SwiftIO VISA Driver

Client: Dr. Owen Hildreth
Advisor: Dr. Jeffery Paone
Team Names: Ethan Vijayabaskaran, Grant Rulon, Mitch Watkins, Ryan Rumana
Date: June 15, 2020

Table of Contents

Introduction **2**
Client Background 2
Product Vision 2
Requirements **3**
Functional 3
Non-Functional 3
System Architecture **4**
Technical Design **8**
Graphical User Interface 8
Back-End Driver Program 10
Quality Assurance **11**
Unit testing 11
User Interface Testing 11
Integration testing 11
User (Client) Acceptance Testing 12
Code Metrics 12
Code Reviews 12
Results **13**
Testing Results 13
Performance Results 13
Future Work **14**
Appendix **15**
VISA Command Documentation 15

Introduction

Client Background

Dr. Owen Hildreth is an Associate Professor in the Mechanical Engineering Department at Colorado School of Mines. As the head of The Hildreth Research Group, Dr. Hildreth performs research primarily on additive manufacturing technologies of the nanometer to centimeter scale. Professor Hildreth has written various data collection and visualization MacOS applications for his research.

Product Vision

The motivation for the SwiftIO VISA Driver project is to more efficiently and effectively control instruments in Dr. Hildreth's lab using a SwiftIO board and surrounding functionality. The SwiftIO microcontroller is a board similar to an Arduino or Raspberry Pi in which there are a set of pins and ports on the board controlled by the processing unit on the board. The main difference between the SwiftIO board and other microcontrollers is that it runs Swift. Before the project, the SwiftIO microcontroller board could be used to control the instruments' functionality, but without development of the client's desired system, changing the functionality of the SwiftIO board is tedious, as the program running on the board must be edited and downloaded onto the board after the desired code change has been made. Also, Dr. Hildreth must have humans look at sensors and gauges to record data, instead of the system being able to capture data from the lab instruments.

The goal is to develop a MacOS application that connects to the SwiftIO board through a wired or wireless connection. In practice, this application is able to send Virtual Instrument Software Architecture (VISA) compliant commands to the board in a dynamic fashion in order to change the functionality of the SwiftIO board without having to edit and redownload the code running natively on the board. The system also sends back the proper response to the MacOS application after communication with the board has taken place. Because the SwiftIO board does not understand VISA compliant commands, a driver interfacing with the commands on the board translates commands into functionality and back into proper responses. This will then allow Dr. Hildreth to wire the SwiftIO board to an instrument and change what the pins and ports are doing on the board in order to change the functionality of the instrument. For example, if the instrument wired to the SwiftIO board is a furnace, the temperature can be set and recorded autonomously. With these ideas in mind, this project hopes to improve the quality of life for our client and their research groups.

Requirements

Functional

- User interface must be a desktop application with the ability to specify a desired serial port and accompanying port parameters.
- The desktop application must be able to open and close a serial port connection with the SwiftIO board, as well as have text fields for sending and receiving data from the SwiftIO board over the serial connection.
- The user desktop application must run on an Intel-based Central Processing Unit (CPU) architecture successfully.
- The SwiftIO board must use the native Universal Asynchronous Receiver/Transmitter (UART) port standard to connect to the controlling desktop computer, either through a Universal Serial Bus (USB) to UART module or a bluetooth to UART module.
- The SwiftIO board must be able to communicate to the computer through the UART port module to allow for reading and writing through the serial connection.
- SwiftIO VISA driver must have the capacity to parse VISA commands into function calls for functionality on the board.
- SwiftIO VISA driver must be able to send proper responses and error messages back through the serial port connection.
- New SwiftIO communication and execution must have better performance, measured in time to change functionality, compared to existing system implementation.

Non-Functional

- User interface desktop application must be a MacOS application written in the Swift language and developed in the Xcode integrated development environment (IDE).
- Buttons and text fields must be used in the user interface in order to send and receive data from the serial connection. Drop down menu items or other intuitive interactive items must be used to select the proper baud rate, other serial parameters, and tell the user the connection status.
- SwiftIO VISA driver must be written in the Swift language, and built using either the MadMachine IDE or the Xcode IDE in tandem with the SwiftIO Software Development Kit (SDK).
- System must be able to use existing Swift VISA application programming interfaces (APIs) and VISA commands used in Dr. Hildreth's lab to dynamically interface with the SwiftIO board.
- All VISA driver and application source code must be located in a public GitHub repository in the Hildreth Research Group collection of repositories.

