CSCI 370 Final Report

Wasm Squad

Brendan Burmeister
Kai Page
Joey Vongphasouk

Revised June 16, 2023

CSCI 370 Summer 2023

Dr. Rob Thompson

Table 1: Revision history

Revision	Date	Comments
Rev1 - Requirements document (due 5/16)	2023-5-15	Added sections: ● Introduction ● Functional Requirements ● Non-Functional Requirements ● Risks ● Definition of Done ● Team Profile ● Appendices (Appendix A) ● Sources
Rev2 - Design document (due 5/25)	2023-5-25	Added sections: ● System Architecture Revised sections: ● Appendices - more definitions added
Rev3 - Software Quality and Ethics document (due 6/2)	2023-6-2	Added sections: ● Software Test and Quality ● Project Ethical Considerations Revised sections: ● Appendices - more definitions added
Rev4 - Results documents (due 6/9)	2023-6-7	Added sections: ● Results ● Future Work ● Lessons Learned ● Acknowledgements Revised sections: ● System Architecture - added interface design and accompanying explanations
Rev5 - Final Report draft document (due 6/12)	2023-6-12	Added sections: ● Appendix B Revised sections: ● Revisions history - formatted table better with better revision explanations ● Introduction - added product vision and Wasm goals Other general revisions: ● Changed verb tense to be of past tense ● Revisions across paper to reflect actual progress
Rev6 - Final Report document (due 6/16)	2023-6-14	Added sections: ● Technical Design Revised sections: ● General revision of terms and grammar across paper ● Introduction - added clearer product vision ● Technical Design - moved data flow from system architecture

Table 1: Revision history.

Table of Contents

I. Introduction

The primary goal of our project was to write and implement an interface that WebAssembly (Wasm) runtimes could expose to programs in order to control GPIO pins. This interface was implemented into an existing Wasm runtime called SpiderLightning and allows users to portably write and compile code that uses GPIO features to interact with external hardware from a variety of target platforms.

Our client is Andrew Gracey, lead project manager for Cloud Native Edge at SUSE who is also an alumni of Mines. One of his team's current research topics is providing support for Wasm as a first-class workload type in cloud architectures. Wasm is a low-level bytecode format originally designed to portably and securely run C and C++ libraries in web applications. The Wasm technology ecosystem has since expanded considerably: additional languages, such as Rust and Go, can be compiled to Wasm, and Wasm binaries are commonly run outside the web in order to take advantage of their security and portability properties.

Andrew wanted our project completed so that his team can post our project to an open-source nonprofit organization, ByteCode Alliance. This maximized the reach of our contributions, making the interface available to any development team working with GPIO applications.

The existing software we modified is called SpiderLightning, which was originally developed by SUSE. SpiderLightning is "a set of WIT definitions and associated implementations to enable app developers to work at a faster pace and require less knowledge of the environment in which they are executing." Our addition provides an interface to this product to expand its capabilities and further the efficiency of its future developers.

II. Functional Requirements

The main deliverables for the project are an *interface definition* in the Wasm Interface Type (WIT) format, and an *implementation* for Raspberry Pi as part of the SpiderLightning runtime. This means that our code must be able to run on a Raspberry Pi device through SpiderLightning. Also our code must be able to have its pin state configured at platform initialization. This required a shift in where we coded our interface so that it would use the slight file to configure its pin state.Additionally, we created a demo *application* that showcases the functionality of the interface definitions. Also we created unit testing to ensure the slightfile formatting was being read incorrectly for testing GPIO functionality on our demo and for future uses of the project through ByteCode Alliance.

The specifics of the requirements needed by the client are listed below:

- Interface definition
  - Configurable pin state during program initialization (input/output direction, etc.)
  - Read pin logic-level (high/low)
  - Write pin logic-level (high/low)
  - Generic with respect to hardware
- Implementation
  - Support all defined capabilities of interface
  - Integrate into SpiderLightning codebase/runtime
  - Broad enough so can be implemented for several devices
- Demo application
  - Showcase capabilities of interface based on example use cases in proposal
    - Communicate with external application using other SpiderLightning capabilities
  - Run on Raspberry Pi implementation
  - Record video and show to client

III. Non-Functional Requirements

The project consists of an interface through which a semi-uninformed user can read the data and easily interpret. This was done by the use of creating test cases in order to ensure proper functionality and to prevent future issues down the line. To make the code easy to understand for a new user we had detailed function names and tried to make our implementation as straightforward as we could. To ensure future success with our code we provided documentation that describes what is going on with anything that would not be self explanatory in the code.

Our code conforms to the standards of SpiderLightning and is well documented. SpiderLightning itself follows the Microsoft open source code of conduct and we followed this accordingly. By doing this we made sure that the code would be good for future developers. It is important that we created high quality code in which one could easily refer to or run demos. We supplied proper documentation of our testing to the github when we sent out our pull request in the form of a video demo as requested by the client.

We opted to do a peer review with some of the clients' accomplices in order to ensure our code is to their standards. This will be done in the next few weeks following the field session and will cover basic fixes to the code that does not actually change the functionality. It may just simply increase efficiency of code or reduce clutter. There may be some code that may be able to be written simpler or we may be breaking some policies that they value.

IV. Risks

Technology risks associated with this project come from the complexity and unfamiliarity of the various underlying platforms. Listed in Table 2 are major technology risks involved in the project.

Risk	**Likelihood**	**Impact**	**Risk Mitigation**
Our test Pi might break	Low	High	Would have consulted client and advisor regarding substitute hardware
Cross-compiling for aarch64 might be harder than we think	Medium	High	Installed the Rust toolchain on the test Pi and natively compile it
Defining and using a callback interface might be difficult	Medium	Medium	Would have referenced other SpiderLightning modules that involve callbacks (e.g. HTTP server) for implementation hints
Container sandboxing may interfere with HAL access	Medium	High	If this turned out to be a problem, we will have to learn a lot of stuff about how container sandboxing works in order to safely expose the hardware interface
Integration into SpiderLightning may cause problems	Medium	Medium	As much as possible, work by extending SpiderLightning instead of working separately and integrating our changes

Table 2. Technology risks and mitigation.

Skill risks associated with the project originate from the incorrect understanding of tools used such as programming languages and concepts. Listed in Table 3 are the major skill risks involved in the project.

Risk	**Likelihood**	**Impact**	**Risk Mitigation**
Rust programming language	High	High	Completed Rust introduction course through official Rust documentation
WebAssembly	Low	High	If it turned out that manually writing WebAssembly code was necessary, we would have consulted the documentation and our advisor (but we did not expect to need to and this expectation turned out to be correct)
Linux (openSUSE)	Low	Medium	Used a different distro that we all already knew how to use (i.e. Debian)
Interface design	Medium	High	Iterated based on feedback from the client

Table 3. Skill risks and mitigation.

