2020-08-13

Modular 3D Printer System Software For Research Environments

Clayton D. Ramstedt
Brigham Young University

Modular 3D Printer System Software For Research Environments

Clayton D Ramstedt

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Greg Nordin, Chair
Adam T. Woolley
Philip B. Lundrigan

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

Modular 3D Printer System Software For Research Environments

Clayton D Ramstedt
Department of Electrical and Computer Engineering, BYU
Master of Science

The Nordin group at Brigham Young University has been focused on developing 3D printing technology for fabrication of lab-on-a-chip (microfluidic) devices since 2013. As we showed in 2015, commercial 3D printers and resins have not been developed to meet the highly specialized needs of microfluidic device fabrication. We have therefore created custom 3D printers and resins specifically designed to meet these needs. As part of this development process, ad hoc 3D printer control software has been developed. However, the software is difficult to modify and maintain to support the numerous experimental iterations of hardware used in our custom 3D printers. This highlights the need for modular yet reliable system software that is easy to use, learn, and work with to adapt to the unique challenges of a student workforce. This thesis details the design and implementation of new 3D printer system software that meets these needs. In particular, a software engineering principle-based design approach is taken that lends itself to several specific development patterns that permit easy incorporation of new hardware into a 3D printer to enable rapid evaluation of and development with such new hardware.

Keywords: SLA 3D printing, microfluidics, lab on a chip, system software architecture

ACKNOWLEDGMENTS

Thank you to my advisor Dr. Greg Nordin and my parents Greg and Susan Ramstedt, whose constant stream of support, encouragement and good ideas made this thesis possible.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Modern scientific research relies heavily on specialized tools. Some of these tools are simple, elegant and powerful, like a whiteboard filled with mathematical derivations and diagrams. Other tools are large and complex, so complex, in fact, that the time spent getting the tools to work far outweighs the amount of time doing actual science.

The field of microfluidics is a prime example of this phenomenon. Microfluidics, the science of manipulating and controlling fluids on the scale of a microliter or less, is a tool frequently used by chemists, biochemists, biologists, and microbiologists to exert fine control over liquid handling for a variety of research and biomedical applications [1] [2] [3]. Often devices are constructed to facilitate a specific experiment. These devices, known as microfluidic devices or lab on a chip (LoC), are typically a block of material that has an interconnected network of tiny voids, like hallways inside of a building, running throughout it, acting as conduits for fluids. Again, similar to hallways, there can also be doors (valves) that can be used to control the flow of the fluid, or even to construct more complicated features like pumps or mixers [4].

As a tool for scientific experimentation, microfluidic devices are superb. However fabricating a LoC can be challenging. Since first proposed in 1998 [5], Polydimethylsiloxane (PDMS), a silicone-based elastomer, has been a popular choice for LoC prototyping using a method known as soft lithography [6]. In soft lithography photolithographic cleanroom processes are typically used to create a mold using photoresist on a silicon wafer. PDMS is then either poured or spin-coated onto the mold, and, following thermal curing, the PDMS layer is separated from the mold and aligned and bonded with similarly molded PDMS layers to create a LoC device. Prior to alignment and bonding, holes are typically punched in various layers to enable layer-to-layer fluidic connections and off-chip connections.

While effective and capable of high resolution features, cleanroom processes for making molds are expensive and time consuming. Cleanroom startup costs involve millions of dollars, followed by significant continuing costs for operation and maintenance. [4] And while designing a new LoC mold isn't hard, being able to fabricate it correctly and repeatedly is. Every new design has a learning curve that the designers must go through before the fabricated LoC mold can be considered reliable. It can take weeks or months before the learning curve has been overcome for a single design. Taken together, these disadvantages severely limit the usefulness of photolithography as a manufacturing process for PDMS based LoC, especially in a research setting where rapid prototyping is ideal. For this reason cheaper fabrication techniques that ultimately produce larger devices are popular.

In recent years an alternate manufacturing technique has been proposed for the production of LoC: 3D printing [7]. While initially 3D printers did not have the resolution to produce microfluidic channels that many researchers would find useful, as of 2017 [8] our group not only demonstrated that it is possible to shrink the minimum reproducible feature size of stereolithographic (SLA) 3D printers to sizes that are usable for LoC, but the feature sizes are in the range of soft lithography based LoC produced with photolithographically defined molds.

3D printing brings several advantages over PDMS based LoC:

- Cost: Depending on the technique used to produce the molds for the soft lithography, 3D printing can be significantly cheaper or more expensive than PDMS. Given that a 3D printer has performance most similar to LoC produced with photolithographic molds, a 3D printer capable of printing LoC is around $50,000 in equipment costs, with recurring expenses being much lower than photolithography comparatively [4].

- Time: microfluidic devices can quickly be designed using any 3D modeling CAD software that easily can turn those designs into a 3D printable format and modify existing designs, unlike soft lithography which requires new molds to be created to modify the device which is a time intensive process. Additionally once the PDMS bulk has been cured, the individual layers of the device need to be aligned and bonded to construct the final device. All told, producing a new PDMS device typically takes a day or two compared to the time required to manufacture and test a 3D printed device is typically under 15 minutes even for prints with several hundred layers.

- Materials: SLA 3D printers use photosensitive resin as the build material. By adjusting the chemical constituents of the resin, new properties can be controlled in the device, like bio-compatibility or mechanical rigidity of the cured material. This gives resin greater flexibility in terms of chemical properties than PDMS.

For these reasons, 3D printing is an attractive alternative to soft lithography that uses photolithographcally produced molds. However as a manufacturing technique it still has a number of teething issues, such as producing devices correctly on a consistent basis. Whereas photolithography has been used to create integrated circuits since the 1960s and has that experience to draw from when creating LoC, 3D printing LoC was only proposed in 1994 [7], however much of the research in the field dates to the early 2010s, coinciding with the boom in popularity 3D printing experienced at that time. Given the newness of the technology, engineers are stll working on overcoming the learning curve of the manufacturing process.

The process to overcome the 3D printing learning curve is the same process that scientists use to do science, including, ironically, spending excessive amounts of time wrestling with tools instead of doing experiments. This problem has become very evident in our lab as we have had to shift to creating our own custom 3D printers to further research of novel microfluidic devices and designs. As new components (such as motorized stages, light engines and calibration mechanisms) are constantly being added and removed from the system to try and characterize the behavior of our 3D printers and increase the consistency and quality of the prints, it has created a tremendous strain on the system software that is used to control and collect data from all of the various electronics that make up the 3D printer due to poor software design.

Previous attempts at system software have been made, but they were lacking in three critical areas: modularity, ease of use and reliability. Modularity is the ability to add and remove hardware drivers to the system software and the ability to use a variety of tools to control the system software. Ease of use is how difficult it is to add new hardware drivers to the system software in a way that makes it work with the rest of the software. And reliability is having the architecture built on good programming practices and having an extensive testing framework to ensure that updates to the code are functioning properly. In this thesis I will outline the process taken to design modular, easy to use and reliable system software and how a create, test and register development pattern was used to realize these design goals.

This thesis will also go through the tools, architecture, code and implementation challenges of building system software for SLA 3D printers for our research group. Topics like inter-process communication protocols, APIs, hardware specific interfaces, automation, testing methodologies, hardware configuration management, web based GUI integration, and development patterns will be discussed, along with appropriate background information.

This thesis will review all of the necessary background information about previous attempts at system software as well as the tools that were selected to implement it in chapter 2. Chapters 3 and 4 will respectively provide low and high detail explanations of the architecture, with chapter 5 being an overview of the code base's organization. Finally chapter 6 explains the development patterns that can be used to streamline various aspects of the development process.

CHAPTER 2. BACKGROUND

2.1 How SLA 3D Printing Works

2.1.1 Resin and light

Devices made on a SLA 3D printer use photosensitive resin as the built material. The resin is specially designed to remain in a liquid state under normal conditions unless it is exposed to certain wavelengths of light, at which point it polymerizes and becomes a solid. The thickness of a layer of polymerized resin is is a function of how long the resin is exposed to the light, with different resins having different absorbency. By stacking thin layers of polymerized resin on top of each other with each layer varying in shape, a three dimensional object can be created. The results of this process can be seen in figures 2.1 and 2.2.

To print an object there needs to be a mechanism for controlling the exposure of light on the resin and a mechanism for stacking the layers of resin on top of each other. To control the light, a device known as a light engine is used to project an image onto the resin. The light engine uses a light source that emits a specific wavelength light to project a grey scale image on the resin. As shown in Figure 2.3, by setting the pixels in the image to black or white, a layer of resin can be polymerized with a specific, detailed pattern. The optics that are attached to the light engine are then used to reduce the overall pixel pitch to a size typically less than 10 µm, which constitutes the minimum feature size of a 3D print.

Stacking polymerized layers of resin requires a mixture of chemically treated surfaces and coordination between at least one motorized stage and the light engine. As shown in Figure 2.4, a build table that is attached to a motorized stage, lowers the most recently exposed layer into a filled resin tray until the distance between it and the bottom of the resin tray is the same as the desired layer height, creating a thin layer of liquid resin. This thin layer of liquid resin is then exposed to an image of a cross section of the device at which point the newly polymerized layer is connected to both the build table and the bottom of the resin tray.

Due to the chemical bond between the new layer to the previous layer being stronger than the bond of the new layer to the bottom of the resin tray, the build table can be moved upward to delaminate the new layer only from the bottom of the resin tray and allow for unpolymerized resin to flow in and replace the polymerized resin. At this point the process is ready to be repeated for the next layer in the device. An example of the hardware required to execute this process can be seen in figure 2.5.

2.1.2 Focus Calibration

Unsurprisingly, having the ability to make micrometer sized features requires micrometer levels of precision in how the hardware components are aligned with each other. For a SLA 3D printer this translates into getting the image that is projected by the light engine in focus relative to the bottom of the resin tray within tens of microns, similar to how the image projected by a movie theater projector is focused on a screen. Furthermore, the plane that the image is being projected in needs to be parallel with the plane made by the bottom of the resin tray, otherwise only portions of the image will be in focus.

To calibrate the 3D printer, the light engine projects its image on an angled mirror that reflects the image upward into the resin tray, which can also be seen in figure 2.5. The height of the image plane is adjusted by changing the distance of the mirror from the light engine optics, and the tip and tilt of the image plane is controlled by a gimbal that the mirror is mounted on. All three of these axes are connected to motors and can be programmatically controlled. Finally, a microscope is used to view pixels on the bottom of the resin tray to observe how focused they are. Currently calibrating the 3D printer can be a time consuming and difficult process that is entirely done by hand, however research is currently being done to try and automate this process.

2.1.3 Electronics

The bare minimum number of electronic components needed to run a microfluidic SLA 3D printer are four motorized stages (one for the build table, three for the calibration mirror), one light engine and a microscope for calibration [8]. All of these components are connected to a single computer that contains the drivers and software routines for controlling the 3D printer. Earlier iterations of the 3D printer used a dedicated desktop computer, but more recent versions are able to run on a Raspberry Pi 3B+. The reasons for this change will be discussed in the section 2.2.

It is important to note that while this basic set of electronic hardware will produce a functional 3D printer, the printer can struggle to produce consistent results, which stymies the reproducibility of data gathered using the devices made on said printer. To better understand why this is, new pieces of electronic driven hardware are frequently being added and taken away from the 3D printer to characterize the hardware and the 3D printing process as a whole. It is anticipated that some of these hardware components may become necessary permanent components of the 3D printer in the future, a process that has already occurred multiple times, resulting in the creation of entirely new 3D printers with different hardware.

2.2 Previous Attempts at System Software

As the hardware for the 3D printer has progressed, the system software that accompanied it has gone through two main evolutions, one as a desktop application and another as a web based application. The unique problems and pitfalls of both of these pieces of software are key to understanding many of the design decisions that were made when creating and implementing the current iteration of system software.

2.2.1 Desktop Application

Initially, the printer was connected to and controlled by a desktop computer running Windows with the system software running as a desktop application with the software built off the Qt frontend framework using Python. Qt, more than anything, was the cause of most of the software limitations in this evolution of the system software.

While Qt provides excellent tools for building and designing GUIs, it heavily relies on a single, small library of pre-made components that are difficult to customize. The components themselves are based on the design ideals from the early to mid 2000s, making it impossible to build GUIs that benefited from modern user interface design principles that have been pioneered by web applications over the last 15 years. This was all exacerbated by the Python version of Qt being a port of Qt's main C++ version, which contained much better documentation and support resources.

Another serious limitation was how opinionated the Qt framework could be. Qt has a very specific way that everything must be done in relation to the GUI, and this eventually caused Qt to become the core of the entire system software. Given the previously mentioned problems with the GUI, it created a situation where Qt couldn't be switched out for a better technology without throwing away the rest of the system software with it. Admittedly this problem could have been avoided if the system software had had its architecture well defined prior to adding Qt, however the students who created the code base had limited experience with software engineering and they allowed a software library that did not perform the main task of the system software to dictate how the system software was constructed.

The more fundamental problem with this iteration of the system software actually had nothing to do with the software. Having a full desktop computer with screen and keyboard run the 3D printer was expensive and it dramatically increased the physical footprint of the 3D printer. Additionally, updating the Qt frontend required creating a new binary file, which creates a dependency for how the GUI gets distributed to users that was difficult to update and maintain. All together this limited the kinds of environments in which users could access the 3D printer.

One thing that the Qt system software did do right was the usage of dummy components for development. Dummy components are used to emulate pieces of software that rely on another piece of software or hardware to function, like a database or a device driver. As a design pattern they are excellent for testing the functionality of the system software and they can allow for emulation of 3D printer behavior in the absence of hardware. However the limitations imposed by the architecture kept the dummy components from acting as truly generic drivers which significantly hampered their ability to emulate different hardware configurations.

2.2.2 Web Application

The next evolution came with the migration of the system software onto a Raspberry Pi where it could run headless as part of a web server, negating the need for the 3D printer to have a dedicated screen and keyboard. This also migrated the GUI to a web frontend, which eliminated the distribution problem that the desktop application had, as simply refreshing the browser would send the most recent version of the GUI. It also gave access to the large and powerful ecosystem of web development and networking tools and their very active communities. In short, the wider variety of tools made it much easier to create powerful system software that met the needs of the researchers.

However a number of design decisions were made that limited the potential of this evolution. For example, the tools used to build the frontend were primitive by web standards, using only Jinja templates, which provides simple customization of page content and basic if/else logic to HTML, and bootstrap, a library of simple, pre-stylized HTML components, and a grid based layout engine. When compared to modern Javascript frontend frameworks, the frontend lacked features like a dedicated debugger, integration with NodeJS and by extent, its extensive package library, unit tests, state management and a framework for organizing the code in an easily maintainable and modular way.

Another interesting design choice was to have the front and backend communicated over web sockets instead of traditional single use HTTP requests. Web sockets are useful because they provide persistent bi-directional communication between the front and backend which easily enables publish-subscribe architectures. HTTP requests on the other hand must be instigated by the frontend and the backend can only respond once to any request. However documenting the communication protocol for a web socket is harder to do than for HTTP requests.

Web sockets do not natively offer any features for structuring data that is sent through them, requiring for the parties on both ends of the socket to be sharing a common messaging schema that can be serialized into binary. HTTP requests on the other hand do provide some data structuring natively, especially for files, which makes them easier to document. Additionally, by nature of HTTP requests being constructed completely of plain text, the tools for sending HTTP requests are simpler and more readily available than the ones for web sockets, which is also due in part to their universal usage.

One area that the system software did go with a more universal model was its usage of the model-view-controller (MVC) architecture. MVC describes how the server for a web application handles responding to incoming HTTP requests. It is based on the assumption that the HTTP requests are asking the server to query a database and return the results of the queried data, possibly with the data embedded inside of some HTML / CSS.

As shown in figure 2.6, a MVC architecture is made up of three parts: the model, the view and the controller:

- The model is a representation of the data in the database and it has the functionality to read and write to the database.
- When the controller completes querying the database, it hands the result to the view, which takes care of formatting the data for consumption by the frontend and responding to the original request.
- The controller handles receiving the HTTP request, telling the model to execute a query and passing that input to the view.

Overall, MVC does an excellent job at rendering static web pages with customized data.

But the implementation of MVC needed to be modified to better accommodate the realities of the hardware. Instead of having a traditional database and models to query the database, the current state of the 3D printer hardware was considered to be the database and the hardware drivers were used as the models. This works well if all the frontend is doing is rendering the current state of the hardware and providing a GUI for sending commands to update the state of the hardware.

However during a print job, the frontend needs to display the ever changing states of the hardware and the software routine that is running the print job. The uni-directional and single use nature of HTTP requests make them inconvenient, but not impossible to use for this kind of task, which is better served by the persistent and bi-directional nature of web sockets. For this reason the MVC architecture was primarily used for initially serving the frontend, using the aforementioned Jinja templates, but a publish-subscribe architecture built with web sockets were used to send updates about the state of the 3D printer to the frontend.

Overall this mixing of architectures is a great way to structure the interface between a web based frontend and the rest of the code that controls the 3D printer. But the implementation of this architecture had its own set of problems, starting with the relationship of the backend web framework to the core functionality of the system software.

Similar to how the core functionality of the system software was tightly integrated with Qt, the web application version of the system software was tightly integrated with the web framework Flask, which will be discussed further in chapter 4. While Flask is not as opinionated as Qt, allowing for greater flexibility in code structure, the system software could not run without starting Flask, thus once again tying the system software to a software library that does not perform the core functionality of the system software.

One unfortunate side effect of how the system software was integrated with Flask was that the greater flexibility of code structure that Flask provides did not translate into well organized or structured code. The poorly defined organization of the code base translated to the core of the system software being a collection of threads that share control over the hardware drivers. While this architectureless arrangement is workable if not ideal, nothing was ever made thread safe and executing a print job relies heavily on what sequence commands are sent from the frontend and trust in how long it takes each thread to execute. It worked, but adding new hardware or modifying the print routine were a nightmare, and the entire system was a software bug bomb that was waiting to detonate.

Another structural issue in the architecture was that all of the threads ran in the same process. For most programming languages this is not an issue, but Python has a fundamental limitation known as the global interpreter lock (GIL) that restricts Python from executing multiple lines of code at once in the same process [10]. Usually multi-threaded applications get an increase in performance from the parallel execution of threads, but unless a Python application is spread across multiple processes, its performance will be bound to slightly less than that of a single thread, due to overhead lost to context switching between threads.

This results in the system software being unable to take full advantage of the computing resources on the Raspberry Pi and adding dedicated processes to software often require architectural considerations. While this limitation does not currently impact the performance of the system software, current research into automating the calibration of the 3D printer depends on cameras and mathematically intense image processing techniques, and will likely require more computing resources.

Finally the original design for the system software regrettably did not include any dummy components and instead relied on the debug features of the flask web server. In very recent additions to the software dummy drivers have been added, but structurally they require that a dummy driver be written for each new driver that is added to the system software which tremendously limits the modularity and scalability of the dummy components.

2.2.3 Summary of Lessons Learned

The following are the lessons learned from previous attempts at creating system software:

- Desktop VS Web Applications The medium over which the system software communicates to the users over matters. The debate between desktop versus web application at its core is a decision between using a communication medium that is built on a specific technology or a collection of protocols that are implemented and integrated into a large number of technologies. For this reason a web application is more flexible and more modular, to the point that the frontend of a web application could be a desktop application as well as a web page.

- The Frontend Needs Its Own Architecture The frontend needs its own architecture that is completely separate from the rest of the system software for the sake of modularity and to encourage structured and well organized code. Web applications benefit from a large and thriving ecosystem but they also require a number of additional tools to make them convenient to build.

- The Backend Needs Its Own Architecture The web server needs its own architecture that is dedicated to sending and receiving messages to the frontend. A combination of the MVC and publish-subscribe architectures work well to cover the needs of the system software.
- The System Software Core Needs Its Own Architecture The core of the system software needs its own architecture that is well defined with an organized code base that does not restrict the software from taking advantage of the full hardware computing resources. It also should not be built on top of another library that has its own architecture. Finally dummy components should be designed into the architecture from the beginning and be deeply integrated into the software to maximize their effectiveness.

Many missteps were made in the previous evolutions of the 3D printer system software that helped develop a better understanding of what the needs of system software are in a research setting. These lessons proved invaluable in the design and implementation of the new architecture.

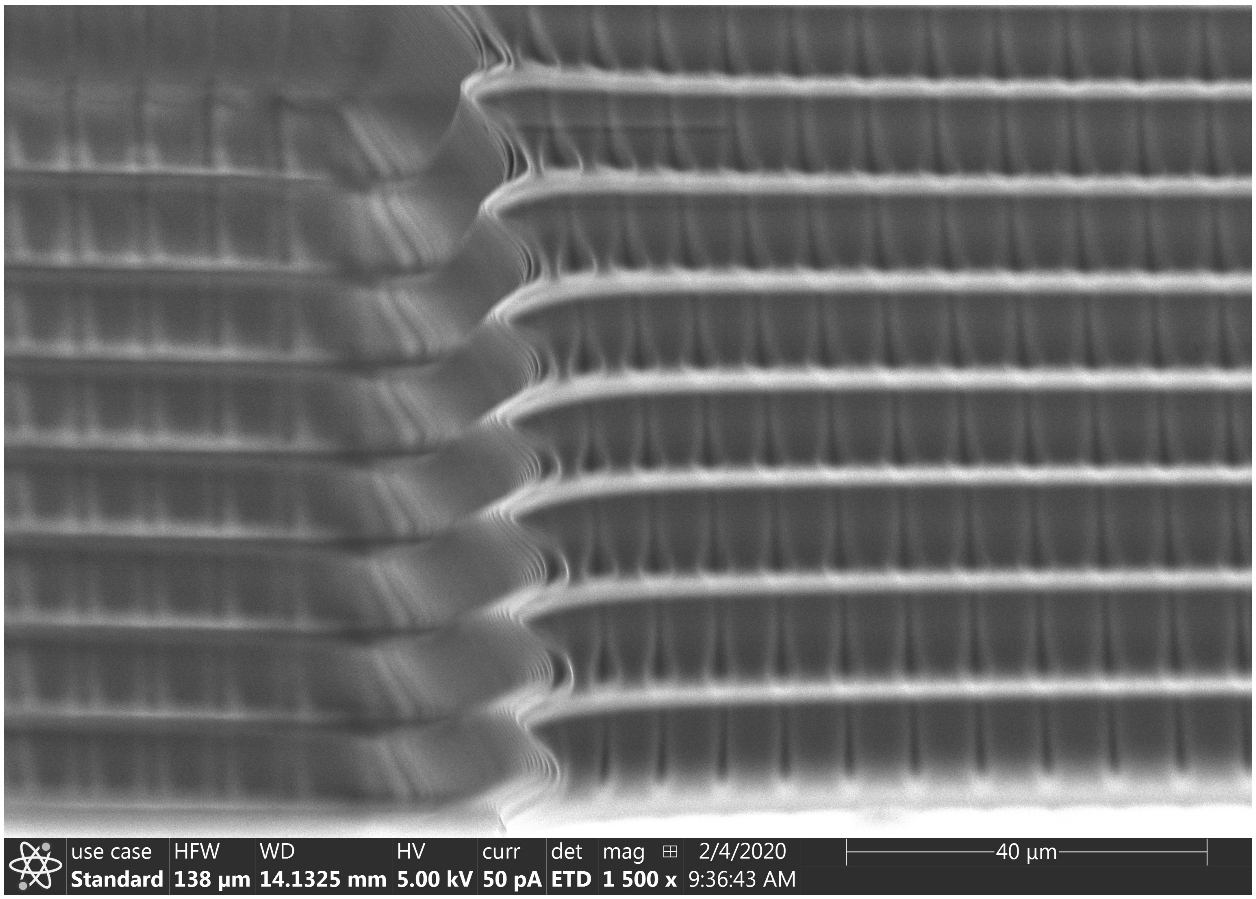

Figure 2.1: 10 µm layers stacked on top of each other as viewed under a scanning electron microscope. The dimples along the layers indicate individual pixels from the cross sectional image.

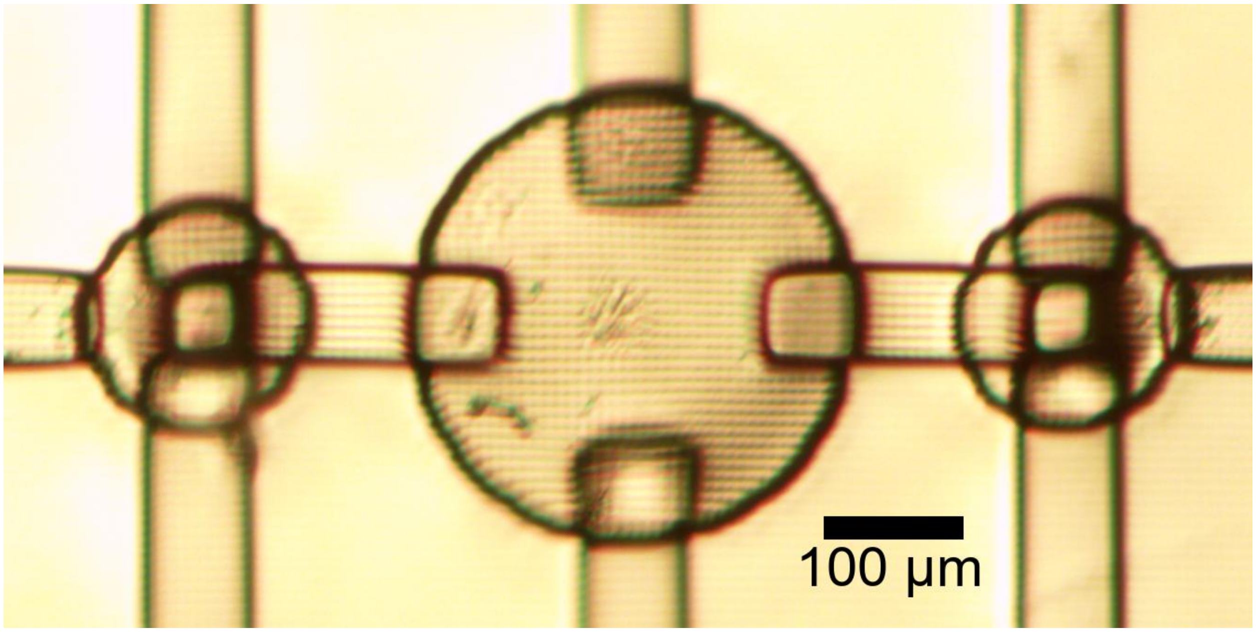

Figure 2.2: 3D printed microfluidic pump. The grid pattern is a by product of the individual pixels in the layer images.

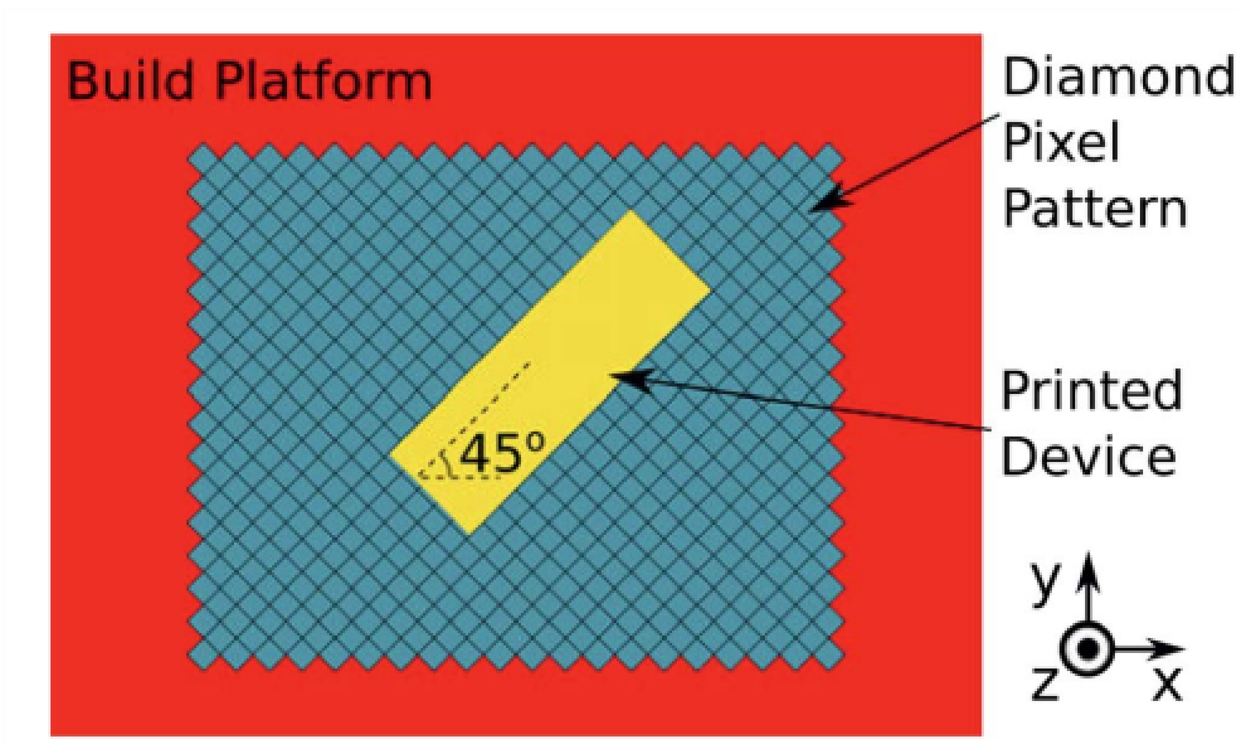

Figure 2.3: Illustration of a cross-sectional layer of a 3D print being projected as an image onto the build platform of the Asiga Pico Plus 3D printer. Grey pixels are turned off and yellow pixels are turn on. Our recent custom 3D printers operate on the same principle but do not have the pixels rotated 45°. Image used with permission from [9].

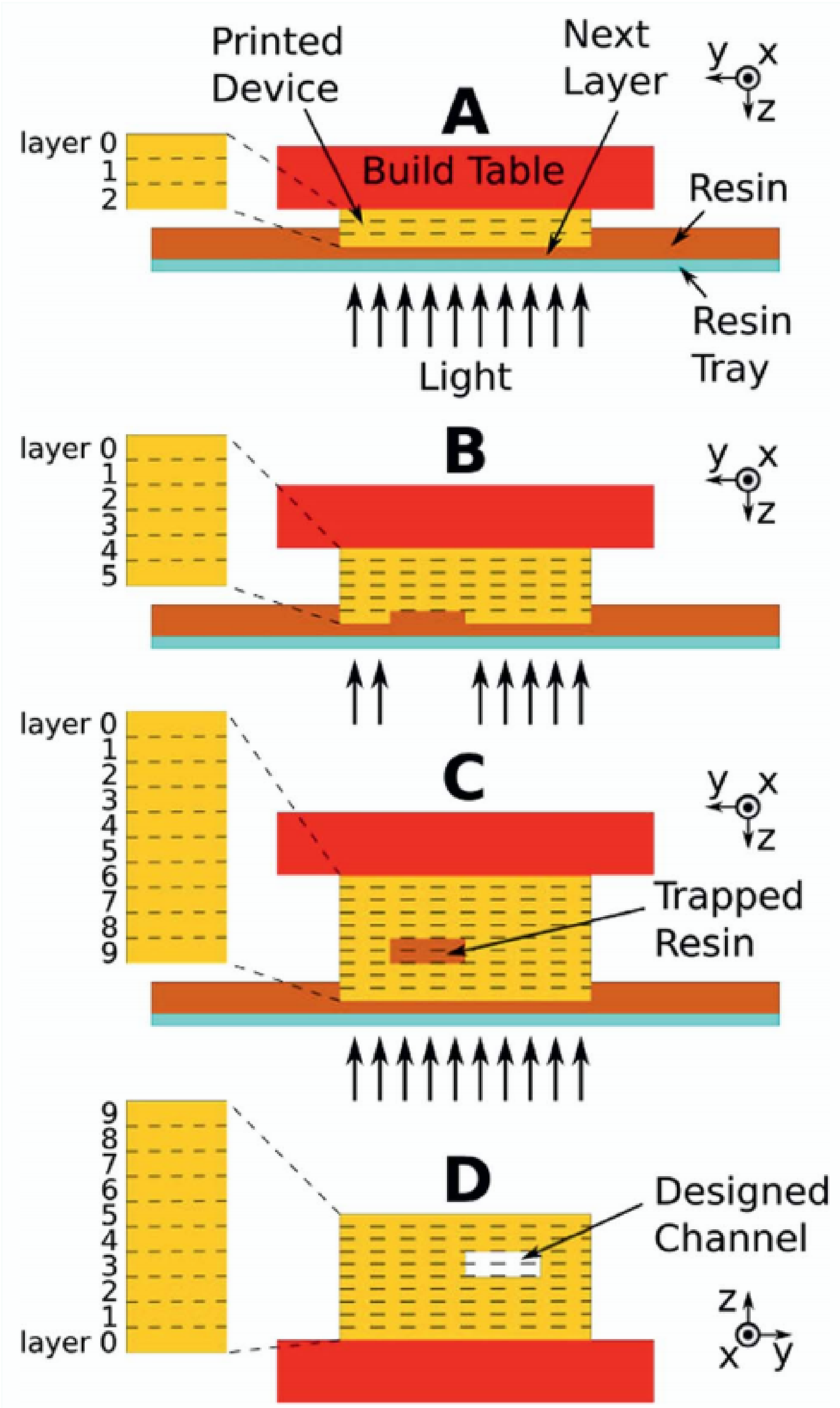

Figure 2.4: Layer-by-layer fabrication process for a simplified device. The device is rotated 180°in (D) relative to (A-C). Image used with permission from [9].