System Architecture

There are two main physical entities in the system: the controlling MacOS application located on an Apple computer, and the SwiftIO board itself. These interact in different ways along with various subsystems. *Figure 1* shows these high level hardware and software entities and their continuous flowing relationship. The client's personal computer is the origin of the VISA driver program running on the SwiftIO board, with downloading onto the SD on the board as an initial required step. After this initial download, dynamic VISA communication between the client's computer and the board is completed through a bluetooth module wired to the SwiftIO board's UART port. In *Figure 1*, the blue colored boxes are the previous implementation of interfacing with the board, and the salmon colored parts are the project's additions to the system in order to achieve higher efficiency. With the additions of bluetooth connectivity and a battery pack, the SwiftIO device is controlled and monitored remotely. The singular wired connection is not required for use after the client initially downloads the program for the device.

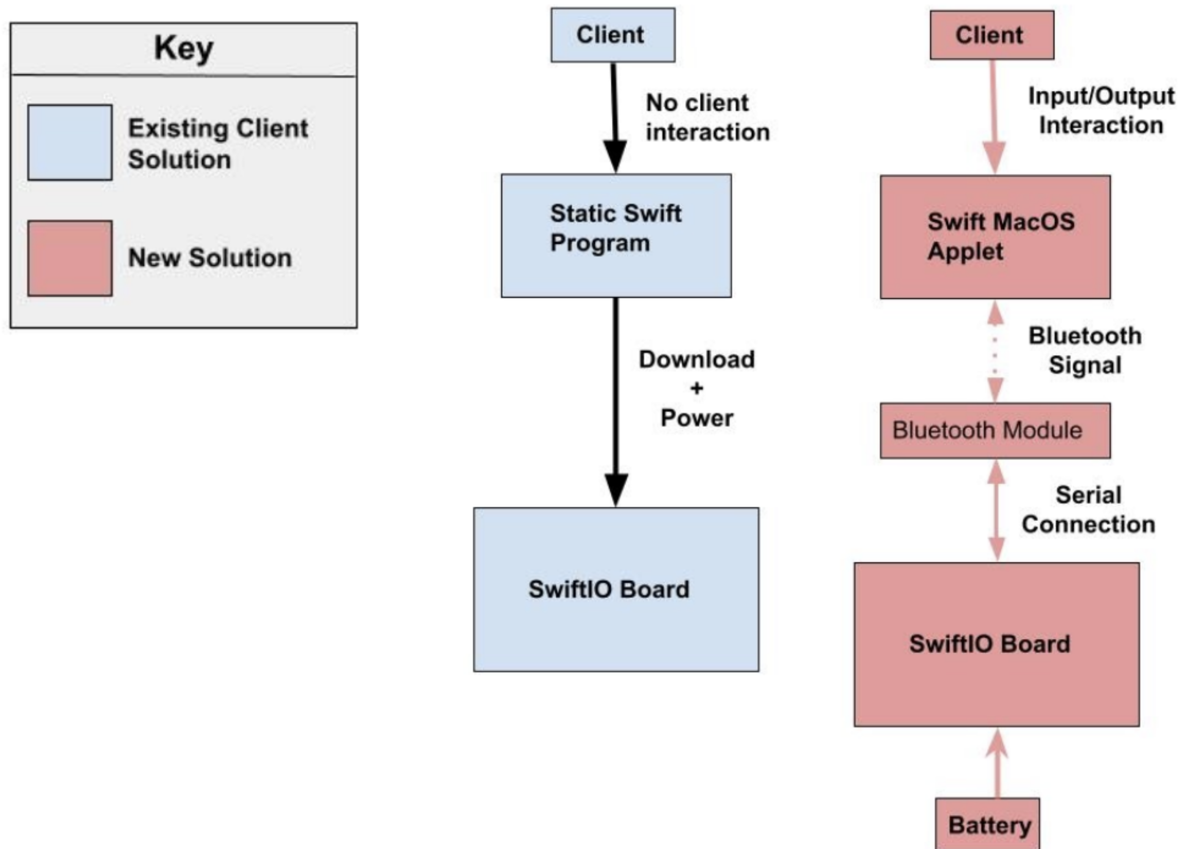

Figure 1: User Interaction Flow Diagram

The client has multiple SwiftIO boards and often uses them to demonstrate functionality in a classroom setting in vibration kits that he has developed which he uses to allow students to study resonance. While each controlling Apple computer can only connect through bluetooth with one module at a time, the client can change the connection between different SwiftIO boards so that only one desktop application can communicate and monitor multiple different boards. As long as the boards are externally powered, the user only needs to switch between bluetooth devices to address different ones with the laptop they are operating on. *Figure 2* demonstrates the capability for the client to remotely connect to any of the SwiftIO boards through the MacOS application.

