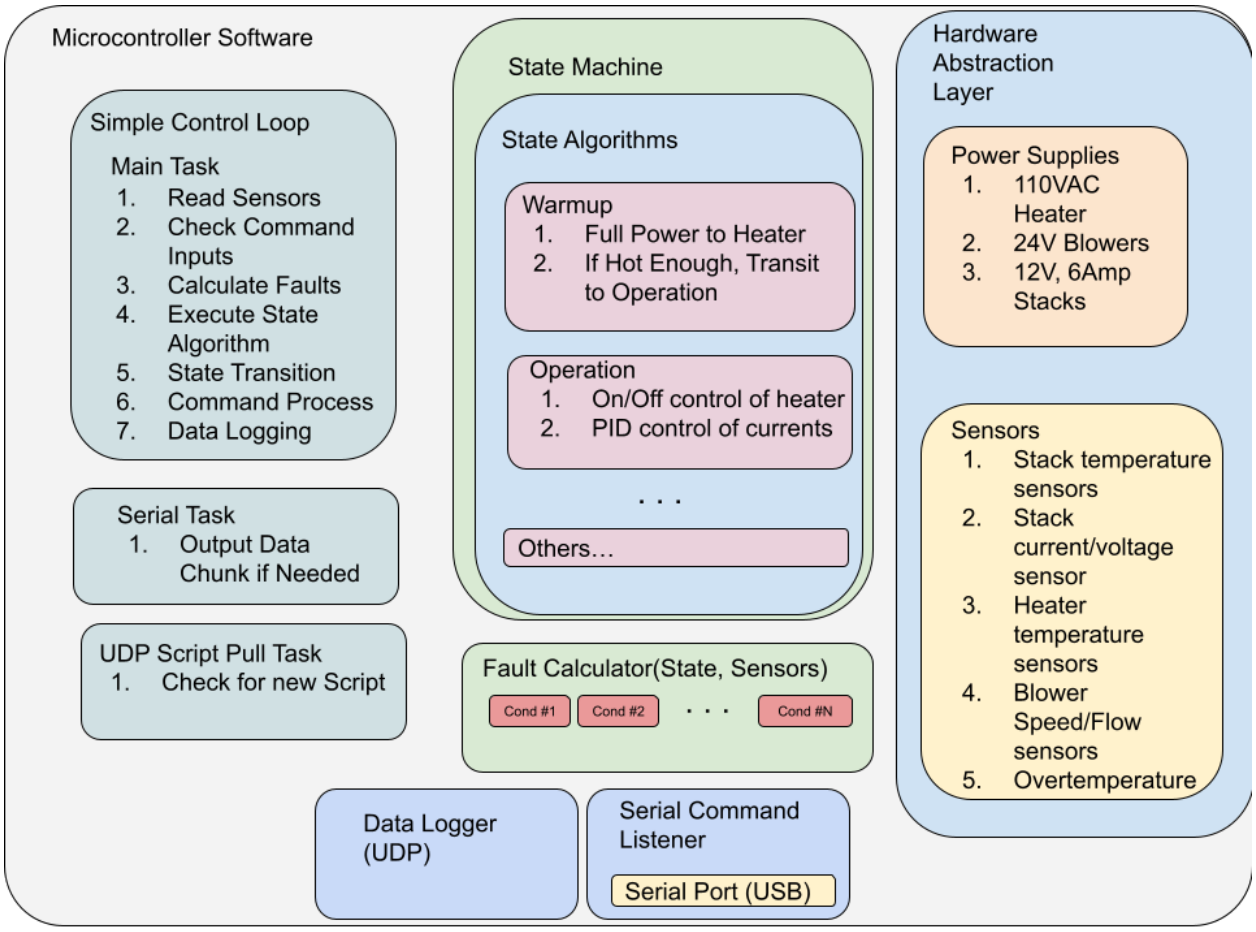

Figure 15: Block diagram of the software components

The Hardware Abstraction Layer (Hardware Drivers into an API)

The control problem is to read sensors, compute something, and drive hardware components appropriately. In the case of the COG systems, most of the sensors will be temperature, pressure, and perhaps flow sensors. Voltage and current are also set and sensed, from which resistances can be computed. Most of the hardware components that need to be driven will be blowers, heaters, and the electrochemical ceramic stacks themselves. These components all draw significant amounts of power. Digital control of the power to these systems is the fundamental problem.

The specific brands and parts needed for this are difficult to choose with absolute certainty ahead of prototype and field testing. We can reasonably expect that these components will be the fastest-evolving parts of the system. However, the fundamental control algorithm/system is likely to evolve much more slowly. Therefore, a careful encapsulation and isolation of hardware components will allow NASA's forward evolution with minimal effort. Although each change of a

sensor, motor, heating element, or stack design will require some work by both an electrical and software engineer, by encapsulating these changes behind software drivers of the hardware, the risk and effort becomes local to that specific change. Although integration testing of the whole subsystem is required after making such a change, careful encapsulation will minimize risk to total-system behavior due to a local change. This is sometimes called a “Hardware Abstraction Layer”.

Current Hardware Abstraction Layer Components

Since the beginning of this project, we have implemented a number of hardware interfaces in the hardware abstraction layer, some of which are not expected to be used. We have, however, retained these, because the design is still in flux. It costs little (in terms of intellectual distraction) to leave the unused components there, and this makes it easier to reuse.

The hardware components currently in use in our Hardware Abstraction Layer are:

- MAX31850.cpp : (digital amplifier to the K-type thermocouples)
- TF800A12K.cpp : (digital DC power supply (up to 12V, 60 amps) for the stacks
- DeltaFans.cpp : The current 4-wire fan interface (x4) (possibly changing)
- SensirionSFM3X00.cpp : The flow sensor (250 slpm limit).

Note these represent four of the five electrical subsystems. The final system, the 110VAC system, is now controllable by a single GPIO pin fed into the solid-state relay, and so needs no Hardware Abstraction to speak of.

As an example of design flexibility, we once implemented a different mass flow sensor, called the “MOSTPLUS”. This works well, but is not accurate at low flows, where the Sensirion flow sensor is far superior. Nonetheless if the flow we intend to use increases (which is highly possible), this might be a superior (and cheaper) sensor. This is implemented with the file “mostplus_flow.cpp”. (Note, it is an analog sensor, so it requires the Analog to Digital converter built in to Arduinos (or a separate chip.) However, it operates at 12V, so I had to make a voltage divider to bring it into range.)

The obsolete files, which are retained as examples are:

- DS18B20_temperature.cpp : A low-temperature thermometer
- DS3502_digital_pot.cpp : a digital potentiometer, which can control a general buck converter
- GGLabsSSR1.cpp : A solid state relay
- Mostplus_flow.cpp : A 12V mass flow hot-wire anemometer
- Abstract_temperature.cpp : an abstract temperature interface
- Mock_temperature_sensor.cpp : an abstract temperature sensor for “mocking”, no longer in use.