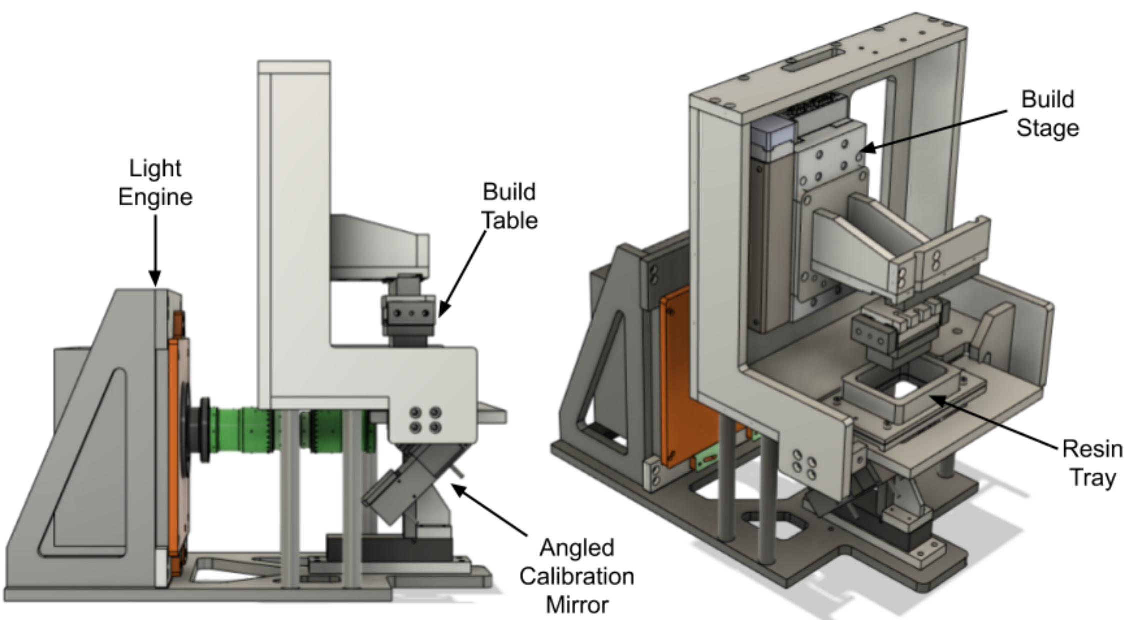

Figure 2.5: CAD model of a SLA 3D Printer

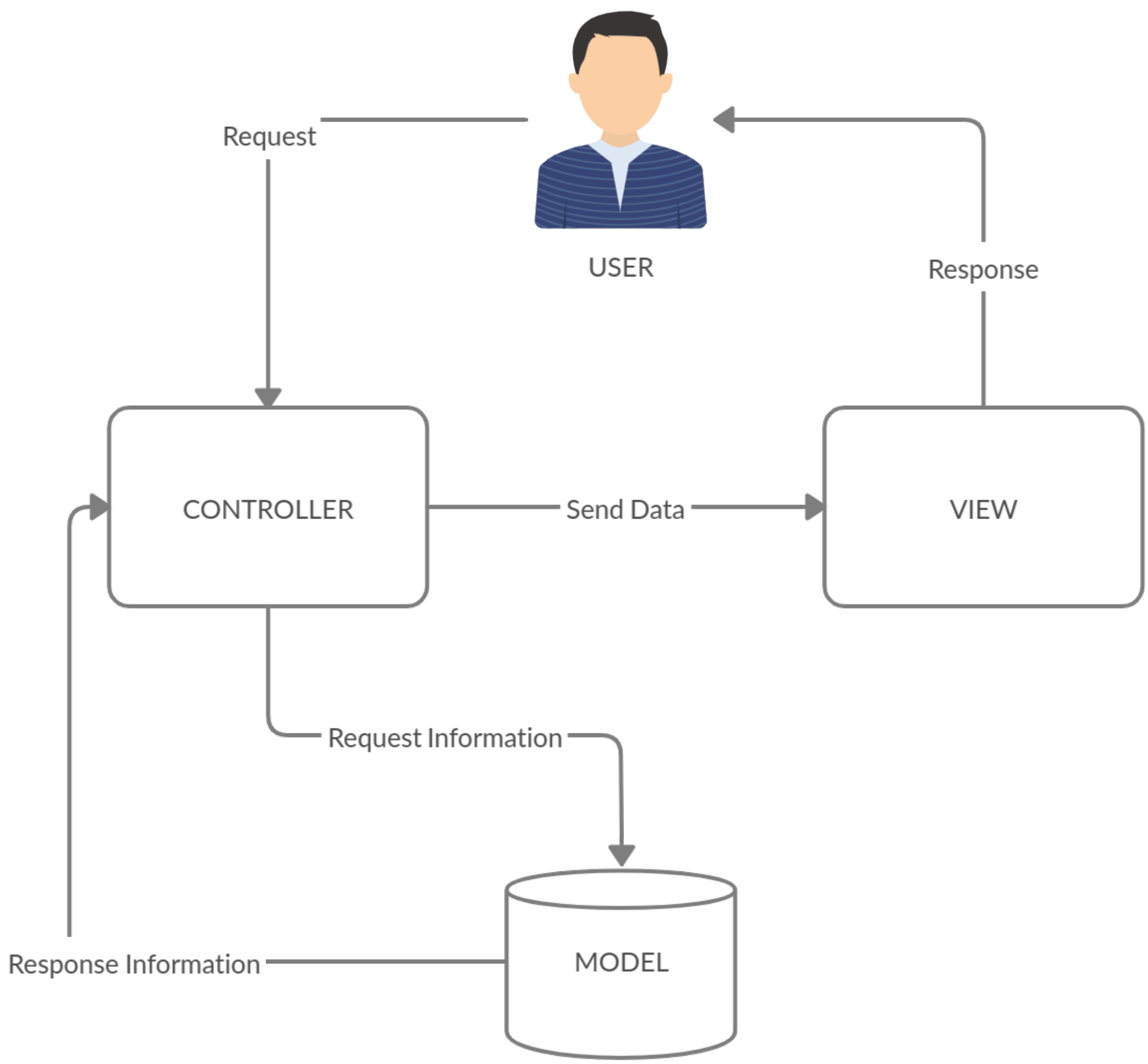

Figure 2.6: Model-View-Controller Architecture

CHAPTER 3. SIMPLIFIED ARCHITECTURE DESCRIPTION

This chapter will provide a simplified model of the system software's architecture. It will focus on the organizational units that make up the architecture, explaining what the job of each component is and what tools each component uses to accomplish its job. Chapters 4 and 5 provide more on the specifics of how the tools execute the job of each component.

3.1 Tools

As was demonstrated in chapter 2.2 the tools used to build the system software have a big influence on the architecture. Because of this, a critical part of understanding the architecture is understanding what tools were selected and why they were selected. The following is an review of the tools that were chosen to build the system software.

3.1.1 Coding Languages

Python was selected as the language for the backend for the following reasons:

- Easy to learn The primary end users of the system software are students with an electrical engineering or chemistry background. While some of the students may have prior programming experience, it is expected that most will have had little to no formal software engineering experience or education, and the system software will likely be the largest and most complicated code base that they have ever worked on. Because Python is easy to learn and has a relatively intuitive syntax, this makes the software more accessible to new users.
- Speed Python is not considered a fast language in terms of execution time. However during normal operation, the system software spends a lot of time communicating with hardware and waiting for the hardware to respond. This is especially prevalent with a

stage that controls an axis, as commanding the axis to move will result in the system software having to stall while waiting for the stage to complete the action. On top of this, many Python packages, especially mathematical computation packages like numpy, scipy and openCV, call binaries that were originally written in C/C++, and therefore are incredibly performant. In short, Python is fast enough or has the ability to be made fast enough for the task at hand.

- Rich ecosystem Python's package ecosystem is one of the largest compared to other programming languages. The package manager pip makes it quick and easy to add new and complicated functionality into the code, which makes Python excel at rapid prototyping and experimentation. These are key features given the trial and error nature of research.

As Javascript is the industry standard for web development, it was the natural choice for the frontend. Frontend development specifically was done with the use of NodeJS, a server side implementation of Javascript, which provided access to the NodeJS package manager, npm. NodeJS, like Python, has a rich and easy to access package ecosystem for frontend development and deployment.

3.1.2 Operating Systems

As discussed in chapter 2, recent versions of the 3D printer are controlled by a Raspberry Pi, which best supports the Raspbian Linux distribution. This makes Linux the production and development environment of choice, although other Unix based operating systems, like macOS, have been shown to be equally capable. Unfortunately the system software is not compatible with Windows or the Windows Subsystem for Linux (WSL 1.0), as the python implementation of mutexes for the NT kernel are not picklizable, which severely limits what architectures can be created. Compatibility has not been tested with WSL 2.0.

Bash is used to run the system software and in some cases bash scripts have been written to automate installation and starting the system software or to act as an alias for unwieldy commands.

3.1.3 Web Server

Flask is a popular backend web framework written in Python that takes an à la carte approach to structure, meaning that the framework doesn't have strict rules about how its application code needs to be formatted and organized, instead providing a loosely connected set of tools that can be adapted to a wide variety of architectures. Given that the system software requires a non-standard architecture in order to remain modular, Flask's flexibility is a critical feature. Flask also provides a convenient development server, which has been used in lieu of a dedicated web server. While this server should be replaced by a dedicated web server at some point in the future, it has proven capable of handling the current work load.

In conjunction with Flask, a package called flask-restplus is used to assist in the defining, validating and documenting of the RESTful API. Critically, the package generates a web page alongside the frontend of the system software that documents all of the API endpoints and also provides a simple interface for sending API commands to the backend, which makes it easy to test and develop new application software for the 3D printer.

Importantly, the package does this by generating a file based on the OpenAPI 3.0 specification, known as a swagger file, that contains a full description of the API. This file can be downloaded from the web server and used to automatically generate a library of functions for the system software API in a variety of languages, which drastically simplifies the process of writing new applications for the 3D printer.

3.1.4 Configuration and Print Settings Files

Due to the JSON file format already being used to describe the settings for print jobs and their nearly identical formatting to Python dictionaries, it was chosen as the file format for the configuration files. To make it easier to validate configuration files and print job settings files, schema files, based on the proposed JSON schema standard, have been created that rigorously define what constitutes as a valid configuration or settings file.

Similar to the Python flask-restplus package, there is a NodeJS package called bootprint that creates a static web page with a description of a given JSON schema, which is critical to making it easy for researchers to create and edit configuration files for the system software.

3.1.5 Frontend Framework

Between Angular, React and Vue, the three most popular web frameworks at the time of this writing, Vue differentiated itself for sharing the same à la carte philosophy as Flask, and for having a reputation for being the simplest to learn. Additionally Vue provides a number of convenient tools, such as the url router, vueter. This removes the need for Flask to handle page requests outside of an initial request for a compiled version of the frontend code, allowing for the backend and the frontend code to be completely independent and separate from each other, thus improving the modularity of the system software.

A significant feature of Vue is its component oriented philosophy, which is a version of object oriented philosophy adapted to web development. Vue components are organized into .vue files that contain all of the HTML / CSS / Javascript for a single UI component. Components can be nested inside of each other, and Vue's rendering engine ensures that the changes to the UI in one component do not impact how another component is rendered.

Vue also has the ability to make use of UI component libraries, like bootstrap, however another similar library called Vuetify was selected over bootstrap. Vuetify is a Vue friendly implementation of Google's Material Design Language that offers a much larger selection of UI components and customization than bootstrap, while also providing its own grid layout system. It also has superb documentation that focuses on providing examples that are editable directly on the documentation's web site, which creates a hands on experience to working and learning about how its API works. All of these features combine to make prototyping new, high quality user interfaces far easier than it was in the previous frontend.

Vue also has a state management system call Vuex, which allows state to be shared between multiple UI components, a useful feature for keeping multiple web pages in sync with the state of the 3D printer. And finally, there is a Vue debugging add-on available for all of the popular web browsers that augments their built in debugging tools so that the state of each component and the values of the variables in the state management system can easily be viewed and changed in a way that reflects the structure of the Vue project.

3.1.6 Unit Testing

Python has a unit testing library as part of its standard library which is appropriately called unittest. It takes an object oriented approach to testing, which enables the tests to roughly mirror the structure of the rest of the code.

3.2 Simplified Architecture Description

As shown in figure 3.1, the system software is split into three distinct sections: the frontend, the web server and the system software core. Each of these sections have their own architecture that is independent of any other section. However it is worth noting that the web server and the core are part of the same code base and often will be referred to collectively as the backend. While the code for the web server and system software core do remain separate from one another, they interface with each other and share some of the same resources, however, they remain distinct entities to the point that they each have their own set of unit tests. The relationship between the web server architecture and the core will be covered in depth in chapter 4.

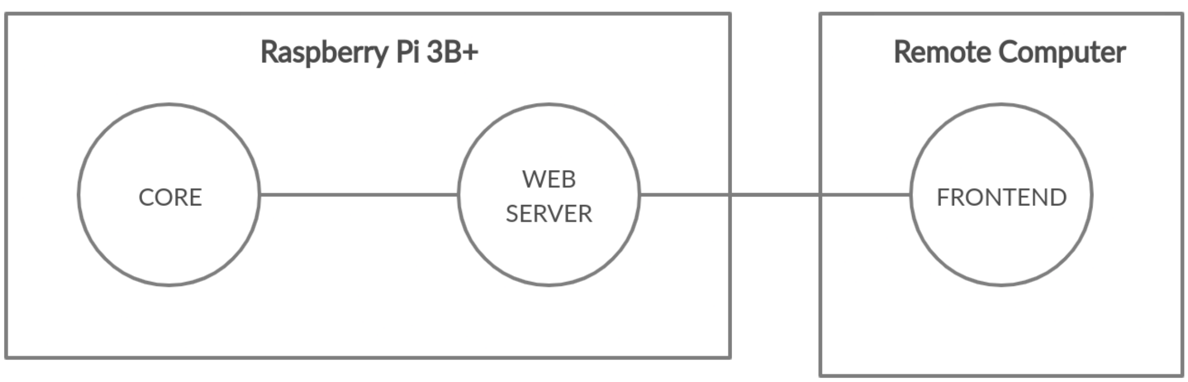

Figure 3.1: 3D Printer System Software Simplified Architecture

3.2.1 Web Server

As the web server is a go-between for communication between the frontend and the core, its design had a significant impact on the other architectures. As was discussed in chapter 2.2, a blended MVC and publish-subscribe architecture is ideal for the system software as a whole. However unlike the previous evolution, it was decided to forgo the use of web sockets and instead rely exclusively on HTTP requests.

In terms of the design goals, web sockets and HTTP requests were determined to have roughly the same level of reliability, but due to HTTP requests universal usage and having some built-in data structuring, it was considered a more modular interface than web sockets. Also from the perspective of researchers who may need to write a quick and dirty bash script to run an experiment on a 3D printer, web sockets would require access to the data structures that make up the messaging schema, which creates an additional dependency that HTTP requests do not have. Because of the ease of generating a web page with the documentation for a HTTP request based API via a swagger file, it is easier for a researcher, especially an inexperienced one, to build bash scripts or other single use programs. This is further aided by flask-restplus which generates a curl command for each API endpoint as part of the documentation.

While the exclusion of web sockets ultimately gives end users a better experience, it makes the job of the programmer and architects harder. Because web sockets no longer can be relied on to implement the publish-subscribe architecture, the frontend and core behavior must be tailored to the web server's MVC architecture to help it emulate a publish-subscribe architecture. For the frontend this means that if there is some data that it wants to become a subscriber of, it will have to continuously poll the data's API endpoint to get the most recent value. Obviously this will cause the web server to be spammed with HTTP requests unless the backend does something to accommodate for this behavior.

The web server compensates for the behavior of the frontend by creating API endpoints that are intended exclusively for subscribers. These endpoints trigger special messages to the core which tell it to respond differently to these messages. The core does this by not immediately responding to the message, but instead waiting until the data in question changes or until a periodic timer runs out before responding.

This approach works well when the data that has been subscribed to is updating infrequently and missing updates is not critical. But for data that is updating faster than the latency of the core responding to a subscriber message, the response being sent over the network back to the frontend and having the frontend send another request to the backend, then core responses will require sending the last N updates to the data with every message response. This change in behavior fits into the frontend and core architectures, however it does require more code to implement on the backend and generally some foreknowledge on the programmers part when adding features to both the front and backend.

3.2.2 Frontend

Aside from the behavior necessary to adapt the web server to a publish-subscribe architecture, the frontend uses the Vue framework described in section 3.1.5 and relies on its architecture. However because the frontend is intended to be modular, the Vue architecture should not be viewed as the definitive architecture for the frontend. Due to the HTTP based API, anything from simple scripts, to a command line prompt to an entirely different framework like Qt could possibly constitute the frontend architecture. For this reason explaining the architecture of any specific frontend is best served by its documentation, and the specifics of the Vue framework's philosophy and architecture [11] [12] will not be covered by this thesis.

3.2.3 System Software Core

The functional portion of the system software core was implemented entirely from scratch, depending as much as possible exclusively on the Python standard library. To avoid the problems with the GIL that were discussed in chapter 2.2.2, the architecture took a process oriented approach, with each type of hardware being given its own process and with each software oriented task being given its own process. Figure 3.2 illustrates the minimum required set of processes for running a 3D printer. Each of the processes do the following:

- Message Router The message router constitutes the core of the system software core. As the name suggests, it handles routing messages between components that reside

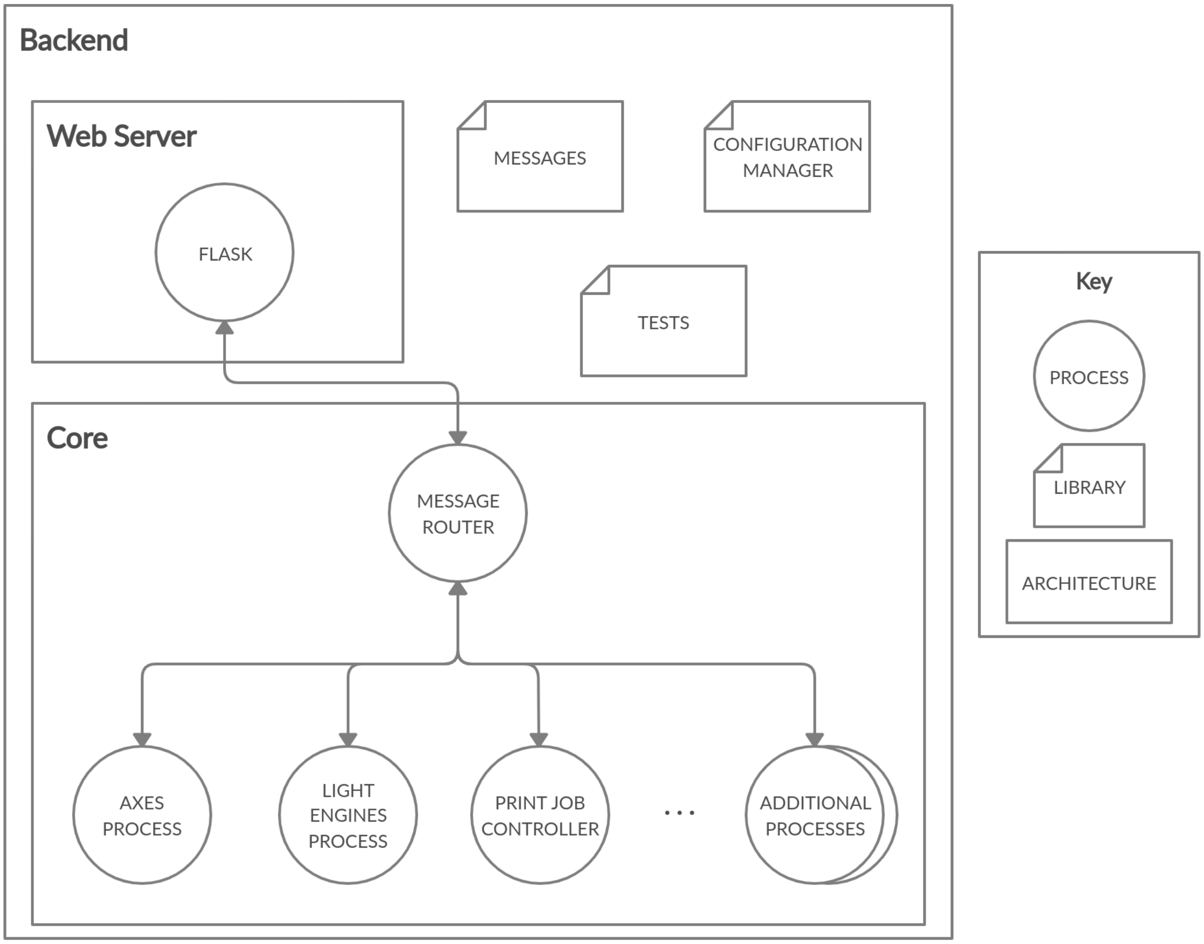

Figure 3.2: Backend architecture for a 3D printer

in different processes. It also handles communicating with the web server, which is represented here as its own process, although in the most current version of the software the web server runs in the message router process. The reasons for this will be discussed in chapter 4.1 with the details of the message router being extensively covered in the same chapter.

- Light Engines Process The light engines process encapsulates the drivers of all of the light engines that the system software is currently configured to control. It also contains code that routes messages to the correct driver, enabling the system software to control multiple light engines at once.

- Axes Process The axes process performs the same job as the light engines process but for the hardware axes. However it differs slightly from the light engines to accommodate the fact that stage drivers often control multiple axes. These differences will be discussed in chapters 4 and 5.

- Print Job Controller The print job controller contains all of the logic for executing a print job. It functions by sending messages to the light engine and axes processes through the message router based on the contents of a print job file. The print job file contains a print_settings.json file, which is formatted according to a JSON schema, and a folder of images of all of the cross sections of the print. Based on the content of the print_settings.json file, the print job controller moves the build axis to the correct layer height, sets the light engine with the image or images for that layer, and then turns on the light engine for the properly configured period of time. This process is then repeated for each layer of the 3D print.

When examined in relation to the entire backend, the benefits of the core's architecture become become more apparent.

3.2.4 Backend

While the backend encapsulates two distinct architectures, out of necessity or convenience several key libraries were created to help integrate the architectures with each other. They are the following:

- Messages The messages library is a standardized set of classes that act as the messaging protocol for talking to the processes in the core. When the web server receives a HTTP request, it creates the appropriate message and sends it to the message router. The message router then uses information in the message to determine which process to forward the message to. The print job controller uses messages in the exact same way.

- Configuration Manager When the backend is first started, it requires that a valid configuration file is provided. The configuration manager parses apart the configuration file with the help of a JSON schema and produces objects that can easily be used to initialize the web server or other pieces of the core. For the light engines and axes processes, this information determines what hardware drivers are loaded and what settings they use during normal operation, such as running in debug mode or running a serial connection at a specific baud rate.

- Tests There are three types of tests in the tests library: hardware, core and web server. Hardware tests are simple python scripts that test if a driver can execute a test sequence on the hardware. Hardware tests do not use unittest because the library has a tendency to break serial connections and the structure of the unittest library often makes hardware tests harder to code than is necessary. Core tests do use the unittest library and are built around specific processes, with each test focused on a message from the processes messaging protocol. Web server tests focus on testing the API handlers and require an instance of the backend to be running on the same computer the tests are being run from.

3.2.5 Putting it all together

Now that all of the components that make up the backend are described, it is possible to see why this architecture enables modular, easy to use and reliable system software. From a researcher's perspective, reconfiguring the system software to work with a different arrangement of hardware, assuming that the drivers have already been written and integrated into the system software, is a matter of adding the configuration information for the driver to the configuration file and restarting the backend. And tweaking hardware behavior for an experiment, such as imposing software defined limits on the travel distance of a stage, also can be as simple as changing the configuration file. Similar changes in previous evolutions of the system software would have required actual changes to the code or extensive rewrites.

If a needed hardware driver doesn't exist, say for a new light engine, but a process for that type of hardware does exist, then it can be added by writing the driver, validating the driver with a hardware test, registering the driver with correct process, creating a configuration handler for the driver and registering the handler with the configuration manager. While this requires significantly more effort than just editing a configuration file, the process is straightforward, well defined, provides a way to test the new driver and ultimately creates an easily configurable hardware driver that integrates with other preexisting software for that process, like the frontend or the print job controller. In other words, the hardest part of adding a new piece of hardware is getting the hardware driver to work, not getting the system software to work with the new hardware.

The most difficult task a researcher will regularly face is adding a new piece of hardware that doesn't already have a process for that type of hardware, like a device for calibrating a 3D printer. What makes this difficult is determining what the API and the messaging protocol need to look like, a process that is fraught with assumptions, several of which will inevitably be incorrect and require time consuming rewrites of the code. There are steps that can be taken to make this process easier and they are the topic of chapter 6.

Once the API and messaging protocols are determined adding the new hardware becomes relatively straightforward. First, messages for the messaging protocol need to be created and registered. Second, the process needs to be created, tested and registered, and a configuration handler needs to be created and registered for it. Finally the same steps for adding a new hardware driver need to be followed. Adding a process that handles a purely software task, like a new print job controller, follows a similar procedure and is discussed in depth in chapter 6.

Overall the architecture of the backend facilitates a create, test and register pattern during development. When combined with the ability to customize the processes that are running in the core via configuration files the architecture fulfills the goals of being modular, easy to use and reliable while also side stepping the performance and dependency limitations of previous evolutions of the system software.

While the simplified architecture described in chapter 3 is not terribly complex at first glance, the dependence on multiple processes allow the code to execute in multiple places at the same time, which makes the state of the system software harder to track. To exacerbate the issue, each process has multiple threads running as part of it with some of these threads running perpetually while others are being created and destroyed as needed. Together they have intricate interactions that, when performed incorrectly, can have disasterous consequences.

It is important to understand how all of the processes function on a thread level and what patterns they follow. As such, the architecture of the individual processes, how the threads inside of each process are started and stopped and how the threads function during normal operation will be the topic of this chapter. However a researcher could feasibly get away with not understanding the information in this chapter simply by following the guidelines and patterns in chapter 6 and using chapter 5 as a reference. But this model of the system software helps make critical sections of the code make sense and will provide patterns for researchers to imitate as they build their own processes.

4.1 Starting and stopping the backend

When the system software is started, it is done by calling a script like so:

```
$ python main.py
```

This script starts a new process with a single thread running inside of it. Over the course of this thread's lifetime it will perform three jobs:

1. Start all of the processes, except for the message router. This first process will become the message router.

2. Setup and perform all the jobs that the message router does.

3. Shutdown all of the processes cleanly when the stop signal is received.

The main.py thread is alive for the entire lifetime of the backend and is responsible for starting and stopping the system software cleanly. While the duties of this thread are straightforward, technical limitations in Flask's development web server forced this thread and process to be architected differently than the ideal.

Ideally, as was shown in Figure 3.2, the web server should run in its own process and communicate with the message router the same as the other processes. However Flask's web server was not designed to run as a background task and as such does not provide an API for stopping the web server in software. This requires the web server to run as a foreground process and can only be stopped from the terminal. If done incorrectly, the web server threads will be orphaned when the system software shuts down, which can cause problems with access to networking resources and, if done repeatedly, will cause the Raspberry Pi to run out of RAM.

The easiest workaround for this problem was to integrate the web server into the main process, which consequently combined the message router and web server into a single process. To date this has not slowed down the backend noticeably and the main.py thread is able to shut down everything cleanly. Unfortunately this hurts modularity of the system software as the core and the web server are connected to each other, although steps have been taken to isolate their code from each other as much as possible. This technical limitation is one of several reasons why Flask's development web server needs to be replaced by a dedicated web server.

4.2 Structure of the message router process

Because of the compromise that had to be made on the web server's account, the thread structure and management of the message router process was inevitably complicated by the addition of the web server. Normally the message router is only made up of the core main.py thread, and a series of threads that each are dedicated to handling incoming and out going messages for each process connected to the message router.

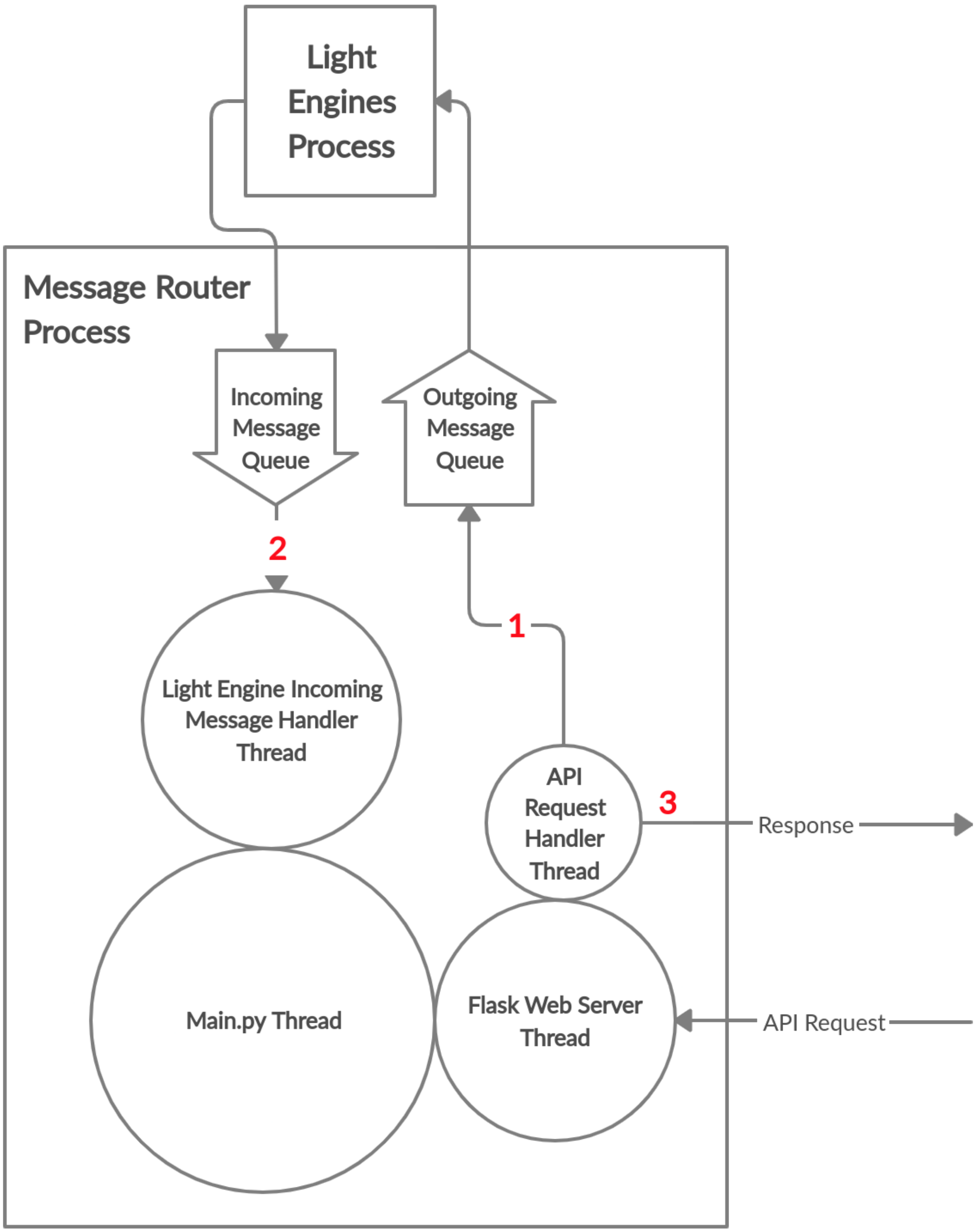

Figure 4.1: Threads used in the message router process to service a light engine api request. Threads that are directly adjacent to each other have a parent-child relationship to each other, with the thread with the largest circle being the parent. Numbered connections relate to the explanation in section 4.2.1 and correlate to Figure 4.2.

As can be seen in Figure 4.1, the message handler threads that normally would have been reserved for the web server process have been replaced wholesale by the Flask web server. This allows the web server to run in the foreground so that when it is stopped by the user, it returns control back to main.py to finish the shutdown process. In the ideal architecture the message router process only forwards messages between processes and handles system level operations, like shutting down the backend. However the inclusion of the web server forces the message router process to also handle all of the message management for the web server.

A key feature of Flask's development server is the ability to handle multiple HTTP requests concurrently, which is necessary to control multiple pieces of hardware at the same time. It does this by creating a new thread to handle each new HTTP request, which in turn uses the message router to send messages to a specific process. In the ideal architecture, this creates a problem where there is no way to tell what thread a response to a message is destined for. In order to make the web server work, a data structure called a job queue is required.