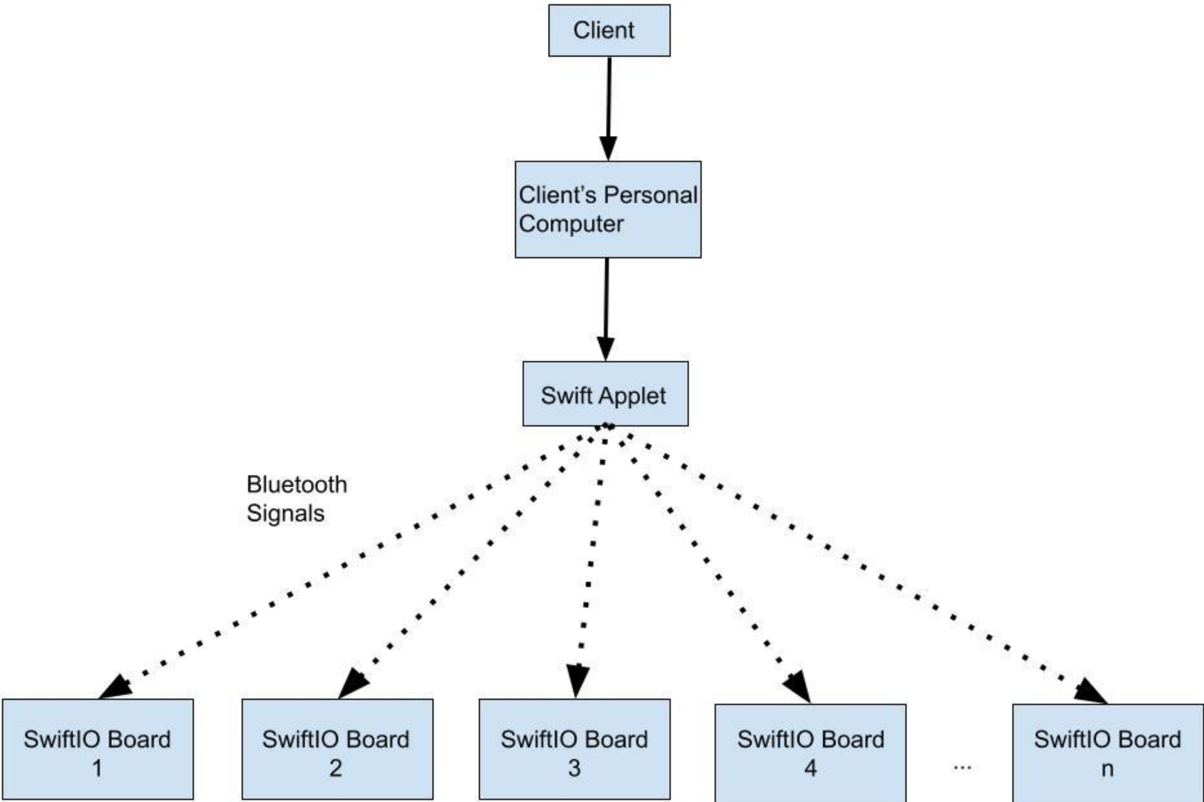

Figure 2: Client Bluetooth Utilization Flow Diagram

Looking at the hardware architecture in *Figure 3*, the bluetooth module is wired to the Arduino Shield to allow wireless serial communication with the computer application. The bluetooth data transfer is done through Universal Asynchronous Receiver/Transmitter (UART) communication system. Other sensors, such as potentiometers, buttons, and LCD displays, can be added to the system as needed by the client. The nine-volt battery is connected to the Arduino Shield, powering the board for wireless operation. *Figure 4* shows a side view of the SwiftIO Arduino shield connected to the SwiftIO board. The SwiftIO board has a series of pin slots on the face of the board in which the SwiftIO Arduino shield fits its pins on the bottom of the shield.