Our Tasks

Although we choose not to use RTOS (Real Time Operating System) or ROS (Robotic Operating System) which supports reactivity and low latency, we did choose to implement a simple task structure which allows tasks to be repeatedly executed at a parametrized frequency. In order for our system to remain “real time”, each of these tasks must take a finite time to complete; luckily, they are so simple that this is easily accomplished.

As can be seen from our the main “super loop” code in main.cpp, we have 4 tasks:

- The CogTask (the main task that directs the heaters),
- The SerialTask (which sees if any serial commands have been entered),
- The FaultTask (which attempts to determine if any faults have occurred),
- A Network Script and Data Logging Task, and
- The FanPIDTask (which controls the fan's speed to match the desired flow).

Each of these tasks is implemented as a separate file in the Tasks directory.
These are subclasses of the Task class, which provided regularity and mechanism for repeated periodic computation as configures.

The CogTask is the largest and most complex task. Fundamentally it:

1. Reads the temperature sensors,
2. Processes the new temperature data based on the current “state”, which may in fact change the state to a “fault” or other condition, and
3. It then records data and outputs a report on the serial port and to the data logger.

The SerialTask listens to see if the user has used the serial port (probably by connecting with the Arduino IDE) to enter a character requesting a change of state. In common usage and for testing, we enter a character where to have the system enter “warmup” or “cooldown” mode. The currently implemented commands are:

1. ‘W’ – begin warmup
2. ‘C’ – begin cooldown
3. ‘E’ – emergency shutdown
4. ‘A’ – acknowledge that an emergency shutdown occurred and transition to a normal “off” state
5. ‘I’ – move from “operate” to “idle” — turn off stack power to stop producing O2
6. ‘O’ – “operate” – move from “idle” to “operate”.
7. ‘P’ – set parameters:
    a. ‘T’ – change temperature
    b. ‘C’ – change current
    c. ‘F’ – change flow

The FaultTask is currently empty. We detect some faults in the CogTask task, but additional faults, based on a heat model, should be detected.

A State Machine

No doubt there will be other states in response to the detection of “panic” conditions which are beyond the scope of this work.

A state machine has a natural graphical representation (Reference Figure 4). This graphical representation is fundamental for understanding the entire system. Furthermore, it makes the system easier to validate, because the system is always in precisely one and only one state at a time. The diagram below shows our current understanding of the formal COG states. Bold lines represent “mandatory” or non-exceptional states. Dashed lines represent exceptional conditions. We assume that there are fault conditions, called Critical Fault Conditions which force an Emergency Shutdown States. There are other faults, called Service-Needed Faults, which allow continued operation, as long as the user acknowledges the fault within a specified time period. For example, if one of the two blowers fails, this may be a tolerable fault as long as the user acknowledges this within 24 hours of the fault. An untimely acknowledgement may convert the Tolerable Fault to a Critical Fault forcing an Emergency Shutdown. (Alternatively, a single fan failure in a two-fan system may more prudently require an immediate emergency shutdown.) Other faults may be detected which will be announced through the user interface, but do not rise to the level of being Service-Needed Faults requiring acknowledgement; these are legitimate faults but do not need to be represented in this state machine. Each state in this machine requires a different algorithm within the control loop.

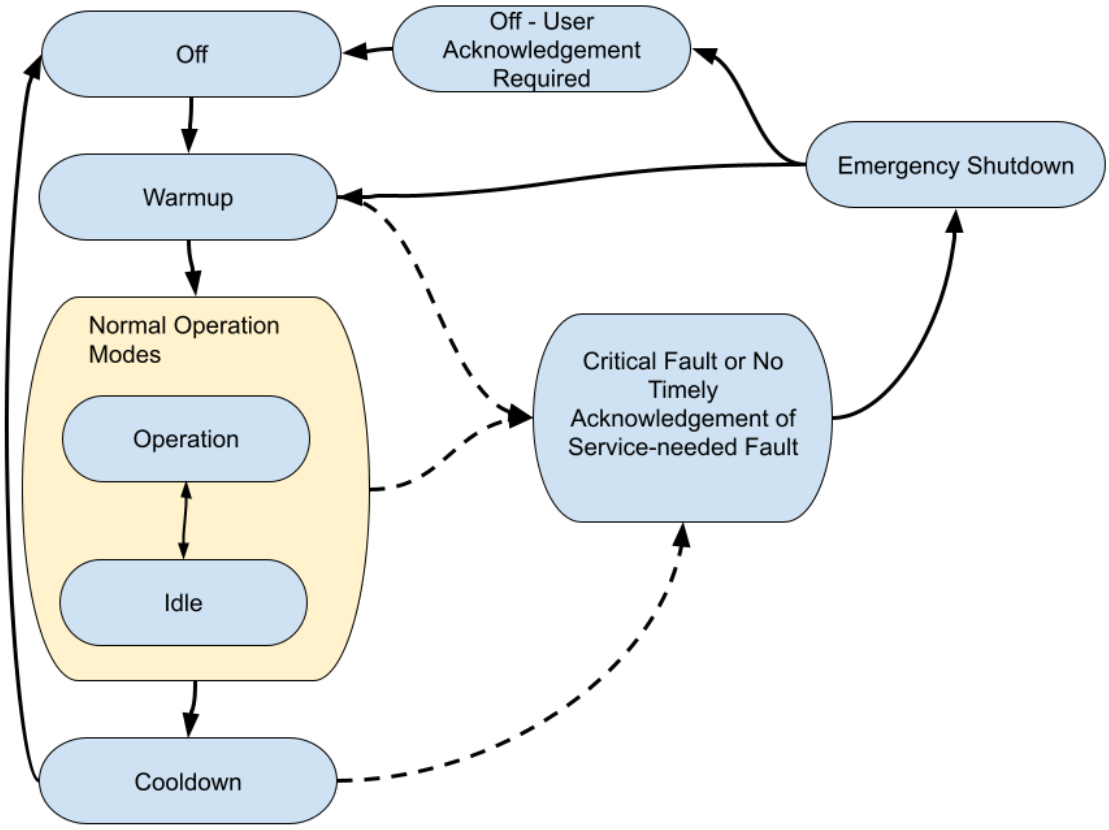

Figure 16: State machine with natural graphical representation

A formal state machine works well with a superloop architecture, which fundamentally has the structure:

Begin Loop
    Read all Sensors
    Check for Faults
    Switch on current state to state-specific behavior
    Decide if we should transition to a new state
End Loop

Each state will be represented by a C++ function that handles the control behavior for that state. For example the “operating” state might be represented algorithmically as:

Begin Operation State
    Based on temperature sensors, decide on current to be sent to each stack.
    Signal power supplies to send the appropriate current to each stack.
    Using a PID controller, modify the voltage to the heater to control the stack entrance temperature to the specified set point of : (600 C)

End Operation State

The states and state transition of normal-mode operation are likely to be stable across the project on the time scale of years. States related to fault and panic conditions may be less stable and evolve more quickly.