4.2.1 Handling web server messages

Figure 4.1 shows an example of the message router process receiving a HTTP request, sending a command to the light engines process, getting a response on the execution status of the command from the light engines process, retrieving the response from the job queue and turning the command response into a response to the original HTTP request. The job queue is a table that relates a specific message to a specific thread. By utilizing meta data that is included in every message, which will be covered in detail in 5.2, the message router can save the response to a command message to the job queue and signal to the thread that originally sent the message that a response has been received. For the example detailed in Figures 4.1 and 4.2 the entire transaction is performed as follows:

1. Flask's web server receives an HTTP request and creates a thread to handle servicing and responding to the request. This thread creates the appropriate message for the request and uses the message router to send the message. The message router adds

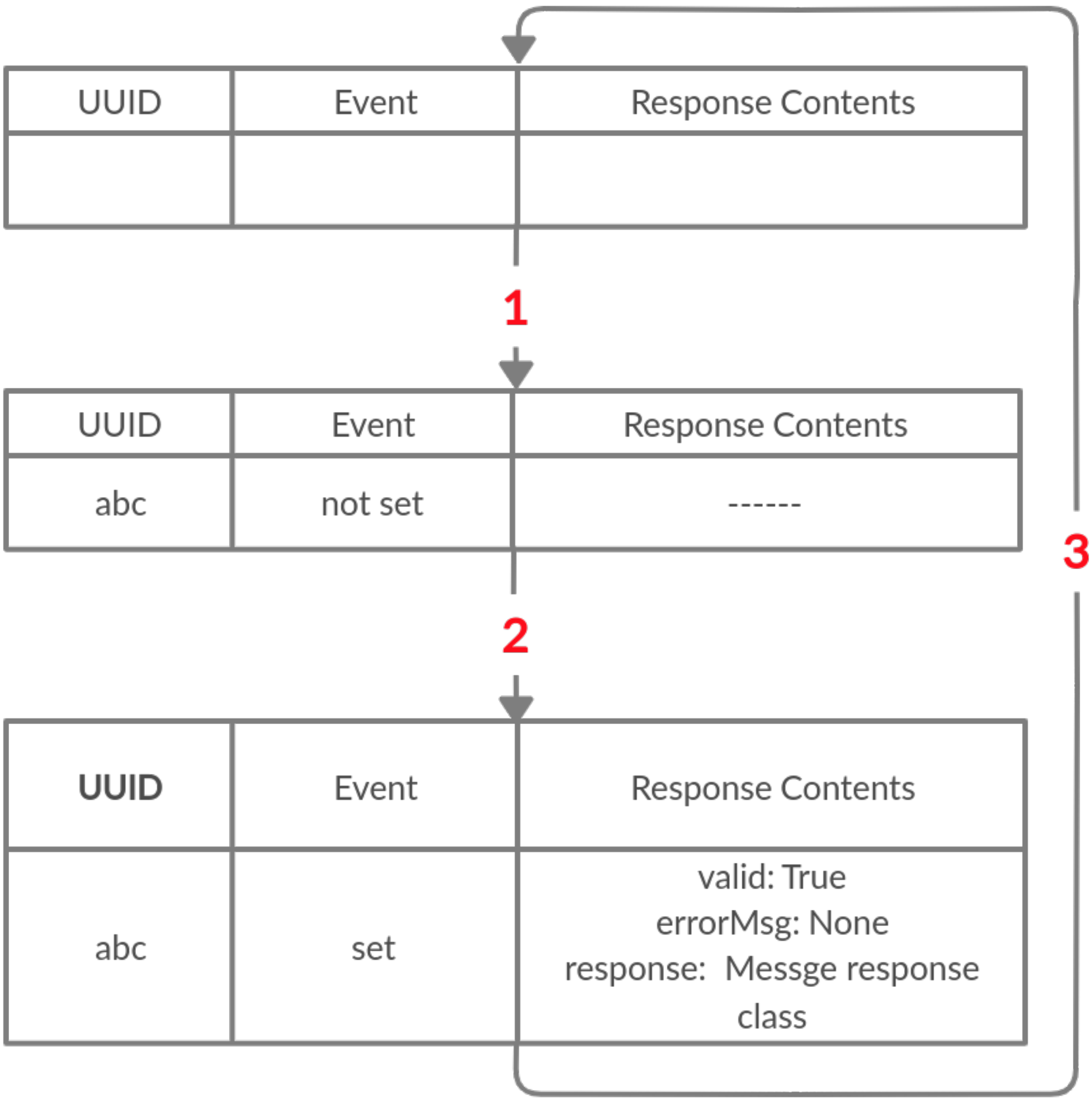

Figure 4.2: State of the job queue for servicing an API call in Figure 4.1. See sections 4.2.1 and 4.2.2 for further explanation.

an entry to the job queue with the message's universal unique id (UUID) and creates an event object that the thread can stall on to avoid using any CPU resources while waiting for a response to the message to come back from the light engines process.

2. When the thread that monitors incoming messages from the light engines process receives a message, it checks the job queue to see if there is an entry that shares the

same UUID. If there is a match, then the incoming message is the light engine processes response to the original message. The thread then saves this response message to that entry in the job queue and sets the event object.

3. When the event is triggered, the API request handler thread wakes up, gets the response message from the job queue and deletes the job queue entry. It then processes the response message so that it can be sent out as part of the HTTP response.

4.2.2 Forwarding messages

If the web server was in its own process then the job of keeping track of responses to threads could be moved to the web server process, allowing the message router to only focus on forwarding messages. Figure 4.3 shows how the message router threads handle forwarding messages between processes with the common task of the print job controller controlling a light engine during a print job. If the print job controller were to send a command to the light engine process, it would go as follows:

1. The print job controller sends a message to the message router where the message is collected by the print job controller's incoming message handler.

2. The handler looks at the meta data of the message to determine which process it is intended for, in this case the light engines process, and then forwards it to that process.

3. After the light engines process is done executing the command indicated by the message, it responds by sending a command status message to the message router process, which is intercepted by the light engines's incoming message handler.

4. Finally the handler forwards the status message back to the print job controller, where it moves on to the next step of its algorithm.

Overall the message router process is a convenient center point for the backend and with some further research it could be simplified to better embody the design principles of modularity, ease of use and reliability.

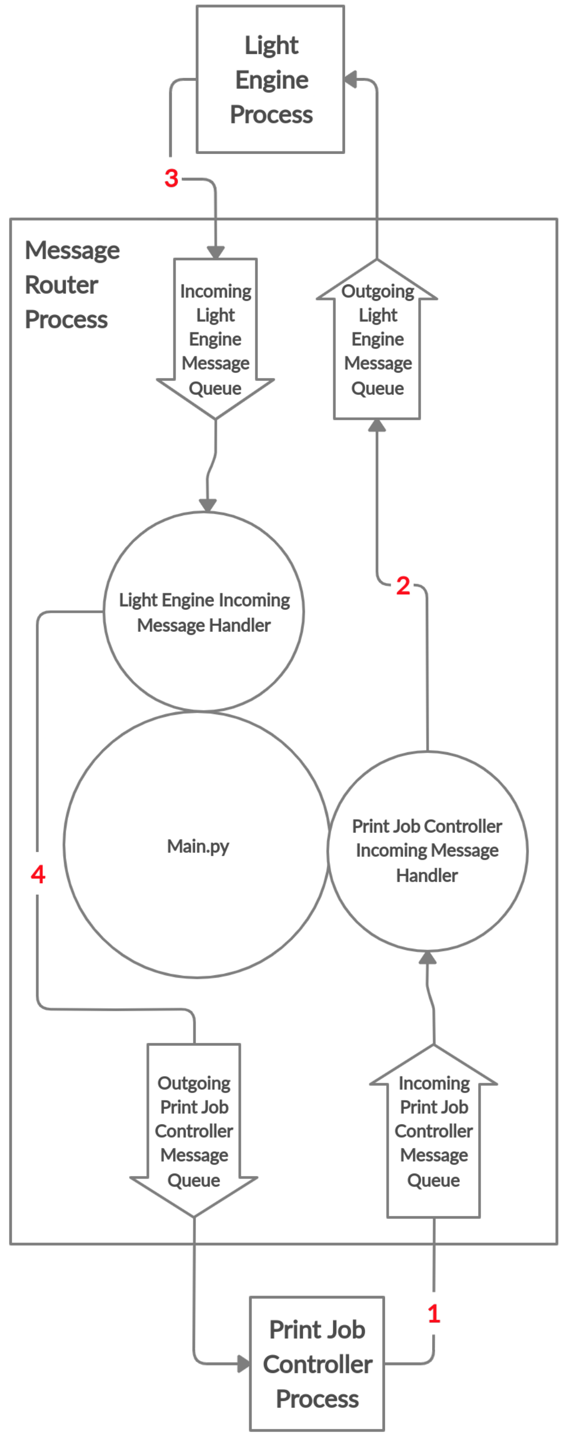

Figure 4.3: Threads used in the message router process to forward messages from one process to another. Threads that are directly adjacent to each other have a parent-child relationship to each other, with the thread with the largest circle being the parent. Numbered connections relate to the explanation in section 4.2.2.

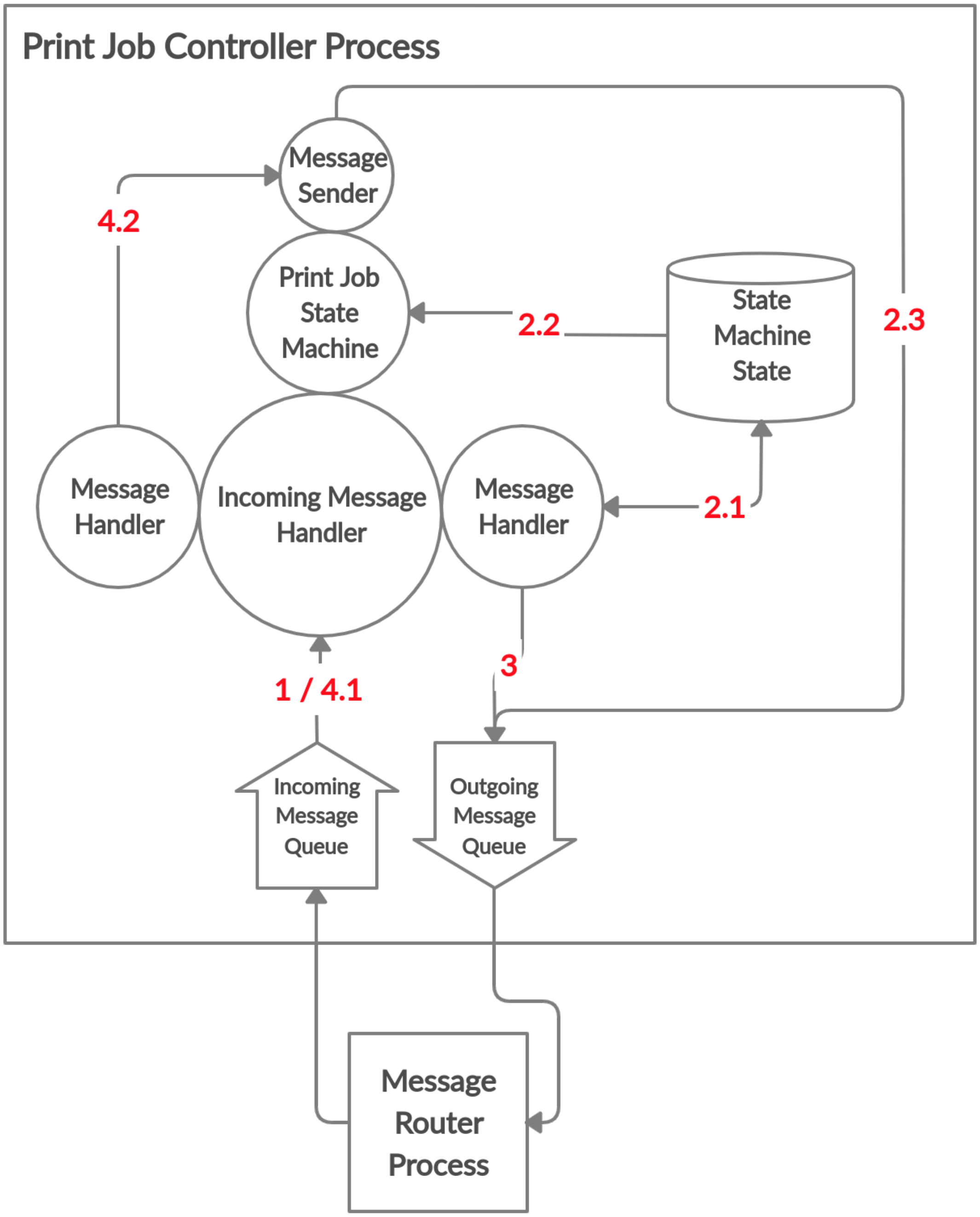

Figure 4.4: Thread architecture for the print job controller process. Refer to the text for further explanation. Numbered connections relate to the explanation in section 4.3.

In order for a process to be able to talk to the message router it requires some infrastructure. This infrastructure is largely pre-built and is provided by an abstract base class that all of the processes share. Aside from setting up the messaging interface, the structure of the communication threads for each process remain separate from any threads that the rest of the process uses to do its job. While the hardware processes share very similar architectures, software oriented processes, like the print job controller, can have a wide variety of architectures and as such this section focuses on explaining the thread architecture of the print job controller process.

When the print job controller process is first started the initial thread creates a state machine thread, before transitioning to the incoming message handler. For every new message that the handler receives, it creates a new thread to handle the message, much like the web server. Figure 4.4 provides an example of all the ways that messages sent to the print job controller process can be handled:

1. A message is received from the message router and a message handler is created for it.
2. If the message is a command that changes the state of the print job controller several things happen:
    - 2.1. The message handler updates the state of the state machine.
    - 2.2. The state machine thread monitors the state machine variable for changes. When the state changes it starts executing the code for the new state.
    - 2.3. If part of the state's execution includes sending a message to another process, then the state machine thread will create a new thread to send the message. This allows for the print job to continue executing without having to wait for a response message. To handle correlating response messages to the threads that they originated from, the print job controller contains a job queue of its own.
3. After the message handler has finished processing the message it creates and sends a response message to the message router.
4. 4.1. If the incoming message is a command status message, a new message handler thread is created.

4.2. The message handler updates the job queue and sets the event object for the message sender thread, which frees the message sender thread.

Like the web server, the job queue is an integral part of the print job controller. If the web server were to be put into a separate process, it would have a similar architecture to the print job controller process. In general, any process that requires the ability to send messages other than as responses to other processes, will require a job queue.

4.4 Hardware processes

Compared to software based processes, processes that control hardware are simple enough to not need a job queue. This is due to hardware processes being built to be slaves to other processes, like the print job controller, or by manual control by a user through the frontend. Figure 4.5 shows an example of how the light engine process handles messages:

1. The light engines process receives the incoming message and creates a message handler thread.

2. If the message is intended to be processed by a light engine, the message handler uses the information in the message to determine which driver to execute the message with. Otherwise if the message was not intended for a specific light engine, like a query to find out what drivers are loaded, then the message handler thread handles processing the message and then creates and sends a command status message.

3. When the driver finishes, the message handler returns the result of the operation in a command status message and uses the meta data of the original message to determine what process the command status message should be sent to. The message is then sent to the message router. Conversely, if the message was not intended for a specific light engine, like a query to find out what drivers are loaded, then the message handler thread handles processing the message and creates and sends a response.

An important feature that this architecture enables is the ability to have multiple pieces of the same hardware connected to the system software and independently controllable at the same time. This is necessary given how many motorized stages current models of the 3D printer utilize. However supporting this feature creates a problem that afflicts the axes process specifically.

While the thread architecture of the axes process is identical to the light engines process, the way that the drivers are created differ. With light engines, one driver can control only one light engine at a time. With axes, motorized stage controllers frequently are used to control multiple stages at the same time. This makes it awkward to create an API for the axes, as researchers care about controlling individual axes and not having to worry or know about how the shared piece of hardware the axes are connected to operates.

As shown in Figure 4.6, to solve this problem an added layer of abstraction has been created. Each axis connected to the stage controller has its own axis shim object which provides a unified API interface that all axes share. The responsibility of the axis shim is to act as a translation layer between the axis API interface and the stage controller driver, with each axis shim only being capable of using the driver to control a single axis. The specifics of this architecture will be covered in further detail in section 5.3.2.

4.5 File system usage

A limitation of the messages used to communicate between processes is the difficulty of sending, receiving and sharing files between processes, due to restrictions in the data types that the queue objects that the processes use to communicate with each other. To remedy this, the file system is used to provide permanent storage for files, allowing for messages to send the path to a file instead of the entire file in order to share the file.

However making files visible to the frontend, especially, for example, a web site that wants to show what the current image a light engine is displaying, means that these files also need to be accessible to the web server. This is complicated by a security feature known as cross-origin resource sharing (CORS). CORS, when enabled, allows for a web server to share restricted resources outside of the domain from where the original resource was served, in this case the web-based frontend. When disabled, only files that are located in the web server's static files folder are accessible, and trying to access other files on the web server's file system will result in a CORS error [13]. Many web servers and web applications disable

CORS by default as it can result in arbitrary file access on the web server if configured incorrectly.

As is shown in Figure 4.7, to avoid frustrating researchers with CORS issues in their code, the shared file location for images, print job files and other yet unknown file types that can be uploaded to the server are saved to the web server's static files folder, thus allowing CORS to remain disabled. It also provides ready access to users to all uploaded files and in the future could easily enable frontend based file management interfaces.

As of the current version of the backend, uploaded print job files and manually set light engine images are saved to the web server's static files folder under the same file name, overwriting the previous file. Until an API is created for managing individual files in the static files folder, this approach keeps the Raspberry Pi from running out of disk storage as new files are continuously uploaded to it.

4.6 Summary

Together the simplified and detailed architectures provide a low and medium detailed view of the system software, which includes the terminology specific to the system software and also the basics of how each component functions. While numerous considerations have been made about how to architect the software to keep it modular, easy to use and resilient, up to this point the philosophy of create, test and register has been conspicuously absent. Moving into chapters 5 and 6 the implementation details of this philosophy will be presented in high levels of detail as the code is finally discussed directly, along with patterns to streamline the development process.

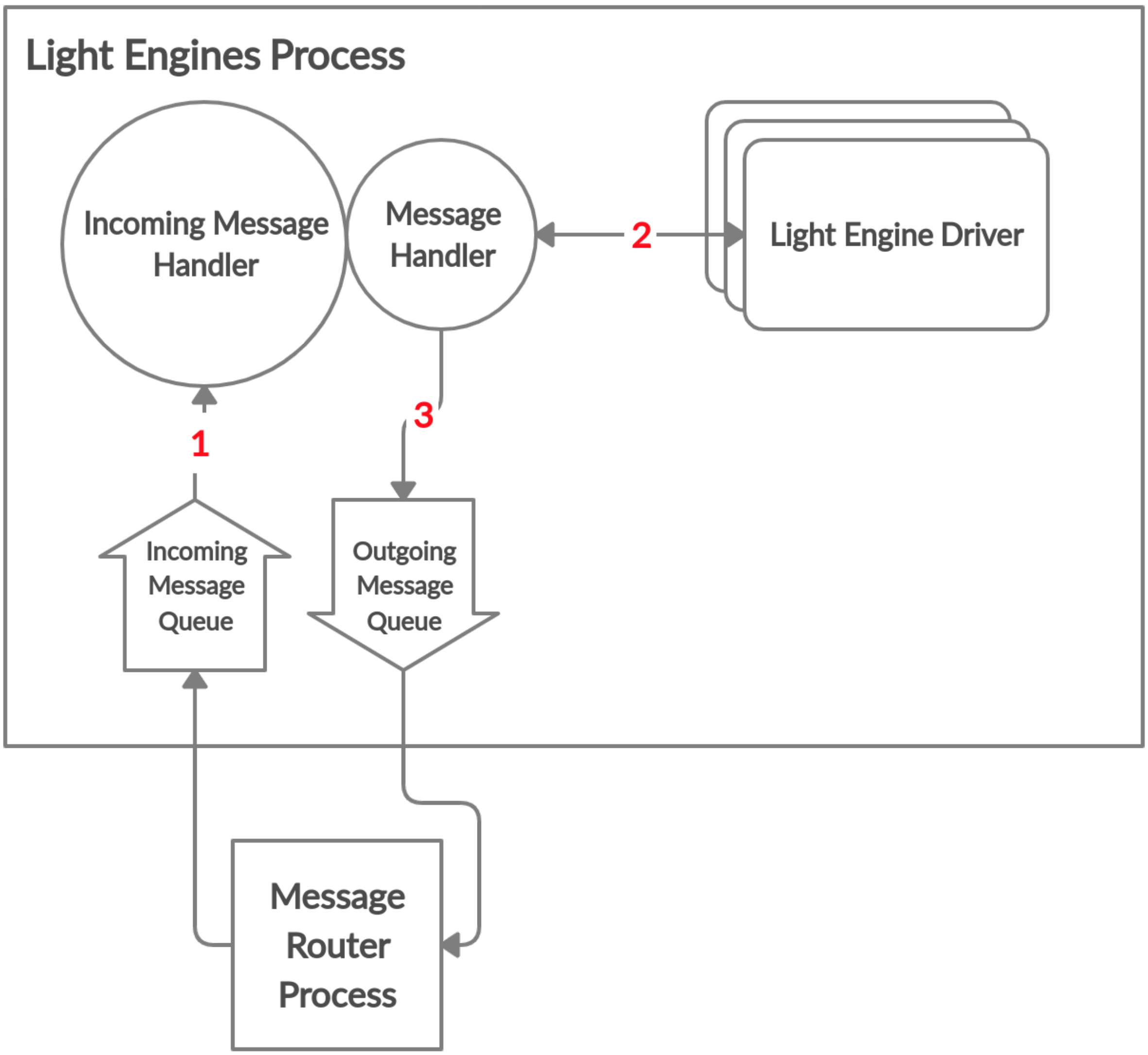

Figure 4.5: Thread architecture for the light engines process. Numbered connections relate to the explanation in section 4.4.

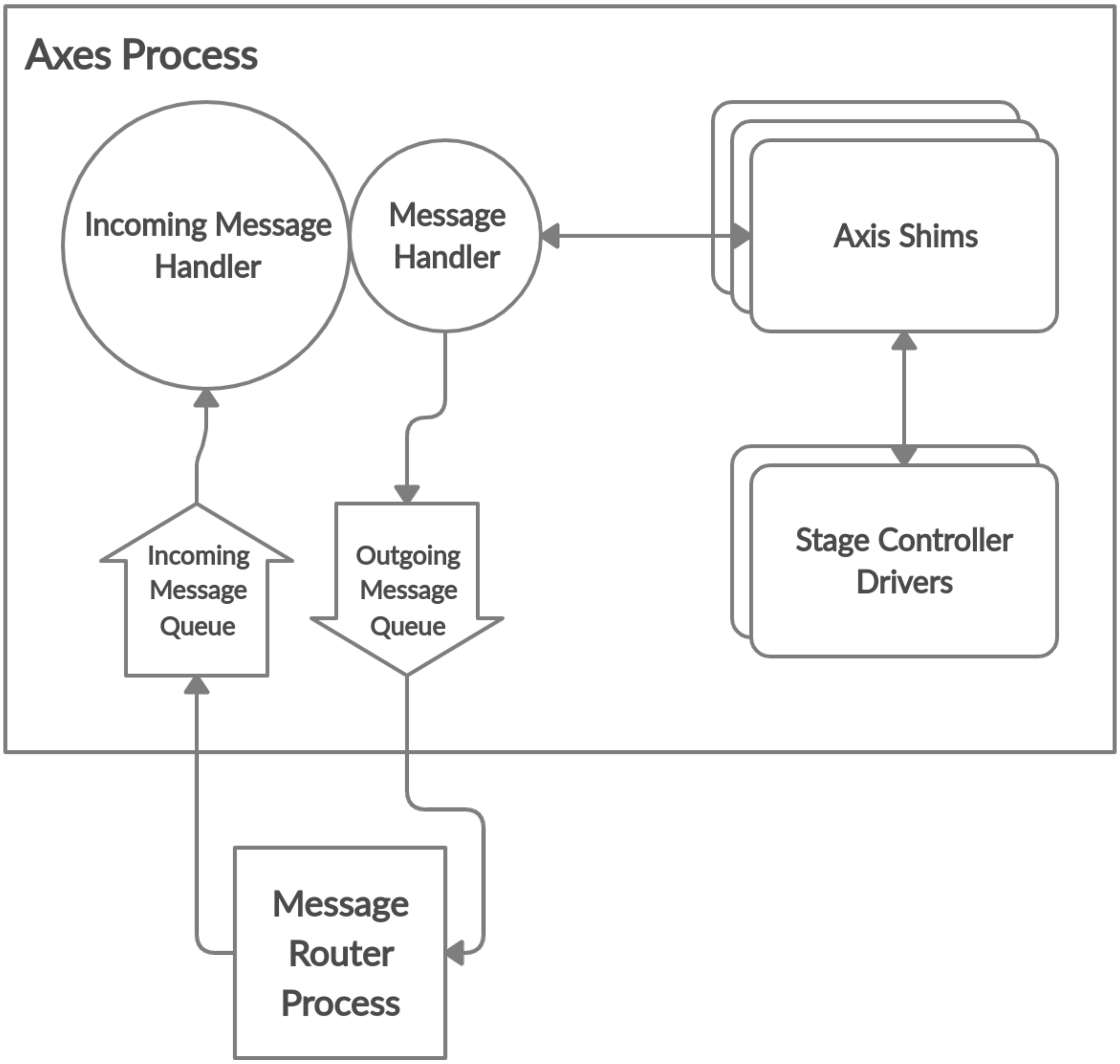

Figure 4.6: Thread architecture for the axes process with the stage controller driver and the axes interface separated from each other.

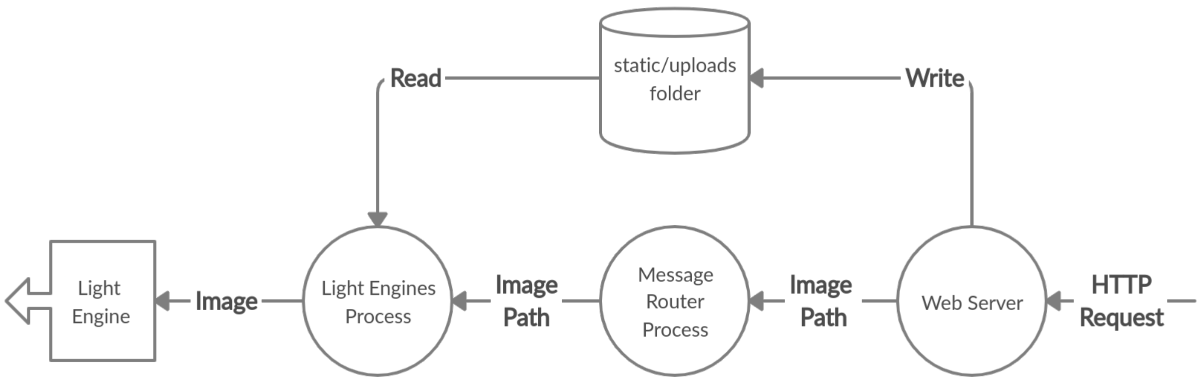

Figure 4.7: Model of how the output image on a light engine is set from the web server.

CHAPTER 5. THE STRUCTURE OF THE CODE BASE

The transformation of an architecture from a series of flow charts to code is drastic, with the resulting collection of file and folder names often bearing little resemblance to the original architecture. To make things more difficult, as time goes on and technologies change and more people work on the code base, the code structure will experience entropy as it ages.

While not a perfect solution, the patterns established by the create, test and register development philosophy can slow down or contain degradation of the code base. This is done by using the object oriented programming concept of polymorphism to establish well defined interfaces for different portions of the system software. All of the processes in the backend of the system software use a series of abstract base classes to signal the expectations of what the code should look like to researchers working in the code. For this reason, the primarily focus of this chapter will be the abstract base classes in the system software and the structure of files that use abstract base classes from other libraries. Additionally portions of the code that are intended for registering new code to the system software will also be addressed.

To avoid copy and pasting files that contain hundreds of lines of code into this chapter, code examples have been simplified and abbreviated where possible. The full text for each file mentioned have been included in the appendix (chapter A). That being said, it is expected that the reader can read and understand Python.

It is also worth noting that all of the different components of the system software are heavily interconnected with each other and that at times some aspects of it will be referenced with little to no explanation, only to be explained later in its own dedicated section. This may cause a sense of confusion during an initial reading. Given the complexity of the topic, this is unavoidable and it will likely take multiple readings to fully understand how all of the pieces of code interact with each other.

5.1 main.py

As was discussed in 4.1, main.py is the first and last thread to execute in the system software and it handles the creation of all of the other processes. Because of this, after a new process as been created and tested, it needs to be registered as part of the system software in main.py.

Before looking at the contents of main.py, it will be useful for the rest of the chapter to discuss the basic folder structure of the code base. The top level directory is structured as follows:

```
system_software
├── config_files
├── src
│   └── main.py
└── test
```

Of all of the folders, src is the only python package, and it contains all of the source code for the system software, including main.py. The src directory also acts as the common root for all python modules, which effects how modules import each other. For example, if a file needs to import the ABC_Interface module, it would be done by starting at src and traversing down the directory structure to the package where the ABC_Interface file resides, like so:

```
from src.process_interfaces import ABC_Interface
```

This pattern is used extensively throughout the code base, and provides a convenient way to find files and give programmers context to what the purpose of the module is based on the package it belongs to. Additionally, all of the abstract base classes include the prefix ABC_ in their name to make them easier to find as one of the best ways to learn how a package is intended to function is to first read through any abstract base classes it contains.

The test directory contains all of the test code for the system software. Like src, it also contains python packages, however it is not a package itself. Finally the config_files directory contains all of the configuration files that have been created for the system software.

The following is an example of main.py with only the message router and light engines processes registered to run:

```
###############################################
```

```
# main.py
#############################################
# configuration imports
from src.config import ConfigManager
from src.data_structs import ConfigInterfaces # enum

# process interfaces imports
from src.process_interfaces.hardware import LightEnginesInterface
from src.process_interfaces.controllers import MessageRouter
from src.webserver import setup, run

# process messages imports
from src.data_structs.internal_messages.hardware import ABC_LightEngineMessage
from src.data_structs.internal_messages.controllers import ABC_RouterMessage
from src.data_structs.internal_messages import Shutdown

# multiprocessing imports
from multiprocessing import Process, Queue
from threading import Thread


defaultConfigPath = "/path/to/config/file/config.json"
cm = ConfigManager(defaultConfigPath) # config manager

# input and output messages queues for each process
inQs = {}
outQs = {}


# create the light engines process
```

```
# create input and output queues for the process
lightEnginesQueueIn = Queue()
lightEnginesQueueOut = Queue()
# create the light engine process interface class
leif = LightEnginesInterface(lightEnginesQueueIn, lightEnginesQueueOut)
LightEnginesConfig = cm.getConfig(ConfigInterfaces.LightEngines)
leifProc = Process(
    target=leif.run, # function to run as process
    kwargs=LightEnginesConfig.getArguments(), # arguments for target
    name=ABC_LightEngineMessage.destination,
)
leifProc.start()
# add light engine message queues to global list of message queues
inQs["light_engines"] = lightEnginesQueueOut
outQs["light_engines"] = lightEnginesQueueIn


"""Create and register additional processes here"""


# create the message router process and start the web server
serverConfig, debug = cm.getConfig(ConfigInterfaces.Router).
    ↪ getArguments()
# pass in the global list of message queues
router = MessageRouter(inQs, outQs)
# start the message routing threads
Thread(
    target=router.run,
    kwargs={"configManager": cm, "debug": debug},
    name=ABC_RouterMessage.destination,
).start()
# setup and run Flask's web server
```

```
setup(router, serverConfig, cm)
run()
# when the web server is shut down by the user, have the message router
    ↪ shutdown the backend
router.shutdown()
```

To summarize, main.py does the following in the following order:

1. Create the ConfigManager object by giving it the path to the configuration file. It handles retrieving and validating configuration data from the configuration file and will be discussed further in section 5.5.

2. Create the inQs and outQs dictionaries to store all of the processes message queues in. Ultimately these variables are used by the MessageRouter to correctly route messages between processes. Aside from initializing and starting a new process, registering a new process is as simple as adding the new process's message queues to these dictionaries.

3. Create the light engines process. This includes creating the light engine message queues, initializing the LightEnginesInterface, calling its run method as a process and registering the message queues. The LightEnginesInterface will be the topic of further discussion in section 5.3.1.

4. Create the MessageRouter and run its run method as a thread instead of a process, as the current process is the message router process.

5. Lastly, setup Flask's web server and tell it to run, which gives control of the console to the web server. When the user stops the web server, the MessageRouter handles shutting itself and the LightEnginesProcess down.

A behavior that isn't clear by the simplified code above is that if there is no configuration information in the configuration file for a process, then main.py can either not start the process, start the process but disable its API in the web server or start the process with a default configuration. This gives flexibility to how many computing resources the backend is using while also keeping the frontend from sending invalid messages through the MessageRouter.

5.2 Messages

The message classes are arguably the most important classes in the entire system software. They determine what tasks a process can and cannot do and ultimately provide the interface that the web server API conforms to. Understanding how the message classes are formatted is critical to understanding the rest of the code in the backend.

All of the message classes reside in the following locations:

```
system_software
└── src
    └── data_structs
        └── internal_messages
            ├── controllers
            │   ├── print_job_messages.py
            │   └── router_messages.py
            ├── hardware
            │   ├── light_engine_messages.py
            │   └── axes_messages.py
            ├── ABC_Message.py
            └── system_messages.py
```

Internal messages are divided into two categories: controllers and hardware. Controllers refer to processes that perform purely software based tasks, like the print job controller, while hardware refers to processes that manage pieces of hardware. This is an organizational pattern that is used throughout the code base to help keep code that uses similar patterns grouped together. While this may seem minor and even unnecessary, researchers want to make use of preexisting code where possible and this abstraction makes it easier to know where to look for preexisting code.