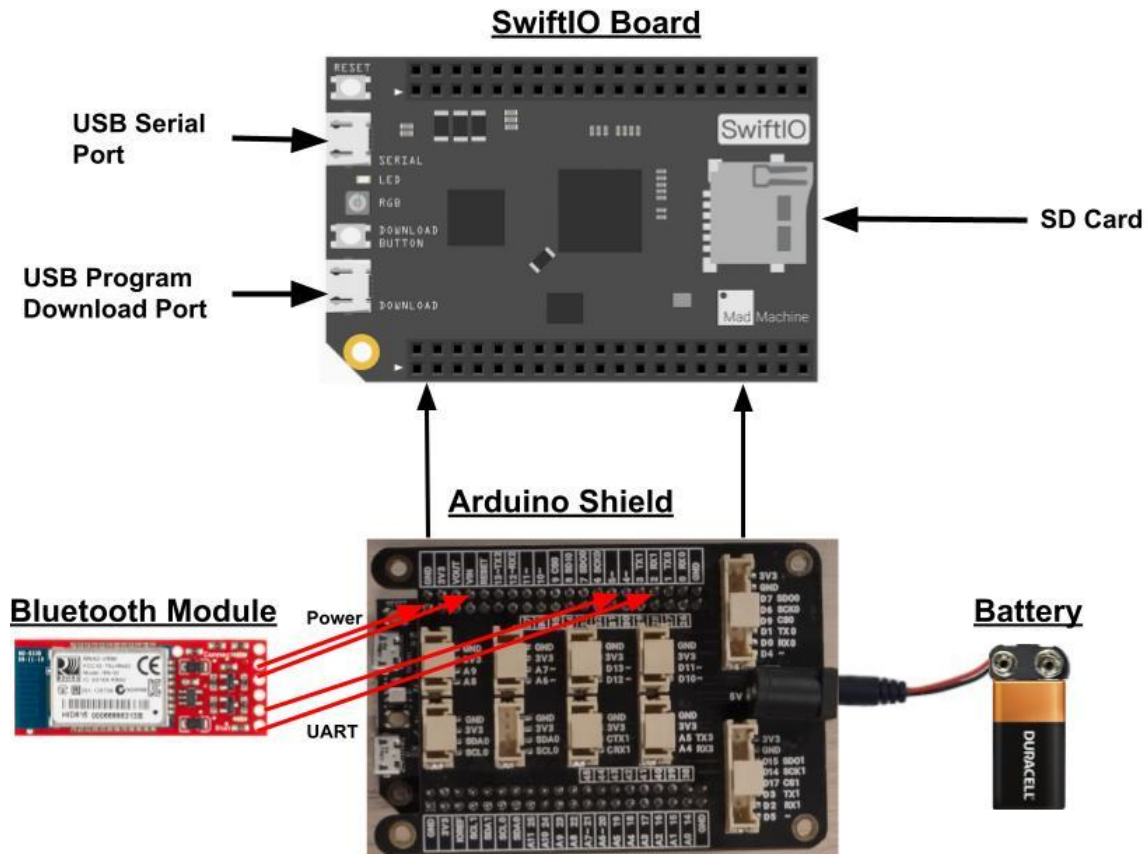

Figure 3: Hardware Configuration Diagram

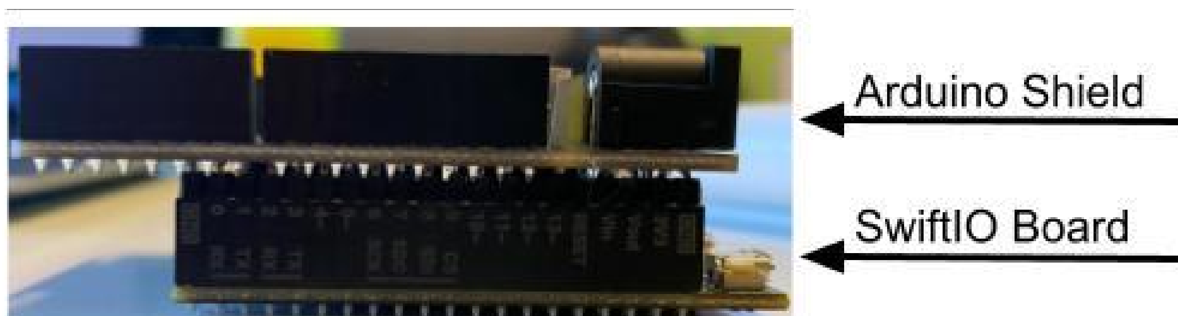

Figure 4: Hardware Orientation

As far as user interface goes, the goal is to provide the client with an easy to use and intuitive system to send commands to the SwiftIO board, as shown in *Figure 5*. While it is a static, barebones application, the UI is much simpler and more understandable when compared to a command line interface, such as a serial terminal. The application has a section to select the proper serial port and accompanying parameters, a section for sending data to the SwiftIO board, and a section for receiving data back from the SwiftIO board. The interface allows for dynamic communication with the board, without having to manually edit the code running on the SwiftIO board.

