Bachelor’s thesis

Information and Communications Technology

2023

Louis Lautz

Exploring Flexibility and Extensibility through a Flashing Tool Implementation

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2023 | 38 pages

Louis Lautz

Exploring Flexibility and Extensibility through a Flashing Tool Implementation

In May of 2022, the German company Pareva GmbH commissioned the development of a flash tool for their main product, that improves their current flash process. A flash tool is used to install the software or firmware onto the hardware of a device.

This thesis provides insight into the planning, design and implementation process, as well as an in depth look into the concepts of flexibility and extensibility and how they relate to the flash tool. Throughout the thesis, the MoSCoW-method was used, to categorize features into 4 categories, based on their priority. Different features were used as examples to show how to design or implement a feature with flexibility and extensibility in mind.

As of May 2023, the project has been successfully completed and Pareva GmbH is using it on a regular basis without issues so far.

Keywords:

Extensibility, Flexibility, Flash Tool, Coupling, Separation of Concerns, Cohesion

Contents

Figures 4

Glossary 5

1 Introduction 6

2 Methods and Technologies 8

2.1 Tkinter and Customtkinter 8

2.2 Device Explanation 8

2.3 Flash adapters 10

2.3.1 Slave Controller Adapter 11

2.3.2 Master Controller Adapter 12

3 Requirements 13

3.1 Previous Flash process 13

3.2 Requirement Priorities 13

3.2.1 Must have 14

3.2.2 Should have 14

3.2.3 Could have 15

3.2.4 Won’t have 15

4 Architecture 17

4.1 Architecture of flash tool 17

5 Implementations 19

5.1 Implementation of flash tool structure 19

5.2 Flexibility 24

5.2.1 Flexibility seen as resistance to change 24

5.2.2 Coupling 25

5.2.3 Temporal Coupling 26

5.2.4 Pros and Cons of Flexibility 27

5.3 Extensibility 28

5.3.1 Separation of Concerns 29

5.3.2 Cohesion 29

6 Testing 30

6.1 Must have requirements 30

6.2 Should have requirements 30

6.3 Could have requirements 31

7 Conclusion 32

Appendices 36

Figures

Figure 1. Lock Controller. 10

Figure 2. ARM Cortex Cable. 11

Figure 3. Slave Controller Adapter. 12

Figure 4. Master Controller Adapter. 12

Figure 5. Flash Tool GUI. 18

Figure 6. Download Status Tags. 20

Figure 7. Flash Tool Architecture Diagram. 23

Glossary

API	Application Programming Interfaces are defined by a set of rules and protocols that allow different software programs to communicate and interact with each other, enabling the exchange of data and services. [1]
GUI	Graphical User Interfaces are visual interfaces, that are used to interact with applications or webpages. They render different visual elements to display information or let the user take actions. [2]

1 Introduction

Flash tools are applications, which are used to install software or firmware onto hardware. This makes them crucial for the manufacturing process of embedded system devices.

Pareva is a German company, that works closely together with the Finnish company Punta. Together, they build smart lockers, that have integrated remote opening methods.
The remarkable property about Pareva's software solutions for their various customers is, that the software for all systems is identical. Yet, every customer can have a customized solution. This service can be achieved with an extremely flexible and extensible software architecture. Flexibility and extensibility play a major role in Pareva's software design.
Having mentioned that, it was clear that a flash tool for them must reflect and support these characteristics.
Pareva's old flash tool had many issues, which motivated them to commission a new and improved flash tool.

Flexibility and extensibility in general are well understood concepts and there is a lot of literature about them. In this thesis, recent publications on these topics are used. There are also less recent works, showing that these practices are well explored. One example of an older book exploring flexibility and extensibility is "*Design Patterns Elements of Reusable Object-Oriented Software*" from 1994. [3]

The problem with many publications about these topics are, that they are too general. The examples rarely involve more than a few functions or classes. Usually that is sufficient to convey the theoretical practices, but it fails to portray flexibility and extensibility on a larger scale.
In contrast to that, this thesis aims to highlight flexibility and extensibility at a greater scale, by using a large project and giving insight into the design process with flexibility and extensibility in mind.

The main focus in the thesis is to give an insight into the design of the flash tool. Chapter 2 “Methods and Technologies”, discusses different technologies that are used in thesis.
In Chapter 3 “Requirements” the MoSCoW-method [4] is established and all features are categorized according to their priority.
Chapter 4, “Architecture”, discusses how different components are arranged and interact with each other. It also contains a diagram to further visualize the relationships between components.
Chapter 5, “Implementations”, deals with the concrete coding techniques used to implement the components. In addition to that, it dives deeper into the topics of flexibility and extensibility and uses implementations used in the flash tool to explain the discussed techniques.
Lastly, Chapter 6, “Testing”, circles back to the MoSCoW categories established in Chapter 3 and confirms that all features are implemented and work as intended.

The author of this thesis created the flash tool single-handedly for Pareva GmbH in a time frame of 1 year and is also responsible for the writing of the thesis.

2 Methods and Technologies

2.1 Tkinter and Customtkinter

Tkinter is a Graphical User Interface (GUI) library in the python standard library. It allows the developer to easily create cross-platform GUIs that can be rendered on most Unix systems, such as macOS, but also on Windows and Linux. [5]

Customtkinter is a wrapper for Tkinter, which is almost a drop-in replacement for Tkinter. Unlike Tkinter though, it looks visually appealing out of the box and is easier to design. Visual properties of elements can be customized more easily. There is also a theme handler, with which the color profile can be adjusted for the entire application. Most Tkinter elements also exist in Customtkinter, which means that elements can quickly be switched out for Tkinter or Customtkinter elements. [6]

2.2 Device Explanation

The devices that need to be flashed are lock controllers in lockers. This allows the user to open locks via an Application Programming Interface (API). The lockers are used for parcel lockers, personal locker and much more. Because they interact with an API, there are many different user interfaces to interact with the lock. Some locks have a card reader, others use keypads with access codes and some even use their own applications to open the locks.

Each lock controller (Figure 1) has 12 output pins, meaning it can open 12 locks per controller or control other things like lights or a built-in scale. The output pins are the white vertical rectangles at the bottom of the picture.

Each controller has an ethernet port for the internet connection and a second ethernet port for chaining multiple controllers together. These ports can be seen in the middle of the top edge of the device.