Notably the ABC_Message class is not part of either the controller or hardware packages. The message classes are organized in a tiered system, with ABC_Message being inherited by another abstract base class that is specific for each process, and then all of the messages for that process inherit from that process's abstract base class. For example, if a message called LightEngineImage were to be created, it would have the following inheritance hierarchy:

```
##############################################
# ABC_Message.py
##############################################
```

```
import uuid

class ABC_Message:
    """
    Parent class for all messages.

    Child classes are intended to be initialized with all of the information that the message
    needs to have. All attributes of the class should be class properties.

    Attributes:
      uuid - unique id for the message
      type - customizable param for specifying the message type
      sender - process that originally sent the process
      destination - process the message is intended for
    """

    _uuid = uuid.uuid4().hex
    _type = None
    _sender = None
    _destination = None

    def __init__(self):
        """
        Creates a uuid for the message
        """
        self.uuid = uuid.uuid4().hex

    """Attribute getters and setters here"""


```

```python
###############################################
# light_engine_message.py
###############################################
from src.data_structs.internal_messages import ABC_Message
# enums
from src.data_structs.enums import MessageType, PublisherType

# inherits from ABC_Message
class ABC_LightEngineMessage(ABC_Message):
"""
    Light Engine specific message parent class

    Attributes:
        light_engine (str) - name of the light engine that the message is intended for
        destination (str) - hard coded value to indicate what process the CommandStatus message needs to be sent to. Also is used as the key for the message queue dictionaries that the MessageRouter uses.
    """

    _light_engine = None
    destination = "light_engines"

    def __init__(self):
        super().__init__()

    """Attribute getters and setters here"""


# inherits from ABC_LightEngineMessage
class LightEngineImage(ABC_LightEngineMessage):
```

```
    """
    Getter/Setter for the image of a light engine.

    Attributes:
        publisherType (PublisherType) - used with getters to determine how the driver should handle
                returning the data
        image (str) - path to where the image is saved on the disk
        type (MessageType) - is the message a setter or a getter
    """

    def __init__(
        self, light_engine, publisherType=PublisherType.none, set=False, image=None
    ):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        if isinstance(publisherType, PublisherType):
            self.publisherType = publisherType
        else:
            raise ValueError("publisherType must be a PublisherType enum")
        self.light_engine = light_engine
        self.image = image
        if set and (image is None):
            raise ValueError("Image cannot be set without valid image value")
```

To expand the example, if a LightEngineImage message is sent to the light engines process by the web server, the LightEnginesInterface would respond with a CommandStatus message after the LightEngineImage message was processed. In the light engine process, the sender and destination fields in the LightEngineImage message would be swapped for the

CommandStatus before sending it back to the web server along with the results from the command. The code for the CommandStatus message is the following:

```
#############################################
# system_messages.py
#############################################
from src.data_structs import ErrorState
from src.data_structs.internal_messages import ABC_Message

class CommandStatus(ABC_Message):
    """
    Execution status of a command recieved either from the API or another process

    Atrributes:
        state (ErrorState) - error code for the command
        traceback (str) - if state is ErrorState.error, then the stack trace is placed here.
        errorMsg (str) - if state is ErrorState.error, then the error string is placed here.
    """

    _returnVal = None
    _state = ErrorState.none
    _errorMsg = ""
    _traceback = ""

    def __init__(
        self,
        uuid,
        destination,
        returnVal=None,
        errorState=ErrorState.none,
```

```
        errorMsg="",
        traceback="",
    ):
        """
        Creates a command status.

        Parameters:
            uuid (uuid.hex) - unique id of the original message that this object is responding to.
            destination (str) - name of the process that the message is going to
            returnVal (any) - any values that are returned by the function
            errorState (ErrorState) - error code that resulted from the command
            errorMsg (str) - error message
            traceback (str) - traceback of the error
        """
        super().__init__()
        self.type = "status"
        self.uuid = uuid
        self.returnVal = returnVal
        self.state = errorState
        self.destination = destination
        self.errorMsg = errorMsg
        self.traceback = traceback

    """Attribute getters and setters here"""
```

These patterns are repeated and customized for all of the processes, with each process having its own file of message classes, such as axes_messages.py for the axes process. The only exception to this is the system_messages.py file, which includes messages that are intended to be created and consumed by multiple processes.

5.3 Process Interfaces

As was discussed in chapter 4, all processes use the same or, in the case of the message router process, very similar thread architectures for handling messages. To facilitate this, all of the processes inherit from a shared ABC_Interface class which provides all of the necessary infrastructure and functionality for interprocess communication. All of the process interfaces are grouped into a process_interfaces package as follows:

```
system_software
└─ src
   └─ process_interfaces
      ├─ controllers
      │  ├─ MessageRouter.py
      │  └─ PrintJobController.py
      ├─ hardware
      │  ├─ AxesInterface.py
      │  └─ LightEnginesInterface.py
      └─ ABC_Interface.py
```

Of all of the process interfaces, the MessageRouter is the most unique and requires several qualifiers. First, the MessageRouter is considered a controller, even though it is the core of entire system software and is not intended to be modular in the same way that other processes are. Second, is that the MessageRouter also inherits from ABC_Interface, however it overrides many of the ABC_Interface's methods to add in message forwarding capabilities alongside the logic for handling the web server.

Finally the overriding of the ABC_Interface methods that MessageRouter performs is unique to it and modifications to the methods in the ABC_Interface should not need to be done for the majority of programming tasks in the system software. Further usage of the ABC_Interface class is best understood by looking at examples in the code. Below is an example of how the LightEnginesInterface implements the ABC_Interface:

5.3.1 Light Engines

```
#############################################
# LightEnginesInterface.py
#############################################

import src.hardware.light_engines as drivers
import traceback
from src.data_structs.internal_messages import CommandStatus, Shutdown
from src.process_interfaces import ABC_Interface
from src.data_structs import ErrorState
from src.data_structs.internal_messages.hardware import (
    LightEnginesNames,
    LightEngineInitialize,
    """Other light engine messages here"""
)
from src.data_structs import MessageType


class LightEnginesInterface(ABC_Interface):
    """
    Interface for the process that controls all hardware light engines.
    Documentation for undocumented functions can be found inside the Interface abstract base class.
    Attributes:
        light_engines (dict): dictionary of all of the light engine classes. The keys are the name of the light engine and the
                    values are the light engine object.
    """

    light_engines = {} # currently loaded light engine drivers
    # where new drivers are registered
    valid_sub_configs = ["DummyDriver", "I2CLightEngine"]
```

```python

    def __init__(self, in_queue, out_queue):
        """
        Sets the input and output queues
        Parameters:
            in_queue (Queue): input queue from the flask process
            out_queue (Queue): output queue from the flask process
        """
        super().__init__(in_queue, out_queue)

    def setupLightEngines(self, light_engine_drivers=[]):
        """
        Initializes all of the light engine and driver objects for the configuration
        specified in the config file.
        All light engine objects will be stored in self.light_engines.
        Parameters:
            light_engine_drivers (dict): passed in configuration of the light engine
        """

        for driverConfig in light_engine_drivers:
            # get the light_engine_drivers class object
            module = getattr(drivers, driverConfig.getClassName())
            # use the driver to create the light_engines object
            initParams = driverConfig.getArguments()
            self.light_engines[driverConfig.getName()] = module(**initParams)

    def run(self, light_engine_drivers=[], debug=False):
        """
        Starting point for the LightEnginesInterface Process.
```

```
        Parameters:
            light_engine_drivers (list) - configuration options for each driver and the light engines that are
                attached to it. Each item in the list should be a dictionary with the config
                params of a driver, with one of the keys containing a list of all the config
                params for all the light engines that will be using the driver. See docs/Config_Files.md
                for more details.
        """
        self.setupLightEngines(light_engine_drivers)
        self.processMessages()

    def messageLogic(self, payload):
        try:
            # provides the names of all of the light engine drivers that are initialized
            if isinstance(payload, LightEnginesNames):
                self.outq.put(
                    CommandStatus(
                        payload.uuid,
                        payload.sender,
                        returnVal=list(self.light_engines.keys()),
                    )
                )

            elif isinstance(payload, LightEngineInitialize):
                if payload.type == MessageType.get:
                    self.sendResponseMessage(
                        payload.uuid,
                        payload.sender,
```

```
                        # function to execute and then send the result of with the CommandStatus
                        self.light_engines[payload.light_engine].get_initialized,
                    )
                else:
                    self.sendResponseMessage(
                        payload.uuid,
                        payload.sender,
                        self.light_engines[payload.light_engine].set_initialized,
                    )

            """Add other message handlers here"""

        except Exception as e:
            self.outq.put(
                CommandStatus(
                    payload.uuid,
                    payload.sender,
                    errorState=ErrorState.error,
                    errorMsg="{}: {}".format(type(e).__name__, e.args),
                    traceback=traceback.print_exc(),
                )
            )

    def shutdown(self):
        super().shutdown()
        print("Light Engines Interface Shutdown")
```

Any child class of ABC_Interface is required to define four methods:

- __init__ - initializes the __init__ method of the ABC_Interface parent class sets up the message handling threads.
- run - handles starting all of the threads that make the process function. Importantly for hardware processes, this is where drivers should be created as allocation of hardware resources, like a serial connection, are often tied to a specific process by the operating system, and if those resources are created before the run method is called in a new process, then the drivers may not be able to use those hardware resources. For this reason the setupLightEngines method is called inside of the run method instead of in __init__.
- messageLogic - defines the handler logic for each of the messages. Methods like sendResponseMessage and self.outq.put belong to ABC_Interface and are used to send CommandStatus messages.
- shutdown - handles all the necessary steps to make sure that the process shuts down cleanly.

For controller process interfaces, once they define the appropriate methods there are no further guidelines for how to structure the process. However hardware process interfaces are expected to forward messages to the appropriate driver and patterns exist for how that structure is formatted. For the LightEnginesInterface this is handled by the light_engines dictionary, the valid_sub_configs array and the ABC_LightEngineDriver.py file.

The light_engines dictionary stores all of the currently initialized drivers as a key-value pair, with the key being a user assigned name that is defined in the configuration file and passed into LightEnginesInterface by main.py, and the value is a light engine driver object. Messages that are intended for a specific light engine all have a name field that needs to match one of the keys in light_engines to be valid. The names of all of the valid light engine names can be retrieved by sending a LightEnginesName message to the process and the LightEnginesInterface will return an array of all of the valid names.

The valid_sub_configs is a list of all of the drivers that are registered with the light engines process and, by extent, the entire system software. So long as the driver is registered in the src.hardware.light_engines package's ___init___.py file, python can use reflection to translate the driver class's name into an actual driver object.

All of this infrastructure in LightEnginesInterface, and for all hardware process interfaces for that matter, only makes sense if all of the light engine drivers share a common API for handling all of the light engine specific messages. The solution is to create an abstract base class for light engine drivers that forces them to implement a common API. The message classes for the light engines process are largely a reflection of what is in the abstract base class. For the light engines process, all of the drivers must inherit from the following class:

```
############################################
# ABC_LightEngineDriver.py
############################################
import abc
from threading import Lock
from functools import wraps
from src.data_structs import Publisher, publisher, PublisherType


class ABC_LightEngineDriver(metaclass=abc.ABCMeta):
    """
    This class defines the minimum interface needed for a driver.
    The purpose of a driver is:
    1. Contain all of the objects necessary for direct communication with the hardware.
    2. Initializing communication with the hardware.
    3. Cleanly disconnecting and shutting down the hardware.
    4. Keeping track of the initialized state variable.
    5. Resetting the hardware driver
    Attributes:
        initialized (bool) - state variable to track if the software connected to the hardware
```

```
        state_lock (Lock) - for use when reading/writing state variables to keep the driver thread safe.
        image_path_publisher (Publisher) - handles when to publish get requests to the image path
    """

    _initialized = False
    _state_lock = Lock()

    @abc.abstractmethod
    def __init__(self, **kwargs):
        """
        Initializes the Light Engine object.
        Returns:
            none or error if invaild
        """
        # create the publihser for the image path.
        # the 1 indicates that for periodic requests it will return
        # every 1 second. This can be customized by the child classes
        self.image_path_publisher = Publisher(1)

    def get_image(self, publisherType):
        """
        Wrapper function for the _get_image method. Manages the publisher
        for the image variable.

        Parameters:
            publisherType (PublisherType) - used by publisher method

        Returns:
            return image as a png
```

```
        """
        return publisher(self.image_path_publisher, self._get_image,
            ↪ publisherType)

    @abc.abstractmethod
    def _get_image(self):
        """
        Getter for the image

        Returns:
            return image as a png
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_initialized(self):
        """
        Getter for the initialized state
        Returns:
            self._initialized (bool): is the hardware connected
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_initialized(self):
        """
        Setter for the initialized state
        """
        raise NotImplementedError

    @abc.abstractmethod
    def reset_driver(self):
```

```
        """
        Resets the state of the driver / hardware to a pre-initialized state
        """
        raise NotImplementedError

    # Additional abstract methods here
```

Creating abstract base classes for drivers is a delicate process. In the case of the ABC_LightEngineDriver class, it was created with a specific set of light engines in mind, however there is no guarantee that future light engines will fit this definition well, with the worst case scenario being the functionality of a new light engine being restricted by the abstract base class. This is painful because updates to the driver abstract base class often require minor changes to much of the backend, including the other light engine drivers. For this reason a less is more approach to driver abstract base class design is best.

In the case of the ABC_LightEngineDriver the only variables the class has are initialized and state_lock. initialized indicates if the driver has established communication with the light engine and state_lock is a mutex object for keeping the driver thread safe. state_lock is particularly important as the heavily multi-process and multi-threaded nature of the system software make thread safety a serious potential problem. From the researcher's perspective, avoiding race conditions is one of the biggest challenges to writing new drivers for the system software.

In contrast, one of the easiest parts of creating new drivers is adding in publish-subscribe fuctionality. A separate Publisher class has been created to handle managing pubisher requests, and a simple example of how it works can be seen from the code examples forABC_LightEngineDriver and LightEngineDummyDriver.

```
##############################################
# LightEngineDummyDriver.py
##############################################
from src.hardware.light_engines import ABC_LightEngineDriver
from src.data_structs import Publisher, publisher, PublisherType
```

```
# other imports here


class LightEngineDummyDriver(ABC_LightEngineDriver):
    """
    Dummy driver class to be used for testing purposes and as an example of what an actual driver
    class may look like. It only controls one light engine.
    Documentation for undocumented functions can be found inside the Driver abstract base class.
    """

    def __init__(self):
        # creates the publisher objects that are defined
        # in the abstract base class
        super().__init__()

    def _get_image(self):
        """
        Gets the image of the light engine
        """
        if not self.initialized:
            raise ValueError("cannot get image. Driver is not connected to the hardware.")
        return self.image_path

    def set_image(self, path):
        """
        Sets the image of the light engine

        Since the image is never loaded onto a light engine and the
        path to the so called image always stays the same, this
```

```
        function does nothing beside checking if it is valid to set the
            ↪ image

        """
        # image setting code

        self.image_path = path
        self.image_path_publisher.setChangePublish()

    def reset_driver(self):
        self.image_path_publisher.setChangePublish()
        # other variables reset here
```

The two main components in the example are the publisher method and the Publisher class. The Publisher class provides queues for subscriber requests for a specific variable and ensures that the requests are serviced in one of three ways: on change, immediately or periodically. In the ABC_LightEngineDriver example above, the Publisher variable image_path_publisher is used by the publisher method to delay when a call to the get_image method is returned. The method can either return immediately, wait until the image that is being displayed on the light engine is changed, or return on a periodic timer, in this case every second.

Integrating the Publisher class for any variable that is accessible by the API is a matter of creating a Publisher object for that variable inside of the abstract base class and creating a getter method for the variable that acts as a wrapper method for the actual getter method that will be defined by the child class. This wrapper method calls the publish method, which handles setting up the variable's Publisher object for that particular getter method call and calling the actual getter method. This emulates the behavior of a decorator function which unfortunately could not be used here as decorated abstract methods do not also decorate the child class's implementation.

Finally the variable's abstract getter method needs to include a variable for the PublisherType enum that is necessary for the decorator to function. These changes also necessitate adding a PublisherType field to the variable's message class and passing that variable into the appropriate message handler in the process interface. The web server's API handler will also need to be updated to complete providing the combined MVC and publisher-subscriber functionality that was discussed in 5.4.

5.3.2 Axes

Compared to the light engine process the axes process is almost identical except that the API for handling axis messages have been placed in a separate class alongside the driver. This difference can be seen in the structure of the hardware package:

```
system_software
└─ src
   └─ hardware
      ├─ axes
      │  ├─ drivers
      │  │  └─ ABC_AxisDriver.py
      │  └─ ABC_AxisShim.py
      └─ light_engines
         └─ ABC_LightEngineDriver.py
```

Computationally speaking, the shim handles all of the preprocessing necessary to create a message that the driver can then send to the stage controller. A shim then uses its driver to send that message to the stage controller and translates any responses into a format that can be sent back as part of a CommandStatus message. On top of handling all of the minutia surrounding communicating with the stage controller the driver also maintains state data that impacts all of the axes, such as if the driver is connected to the stage controller.

However shims can also have their own state data and determining if a state variable should be part of the shims or the driver varies depending on the hardware. For example, homing axes can be handled differently from controller to controller. Some controllers will home all of the axes at once, making the homed state variable ideal for the driver, while others allow for individual axes to be homed, necessitating that the homed variable be tied to a shim. These kinds of design problems are compounded by the fact that multiple shims can share a single driver.

One of these problems is how to enable the AxesInterface to create a set of shim objects that all share a single driver. Unlike the LightEngineInterface, the AxesInterface keeps track of the shims instead of the drivers. However letting a shim create the driver makes it difficult to share that driver with subsequent shims, meaning that it is simpler to create the driver first and then pass it in as a initializing parameter to the shims. But since it would be messy for the AxesInterface to juggle the shims and a driver at the same time, especially if there is any custom logic that needs to be ran during the initialization of the shims and drivers, we are left with a chicken and egg style problem.

To side step this problem, the ABC_AxisDriver has a static abstract method, meaning it can be run independent of a driver object, called createAxes that handles the creation of the driver and shims and returns only the shims back to the AxesInterface. Below is an example of how the AxesInterface uses the AxisDummyDriver's createAxes function to create dummy axes:

```
##############################################
# AxisDummyDriver.py
##############################################
from src.hardware.axes.drivers import ABC_AxisDriver
import src.hardware.axes as axes


class AxisDummyDriver(ABC_AxisDriver):
    """
    Dummy driver class to be used for testing purposes and as an example of what an actual driver
    class may look like. It only controls one axis.

    In this case, the homed state is part of the driver.

    Documentation for undocumented functions can be found inside the Driver abstract base class.
    """

    """Other driver methods here"""
```

```
    @staticmethod
    def createAxes(driverConfig={}, shims=[]):
        """
        Given configuration parameters, this function creates a
            ↪ properly configured driver
        and uses it to create properly configured axis objects.

        This function can be called without creating an object first.

        Parameters:
            driverConfig (dict) - kwargs for DummyDriver.__init__()
            shims (list of AxisShimConfig) - configs for all of the
                ↪ axis associated with this driver.

        Returns:
            output (dict) - dictionary of all the axis objects of the
                ↪ format {axisName: axisObject}
        """
        # create the driver
        driver = AxisDummyDriver(**driverConfig)
        output = {}
        # create the shims
        for axisConfig in shims:
            # verify the axix is compatible with the driver
            if axisConfig.getClassName() in AxisDummyDriver.validAxes:
                module = getattr(axes, axisConfig.getClassName())
                kwargs = axisConfig.getArguments()
                # add driver to input kwargs
                kwargs["driver"] = driver
                output[axisConfig.getName()] = module(**kwargs)

```

```
                else:
                    raise ValueError(
                        "The axis {} is not a valid axis to use with the
                          ↪ DummyDriver".format(
                            axisConfig.getClassName()
                        )
                    )
        return output


###############################################
#  AxesInterface.py
###############################################
from src.process_interfaces import ABC_Interface
import src.hardware.axes.drivers as drivers

class AxesInterface(ABC_Interface):
    """
    Interface for the process that controls all hardware axes.

    Documentation for undocumented functions can be found inside the
      ↪ Interface abstract base class.

    Attributes:
        axisShims (dict): dictionary of all of the axis classes. The
          ↪ keys are the name of the axis and the
                        values are the axis object.
    """

    def setupAxes(self, axisDrivers=[]):
        """
```

```
        Initializes all of the axis and driver objects for the configuration
        specified in the config file.

        All axis objects will be stored in self.axisShims.

        Parameters:
            axisDrivers (list of AxesDriverConfig): passed in configuration of the axis
        """
        # for each driver
        for driverConfig in axisDrivers:
            # get the driver class object
            module = getattr(drivers, driverConfig.getClassName())
            # use the driver to create the axes objects
            config, shims = driverConfig.getArguments()
            axes = module.createAxes(config, shims)
            # save them to AxesInterface.axes
            for name, obj in axes.items():
                self.axisShims[name] = obj
```

Overall, axes require more thought to implement compared to light engines. However it is a useful design pattern that feasibly could be reused for other types of hardware.

5.3.3 Print job controller and print job file validator

Compared to the light engine and axes processes, the print job controller does not make use of any abstract base classes beyond ABC_Interface as the extra infrastructure would only serve to complicate the print job controller. This approach means that if a 3D printer's hardware architecture changes significantly enough, such as adding an extra light engine, the best way to adapt the print job controller would be to write a new controller. While this may seem to violate the design goal of keeping the system software modular, print job controllers are functionally the most complicated pieces of code in the system software and modularizing them to be agnostic of future changes in hardware is non-trivial.

Part of what makes the print job controller complicated is its reliance on a state machine. While the state machine model does an excellent job describing a 3D printer's behavior during a print job in an environment where external messages can be received at any time, it is incredibly difficult to get an intuitive understanding for what the state machine looks like by only looking at the code. For this reason, figure 5.1 has been provided as a map to the print job controller code.

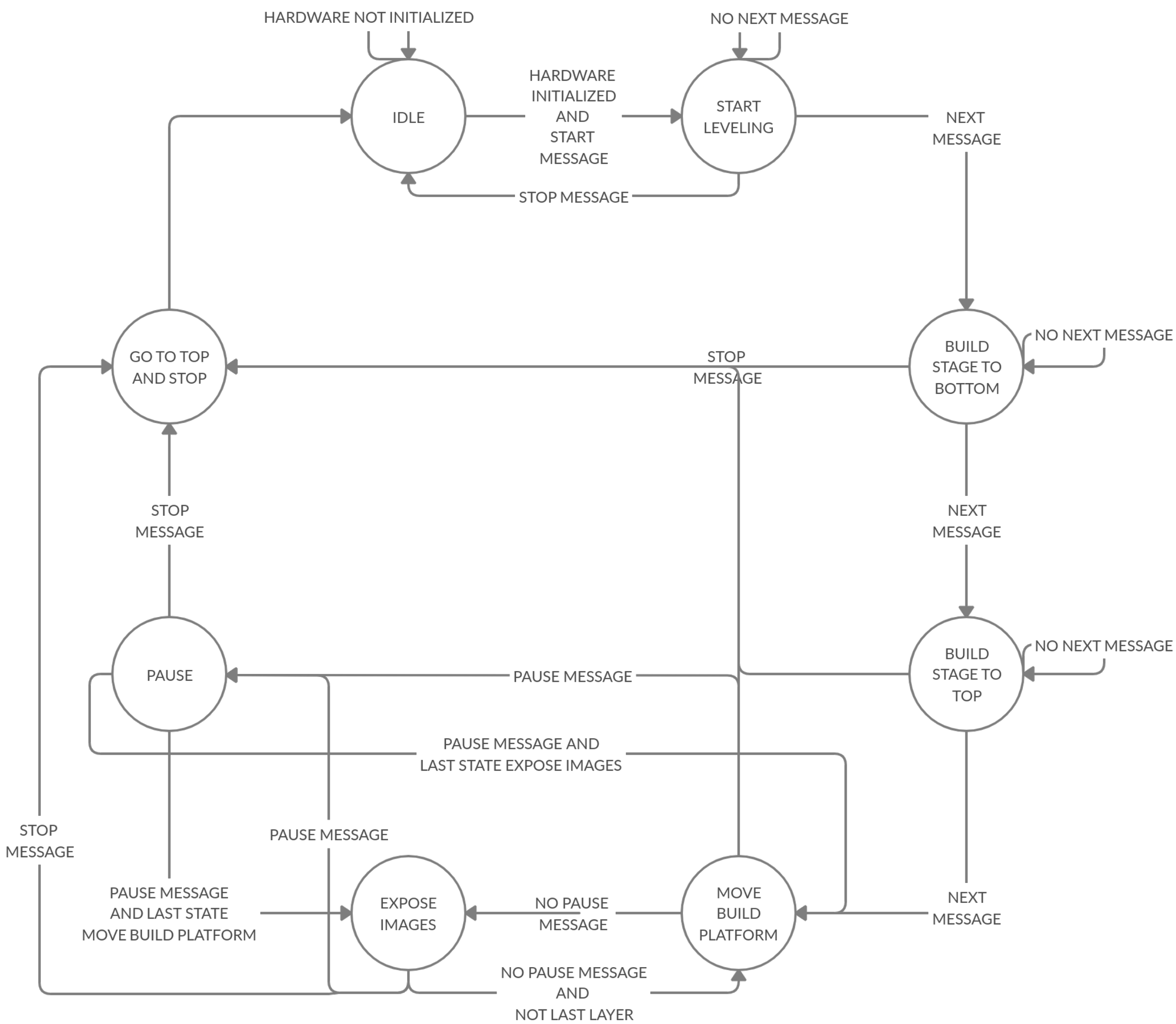

Figure 5.1: State machine for the print job controller process. External messages that can change the state and that can be received at any time are Start, Stop, Next, and Pause.

Integral to the functionality of the state machine is how the relevant information for each layer of a print gets to the expose images and move build platform states. Print job files are sent to the print job controller via an upload message and conform to the following file structure:

```
static/uploads
└─ print_job
   ├─ slices
   │  └─ 0000.png
   └─ print_settings.json
```

print_setting.json contains all of the information for the position of the build stage and the image to display on each layer, with the slices folder containing all of the images that make up the 3D print. To date, several versions of the print_settings.json file have been created and it is reasonable to assume that the format for print job files will continue to evolve as changes in 3D printer hardware architecture are made. To accommodate this, a separate printjob package was created to make the reading and validating of different print_settings.json files easier and more modular. The printjob package has its own code base and has been created to be installed with pip.

From the print job controller's perspective, using the printjob package is straightforward:

```
##############################################
# PrintJobController.py
##############################################
from printjob import getPrintJob
from src.process_interfaces import ABC_Interface

class PrintJobController(ABC_Interface):

    def handleStartMessage(self):
        """
        Handles state changes when a start message is received

        Passes errors back to the caller function
```

```
        """

        with self._stateLock:
            self.printjob = getPrintJob(
                self.printJobFilePath, printJobSettingsFileName="print_settings.json",
            )


    def stateMachine(self):
        """
        State machine thread that drives a print job
        """

        # other states

        elif state == State.move_bp:
            # stop if out of layers
            if self.currentLayerNum > self.printjob.getNumberOfLayers():
                with self._stateLock:
                    self.currentState = State.move_bp_top
                continue
            # move build platform for the next layer
            self.currentLayer = self.printjob.getLayer(self.currentLayerNum)
            self.currentLayerNum += 1
            self.updateBuildPlatformPosition()
            # update to the expose state
            with self._stateLock:
                # check if paused
                if self.currentState != State.pause:
```

```
                    self.currentState = State.expose
                else:
                    self.nextState = State.expose
        elif state == State.expose:
            # update state to finish the print job
            self.performExposures()
            with self._stateLock:
                # check if paused
                if self.currentState != State.pause:
                    self.currentState = State.move_bp
                else:
                    self.nextState = State.move_bp

    def updateBuildPlatformPosition(self):
        """
        Moves the build platform based on the config in the layer
        """
        # wait before moving bp
        time.sleep(self.currentLayer.init_wait)
        self.elapsedTime += self.currentLayer.init_wait
        # move up
        upDistance = self.currentLayer.distance_up
        self.moveAxis(upDistance, MoveMode.relative)
        # wait time at top
        time.sleep(self.currentLayer.up_wait)
        self.elapsedTime += self.currentLayer.up_wait
        # move to the thickness height
        downDistance = (
            self.currentLayer.thickness - upDistance
        )
        self.moveAxis(downDistance, MoveMode.relative)
        # wait before moving on
```

```
        time.sleep(self.currentLayer.final_wait)
        self.elapsedTime += self.currentLayer.final_wait

    def performExposures(self):
        """
        Helper function to do all of the exposures on a single layer.
        """
        for exposure in self.currentLayer.exposures:
            # set the light engine settings
            if exposure.power != self.power:
                self.power = exposure.power
                self.sendCommand(
                    LightEngineBrightness(
                        self.lightEngineName, set=True, brightness=self.power
                    )
                )
                # wait before exposure
            time.sleep(exposure.wait_before)
            self.elapsedTime += exposure.wait_before
            # set the image
            self.sendCommand(
                LightEngineImage(
                    self.lightEngineName,
                    set=True,
                    image=self.printJobFilePath + "slices/" + exposure.image,
                )
            )
            # expose the image
            self.sendCommand(
```

```
                LightEnginePerformExposure(self.lightEngineName,
                    ↪ exposure.exposure_time)
            )
            self.elapsedTime += exposure.exposure_time
            # wait after exposure
            time.sleep(exposure.wait_after)
            self.elapsedTime += exposure.wait_after
```

Functionally, the printjob package does the following in the code above:

1. The call to getPrintJob passes in the location of the print job and the name of the settings file. The function then reads in the settings file and determines which settings file version it conforms to.

2. Each settings file version has its own PrintJob class which is responsible for validating, parsing and creating easy to access data structures for the print job controller. The getPrintJob function creates the appropriate PrintJob object and assuming that the settings file is valid, the function will return a PrintJob object. (Note: the code for handling invalid settings files is not shown in the code sample).

3. For each layer of the print, the PrintJob object will return a Layer object that contains all of the settings for a particular layer of the print job. This includes information about the positioning of the build platform and an array of Image objects that contain the settings for each image that will be exposed on that layer.

While the print job controller is not modular, the printjob package is modular and can help in adapting the print job controller to different 3D printer hardware architectures without having to change the print job controller. Like the rest of the system software, it conforms to the create, test and register philosophy and structurally is similar to the system software's code base:

```
print_job_validator
├── schemas
│   └── v1.json
└── src
```

```
│__ data_structs
│   │__ Layer.py
│   └__ Exposures.py
│__ print_jobs
│   │__ helper_functions.py
│   │__ ABC_PrintJob.py
│   └__ PrintJobV1.py
└__ test
    └__ test_PrintJobV1.py
```

Adding new print job validators requires defining a new JSON schema file that the print_settings.json file must conform to, which is shown in the directory structure above by v1.json. Each JSON schema file that is defined has an accompanying print job class that uses the JSON schema file to validate settings files and that takes the JSON data and puts it into a format that is more easily accessible to the print job controller. The basis for all print job classes is provided by theABC_PrintJob abstract base class, which the print job class PrintJobV1 from the example above inherits from.

New Layer and Exposures data structs can be defined or preexisting ones can be modified to work with the new schema and will be used as the data structures that the print job controller uses to get the layer and exposure settings and image data for each layer. Tests are required to validate that the new print job class and data structures are working correctly, and finally, the new print job handler needs to be registered with the getPrintJob function that is defined in helper_functions.py and provided with a way to distinguish the new schema from the other schemas, usually through the use of a version field in the schema.

5.4 Web server

While the web server requires a number of steps to set up, the actual code for creating and registering new API handlers with flask-restplus is relatively straightforward. Structurally the file system uses the following pattern:

```
system_software
└__ src
    └__ web_server
        └__ api
            │__ controllers
            │   └__ print_jobs ................................ Print job API handlers here
            │__ hardware
```

```
│
│       ├── axes ........................................ Axes API handlers here
│       └── light_engines ........................ Light Engine API handlers here
│           └── LightEngineBrightness.py
├── dist
│   ├── static ................................ webpacked files, like the
│   │   │                                        ones in this directory, are
│   │   │                                        given a unique hash for a
│   │   │                                        name every time the fron-
│   │   │                                        tend is built
│   │   ├── abc.css
│   │   └── xyz.js
│   └── index.html .................................. webpacked Vue frontend
├── frontend
├── routes.py
└── server.py
```

The flask-restplus API handlers are found under the webserver/api directory with each handler, like LightEngineBrightness.py, usually corresponding directly to one of the messaging classes. How these classes are coded is best explained by flask-restplus's documentation [14] and by looking at the format of other API handlers, but registering new handlers requires modifying the setup process for Flask's web server as follows:

```
############################################
# server.py
############################################
from flask import Flask, render_template, Response, request, Blueprint
from flask_restplus import Api, Resource
from src.webserver import flaskapp as app # Flask object

cm = None # config manager
config = None # flask specific config
router = None # message router
tempDir = None

def initAPI(): # called during setup
    """
```

```
    Initializes the API.

    Must be called after the outgoing Queue object has been created.
    """
    from src.webserver.api import api

    apiBlueprint = Blueprint("api", __name__, url_prefix="/api")
    # initialize the apis
    initLightEnginesAPI(api)

    # other process apis here

    # add api to the blueprint
    api.init_app(apiBlueprint)
    # register the api blueprint
    app.register_blueprint(apiBlueprint)

def initLightEnginesAPI(api):
    """
    Initializes all of the api endpoints for the light engines module

    Parameters:
        api (API) - flask_restplus object that handles the api endpoints
    """
    try:
        # check if the light engines has been configured. If not, then don't register its api endpoints
        global cm # configuration manager
        cm.getConfig(ConfigInterfaces.LightEngines)
    except Exception as e:
        print(str(e))
```

```
            return

    # import API handlers here
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesBrightness

    # add other API handlers to flask here
    api.add_namespace(lightEnginesBrightness)
```

Aside from the API handlers, the web server also can serve a frontend web page. The specifics of how this is done is going to vary depending on the frontend, but for the Vue frontend there are several noteworthy features. Foremost is that the webserver/frontend directory is actually a separate git repository that is a submodule of the main system software git repository which makes it convenient to develop the frontend independently of the backend. The Vue frontend also makes use of a technology known as webpack which takes a complex nodejs project and complies it down to minified, obfuscated javascript and CSS files and a single index.html file. When the Vue project is built using npm or yarn, it saves the built files in webserver/dist and webserver/dist/static directories where flask can serve the files from.