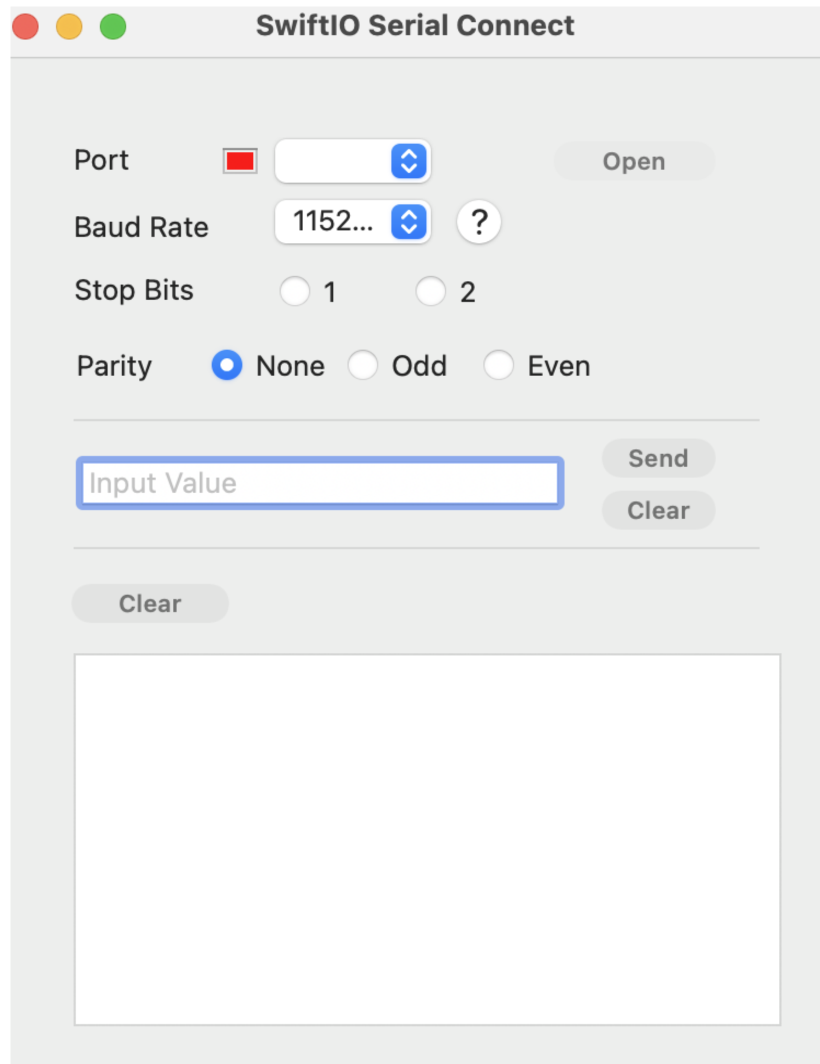

Figure 5: MacOS Application User Interface

Technical Design

Graphical User Interface

The MacOS application is made using Xcode and the Swift language utilizes the Cocoa API. Cocoa is an object oriented API created by Apple for use with its proprietary application development. The project's actual application is made using Cocoa in conjunction with Xcode's interface builder to populate the interface with elements such as buttons, drop-down selectors, text fields, etc. These application elements are connected to methods and instance variables in the Cocoa environment.

Figure 6 shows an example interface instance with a single button, labeled "Open". This button is connected to the code with an @IBOutlet variable which acts as an object within the code for changing and monitoring the button's parameters. Furthermore, this "Open" button is also connected to an @IBAction method, which holds the code that runs immediately after this particular button is pressed.

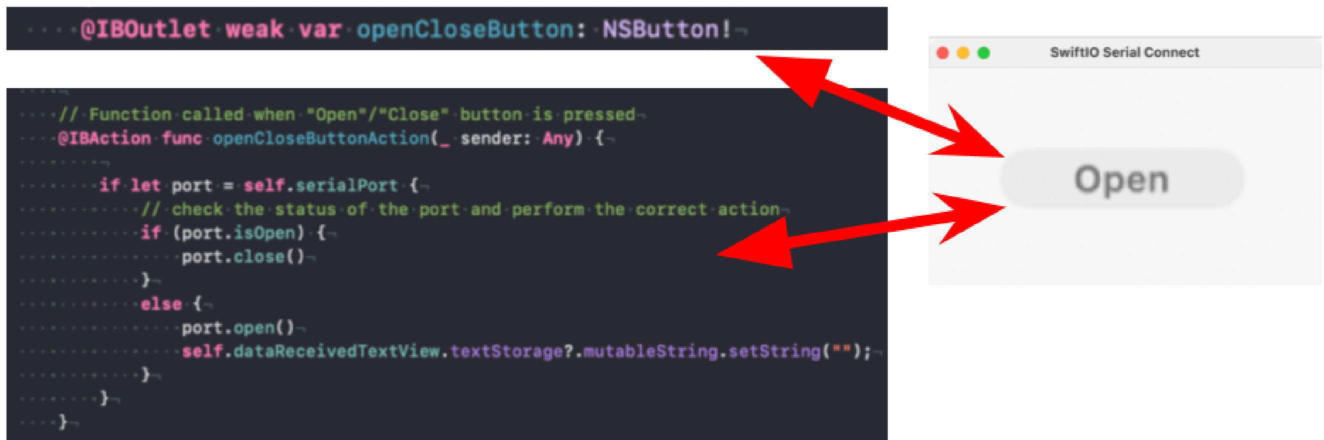

Figure 6: User Interface Button Code Connection

Figure 7 shows a different example interface instance with a single drop-down selector. This element is connected to the application's code through an Xcode binding. The available selector options are populated with names of the available serial ports in the MacOS computer, and the port name that is selected is connected to the serial port instance variable upon the selection of the "Open" button. After parameters from the interface are set and the proper buttons are pressed, the application connects to the serial port of the SwiftIO board as an instance of ORSSerial, a wrapper class for the serial connection. This ORSSerial instance is able to send the specified data to the board and capture the data sent from the board to the controlling computer.