V. Definition of Done

The project is considered "done" when our interface and implementation have been submitted as a pull request to the main SpiderLightning repository, along with a video of our demo applications working. This implies that all of the features specified in the project description must be supported by both the interface and the implementation. We ended up not completing each task originally given and reached out to the client and ensured that our project was done to his standards. We determined that it was out of scope of the time given for the project to do some of the goals given and decided instead to take on a stretch goal that seemed simpler. The working demo applications constitute our primary test for completeness and correctness, as well as for proper integration with the rest of SpiderLightning, since they rely on the existing message bus interfaces. This was delivered at the end of the five-week session via a GitHub PR.

VI. System Architecture

High-Level Design

The implemented system looks to add a GPIO interface and implementation onto the already existing SpiderLightning library. As a result, the system architecture largely matches that of SpiderLightning and Wasm interactions. Outside WebAssembly applications and programs communicate with the WIT interface, which acts as an entrypoint to SpiderLightning's added functions. Then through the Webassembly Engine, the WIT pulls corresponding calls to the code provided from the capabilities system. Various submodules then provide the function implementations for which the capabilities give. Lastly, various rust crates provide abstracted lower level code to be used in the implementations.

A visualization of the design is shown below in Figure 1. Elements on the left side of the diagram are in WebAssembly denoted by its logo, whilst elements on the right side are in Rust in a similar manner. Elements marked with an asterisk are modules modified or added by the team in order to fulfill the project requirements.

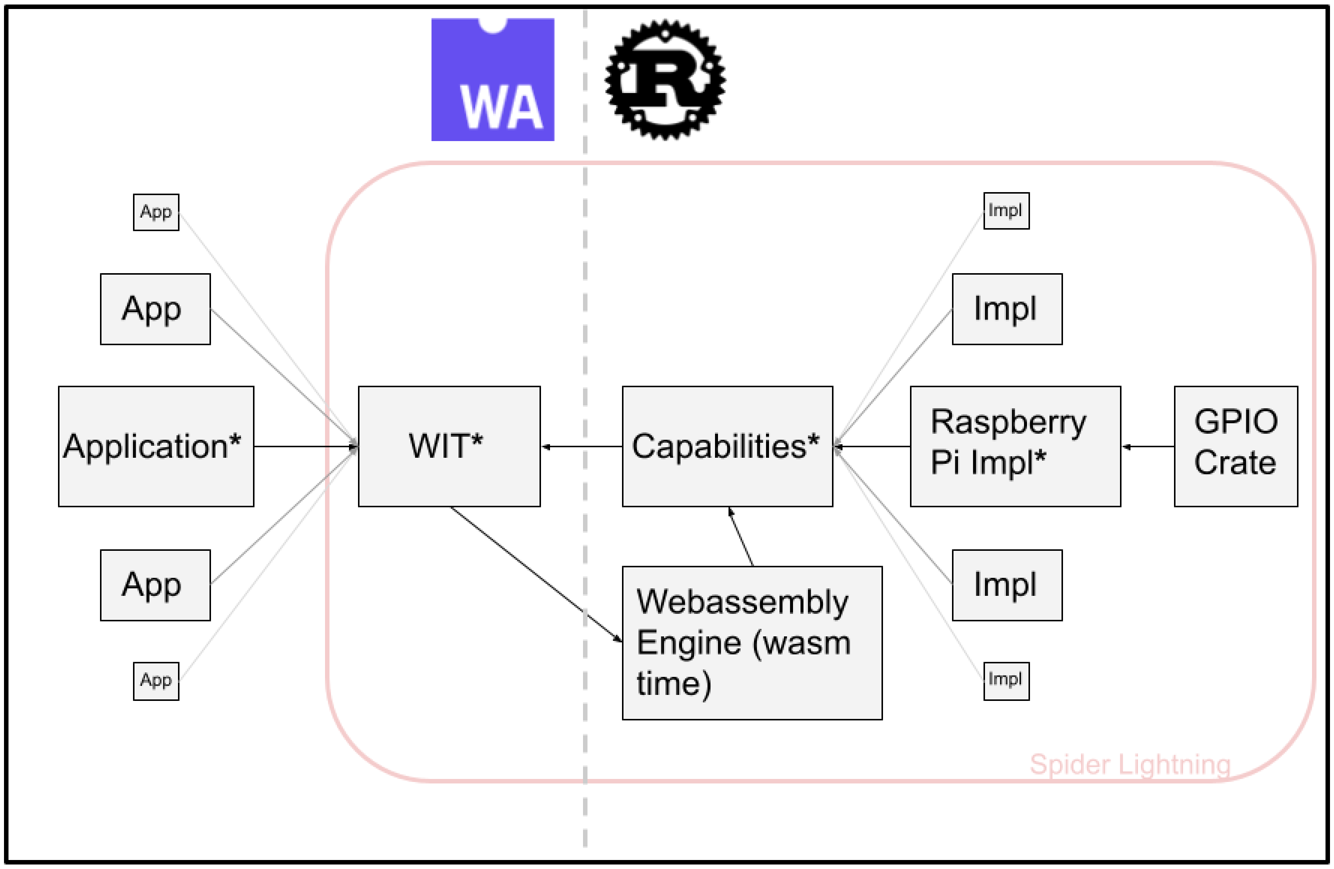


Figure 1. SpiderLightning System Architecture.

Detail Design

Application and WIT

The applications are outside programs that make use of the capabilities provided by SpiderLightning, done by communicating with SpiderLightning's WIT. A WIT is an interface in which Wasm uses to provide tools to a developer. It is an IDL that makes the developers life easier by describing imports and exports. It also provides a basis of sharing types and definitions in an ecosystem of components. We created a demo application to test our code and show our modification of adding a Raspberry Pi implementation.

Capabilities and Wasm Engine

The Wasm engine is what runs the WIT. WIT is inputted into the Wasm engine and creates capabilities that can be used with implementations to complete a task for the applications sake. This in turn, allows a developer to have easier access to the data they are working with causing more efficient labor on the developer side. We modified the existing capabilities of SpiderLightning to fulfill the project requirements in GPIO functionality.

Implementations and GPIO Crate

The implementations (impl in figure 1 above) uses the Rust programming language and look to define the functionality of the various capabilities. Existing implementations in the SpiderLightning library include keyvalue, messaging, SQL, and various HTTP functionality. Each of these implementations use respective Rust crates to assist in providing the implementations, however is abstracted in the above visualization (Figure 1) for project clarity. To realize the project's goal, a new implementation is to be added carrying out various GPIO functionality. GPIO crates such as rppal (v0.14.1), gpio-cdev (v0.5.1), or sysfs_gpio (v0.6.1) were used in this implementation. We decided to use rppal due to its high quality documentation and it had functionality for our minimal viable product (MVP).

Interface Design

The interface we ultimately designed was built to maximize simplicity and extensibility. Application code references input and output pins by name rather than by number, in contrast to most other GPIO APIs. The mapping of names to numbers is defined in slightfile.toml, a SpiderLightning-specific configuration file. The configuration also describes what each pin is used for and any mode-specific options, such as whether to use pullup or pulldown resistors for an input pin. This makes it easier to write application code that works across multiple hardware setups, since all of the hardware details are abstracted behind the named configurations. Additionally, defining upfront what each pin should be used for means that the application can fail fast if a configuration is not supported by the hardware.