5.5 Configuration Management System

Thus far the configuration manager has been referenced several times in the code examples above and, as was discussed in section 5.1, it determines which pieces of the modular system software run and how they run when the backend is started. Simply put, the configuration system takes the modular features in the system software and makes them accessible to people other than programmers by allowing all of the modular aspects of the system software to be represented in a configuration file. This can been seen by looking at an actual configuration file:

```
################################################
# dummy_config.json
```

```
################################################
{
    "General": {
        "name": "test file",
        "comment": "dummy config file for testing",
        "debug-all": true
    },
    "Axes": {
        "drivers": [
            {
                "name": "DummyX",
                "class": "AxisDummyDriver",
                "configuration": {
                    "acceleration": 1,
                    "deceleration": 1,
                    "velocity": 1,
                    "maxPos": 5,
                    "minPos": 2
                },
                "axes": [
                    {
                        "name": "X",
                        "class": "AxisDummyShim",
                        "calibratedPosition": 5,
                        "configuration": {}
                    }
                ]
            }
        ]
    },
    "LightEngines": {
        "drivers": [
```

```
            {
                "name": "DummyA",
                "class": "LightEngineDummyDriver",
                "configuration": {
                    "image_width": 30,
                    "image_height": 50,
                    "brightness_max": 1000,
                    "brightness_min": 0,
                    "refresh_rate_max": 50,
                    "refresh_rate_min": 0
                }
            }
        ]
    },
    "Router": {
        "server configuration": {
            "host": "0.0.0.0",
            "port": 5000,
            "debug": true
        }
    },
    "PrintJob": {
        "light engine name": "DummyA",
        "build platform axis name": "X",
        "build platform axis top position": 5,
        "build platform axis bottom position": 0,
        "build platform axis swap min/max": false
    }
}
```

Each of the processes have their own section that contains the configuration information for their process interface classes and all of the classes that run in the process, like the drivers for the light engines process. The configuration fields roughly represent the arguments and key word arguments that each class accepts when it is initialized. Each of the process interface classes expects the configuration data for all of the class objects it creates at run time to be passed into its run method when the process is started. It will then use the configuration data to create those class objects during setup. This approach requires that the config package contain a rough reflection of the class structure of the backend, which is one of two reasons why structurally the config package is the most complex package in the entire system software.

The other reason is that there are actually two inheritance and registration hierarchies in the package: one for the python code and one for the JSON schemas. Conveniently the files in these hierarchies mirror each other so that every python class is paired with its own JSON schema file. However the way the python code handles inheritance and registration differs from the way JSON schemas do it, which is that registration and inheritance are seen as the same thing. To better understand this, it is best to start by looking at an simple example where there is no inheritance:

Figure 5.2: Simple relationship diagram of the print job process's configuration handler, its JSON schema and a configuration file.

As seen in Figure 5.2, the PrintJobConfig class acts as a configuration handler that parses all of the red text in the configuration file and translates it into a format that the PrintJobController class can accept into its run method. The config_schema.json file is used to define the format of the print job controller's configuration options and is used by the ConfigManager to validate that a provided configuration file conforms to the schema. It is also used as the root JSON schema file that all other JSON schema files must be referenced by to be considered part of the configuration file schema. In this case, the PrintJobController has no classes that it needs to pass configuration information to, therefore the schema for all of the print job controller's configuration data can be contained in config_schema.json.

Processes that need to pass configuration information to other classes, like the axes process, require substantially more infrastructure. As demonstrated in Figure 5.3, the AxesInterfaceConfig configuration handler utilizes other configuration handlers to parse apart the configuration data for the modular components in the axes process. This allows for the schema to adapt to the specific configuration needs of every new axis driver and/or shim.

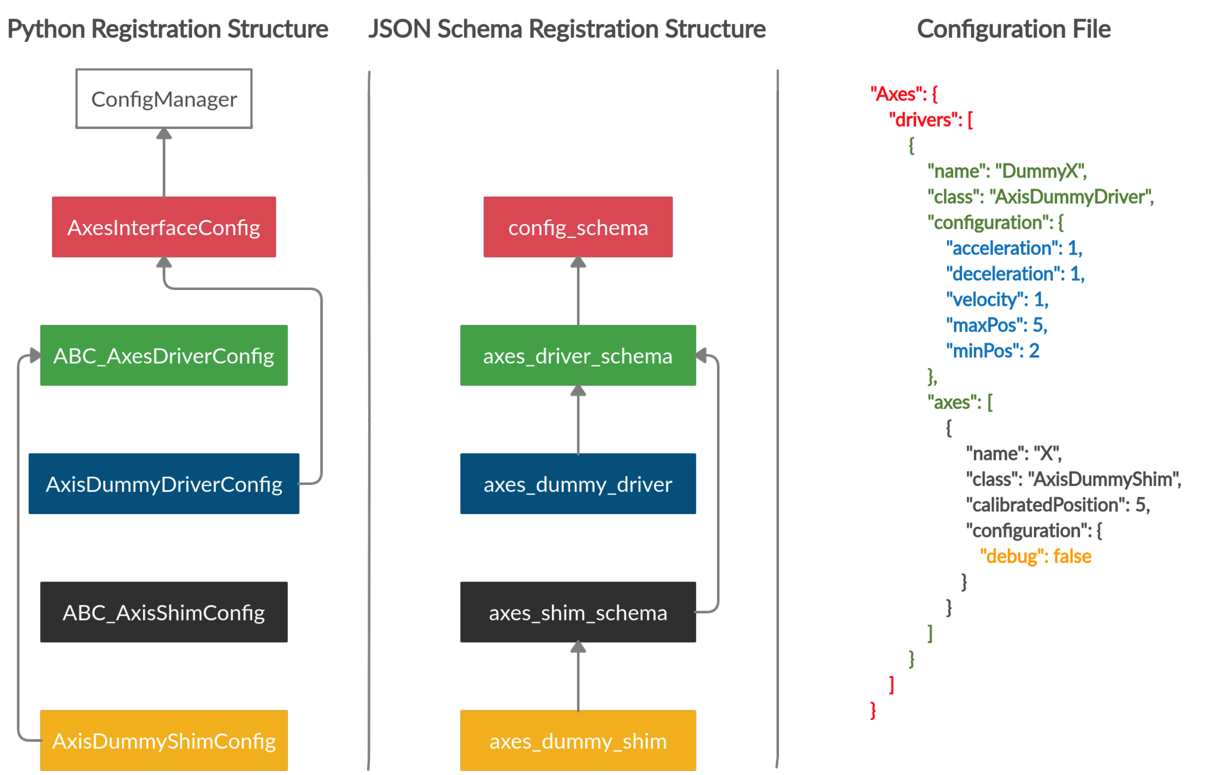

Figure 5.3: Relationship diagram of the configuration management system for the axes process. The arrows indicate what file the originating file or object registers itself with. Not shown in the diagram is the python inheritance structure but for reference, ABC_AxesDriverConfig is the parent of AxisDummyDriverConfig and all other axis drivers, and ABC_AxisShimConfig is the parent of AxisDummyShimConfig and all other axis shims.

There are several other noteworthy features of this architecture. One is that the abstract base classes handle configuration data that is common between all drivers or shims, like what name they should have or what class to use. Child classes of these abstract base classes contain all of the specific configuration information each driver or shim needs, such as what baud rate to run at. Conveniently, the light engines process shares the same structure, just with all of the references to the shims removed.

It is also worth noting that the AxisDummyShimConfig is not registered with the AxisDummyDriverConfig as might be intuitively expected. This is because of the initialization gymnastics that were discussed in section 5.3.2 and the configuration management system has been forced to reflect this. Finally while the ConfigManager mainly reads in and processes configuration files, it also has the ability to write back to configuration files. This is incredibly useful for saving settings that need to persist after restarting the backend, like for what the calibrated position of a particular axis is so that it can be sent back to that position after it has been homed.

While it would be useful to show a full example of how a driver would be integrated into the configuration manager, the size of that code sample is prohibited and thus has been placed in the appendix. However it is worth noting that the JSON schema code in the example is best understood in the context of its documentation [15]. Unfortunately this highlights an issue with the current configuration manager of the system software.

As configuration files are likely to be subjected to the most change by researchers who may not also be programmers, having the rules for the configuration files integrated into the code base in the JSON schema format has a significant negative impact on the how easy it is to create new configuration files. Thankfully there are tools that can take a JSON schema file and produce user friendly documentation in a variety of formats. Specifically the bootprint [16] npm package will return the documentation in a web friendly format that easily could be integrated into the frontend. Unfortunately due limitations in the web server it has proven difficult to serve this documentation page alongside the Vue frontend. This is yet another area that could would benefit from a re-architecting of the web server.

5.6 Tests

The heavily multi threaded nature of the system software makes testing changes or new additions to the code base surprisingly difficult using traditional approaches to development and debugging. For this reason, unit tests have become the preferred development tool for validating that each of the modular pieces of the code are running correctly. They also have proven invaluable during the prototyping phase of new architectures by quickly uncovering where changes have broken the interfaces between process or the interface between the front and backend.

To aid in testing, dummy light engine and axis drivers, axis shims and configuration files have been created to mimic the behavior of real drivers, shims and configuration files. They also provide an excellent prototyping environment that is invaluable when designing the interface for a new hardware process.

The tests have been divided into three categories: driver, process and API that are organized in the following directory structure:

```
system_software
└─ test
    ├─ hardware_tests
    ├─ interface_unit_tests
    └─ api
```

5.6.1 Driver tests

Driver tests are intended to validate both if a hardware driver/ shim is working correctly and can be used to verify hardware integrity. Unlike the process and API tests, they do not make use of Python's built in unittest package. Instead they function as a simple script that requires the user to validate that the actions the script is preforming are actually being executed on the hardware. A simple example is as follows:

```
###############################################
# grbl_test.py
###############################################
from src.hardware.axes.drivers import GrblDriver
from src.hardware.axes import GrblAxisShim
```

```
from src.data_structs import MoveMode

# create the drivers
driver = GrblDriver(numOfAxes=1)
axisZ = GrblAxisShim(driver=driver, grblAxisName="Z")
axisX = GrblAxisShim(driver=driver, grblAxisName="X")
axisZ.initialize()
axisZ.home()
print("Current Position: ", axisZ.getPosition())

print("Moving the printer")
axisZ.setPosition(-5.0, MoveMode.absolute)
axisX.setPosition(-1.0, MoveMode.absolute)
print("Current Position - Z: ", axisZ.getPosition())
print("Current Position - X: ", axisX.getPosition())
```

5.6.2 Processes

Process tests focus on testing a single process interface. These tests work by creating a process, similar to how main.py works, and then sending it messages and monitoring the responses that it gives. A dummy configuration file is used to create the process and for hardware processes, the configuration file specifies that it load dummy drivers and/or shims. This is all done with the unittest package and it creates unit tests that look like the following:

```
#############################################
# test_MessageRouter.py
#############################################
import unittest
from threading import Thread
import time
import os
from multiprocessing import Queue, Process
```

```
from src.config import ConfigManager
from src.process_interfaces.controllers import MessageRouter
from src.data_structs.internal_messages import (
    Shutdown,
    CommandStatus,
)
from src.data_structs.internal_messages.hardware import AxesNames, ABC_AxisMessage
from src.data_structs.internal_messages.controllers import SaveCalibratedPositionToConfig


class TestMessageRouter(unittest.TestCase):
    """
    Class for testing the MessageRouter controller.

    Imitates two processes talking to each other through the MessageRouter.
    """

    dummyPath = (
        os.path.abspath(os.path.dirname(__file__))
        + "/../../../config_files/dummy_config.json"
    )
    wait = 0.1

    def setUp(self):
        """
        Creates a MessageRouter and the message queues for sending test messages to it.
        """
        self.inq = {}
```

```
        self.outq = {}
        self.inq[ABC_AxisMessage.destination] = Queue()
        self.outq[ABC_AxisMessage.destination] = Queue()
        self.inq["proc"] = Queue()
        self.outq["proc"] = Queue()
        self.cm = ConfigManager(self.dummyPath)
        self.router = MessageRouter(self.inq, self.outq)
        Thread(
            target=self.router.run, kwargs={"configManager": self.cm, "
              ↪ debug": True}
        ).start()

    def tearDown(self):
        """
        Shutdown the MessageRouter
        """
        self.router.shutdown()
        for key, value in self.outq.items():
            payload = value.get(timeout=0.1)
            self.assertIsInstance(payload, Shutdown)

    def test_sendMessage(self):
        """
        Sends a basic message from one process to the other.
        """
        msg = AxesNames()
        # send a message to the axes
        self.inq["proc"].put(msg)
        payload = self.outq[ABC_AxisMessage.destination].get(timeout=
            ↪ self.wait)
        # check if we received the message from the axes
        self.assertIsInstance(payload, AxesNames)
```

```
            # send a CommandStatus back to the proc
            self.inq[ABC_AxisMessage.destination].put(
                CommandStatus(payload.uuid, payload.sender)
            )
            payload = self.outq["proc"].get(timeout=self.wait)
            # check if proc got the message
            self.assertIsInstance(payload, CommandStatus)
            self.assertEqual(msg.uuid, payload.uuid)

    # more tests here
```

Running the tests requires that the system software has been initialized once before and that the current terminal session has the python virtual environment currently sourced. Once those conditions are satisfied, tests can be run with the command:

```
$ python -m unittest path/to/test/test_mytest.py
```

For further detail on the unittest package and its command line options refer the package's documentation [17].

5.6.3 API

API tests test the end-to-end functionality of the backend, from the web server API handlers down to the dummy drivers. They function by sending HTTP requests to the API endpoints and validate the responses from the backend. In order for API tests to run, an instance of the web server must already be running on the localhost. Aside from this, the API unit tests also use the unittest package and follow a similar testing methodology.

5.7 Final thoughts

Taken as a whole, the system software's code base is very complex. However the modular design helps breaks the complexity down into smaller and more manageable pieces. Each of these pieces are built on top of patterns that significantly reduce the mental load required to solve problems in the software. But as useful as it is to know these patterns, knowing when and where to use them during development requires a different set of patterns.

CHAPTER 6. DEVELOPMENT PATTERNS

Given the complexity of the system software, determining a workflow that is efficent and effective is important to keeping the software easy to use. This chapter will provide an overview of what files need to be changed, when they need to be changed and how to approach designing those changes for the most common tasks that researchers will undertake in the code base. It also is the simplest explanation of how the create, test and register philosophy has been implemented in the system software and provides a framework for asking intelligent questions and making informed design decisions about the code base, even to those who are not familiar with the code.

6.1 Adding hardware

Hardware is constantly being added, removed and modified on the 3D printers, which means that changes to the drivers will be one of the most common tasks that will need to be performed. The main variable that determines how much work needs to be done is if an interface for the type of hardware that is being added has already been created. For example, adding a new light engine to the system software is a relatively easy task as the light engines process already has a defined API for how to control a light engine, and adding the new hardware only involves creating, testing and registering a new driver that conforms to that interface. However if a new type of hardware, like a strain gauge, was added to the 3D printer, an entirely new API and strain gauge process would need to be created before the strain gauge driver could be added to the system, not to mention changes that may need to be made to the print job controller in order to take advantage of the new hardware.

Generally speaking, the difficulty of a task is directly related to how much of the creation process involves writing registration code. This is because the more registration code that needs to be written, the more time and energy it will take to design the infrastructure of everything that can be registered into that point of the system software. And with more infrastructure comes a more complex system that will take more time to fully test.

Thankfully the structure of the system software naturally produces patterns that can be replicated by subsequent tasks, making it easier to create registration code. These patterns are general enough that they cover the most common development tasks that can be done in the system software, even for hardware that will be added in the future. These patterns are the topic of discussion for the upcoming sections.

6.1.1 Pattern 1: creating a driver for an existing interface

The following steps make up the general pattern for adding a new driver that already has an interface defined for that hardware type:

1. Write a driver that inherits from the driver abstract base class and implement the interface. Depending on how the process interface is setup, this step may include creating a shim and the logic that registers the driver to all of its shims. The shim will also inherit from the shim abstract base class.

2. Write a hardware test script for the new driver and its shims as appropriate. Make sure that all of the interfaces have the desired effect on the hardware.

3. Register the new driver/shims with the process interface.

4. Create a JSON schema file for the driver/shims.

5. Register the new JSON schema file(s) with their appropriate driver/shim JSON schema.

6. Create a configuration handler for the new driver and shim JSON schemas.

7. Register the new configuration handler(s) with their appropriate abstract base class.

8. Write a configuration file and test that the new driver/shims can be correctly controlled by the frontend or the API handlers. Debug as needed and update the hardware test script as appropriate.

Of all of the patterns, this one requires the least work as it has little to no creation of registration code. If a hypothetical new light engine driver were to be added to the light engines process, it would start with defining a new light engine driver class that inherits from ABC_LightEngineDriver and that would be located in src/hardware/light_engines. Once the driver and all of its abstract methods are defined, a new hardware test script can be created in test/hardware_tests/light_tests/.

Next all of the configuration code needs to be defined for the new driver before it can be used in the system software. A new JSON schema file would be created in src/config/schema/light_engines/ and a reference the new schema file would be placed in the light_engine_schema.json file. Then a new configuration handler would be created that inherited from ABC_LightEngineDriverConfig and placed in src/config/hardware/light_engines/light_engine_drivers/. Finally the new configuration handler would be registered in LightEnginesInterfaceConfig.

A new axis driver would follow a very similar pattern to the light engine driver, but with the added complexity of the shims. The process of creating shims and their shared driver technically counts as registration code and is handled by the abstract method createAxes that is defined as part of the ABC_AxisDriver class. Adding the shims to the configuration code is similar to the drivers. Create a JSON schema, register it to the axis_shim_schema.json file, create a configuration handler that inherits from ABC_AxisShimConfig and register it with the ABC_AxisDriverConfig class. More information on the registration structure of the axis process can be found in section 5.5.

6.1.2 Pattern 2: creating a new hardware interface

Adding a new hardware interface will use the following pattern:

1. Design an interface.
2. Create the messaging classes.
3. Create the abstract base class for the driver.
4. Create a dummy driver.

5. Create the process interface.
6. Register the dummy driver with the process interface.
7. Create the configuration handlers for the process interface and the dummy driver.
8. Write tests for the new process interface.
9. Register the new process interface in main.py.
10. Create the API handlers for the web server.
11. Register the API handlers with the web server.
12. Write tests for the API handlers.
13. Add the new hardware functionality to the print job controller or any other relevant processes and test that the changes work.
14. Test run a print job with the new hardware and debug as needed.

This pattern requires significantly more work, starting with designing the interface. For any new process interface, designing the interface is both the most important and often the hardest part of the process, as the interface determines how the messaging classes, API handlers, process interface message handler method and the abstract base class and the drivers, both dummy and real, are coded. Without accounting for drivers, this accounts for over 3000 lines of code apiece for both the light engines and axes processes. With so much code depending on the interface, making major mistakes during the design of the interface can be time consuming to correct.

There are two tactics that can help minimize design errors in the hardware interface. First is to properly research the capabilities of the hardware that is being added. This can include creating a prototype driver to use for experimenting with the capabilities of the hardware. It also should involve researching other models of the hardware and seeing what common features they have, as the design of the interface should strive to be as generic as possible.

Secondly, one of the best ways to figure out what API handlers will be required is to design the user interface for controlling the hardware before writing any code on the backend. This can be done initially with rough sketches, however Vue and Vuetify excel at creating quick, high quality, non-functional user interfaces. Generally speaking, creating a user interface using the same technology that runs the frontend will always provide more information about exactly what information from the backend it will take to drive each of the user interface components.

If hypothetically a strain gauge were to be added to the 3D printer, it would require a new process and process interface to be created. The beginnings of the design process should include researching the hardware by writing a prototype driver, which could be converted to an actual driver once the driver abstract base class is created. It also should include creating a prototype user interface, which for the strain gauge may include the current value the sensor reading, an on/off switch and possibly some sort of calibration functionality. At the end of this process there should be a list of data that needs to be readable from the backend and commands that must be supported which together makes up the hardware interface.

Next the messaging classes must be created, which will be placed in a file src/-data_structs/internal_messages/hardware/strain_gauge_messages.py. Then the driver abstract base class needs to be defined, preferably alongside a dummy driver to enable testing as soon as possible. At this point the strain gauge's hardware interface is defined and only the API handlers and abstract base classes for the configuration handlers need to be defined before the remaining steps conform to pattern 1.

Creating and registering the API handlers is done in src/webserver/api/hardware/s-train_gauges/ and the API tests will need to be created in test/api/hardware/. Finally configuration handlers will need to be created for the strain gauge process interface alongside a driver abstract base class that covers all of the common configuration options between strain gauges. They will need to be registered to each other and the process interface configuration handler will need to be registered to the ConfigManager. A new JSON schema entry will need to be added to the config_schema.json file for the process interface and a separate JSON schema file will need to be created for the driver abstract base class and registered to the config_schema.json.

6.1.3 Pattern 3: adding new controller interfaces

Adding new controllers shares many of the same steps as adding a new hardware interface, but with fewer restrictions. This is a good thing as it gives controllers a tremendous amount of flexibility in what kinds of tasks they are able to perform, but it is also a bad thing in that this can make the design process much more difficult and involved as there are few patterns to guide development. The generalized steps for creating a new controller interface is the following:

1. Design an interface.
2. Create the messaging classes.
3. Create and register any classes that sit behind the controller's process interface class.
4. Create the process interface.
5. Create the configuration handlers for the process interface.
6. Write tests for the new process interface.
7. Register the new process interface in main.py.
8. Create the API handlers for the web server.
9. Register the API handlers with the web server.
10. Write tests for the API handlers.
11. Integrate the new controller into any other relevant processes and test that the changes work.
12. Test run a print job with the new controller and debug as needed.

Once again, designing the interface is the most challenging step. Just as with the new hardware interface, prototyping the user interface before writing any of the backend code is tremendously helpful. However once an interface has been decided upon, finding and determining the edge case behavior for various parts of the software can be tricky. To help

GUI Elements	State					
	Idle	Start Leveling	Leveling	Wait	Printing	Pause
Upload Btn	✓					
Start/Pause Btn	✓				✓	✓
Stop Btn		✓	✓	✓	✓	✓
Begin Printing Btn				✓		
Image	N/A	N/A	N/A	N/A	✓	✓
Number of Layers		✓	✓	✓	✓	✓
Current Layer Number		N/A	N/A	N/A	✓	✓
Run time		✓	✓	✓	✓	✓
Elapsed time		N/A	N/A	N/A	✓	✓
Start modal		✓				
Leveling modal			✓			

Table 6.1: Table used to find all of the edge cases in the GUI components for the print job controller web UI.

find these edge cases, creating a case table is essential. For example, during the creation of the print job controller, the table below was created to determine what user interface elements needed to be enabled or disabled during different states in the print job controller's state machine:

Case tables are an excellent way to rigorously define the behavior of an interface and are an incredibly useful design tool, even for new hardware interfaces. It is worth noting that any interesting behavior that is discovered with a case table should be documented in the appropriate API handlers.

6.2 Summary

All of the designing and explanation in the previous chapters have culminated in these patterns. Once these patterns are understood, the hardest part of working with the system software is designing interfaces. Building the infrastructure of an interface is very straightforward, allowing for researchers to focus on solving research related problems instead of problems with the system software.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

System software is complex without considering the unique challenges and work flow that academic research imposes on software. To meet these specific needs, the system software had to be designed from the ground up with the principles of modularity, ease of use and reliability in mind at every step. This was done by organizing the software into different processes to enforce divisions between the different portions of the code which encourages process specific architecture and enables the software to take full advantage of the Raspberry Pi's computing resources. These divisions also provided a natural place to introduce process specific unit tests and when combined with unit tests for the web API and driver specific tests, the system software is fully end-to-end testable. When combined with a powerful and extensible configuration system, the system software becomes an easy to use tool for rapid 3D printer hardware development.

Additionally by having the modular components of the system software conform to a create, test and register work flow, it is possible to generalize the development process of the software into three straightforward and repeatable patterns. Together this allows the system software to be as flexible as possible while accommodating for our group's ever changing student researcher workforce and allowing research to be the primary focus of the researchers instead of spending time getting the tools that are used to do research working.

For as good as the architecture is, the implementation is not perfect. Specifically the inability to separate the web server into its own process, due to the limitations of Flask's development server, hurts the modularity of the software, although steps have been taken to separate the web server code from the rest of the system software core as much as possible. Fixing this issue is the topic of section 7.2.2. Another problem that is the topic of further discussion in section 7.2.3 is the reliance of the system software on external dependencies and their potential to break the system software.

Finally, when compared to the architectures of any other part of the system software, the configuration manager has the most complicated architecture and can be difficult to reason about in an abstract sense. This is regrettable for such an important piece of the system software, but given how easy it can be to customize the core, it is no surprise that providing a framework for managing all of that potential customization would become a major undertaking in and of itself. As the saying goes, there is no such thing as a free lunch, and the price of having a complex piece of software be highly modular and configurable is that configuring that software in a user friendly manner is going to be complicated.

7.2 Future Research

7.2.1 Integration into production research 3D printer

To date, the system software has successfully controlled motorized stages and a light engine and correctly run a print job using dummy drivers. However it has not yet controlled an entire 3D printer by running an actual print job. To get the system software into a production ready state, drivers need to be written for an existing 3D printer and the software needs to be installed on a 3D printer. Finally a series of test prints need to be run and compared to prints that were created with the old system software. Once the software is performing satisfactorily, further development tasks can be planned.

7.2.2 Replace Flask's development web server with a dedicated web server

As has been expressed multiple times, Flask's web server imposes severe restrictions on the system software and needs to be replaced. Architecturally this imposes several interesting challenges, namely how to manage starting and stopping the core of the system software alongside the web server and how to adapt the development work flow of the API handlers to keep them relatively easy to create and test. These problems relate heavily to the bash scripts that are currently used to install and start the backend and a change in the web server will likely require those scripts to be rewritten.

Replacing the web server and bash scripts is a key step to creating a software ecosystem around the 3D printers. A better web server could serve more web applications alongside the system software's frontend, meaning that all of the tools needed to create a print job could be hosted on a 3D printer. Currently the slicer that is used to turn a CAD model into a series of cross-sectional images is a web application and it would be convenient to bundle it directly with the 3D printer. Additionally tools that help create print job files and that can validate print_settings.json files could be created and added as part of the installation process in the bash scripts. If done correctly, the system software could become the basis for a much larger software ecosystem.

7.2.3 API handler refactor

Late in the development of the system software it was discovered the flask-restplus package that was being used to create the API handlers and their associated documentation web page was no longer being updated and had died as a project. A fork of the project called flask-restx has been created and migrating to it would be ideal. Depending on how much has changed, this could be a large effort, as the API handlers account for 2,000 of the roughly 15,000 lines of code in the backend, or about 13% of the total code base. However this highlights a larger issue with the system software.

As a best practice, system software should minimize the number of external dependencies it relies on, ideally only making use of the programming language's standard library. This is because if something happens to one of these external dependencies, it can result in massive refactoring in the system software. Currently the system software relies on Flask, flask-restplus and a JSON schema validation package as external dependencies. Of the three, Flask has the largest supporting community and is the lowest risk package to employ and the JSON schema package is very actively maintained and has 2.8k stars on GitHub as of this writing.

It would be ideal to remove any kind of dependency for the API handlers. Building the handlers from scratch using Flask is possible but the primary feature that flask-restplus has is the ability to create swagger files and a built in documentation page for the API. Further research in this area may prove fruitful.

7.2.4 Logging

Aside from printing logging messages to the terminal while running, the system software does not include a formal logging system. Currently the logging needs of a 3D printer are not well articulated which is the main challenge in terms of design. Two possible approaches are adding a logging package directly to the system software, or have the logger function as a Linux daemon that services the logging need of all of the software tools that run on the 3D printer. Also there would be design decisions about what format the log data should be saved as, like plain text in a text file or an entry in an SQL database. Finally all of these decisions need to account for the memory limitations of the Raspberry Pi and provide a concrete solution to freeing disk space by deleting old logs.

7.2.5 File browser

Currently the system software only allows for a single print job file or image file for a light engine to be uploaded at a time. Often it is useful to be able to store frequently printed print jobs on the 3D printer for easy access, but this leads to a file management and storage problem which is avoided by the single file upload approach. It would be useful to add API handlers that allow for uploading, deleting, reading the directory structure, renaming, copying, pasting and moving files on the Raspberry Pi so that the file management problem can be outsourced to the users. Such an interface should also include handlers that return how much of the total disk space is currently in use.

7.2.6 Impact assessment and prospects

The primary impact of this work comes from making the system software more accessible in both a technical and human sense. Technically it removes the processing and hardware modularity limitations that plagued previous iterations of the software, while also providing a platform that is capable of interfacing with a much greater variety of software tools. In the human sense, the software provides researchers multiple ways to control a 3D printer, ranging from a polished web GUI to a bash script that is make curl requests. Additionally, this thesis, specifically the information found in chapters 5 and 6, also provides a clearly defined tutorial of how the system software works and provides a way to train more of our group's student researchers on the code base. As the future prospects of the software are tied to having the skilled man power to port the existing hardware drivers over to the system software and to test and debug the software on hardware, having this thesis as a resource is invaluable.

Bibliography

[1] P. Gravesen, J. Branebjerg, and O. S. Jensen, "Microfluidics-a review," Journal of micromechanics and microengineering, vol. 3, no. 4, p. 168, 1993. 1

[2] R. M. Camacho, D. Fish, M. Simmons, P. Awerkamp, R. Anderson, S. Carlson, J. Laney, M. Viglione, and G. P. Nordin, "Self-sustaining 3d thin liquid films in ambient environments," Advanced Materials Interfaces, vol. 7, no. 9, p. 1901887, 2020. 1

[3] M. J. Beauchamp, A. V. Nielsen, H. Gong, G. P. Nordin, and A. T. Woolley, "3d printed microfluidic devices for microchip electrophoresis of preterm birth biomarkers," Analytical chemistry, vol. 91, no. 11, pp. 7418–7425, 2019. 1

[4] C. I. Rogers, K. Qaderi, A. T. Woolley, and G. P. Nordin, "3d printed microfluidic devices with integrated valves," Biomicrofluidics, vol. 9, no. 1, p. 016501, 2015. 1, 2

[5] Y. Xia and G. M. Whitesides, "Soft lithography," Annual review of materials science, vol. 28, no. 1, pp. 153–184, 1998. 1

[6] K. Raj M and S. Chakraborty, "Pdms microfluidics: A mini review," Journal of Applied Polymer Science, vol. 137, no. 27, p. 48958, 2020. 1

[7] K. Ikuta, K. Hirowatari, and T. Ogata, "Three dimensional micro integrated fluid systems (mifs) fabricated by stereo lithography," in Proceedings IEEE Micro Electro Mechanical Systems An Investigation of Micro Structures, Sensors, Actuators, Machines and Robotic Systems, 1994, pp. 1–6. 2, 3

[8] H. Gong, B. P. Bickham, A. T. Woolley, and G. P. Nordin, "Custom 3d printer and resin for 18 m $\times$ 20 m microfluidic flow channels," Lab Chip, vol. 17, pp. 2899–2909, 2017. [Online]. Available: http://dx.doi.org/10.1039/C7LC00644F 2, 7

[9] H. Gong, M. Beauchamp, S. Perry, A. T. Woolley, and G. P. Nordin, "Optical approach to resin formulation for 3d printed microfluidics," RSC advances, vol. 5, no. 129, pp. 106 621–106 632, 2015. xiii, 15, 16

[10] Python Global Interpreter Lock (GIL), accessed on July 26, 2020, https://wiki.python.org/moin/GlobalInterpreterLock. 11

[11] Guide - Vue.js, accessed on July 4, 2020, https://vuejs.org/v2/guide/#top. 25

[12] Component API Overview, accessed on July 4, 2020, https://vuetifyjs.com/en/components/api-explorer/. 25

[13] Cross-origin Resource Sharing, accessed on July 26, 2020, https://fetch.spec.whatwg.org/. 41

[14] Flask-RESTPlus's documentation, accessed on July 4, 2020, https://flask-restplus.readthedocs.io/en/stable/. 82

[15] Understanding JSON Schema, accessed on July 4, 2020, https://json-schema.org/understanding-json-schema/. 89

[16] bootprint, accessed on July 4, 2020, https://www.npmjs.com/package/bootprint. 89

[17] unittest - Unit testing framework, accessed on July 4, 2020, https://docs.python.org/3.7/library/unittest.html. 94