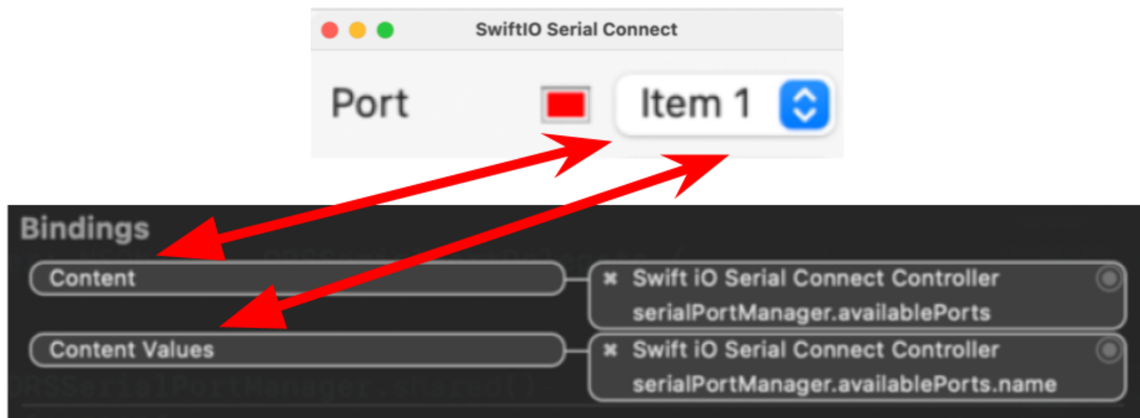

Figure 7: User Interface Drop-Down Code Connection

Back-End Driver Program

Figure 8 shows the UML diagram of the driver program on the SwiftIO board. There are a few additional frameworks inside the program that are integral to the code, however, because they are not developed as a part of this project, they have not been included. The main class only acts as a container for the other classes and to facilitate the communication with the GUI. The program implements a singleton design, where a single instance of the SCPI class contains all of the controls for the board. There are a total of 19 Pins contained in the SCPI class, each with parameters for identifying the current state. The enumerated types pinN and pinT are for simply assigning the pins to a type, and switch statements are being used in conjunction with the enumerated types to efficiently identify pins.