A Fault Detection and Handling Subsystem

We propose a software fault detection system that organizes all fault calculations. Fault handling will be state-dependent, and should be handled in the code for each separate state.

The Fault Detection module will be a separate C++ file, likely organized as a tree of decisions or simply a list of conditions to check. Such a file is likely to be long, but somewhat simple and monotonous. It will be able to be represented naturally as a table of fault conditions. Keeping a documented table of fault conditions in sync with the code which detects them will likely be an important part of the maintenance and the development of the system.

Internally faults can be usefully organized into two classes of faults: those can be detected from a single sensor, and those which require some sort of model of the system. For example, some digital electronic devices clearly report communication failures, or report a nonsensical value (such as an extremely low temperature) which make it possible to detect a fault in the sensor itself with no other information. Call these faults “Single Sensor” faults.

Other conditions can only be detected based on an understanding of the entire system. For example, if temperature sensor B is physically between temperature sensor A and C and B has a lower temperature than A or C and there is no physical mechanism for this drop in temperature, it is probably a fault. Possibly it represents an air leak, or a somewhat faulty sensor. Call these faults “Multi-Sensor faults”. Multi-sensor faults are inherently complicated, and the code to detect them would be expected to evolve and become more robust and sophisticated over time.

Every detected fault should be reported on the serial port and/or the network. Some faults are survivable; others require immediate response. This is expressed by a change in the state machine for the system (discussed previously). A critical fault causes the machine to enter the “Critical Fault” state. This inevitably triggers an “Emergency Shutdown” state which attempts to shutdown the machine in a way that is safe both for the machine components and the operators. That is, it attempts to shut the machine down slowly to prevent thermal shock, which is expected to take several hours. The operator can leave this state only by acknowledging the fault, which allows the operator to enter the normal “off” state or to re-enter the warmup state. If this happens quickly, the warmup state may immediately transition to the operating state without user interaction.

Specific Fault Tolerance

(Note: This section was taken verbatim from our first report. The physical design of the Oxygen Engine has evolved to be smaller, having perhaps only one blower and one heater.)

We recommend the following approaches to detecting and handling specific faults. In some architectures, the “secondary” component listed below may be one or more components.

1. **Loss of system level power**: The system cannot itself operate if the power to the microcontroller is lost. This is generally handled by having the microcontroller produce a “heartbeat” signal which is monitored by an external alarm system. If the “heartbeat” is not detected in a predetermined amount of time, an alarm is raised. This heartbeat can be output as a 3.3V signal using a GPIO pin on the Arduino Due. The design of the alarm system is out of scope for this project.
2. **Loss of primary blower**: Physically, we recommend using two fans and a flow sensor to deal with this. We recommend using the commonly available and inexpensive “r-wire” voltage controlled fan which produces a tachometer signal that can be read by the microcontroller. We have provided an example of this in our code and in the NASA-COG software system. This allows a direct Single Sensor detection of the failure of a fan. Additionally, a flow sensor allows further confirmation that air is flowing sufficiently. The flow sensor could detect a blocked inlet which would prevent flow even if the two fans were working properly.
3. **Loss of primary controlling thermocouple**: Most of the thermocouples are deployed in pairs. K-type thermocouples are fairly reliable, but require amplification and digitization. If communication with thermocouple fails, it is detectable as a Single Sensor fault. If communication works but the thermocouple fails to provide a good reading, it may be detectable as a “multi-sensor fault”, depending on its position.
4. **Loss of secondary controlling thermocouple**: Secondary thermocouples are analogous to primary thermocouple. The thermocouples at the trailing edge of a stack may be considered a secondary thermocouple. These thermocouples can be used to make sure a given stack is not overheating. If a thermocouple detectably fails, it needs to be reported as a non-critical fault. If all thermocouples in location fail, it becomes a critical fault which forces a controlled shut-down of the machine.
5. **Loss of pressure integrity**: Pressure inside the oxygen pathway is higher than the surrounding atmosphere under normal operation. If the gauge pressure drops to zero (that is, becomes the same as the external environment), the machine should enter an “Idling” state. If this occurs due to a mechanical failure, the machine needs to shutdown. We recommend that when changing oxygen tanks the operator force the machine into the Idling state first. A pressure drop while Idling is NOT a critical fault. When the tank is correctly reattached, the operator can move back to the Normal Operation state.
6. **Loss of additional storage capacity of oxygen**: A sensor may be able to detect the available storage capacity. In general this would be a pressure sensor on an oxygen tank detecting when the pressure reaches the maximum pressure of the tank. As available capacity decreases, this becomes a non-critical fault which can be reported and

responded to. As the capacity further decreases and approaches zero, this becomes a critical fault that forces an emergency shutdown.

7. **Loss of cell pressure integrity in an individual cell stack**: The loss of pressure integrity within an individual stack can be detected with a pressure sensor in the O2 path of the stack if the stack is separated from the main O2 flow by a check (or one-way valve). If the pressure drops and the one-way valve prevents pressure from other stacks from reaching the defective stack, a sensor can generate a fault. In a multi-stack system this might not be considered a critical failure, but it is certainly a reportable fault that will have to be responded to.
8. **Loss of power supplied to an individual cell stack**: The power to each stack must be measured for both voltage and current. If the current drops to zero, the stack may have failed or the power connection to the stack may have failed. These are both reportable stack conditions, and may represent a critical failure. In the code this is a Single Sensor fault based on the current.
9. **Loss of primary heater**: (There may be multiple heaters.) The most common failure mode for a heater is a cracked wire which prevents current flow. Since every heater has a pre-heater temperature and a post-heater temperature, a heater failure is always easily detectable as a Multi-Sensor fault. If the temperature does not rise from the pre-heater position to the post-heater position when the heater is powered, this is a fault. Depending on the number of heaters, this may not be a critical fault.
10. **Loss of secondary heater**: Over and above the loss of an individual heater, the secondary heater may be interpreted as the final heater before the stacks. If the air here cannot reach the desired temperature after sufficient time, this is a fault. If the entire heating system fails catastrophically and quickly, this will be expressed as a low pre-stack temperature. This is dangerous to the stacks, because blowing unheated air over hot stacks may damage them. An unexpected loss of temperature here is therefore a critical fault that forces a controlled shutdown.