Technical Issues

Creating the design of the system architecture had relatively few issues as the project only aims to expand already functional software by adding more submodules. Pre-existing submodules also helped us place our code in accordance with the library. However, the project did run into some issues concerning the communications between the WIT and GPIO modules, specifically function triggers on rising and falling edges with configurable debounce time which were part of the original project requirements.

A feature that our client had initially requested as part of the MVP was the ability to designate edge-triggered functions that would run when the logic level read by a particular pin changed. However, this turned out to be far more complicated to implement than we or the client anticipated. One could envision an API where the user would call a particular function to register their own function pointer as a callback; however, WIT interfaces do not support function pointers, so there is no way for us to declare such a function. Indeed, most of the other SpiderLightning interfaces use blocking I/O rather than callbacks; the exception is the HTTP server, which instead relies on a complicated scheme involving procedural macros and unsafe code, both of which are highly advanced Rust features, to implement an inversion-of-control system in the SpiderLightning runtime. The client additionally wanted these edges to be debounced, which would have introduced further hurdles to this already-difficult task.

We determined that it would not be feasible for us to build a similar system in the short time available, particularly given our team's general lack of experience with Rust. We brought this issue to the client's attention, and he encouraged us to explore alternatives; he also agreed to consult his colleagues in the hopes that someone more experienced would be able to advise us. In the end, one of his coworkers agreed to take on the task of implementing this themself, acknowledging that we lacked the experience or the time to contribute meaningfully.

VII. Technical Design

Our product is built on top of SpiderLightning which consists of a Wasm runtime, WIT files, and function implementations. To realize the GPIO interface, code was added into the slight files, which are config files of SpiderLightning. We also had to modify the .toml files and resource.rs files in the same manner as other SpiderLightning implementations to get the software to accept it as a valid interface and compile.

Our interface is built on top of the above system architecture in Figure 1. We created our GPIO implementation specifically for Raspberry Pi, but it is written in a way that would allow for easy modification for new hardware to run it. In order for SpiderLightning to accept our new interface, we had to add our GPIO functionality by creating a gpio.wit file which the wasm engine can interpret using SpiderLightning's architecture. We also had to add our GPIO object into the lib.rs and resource.rs file of the slightfile used by SpiderLightning. In order to give the slightfile GPIO functionality, we created a gpio crate which used the rust crate, rppal, to run our desired functions. Once we had our interface implemented, we created a demo application that had a mock gpio object created at runtime to ensure proper results on a breadboard for testing PWM and basic flickering of an LED.

By creating our interface using the slight file we were able to ensure that the pin state was configurable at platform initialization. Originally we attempted to code the implementation inside a standard rust file and this would not meet the requirement. Through setting the pin number in the slightfile, we were able to run our tests for our code.

Code Structure and API Design

Implementing the functionality of the project requires the demo application to utilize the added GPIO capabilities. As a result, the form that gives the functionality to the user must be defined, or in other words, the application programming interface (API) must be designed. Research and initial thoughts found 4 different designs the API of our project could follow:

- Exclusive pin objects
- Non-exclusive pin objects
- GPIO objects
- Arduino-style static calls

These API designs are commonly found in other GPIO related software and specialize in specific applications. Discussions were held in relation to which API design should be used. Using the SOLID principles of software engineering, the API designs using GPIO objects and static calls were taken out of consideration. Violating the Open-Close principle and Dependency Inversion principle, these designs would be hard to implement in such a way that would be expandable to other hardware.

This leaves either the exclusive or non-exclusive pin object designs to be implemented, where exclusivity refers to having either a single reference or multiple references to an object. A comparison diagram is shown below in Figure 2 for better clarity of the two designs. To decide on which to use, further discussions amongst the team were held. It was found that both designs easily translated into rppal objects, and fit with how the WIT interacted with the Wasm program.

The difference between the two designs was narrowed down to how each was made during the Wasm program initialization. Discussing this with the client, we confirmed that the slightfile of the Wasm program was responsible for creating the pin objects, thus having multiple references to an object would be easier to implement. It was then decided that the API takes a non-exclusive pin object design.

One concern that we had had about non-exclusive pin objects was the possibility of accidentally referencing the same pin in multiple places that would interfere with each other, but once we decided that applications would construct pins by name rather than by number, we realized that this would make it much easier to search the code for places where a particular pin was referenced, alleviating this concern.

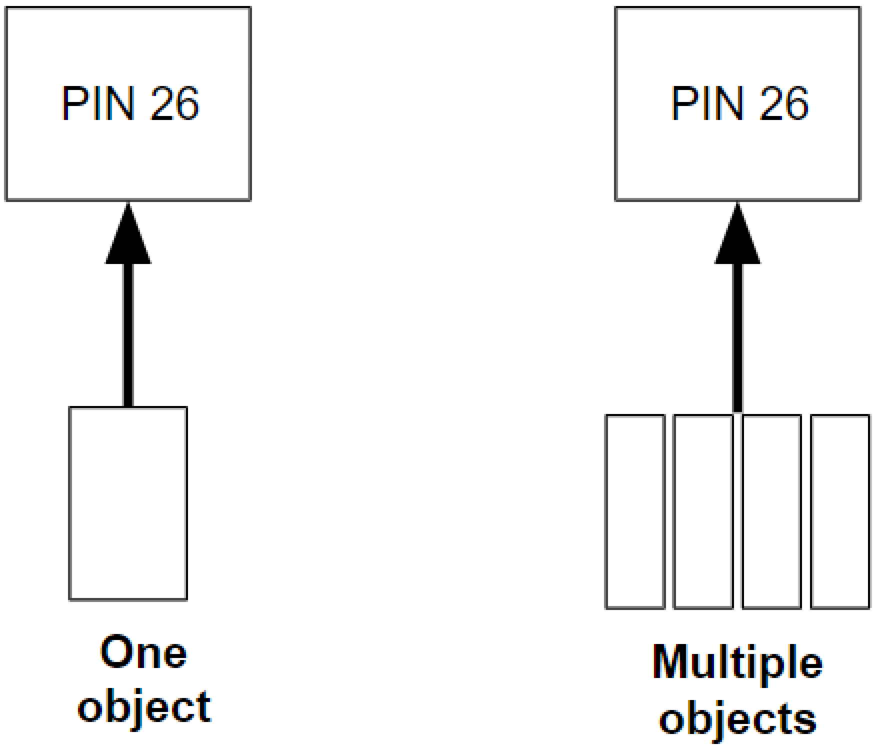


Figure 2. Exclusive pin objects in relation to non-exclusive pin objects.