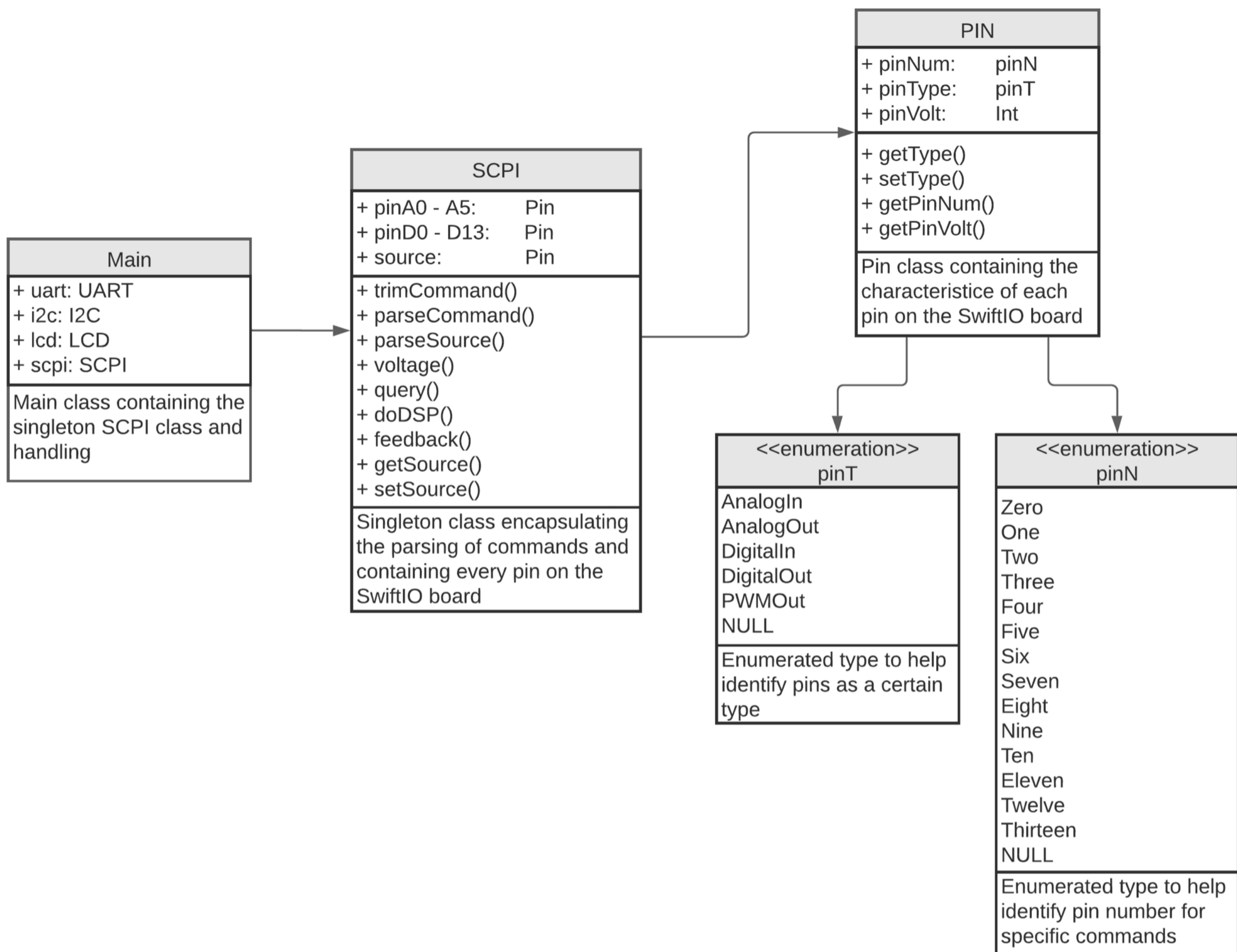

Figure 8: SwiftIO Driver Code UML Diagram

Quality Assurance

Unit testing

The unit testing performed on the project was focused on the low level command processing and data transferring. These unit tests are meant to ensure that the additions to the existing system are built on a solid foundation of security and expected functionality, so that abstractions can be made in a systematic manner.

- Testing serial port reliability and consistency.
  - Test sends strings over serial to the board and prints ASCII codes to a connected liquid crystal display (LCD) screen, and prints a string with the ASCII codes turned into characters.
  - This ensures that the integrity of the data sent to the computer is maintained during the conversion from a string, into individual bytes, and back into a string.
- Testing VISA string conversion into SwiftIO functions processed within the driver.
  - Test sends VISA commands from MacOS application to the board, asserting correct functionality.
  - Test asserts that the correct response or error is written back through the serial connection to the application.

User Interface Testing

The user interface testing portion of the quality assurance plan is testing the interactive portion of the MacOS application. The interface has a top section that is meant to choose parameters from a list of possibilities, in addition to showing the user that the port is either open or closed. The middle portion is for sending data, and the bottom portion is for receiving data.

- Testing to ensure that all the visual features work as intended.
  - Tests have humans interact with every button and assert that each action results in proper information stored in the serial port instance.
- Testing the GUI's ability to send and receive data.
  - Testing various strings in the user interface along with the send button ensures that the values inputted from the user are passed into the SwiftIO board so that the user input can be loaded into the VISA driver.

Integration testing

As far as integrating the developed software with the SwiftIO hardware, there are two options that the client is able to use: wired connection through USB to UART module or wireless bluetooth connection. Both require separate modules to allow for the connection, and both require their own, but similar tests, to ensure a stable and reliable connection needed to pass through commands accurately.

- Testing initial downloading of VISA driver program to SwiftIO board.

    - Ensures the SwiftIO board's program is correctly downloaded to the board.
- Testing USB or Bluetooth serial connection.
    - Ensures that the serial connection is established without any impediments using the USB to UART module or the bluetooth to UART.

User (Client) Acceptance Testing

The final project was tested in a manner that made sure that Dr. Hildreth was able to fluidly utilize the application with all concepts and modules that he currently uses.

- Testing with Dr. Hildreth's research instruments using VISA commands, sending and receiving expected data to and from SwiftIO Board.
    - This testing makes sure that the definition of done has been achieved.
    - Testing different edge cases
- Application and SwiftIO board interaction testing with Dr. Hildreth's classroom hardware implementation.
    - This test makes sure that different instruments send and receive commands expectedly, as well as switching between hardware configurations.

Code Metrics

Consistent and quality coding principles must be followed to allow for easy expansion by future developers. To explain what the programming part of the project does, consistent commenting of methods and code blocks is necessary to effectively convey its function. Further along that line, coherent naming schemes for variables, methods, and class names (camelCase, PascalCase) are also necessary to keep the clarity of the code.

For the structure of the code, software engineering principles such as S.O.L.I.D. are used to create modular code that won't require much, if any, changing once the code is taken over. Design patterns also help with this, like with using helper methods to ease the process of communication between the board without repeating code.