Cooldown and Emergency Shutdown

Both normal cooldown and emergency shutdown states must algorithmically allow the system to cool in a very slow, controlled manner to avoid thermal shock to the stacks.

An algorithm for this is:

1. Place the stacks in idle mode by lowering the current (over perhaps a minute.)
2. Begin lowering the power to the heaters (over several hours) controlling the temperature decrease to the stacks at a predetermined rate.
3. Maintain normal air flow during this procedure.

Note, however, in the presence of certain faults, this algorithm needs to be more complicated. For example, if the main heaters have failed, it may be better to turn the fans off entirely in order to decrease the rate of cooling in the stacks.

Steps Needed to Fully Test the Prototype OEDCS

As of May 15, 2023, the steps needed to fully test the OEDCS as a system for controlling a single-stack cross-flow cylinder produced by American Oxygen are that Public Invention needs to:

1. Solve the problem of the flow sensor not working on a 2-meter cable.
2. Demonstrate a working fan controller, possibly change to a 24V 4-wire model, over a 2-meter cable length with proper wire amperage.
3. Integrate Geoff Mulligans IoT scripting code to allow remote scripting of the engine
4. Demonstrate data logging to a public data cloud.
5. Test the OEDCS with the simulated stack that Lawrence Kincheoloe is constructing for this purpose.
6. Physically transport the OEDCS to the CFC and connect it electrically and test it.

General Testing

When considering the future development of the OEDCS and large installations of many Oxygen Engines, testing, troubleshooting and quality assurance will be of the utmost importance. This is especially true since the system is still evolving. The software industry has learned that automated, repeatable test processes (called “automated unit testing”) allows software to be developed very rapidly. Electromechanical systems can generally not be tested so rapidly; nonetheless, a repeatable test process, even if it takes a few days to execute, allows engineering changes to be made with confidence.

We have both the blessing and the curse of being able to operate as a distributed remote team. However, there can only be one prototype in one place at one time. We therefore believe the following principles will be necessary for MCOG program to make rapid progress in the future:

- Remote control operation of the OEDCS and Oxygen Engine is the only way to fully employ the skills of the team in a safe facility.
- The development of a simulator of the Oxygen Engine will allow a much greater range of rapid testing, even though it cannot completely replace actual physical tests of “hot” equipment.
- Documentation of test procedures so that they become repeatable will be needed as soon as we have a functional prototype of the OEDCS to match to drive a CFC (or, ideally, before that.)

Future Directions (Short Term)

This is an ongoing project, with continuous improvement in the underlying ceramic technology, as well as improvements in the digital control system.

We summarize here some potential improvements.

Software Extensions

At present, there are three large tasks for the software:

1. Integration of the experimental “scripting” code into the current embedded system.
2. Testing to improve fault detection and tolerance. This could include single-component failure and failure of power components in particular.
3. Demonstration of data analysis.

Hardware Extensions

We recommend maintaining engineering flexibility in the short term. This means giving AmOx the ability to choose the number of stacks associated with Oxygen Engine. Since the internal number of digital DC power supplies depend on this, we should remain flexible until this is decided.

Enclosure Design

The current enclosure design described in this report is intended to provide maximum engineer flexibility. However, as the design solidifies, it may be possible to make decisions that optimize cost and space instead.

We believe a rack-mount design may be the best solution eventually for actual deployment. The cabling of such a system remains an unaddressed issue. In particular, we know that in a large installation, a number of oxygen engines will be placed together, and will generate a lot of heat. A rack-mounted design has the advantage of placing all OEDCSs close to each other, but it may be difficult to cable these to a large number of Oxygen Engines. Possibly necessary insulation and the need to keep the control electronics and power supplies cool will make this challenging.

Switch to 220VAC power

If we have to support a large number of heaters or even just 4-6 stacks powered by a single OEDCS, it will be necessary to switch to 220VAC power. This makes the OEDCS a bit harder to use in a desktop experimental system. However, support for both 110VAC and 220VAC would give us great flexibility.

Additional PCB design

In 2023, PCB design is fast, cheap and reliable. Systems such as the OEDCS typically evolve:

1. From breadboards connected by small jumper wires push into place, which are quite fragile, to
2. Screw-clamp connections for jumper wires, which are more robust, to

3. Printed circuit boards, which are extremely reliable.

In May 2023, our experimental OEDCS uses all three of these. However, before a system is deployed in the field, we absolutely must remove all breadboards. (We also use commercial breakout boards which are PCBs. These PCBs are highly reliable, but the connections to them are not if they are put on breadboards. However, they can be soldered into “perfboard” and made very reliable in that way.)

At the time of this writing, we have created a custom PCB to act as a fan controller, as it required a large number of wires to manage. The next step is probably to build a perfboard-based solution for the thermocouple digital amplifiers. Finally, we are still using a breakout board on a breadboard for level-shifting, and this should probably be replaced with a soldered perfboard solution.

However, it is always possible to integrate two PCBs (such as breakout boards) into one PCB, thereby making the connections between them more reliable. Typically in mass-produced consumer electronics, there is a single PCB.

This is also theoretically possible for the OEDCS. It is very common to integrate all electronics next to the microcontroller on a single PCB. However, this makes the system inflexible; you cannot change one electronic subsystem without changing the entire PCB. Furthermore, single-PCB solutions are generally used for stable solutions measured in the 100s of production units or 1000s.

Our recommendation therefore is to aim for one PCB for each of our 5 electronic subsystems when this is required.

Open Questions

WARNING: The current OEDCS has not been tested for heat management with heavy loads.

WARNING: The need to place a (very slight) positive bias voltage on each stack, even when it is quiescent, has been mentioned. Our experimental hardware currently does not do that. It is unclear to use if this can be accomplished in software with our digital power supplies or is best done with a simple analog electrical circuit.

LIMITATION: This report and our designs do not currently address O2 storage and management. For example, a practical system for therapeutic oxygen must store the oxygen in a pressure vessel of some kind, and this vessel may reach the state of being completely full and unable to receive more O2. This probably requires at least some electronic alerts to humans to take physical action and possibly some automatic action on the part of the OEDCS. This “gas management” is not addressed in this report, except insofar that our embedded system has an “idle” state in the state machine.

Future Directions (Long Term)

Large Clinical Deployment

Although outside of the scope of this contract, there have been discussions of building a large system to produce hundreds of liters per minute of O2, which would be sufficient for medium size clinical installations in Low- and Middle-Income countries. Such a large installation might also be prelude to space-based applications.

To build such a system, a large number of individual Oxygen Engines must be coordinated and physically tied together, at a minimum by piping the produced O2 into a single vessel or pressure port.