A controller can take on two different roles. The master controller communicates with the API and needs to be connected to the Internet. Additionally, it can also send commands to other connected slave controllers. The slave controller has to be connected to a master controller via ethernet cable and works as an extension of the master controller. This way, one master controller can control multiple sets of 12 locks.

Every controller has an Atmel Atmega microcontroller [7]. It connects to all 12 lock output pins and is responsible for opening the locks when the appropriate signal is given.

In addition to the Atmega microcontroller, only master controllers have an ESP32. ESP32 chips can connect to Bluetooth and Wi-Fi, which adds multiple integration options to the master controllers [8]. This chip is connected to the Ethernet adapter and manage communication with external parts, like an API or external interface devices such as tablets or card readers.

Because the slave controller only works as an extension of the master controller, it does not require an ESP32 chip and does not communicate with any external devices. The slave controller only reacts to signals from the master controller.

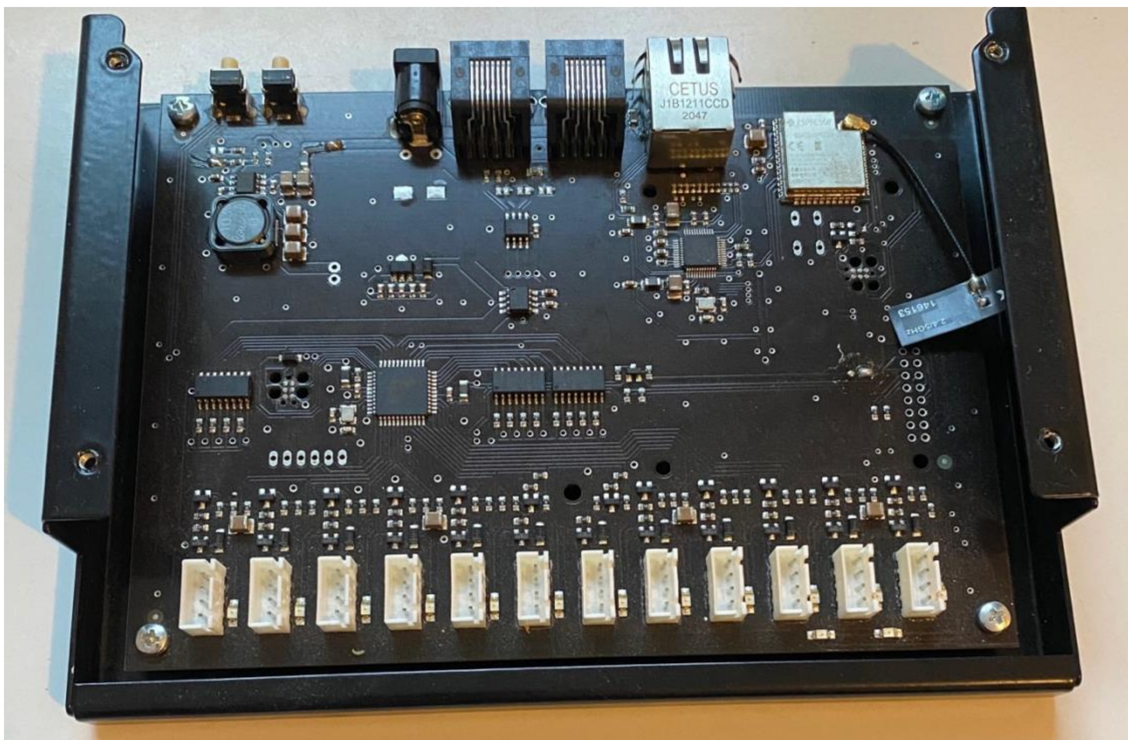

Figure 1. Lock Controller.

In the flash tool, the master and slave controllers have different name. The master controller is called “Baseunit” because it builds the basis for a line of lock controllers. The slave controllers are just called “Controllers” because their only job is to control their designated locks.
Outside of the development environment, however, the controllers are referred to as master and slave controllers, since it conveys their hierarchy better. For the rest of this thesis, the master and slave nomenclature will be used.

2.3 Flash adapters

Each chip has a flash port for a 6 pin ARM Cortex cable, shown in Figure 2 [9].

The cables have 3 stabilizing rods, 4 claws to keep it in place and 6 pins that transfer data. These cables connect to the controller board and to one of two adapters.

The adapters are used as a middleman between the flash tool and the controller. The flash tool only sends the firmware file to the adapter, but the adapter handles the exact flashing of the device by writing the bits to correct memory addresses.

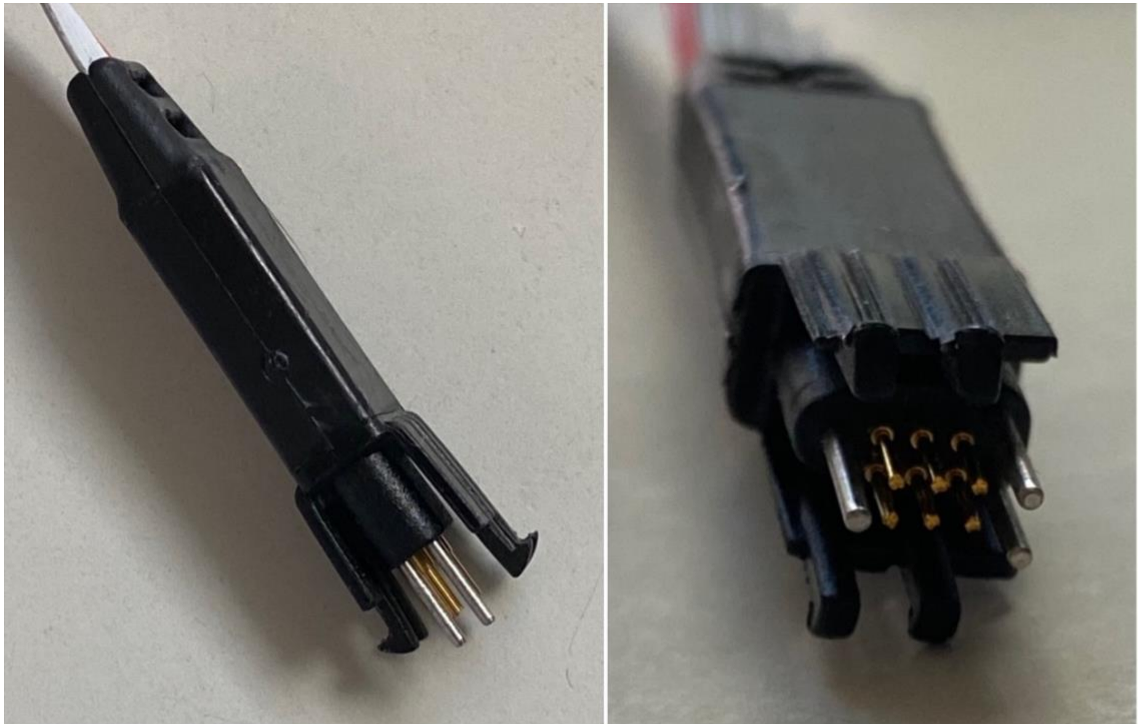

Figure 2. ARM Cortex Cable.