A.1 main.py

```
from src.config import ConfigManager
from multiprocessing import Process, Queue, Array
from src.process_interfaces.hardware import AxesInterface, LightEnginesInterface
from src.process_interfaces.controllers import MessageRouter, PrintJobController
from src.data_structs.internal_messages.controllers import ABC_PrintJobMessage
from src.data_structs.internal_messages.hardware import (
    ABC_AxisMessage,
    ABC_LightEngineMessage,
)
from src.data_structs.internal_messages.controllers import ABC_RouterMessage
import argparse
from src.data_structs import ConfigInterfaces
from src.data_structs.internal_messages import Shutdown
import signal, sys, os, subprocess
import traceback
from threading import Thread
from src.webserver import setup, run


defaultConfigPath = (
    os.path.abspath(os.path.dirname(__file__)) + "/../config_files/dummy_config.json"
)
cm = None
inQs = {}
outQs = {}
```

```
router = None
aifProc = None
leifProc = None
pjifProc = None


def startAxesProcess():
    try:
        axesQueueIn = Queue()
        axesQueueOut = Queue()
        aif = AxesInterface(axesQueueIn, axesQueueOut)
        global aifProc
        axesConfig = cm.getConfig(ConfigInterfaces.Axes)
        aifProc = Process(
            target=aif.run,
            kwargs=axesConfig.getArguments(),
            name=ABC_AxisMessage.destination,
        )
        aifProc.start()
    except:
        traceback.print_exc()
        # if process failed to create, don't create the process
        return

    global inQs, outQs
    inQs[ABC_AxisMessage.destination] = axesQueueOut
    outQs[ABC_AxisMessage.destination] = axesQueueIn


def startLightEnginesProcess():
    try:
        lightEnginesQueueIn = Queue()
```

```
        lightEnginesQueueOut = Queue()
        leif = LightEnginesInterface(lightEnginesQueueIn, lightEnginesQueueOut)
        global leifProc
        LightEnginesConfig = cm.getConfig(ConfigInterfaces.LightEngines)
        leifProc = Process(
            target=leif.run,
            kwargs=LightEnginesConfig.getArguments(),
            name=ABC_LightEngineMessage.destination,
        )
        leifProc.start()
    except:
        traceback.print_exc()
        # if process failed to create, don't create the process
        return

    global inQs, outQs
    inQs[ABC_LightEngineMessage.destination] = lightEnginesQueueOut
    outQs[ABC_LightEngineMessage.destination] = lightEnginesQueueIn


def startPrintJobProcess():
    try:
        pjQueueIn = Queue()
        pjQueueOut = Queue()
        pjif = PrintJobController(pjQueueIn, pjQueueOut)
        global pjifProc
        PrintJobConfig = cm.getConfig(ConfigInterfaces.PrintJob)
        pjifProc = Process(
            target=pjif.run,
            kwargs=PrintJobConfig.getArguments(),
```

```
            name=ABC_PrintJobMessage.destination,
        )
        pjifProc.start()
    except:
        traceback.print_exc()
        # if process failed to create, don't create the process
        return

    global inQs, outQs
    inQs[ABC_PrintJobMessage.destination] = pjQueueOut
    outQs[ABC_PrintJobMessage.destination] = pjQueueIn


def startRouterProcess():
    """
    Creates the Router process.

    Must run after all of the other processes have been created, as the
        ↪ last thing it does is
    call the flask server, which stalls.

    Paramters:
        cm (ConfigManager) - contains config info for the Router and
            ↪ the flask server.
        inQueues (dict) - dictionary of incoming message queues and
            ↪ what processes they belong to.
        outQueues (dict) - dictionary of outgoing message queues and
            ↪ what processes they belong to.
    """
    try:
        global inQs, outQs, cm, router
```

```python
        serverConfig, debug = cm.getConfig(ConfigInterfaces.Router).getArguments()
        router = MessageRouter(inQs, outQs)
        Thread(
            target=router.run,
            kwargs={"configManager": cm, "debug": debug},
            name=ABC_RouterMessage.destination,
        ).start()
        setup(router, serverConfig, cm)
        run()
        router.shutdown()

    except Exception as e:
        traceback.print_exc()
        for _, queue in outQs.items():
            queue.put(Shutdown())


def main():
    """
    Sets up and starts the system software.
    """
    # setup and start all of the processes
    startAxesProcess()
    startLightEnginesProcess()
    startPrintJobProcess()
    startRouterProcess()


parser = argparse.ArgumentParser(description="System software for the HR3 3D printer.")
parser.add_argument(
```

```
    "-g",
    "--gen_docs",
    action="store_true",
    help="generates updated docs for the configuration file",
)
parser.add_argument(
    "-r",
    "--run",
    action="store_true",
    help="Generates the docs and starts the system software with the dummy configuration unless \
        another config file is provided with the --config flag.",
)
parser.add_argument("-c", "--config", type=str, help="path to config file")
args = parser.parse_args()
if args.gen_docs:
    ConfigManager(defaultConfigPath).generateSchemaDocumentation(
        "src/config/schema/single_file_config_for_documentation.json",
        "src/webserver/config_documentation/",
    )
elif args.run:
    cm = ConfigManager(args.config if args.config else defaultConfigPath)
    main()
else:
    parser.print_help()
```

A.2 ABC_Message.py

```
import uuid

```

```
class ABC_Message:
    """
    Parent class for all messages.

    Child classes are intended to be initialized with all of the information that the message
    needs to have. All attributes of the class should be class properties.

    Attributes:
      uuid - unique id for the message
      type - customizable param for specifying the message type
      destination - process the message is intended for
    """

    _uuid = uuid.uuid4().hex
    _type = None
    _sender = None
    _destination = None

    def __init__(self):
        """
        Creates a uuid for the message
        """
        self.uuid = uuid.uuid4().hex

    @property
    def uuid(self):
        return self._uuid

    @uuid.setter
```

```
    def uuid(self, uuid):
        if isinstance(uuid, str):
            self._uuid = uuid
        else:
            raise ValueError(”uuid must be a string”)

    @property
    def type(self):
        return self._type

    @type.setter
    def type(self, newType):
        self._type = newType

    @property
    def sender(self):
        return self._sender

    @sender.setter
    def sender(self, newType):
        self._sender = newType

    @property
    def destination(self):
        return self._destination

    @destination.setter
    def destination(self, newType):
        self._destination = newType
```

A.3 light_engine_message.py

```
from src.data_structs.internal_messages import ABC_Message
from src.data_structs.enums import MessageType, PublisherType
from flask import Flask, render_template, jsonify, send_file
from PIL import Image
import numpy as np


class ABC_LightEngineMessage(ABC_Message):
    """
    Light Engine specific message parent class
    """

    _light_engine = None
    destination = "light_engines"

    def __init__(self):
        super().__init__()

    @property
    def light_engine(self):
        return self._light_engine

    @light_engine.setter
    def light_engine(self, light_engine):
        if isinstance(light_engine, str):
            self._light_engine = light_engine
        else:
            raise ValueError("light engine name must be of type str.")


class LightEnginesNames(ABC_LightEngineMessage):
```

```
    """
    Gets the name of all the light engines.
    """

    def __init__(self):
        super().__init__()

    def __str__(self):
        return "LightEngineNames: {}".format({"uuid": self.uuid, "type"
            ↪ : self.type})


class LightEngineInitialize(ABC_LightEngineMessage):
    """
    Getter/Setter for the initializations state of a light engine.
    """

    def __init__(self, light_engine, set=False):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineInitialize: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self
                ↪ .light_engine}
        )


class LightEnginePower(ABC_LightEngineMessage):
    """
    Getter for the power of a light engine.
```

```
    """

    _power = None

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEnginePower: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self.light_engine,}
        )


class LightEngineImage(ABC_LightEngineMessage):
    """
    Getter/Setter for the image of a light engine.
    """

    def __init__(
        self, light_engine, publisherType=PublisherType.none, set=False, image=None
    ):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        if isinstance(publisherType, PublisherType):
            self.publisherType = publisherType
        else:
            raise ValueError("publisherType must be a PublisherType enum")
        self.light_engine = light_engine
```

```
        self.image = image
        if set and (image is None):
            raise ValueError("Image cannot be set without valid image
                ↪ value")

    def __str__(self):
        return "LightEngineImage: {}".format(
            {
                "uuid": self.uuid,
                "type": self.type,
                "light_engine": self.light_engine,
                "image": self.image,
                "publisherType": self.publisherType,
            }
        )


class LightEngineBrightness(ABC_LightEngineMessage):
    """
    Getter/Setter for the brightness of a light engine.
    """

    _brightness = None

    def __init__(self, light_engine, brightness=None, set=False):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        self.light_engine = light_engine
        if brightness is not None:
            self.brightness = brightness
        if set and (brightness is None):
```

```
            raise ValueError("Image cannot be set without valid
                ↪ brightness value")

    @property
    def brightness(self):
        return self._brightness

    @brightness.setter
    def brightness(self, brightness):
        self._brightness = brightness

    def __str__(self):
        return "LightEngineBrightness: {}".format(
            {
                "uuid": self.uuid,
                "type": self.type,
                "light_engine": self.light_engine,
                "brightness": self.brightness,
            }
        )


class LightEngineImageDimensions(ABC_LightEngineMessage):
    """
    Gets the required dimensions of an image for this light engine
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
```

```
        return "LightEngineMaxBrightness: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self.light_engine}
        )


class LightEngineMaxBrightness(ABC_LightEngineMessage):
    """
    Gets the max valid brightness for a light engine.
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineMaxBrightness: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self.light_engine}
        )


class LightEngineMinBrightness(ABC_LightEngineMessage):
    """
    Gets the min valid brightness for a light engine.
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
```

```
        return "LightEngineMinBrightness: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self
             ↪ .light_engine}
        )


class LightEngineRefreshRate(ABC_LightEngineMessage):
    """
    Getter/Setter for the refresh rate of a light engine.
    """

    _refresh_rate = None

    def __init__(self, light_engine, refresh_rate=None, set=False):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        self.light_engine = light_engine
        if refresh_rate is not None:
            self.refresh_rate = refresh_rate
        if set and (refresh_rate is None):
            raise ValueError("Image cannot be set without valid refresh
             ↪  rate value")

    @property
    def refresh_rate(self):
        return self._refresh_rate

    @refresh_rate.setter
    def refresh_rate(self, refresh_rate):
        self._refresh_rate = refresh_rate

    def __str__(self):
```

```
        return "LightEngineRefreshRate: {}".format(
            {
                "uuid": self.uuid,
                "type": self.type,
                "light_engine": self.light_engine,
                "refresh_rate": self.refresh_rate,
            }
        )


class LightEngineMaxRefreshRate(ABC_LightEngineMessage):
    """
    Gets the max valid refresh rate for a light engine.
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineMaxRefreshRate: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self
                ↪ .light_engine}
        )


class LightEngineMinRefreshRate(ABC_LightEngineMessage):
    """
    Gets the min valid refresh rate for a light engine.
    """

    def __init__(self, light_engine):
```

```
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineMinRefreshRate: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self.light_engine}
        )


class LightEnginePerformExposure(ABC_LightEngineMessage):
    """
    Command to perform an exposure using the current configuration.
    """

    _exposure_time = None

    def __init__(self, light_engine, exposure_time):
        super().__init__()
        self.light_engine = light_engine
        self.exposure_time = exposure_time

    @property
    def exposure_time(self):
        return self._exposure_time

    @exposure_time.setter
    def exposure_time(self, exposure_time):
        self._exposure_time = exposure_time

    def __str__(self):
        return "LightEnginePerformExposure: {}".format(
```

```
                {
                    "uuid": self.uuid,
                    "type": self.type,
                    "light_engine": self.light_engine,
                    "exposure_time": self.exposure_time,
                }
            )


class LightEngineLogMessage(ABC_LightEngineMessage):
    """
    Getter for a new log message
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineDMD: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self
                ↪ .light_engine,}
        )


class LightEngineLogging(ABC_LightEngineMessage):
    """
    Getter/Setter for the logging state of a light engine.
    """

    _logging = None

```

```
    def __init__(self, light_engine, logging=None, set=False):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        self.light_engine = light_engine
        self.logging = logging
        if set and logging is None:
            raise ValueError("DMD cannot be set without valid logging value")

    @property
    def logging(self):
        return self._logging

    @logging.setter
    def logging(self, logging):
        self._logging = logging

    def __str__(self):
        return "LightEngineDMD: {}".format(
            {
                "uuid": self.uuid,
                "type": self.type,
                "light_engine": self.light_engine,
                "logging": self.logging,
            }
        )


class LightEngineLED(ABC_LightEngineMessage):
    """
    Getter/Setter for the led of a light engine.
    """
```

```
    _led = None

    def __init__(self, light_engine, led=None, set=False):
        super().__init__()
        self.type = MessageType.set if set else MessageType.get
        self.light_engine = light_engine
        self.led = led
        if set and led is None:
            raise ValueError("LED cannot be set without valid led value
                ↪ ")

    @property
    def led(self):
        return self._led

    @led.setter
    def led(self, led):
        self._led = led

    def __str__(self):
        return "LightEngineLED: {}".format(
            {
                "uuid": self.uuid,
                "type": self.type,
                "light_engine": self.light_engine,
                "led": self.led,
            }
        )


class LightEngineReset(ABC_LightEngineMessage):
```

```
    """
    Driver reset message.
    """

    def __init__(self, light_engine):
        super().__init__()
        self.light_engine = light_engine

    def __str__(self):
        return "LightEngineReset: {}".format(
            {"uuid": self.uuid, "type": self.type, "light_engine": self.light_engine}
        )
```

A.4 system_messages.py

```
from src.data_structs import ErrorState
from src.data_structs.internal_messages import ABC_Message


class Shutdown(ABC_Message):
    """
    Sent to processes to force them to shutdown cleanly.
    """

    def __init__(self):
        super().__init__()
        self.type = "shutdown"

    def __str__(self):
        return "Shutdown: {}".format({"uuid": self.uuid, "type": self.type})
```

```python


class CommandStatus(ABC_Message):
    """
    Execution status of a command recieved either from the API or another process

    Atrributes:
        state (ErrorState) - error code for the command
        traceback (str) - if state is ErrorState.error, then the stack trace is placed here.
        errorMsg (str) - if state is ErrorState.error, then the error string is placed here.
    """

    _returnVal = None
    _state = ErrorState.none
    _errorMsg = ""
    _traceback = ""

    def __init__(
        self,
        uuid,
        destination,
        returnVal=None,
        errorState=ErrorState.none,
        errorMsg="",
        traceback="",
    ):
        """
        Creates a command status.

```

```
        Parameters:
            uuid (uuid.hex) - unique id of the original message that
                ↪ this object is responding to.
            destination (str) - name of the process that the message is
                ↪  going to
            returnVal (any) - any values that are returned by the
                ↪ function
            errorState (ErrorState) - error code that resulted from the
                ↪  command
            errorMsg (str) - error message
            traceback (str) - traceback of the error
        """
        super().__init__()
        self.type = "status"
        self.uuid = uuid
        self.returnVal = returnVal
        self.state = errorState
        self.destination = destination
        self.errorMsg = errorMsg
        self.traceback = traceback

    @property
    def returnVal(self):
        """
        Getter for the returnVal

        Returns:
            anything - return value of the command
        """
        return self._returnVal

    @returnVal.setter
```

```
    def returnVal(self, retVal):
        """
        Setter for the returnVal
        """
        self._returnVal = retVal

    @property
    def state(self):
        """
        Getter for the state

        Returns:
            ErrorState - error status of the command
        """
        return self._state

    @state.setter
    def state(self, newState):
        """
        Setter for the state
        """
        if isinstance(newState, ErrorState):
            self._state = newState
        else:
            raise ValueError("State must be of type ErrorState")

    @property
    def traceback(self):
        """
        Getter for the traceback

        Returns:
```

```
                str - full stack traceback of any errors
            """
            return self._traceback

        @traceback.setter
        def traceback(self, newMessage):
            """
            Setter for the traceback
            """
            if isinstance(newMessage, (str, type(None))):
                self._traceback = newMessage
            else:
                print(newMessage)
                raise ValueError("traceback must be of type str")

        @property
        def errorMsg(self):
            """
            Getter for the error message

            Returns:
                str - error message
            """
            return self._errorMsg

        @errorMsg.setter
        def errorMsg(self, newMessage):
            """
            Setter for the error message
            """
            if isinstance(newMessage, (str, type(None))):
                self._errorMsg = newMessage
```

```
        else:
            raise ValueError("errorMsg must be of type str")

    def __str__(self):
        """
        Human readable print string.
        """

        return "CommandStatus <obj>: {}".format(
            {
                "uuid": self.uuid,
                "sender": self.sender,
                "destination": self.destination,
                "returnVal": self.returnVal,
                "state": self.state,
                "errorMsg": self.errorMsg,
                "traceback": self.traceback,
            }
        )
```

A.5 LightEnginesInterface.py

```
import src.hardware.light_engines as drivers
import traceback, sys
from src.data_structs.internal_messages import CommandStatus, Shutdown
from src.process_interfaces import ABC_Interface
from src.data_structs import ErrorState
from threading import Thread
from src.data_structs.internal_messages.hardware import (
    LightEnginesNames,
    LightEnginePower,
    LightEngineInitialize,
```

```
    LightEngineReset,
    LightEngineLogging,
    LightEngineLogMessage,
    LightEngineLED,
    LightEngineRefreshRate,
    LightEngineMaxRefreshRate,
    LightEngineMinRefreshRate,
    LightEnginePerformExposure,
    LightEngineImage,
    LightEngineBrightness,
    LightEngineMaxBrightness,
    LightEngineMinBrightness,
    LightEngineImageDimensions,
)
from src.data_structs import MessageType


class LightEnginesInterface(ABC_Interface):
    """
    Interface for the process that controls all hardware light engines.
    Documentation for undocumented functions can be found inside the Interface abstract base class.
    Attributes:
        light_engines (dict): dictionary of all of the light engine classes. The keys are the name of the light engine and the
                        values are the light engine object.
    """

    light_engines = {}
    valid_sub_configs = ["DummyDriver", "I2CLightEngine"]

```

```
    def __init__(self, in_queue, out_queue):
        """
        Sets the input and output queues
        Parameters:
            in_queue (Queue): input queue from the flask process
            out_queue (Queue): output queue from the flask process
        """
        super().__init__(in_queue, out_queue)

    def setupLightEngines(self, light_engine_drivers=[]):
        """
        Initializes all of the light engine and driver objects for the configuration
        specified in the config file.
        All light engine objects will be stored in self.light_engines.
        Parameters:
            light_engine_drivers (dict): passed in configuration of the light engine
        """

        # TODO: probably something wrong here
        for driverConfig in light_engine_drivers:
            # get the light_engine_drivers class object
            module = getattr(drivers, driverConfig.getClassName())
            # use the driver to create the light_engines object
            initParams = driverConfig.getArguments()
            self.light_engines[driverConfig.getName()] = module(**initParams)

    def run(self, light_engine_drivers=[], debug=False):
        """
        Starting point for the LightEnginesInterface Process.
```

```
        Parameters:
            light_engine_drivers (list) - configuration options for each driver and the light engines that are
                attached to it. Each item in the list should be a dictionary with the config
                params of a driver, with one of the keys containing a list of all the config
                params for all the light engines that will be using the driver. See docs/Config_Files.md
                for more details.
        """
        self.debug = debug
        self.setupLightEngines(light_engine_drivers)
        self.processMessages()

    def messageLogic(self, payload):
        if self.debug:
            print("LightEnginesInterface received payload: ", payload)
        try:
            if isinstance(payload, LightEnginesNames):
                self.outq.put(
                    CommandStatus(
                        payload.uuid,
                        payload.sender,
                        returnVal=list(self.light_engines.keys()),
                    )
                )

            elif isinstance(payload, LightEngineInitialize):
                if payload.type == MessageType.get:
                    self.sendResponseMessage(
                        payload.uuid,
```

```
                    payload.sender,
                    self.light_engines[payload.light_engine].get_initialized,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].set_initialized,
                )

        elif isinstance(payload, LightEnginePower):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_power,
            )

        elif isinstance(payload, LightEngineImageDimensions):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_image_dimensions,
            )

        elif isinstance(payload, LightEngineImage):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
```

```
                    self.light_engines[payload.light_engine].get_image,
                    (payload.publisherType),
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].set_image,
                    (payload.image),
                )

        elif isinstance(payload, LightEngineRefreshRate):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].get_refresh_rate,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].set_refresh_rate,
                    (payload.refresh_rate),
                )

        elif isinstance(payload, LightEngineMaxRefreshRate):
            self.sendResponseMessage(
```

```python
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_max_refresh_rate,
            )

        elif isinstance(payload, LightEngineMinRefreshRate):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_min_refresh_rate,
            )

        elif isinstance(payload, LightEnginePerformExposure):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].perform_exposure,
                (payload.exposure_time),
            )

        elif isinstance(payload, LightEngineBrightness):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].get_brightness,
                )
            else:
```

```
                    self.sendResponseMessage(
                        payload.uuid,
                        payload.sender,
                        self.light_engines[payload.light_engine].set_brightness,
                        (payload.brightness),
                    )

        elif isinstance(payload, LightEngineMaxBrightness):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_max_brightness,
            )

        elif isinstance(payload, LightEngineMinBrightness):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_min_brightness,
            )

        elif isinstance(payload, LightEngineLogging):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].get_logging,
                )
```

```
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].set_logging,
                    (payload.logging),
                )

        elif isinstance(payload, LightEngineLogMessage):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.light_engines[payload.light_engine].get_hw_log,
            )

        elif isinstance(payload, LightEngineLED):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].get_led,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].set_led,
                    (payload.led),
```

```
                    )

            elif isinstance(payload, LightEngineReset):
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.light_engines[payload.light_engine].reset_driver,
                )

        except Exception as e:
            self.outq.put(
                CommandStatus(
                    payload.uuid,
                    payload.sender,
                    errorState=ErrorState.error,
                    errorMsg="{}: {}".format(type(e).__name__, e.args),
                    traceback=traceback.print_exc(),
                )
            )

    def shutdown(self):
        super().shutdown()
        print("Light Engines Interface Shutdown")
```

A.6 ABC_LightEngineDriver.py

```
import abc
from threading import Lock
from functools import wraps


```

```python
class ABC_LightEngineDriver(metaclass=abc.ABCMeta):
    """
    This class defines the minimum interface needed for a driver.
    The purpose of a driver is:
    1. Contain all of the objects necessary for direct communication with the hardware.
    2. Initializing communication with the hardware.
    3. Cleanly disconnecting and shutting down the hardware.
    4. Keeping track of the initialized state variable.
    5. Resetting the hardware driver
    Attributes:
        initialized (bool) - state variable to track if the software connected to the hardware
        state_lock (Lock) - for use when reading/writing state variables to keep the driver thread safe.
        power (bool) - state variable to track if the power to the light engine is on or off
    """

    _initialized = False
    _state_lock = Lock()
    _power = False

    @abc.abstractmethod
    def __init__(self, **kwargs):
        """
        Initializes the Light Engine object.
        Returns:
            none or error if invaild
        """
        raise NotImplementedError

```

```
    @abc.abstractmethod
    def get_image_dimensions(self):
        """
        Gets the dimensions of the expected image

        Returns:
            dict - width and height keywords
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_initialized(self):
        """
        Getter for the initialized state
        Returns:
            self._initialized (bool): is the hardware connected
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_initialized(self):
        """
        Setter for the initialized state
        """
        raise NotImplementedError

    @abc.abstractmethod
    def reset_driver(self):
        """
        Resets the state of the driver / hardware to a pre-initialized
            ↪ state
        """
```

```
            raise NotImplementedError

    @abc.abstractmethod
    def get_power(self):
        """
        Getter for the power state
        Returns:
            self._power (bool as string)
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_led(self, pos):
        """
        Params:
            on/off
        Setter for the LED to on or off
        Returns:
            none or error if invalid
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_logging(self, pos):
        """
        Params:
            on/off
        Setter for the logging to on or off
        Returns:
            none or error if invalid
        """
        raise NotImplementedError
```

```

    @abc.abstractmethod
    def set_image(self, path):
        """
        Params:
            image file
        Setter for the image to given file
        Absolute path to the image is given. It is expected for this
           ↪ function to validate the
        image, including if it is the correct dimensions.
        Returns:
            none or error if invalid
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_brightness(self, brightness):
        """
        Params:
            brightness
        Setter for the brightness
        Returns:
            none or error if invalid
        """
        raise NotImplementedError

    @abc.abstractmethod
    def set_refresh_rate(self, rate):
        """
        Params:
            refresh rate
        Setter for the refresh rate
```

```
        Returns:
            none or error if invalid
        """
        raise NotImplementedError

    @abc.abstractmethod
    def perform_exposure(self, exposure_time):
        """
        Exposes the current image using the currently configured
            ↪ brightness and refresh rate
        Params:
            exposure time
        Returns:
            none or error if invalid
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_led(self):
        """
        Getter for the LED
        Returns:
            on or off as string
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_logging(self):
        """
        Getter for the logging
        Returns:
            on or off as string
```

```
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_image(self):
        """
        Getter for the image
        Returns:
            return image as a png
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_brightness(self):
        """
        Getter for the brightness
        Returns:
            brightness as string
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_refresh_rate(self):
        """
        Getter for the refresh rate
        Returns:
            refresh rate as string
        """
        raise NotImplementedError

    @abc.abstractmethod
    def get_hw_log(self):
```

```
        """
        Getter for log messages about the hardware.

        Since it is up to the front end to ask for log messages, there is potential
        for this function to be spammed. Best practice is to queue up all calls to this
        function and periodically return a single log message to all of the calls at the
        same time. See LightEngineDummyDriver for an example.

        Returns:
            str - log message
        """
        raise NotImplementedError

    @abc.abstractmethod
    def __str__(self):
        """
        Returns message string
        """
```

A.7 LightEngineDummyDriver.py

```
from src.hardware.light_engines import ABC_LightEngineDriver
from threading import Event, Thread
from src.data_structs import ErrorState
from src.errors import InitializationError
import traceback
import numpy as np
import os
import time
```

```python
from PIL import Image
from src.data_structs import Publisher, publisher, PublisherType


class LightEngineDummyDriver(ABC_LightEngineDriver):
    """
    Dummy driver class to be used for testing purposes and as an example of what an actual driver
    class may look like. It only controls one light engine.
    Documentation for undocumented functions can be found inside the Driver abstract base class.
    """

    def __init__(
        self,
        image_width=20,
        image_height=20,
        brightness_max=100,
        brightness_min=1,
        refresh_rate_max=50,
        refresh_rate_min=1,
        light_engine_name="Dummy",
        tempDir="",
        logging=False,
        loggingFreq=2,
    ):
        """
        Initializes a dummy light engine
        Parameters:
            power (bool) - power of the light engine
            image_path (string) - path to the image of the light engine
            image_width (int) - width of the image
```

```
            image_height (int) - height of the image
            brightness_max (int) - max for brightness
            brightness_min (int) - min for brightness
            refresh_rate_max (int) - max for refresh rate
            refresh_rate_min (int) - min for refresh rate
            dmd (bool) - dmd of the light engine
            led (bool) - led of the light engine
            light_engine_name (string) - name of the light engine
        """
        self.initialized = False
        self.power = True
        self.image_width = image_width
        self.image_height = image_height
        self.image_path = ""
        self.image_path_publisher = Publisher(1)
        self.brightness = brightness_min
        self.brightness_max = brightness_max
        self.brightness_min = brightness_min
        self.refresh_rate = refresh_rate_min
        self.refresh_rate_max = refresh_rate_max
        self.refresh_rate_min = refresh_rate_min
        self.led = False
        self.light_engine_name = light_engine_name
        self.tempDir = tempDir
        # logging stuff
        self.logging = logging
        self.loggingFreq = loggingFreq
        self.logEvent = Event()
        self.logQLength = 0
        self.logMsg = ""

        loggingThread = Thread(target=self.setLog)
```

```
        loggingThread.setDaemon(
            True
        )  # set as a daemon so that the thread stops when the program exit
        loggingThread.start()

    def get_image_dimensions(self):
        return {"width": self.image_width, "height": self.image_height}

    def get_initialized(self):
        """
        Gets value of initialized
        """
        with self._state_lock:
            return self.initialized

    def set_initialized(self):
        with self._state_lock:
            self.initialized = True

    def get_power(self):
        with self._state_lock:
            return self.power

    @publisher("image_path_publisher")
    def get_image(self, publisherType):
        """
        Gets the image of the light engine

        Parameters:
            publisherType (PublisherType) - used by @publisher
        """
```

```python
        if not self.initialized:
            raise ValueError("cannot get image. Driver is not connected to the hardware.")
        return self.image_path

    def set_image(self, path):
        """
        Sets the image of the light engine

        Since the image is never loaded onto a light engine and the
        path to the so called image always stays the same, this
        function does nothing beside checking if it is valid to set the image

        """
        # make sure image can be set
        if not self.initialized:
            raise InitializationError("driver not initialized")
        if not self.power:
            raise ValueError("cannot set image, power not on")
        if not os.path.exists(path):
            raise ValueError("Image file at {} does not exist.".format(path))
        if "png" not in path:
            raise ValueError("Light engine only displays pngs")
        # TODO: validate that the image is the correct dimensions
        self.image_path = path
        self.image_path_publisher.setChangePublish()
        # NOTE: normally the image would be sent to the light engine at this point

    def get_logging(self):
```

```
        with self._state_lock:
            return self.logging

    def set_logging(self, logging):
        with self._state_lock:
            self.logging = logging

    def get_led(self):
        if not self.power:
            raise ValueError("cannot get led, power not on")
        if not self.initialized:
            raise ValueError("cannot get led. Driver is not connected to the hardware.")
        with self._state_lock:
            return self.led

    def set_led(self, led):
        if not self.power:
            raise ValueError("cannot set led, power not on")
        if not self.initialized:
            raise ValueError("cannot get led. Driver is not connected to the hardware.")
        if isinstance(led, (bool)):
            with self._state_lock:
                self.led = led
        else:
            raise ValueError("led can only be a boolean")

    def get_brightness(self):
        if not self.power:
            raise ValueError("cannot get brightness, power not on")
        if not self.initialized:
```

```
            raise ValueError(
                "cannot get brightness. Driver is not connected to the hardware."
            )
        with self._state_lock:
            return self.brightness

    def set_brightness(self, brightness):
        if not self.power:
            raise ValueError("cannot set brightness, power not on")
        if not self.initialized:
            raise ValueError(
                "cannot get brightness. Driver is not connected to the hardware."
            )
        if brightness >= self.brightness_min and brightness <= self.brightness_max:
            with self._state_lock:
                self.brightness = brightness
        else:
            raise ValueError(
                "Brightness {} is invalid. Brightness should be within {} and {}.".format(
                    brightness, self.brightness_min, self.brightness_max
                )
            )

    def get_min_brightness(self):
        if not self.power:
            raise ValueError("cannot get min brightness, power not on")
        with self._state_lock:
```

```python
            return self.brightness_min

    def get_max_brightness(self):
        if not self.power:
            raise ValueError("cannot get max brightness, power not on")
        with self._state_lock:
            return self.brightness_max

    def get_refresh_rate(self):
        if not self.power:
            raise ValueError("cannot get refresh rate, power not on")
        if not self.initialized:
            raise ValueError(
                "cannot get refresh rate. Driver is not connected to the hardware."
            )
        with self._state_lock:
            return self.refresh_rate

    def set_refresh_rate(self, refresh_rate):
        if not self.power:
            raise ValueError("cannot set refresh rate, power not on")
        if not self.initialized:
            raise ValueError(
                "cannot get refresh rate. Driver is not connected to the hardware."
            )
        if (
            refresh_rate >= self.refresh_rate_min
            and refresh_rate <= self.refresh_rate_max
        ):
            with self._state_lock:
```

```
                self.refresh_rate = refresh_rate
        else:
            raise ValueError(
                "Refresh Rate {} is invalid. Refresh rate should be within {} and {}.".format(
                    refresh_rate, self.refresh_rate_min, self.refresh_rate_max
                )
            )

    def get_min_refresh_rate(self):
        if not self.power:
            raise ValueError("cannot get min refresh rate, power not on")
        with self._state_lock:
            return self.refresh_rate_min

    def get_max_refresh_rate(self):
        if not self.power:
            raise ValueError("cannot get max refresh rate, power not on")
        with self._state_lock:
            return self.refresh_rate_max

    def perform_exposure(self, exposure_time):
        # check if the current image is valid
        if not os.path.exists(self.image_path):
            raise ValueError("No valid image to expose")
        # turn on the lighting elements
        # it is assumed that the image was written to the light engine by the image setter
        self.set_led(True)
```

```python
        # simulate wait for the exposure time to pass
        with self._state_lock:
            # convert the exposure time to ms
            time.sleep(exposure_time / 1000)
        # turn everything off
        self.set_led(False)

    def get_hw_log(self):
        if not self.logging:
            raise ValueError("Logging is not enabled")
        with self._state_lock:
            # increment queue size
            self.logQLength += 1
        # wait for new log message
        self.logEvent.wait()
        with self._state_lock:
            # clear the log event  if last call in the queue
            if self.logQLength == 1:
                self.logEvent.clear()
            # take function call off of the queue
            self.logQLength -= 1
        return self.logMsg

    def setLog(self):
        while True:
            with self._state_lock:
                self.logMsg = "Dummy log message"
            if self.logQLength > 0:
                self.logEvent.set()
            time.sleep(self.loggingFreq)

    def __str__(self):
```

```
        return (
            "This LE currently has image: "
            + str(self.image_path)
            + ", power: "
            + str(self.power)
            + ", led: "
            + str(self.led)
            + ", logging: "
            + str(self.logging)
            + ", brightness: "
            + str(self.brightness)
            + ", refresh rate: "
            + str(self.refresh_rate)
        )

    def reset_driver(self):
        self.initialized = False
        self.image_path = ""
        self.image_path_publisher.setChangePublish()
        self.brightness = self.brightness_min
        self.refresh_rate = self.refresh_rate_min
        self.logging = False
        self.logEvent.clear()
        self.logQLength = 0
        self.logMsg = ""
        self.led = False
```

A.8 AxisDummyDriver.py

```
from src.hardware.axes.drivers import ABC_AxisDriver
from threading import Lock
import src.hardware.axes as axes
```

```


class AxisDummyDriver(ABC_AxisDriver):
    """
    Dummy driver class to be used for testing purposes and as an example of what an actual driver
    class may look like. It only controls one axis.

    In this case, the homed state is part of the driver.

    Documentation for undocumented functions can be found inside the Driver abstract base class.
    """

    _homed = False
    _position = 0
    _acceleration = 1
    _deceleration = 1
    _velocity = 10
    _maxPos = 5
    _minPos = 0
    validAxes = ["AxisDummyShim"]

    def __init__(self, acceleration=1, deceleration=1, velocity=10, maxPos=5, minPos=0):
        """
        Initializes the values of the driver.

        Also this docstring is used for generating documention for the config file, so please
        make sure it's been filled out.