Conclusions

The fundamental conclusion of this report is that it is possible and a convenient to:

1. Create a particular module called an Oxygen Engine that can be replicated in large scale to build large systems.
2. The control of these systems should be accomplished by a replicable Oxygen Engine Digital Control System (OEDCS) using a single board computer controlling a replicated power adapter/supply that controls and uses high-voltage AC power.
3. To build a large system we should tie together a large number of Oxygen Engines controlled by a large number of Digital Control Systems connected by a local area network implemented with ethernet.
4. We propose an OEDCS design that can be made for a few thousand dollars which probably can control an Oxygen Engine.
5. We have constructed a prototype of such and OEDCS, but have not yet controlled an actual oxygen engine with it.

Bill of Materials (BOM)

In order to be sure of our proposal and to improve the quality of our recommendations, we have been building a prototype OEDCS which we hope will be fully functional and deliverable. As of May 31st, it is partially testable, but not sufficiently tested to drive the American Oxygen Cross-flow Cylinder, which has been evolving during the course of this contract. Therefore some aspects of the BOM are still in flux. The major components (already mentioned in the Process and Instrumentation Diagram above) are:

Parts List, Major Components

Quantity	Description	Manufacturer	Man Part Number	Distributor	Link	Cost Each
1	The Front Panel (Please see the Section Below for a detail of this assembly.)	Public Invention	TBD	NA	NA	$158.00
1	External Cables	Various		NA	NA	$175
1	The Case (Please see the Section Below for a detail of this assembly.)	Public Invention	TBD	NA	NA	$125.00
1-4	Stack Power Supply (Programmable Digital PS)	SL Power Systems	TF800A12K	Mouser	https://www.mouser.com/ProductDetail/Condor-SL-Power/TF800A12K?qs=MLItCLRbWsxzuGnNhvURuQ%3D%3D	377.26
1	24V Power Supply for Blower	Meanwell	MDR-100-24	Mouser	https://www.mouser.com/ProductDetail/MEAN-WELL/MDR-100-24?qs=TaOZSEYtRiUHDfSuqlnTDA%3D%3D	49.16
1	12V Power Supply for Microcontroller	Meanwell	MDR-100-12	Mouser	https://www.mouser.com/ProductDetail/MEAN-WELL/MDR-100-12?qs=TaOZSEYtRiWzDk42NLdJzQ%3D%3D	44.50
1	Ethernet Due Shield	SEEED Studio	103030021	Digikey	https://media.digikey.com/pdf/Data%20Sheets/Seeed%20Technology/	31.90

			(W5500 Ethernet Interface)		W5500_Ethernet_Shield_v1.0_Web.pdf	
1	Arduino Due	Arduino	A000062	Mouser	https://www.mouser.com/ProductDetail/Arduino/A000062?qs=D9UofrEmuWn3CLF7hFoZGA%3D%3D	48.40
1	Solid State Relay for Heater Control	Sparkfun	COM-13015	Mouser	https://www.mouser.com/ProductDetail/SparkFun/COM-13015?qs=WyAARYrbSnaM22fplj8n7A%3D%3D	9.95
1	Heat Sink	CGELE	NA	Amazon	https://www.amazon.com/CGELE-Aluminum-Radiator-Dissipation-Single/dp/B091HQL9TM/ref=asc_df_B091HQL9TM/?tag=hyprod-20&linkCode=df0&hvadid=507647825193&hvpos=&hvnetw=g&hvrand=3363593573841948259&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9028279&hvtargid=pla-1291576168501&psc=1	6.59
1	Dinkle DIN Rail Terminal Block Kit	Dinkle	TBD	Amazon	https://www.amazon.com/DIN-Rail-Block-Kit-Black/dp/B07NVVP9X7?pd_rd_w=9YYAP&content-id=amzn1.sym.724fac2e-0491-4f7a-a10d-2221f9a8bc9a&pf_rd_p=724fac2e-0491-4f7a-a10d-2221f9a8bc9a&pf_rd_r=ESHZAP7K11S0VBSDGSE0&pd_rd_wg=Zgzdq&pd_rd_r=21930177-864b-4df4-a541-89cb840817fe&pd_rd_i=B07N	28.99

					VVP9X7&psc=1&ref_=pd_bap_d_grid_rp_0_3_i	
1	DIN Rails	T&G	NA	Amazon	Link	16.95
1	Flow Sensor	Sensirion	SFM 3400-33-D	Mouser	https://www.mouser.com/ProductDetail/Sensirion/SFM3400-33-D?qs=7MVldsJ5Uaxrv4AHTYvpeA%3D%3D	45.02
6	Thermocouple Amplifier	Adafruit	1727	Adafruit	https://www.adafruit.com/product/1727	14.95
6	Thermocouple	Pico Technology	Type K, hi temp, 2m	Mouser	https://www.mouser.com/ProductDetail/Pico-Technology/Type-K-hi-temp-2m?qs=xZFQr2mActdrx7DPtwzdbg%3D%3D&mgh=1&gclid=Cj0KCQjw4NujBhC5ARIsAF4Iv6dh3TogGD_nuOoNWfHZcbB0NEPq398j-Xug_WD3qdhbti7bea43pH8aAgsYEALw_wcB	44.60
1	Arduino Due DIN rail Mount	HCDC		Amazon	https://www.amazon.com/Pinout-Breakout-Terminal-Arduino-MEGA-2560/dp/B08LKVW2ML/ref=sr_1_2?crid=35KW4GODOC1IW&keywords=Arduino+Due+carrier+DIN&qid=1685592844&s=industrial&sprefix=arduino+due+carrier+din%2Cindustrial%2C107&sr=1-2	35.00
1	Thermocouple Jack Panel	NA	NA	Omega?		

1	6AWG Anderson connector cable	Power werx			https://powerwerx.com/single-conductor-custom-cable	Variable

Assuming a 1-stack CFC and adding 10% for incidental parts such as internal wiring, the bag-of-parts cost for the current proposed OEDCS is approximately $1500 per unit. Assembly takes quite a few hours of semi-skilled labor.

Case Mechanical Construction

The Case was made by hand by Lee Erickson, but uses standard COTS parts, as detailed below. Lee worked about 2 full weeks on both the design and manufacture of the prototype. However, having been designed, new case units could be made with approximately 11-14 hours of labor per unit. (Internal wiring and construction is a separate operation which we have not yet timed, but might take 8 hours, reducible to 2 hours with practice.)