2.3.1 Slave Controller Adapter

The adapter for the Slave Controller uses a program called AVRDUDE [10]. AVRDUDE is a program to write a memory image onto AVR chips. It was first developed in 2003 but is still updated. The adapter can be seen in Figure 3.

Figure 3. Slave Controller Adapter.

2.3.2 Master Controller Adapter

The second adapter is for the Master Controller and uses the esptool [11] command line tool for the flash process of espressif chips like the ESP32 in this case. The master controller adapter is shown in Figure 4**Error! Reference source not found.**.

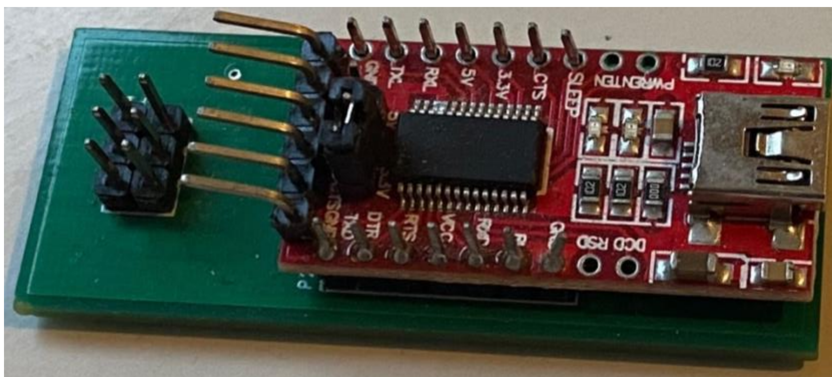

Figure 4. Master Controller Adapter.

3 Requirements

Requirements are a list of features that are devised by the contractor and project manager or the team of developers. This list builds the foundation of a project and is used to determine the progress and success or failure of the project.

3.1 Previous Flash process

Before the flash tool, technicians at Pareva used very simple command line scripts to flash the chips. These scripts had a number of issues. One of them was that the desired firmware version still had to be typed in manually as a parameter to the script, which is prone to errors and not user-friendly.
Another issue was that each chip had its own flash script. In the case of the lock controller, 2 scripts.
The biggest issue, however, was that USB-ports, which the flash adapter were connected to, were hard-coded into the script. This is a problem on Linux, because it is not guaranteed that a device is assigned the same name after unplugging and replugging.

Other smaller issues included that the error messages were hard to distinguish from the other text generated by the flash commands. So, when anything went wrong, it was hard to recognize.

3.2 Requirement Priorities

In order to make a new and improved flash tool, a set of requirements had to be created. The following list of requirements is categorized according to the MoSCoW method [4], which uses a system of four categories: Must have; Should have; Could have; Won't have.

3.2.1 Must have

All features in this category are crucial for the success of the project and have the highest priority. If any of them are missing, the final product does not work.

Selectable Firmware
It was deemed important to be able to select the desired firmware more easily. The different available firmware versions also have to be displayed to the user, so he can choose the correct one.

Easy starting/stopping method
The new flash tool needs a better way to start a new flash process and pause it if necessary.

Error message display
In order to make debugging easier, the error messages have to be intercepted, interpreted and displayed in a more understandable way.

Baseunit flashing
Pareva's main product is the lock controller. They have two variants, as outlined in chapter "2.2 Device ". Both of them need be flashed with the flash tool.

3.2.2 Should have

These features are not essential to the core behavior of the project, but greatly increase its functionality.

Adapter recognition
When the flash adapters are plugged in, their name assignment is fairly unpredictable. A better way of recognizing which adapter is plugged into which port should be developed.

Flexible design
The functionality of the flashing process should be adjustable for varying future requirements.

Extensible design
Functionality should be extensible and more flash devices should be easily addable to the flash tool.

Works on Linux
The whole flash tool should be running on Linux.
Initially there were other Operating System options, which is why this point is under “Should have” and not under “Must have”

3.2.3 Could have

This category describes features that increase the user experience, without adding crucial functionality. The implementation of these features is optional.

Visual Progress indicators
The command line output that indicates progress could be intercepted and displayed in a more visual way. Ideally, there will be no command line outputs necessary for the operation of the flash tool.

Device connection check
When the flash adapters are connected to the board of the lock controller, there could be a way to detect whether or not the adapters are connected properly.

3.2.4 Won’t have

Because the work on the flash tool was not limited to the professional work placement it was not clear how much time there would be for additional features. Hence, the “Won’t have” category was not decided at the start of the project. Instead, this category showcases features that were discussed at some point during the project but were not implemented after all.

Card reader flashing
During development, an additional feature was proposed. The lock controllers

can be coupled with a card reader, which was supposed to be added to the flash tool. Unfortunately, due to time constraints and different priorities, this feature was demoted to the “Won’t have” category.

4 Architecture

The architecture of a flash tool refers to its structural elements and how they interact with each other. Whilst the exact implementation of these elements is not the focus of this step, general design principles such as flexibility and extensibility should be kept in mind already.

4.1 Architecture of flash tool

The flash tool is contained in a single window, which is created when the application starts. This window contains every component of the tool with the exception of the settings window and various smaller popup windows, which are not crucial to the core operation of the tool.

The top of the window, which can be seen in Figure 5, holds the exchangeable top section. Every flashable device can have its own top section, which is rendered based on the selected device. The device can be selected in the top section itself. In addition to that, the user can select the desired firmware from a dropdown menu in the top section.
For the lock controller, the top section also features two checkboxes to select whether the baseunit, the controller or both should be flashed. In addition to that, there is a checkbox to automatically select the most recent firmware version automatically.