Another point of interest was how each object was formed using this design. Implementation of the API originally used a combination of signed integers (i32) representing the GPIO pin. This was later changed to be a single string which would then be parsed for general pin information. This allows for more GPIO functionality to be added later on and a more consistent configuration input into the slightfile.

Data Flow

Figure 3 below is a simplified overview of how data moves between the application and the GPIO implementation. It ignores the abstract capability structure that would otherwise sit between the WebAssembly interface and the rppal library code in order to more clearly show how the interface and library work together in our Raspberry Pi-specific implementation. It also does not include Pulse-Width Modulation (PWM) since we made the diagram after completing the MVP; however, the implementation of PWM largely follows that of the input-pin and output-pin resources.

When the runtime is initialized and the GPIO capability is loaded, it reads the slightfile and uses the configuration to construct rppal objects representing the pins used by the application. The application code then retrieves handles to these pin objects by name and calls methods against these handles to interact with the hardware through the rppal objects.

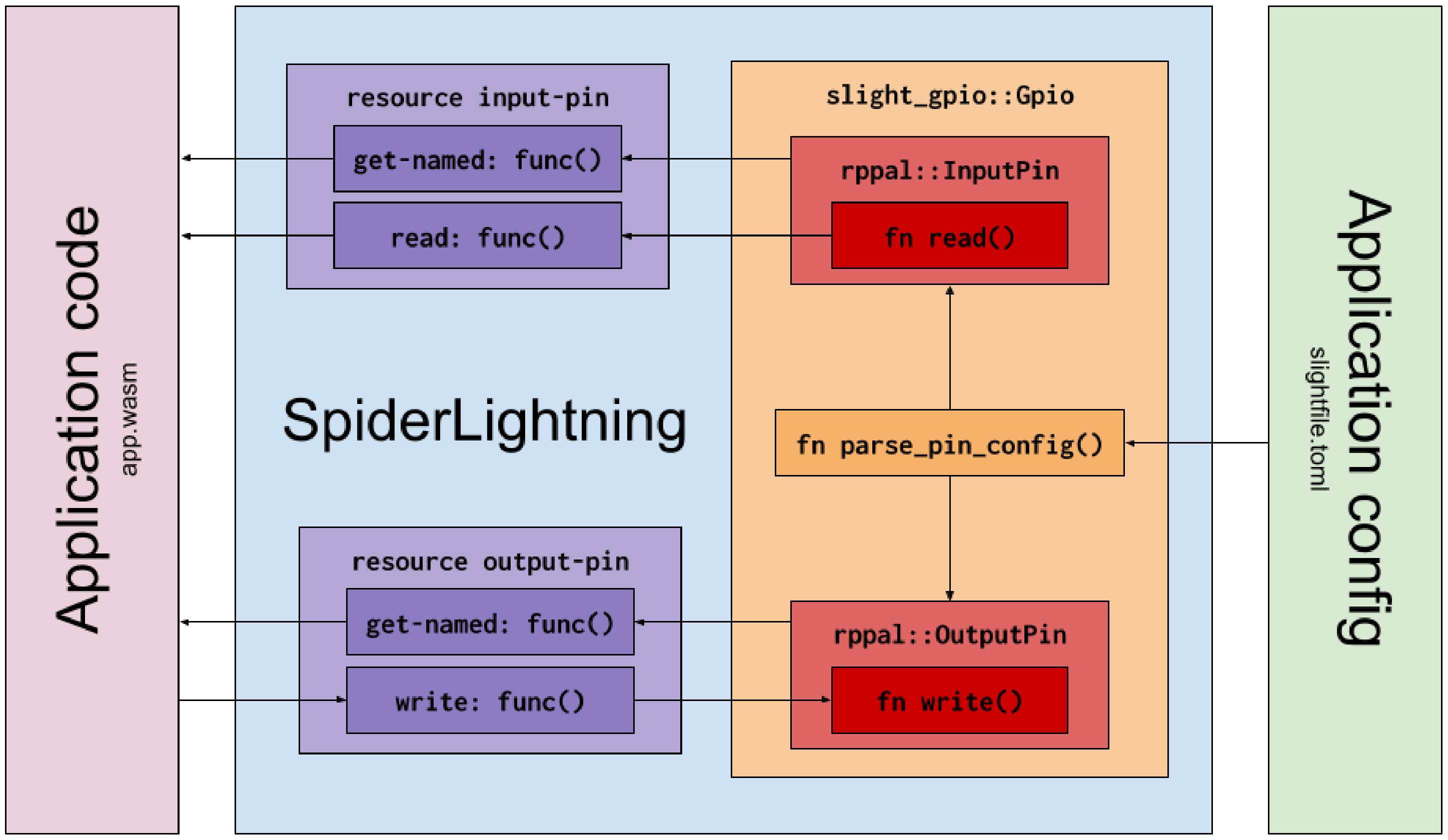


Figure 3. Data flow between the application and runtime.

Our interface is designed to work by separating input and output pins as different types so that an input pin cannot be used accidentally for the functionality one would expect for an output pin. This design helps to ensure proper usage of the program through more explicit coding for the actual application and minimized impedance mismatch of pins on the GPIO. The design also ensured smooth library usage, making the program run without bugs or wasting additional resources. It considers the input and output pin types as separate objects for the function call but the same for the application. Finally, the method we used was cleaner and much simpler to design, increasing overall readability.

VIII. Software Test and Quality

In order to ensure that our code is both functional and well-written, we made use of three different approaches: **unit testing**, **functional testing**, and **code review**.

Unit Testing

Unit testing is an important component of software quality assurance; unit tests themselves are a simple way to verify the functionality of individual pieces of code. Rust has great built-in support for unit testing; the `cargo test` command scans a project for functions annotated with `#[test]`, and the standard library includes multiple assertion macros that test functions can use to validate results. However, unit testing is not a great fit for testing interactions with GPIO hardware; anyone else running our unit tests would need to have their pinout set up exactly like the test expected. Since unit tests are generally run automatically on systems that don't have GPIO capabilities, this would mean that our tests would usually fail even if the code itself worked as intended. In fact, most of the existing SpiderLightning modules have few to no unit tests, since they also involve interactions with external systems. Our unit tests were therefore rather narrow in scope: specifically, we unit-tested the configuration parsing to ensure that it correctly understood properly formatted configurations and rejected erroneous inputs. We ran these unit tests we had whenever we pushed to our repository so that we understood the state of the code. Unfortunately, the SpiderLightning repository does not include a continuous integration setup, so we were not able to automatically run the tests.

Functional Testing

The most important testing we performed is functional testing: manually observing the behavior of application code designed specifically to exercise the project's functionality. For each feature of the project, we have a specific testing procedure involving a demo application written to allow us to test the functionality of that feature. The procedure specifies what should be connected to the pins of the Raspberry Pi we are testing with, what the tester should do while the demo application runs, and what behavior the tester should expect to observe. The intent is for anyone with a Raspberry Pi and the other necessary components, to be able to run the tests in the same way. While the functional tests take more time and effort to run than the unit tests, they will ultimately be more valuable to us since they can be used to validate the hardware interactions that are central to our project. Accordingly, it was not feasible for us to perform the functional tests as often as the unit tests, but we still ran them before calling any particular feature "done," and before demonstrating or submitting the project.