Parts List, Front Panel

Quantity	Description	Manufacturer	Man Part Number	Distributor	Distributor Part Number	Drawing	Cost Each
1	frontpanel_Rev1_3.dxf	Public Invention	TBD	NA	NA	frontpanel_Rev1_3.dxf	
3	Panel Plate SB1 Dual			Powerwerx	PanelPlateSBDual	SB1 Drawing	14.99
6	Protective Plug with Lanyard			Powerwerx	PLUG-SB50	https://powerwerx.com/rigid-plug-cap-lanyard-sb50	4.49

3	Circuit Breaker	Carling Technologies	AA1-B0-34-615-5D1-C	Granger	10C591	Breaker Drawing	14.77
3	GFI Receptical			Harbor Freight	57958	https://www.harborfreight.com/15-amp-125v-self-test-gfci-duplex-outlet-57958.html	9.99
1	Snap Strip for Thermocouple		MMS-12	Omega	MSS-12	SnapStrip Drawing	12.34
1	DB25 Panel Mount, FanControllerBareBoardPCB	PublicInvention	TBD			FanControllerBareBoardPCB.jpg	
1	DB9 Panel Mount, I2C interface	PublicInvention	TBD				

This sums to $158 for the front panel (not including labor.)

Some deviations of note:

Lee drilled and tapped the holes for mounting the GFCI receptacles for 6-32 screw holes and screwed the GFI from the rear with shortened screws. The protruding ends of these screws are a minor hazard for scratching flesh.
Lee drilled and tapped the screw holes for 10-32 for mounting the ground bar. I attached a single tethered connector cap for the Anderson connector with a plastic cable clamp to a near by screw mounting the circuit breaker.

Parts List, Enclosure

Parts List (Partial)

Quantity	Description	Manufacturer	ManPartNumber	Distributor	Distributor Part Number	Drawing	Cost Each
1	BottomPanelTechDrawVersion120230406_0024.pdf	PublicInvention	TBD	NA	NA	BottomPanelTechDrawVersion120230406_0024.pdf	
1	RearPanel2Version120230407_1358.pdf	PublicInvention	TBD			RearPanel2Version120230407_1358.pdf	
1	Side Panel Left	PublicInvention	TBD			None, Freehanded.	
1	Side Panel Left	PublicInvention	TBD			None, Freehanded.	
10 approximately	2020 Corner Bracket Plate Anodised L Shape 5 Hole 90 Degree Outside Joining Plate			Amazon	10 approximately		1.50
150	2020 Series Screws and T Nuts T-Slot Nut Hammer Head Fastener, (M5)			Amaz	https://www.amazon.com/Fastener-Nickel-Plated-Sliding-Aluminum-Profile/dp/B086MKNYDS/ref=pd_bxgy_		0.06

				on	vft_none_img_sccl_2/135-2622305-4888003		
2	400mm V Slot 2040 Aluminum Extrusion European Standard Anodized Black Linear Rail			Amazon	https://www.amazon.com/Aluminum-Extrusion-European-Standard-Anodized/dp/B0BZ57L6CB/ref=sr_1_1_sspa		9.50
4	400mm T Slot 2020 Aluminum Extrusion European Standard Anodized Linear Rail			Amazon	https://www.amazon.com/European-Standard-Anodized-Aluminum-Extrusion/dp/B099MRRKJ2/ref=sr_1_2_sspa		4.75
4	700mm T Slot 2020 Aluminum Extrusion European Standard Anodized Linear Rail			Amazon	https://www.amazon.com/Aluminum-Extrusion-European-Standard-27-56inch/dp/B09SF2TX3G/ref=sr_1_1	Cut down for cabnet length.	10.00
1	Rocker Switch DPST 4 Pin Red Lighted 120V Rocker Toggle Switch ON Off			Amazon	https://www.amazon.com/Yoiilnz-250VAC-125VAC-Rocker-Lighted/dp/B0BB9KJDCN/ref=sr_1_1		2.73
1	10 ft - 12 Gauge Heavy Duty Lighted SJTW Indoor/Outdoor Yellow Extension			Amazon	https://www.amazon.com/10-Extension-12-Gauge-Grounded-Power-Cord/dp/B0835TCLF5/ref=sr_1_2		
1	1-1/4-in Die Cast Zinc Clamp-on Type Service Entrance Connector Conduit Fittings			Lowes	https://www.lowes.com/pd/Sigma-Electric-ProConnex-1-1-4-in-Clamp-on-Type-Service-Entrance-Connector-Conduit-Fitting/1087431		2.65

1	3/4-in Die Cast Zinc Clamp-on Type Service Entrance Connector Conduit Fittings			Lowes	https://www.lowes.com/pd/Sigma-Electric-ProConnex-3-4-in-Clamp-on-Type-Service-Entrance-Connector-Conduit-Fitting/1087283		1.50
1	1/2" NM Clamp Type Cable Connectors			Amazon	https://www.amazon.com/Connectors-Metallic-Conduit-Protect-Silver-Zinc/dp/B09M3MKL74/ref=sr_1_1_sspa		0.91
3	DIN Rail Slotted Aluminum RoHS 12" Inches Long 35mm Wide 7.5mm High			Amazon	https://www.amazon.com/Pieces-Slotted-Aluminum-Inches-7-5mm/dp/B079TX7WDQ/ref=sr_1_1_sspa	Cut to fit.	5.00

This sums to approximate $125 in parts for the enclosure. The labor costs to build it are much higher.

Other: This does not include the internal wiring description. Does not include the COG power supply, Due controller, thermocouple amplifiers, Fan Relays, air heater Controllers, This does not include systems external to the enclosure such as thermocouple wires, air flow sensors or their wiring.

References

[UNICEF] Statistics on Childhood Pneumonia Deaths: https://data.unicef.org/topic/child-health/pneumonia/#:~:text=Pneumonia%20kills%20more%20children%20than,This%20includes%20over%20200%2C000%20newborns.

[NASA Live Demo] NASA Live Demo of Medical Oxygen Generation Using Ceramic Ion Transport Membrane (ITN) Technology

https://www.youtube.com/watch?v=VUg6Jj4lc_k&t=2s

[Eutectic Fuse] Eutectic fuse provides current and thermal protection under high vibration, N. Ierokomos. https://ntrs.nasa.gov/citations/19670000534

Appendix A: The Fan Controller

We chose to make a custom printed circuit board (PCB) for the fan controller because when this contract began the specifications led us to use two fans, each of which was a dual-fan assembly. Since each was a 4-wire fan, this created a messy wiring situation, which was mitigated by placing the wiring on a PCB.

American Oxygen has since decided to focus on the Cross-flow Cylinder design, and has specified a different blow, though we still need it to be a 4-wire blower. This obviates the problem of the wiring.

However, we still must get 2 amps of power out of the case, through a connector, and into a 2-meter cable and into the fan. Each wire in the DB25 cable can carry only 0.5 amps, but by devoting 4 wires to 24V and 4 wires to GND, we solve that problem. The PCB design is in principle a good way to do that and to use a DB25 connector, although this solution does not need so many wires. We intend to keep the DB25 connector and the fan controller until a redesign is appropriate.