In order to better understand what the tool does and what is expected of the user, the middle section of the tool contains customizable panels. There is a panel for each individual flash step and every device can have its own set of flash steps. The panels light up when they are active, so the user knows what the flash tool is doing.

Another helpful feature is the introduction of custom widgets that can be added to a panel. They can display labels, progress bars, buttons or switches. Using

these widgets, the user can adjust the behavior of the flash process during run-time or read useful information displayed by the tool.

To combat the issue of not knowing when the flash adapters connect to the lock controllers, two separate steps were added. One before and one after the actual flash step to make sure the adapters are connected and disconnected correctly.

Underneath the progress panels, the tool contains a collapsible console, which streams the contents of the flash commands. This makes sure that no information is lost and can be used to further troubleshoot if the progress panels do not suffice.

Finally, at the very bottom of the main window, the user can find buttons to start, pause and stop the flash process, along with a label that displays the completed flash processes.

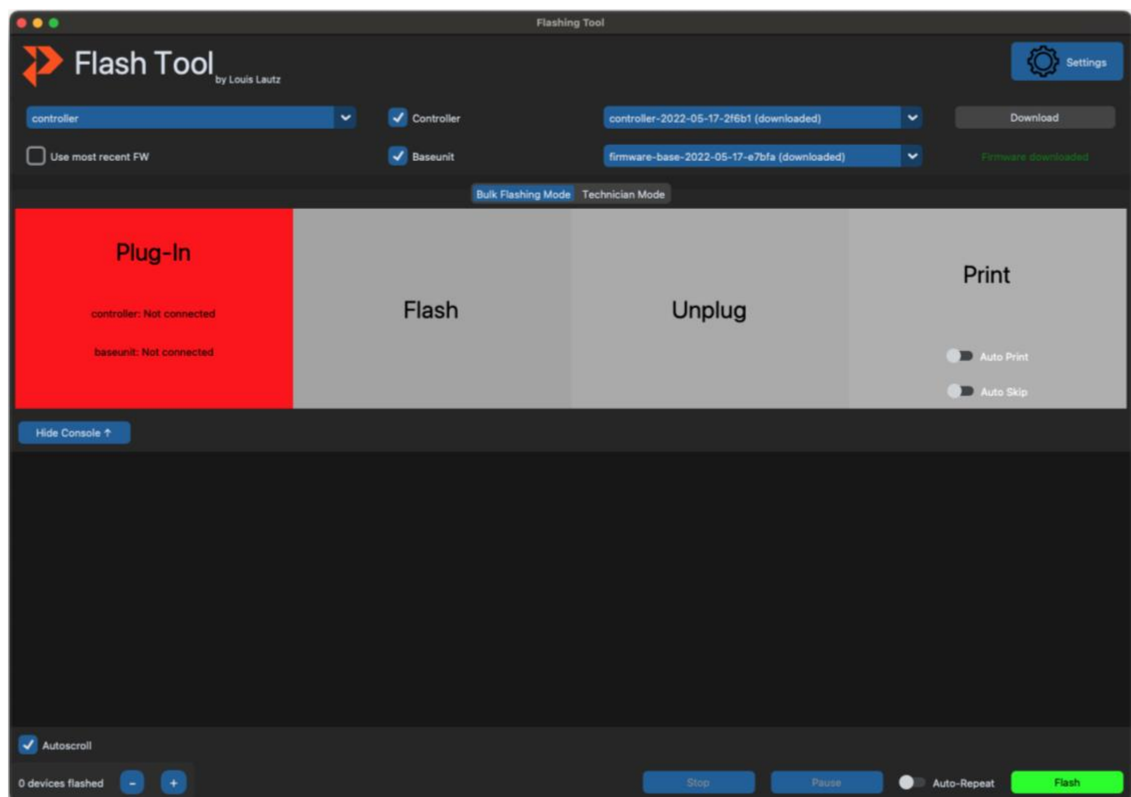

Figure 5. Flash Tool GUI.

5 Implementations

In the implementation phase of the project, the features are built into the code. This step takes up the majority of the work time.

5.1 Implementation of flash tool structure

The main GUI is initialized with the tkinter, or in this case the Customtkinter base class CTk(). This class simply instantiates an empty window, that can be populated with different widgets at run time.

The main app contains two sections that can be exchanged to achieve more flexibility. The top section and the flash panels. Both of them are contained in a separate class, which makes it easier to add more or change existing components.

In order to guarantee a reliable base behavior of the top section, each top section inherits a set of functions from a top_section_base_class(). The base class implements all functions that any top section needs for its basic operation. These functions are also the only functions that will be called from the main app. Additional functions are either “private” functions, that are used internally by the class itself, or are called from flash functions inside the flash panels.
So far, the available top sections all feature a dropdown menu to select the firmwares the user wants to flash. These firmwares come from an API, which holds all available firmwares. A list of all available firmwares, individual firmwares, as well as drivers can be downloaded via the API. When a firmware is already saved on the host computer, a label with “(downloaded)” appears behind the firmware version in the dropdown menu, as shown in Figure 6.

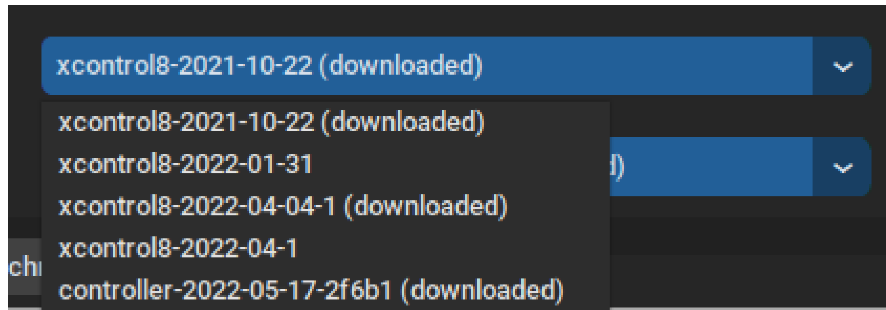

Figure 6. Download Status Tags.

Similar to the top sections, the flash panels need to be exchangeable. However, there is one more layer between the panels and the main app. That is because there are multiple panels, which need to be contained somewhere, whilst being exchangeable as a unit. For this purpose, a "device_class()" was created. The device_class() is specific to each flashable device. It holds general information about subdevices (in case of the lock controller, which consists of multiple chips), flash adapters, flash drivers and the top_section type. In addition to that, the device_class() defines the flash functions and the flash panels.