Code Review

Our last major approach to ensuring a well-written product was conducting code reviews onto any changes or additions. To ensure that our code is high quality, a system of branching and pull-requests was implemented within the team. By using an upstream branch `main`, a staging branch `gpio`, and feature branches such as `gpio-dev` for the initial implementation and `gpio-pwm` for the PWM stretch goal, we then implemented version control and pull requests. This allows for all of the team to review code changes and be on the same page when a pull request is merged. It also allowed for the team to notice any bugs unnoticed by the one programming at the time that may impact the code's functionality. Additionally, peer programming was used to help the process of writing code and catching potential bugs in the additional changes.

IX. Project Ethical Considerations

For the team to succeed, the project must also be ethical in both content and conduct. The team and the overall project saw that all sections of the Institute for Electrical and Electronics Engineers (IEEE), and the Association for Computing Machinery (ACM) Code of Ethics should be followed. The project dealt with some ethical considerations more than others though, thus it was essential that a more extensive look into these considerations was conducted. The following ACM and IEEE principles highlight the ethical considerations most applicable to the project:

- ACM 1.3: "Be honest and trustworthy."
- ACM 2.2: "Maintain high standards of professional competence, conduct, and ethical practice."
- ACM 3.6: "Use care when modifying or removing existing systems."
- IEEE 3.07: "Strive to fully understand the specifications for software on which they work."
- IEEE 3.11: "Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project which they work."

ACM 1.3

The principle stated here is crucial to maintain as honesty in both code documentation and team progress prevents false progress from being reported. This allowed for the team to properly adjust the scope and time frame needed. It also allowed for a securer relationship between the team members in the capabilities one can do. Failure to abide by this principle meant hiding information from the client and other group members which could have led to larger problems in the future and a false reliance on code functionality.

ACM 2.2

The team needed to maintain high standards in professional competence and conduct to both make a suitable team environment that promotes work efficiency, as well as effectively communicate with the client about any significant problems. This helped the team complete the project within the time frame and maintain a positive relationship with the client. Violation of this principle could have left the client unsatisfied with our performance even if the project were to be completed.

ACM 3.6

This principle states that one should be careful when modifying existing systems. Our project aimed to add an additional GPIO interface onto the SpiderLightning software, thus the team had to be careful when handling existing code. Implementing code that achieves the goals set by the project description but also removes functionality from other parts of SpiderLightning would in contrary, set back the state of SpiderLightning. Failure to follow this principle would have resulted in unnecessary time spent debugging and risk the additional GPIO capabilities being discarded.

IEEE 3.07

To be successful in this project, the team needed to first understand both the existing software and the tools used to build the software. This allowed us to efficiently create and document code that adhered to the current style of SpiderLightning. Understanding the tools required for the project, such as Rust and Webassembly, was also a necessity to understanding and implementing the project's concepts. If this principle were to be violated, the team may have failed to implement the correct requirements needed by the client.

IEEE 3.11

This principle was also important to uphold, especially because of the nature of our project. Because we were building off an already existing and open-source project, other people will need to be able to follow what we programmed. Having adequate documentation will help them in this process. Additionally, documentation of significant problems helped in explaining unimplemented tasks to the client as well as other SpiderLightning workers. Failure to document the code or any significant problems could have resulted in possible unnoticed bugs that can affect the customer's experience using the additional GPIO capabilities.

Microsoft Open Source Code of Conduct

It was also important to follow the pre-existing guidelines that SpiderLightning already follows. As SpiderLightning is an open-source project sponsored by Microsoft, the software adheres to the Microsoft Open Source Code of Conduct. It was important that our work on the SpiderLightning software follows those listed principles and upholds a positive environment for those who continue to work on SpiderLightning and those that use the software in future projects.

Michael Davis questions

The Michael Davis questions were applied onto the wider scope of this project and furthermore, used in implementing the project's requirements. These questions allow for a perspective into the results of the product delivered to the client.

One test that we applied was the **reversibility test**: we considered the impact that our work extending SpiderLightning might have on other players in the broader WebAssembly ecosystem. For existing users of SpiderLightning, our work is pretty clearly a net positive: it allows them to do more things than they could before. However, SpiderLightning is far from being the only Wasm runtime/API project, and so we additionally considered the effect of extending SpiderLightning specifically might be on users and developers of other Wasm runtimes. We recognized that adding unique features to SpiderLightning would help it compete in the market for Wasm runtimes, potentially being the only viable option for some users if other runtimes don't or can't prioritize GPIO support; what would the ripple effects be of putting SpiderLightning in this more dominant position? We determined that SpiderLightning's status as free and open source software mitigated the risks to users that this might pose: whereas dominant software products sometimes act against users' interests by charging users more money or imposing restrictive licensing terms, SpiderLightning is currently distributed under the Apache License 2.0, which allows it to be freely redistributed, preventing this sort of anti-consumer behavior. Additionally, it allows developers of other Wasm runtimes to incorporate part or all of our work on GPIO, including the API design and the actual implementation code, making it easier for GPIO to become a more standard feature across the Wasm ecosystem, giving users more freedom in their choice of Wasm runtime.

There is still a situation where our code could be used to enable anti-consumer business practices: if our client or a third party were to create a closed-source fork of SpiderLightning (the Apache License permits this, in contrast to "copyleft" licenses such as the GPL), extend their fork with proprietary features, and use the fact that their fork includes all the features of SpiderLightning (including GPIO) *and more* to gain a dominant market position that allows them to shake down users. We determined that the risk of this was minimal, especially from our client: SUSE is in the process of donating SpiderLightning to the Bytecode Alliance, a nonprofit focused on using WebAssembly and related technologies for the public good, giving us confidence that they are acting in good faith and the public interest.

Another test put into consideration was the **common practice test**, specifically regarding the quality of our code. In consideration of other SpiderLightning developers, it would be unfair to provide code that does not match the standards already set in place. If the wider scope of SpiderLightning developers were to produce low quality code, then the software would become hard to maintain and use in future WebAssembly applications. Furthermore, the SpiderLightning software is set to be added into the larger Bytecode Alliance collection. If the majority of SpiderLightning developers were to make poorly-written code, then other repositories in Bytecode Alliance could be portrayed negatively and lower Bytecode Alliance's future contributions.

X. Results

Ultimately, we were able to meet all of the minimum requirements of the project (after edge-triggered functions were removed from the list) as well as the PWM output stretch goal. Our Wasm demo application was able to interact with the GPIO pins through the SpiderLightning runtime. However, we were not satisfied with the mere appearance that our project was complete, and so we subjected our work to the previously-outlined testing procedures we had decided on.