Appendix B: The Flow Sensor

The process air needs to have sufficient flow to sweep away oxygen depleted air and to keep a stable thermal environment around the stacks, but not so much that it drives up the cost of producing O2 by requiring more heating than is necessary. Furthermore, some modes, such as “warmup” and “cooldown” and “emergency shutdown” and even “off” obviously require different flows.

The flow is loosely proportional to the power given to the fan and to its revolutions per minute (RPM). However, the airpath might differ in different Oxygen Engines, and could even change over time—for example, a mistake or mishap or breakage could obstruct the flow. It is therefore useful to measure the flow directly, rather than merely the fan speed, in some circumstances. Whether those circumstances are likely to occur is debatable.

The Sensirion SFM3400-33-D is a medical industry standard for measuring flows less than 250 standard liters per minute (slpm). It has a shelf life of 3 years. We have tested it personally and it is astoundingly accurate even at very low flows (1% against 500 ml measured with a syringe.) However, it has two disadvantages for this application: it reduces the airflow to a hole smaller than a 22mm-diameter circle, and it uses I2C. The current blow selected by American Oxygen

has an output port of 28mm x 48mm, considerably larger than this sensor. Although I2C is a reliable protocol commonly used between chips near each other on the same printed circuit board, it is problematic on a 2-meter cable.
We investigated using a nifty COTS product to allow I2C to operate over 2 meters: the LTC4311 I2C Active Terminator made by Adafruit. Unfortunately after some success we have not been able to get this to work, possibly due to a lack of development time.

We also implemented a simpler and cheaper “hot wire” flow sensor, the MOSTPLUS flow sensor used as an automotive part. However, it is designed to operate on a flow at least one order of magnitude higher than what we need for this application, and is not terribly accurate for low flows. This may not be a problem. It is an inherently 12V analog device. Using the Analog-to-Digital converter built into the Arduino Due and a voltage divider circuit, we tested this in the Ribbonfish prototype prior to this contract.

The options therefore appear to be:

1. Don’t use a flow sensor, and rely on the fan tachometer and assume that there is no change in the physical flow path that needs to be accounted for.
2. Implement the MOSTPLUS flow sensor and accept that it will be imprecise.
3. Put more work into getting the I2C to be robust at a long cable length.
4. Use an Arduino Nano directly at the flow sensor position and use RS485 or some other differential serial protocol to read the flow. (This adds some complexity.)

As of May 31st, we believe more study and blower testing is required to make a recommendation between these options.

Appendix C: The Stack Power Supply

We have so far been happy with the TF800A12K digital power supply. However, at low amperage commands, the amperage that it reports can be off by 0.2 amps. At a commanded 25 amps, it reports providing 25.16 amps. As the voltage and amperage increase, it becomes somewhat more accurate. The fact that it is inaccurate at low voltages is documented in the manual of this power supply.

A 30-amp-Power Experiment

In order to “shake out” and “burn in” the power supply and more importantly our complete cabling system, we experimented with it at high amperages. (We had previously experimented quite a bit at less than 3 amps.)

In order to do this, Lawrence Kincheloe created a “simulated stack” — a resistor of 0.4 ohms which should be able to survive to 800C (red-hot.) This was created by wrapping nichrome wire around a ceramic tube. We made the simplest possible connection to the lugs of our 6 AWG cables carrying the DC power via an Anderson connector, as shown below. When we did this and ordered the power supply to go up to 25 amps (which it determined required 11.01 volts), we tested the complete “stack circuit”. This is an important but partial test.

We noticed no particular heat problems in the enclosure. At this amperage, our ceramic “simulated stack” emitted some acrid smoke for a few minutes. It got nearly red hot. The cables got warm to the touch (about 135 F) very near the heater end of the cable. We set a small fan to blow over these connectors to keep them cool. This seemed completely stable at a test of 20 minutes. A watt meter at the 110VAC input to the enclosure read 325 watts. The computed power to the stacks was 11.01 V * 25.16 watts = 277.02 watts, so assume that the power supply itself (which claims to be about 90% efficient) is using the remaining 48 watts (there may also be small losses in the power switch and wiring into the power supply.

We conjecture that we must be very careful with the connection to the stack, which has to flow from a high-heat region to a room-temperature region. However, using silver wires, which we did not have, would make this better. We cannot say how much of the undesirable heat at the tips of the cables came from resistance in the lug and conductance of heat from the direct connection to the nichrome wire.

We then increased the amperage to 30 amps, the maximum we could pull with 12 volts with our 0.4 ohm resistor.

This caused the power supply to apply 11.79 volts, and 30.25 amps, or 356.65 watts. The watt meter at the wall read 413 watts. This caused a small portion of our resistor to be fully red hot, as shown in the photo below. After 20 minutes the cables ends were still not painful to touch.

Figure 16: A top view of the open-kiln with a red-hot simulated stack load.

A 60-amp Power Experiment

In order to test fully to 60 amps, we purchased a $230 power resistor with a low ohmage of 1/4 ohms.

Figure 17: The power resistor attached to the 60A, 12V stack cables tested at 60A.

When connected this way (not loosely connected to nichrome), the ends of the cables did not get noticeably warm at 30 amps after 10 minutes. At 40 amps the resistor itself does get warm, but the cables do not.

At 60 amps, the power supply applied 8.78 volts, putting 526.8. watts into the power resistor. At this amperage, the cable was noticeably warm at the middle of the cable, but only a few degrees higher than room temperature. The cable ends were getting hot to the touch but were in close proximity to the radiating resistor. We recommend using a fan to carry heat away from the power resistor if being used for more than 10 minutes. Doing that, the cable ends were not hot to the touch after 10 minutes at full power (60 amps) against a 1.4 ohm load.

Appendix D: On Arduino PWM Control

Brushless DC Motors can sometimes be controlled with Pulse-Width Modulation of the incoming power signal, but doing so tends to wear out the motors and is problematic in general, because it can react with the internal electronic controller in mysterious ways. Fans and Blowers designed to accept PWM solve this problem by designing for it.

We therefore recommend 4-wire fans and blowers rather than the one that is currently preferred by AmOx, which is a simple brushless DC motor, requiring a smooth lower-voltage signal to decrease the fan speed.

The Arduino UNO and Due produce a 1KHz PWM signal by default. However, most modern fans want a much faster PWM signal, usually 25KHz. These microcontrollers can be modified (in software code) to provide much faster signals (we have done it), but it is complicated and not well documented. Furthermore, each kind of microcontroller requires different and very opaque settings to make this happen. Please see our code:
https://github.com/PubInv/NASA-COG/blob/develop/experimentation/TestPWM/TestPWM.ino
…for an example of this.