Code Reviews

To ensure code quality, the project's code is reviewed and tested at the end of each working week. These tests include testing the functionality of the code as well as tests for readability and presentation of the code. For this reason, both the front end and back end of development are accounted for in the quality assurance of the project regularly.

Results

Testing Results

The overall goal of the SwiftIO board project is to send VISA compliant commands through a MacOS application to a SwiftIO board, and have the board translate and execute the commands properly while also sending a response back to the application. The MacOS application's graphic user interface (GUI) has been tested to ensure that each user selection in the interface sets the expected serial connection parameters. To add, sending data and receiving data to and from the MacOS application over a serial connection has been tested and proven to work with bluetooth and USB to UART modules. The data was captured dynamically and displayed with the help of an LCD screen connected to the SwiftIO board. Further, this application has been used and performed as expected on M1, M1 Rosetta, Intel system architectures.

There was not enough time during this five week project to implement every single VISA command, as they are very extensive, but every command implemented is working properly. Every command implemented has the associated error codes that have been tested extensively. Due to the client's schedule, testing on laboratory instruments has not yet been completed.

Performance Results

Performance, measured by the time required to change functionality on the SwiftIO board, has drastically increased, as the additions to the system have allowed for real time, dynamic communication and functionality of the board. The previous implementation was to manually edit the SwiftIO program and redownload the program onto the SwiftIO board, or adjust other physical inputs such as dials or buttons. With this previous system, making changes to the functionality of the SwiftIO board took at least twenty seconds to complete and could range . On the other hand, the additions to the system are able to send and receive data to and from the SwiftIO board without having to edit and redownload the SwiftIO board's program. This allows for nearly instantaneous communication and change in functionality, therefore the system performance is only restricted by the time it takes to type in a VISA compliant command.

Future Work

The SwiftIO VISA driver project is part of a bigger picture in Dr. Hildreth's lab. The future work that could stem from the project includes the creation of networks of SwiftIO boards connected to a single application interface to facilitate communication of different instruments utilizing VISA commands. This central, more powerful application could be expanded to work on iOS, or other Apple operating systems. Also, the VISA instruction set compatible with the SwiftIO board could be expanded. The client wants to eventually automate the operations of certain instruments that rely on VISA communication from other instruments. Later on, broadcasting from the client's computer to a network of SwiftIO boards to improve time efficiency may be possible, however, as there is no documentation or integration of this for the hardware, so this is not currently feasible.

As the project is open for future developers to improve upon as per the client's design, they may find the IDE and hardware is easier to use with future progress put in from the hardware's manufacturers. Because of this, this project's framework may become outdated over time. Although this project works well and efficiently, improvements made to the software and hardware used in the SwiftIO board allow for more innovations and quality of life improvements for the users of this project.

Though many options and setups were tested for the completion of this project, many proved to be ineffective or unsupported in the current state of the hardware. For the sake of future developers, documentation is provided in order to clarify what angles and ideas they need to avoid when adding onto this project later down the line. And although some options may not have worked now, future hardware and software developments may allow for them to work when implemented, so future developers may need to take note of that and understand the documentation not only in the project, but what the project is based on, as well.

Appendix

The code developed during this project is delivered in a GitHub repository located at:
https://github.com/HildrethResearchGroup/SwiftIO-VISA-Driver

VISA Command Documentation

- SOURce()
  - Used for specifying the source pin. The number of the pin in question is put in the parentheses. This command can also be shortened to “SOUR()”.
  - For instance: “SOURce(12)” would set the current source pin to digital pin 12.

- DISPlay()
  - Used to display text within the parentheses directly to the screen connected to the SwiftIO board. This command can also be shortened to “DISP()”.
  - For instance: “DISP(Hello World)” would display “Hello World” to the screen.

- VOLTage arg1
  - Used to apply a certain voltage or put a pin into a digital mode. Arg 1 is the voltage or mode in question. This command can also be shortened to “VOLT”.
  - For instance: “VOLT 1.22” set the voltage of the current source pin to 1.22 volts.

- PWM arg1
  - Used to apply a certain duty cycle to a pin capable of pulse width modulation (PWM). Arg 1 is the duty cycle question, taken in as a floating point number between 0 and 1. This command cannot be shortened
  - For instance: “PWM 0.50” will set the duty cycle of the source pin to 50%.

- ? (Query)
  - Any command or variable can be queried to display it’s current value simply by adding a question mark to the end of the command.
  - For instance: “SOUR?” would return the current source pin.

- : (Command separator)
  - To separating commands made in the same call
  - For instance: “SOURce(12):DISP(Hello World)” would execute both commands as previously mentioned in the order specified.
  - SOURce can also be uniquely paired with commands as follows to apply a command to a specified pin “[SOURce(12)] VOLTage …”.