Unit Testing

The unit tests were conducted using Rust's built-in testing framework as mentioned before. Here, unit tests concerning both valid configurations and invalid configurations for pin initialization were run. Additionally before testing, each unit test written was tested off of a stub function, which had given us failing tests at first. Once the code for the specified functionality was written, the unit tests were run to check for any errors. This allowed us to rule out bugs that had been in the program, specifically bugs regarding pin mapping and the initializer for the GPIOImplementer.

The inputs in each unit test were written in the form of expected slightfile.toml configurations. That is, each input to test the pin initialization was in the form of `pin_number/pin_mode/mode_option`; for example, `7/input/pullup`. Valid configuration inputs tested the compatibility between different pin numbers, pin modes, and input modes if given. These were then compared to the pin map defined in the GPIOImplementer, which would pass if values matched, and fail if not. Invalid configurations tested various combinations of inputs into the slightfile.toml configurations. The following list described external cases tested:

- Invalid pin numbers
  - Large pin numbers
  - Negative pin numbers
  - Invalid types (strings, floats, etc.)
- Invalid pin mode
  - Strings not input or output
  - Invalid types
- Invalid mode option
  - Strings not pullup, pulldown, low, or high
  - Invalid types
- General edge cases
  - Blank configuration
  - Configurations with missing information
  - Random strings and conflicting information

These cases were inputted into the pin initialization functions to test if the function correctly gave the expected output, specifically a `gpio-error` variant type found in the lib.rs file.

Functionality Testing

The results of our functionality tests showed completion of the goals that we found to be reasonable for the scope of the project. Also, by developing an implementation for PWM using a new output type, we reached one of the stretch goals defined by the client. Following each new feature of our code, we made sure that it worked on our Raspberry Pi device using our modified SpiderLightning.

Our first demo involved making a flickering LED with a changeable speed by holding down a button. This showed our completion of the required functionality of the MVP of querying the pin state and having a configurable pin state at platform initialization. Our second demo involved a LED that changes brightness over time based on a different button being pressed. In this demo, the LED gets brighter over time when the button is held down, and dimmer when the button is let go of. This demo showed completion of the PWM pin definition and manipulation stretch goal defined by the client. For more information regarding how the hardware was set up to perform these demonstrations, refer to Appendix B.

To ensure our demos met the project requirements, a streamlined testing process was utilized. Firstly, a functional demo was written that displayed a working part of the project requirement. Next, the demo was documented to both include expected and actual results when run. Finally, additional edge cases regarding hardware interactions were conducted to rule out any bugs that may still be present in the software.

XI. Future Work

Though our work here is done, there is still quite a bit of GPIO-related functionality that our interface does not cover, and that could be the basis for further development. The most important feature that we could not build was the ability to trigger functions on a rising or falling signal edge, with configurable debouncing. This had originally been part of the minimum viable product that the client requested, but after some investigation, we discovered that this feature would be massively more complicated than the client had realized, and far beyond our ability to design and implement, especially given the short timeframe we had available. After we discussed this with the client, he consulted with an expert coworker who confirmed our assessment and offered to add this feature themselves after we submitted our work.

An additional way in which our work could be extended is by adding support for serial communication protocols such as I^2C, SPI, and RS-232, which are commonly used to interface with more advanced peripheral devices, such as sensors and other data collection instruments. The ability to use these from SpiderLightning would therefore massively expand its utility and allow for more sophisticated applications. A viable starting point for this addition would be using the rppal crate we used in this project to extend Raspberry Pi functionality. This library includes I^2C and SPI support for Raspberry Pi and can be used similarly to the PWM module implemented.

Also our client asked if we would like to be a part of a code review for our project when our field session is over. This would involve a few of his colleagues giving professional advice to what should be improved from on our project. By completing this process our pull request would be accepted and this would mean our code would become a new part of SpiderLightning. This process would involve simple changes to our existing code throughout the next couple of weeks. Our client told us that for peer reviews it tends to be fairly harsh. The peer review process is future work that we plan on actually doing assuming that the client still decides this is the best course of action in the next few weeks.

Recently our client also offered us future work on the project in the form of a short internship for the rest of summer where we would presumably be implementing the rest of the features in the project or a different assignment. We have reached out to our client and showed an interest in this experience that we could have in the near future as this would be a great experience for us all to continue what we have been working on. We have already spent the time learning how most of SpiderLightning works so it would be easier than if we had been just starting. This way we can potentially implement the rest of the MVP and stretch goals of our project by the end of summer.

Finally, future developers could add support for GPIO devices other than the Raspberry Pi. This would allow application developers to run their same code on multiple platforms, leveraging the portability of WebAssembly. One Rust crate that could be used here is the sysfs_gpio crate, which accesses GPIO under Linux and gives methods around manipulating the Pin struct. Another crate worth mentioning is the gpio_cdev crate, which gives access to the GPIO character device ABI.

XII. Lessons Learned

As a team we have learned a lot through this course. We worked on client communication and developed skills in Rust and now have a better understanding of WebAssembly. It was interesting to understand how workflow works for products that are for an actual workplace environment. Looking into SpiderLightning's github it was intriguing to see how actual developers push out code and how they document it. This is the first actual workplace code most of us have seen. Learning that some goals do not need to be reached in specific time spans is a very different perspective than we are used to in the educational background. In our current experience you are supposed to complete all work by the due date or you are deducted which is beneficial for being a productive worker in the future, but in the workplace it is difficult to determine how much work can be done in a set time as explained by some guest speakers and work on the project.

Another important lesson we learned was to keep the changes in a pull request scoped to the specific issue the pull request is intended to address. Our pull requests would frequently be laden with minor, offhand changes to unrelated parts of the code, making them more difficult to understand and review. Worse, this could sometimes lead to merge conflicts: since nobody could be entirely sure what parts of the code everyone else was likely to touch, we would sometimes run into the situation where two people touched the same part of the code without either person knowing the other was going to work there. This slowed the pace of development by forcing us to resolve these merge conflicts. Eventually, we realized that the proper procedure was to create a new ticket for every change that needed to be made, even minor ones, so that someone could be assigned that ticket and everyone else would know what they were working on. This not only addressed the above problems, but also made progress tracking easier, since we could know exactly what had gotten done by looking at which tickets had been marked as completed.

More generally, we realized partway through the project that we needed to get way more organized if we wanted to have any chance of finishing on time. Some of the early tasks that we had to complete — for example, familiarizing ourselves with the languages and technologies involved, and learning to navigate the SpiderLightning repository — ended up taking much more time than we had expected due to a combination of factors including poor coordination, miscommunications, and a general looseness when it came to planning. By the beginning of the third week, it became apparent to us that we weren’t on track to finish within the timeframe unless we shaped up. We started paying more attention to Agile organization tools like our kanban board and immediately saw a marked increase in our team’s productivity and effectiveness. This more than anything else drove home the importance of proper organization in completing software projects.