A useful overview of various fan control mechanisms can be found here:
https://www.analog.com/en/analog-dialogue/articles/how-to-control-fan-speed.html

As of June 7th, American Oxygen was planning to use the 9BMB24K201 3-wire blower.
We recommend a change to the model of this blower. We have tested this model and our slightly different recommended model extensively, and believe this minor change will be imperceptible to AmOx if used it in an analog mode (with a bench-top power supply, for example), but will be a huge time saver for the OEDCS.

We recommend the use of the 4-wire version of the 3-wire blower currently in use. This is widely available at Mouser (see link below.) This model is also a Sanyo Denki San ACE 24V model, but has model number: 9BMB24P2K01, instead of 9BMB24K201. These model numbers are rather difficult to keep straight!

Here is a link to the P2K01 that we recommend:

https://www.mouser.com/ProductDetail/Sanyo-Denki/9BMB24P2K01?qs=t9Lg9qrXjEyExan66RDEjA%3D%3D

Lee Erickson has tested the 4 wire fan extensively, with beautiful graphs, which are documented here:

https://github.com/PubInv/NASA-COG/issues/81

His tests show that you can treat it just like a 3-wire by simply controlling the voltage between 24V and 8V and leaving the PWM wire unattached. Below 8V it stops turning. We do not believe you will be able to notice a difference.

Perhaps surprisingly, the 3-wire version of the blower (the one that AmOx specified) reacts BADLY with a chopped voltage signal. This would not be true of a brushed motor, and in fact in my experience is usually not true of even brushless motors, but for this particular blower, it is awful. If we don't make this change, our only hope for blower control will add at least $300 to the project hardware BOM, and probably 24 or more hours of programming complexity as well.

Note that both Lee and I verified that the 3-wire fan is not controllable with a normal PWM signal, which Lee has documented very nicely here:

https://github.com/PubInv/NASA-COG/issues/78#issuecomment-1577409514

Note, however, that PWM can operate at various frequencies, which could affect the performance. However, I specifically raised the PWM frequency to 30K, checking it with an oscilloscope, using some rather arcane register settings on a Due, and it appeared to make no difference---the 3-wire blower is just hopelessly uncontrollable without smooth DC power. Presumably its internal electronic motor controller makes this so.

We tested our recommended 4-wire fan at the slow 1KHz PWM frequency and it seemed to work fine.

Appendix E

> **“Greenspun's tenth rule of programming** is an aphorism in computer programming and especially programming language circles that states:[1][2]
>
> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.”

The configuration of a dynamic system is generally called “scripting”. Scripting can become so complicated that it becomes programming, with all the complexity that entails. In this situation we want to make the scripting system just powerful enough to accomplish what we need to accomplish with only the absolutely necessary complexity.

The definition and interpretation of this script is still being developed at the time of this writing, but can be usefully described.

In the script below we have certain fields:

- Name: A name that will be repeated in the data logs to make data interpretation easier
- TimeStamp: A timestamp, measured in “Unix Epoch” (seconds since the start of 1970.
- DryRun: A true/false value used for testing.
- Nonce: A number used to establish when the script has changed. If the file has a “nonce” number higher than the last “nonce” read, the system enacts the “new” script. When a new script is executed, the “state” of the finite state machine stays the same.
- MaxDeltaC: A global maximum rate of change in degrees C per minute. This is intentionally designed to allow a safe delta to be exceeded (for the duration of the script) for the purpose of destructive or limit testing. The DeltaC will in no case be exceeded.
- States: This is a list of states, identified by name (see diagram of states in the finite state machine earlier). Each state consist of a list of Phases
- Phases: A phase has:
    - Duration: The time (in seconds) to execute the settings that follow in this phase. A negative Duration means “continue forever, or until you leave this state for some other reason”.
    - Fan: A setting of Speed as a percentage of maximum speed.
    - Heater:
        - Temp: The target temperature (in degrees C)
        - Current: A maximum current to apply to the heater
        - Ramp: An override of the maximum allowable change in degrees C
    - Stack:
        - Temp: The target temperature (in degrees C)
        - Current: A maximum current to apply to the heater
        - Ramp: An override of the maximum allowable change in degrees C

Consider this example (and then see notes below):

```
{
 "Name": "Experiment#1",
 "TimeStamp": 1680759024,
 "DryRun": "false",
 "Nonce":357,
 "MaxDeltaC":5,
 "States":
 [
  {
   "Name": "Warmup",
   "Phases": [
    { "Duration": 3600,
          "Fan": {
            "Speed": 20
          },
          "Heater": {
```

```
            "Temp": 400,
        "Current": 10,
        "Ramp": 40
            },
            "Stack": {
              "Temp": 400,
        "Current": 60,
        "Ramp": 5
            }
    },
    { "Duration": -1,
            "Fan": {
              "Speed": 20
            },
            "Heater": {
              "Temp": 700,
        "Current": 10,
        "Ramp": 3
            },
            "Stack": {
              "Temp": 700,
        "Current": 60,
        "Ramp": 0.5
            }
    }
  ]
},
{
  "State": {
    "Name": "Operate",
    "Phases": [
      { "Duration": 3600,
            "Fan": {
              "Speed": 10
            },
            "Heater": {
              "Temp": 700,
        "Current": 10,
        "Ramp": 0.5
            },
            "Stack": {
              "Temp": 730,
        "Current": 60,
        "Ramp": 0.5
            }
      },
      { "Duration": 7200,
            "Fan": {
              "Speed": 10
            },
            "Heater": {
              "Temp": 730,
        "Current": 10,
        "Ramp": 0.5
```

```
            },
            "Stack": {
              "Temp": 760,
        "Current": 60,
        "Ramp": 0.5
            }
       }
      ]
    }
   },
   {
    "State": {
     "Name": "Cooldown",
     "Phases": [
      { "Duration": -1,
            "Fan": {
              "Speed": 30
            },
            "Heater": {
              "Temp": 35,
        "Current": 10,
        "Ramp": 4
            },
            "Stack": {
              "Temp": 35,
        "Current": 0,
        "Ramp": 1
            }
       }
      ]
    }
   }
 ]
}
```

It is important to note that this script is interpreted as an override of limits which are coded in C++. For example, the system can still enter an Emergency Shutdown due to a fault internally coded.

Because we retain the structure of the finite state machine for simplicity and safety, this script is not like a general purpose programming language. Nonetheless, there are some subtleties as to how it should be interpreted. For example, when should a parameter be interpreted as a target, and when as an absolute maximum? What is the transition condition between states, and is it changed by these phases? If a Duration is specified in the Warmup phase but the target temperature is reached before that time, should the transition to the operating state occur immediately, or when the duration is up?

We will not be able to completely specify a precise interpretation until we have some experience with the scripting. As we refine the script over time with experience, we will learn how to make it clear and usable for power scripting.