The controller code is listed in Appendix 1

That means, that all information about a device is defined in one place, which increases extensibility. This can be seen in Figure 7. In order to add a new device to the flash tool, the user only has to create one file, the device specific device_class(), and save it in a particular file path. The flash tool will scan that folder at start up and make the devices available in the device selector dropdown menu in the top section.

As mentioned before, the device_class() instantiates the flash panels. Unlike the top section or device class, the flash panels do not inherit from a general base class. That is due to their built-in flexibility.
The flash panels are supposed to display information about the current flash step or allow for inputs at runtime. It achieves that through the use of widgets and through a unique flash function. More on those two concepts later.

Interestingly, these two concepts provide so much flexibility in what a flash panel can do, that the panels themselves have very little own functionality. Their only tasks are to display which panel is currently active, run the flash function and render their widgets. Those tasks are so general, that every panel can be the same, but can still be customized with the injection of the flash function and the widgets.

The widgets themselves are simple frames with one or two components inside. They are rendered onto the flash panels and their states can be read by the flash function to react to user inputs during run time.
Widgets have 5 important variables. “active”, “visible”, “always_active”, “always_visible”, “always_hidden”.
The first two variables are changed in every flash cycle. Whenever a panel gets activated, all of its widgets switch to active and visible. “Active” means, that the widget can be interacted with. Buttons can be pressed and switches can be switched. “Visible” simply renders the widget on the screen, making it visible. The true potential of the widgets, however, comes from the other three variables.
“always_active” overrides the “active” variable, meaning that a widget can always be interacted with. Even when the panel is deactivated. This option allows the user to interact with upcoming panels before they are activated.
“always_visible” and “always_hidden” override the “visible” variable, which, as the names suggest, make the widget always visible or always hidden.
Any widget with “always_active” set to True, also should have “always_visible” set to True, so the user can see the widget and interact with it.
“always_hidden” is used for error messages. The label for the error message is kept hidden until an error occurs, at which point the error message is written into the widget and the widget is displayed. For this purpose, the widgets have a “reveal” method, which renders the widget even when they are set to “always_hidden”.

The flash panel code is listed in Appendix 2

Underneath the flash panels, the collapsible console is located. The console is a Customtkinter textbox, which has numerous functions to edit the display text, as well as a built-in scrollbar. To decrease memory consumption of the tool, a 500-line limit is implemented into the console. This allows the user to inspect past error messages, but also implements some memory efficiency.

At the bottom of the main window, the user can start, pause and end the flash process. This functionality is implemented with simple buttons. When the flash button is pressed, the flash tool checks a few criteria to determine whether the flash process is ready to be started. All selected firmwares need to be downloaded and the flash adapters need to be connected to the computer. If either of these criteria are not met, the user is prompted to fix these issues. Additionally, the buttons change the flash tools “app_state” variable. This variable can have different values, reflecting what the flash tool is supposed to do at the moment. Based on the app_state, certain parts of the GUI get disabled or enabled, to prevent the user from making unintended inputs. For example, during the flash process, the user cannot change the firmware. The user is only able to change what firmware will be flashed, when the flash process is stopped.

The function to start the flash cycle is listed in Appendix 3

Another important addition made to the flash tool is the ability to create UDEV rules for the flash adapters in a device setup window. This solves the problem of the flash adapters changing their names when they are unplugged and plugged in. Whenever the flash cycle is started, the flash tool checks if there is already a UDEV rule for the required flash adapters. If there is not, the user is prompted to unplug and replug the flash adapter, so that it can be found. Then, a new UDEV rule is created. This UDEV rule gives the flash adapter a unique name that it will always be assigned when its connected. UDEV rules require the sudo password, so the last step of the device setup window is to input the password, which is verified before the UDEV rule is created. After that, the setup will never have to be repeated until a new flash adapter is needed in the future.