Our project involved a lot of code in which we were not familiar. It was a very beneficial learning experience to see a new codebase which was developed by advanced coders who are actually employed to do so. By learning the inner-workings of SpiderLightning’s WIT files, slight files and standard rust files we furthered our coding skills. It helped us gain technical knowledge and learn how to ask questions to our client on aspects in which we believed that we could not do on our own. It is an uncomfortable space to tell somebody you are doing a task because you think a part is not able to be completed in a given time span. We now understand how to communicate our issues to a client and ensure proper communication and a smooth workplace relationship.

XIII. Acknowledgments

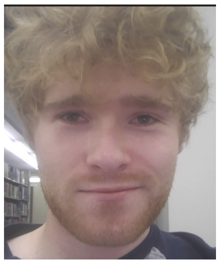
Andrew Gracey:

Andrew was our client who gave us our project. He was a great resource and was very lenient on the requirements and understood our feedback on the scope of the project. When we had issues involving the MVP or the basic structure of SpiderLightning he supported us through email or meetings. Also Andrew was very approachable when we had our meetings with him.

Rob Thompson:

Rob was our course advisor. He helped us with understanding client relations and guided us in making a high-quality product for the client. Additionally, he helped us figure out how to divide up large workloads in the middle of the course where we were not sure how to do our pair-programming. Rob validated our sketches for models and explained required terms for the essay such as the Davis tests which we were confused on how to apply to our project.

XIV. Team Profile



Brendan Burmeister - Main Scrummaster
Senior, Computer Science
Hometown: Littleton, Colorado
Work Experience: N/A
Sports/Activities/Interests: Climbing, snowboarding, basketball.
I am excited to learn more about rust and integrations between it and WebAssembly. I have been interested in integrations and this will be an interesting application of this which will end up being posted to open-source.

Kai Page - Advisor POC
Senior, Computer Science
Hometown: Houston, TX
Work experience: Web development internship at USGS, TA for CSCI 341 at Mines
Activities: Cooking, theater, tabletop roleplaying
I think WebAssembly is really neat and am thrilled at the opportunity to expand its capabilities!

Joey Vongphasouk - Client Liaison
Senior, Computer Science
Hometown: Thornton, CO
Work experience: CSCI 101 and 128 TA for Colorado School of Mines
Hobbies/Interests: Hiking, playing video games
I am eager to learn about WebAssembly and look forward to building upon the current SpiderLightning library.

References

ACM Code 2018 Task Force. “ACM Code of Ethics and Professional Conduct.” *Association for Computing Machinery*, 22 June 2018, www.acm.org/code-of-ethics.

IEEE-CS/ACM joint task force. “Code of Ethics.” *IEEE Computer Society*, 13 May 2023, www.computer.org/education/code-of-ethics.

ikikVC. “What Is Pulse Width Modulation (PWM)? Applications and Accessories.” *Latest Open Tech From Seeed*, 15 Jan. 2021, www.seeedstudio.com/blog/2020/06/16/basic-electronics-pulse-width-modulationpwm-and-arduino-applications/.

“Code of Conduct.” *Microsoft Open Source*, opensource.microsoft.com/codeofconduct/. Accessed 31 May 2023.

r-muhairi. (2023, February 25). *The “wit” format*. GitHub. https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md

Appendices

Appendix A - Key Terms

Descriptions of technical terms, abbreviations and acronyms

Term	Definition
Common Practice Test	*“What if everyone behaved this way?”*
Crate	*A compilation unit in rust similar to a library in different coding languages.*
GPIO	*General-purpose input/output: programmable “pins” on devices such as the Raspberry Pi that can be used to control or otherwise interface with electronic circuits.*
GPIO character device ABI	*An API that deprecates the legacy sysfs interface to GPIOs*
HAL	*Hardware abstraction layer; the interface between software and hardware provided by the kernel.*
I2C	*The Inter-Integrated Circuit. It is used to allow multiple peripheral digital integrated circuits aka chips to communicate with one or more controller chips. Similar to SPI, but is intended for short distance communication with a single device.*
MVP	*Minimum viable product. This is what the client wants us to produce by the end of our working period.*
PWM	*Pulse-width Modulation; Technology that uses a changeable amount of power delivered to a device.*
Reversibility Test	*“Would this choice still look good if I traded places?”*
RS-232	*A standard introduced in 1960 for serial communication transmission of data. It is used by repair technicians to perform diagnostics and service updates.*
SPI	*Serial Peripheral Interface; an electronic interface that provides a serial exchange of data between two devices*
SpiderLightning	*A set of WIT definitions and associated implementations to enable app developers to work at a faster pace and require less knowledge of the environment in which they are executing*
Wasm Interface Type (abbr. WIT)	*A format to provide tools to a WebAssembly component.*
WebAssembly (abbr. Wasm)	*An executable bytecode format specified by the World Wide Web Consortium. WebAssembly provides security and portability guarantees that are necessary for running code in web browsers. It has turned out to be useful in other contexts, since security and portability are both generally good things.*

Appendix B - Hardware Setup and Instructions

Below is the mock hardware setup for the demo used in the project pulled from Figure 3. Note that the software used to create this diagram did not include the “Raspberry Pi 4 Model B” programmable circuit board, which had been our main circuit board for the project. An “Arduino UNO R3” was used instead, however for the demo example used in the SpiderLightning library, a Raspberry Pi model must be used.