```

```
        Parameters
            acceleration (float) - acceleration of the axis
            deceleration (float) - deceleration of the axis
            velocity (float) - velocity that the axis moves at
            maxPos (float) - max valid position of the axis
            minPos (float) - min valid position of the axis
        """
        self.acceleration = acceleration
        self.deceleration = deceleration
        self.velocity = velocity
        self.maxPos = maxPos
        self.minPos = minPos

    def reset_driver(self):
        super().reset_driver()
        self.homed = False

    @property
    def homed(self):
        """
        Getter for the homed state

        Returns:
            self._homed (bool): is the hardware homed
        """
        with self.stateLock:
            return self._homed

    @homed.setter
    def homed(self, state):
        """
        Setter for the homed state
```

```
        """
        newState = state and self.initialized
        with self.stateLock:
            # and-ing the value ensures that the driver cannot be homed
            # without also being initialized
            self._homed = newState

    @property
    def position(self):
        """
        Gets the position of the axis
        """
        with self.stateLock:
            return self._position

    @position.setter
    def position(self, pos):
        """
        Sets the position of the axis

        Parameters:
            pos (float): new position
        """
        # precalculated to avoid the lock waiting for itself to release
        validPosition = self.minPos <= pos <= self.maxPos
        if validPosition:
            with self.stateLock:
                self._position = pos
        else:
            raise ValueError(
                "Position: {} is out of bounds. Values must be between
                    ↪ {} <= position <= {}.".format(
```

```
                    pos, self.minPos, self.maxPos
                )
            )

    @property
    def acceleration(self):
        with self.stateLock:
            return self._acceleration

    @acceleration.setter
    def acceleration(self, pos):
        with self.stateLock:
            self._acceleration = pos

    @property
    def deceleration(self):
        with self.stateLock:
            return self._deceleration

    @deceleration.setter
    def deceleration(self, pos):
        with self.stateLock:
            self._deceleration = pos

    @property
    def velocity(self):
        with self.stateLock:
            return self._velocity

    @velocity.setter
    def velocity(self, pos):
        with self.stateLock:
```

```
            self._velocity = pos

    @property
    def maxPos(self):
        with self.stateLock:
            return self._maxPos

    @maxPos.setter
    def maxPos(self, val):
        if isinstance(val, (int, float)):
            with self.stateLock:
                self._maxPos = val
        else:
            raise ValueError("maxPos can only be an int or float")

    @property
    def minPos(self):
        with self.stateLock:
            return self._minPos

    @minPos.setter
    def minPos(self, val):
        if isinstance(val, (int, float)):
            with self.stateLock:
                self._minPos = val
        else:
            raise ValueError("minPos can only be an int or float")

    @staticmethod
    def createAxes(driverConfig={}, shims=[]):
        """
```

```
        Given configuration parameters, this function creates a properly configured driver
        and uses it to create properly configured axis objects.

        This function can be called without creating an object first.

        Parameters:
            driverConfig (dict) - kwargs for DummyDriver.__init__()
            shims (list of AxisShimConfig) - configs for all of the axis associated with this driver.

        Returns:
            output (dict) - dictionary of all the axis objects of the format {axisName: axisObject}
        """
        # create the driver
        driver = AxisDummyDriver(**driverConfig)
        output = {}
        # create the shims
        for axisConfig in shims:
            # verify the axix is compatible with the driver
            if axisConfig.getClassName() in AxisDummyDriver.validAxes:
                module = getattr(axes, axisConfig.getClassName())
                kwargs = axisConfig.getArguments()
                # add driver to input kwargs
                kwargs["driver"] = driver
                output[axisConfig.getName()] = module(**kwargs)

            else:
                raise ValueError(
                    "The axis {} is not a valid axis to use with the DummyDriver".format(
```

```
                                axisConfig.getClassName()
                            )
                        )
        return output
```

A.9 AxesInterface.py

```
from src.process_interfaces import ABC_Interface
import src.hardware.axes.drivers as drivers
import traceback, sys
from src.data_structs import ErrorState
from src.data_structs.internal_messages import CommandStatus, Shutdown
from threading import Thread
from src.data_structs.internal_messages.hardware import (
    AxesNames,
    AxisAcceleration,
    AxisDeceleration,
    AxisCalibratedPosition,
    AxisGoToCalibratedPosition,
    AxisHome,
    AxisInitialize,
    AxisMaxPosition,
    AxisMinPosition,
    AxisReset,
    AxisPosition,
    AxisVelocity,
)
from src.data_structs.internal_messages.controllers import
    ↪ SaveCalibratedPositionToConfig
from src.data_structs import MessageType


```

```
class AxesInterface(ABC_Interface):
    """
    Interface for the process that controls all hardware axes.

    Documentation for undocumented functions can be found inside the Interface abstract base class.

    Attributes:
        axisShims (dict): dictionary of all of the axis classes. The keys are the name of the axis and the
                          values are the axis object.
        validSubConfigs (List): list of the valid classes that can be configured. Used mainly
            for the documentation of the configuration manager.
    """

    axisShims = {}
    validSubConfigs = ["AxisDummyDriver, GrblDriver"]

    def __init__(self, inQueue, outQueue):
        """
        Sets the input and output queues

        Parameters:
            inQueue (Queue): input queue from the flask process
            outQueue (Queue): output queue from the flask process
        """
        super().__init__(inQueue, outQueue)

    def setupAxes(self, axisDrivers=[]):
        """
```

```
        Initializes all of the axis and driver objects for the configuration
        specified in the config file.

        All axis objects will be stored in self.axisShims.

        Parameters:
            axisDrivers (list of AxesDriverConfig): passed in configuration of the axis
        """
        # for each driver
        for driverConfig in axisDrivers:
            # get the driver class object
            module = getattr(drivers, driverConfig.getClassName())
            # use the driver to create the axes objects
            config, shims = driverConfig.getArguments()
            axes = module.createAxes(config, shims)
            # save them to AxesInterface.axes
            for name, obj in axes.items():
                self.axisShims[name] = obj

    def run(self, axisDrivers=[], debug=False):
        """
        Starting point for the AxesInterface Process.

        Parameters:
            axisDrivers (list) - configuration options for each driver and the axes that are
                attached to it. Each item in the list should be a dictionary with the config
                params of a driver, with one of the keys containing a list of all the config
```

```
                params for all the axes that will be using the driver. See docs/Config_Files.md
                for more details.
        """
        self.debug = debug
        self.setupAxes(axisDrivers)
        self.processMessages()

    def messageLogic(self, payload):
        if self.debug:
            print("AxesInterface received payload: ", payload)
        try:
            if isinstance(payload, AxesNames):
                self.sendMessage(
                    CommandStatus(
                        payload.uuid,
                        payload.sender,
                        returnVal=list(self.axisShims.keys()),
                    )
                )

            elif isinstance(payload, AxisInitialize):
                if payload.type == MessageType.get:
                    self.sendResponseMessage(
                        payload.uuid,
                        payload.sender,
                        self.axisShims[payload.axis].getInitialized,
                    )
                else:
                    self.sendResponseMessage(
                        payload.uuid,
                        payload.sender,
```

```
                    self.axisShims[payload.axis].initialize,
                )

        elif isinstance(payload, AxisHome):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].getHomed,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.axisShims[
                      ↪ payload.axis].home
                )

        elif isinstance(payload, AxisPosition):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].getPosition,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].setPosition,
                    payload.pos,
                    payload.mode,
                )

```

```
        elif isinstance(payload, AxisCalibratedPosition):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].getCalibratedPosition,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].setCalibratedPosition,
                )
                # update the config file
                self.sendMessage(
                    SaveCalibratedPositionToConfig(
                        payload.axis, self.axisShims[payload.axis].getPosition()
                    )
                )

        elif isinstance(payload, AxisGoToCalibratedPosition):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].goToCalibratedPosition,
            )

        elif isinstance(payload, AxisMaxPosition):
```

```
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].getMaxPosition,
            )

        elif isinstance(payload, AxisMinPosition):
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].getMinPosition,
            )

        elif isinstance(payload, AxisAcceleration):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].getAcceleration,
                )
            else:
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.axisShims[payload.axis].setAcceleration,
                    payload.accel,
                )

        elif isinstance(payload, AxisDeceleration):
            if payload.type == MessageType.get:
                self.sendResponseMessage(
                    payload.uuid,
```

```
                payload.sender,
                self.axisShims[payload.axis].getDeceleration,
            )
        else:
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].setDeceleration,
                payload.decel,
            )

    elif isinstance(payload, AxisVelocity):
        if payload.type == MessageType.get:
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].getVelocity,
            )
        else:
            self.sendResponseMessage(
                payload.uuid,
                payload.sender,
                self.axisShims[payload.axis].setVelocity,
                payload.vel,
            )

    elif isinstance(payload, AxisReset):
        self.sendResponseMessage(
            payload.uuid,
            payload.sender,
            self.axisShims[payload.axis].reset_driver,
        )
```

```
        except Exception as e:
            self.sendMessage(
                CommandStatus(
                    payload.uuid,
                    payload.sender,
                    errorState=ErrorState.error,
                    errorMsg="{}: {}".format(type(e).__name__, e.args),
                    traceback=traceback.print_exc(),
                )
            )

    def shutdown(self):
        super().shutdown()
        print("Axes Interface Shutdown")
```

A.10 PrintJobController.py

```
from src.process_interfaces import ABC_Interface
import glob
import os
from printjob import getPrintJob
from threading import Thread, Lock, Event
from src.data_structs.internal_messages import Shutdown, CommandStatus
from src.data_structs import ErrorState, MoveMode
from src.errors import PrintJobCommandError
import traceback
from functools import wraps
import time
from src.data_structs import PrintJobState as State
from src.data_structs.internal_messages.controllers import (
    PrintJobStart,
```

```
    PrintJobStop,
    PrintJobPause,
    PrintJobNext,
    PrintJobIsRunning,
    PrintJobState,
    PrintJobGetCurrentImage,
    PrintJobGetNumberOfLayers,
    PrintJobGetCurrentLayerNumber,
    PrintJobRunTime,
    PrintJobElapsedTime,
    PrintJobLogMessage,
    PrintJobFolderLocation,
)
from src.data_structs.internal_messages import ABC_Message
from src.data_structs.internal_messages.hardware import (
    AxisInitialize,
    AxisPosition,
    AxisHome,
)
from src.data_structs.internal_messages.hardware import (
    LightEngineImage,
    LightEngineInitialize,
    LightEngineBrightness,
    LightEnginePerformExposure,
)


class PrintJobController(ABC_Interface):
    """
    Interface for the process the runs a print job.

```

```
    Before messing with this code, make sure to read up on how mutexes and the threading.Lock class work.
    Otherwise, if you use the stateLock variable incorrectly, you can get some nasty shared data bugs, or
    cause the entire controller to lock up.

    Documentation for undocumented functions can be found inside the Interface abstract base class.

    Attributes:
        current_state (PrintJobState) - state of the print job
    """

    _stateLock = Lock()
    _currentState = State.idle
    _sleep_duration = 0.1

    def __init__(self, inQueue, outQueue):
        """
        Sets the input and output queues

        Parameters:
            inQueue (Queue): input queue from the flask process
            outQueue (Queue): output queue from the flask process
            debug (bool): include debug printout
            lightEngineName (str): name of the light engine to send commands to
            axisName (str): name of the axis to use as the build platform
            topPosition (float): position to move the build platform to when not in use.
```

```
            swapMinMax (bool): changes which direction is considered down.
        """
        super().__init__(inQueue, outQueue)
        # logging stuff
        self.logEvent = Event()
        self.logQLength = 0
        self.logMsg = ""
        self.resetPrintJobSettings()

    def run(
        self,
        lightEngineName="",
        axisName="",
        topPosition=0,
        bottomPosition=0,
        swapMinMax=False,
        debug=False,
        tempDir="",
    ):
        """
        Validating if the input parameters are correct is left to the ConfigManager.
        """
        self.debug = debug
        self.tempDir = tempDir
        self.lightEngineName = lightEngineName
        self.axisName = axisName
        self.topPosition = topPosition
        self.bottomPosition = bottomPosition
        self.swapMinMax = swapMinMax
        self.nextState = None
```

```
        # start the message threads
        Thread(target=self.stateMachine, name="print_job_state_machine"
            ↪ ).start()
        self.processMessages()

    def messageLogic(self, payload):
        if self.debug:
            print("PrintJobController received payload: ", payload)
        try:
            if isinstance(payload, Shutdown):
                self.shutdown()
            elif isinstance(payload, PrintJobIsRunning):
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.getIsRunning,
                )
            elif isinstance(payload, PrintJobState):
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.
                        ↪ getCurrentStateName
                )
            elif isinstance(payload, PrintJobFolderLocation):
                self.sendResponseMessage(
                    payload.uuid,
                    payload.sender,
                    self.handleSetFolderLocationMessage,
                    payload.path,
                )
            elif isinstance(payload, PrintJobStart):
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.
                        ↪ handleStartMessage
                )
```

```
        elif isinstance(payload, PrintJobStop):
            self.sendResponseMessage(
                payload.uuid, payload.sender, self.handleStopMessage
            )
        elif isinstance(payload, PrintJobPause):
            self.sendResponseMessage(
                payload.uuid, payload.sender, self.handlePauseMessage
            )
        elif isinstance(payload, PrintJobNext):
            self.sendResponseMessage(
                payload.uuid, payload.sender, self.handleNextMessage
            )
        elif isinstance(payload, PrintJobGetCurrentImage):
            self.sendResponseMessage(
                payload.uuid, payload.sender, self.getCurrentImage, payload.publisher
            )
        elif isinstance(payload, PrintJobGetNumberOfLayers):
            if self.printjob is not None:
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.printjob.getNumberOfLayers
                )
            else:
                raise PrintJobCommandError("No print job currently being executed.")
        elif isinstance(payload, PrintJobGetCurrentLayerNumber):
            if self.printjob is not None:
                self.sendResponseMessage(
```

```
                    payload.uuid, payload.sender, self.getCurrentLayerNumber
                )
            else:
                raise PrintJobCommandError("No print job currently being executed.")
        elif isinstance(payload, PrintJobRunTime):
            if self.printjob is not None:
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.getPrintJobRunTime
                )
            else:
                raise PrintJobCommandError("No print job currently being executed.")
        elif isinstance(payload, PrintJobElapsedTime):
            if self.printjob is not None:
                self.sendResponseMessage(
                    payload.uuid, payload.sender, self.getElapsedTime
                )
            else:
                raise PrintJobCommandError("No print job currently being executed.")
        elif isinstance(payload, PrintJobLogMessage):
            self.sendResponseMessage(payload.uuid, payload.sender, self.getLogMessage)
    except Exception as e:
        self.sendMessage(
            CommandStatus(
                payload.uuid,
                payload.sender,
```

```
                    errorState=ErrorState.error,
                    errorMsg="{}: {}".format(type(e).__name__, e.args),
                    traceback=traceback.print_exc(),
                )
            )

    @property
    def currentState(self):
        """
        Getter for currentState

        Isn't wrapped in self._stateLock because that lock often needs
            ↪ to
        be used over larger sections of code.
        """
        return self._currentState

    @currentState.setter
    def currentState(self, newState):
        """
        Setter for currentState

        Also updates the previous state variable
        """
        if isinstance(newState, State):
            self._currentState = newState
        else:
            raise ValueError("Print Job state must be of type
                ↪ PrintJobState")

    def getCurrentStateName(self):
        """
```

```
        Gets the current state name

        Returns:
            str - name
        """
        with self._stateLock:
            return self.currentState.name

    def getCurrentImage(self, publisher):
        """
        Gets the path to the image that the light engine is currently using.
        Returns a blank string if the print job is not running.

        Parameters:
            publisher (PublisherType) - what kind of getter message to send to the light engine

        Returns:
            str - path to image if running, blank otherwise
        """
        if self.getIsRunning():
            output = self.sendCommand(
                LightEngineImage(self.lightEngineName, publisherType=publisher),
                self.getIsRunning,
            )
            # return self.sendCommand(LightEngineImage(self.lightEngineName))
            return output if output is not None else ""
        return ""

```

```
    def getIsRunning(self):
        """
        Gets if a print job is currently running

        Returns:
            bool
        """
        with self._stateLock:
            return self.currentState != State.idle

    def getElapsedTime(self):
        """
        Gets the elapsed time for the print job

        Returns:
            int
        """
        return self.elapsedTime * 1e-3  # convert from ms to seconds

    def handleSetFolderLocationMessage(self, path):
        """
        Handles setting the file path of the print job folder. Checks
            ↪ if it is valid
        """
        if os.path.exists(path):
            files = glob.glob(path + "*")
            if path + "print_settings.json" in files:
                self.printJobFilePath = path
                return
            raise PrintJobCommandError(
                "Print job folder {} does not contain a print_settings.
                    ↪ json file".format(
```

```
                    path
                )
            )
            raise ValueError("Folder path {} does not exist".format(
                ↪ path))

    def handleStartMessage(self):
        """
        Handles state changes when a start message is received

        Passes errors back to the caller function
        """
        # the state must stay the same for this entire transaction
        with self._stateLock:
            if self.currentState == State.idle:
                # setup and validate the print job file
                self.printjob = getPrintJob(
                    self.printJobFilePath, printJobSettingsFileName="
                        ↪ print_settings.json",
                )
                # if hardware is initialized, start leveling process
                if self.getInitHardware():
                    self.currentState = State.start_leveling
                # if init fails, raise error
                else:
                    self.setInitHardware(set=True)
                    self.currentState = State.start_leveling
            else:
                # send the error back to the router
                raise PrintJobCommandError("Start command only starts
                    ↪ print jobs")

```

```
    def handleStopMessage(self):
        # the state must stay the same for this entire transaction
        with self._stateLock:
            if self.currentState == State.start_leveling:
                self.currentState = State.idle
            elif self.currentState == State.leveling:
                self.currentState = State.move_bp_top
            elif self.currentState == State.finish_leveling:
                self.currentState = State.idle
            elif self.currentState == State.move_bp:
                self.currentState = State.move_bp_top
            elif self.currentState == State.expose:
                self.currentState = State.move_bp_top
            elif self.currentState == State.pause:
                self.currentState = State.move_bp_top
            elif self.currentState == State.move_bp_top:
                pass  # when this state finishes executing, it goes to
                    ↪ idle
            else:
                # send the error back to the router
                raise PrintJobCommandError(
                    "Stop command does not work in the {} state".format
                        ↪ (self.currentState)
                )

    def handlePauseMessage(self):
        # the state must stay the same for this entire transaction
        with self._stateLock:
            if self.currentState == State.pause:  # unpause
                self.currentState = self.nextState
            # save what state was next so that we can go to that state
                ↪ when we unpause
```

```
            # self.nextState = self.currentState
            elif self.currentState == State.move_bp:
                self.currentState = State.pause
                # check if the print job as complete
                if self.currentLayerNum > self.printjob.getNumberOfLayers():
                    self.nextState = State.move_bp_top
                else:
                    self.nextState = State.expose
            elif self.currentState == State.expose:
                self.currentState = State.pause
                self.nextState = State.move_bp
            else:
                # send the error back to the router
                raise PrintJobCommandError(
                    "Pause command only works during the print cycle. Try using stop."
                )

    def handleNextMessage(self):
        # the state must stay the same for this entire transaction
        with self._stateLock:
            if self.currentState == State.start_leveling:
                self.currentState = State.leveling
            elif self.currentState == State.leveling:
                self.currentState = State.finish_leveling
            elif self.currentState == State.finish_leveling:
                self.currentState = State.move_bp
            else:
                # send the error back to the router
                raise PrintJobCommandError(
```

```
                    "Next command does not work in the {} state".format(self.currentState)
                )

    def stateMachine(self):
        """
        State machine thread that drives a print job
        """
        while not self.stopEvent.is_set():
            with self._stateLock:
                # just get the state long enough to know what task to perform
                # this also allows for checking for state changes mid task
                state = self.currentState
            if state == State.start_leveling:
                # wait until user acknowledges the warning with a next message
                print(
                    "\nWARNING!: make sure that the printing area is clear and that the build platform is not connected to the Z axis!"
                )
                print(" send 'next' to acknowledge\n")
                self.waitForStateChange(state)
            elif state == State.leveling:
                self.moveAxis(self.bottomPosition, MoveMode.absolute)
                print("\nSend 'next' after the build platform has been leveled.\n")
                self.waitForStateChange(state)
            elif state == State.finish_leveling:
                self.moveAxis(self.topPosition, MoveMode.absolute)
```

```
            print("\nSend 'next' when you are ready to run the print job.\n")
            self.waitForStateChange(state)
            with self._stateLock:
                # only send build platform to the bottom if stop command has not been sent
                if self.currentState == State.move_bp:
                    self.moveAxis(self.bottomPosition, MoveMode.absolute)
        elif state == State.move_bp:
            # stop if out of layers
            if self.currentLayerNum > self.printjob.getNumberOfLayers():
                with self._stateLock:
                    self.currentState = State.move_bp_top
                continue
            # move build platform for the next layer
            self.currentLayer = self.printjob.getLayer(self.currentLayerNum)
            self.currentLayerNum += 1
            self.updateBuildPlatformPosition()
            # update to the expose state
            with self._stateLock:
                # check if paused
                if self.currentState != State.pause:
                    self.currentState = State.expose
                else:
                    self.nextState = State.expose
        elif state == State.expose:
            # update state to finish the print job
            self.performExposures()
            with self._stateLock:
```

```
                    # check if paused
                    if self.currentState != State.pause:
                        self.currentState = State.move_bp
                    else:
                        self.nextState = State.move_bp
            elif state == State.pause:
                self.waitForStateChange(state)
                # time.sleep(self._sleep_duration)
                # with self._stateLock:
                #     if self.currentState != State.pause:
                #         self.currentState = self.nextState
            elif state == State.move_bp_top:
                self.moveAxis(self.topPosition, MoveMode.absolute)
                self.resetPrintJobSettings()
                with self._stateLock:
                    self.currentState = State.idle
            else:  # idle state
                time.sleep(self._sleep_duration)

    def performExposures(self):
        """
        Helper function to do all of the exposures on a single layer.
        """
        for exposure in self.currentLayer.exposures:
            # set the light engine settings
            if exposure.power != self.power:
                self.power = exposure.power
                self.sendCommand(
                    LightEngineBrightness(
                        self.lightEngineName, set=True, brightness=self
                        ↪ .power
                    )
```

```
                )
                # wait before exposure
            time.sleep(exposure.wait_before)
            self.elapsedTime += exposure.wait_before
            # set the image
            self.sendCommand(
                LightEngineImage(
                    self.lightEngineName,
                    set=True,
                    image=self.printJobFilePath + "slices/" + exposure.
                        ↪ image,
                )
            )
            # expose the image
            self.sendCommand(
                LightEnginePerformExposure(self.lightEngineName,
                    ↪ exposure.exposure_time)
            )
            self.elapsedTime += exposure.exposure_time
            # wait after exposure
            time.sleep(exposure.wait_after)
            self.elapsedTime += exposure.wait_after

    def resetPrintJobSettings(self):
        """
        Helper function to reset all of the settings states for the
            ↪ axis and light engine.
        """
        self.printjob = None
        self.elapsedTime = 0
        self.exposure_time = 0
        self.power = 0
```

```
        self.relative_focus_position = 0
        self.wait_before_exposure = 0
        self.wait_after_exposure = 0
        self.logMsg = ""
        self.currentLayerNum = 1
        self.printJobFilePath = ""
        self.logEvent.clear()
        self.logQLength = 0

    def updateBuildPlatformPosition(self):
        """
        Moves the build platform based on the config in the layer
        """
        # wait before moving bp
        time.sleep(self.currentLayer.init_wait)
        self.elapsedTime += self.currentLayer.init_wait
        # move up
        upDistance = self.currentLayer.distance_up * (-1 if self.
            ↪ swapMinMax else 1)
        self.moveAxis(upDistance, MoveMode.relative)
        # wait time at top
        time.sleep(self.currentLayer.up_wait)
        self.elapsedTime += self.currentLayer.up_wait
        # move to the thickness height
        downDistance = (
            self.currentLayer.thickness * (-1 if self.swapMinMax else
                ↪ 1) * 1e-3
            - upDistance
        )
        self.moveAxis(downDistance, MoveMode.relative)
        # wait before moving on
        time.sleep(self.currentLayer.final_wait)
```

```
            self.elapsedTime += self.currentLayer.final_wait

    def shutdown(self):
        self.stopEvent.set()
        print("PrintJobController shutdown")

    def waitForStateChange(self, state):
        """
        Helper function to help states stall

        Parameters:
            state (PrintJobState) - state to compare the current state
                ↪ against
        """
        while not self.stopEvent.is_set():
            with self._stateLock:
                if state != self.currentState:
                    break
            time.sleep(self._sleep_duration)

    def moveAxis(self, pos, mode):
        """
        Helper function for moving the axis

        Parameters:
            pos (float) - position to move the axis to
            mode (MoveMode) - should the pos be interpreted absolutely
                ↪ or relatively

        Return:
            bool - success or failure to move
        """
```

```
        uuid = self.sendMessage(
            AxisPosition(self.axisName, set=True, position=pos, mode=mode)
        )
        response = self.waitForResponse(uuid)
        return response.state != ErrorState.error

    def getInitHardware(self, set=False):
        """
        Helper function that checks if the hardware is initialized

        Initializes all hardware concurrently, using threads.

        Returns:
            bool - success
        """
        axisResult = None
        lightEngineResult = None
        # create hardware threads
        if set:
            axisInitThread = Thread(target=self.initAxis)
            lightEngineInitThread = Thread(target=self.initLightEngine)
        else:
            axisInitThread = Thread(target=self.getAxisInit, args=(axisResult,))
            lightEngineInitThread = Thread(
                target=self.getLightEngineInit, args=(lightEngineResult,)
            )
            # start threads
        axisInitThread.start()
        lightEngineInitThread.start()
```

```
        # wait for threads to join
        axisInitThread.join()
        lightEngineInitThread.join()
        return axisResult and lightEngineResult

    def setInitHardware(self, set=False):
        """
        Helper function that initializes the hardware

        Initializes all hardware concurrently, using threads.
        Returns:
            bool - success
        """
        try:
            axisResult = None
            lightEngineResult = None
            # create hardware threads
            axisInitThread = Thread(target=self.initAxis)
            lightEngineInitThread = Thread(target=self.initLightEngine)
            # start threads
            axisInitThread.start()
            lightEngineInitThread.start()
            # wait for threads to join
            axisInitThread.join()
            lightEngineInitThread.join()
            return True
        except Exception as e:
            print(str(e))
            return False

    def getLightEngineInit(self, result):
        """
```

```
        Gets light engine init state

        Parameters:
            result (bool) - return value

        Returns:
            None - return val is set to result
        """
        # check if the hardware is already iniitialized
        leGetInitUUID = self.sendMessage(LightEngineInitialize(self.
            ↪ lightEngineName))
        leResponse = self.waitForResponse(leGetInitUUID)
        if leResponse.state == ErrorState.error:
            if self.debug:
                raise ValueError(leResponse.errorMsg)
            else:
                raise ValueError(leResponse.traceback)
        result = leResponse.returnVal

    def getAxisInit(self, result):
        """
        Gets the init state of the build platform

        Parameters:
            result (bool) - return value

        Returns:
            None - return val is set to result
        """
        # check if the hardware is already initialized
        axisGetInitUUID = self.sendMessage(AxisInitialize(self.axisName
            ↪ ))
```

```python
        axisResponse = self.waitForResponse(axisGetInitUUID)
        if axisResponse.state == ErrorState.error:
            if self.debug:
                raise ValueError(axisResponse.errorMsg)
            else:
                raise ValueError(axisResponse.traceback)
        result = axisResponse.returnVal

    def initAxis(self):
        """
        Initializes axis hardware
        """
        axisSetInitUUID = self.sendMessage(AxisInitialize(self.axisName, set=True))
        axisResponse = self.waitForResponse(axisSetInitUUID)
        if self.debug:
            print("Print Job - axis response: ", axisResponse)
        if axisResponse.state == ErrorState.error:
            if self.debug:
                raise ValueError(axisResponse.errorMsg)
            else:
                raise ValueError(axisResponse.traceback)
        # home the axis
        axisHomeUUID = self.sendMessage(AxisHome(self.axisName, set=True))
        axisResponse = self.waitForResponse(axisHomeUUID)
        if self.debug:
            print("Print Job - axis response: ", axisResponse)
        if axisResponse.state == ErrorState.error:
            if self.debug:
                raise ValueError(axisResponse.errorMsg)
            else:
```

```
                raise ValueError(axisResponse.traceback)

    def initLightEngine(self):
        """
        Initializes light engine hardware
        """
        leSetInitUUID = self.sendMessage(
            LightEngineInitialize(self.lightEngineName, set=True)
        )
        leResponse = self.waitForResponse(leSetInitUUID)
        if leResponse.state == ErrorState.error:
            if self.debug:
                raise ValueError(leResponse.errorMsg)
            else:
                raise ValueError(leResponse.traceback)

    def getPrintJobRunTime(self):
        """
        Iterates through all of the layers in the print job and
          ↪ calculates the
        total time the print job will take.

        Returns:
            float - number of seconds it will take the print to
              ↪ complete
        """
        if self.printjob is not None:
            totalTime = 0
            for i in range(self.printjob.getNumberOfLayers()):
                layer = self.printjob.getLayer(i + 1)
                totalTime += layer.init_wait
                totalTime += layer.up_wait
```

```
                totalTime += layer.final_wait
                for exposure in layer.exposures:
                    totalTime += exposure.exposure_time
                    totalTime += exposure.wait_before
                    totalTime += exposure.wait_after
            return (
                totalTime / 1000
            )  # total time is in milliseconds. Divide by 1000 to get seconds
        else:
            raise PrintJobCommandError("No print job currently being executed.")

    def getCurrentLayerNumber(self):
        """
        Gets the current layer number

        Returns:
            int - current layer number
        """
        return self.currentLayerNum - 1

    def getLogMessage(self):
        """
        Get log message about the state of the print job controller

        Since it is up to the front end to ask for log messages, there is potential
        for this function to be spammed. Best practice is to queue up all calls to this
        function and periodically return a single log message to all of the calls at the
```

```
        same time.

        Returns:
            str - log message
        """
        with self._stateLock:
            # increment queue size
            self.logQLength += 1
        # wait for new log message
        self.logEvent.wait()
        with self._stateLock:
            # clear the log event  if last call in the queue
            if self.logQLength == 1:
                self.logEvent.clear()
            # take function call off of the queue
            self.logQLength -= 1
        return self.logMsg

    def sendCommand(self, message, cancelCondition=None):
        """
        Sends a message and returns the result. Also handles errors.

        Parameters:
            message (Message) - message to send to the router
            cancelCondition (method) - method that returns a boolean value if to continue or not
                True => keep waiting; False => cancel the command

        Returns:
            any - return value depends on the message
            None - returned if the command was cancelled
        """
```

```
        if isinstance(message, ABC_Message):
            uuid = self.sendMessage(message)
            if cancelCondition is None:
                sts = self.waitForResponse(uuid)
            else:
                while cancelCondition():
                    sts = self.waitForResponse(uuid, timeout=0.1)
                    if sts is not None:
                        break
                # check if the cancelEvent is was caused the while loop to stop
                if not cancelCondition():
                    return None
            if sts is None:
                return None
            if sts.state != ErrorState.none:
                raise PrintJobCommandError(sts.traceback if self.debug else sts.errorMsg)
            return sts.returnVal
```

A.11 server.py

```
from flask import Flask, render_template, Response, request, Blueprint
from flask_restplus import Api, Resource
from flask_cors import CORS
import copy
import time
import uuid
import sys
from threading import Thread, Lock, Event
from src.data_structs.internal_messages import CommandStatus
from src.data_structs import ErrorState, JobQEntry
```

```
from src.errors import MessageError
from src.webserver import flaskapp as app
from src.data_structs import ConfigInterfaces

cm = None
config = None
debug = False
router = None
tempDir = None

# gets rid of an annoying error message in the web
# console when running development server
cors = CORS(app, resources={r"/api/*": {"origins": "*"}})


def initAxesAPI(api):
    """
    Initializes all of the api endpoints for the axes module

    Parameters:
        api (API) - flask_restplus object that handles the api endpoints
    """
    try:
        # check if the axes has been configured. If not, then don't
        # register its api endpoints
        global cm
        cm.getConfig(ConfigInterfaces.Axes)
    except Exception as e:
        print(str(e))
        return

```

```
    from src.webserver.api.hardware.axes import axesNames
    from src.webserver.api.hardware.axes import axesInit
    from src.webserver.api.hardware.axes import axesHome
    from src.webserver.api.hardware.axes import axesPosition
    from src.webserver.api.hardware.axes import axesCalibratedPosition
    from src.webserver.api.hardware.axes import axesMax, axesMin
    from src.webserver.api.hardware.axes import axesAcceleration
    from src.webserver.api.hardware.axes import axesDeceleration
    from src.webserver.api.hardware.axes import axesVelocity
    from src.webserver.api.hardware.axes import axesReset

    api.add_namespace(axesNames)
    api.add_namespace(axesInit)
    api.add_namespace(axesHome)
    api.add_namespace(axesPosition)
    api.add_namespace(axesCalibratedPosition)
    api.add_namespace(axesMin)
    api.add_namespace(axesMax)
    api.add_namespace(axesAcceleration)
    api.add_namespace(axesDeceleration)
    api.add_namespace(axesVelocity)
    api.add_namespace(axesReset)


def initPrintJobAPI(api):
    """
    Initializes all of the api endpoints for the print job controller
        ↪ based on which ones have been
    configured.

    Parameters:
```

```python
        api (API) - flask_restplus object that handles the api endpoints
    """
    try:
        # check if the print job controller has been configured. If not, then don't
        # register its api endpoints
        global cm
        cm.getConfig(ConfigInterfaces.PrintJob)
    except Exception as e:
        print(str(e))
        return

    from src.webserver.api.controllers.printJob import printJobStart
    from src.webserver.api.controllers.printJob import printJobPause
    from src.webserver.api.controllers.printJob import printJobNext
    from src.webserver.api.controllers.printJob import printJobStop
    from src.webserver.api.controllers.printJob import printJobState
    from src.webserver.api.controllers.printJob import printJobIsRunning
    from src.webserver.api.controllers.printJob import printJobValidStates
    from src.webserver.api.controllers.printJob import printJobGetCurrentImage
    from src.webserver.api.controllers.printJob import printJobGetCurrentImageOnChange
    from src.webserver.api.controllers.printJob import printJobGetCurrentImagePeriodic
    from src.webserver.api.controllers.printJob import printJobGetCurrentLayerNumber
    from src.webserver.api.controllers.printJob import printJobGetNumberOfLayers
```

```
    from src.webserver.api.controllers.printJob import printJobRunTime
    from src.webserver.api.controllers.printJob import
        ↪ printJobElapsedTime
    from src.webserver.api.controllers.printJob import
        ↪ printJobLogMessage
    from src.webserver.api.controllers.printJob import printJobUpload

    api.add_namespace(printJobStart)
    api.add_namespace(printJobPause)
    api.add_namespace(printJobNext)
    api.add_namespace(printJobState)
    api.add_namespace(printJobStop)
    api.add_namespace(printJobIsRunning)
    api.add_namespace(printJobValidStates)
    api.add_namespace(printJobGetCurrentLayerNumber)
    api.add_namespace(printJobGetCurrentImage)
    api.add_namespace(printJobGetCurrentImageOnChange)
    api.add_namespace(printJobGetCurrentImagePeriodic)
    api.add_namespace(printJobGetNumberOfLayers)
    api.add_namespace(printJobRunTime)
    api.add_namespace(printJobElapsedTime)
    api.add_namespace(printJobLogMessage)
    api.add_namespace(printJobUpload)


def initLightEnginesAPI(api):
    """
    Initializes all of the api endpoints for the light engines module

    Parameters:
        api (API) - flask_restplus object that handles the api
            ↪ endpoints
```

```
    """
    try:
        # check if the light engines has been configured. If not, then
            ↪ don't
        # register its api endpoints
        global cm
        cm.getConfig(ConfigInterfaces.LightEngines)
    except Exception as e:
        print(str(e))
        return

    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesInit
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesReset
    from src.webserver.api.hardware.light_engines import
        ↪ lightEngineNames
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesBrightness
    from src.webserver.api.hardware.light_engines import brightnessMin,
        ↪  brightnessMax
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesLogging
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesLogMessage
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesImage
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesLED
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesPower
```

```
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesRefreshRate
    from src.webserver.api.hardware.light_engines import refreshRateMin
        ↪ , refreshRateMax
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginesImageDimensions
    from src.webserver.api.hardware.light_engines import
        ↪ lightEnginePerformExposure

    api.add_namespace(lightEnginesInit)
    api.add_namespace(lightEnginesReset)
    api.add_namespace(lightEngineNames)
    api.add_namespace(lightEnginesBrightness)
    api.add_namespace(brightnessMin)
    api.add_namespace(brightnessMax)
    api.add_namespace(lightEnginesLogging)
    api.add_namespace(lightEnginesLogMessage)
    api.add_namespace(lightEnginesImage)
    api.add_namespace(lightEnginesLED)
    api.add_namespace(lightEnginesPower)
    api.add_namespace(lightEnginesRefreshRate)
    api.add_namespace(refreshRateMin)
    api.add_namespace(refreshRateMax)
    api.add_namespace(lightEnginePerformExposure)
    api.add_namespace(lightEnginesImageDimensions)


def initAPI():
    """
    Initializes the API.

    Must be called after the outgoing Queue object has been created.
```

```
    """
    from src.webserver.api import api

    apiBlueprint = Blueprint("api", __name__, url_prefix="/api")
    # initialize the api
    initAxesAPI(api)
    initPrintJobAPI(api)
    initLightEnginesAPI(api)
    # add api to the blueprint
    api.init_app(apiBlueprint)
    # register the api blueprint
    app.register_blueprint(apiBlueprint)


def setup(messageRouter, serverConfig, configManager):
    """
    Sets up the Flask process as the communication hub of the application.

    Parameters:
        inq (dict): Contains all of the Queues that will be handling incoming messages
                    from the various processes. The dicitonary is of the format
                    {<process name>: Queue(), etc.}
        outq (dict): Contains all of the Queues that will be handling outgoing messages
                    to the various processes. The dicitonary is of the format
                    {<process name>: Queue(), etc.}
        configManager (ConfigManager): Used for writing values back to the config file. This means that the
```

```
                        object must already have had a configuration file
                          ↪ loaded.
            flaskConfig (dict): all of the configuation information for the
              ↪ flask web server
        """
        global cm, config, router, tempDir

        # set global variables
        router = messageRouter
        cm = configManager
        tempDir = cm.getConfig(ConfigInterfaces.Router).tempDir()
        config = serverConfig

        # initialize the API
        initAPI()


def getTempDirectory():
        """
        Gets the location of the temporary directory

        Returns:
            str - temp directory location
        """
        global tempDir
        return tempDir


def run():
        """
        Starts the web server and the router.