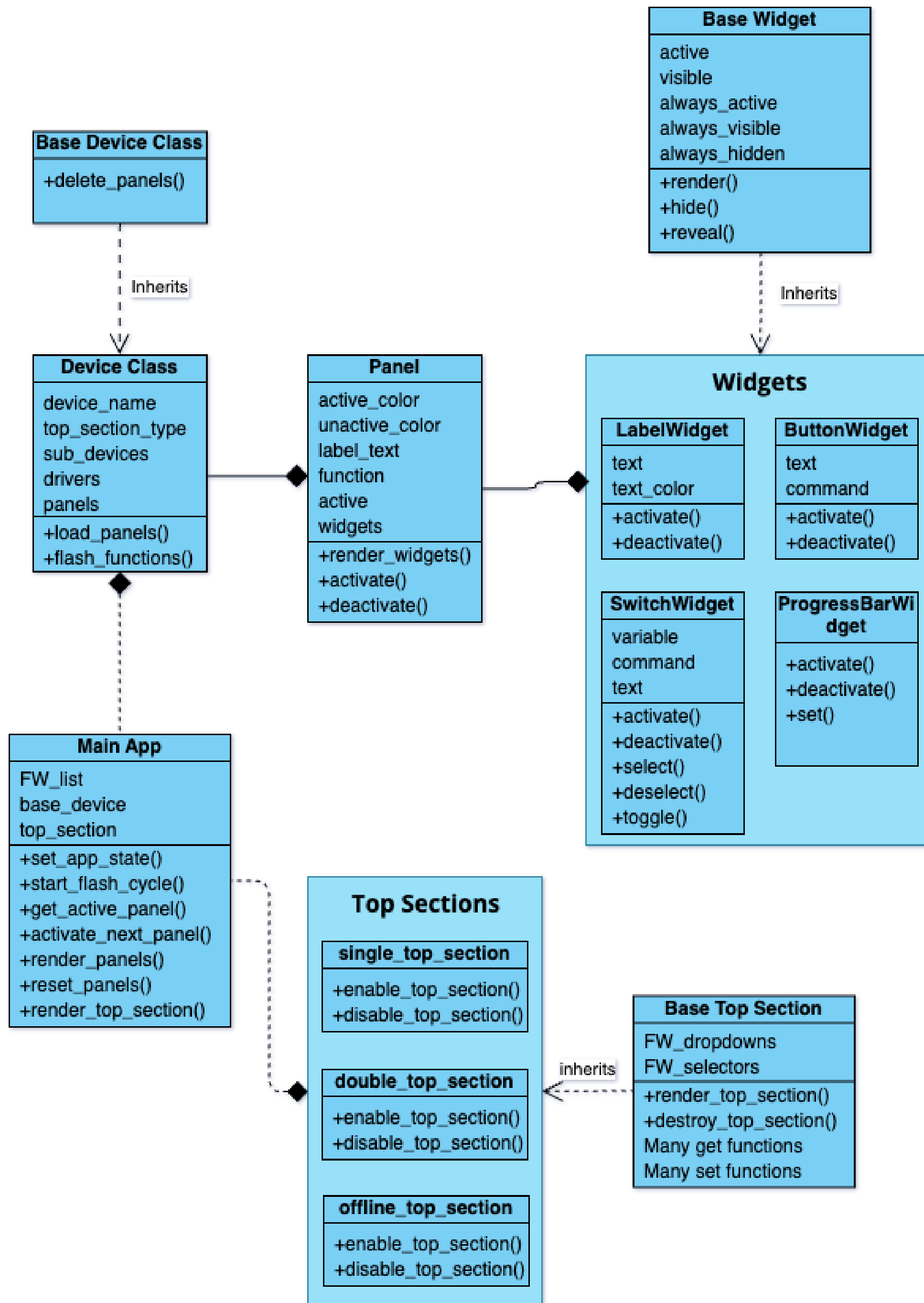

Figure 7. Flash Tool Architecture Diagram.

5.2 Flexibility

Flexibility is a property of code, which allows fast and easy adaptation to incoming changes.

5.2.1 Flexibility seen as resistance to change

A core idea expressed by Christian Clausen is that flexibility is measured in its resistance to change. In other words, making a change to existing code gets easier if the code was designed with flexibility in mind. [12]
In the case of the flash tool, there are many examples of how changes can be made without meeting a lot of resistance. On the other hand, some features are more rigid, due to a lack of prioritization of that feature or because the behavior of that feature is not planned to change in the near future.

One example of low resistance to change is the isolation of flash function inside of the flash panels. The flash functions only requirement is that they need to return True, when they finish successfully and return False, if something went wrong. With this freedom, the developer can simply write a function in a separate test bed, test it thoroughly and paste it into the flash function of a flash panel. No other external component from the flash tool can interfere with the flash functions.

Looking at the settings implementation however, it becomes obvious that not every aspect of the flash tool is optimized at this point in time.
Each setting is rendered with its own hardcoded GUI components and is updated with individual conditions that check which settings were updated by the user. In addition to that, each setting has a custom property in two separate JSON files. This approach works for the current set of settings, but it makes future change very difficult. If a developer has to add a new setting, all of the previously mentioned hardcoded lines of code have to be added for the incoming new setting. The developer experiences more resistance from the code.

Dane Hillard in his book “Practices of the Python Pro” calls this approach the “shotgun surgery” approach, as many small changes need to be made in many locations, to implement one larger change. [13]

5.2.2 Coupling

Another concept that is often talked about is coupling. Coupling occurs when two features depend on each other’s implementation. When one implementation changes, the other one breaks as well.
Decoupling functions from each other can be a useful tool to make an application more robust, because one change will not require other changes.

In the flash tool, this concept can be observed in the panels. The panels have a list of widgets that will be rendered inside of it. To add these widgets, the panel has a function *add_widgets()* to add more widgets to it. Initially the function only accepted a list, which meant that if only one widget was added, the developer had to type-cast the parameter to a list with [] brackets. To avoid that, the function now can take a single widget or a list of widgets. If only one widget is supplied, the single widget will be added to an empty list and is processed like any other list of widgets.

```
def add_widgets(self, widgets):
    """Adds one or more widgets to the panel and displays them"""
    if type(widgets) is not list:     # Checks if only one widget was added
        widgets = [widgets]           # Turns the single widget into a list

    for widget in widgets:
        self.widgets.append(widget)   # Adds all widgets to the panel
    self.render_widgets()
```

The same implementation was used to make the tool more reliable for flash operations with multiple devices. For example, the device classes have a property called *sub_devices*. The controller has two separate chips, so the subdevices are “controller” and “baseunit” in a list. If a device only has one chip, it is still saved in a list. This way, no functions rely on the number of subdevices

and are therefore not coupled to that number.
However, the functions are still coupled to some extent, because they only rely on lists.

Coupling is not always completely avoidable, but it is important to keep in mind how functions are dependent on each other and what future changes might break them.

5.2.3 Temporal Coupling

During development a big issue came up, which broke the entire application for a while. The problem was identified as *temporal coupling*.
Temporal coupling is a type of coupling, where functions in a system are dependent on the order in which they are executed. Certain functions require values that other functions provide or call functions that perform a task which is required for the current function.
The problem during development was, that so many functions called other functions, that it was very difficult to understand the order in which they are called. Some functions were called multiple times or at inappropriate times.
The messy function flow was fixable, but the general problem of temporal coupling is difficult.
Certain tasks simply have to be executed before others. Robert C. Martin in his book “Clean Code: A Handbook of Agile Software Craftsmanship” writes: ” *Temporal coupling is often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious*.” and proposes a solution in which each function call produces a result, which the next function can use. This not only reveals the coupling, but also strictly enforces it, since the functions now cannot work without the previously computed arguments. However, he also admits that this approach further complicates the function.

In this application, removing this temporal coupling may not be an optimization that is worthwhile. The flash tool is already fairly complicated on account of the

various functions that call each other. Artificially inserting return statements and function arguments into this complex structure could make the temporal coupling more obvious but would also make the rest of the function flow less obvious, which was not a tradeoff worth making.

This decision reveals an interesting dilemma many developers find themselves in:

5.2.4 Pros and Cons of Flexibility

Is the additional flexibility really beneficial at this point in time?
Answering this question requires the developer to weigh up the pros and cons of flexibility.

Pros:

Implementations to increase flexibility can makes future implementations or changes to existing ones much easier.

Another advantage of flexibility is that bug fixes are faster. Flexible systems are often more encapsulated and components work independently, which makes identifying bugs easier. The developer can analyze smaller, compact sections of the code base, without getting lost in the larger architecture. Moreover, the independence of the components reduces coupling, which decreases the chance of breaking working components when chances are made.

Cons:

Flexibility can be seen as a time investment. Developing a flexible system usually takes more time to develop than a simpler system that works for only one desired use case.