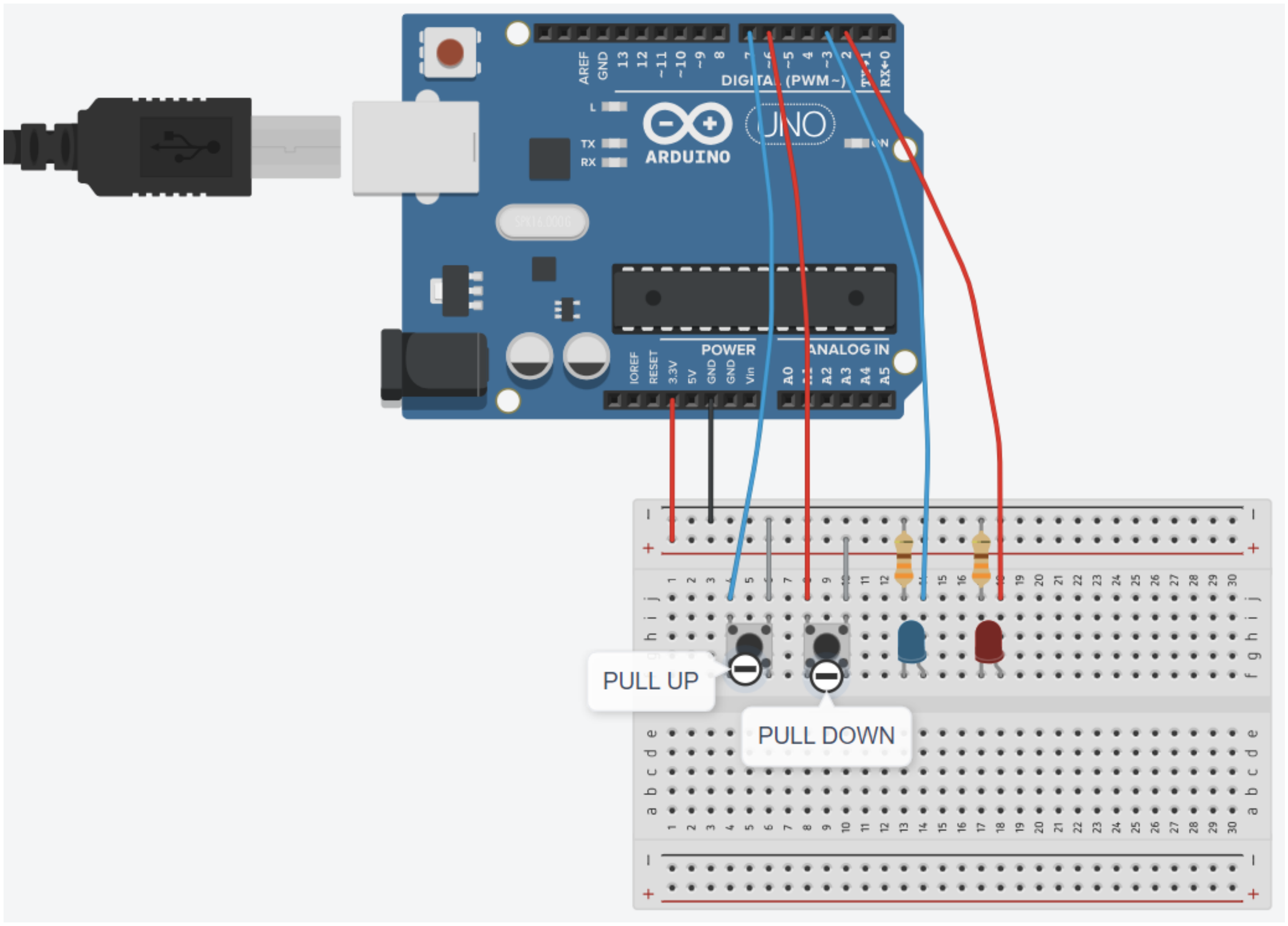


Figure 4. Mock hardware setup of demonstrations 1 and 2.

In the diagram, both demos used in the functional testing are set up. The first demo uses 2 different circuits. The first circuit uses a red LED that is connected to a 330Ω resistor connected to ground. The other side of the LED is connected to a GPIO pin from the programmable circuit board. The second circuit in the first demo uses a button input which is in PULL_DOWN configuration (connected to high, thus GPIO pin must pull down signal). The second demo setup is largely similar to the first demo setup, however the button for the second demo is in PULL_UP configuration.

It is important to note that the functional demos included in the example uses specific GPIO pins in order to function properly. Listed below are the pins that each wire should be hooked up to:

- Red LED -> GPIO pin 5
- Red Button -> GPIO pin 17
- Blue LED -> GPIO pin 6
- Blue Button -> GPIO pin 18

A picture of the exact setup the team used for the project is included below.

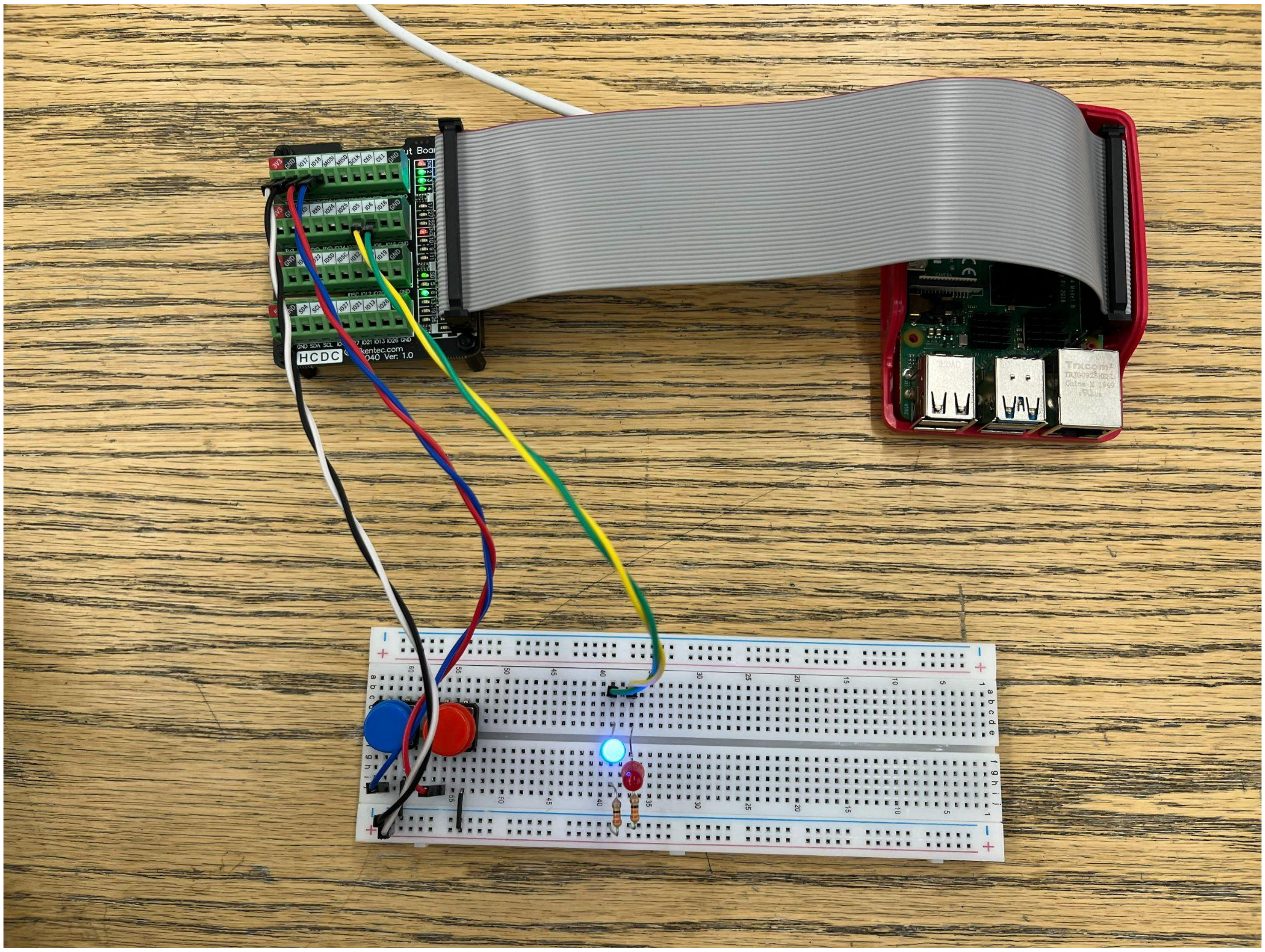


Figure 5. Hardware setup used by Wasm Squad for functional testing demonstrations.