```

```
    Stops when shutdown() is called.
    """
    global config
    # start the flask server
    # add the threaded option because this should always be running in
        ↪ threaded mode.
    config["threaded"] = True
    app.run(**config)


def send2Process(payload):
    """
    Send a message to a process.

    Acts as a wrapper around the message router

    Parameters:
        payload (dict): message contents. Must match the format
                        that the process is expecting.

    Returns:
        messageStatus (bool) - did the process execute correctly?
        output (any) - response to the original sender of the message.
                        Data type will be variable. In cases where the
                        process did not execute correctly, this will
                        contian the error information.
    """
    uuid = router.sendMessage(payload)
    status = router.waitForResponse(uuid)
    return processCommandStatus(status)
```

```
def processCommandStatus(sts):
    """
    Process a command status message for consumption by another part of the program.

    Parameters:
        sts (CommandStatus) - command status response from another process.

    Returns:
        messageStatus (bool) - did the process execute correctly?
        output (any) - response to the original sender of the message.
                        Data type will be variable. In cases where the
                        process did not execute correctly, this will
                        contian the error information.
    """
    if not isinstance(sts, CommandStatus):
        print(sts)
        raise Exception(
            "Invalid response. Response was not formatted as a CommandStatus object."
        )
    print("status: ", sts)
    if sts.state == ErrorState.none:
        return True, sts.returnVal
    elif debug:
        return False, sts.traceback
    else:
        return False, sts.errorMsg
```

A.12 LightEngineBrightness.py

```
from flask_restplus import Resource, fields, Namespace, reqparse
from src.webserver import send2Process
from src.data_structs.internal_messages.hardware import (
    LightEngineBrightness as Brightness,
)

lightEnginesBrightness = Namespace(
    "light_engines/brightness",
    description="getting and setting the brightness of the light engines",
)

getModelResponse = lightEnginesBrightness.model(
    "response to a command to get the brightness of the light engine",
    {
        "brightness": fields.Float(),
        "valid": fields.Boolean(),
        "errorMsg": fields.String(),
    },
)
setModelResponse = lightEnginesBrightness.model(
    "response to a command to set the brightness of the light engine",
    {"valid": fields.Boolean(), "errorMsg": fields.String(),},
)
setModelParams = lightEnginesBrightness.model(
    "param to set the brightness of the light engine",
    {
        "brightness": fields.Float(
            required=True, example=100, description="brightness value"
        )
    },
```

```
)


@lightEnginesBrightness.route("/<string:lightEngineName>")
@lightEnginesBrightness.param("lightEngineName", "The name of the light engine")
class LightEngineBrightness(Resource):
    """
    API to get and set the brightness of different light engines.
    """

    @lightEnginesBrightness.marshal_with(getModelResponse)
    def get(self, lightEngineName):
        """
        Get brightness value.
        Must be performed after the light engine power is on
        """
        try:
            valid, response = send2Process(Brightness(lightEngineName))
            return (
                {"brightness": response, "valid": valid, "errorMsg": str(response)},
                200,
            )
        except Exception:
            return 500

    @lightEnginesBrightness.expect(setModelParams)
    @lightEnginesBrightness.marshal_with(setModelResponse)
    def post(self, lightEngineName):
        """
        Set the brightness value.
```

```
        Must be performed after the light engine has been initialized
            ↪ and the power is on.
        """
        try:
            parser = reqparse.RequestParser()
            parser.add_argument(
                "brightness", type=int, help="brightness value to set
                    ↪ the light engine to"
            )
            args = parser.parse_args()

            valid, response = send2Process(
                Brightness(lightEngineName, set=True, brightness=args["
                    ↪ brightness"])
            )
            return {"valid": valid, "errorMsg": response}, 200
        except Exception as e:
            print(str(e))
            return 500
```

A.13 ConfigManager.py

```
from jsonschema import Draft7Validator, validate, RefResolver
import json
import os
import copy
import enum
from threading import Lock
from src.data_structs import ConfigInterfaces
from src.config.controllers import PrintJobConfig
from src.config.hardware.axes import AxesInterfaceConfig
```

```
from src.config.hardware.light_engines import LightEnginesInterfaceConfig
from src.config.controllers import RouterConfig, PrintJobConfig
import subprocess


class ConfigManager:
    """
    Handles the initial parsing and validation of config files against the JSON schema. Also produces
    interface specific configs.

    Any additional validation should be done inside individual config classes

    Attributes:
        _interfaces (dict) - lookup table for quick access to the different interface config classes.
        config (dict) - all configuration settings, validated through jsonschema.
    """

    _interfaces = {}
    _configFileLock = Lock()

    def __init__(self, configFilePath):
        """
        Loads config file and validates it against the schema.

        Parameters:
            configFilePath (str) - path to the config file
        """
```

```
        self.configFilePath = configFilePath
        self.path = os.path.abspath(os.path.dirname(__file__)) + "/"
        with open(self.path + "schema/config_schema.json", "r") as f:
            self.schema = json.load(f)
        with open(configFilePath, "r") as f:
            self.config = json.load(f)

        # create resolver to handle jsonschema $ref statements
        self.resolver = RefResolver("file://%s" % self.path + "schema/"
            ↪ , None)
        validate(self.config, self.schema, resolver=self.resolver)

        self.createConfigs()

    def createConfigs(self):
        """
        Creates all of the top level Config objects for the config file
            ↪ .
        """
        if self.axes() is not None:
            self._interfaces[ConfigInterfaces.Axes] =
                ↪ AxesInterfaceConfig(self.config)
        if self.light_engines() is not None:
            self._interfaces[ConfigInterfaces.LightEngines] =
                ↪ LightEnginesInterfaceConfig(
                self.config
            )
        if self.router() is not None:
            self._interfaces[ConfigInterfaces.Router] = RouterConfig(
                ↪ self.config)
        if self.printJob() is not None:
```

```python
            self._interfaces[ConfigInterfaces.PrintJob] = PrintJobConfig(
                self.config,
                AxesInterfaceConfig(self.config),
                LightEnginesInterfaceConfig(self.config),
            )

    def getConfig(self, configInterface):
        """
        Gets an interface config.

        Returns:
            Config/None - interface config or None, depending on if the Config was specified
        """
        if not isinstance(configInterface, ConfigInterfaces):
            raise ValueError(
                "ConfigManager.createConfig only accepts the ConfigInterfaces enum data type"
            )
        return self._interfaces.get(configInterface)

    def axes(self):
        return self.config.get("Axes")

    def light_engines(self):
        return self.config.get("LightEngines")

    def general(self):
        return self.config["General"]

    def router(self):
```

```
        return self.config.get("Router")

    def printJob(self):
        return self.config.get("PrintJob")

    def resolveSchema(self, value):
        """
        Resolves the value of a given JSON schema $ref

        Parameters:
            value (str) - valid ref string

        Returns:
            (dict) - python dictionary of the resolved JSON schema
        """
        path, ref = self.resolver.resolve(value)
        return ref

    def buildSchema(self, schema):
        """
        Recursive function that resolve all schema files into a single schema file for
        the creation of documentation.

        Parameters:
            schema (dict) - python dictionary of valid JSON schema

        Returns:
            (dict) - valid JSON schema with the JSON schemas at refs explicitly added in
        """
        output = copy.deepcopy(schema)
```

```
        for key, value in schema.items():
            # print(key)
            if key == "$ref":
                replacement = self.buildSchema(self.resolveSchema(value
                    ↪ ))
                del output[key]
                output.update(replacement)
            elif isinstance(value, dict):
                replacement = self.buildSchema(value)
                output[key] = replacement
            elif isinstance(value, list):
                if len(value) > 0:
                    if isinstance(value[0], dict):
                        for index, i in enumerate(value):
                            replacement = self.buildSchema(i)
                            output[key][index] = replacement
        return output

    def createSingleFileConfig(self):
        """
        Creates a single config file with all of the JSON schema
            ↪ references resolved.

        Used for easy creation of documentation of the config file.
        """
        output = self.buildSchema(self.schema)
        with open(
            self.path + "schema/single_file_config_for_documentation.
                ↪ json", "w"
        ) as file:
            json.dump(output, file, indent=2)

```

```
    def generateSchemaDocumentation(self, schemaFilePath, outputFilePath):
        self.createSingleFileConfig()
        genDocsCommand = "bootprint json-schema {} {}".format(
            schemaFilePath, outputFilePath
        )
        try:
            subprocess.run(genDocsCommand, check=True, shell=True)
        except Exception as e:
            print(str(e))

    def saveAxisCalibrationPosition(self, axisName, calibratedPosition):
        """
        Saves the calibrated position of an axis to the config file.

        Parameters:
            axisName (string): name of the axis to assign the value to
            calibratedPosition (float): value to save to the config file.
        """
        newConfig = self.getConfig(ConfigInterfaces.Axes).setCalibrationPosition(
            axisName, calibratedPosition
        )
        self.saveToConfigFile(newConfig)

    def saveToConfigFile(self, config):
        """
        Saves a modified config back to the config file

        Parameters:
```

```
            config (dict) - modified config to save to the config file
        """
        with self._configFileLock:
            with open(self.configFilePath, "w") as file:
                json.dump(config, file, indent=4)
```

A.14 config_schema.json

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "id": "root",
    "type": "object",
    "properties": {
        "General": {
            "$comment": "properties that apply generally to the system
                ↪ software",
            "type": "object",
            "properties": {
                "name": {
                    "$comment": "name of the config file",
                    "type": "string"
                },
                "comment": {
                    "$comment": "for long form explanation of the
                        ↪ config file",
                    "type": "string"
                },
                "debug-all": {
                    "$comment": "enables global debug printout.
                        ↪ Overriden by local debug levels",
                    "type": "boolean"
                }
```

```
            },
            "additionalProperties": false,
            "required": [
                "name",
                "debug-all"
            ]
        },
        "Axes": {
            "$comment": "configuration info for the AxesInterface",
            "type": "object",
            "properties": {
                "comms-debug": {
                    "$comment": "debugging for messages being sent between the Router and the AxesInterface",
                    "type": "boolean"
                },
                "drivers": {
                    "$comment": "config info for each of the drivers and axes/shims",
                    "type": "array",
                    "items": {
                        "$ref": "axes/axes_driver_schema.json#/driver"
                    },
                    "uniqueItems": true
                }
            },
            "additionalItems": false,
            "required": [
                "drivers"
            ]
        },
        "LightEngines": {
```

```
            "$comment": "configuration info for the LightEnginesInterface",
            "type": "object",
            "properties": {
                "comms-debug": {
                    "$comment": "debugging for messages being sent between the Router and the LightEnginesInterface",
                    "type": "boolean"
                },
                "drivers": {
                    "$comment": "config info for each of the drivers",
                    "type": "array",
                    "items": {
                        "$ref": "light_engines/light_engines_driver_schema.json#/driver"
                    },
                    "uniqueItems": true
                }
            },
            "additionalItems": false,
            "required": [
                "drivers"
            ]
        },
        "Router": {
            "$comment": "config info the router and flask web server",
            "type": "object",
            "properties": {
                "debug": {
                    "$comment": "debug level for the router",
                    "type": "boolean"
```

```
        },
        "server configuration": {
            "$comment": "configuration for the flask server",
            "type": "object",
            "properties": {
                "host": {
                    "$comment": "hostname for the flask server.
                       ↪  Usually some variation of localhost
                       ↪ or 0.0.0.0",
                    "type": "string",
                    "format": "hostname"
                },
                "port": {
                    "$comment": "port to serve the flask
                       ↪ webserver on",
                    "type": "integer",
                    "minimum": 0,
                    "maximum": 65535,
                    "default": 5000
                },
                "debug": {
                    "$comment": "debug output for the flask
                       ↪ server",
                    "type": "boolean"
                }
            },
            "additionalItems": false,
            "required": [
                "host",
                "port"
            ]
        }
```

```
            },
            "additionalItems": false,
            "required": [
                "server configuration"
            ]
        },
        "PrintJob": {
            "$comment": "config info for the print job controller",
            "type": "object",
            "properties": {
                "comms-debug": {
                    "type": "boolean"
                },
                "light engine name": {
                    "type": "string"
                },
                "build platform axis name": {
                    "type": "string"
                },
                "build platform axis top position": {
                    "type": "number"
                },
                "build platform axis bottom position": {
                    "type": "number"
                },
                "build platform axis swap min/max": {
                    "type": "boolean"
                }
            },
            "additionalProperties": false,
            "required": [
                "light engine name",
```

```
                    "build platform axis name",
                    "build platform axis top position",
                    "build platform axis bottom position",
                    "build platform axis swap min/max"
                ]
            }
    },
    "required": [
        "General"
    ],
    "additionalProperties": false
}
```

A.15 AxesInterfaceConfig.py

```
from src.config import ABC_Config
from src.config.hardware.axes.axesDrivers import (
    AxisDummyDriverConfig,
    TipTiltDriverConfig,
)


class AxesInterfaceConfig(ABC_Config):
    """
    Interface for retrieving the configuration information needed to configure the AxesInterface class
    through its run() function.

    Documentation for undocumented functions can be found inside the Interface abstract base class.

    Attributes:
```

```
        _driverTypes (dict) - keeps track of all of the different drivers
    """

    _driverTypes = {  # driver key names should match the enum defined for the class property in axes_driver_schema.json
        "AxisDummyDriver": AxisDummyDriverConfig,
        "TipTiltDriver": TipTiltDriverConfig,
    }

    def __init__(self, config):
        super().__init__(config)
        self.driverConfigs = []
        for driver in self.drivers():
            driverName = driver["name"]
            driverClassName = driver["class"]
            driverConfigClass = self._driverTypes[driverClassName](
                driverName, self._config
            )
            self.driverConfigs.append(driverConfigClass)

    def getArguments(self):
        return {"debug": self.debug(), "axisDrivers": self.driverConfigs}

    def getConfig(self):
        return self._config.get("Axes")

    def drivers(self):
        return self.getConfig().get("drivers")

    def getDriverConfigs(self):
```

```
        return self.driverConfigs

    def shims(self):
        output = []
        for driver in self.driverConfigs:
            output += driver.getShims()
        return output

    def debug(self):
        """
        Use global default, unless otherwise specified.

        Returns:
            bool - set debug mode
        """
        output = self.globalDebug()
        localDebug = self.getConfig().get("comms-debug")
        # give priority to local debug over global if defined
        if localDebug is not None:
            output = localDebug
        return output

    def __str__(self):
        return str(self.getArguments())

    def setCalibrationPosition(self, axisName, calibratedPosition):
        """
        Finds the given axis and has it save over the config file with
           ↪ the
        calibration position parameter set.

        Parameters:
```

```
            axixName (str) - name of the axis to save the calibration
                ↪ position to
            calibratedPosition (str) - position to save
        """
        for shim in self.shims():
            # find the shim
            if shim.getName() == axisName:
                return shim.updateCalibratedPosition(calibratedPosition
                    ↪ )
        raise ValueError("Axis {} not found".format(axisName))
```

A.16 ABC_AxesDriverConfig.py

```
import copy
from src.config.hardware.axes.axesShims import AxisDummyShimConfig,
    ↪ TipTiltAxisShimConfig
from src.config import ABC_Config


class ABC_AxesDriverConfig(ABC_Config):
    """
    Defines basic access functions for an AxisDriver.

    Attributes:
        _axes (dict) - lookup table for which Configs go with which
            ↪ Shims.
        driver (dict) - reference to the part of the config file that
            ↪ contains the
            config info for this particular driver.
        shims (list of AxesShimConfig) - list of all of the shim
            ↪ configs associated
            with this driver.
```

```
    """

    # what is called an axis in the config file is call a axis shim in the code
    _axes = {"AxisDummyShim": AxisDummyShimConfig, "TipTiltShim": TipTiltAxisShimConfig}

    def __init__(self, name, config):
        """
        Finds and creates the configuration.

        Parameters:
            name (str) - name of the driver.
            config (dict) - dictionary representation of the entire JSON config file.
        """
        # create the config info
        super().__init__(config)

        # find the specific driver
        self.shims = []
        drivers = self.getDrivers()
        for driver in drivers:
            if driver["name"] == name:
                self.driver = copy.deepcopy(driver)
                # create all of the axis Configs
                for axis in driver["axes"]:
                    axisConfig = self._axes[axis["class"]](
                        name, axis["name"], self._config
                    )
                    self.shims.append(axisConfig)
                break
```

```

    def getClassName(self):
        """
        Gets the name of the driver class that this configuration is intended for.

        Returns:
            str - class name
        """
        return self.driver["class"]

    def getName(self):
        """
        Gets the name of the driver that this configuration is intended for.

        Returns:
            str - driver name
        """
        return self.driver["name"]

    def getDrivers(self):
        """
        Gets the dictionary objects for all of the drivers

        Returns:
            (list of dict)
        """
        return self._config["Axes"]["drivers"]

    def getConfiguration(self):
        """
```

```
        Gets the configuration information for the specific driver

        Returns:
            dict - config info
        """
        return self.driver["configuration"]

    def getAxes(self):
        """
        Gets all of the axes config info that are associated with this driver

        Returns:
            (list of dict)
        """
        return self.driver["axes"]

    def getDebug(self):
        """
        Gets the debug level for the driver. Gives priority to local debug definition over global.

        Returns:
            bool
        """
        if self.driver.get("debug") is not None:
            return self.driver["debug"]
        else:
            return self.globalDebug()

    def getShims(self):
        """
```

```
        Gets all of the shim Config objects

        Returns:
            (list of AxisShimConfig)
        """
        return self.shims
```

A.17 AxisDummyDriverConfig.py

```
from src.config.hardware.axes.axesDrivers import ABC_AxesDriverConfig


class AxisDummyDriverConfig(ABC_AxesDriverConfig):
    """
    Handles formatting the configuration data for the DummyDriver
    """

    def getArguments(self):
        """
        Gets the arguments that the DummyDriver needs to be initialized

        Returns:
            driverConfigs (dict) - kwargs for DummyDriver.__init__()
            shims (list of AxisShimConfig) - Config objects for the shims that are associated
                with this driver.
        """
        driverConfigs = {}
        if self.getConfiguration().get("acceleration") is not None:
            driverConfigs["acceleration"] = self.getConfiguration().get("acceleration")
        if self.getConfiguration().get("deceleration") is not None:
```

```
            driverConfigs["deceleration"] = self.getConfiguration().get
                ↪ ("deceleration")
        if self.getConfiguration().get("velocity") is not None:
            driverConfigs["velocity"] = self.getConfiguration().get("
                ↪ velocity")
        if self.getConfiguration().get("maxPos") is not None:
            driverConfigs["maxPos"] = self.getConfiguration().get("
                ↪ maxPos")
        if self.getConfiguration().get("minPos") is not None:
            driverConfigs["minPos"] = self.getConfiguration().get("
                ↪ minPos")
        return driverConfigs, self.getShims()

    def __str__(self):
        return str(self.driver)
```

A.18 axes_driver_schema.json

```
{
    "$schema": "http://json-schema.org/schema#",
    "id": "axes_drivers",
    "description": "schemas for all of the drivers",
    "driver": {
        "$comment": "config info and options for axis drivers",
        "type": "object",
        "properties": {
            "name": {
                "$comment": "name that will be used to refer to the
                    ↪ axis throughout the code",
                "type": "string",
                "pattern": "^[a-zA-Z0-9_]+$"
            },
```

```
            "class": {
                "$comment": "class to use as the driver",
                "type": "string",
                "enum": [
                    "AxisDummyDriver",
                    "TipTiltDriver"
                ]
            },
            "configuration": {
                "$comment": "variables to configure the driver with. Register the file that contains theses parameters here.",
                "anyOf": [
                    {
                        "$ref": "axes_driver_schemas/axes_dummy_driver.json#/dummy-driver"
                    },
                    {
                        "$ref": "axes_driver_schemas/tip_tilt_driver.json#/driver"
                    }
                ]
            },
            "debug": {
                "$comment": "enable/disable printout info for the driver. Overrides the global setting.",
                "type": "boolean"
            },
            "axes": {
                "$comment": "list of the config info for all of the axes associated with this driver.",
                "type": "array",
```

```
                    "items": {
                        "$ref": "axes_shim_schema.json#/shim"
                    },
                    "uniqueItems": true
                }
            },
            "additionalItems": false,
            "required": [
                "name",
                "class",
                "configuration",
                "axes"
            ]
        }
}
```

A.19 axes_dummy_driver.json

```
{
    "$schema": "http://json-schema.org/schema#",
    "id": "axes-dummy-driver",
    "dummy-driver": {
        "$comment": "config parameters for the axis dummy driver",
        "type": "object",
        "properties": {
            "acceleration": {
                "$comment": "default acceleration value",
                "type": "number",
                "exclusiveMinimum": 0
            },
            "deceleration": {
                "$comment": "default deceleration value",
```

```
                "type": "number",
                "exclusiveMinimum": 0
            },
            "velocity": {
                "$comment": "default velocity value",
                "type": "number",
                "exclusiveMinimum": 0
            },
            "maxPos": {
                "$comment": "maximum valid position",
                "type": "number",
                "exclusiveMinimum": 0
            },
            "minPos": {
                "$comment": "minimum valid position",
                "type": "number",
                "minimum": 0
            }
        },
        "additionalProperties": false
    }
}
```

A.20 grbl_test.py

```
from src.hardware.axes.drivers import GrblDriver
from src.hardware.axes import GrblAxisShim
from src.data_structs import MoveMode
import time

"""
```

```
This is a simple test of the GrblDriver and GrblAxis_v0_9 axis interface.
It should not be used for thorough testing.

It was originally tested on the HR1 Solus mechanism.
"""
# driver = GrblDriver(numOfAxes=1, verbose=True)
driver = GrblDriver(numOfAxes=1)
axisZ = GrblAxisShim(driver=driver, grblAxisName="Z")
axisX = GrblAxisShim(driver=driver, grblAxisName="X")
axisZ.initialize()
axisZ.home()
print("Current Position: ", axisZ.getPosition())

print("Moving the printer")
axisZ.setPosition(-5.0, MoveMode.absolute)
axisX.setPosition(-1.0, MoveMode.absolute)
print("Current Position - Z: ", axisZ.getPosition())
print("Current Position - X: ", axisX.getPosition())

# print("Max position: ", axisZ.getMaxPosition())

# print("Acceleration: ", axisZ.getAcceleration())
# axisZ.setAcceleration(10.0)
# print("Acceleration: ", axisZ.getAcceleration())
# axisZ.setAcceleration(100.0)
# print("Acceleration: ", axisZ.getAcceleration())

# print("Velocity: ", axisZ.getVelocity())
# axisZ.setVelocity(80.0)
# print("Velocity: ", axisZ.getVelocity())
# axisZ.setVelocity(800.0)
```

```
# print("Velocity: ", axisZ.getVelocity())

axisZ.reset_driver()

axisX.initialize()
axisX.home()
```

A.21 test_MessageRouter.py

```
import unittest
from threading import Thread
import time
import os
from multiprocessing import Queue, Process
from src.config import ConfigManager
from src.process_interfaces.controllers import MessageRouter
from src.data_structs.internal_messages import (
    Shutdown,
    CommandStatus,
)
from src.data_structs.internal_messages.hardware import AxesNames, ABC_AxisMessage
from src.data_structs.internal_messages.controllers import SaveCalibratedPositionToConfig


class TestMessageRouter(unittest.TestCase):
    """
    Class for testing the MessageRouter controller.

    Imitates two processes talking to each other through the MessageRouter.
```

```
    """

    dummyPath = (
        os.path.abspath(os.path.dirname(__file__))
        + "/../../../config_files/dummy_config.json"
    )
    wait = 0.1

    def setUp(self):
        """
        Creates a MessageRouter and the message queues for sending test
            ↪ messages to it.
        """
        self.inq = {}
        self.outq = {}
        self.inq[ABC_AxisMessage.destination] = Queue()
        self.outq[ABC_AxisMessage.destination] = Queue()
        self.inq["proc"] = Queue()
        self.outq["proc"] = Queue()
        self.cm = ConfigManager(self.dummyPath)
        self.router = MessageRouter(self.inq, self.outq)
        Thread(
            target=self.router.run, kwargs={"configManager": self.cm, "
                ↪ debug": True}
        ).start()

    def tearDown(self):
        """
        Shutdown the MessageRouter
        """
        self.router.shutdown()
        for key, value in self.outq.items():
```

```
            payload = value.get(timeout=0.1)
            self.assertIsInstance(payload, Shutdown)

    def test_sendMessage(self):
        """
        Sends a basic message from one process to the other.
        """
        msg = AxesNames()
        # send a message to the axes
        self.inq["proc"].put(msg)
        payload = self.outq[ABC_AxisMessage.destination].get(timeout=self.wait)
        # check if we received the message from the axes
        self.assertIsInstance(payload, AxesNames)
        # send a CommandStatus back to the proc
        self.inq[ABC_AxisMessage.destination].put(
            CommandStatus(payload.uuid, payload.sender)
        )
        payload = self.outq["proc"].get(timeout=self.wait)
        # check if proc got the message
        self.assertIsInstance(payload, CommandStatus)
        self.assertEqual(msg.uuid, payload.uuid)

    def test_messageForRouter(self):
        msg = SaveCalibratedPositionToConfig("Y", 1.0)
        # send message to the message router
        self.inq[ABC_AxisMessage.destination].put(msg)
        # sleep to give time for the router to do it's thing
        time.sleep(self.wait)
        # check that none of the other queues have messages
        for _, value in self.inq.items():
            print("testing {} queue".format(_))
```

```
                self.assertEqual(value.qsize(), 0)
        for _, value in self.outq.items():
                print("testing {} queue".format(_))
                self.assertEqual(value.qsize(), 0)

    def test_apiMessage(self):
        msg = AxesNames()
        # use the message router like a regular process interface
        uuid = self.router.sendMessage(msg)
        # get the message and send a response back
        payload = self.outq[ABC_AxisMessage.destination].get(timeout=self.wait)
        self.inq[ABC_AxisMessage.destination].put(
            CommandStatus(payload.uuid, payload.sender)
        )
        retval = self.router.waitForResponse(uuid, timeout=self.wait)
        # make sure the function did not time out
        self.assertIsNotNone(retval)
        # check for how bad keys are handled
        with self.assertRaises(KeyError):
            retval = self.router.waitForResponse(uuid, timeout=2)
```

A.22 Publisher.py

```
from threading import Event, Lock, Thread
import time
from src.data_structs import PublisherType


def publisher(target):
    """
```

```
    Determines the publishing behavior of a getter method. (i.e. on change or periodic)

    Parameters:
        target (method) - decorated method
    """

    def deco(function):
        def inner(self, *args, **kwargs):
            # get the Publisher object from the class
            if getattr(self, target) is not None:
                publisher = getattr(self, target)
                # validate that the type of the arg is correct
                if isinstance(args[0], PublisherType):
                    if args[0] == PublisherType.none:
                        return function(self, *args, **kwargs)
                    elif args[0] == PublisherType.onChange:
                        publisher.waitForOnChangeEvent()  # wait for a change in the variable before calling the function
                        output = function(self, *args, **kwargs)
                        return output
                    else:
                        publisher.waitForPeriodicEvent()  # wait for periodic event to call the function
                        return function(self, *args, **kwargs)
                else:
                    ValueError(
                        "getter methods decorated with @publisher only accept PublisherType values for their first arg"
                    )
```

```

        return inner

    return deco


class Publisher:
    """
    Used in conjunction with an API getter function to turn it into a publisher of data.

    When a member wants to subscribe to the publisher, it sends an API call to the getter
    and it is put into a queue by the publisher until the monitored data variable is updated
    or a predefined time as elapsed. At that point in time, the publisher will service all of
    the API requests in the queue and they will return back to the caller. Finally, when the
    API caller receives the request response, it will do what even it needs to with that data
    then immediately send another API request to get another spot in the queue.

    This class supports two modes of operation: publish on change, and periodic publishing.

    This class contains the queue, event handlers and the logic for updating the queue.

    Parameters:
```

```
        period (float) - time in seconds for how often to publish updates when using periodic publishing
        onChangeEvent (Event) - event that gets set whenever the variable changes
        onPeriodicEvent (Event) - event that gets on a periodic basis
        periodicQLength (int) - length of the queue
        onChangeQLength (int) - length of the queue
        periodicLock (Lock) - lock for the periodic queue
        onChangeLock (Lock) - lock for the on change queue
    """

    def __init__(self, period):
        self.period = period
        self.periodicEvent = Event()
        self.onChangeEvent = Event()
        self.periodicQLength = 0
        self.onChangeQLength = 0
        self.periodicLock = Lock()
        self.onChangeLock = Lock()

        # period publishing thread
        self.periodicPublishThread = Thread(target=self.periodicPublish)
        self.periodicPublishThread.setDaemon(True)
        self.periodicPublishThread.start()

    def periodicPublish(self):
        """
        Releases all of the requests from the periodic queue
        """
        while True:
            with self.periodicLock:
```

```
                # only set the event if the queue if full
                if self.periodicQLength > 0:
                    self.periodicEvent.set()
            time.sleep(self.period)

    def incPeriodicQueue(self):
        """
        Puts a request in the periodic queue
        """
        with self.periodicLock:
            self.periodicQLength += 1

    def decPeriodicQueue(self):
        """
        Removes a request in the periodic queue
        """
        with self.periodicLock:
            self.periodicQLength -= 1
            # clear the event if last member of the queue
            if self.periodicQLength == 0:
                self.periodicEvent.clear()

    def incOnChangeQueue(self):
        """
        Puts a request in the on change queue
        """
        with self.onChangeLock:
            self.onChangeQLength += 1

    def decOnChangeQueue(self):
        """
        Removes a request in the on change queue
```

```
        """
        with self.onChangeLock:
            self.onChangeQLength -= 1
            # clear the event if last member of the queue
            if self.onChangeQLength == 0:
                self.onChangeEvent.clear()

    def setChangePublish(self):
        """
        Sets the change event.
        """
        with self.onChangeLock:
            # only set event if the queue is full
            if self.onChangeQLength > 0:
                self.onChangeEvent.set()

    def waitForPeriodicEvent(self):
        """
        Waits for a periodic event
        """
        self.incPeriodicQueue()
        self.periodicEvent.wait()
        self.decPeriodicQueue()

    def waitForOnChangeEvent(self):
        """
        Waits for a on change event
        """
        self.incOnChangeQueue()
        self.onChangeEvent.wait()
        self.decOnChangeQueue()
```