The developer, or their contractors always have to try to predict future changes in order to build a flexible system, which can accommodate these changes.

That means, that the predictions can be wrong. In which case the initial time investment did not pay off, but it can also lead to a system design that is less suitable for the actual incoming changes.

If a system is prepared for a change that never comes, it is very possible that the developers now have to work around an existing architecture, that does not work well for the actual new change. In other words, flexibility is not universal. There is no objective flexibility. Instead, flexibility has to be specific to the requirements.
The flash tool for example, works under the assumption, that there are multiple devices that all have a certain firmware version and have different steps to be flashed. So, the measures to provide flexibility are built around that assumption. They make it easier to add new devices and change the flash steps. If we pretend this assumption was wrong for a moment, we could imagine that a new device came out, that has to be flashed in multiple rounds and does not have a particular firmware, but certain combinations of firmwares and drivers. All measures to make the system more flexible would not be of any value for this new hypothetical device, because they were designed around a different set of core requirements.
So, making flexible systems can actually make future changes harder, if the system is flexible in the wrong way.

5.3 Extensibility

In contrast to Flexibility, Extensibility aims to make the addition of new code or behaviors as easy as possible. Ideally, a system is so extensible, that the addition of a new feature strictly requires the writing of new code, without having to change anything else.

5.3.1 Separation of Concerns

Separation of concerns is imperative for an extensible system design. It is implemented by breaking the code into chunks, but in a specific and strategic way.

Each chunk should solve a concrete and understandable task. It is also important that these chunks can solve the task by themselves, without having to reference external functions or data. This reduces coupling between chunks and promotes flexibility and reusability. In addition to that, reduced coupling also sets up separation of concerns, together with one other process: Increasing of cohesion. [14]

5.3.2 Cohesion

Cohesion describes the concept of grouping together similar code and separating different code. Looking at the chunks again, it is important that chunks that are different, get their own function to solve their own individual task.
In the same vein, chunks that are similar can be grouped together, to create a collection of similar code. This could be implemented in a module or a class, allowing them to be reused. [14]

Separation of concerns and cohesion can make a system more clear, reusable and testable because it is divided into logical chunks that work independent from each other. It can also speed up development between multiple developers, since each developer can work on a separate part of the system, or a separate concern.

6 Testing

To confirm that the implementation of features works as expected, tests should be performed throughout the implementation process or at least afterwards. It can be helpful to reference the feature list again, so no tests are left out.

6.1 Must have requirements

Selectable Firmware
The desired firmware can be easily selected for each sub device, using the dropdown menu in the top section. Additionally, the firmware entries display whether or not they are already downloaded to the device.

Error message display
Errors that occur during the flash process are filtered and custom display messages are displayed in the flash panels. However, the original output of the commands is still streamed to a console in the tool.

Baseunit flashing
The lock controller can be flashed and the technicians had no complains after the final fixes.

6.2 Should have requirements

Adapter recognition
Custom UDEV rules are added for each device. This eliminates the problem of inconsistent port names. The UDEV rule setup is handled through a separate pop-up window that guides the user through the setup step by step.

Flexible design
Different strategies for the implementation of a flexible design were employed. Such as decoupling components from another and isolating similar functionalities.

Extensible design

The tool can be extended at various different points.
New devices can be added with the device classes. The device classes implement other extensible systems, like the flash functions, flash panels and widgets. Lastly, the top section is also encapsulated into a class, which makes them extensible as well.

Works on Linux

After the first iteration of the tool, where mostly the layout and basic functions were implemented, the tool was ported to Linux and since then developed in Linux.

6.3 Could have requirements

Visual Progress indicators

The flash panels have progress bar for some of the flashing steps. In general, the flash panels serve as a progress indicator since each panel indicates one step in the flash process.

Device connection check

Now, a new step at the beginning and at the end of the flash process can be added to detect whether the adapters are properly connected to the controllers.

7 Conclusion

The goal of this project was to replace Parevas old flash tool and improve the productivity of the flashing process. All in all, this goal was achieved. The most important metric for this conclusion should be the technician's judgment of the flash tool and they expressed their satisfaction with the finished product. The flash tool runs smoothly and is now an integral part of their manufacturing process.
The tool should be sufficiently future proofed, meaning that new devices should be easy to implement.

The greatest struggles during development can be ascribed to poor planning during the early stages of the project. For future projects, a more extensive project plan should be devised.
Concretely, categorizing features with the MoSCoW method can be a useful method to set priorities and should be implemented from the beginning.
Additionally, it would have been helpful to think more about extensibility and especially flexibility from the start. Identifying exact features that need to be flexible or extensible should be the strategy for future projects.
Moreover, discussing possible future changes with the contractor can help fine-tune the direction of development in terms of flexibility and extensibility.
Another helpful improvement for the next project would be to learn more about the used framework instead of "learning on the job". This could save time refactoring code to clear of bad habits. Furthermore, it can improve the codes robustness and align it with common practices of the given framework.

Most projects are never truly finished. The flash tool is no exception. If there was more time and resources to continue the development of the flash tool, the settings menu could be made modular and extensible as well. This could allow the developer to dynamically add settings that only affect a single device. At the moment, the settings are hard coded and would require more effort to change than if they were properly implemented with extensibility in mind.

Other features, such as the app states could be made into classes as well to, again, add a new source of extensibility to the tool.

Reference

[1] Wikipedia, "API," 2023. [Online]. Available: https://en.wikipedia.org/wiki/API. [Accessed 2 November 2023].

[2] J. Juviler, "HubSpot Blog," 30 August 2023. [Online]. Available: https://blog.hubspot.com/website/what-is-gui. [Accessed 2 November 2023].

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1994.

[4] Wikipedia, "MoSCoW Method," 2023. [Online]. Available: https://en.wikipedia.org/wiki/MoSCoW_method. [Accessed 4 November 2023].

[5] Python Software Foundation, "Python Documentation," 2023. [Online]. Available: https://docs.python.org/3/library/tkinter.html. [Accessed 23 October 2023].

[6] T. Schimansky, "GitHub," 2023. [Online]. Available: https://github.com/TomSchimansky/CustomTkinter. [Accessed 23 October 2023].

[7] Microchip, "ATmega324PB - Documentation," 2023. [Online]. Available: https://www.microchip.com/en-us/product/atmega324pb. [Accessed 7 November 2023].

[8] Espressif, "ESP32 Product Overview," 2023. [Online]. Available: https://www.espressif.com/en/products/socs/esp32. [Accessed 6 November 2023].

[9] Tag-Connect, "Arm Cortex Cable - Documentation," 2023. [Online]. Available: https://www.tag-connect.com/product/tc2030-ctx-6-pin-cable-for-arm-cortex. [Accessed 10 November 2023].

[10] AVR Dudes, "AVRDUDE Documentation," 2023. [Online]. Available: https://github.com/avrdudes/avrdude. [Accessed 4 November 2023].

[11] Espressif, "esptool Documentation," 2023. [Online]. Available: https://docs.espressif.com/projects/esptool/en/latest/esp32/. [Accessed 4 November 2023].

[12] C. Clausen, "Good and bad flexibility in code," 2021. [Online]. Available: https://freecontent.manning.com/good-and-bad-flexibility-in-code/. [Accessed 26 Sep 2023].

[13] D. Hillard, What Makes Code Extensible and Flexible?, Manning Publications, 2019.

[14] A. Naumov, "Separation of Concerns in Software Design," 2020. [Online]. Available: https://nalexn.github.io/separation-of-concerns/. [Accessed 22 November 2023].

[15] R. C. Martin, Clean Code: A Handbook of Agile Software Craftmanship, Philadelphia: Prentice Hall, 2008.

Appendices

Appendix 1: Controller code

```
class controller(device_base_class):
    def __init__(self, app, master_widget):
        super().__init__()

        self.device_name = "controller"
        self.app = app                              # A reference to the main tkinter app
        self.master_widget = master_widget          # The widget that the panels will be placed in
        self.top_section_type = "Double"            # Type of top section. "Single" for one firmware
                                                    # selector. "Double" for two
        self.sub_devices =[                         # Stores a list of device names that this device
                                                    # used
            {
                "sub_device_name": "controller",
                "display_name": "Controller Flash Adapter",
                "udev_name": "FT_controller_flash_adapter",
            },
            {
                "sub_device_name": "baseunit",
                "display_name": "Baseunit Flash Adapter",
                "udev_name": "FT_baseunit_flash_adapter",
            }
        ]
        self.drivers = {"baseunit": ["bootloader_dio_40m", "boot_app0"]}
        self.panels = []

        # Custom variables
        self.baseunit_serial_number = ""
        self.print_bool = False
        self.skip_bool = False
        self.auto_print = customtkinter.BooleanVar(value=False)
        self.auto_skip = customtkinter.BooleanVar(value=False)

        self.render_USB_setup_bool = False
        self.timeout_count = 0
        self.error_timeout = False

        self.load_panels()
```

Appendix 2: Code for flash panels

```
def render_panel(self):
    """Places the panel on screen and places the label inside of the panel"""
    self.grid(row = 0, column = self.col_num, sticky = "nsew")      # Places panel on screen
    self.columnconfigure(0, weight=1)                                # Sets panel content scaling behaviour of column
    self.rowconfigure(0, weight=1)                                   # Sets panel content scaling behaviour of row
    self.master_widget.columnconfigure(self.col_num, weight=1)       # Sets panel scaling behaviour
    self.label.grid(row = 0, column = 0, sticky="nsew")              # Places label inside of panel

    self.render_widgets()


def render_widgets(self):
    """Places the widgets inside of the panel"""

    if self.active:                                    # Checks if panel is active
        for widget in self.widgets:                    # Loops through widgets
            widget.activate()
            widget.render()
    else:
        for widget in self.widgets:
            widget.hide()
            widget.deactivate()


def activate(self):
    """Sets the panel into its active state"""
    self.active = True
    self.configure(bg_color = self.active_color, fg_color = self.active_color)          # Changes frame color
    self.label.configure(bg_color = self.active_color, fg_color = self.active_color)    # Changes label color
    self.render_widgets()                     # Rerenders only the widgets


def deactivate(self):
    """Sets the panel into its deactive state"""
    self.active = False
    self.configure(bg_color = self.unactive_color, fg_color = self.unactive_color)          # Changes frame color
    self.label.configure(bg_color = self.unactive_color, fg_color = self.unactive_color)    # Changes label color
    self.render_widgets()                     # Rerenders only the widgets


def add_widgets(self, widgets):
    """Adds one or more widgets to the panel and displays them"""
    if type(widgets) is not list:       # Checks if only one widget was added
        widgets = [widgets]             # Turns the single widget into a list

    for widget in widgets:
        self.widgets.append(widget)     # Adds all widgets to the panel
    self.render_widgets()


def remove_widgets(self, widgets=None, all=False):
    """Removes certain widgets or all widgets"""
    if type(widgets) is not list:               # Checks if only one widget was added
        widgets = [widgets]                     # Turns the single widget into a list

    if not all:                                 # Checks if all widgets need to be removed
        for widget in widgets:                  # Loops through widgets
            if widget in self.widgets:          # Checks if supplied widget exists in the panel
                self.widgets.remove(widget)     # Removes individual widget
    else:
        self.widgets.clear()                    # Removes all widgets

    self.render_widgets()                  # Rerenders panel to update widget rendering
```

Appendix 3: Flash cycle start function

```
def start_flash_cycle(self):
    download_order = self.get_download_order()     # Gathers all resources to be downloaded
    if not download_order:                         # Checks if anything needs to be downloaded
        # Checks if all device ports are configured
        if check_udev_rules(self.obj_device.sub_devices):

            self.set_app_state("Active")
            # Checks if at least one subdevice was selected to be flashed
            if len(self.top_section.get_selected_FW()) > 0:
            # Runs the flash cycle in a separate thread so other functions can run at the same time
                self.flash_thread = StoppableThread(target=self.flash_cycle)
                self.flash_thread.start()
                for thread in threading.enumerate():     # Loops through all threads
            # Checks if current thread is not the flash thread or the main thread
                    if thread != self.flash_thread and thread != threading.main_thread():
                        thread.stop()     # Stops thread, because it is no longer needed
            else:    # Renders popup window that reminds user to select at least one subdevice
                render_no_FW_selected_popup(self)
        else:    # Renders popup window that steps user through flash port setup
            render_USB_setup_window(self)
    else:    # Checks if download popup should be rendered
        if get_setting("SETTINGS", "show_flash_download_popup"):
        # Renders popup window where user is asked if selected firmware should be downloaded
            render_FW_not_downloaded_popup(self)
        else:
            self.download_button_event().   # Runs download process
```