CSCI 370: Final Report

Team CSM Hildreth

Connor Barnes, Luke Henke, Ivan Krukov, Rena Loving

June 18, 2019

"The trouble with programmers is that you can never tell what a programmer is doing until it's too late"
—Seymour Cray

1 Introduction

The National Instruments Virtual Instrument Software Architecture (NI-VISA) is an API written in C used to interact with serial devices in many research institutes and laboratories. VISA itself is actually a standard for configuring, programming, and troubleshooting instrumentation systems. This includes GPIB, VXI, PXI, Serial, Ethernet, and USB interfaces. It provides an interface between the hardware devices and the software so that users can control their devices through software. NI-VISA is an actual software library in C with interactive functions for actions such as reading and writing.

To ease development of device reading/control, several major solutions exist, including BenchVue, LabVIEW, and PyVISA. BenchVue utilizes a drag-and-drop programming language similar to Scratch. LabVIEW is mainly employed by Electrical Engineers as a means of programming by creating circuits. PyVISA is a Python wrapper for the native C library which enables users to call the underlying C functions through Python. This is useful as it simplifies the programming aspects and provides a user friendly way to utilize the NI-VISA library.

However, this project exists because the client, Dr. Owen Hildreth, wants something better than PyVISA. Python does not easily support multithreading, is not type-safe, is slower, and does not work with the native MacOS APIs. Dr. Hildreth is an Assistant Professor in the Mechanical Engineering department here at the Colorado School of Mines. His research includes the use of electronic instruments (such as waveform generators and digital multimeters) to control a nanoinkjet printer, and he wants to be able to do this programmatically. Dr. Hildreth and his research group do their work on Apple computers, so it is important to have compatible programs. Thus, he requested for a Swift version of PyVISA – SwiftVISA. Swift is native to Apple, supports multithreading, and works faster.

The goal of this project is to develop the beginnings of a Swift wrapper for the VISA library to give researchers the ability to programmatically control lab devices without having to learn a more difficult programming language like C or have to do all of their programming in Python, which, as in Dr. Hildreth's case, may not be the ideal circumstance. While this is rather open ended, Dr. Hildreth wanted the wrapper complete enough for his use (which only contains instruments controlled using SCPI commands) by the end of the summer.

2 Requirements

2.1 Functional Requirements

1. Implement common VISA variables, constants, and message codes as native Swift types.
2. Provide functionality to programmatically connect to devices.
3. Read values from connected sensors in Swift and properly format values for the user.
4. Write values to connected sensors/devices in Swift.
5. Replicate the more advanced functionality provided by PyVISA such as reading values over an n second interval.
6. Implement a test suite to go with the SwiftVISA code.
7. Provide robust user documentation for usage with code examples.
8. Integrate a multithreading paradigm into the design to allow for concurrent operations.

2.2 Non-functional Requirements

1. The wrapper will support/be written in Swift 5 or higher.
2. The wrapper will be kept and maintained on a public GitHub Repository.
3. The wrapper will be licensed under the MIT license.
4. Code will be tested using an Agilent Multimeter and an Agilent Waveform Generator.
5. The documentation will be hosted under Github's Wiki service.

3 System Architecture

3.1 Instrument Hierarchy

Figure 1 shows the hierarchy of the instrument protocols and classes. “Instrument”, “Message Based Instrument”, and “Register Based Instrument” are the overarching protocols that have most of the functionality needed for any given instrument. A messaged based instrument communicates using string commands, while the register based instruments communicate with memory commands. The other thirteen boxes are classes, with specific functions for that instrument. When a user connects a new device, it is created as an instrument and the user can cast it to either message based or register based, depending what kind of instrument they have connected. If they want to use some of the specific functions for an instrument, they can cast it to that (such as USB Instrument). For the majority of use cases, casting to the lower-level instruments will not be required, as the vast majority of implementation is shared in the parent protocols.

Our client provided us with two Message Based Instruments to test with, so we have fully implemented and tested that branch. Unfortunately, we do not have any of the other types of instruments, so while we began the development for those classes, they are not fully implemented. Since this is an open-sourced project, somebody else who does have those instruments will be able to extend our project to fully incorporate those other instruments.

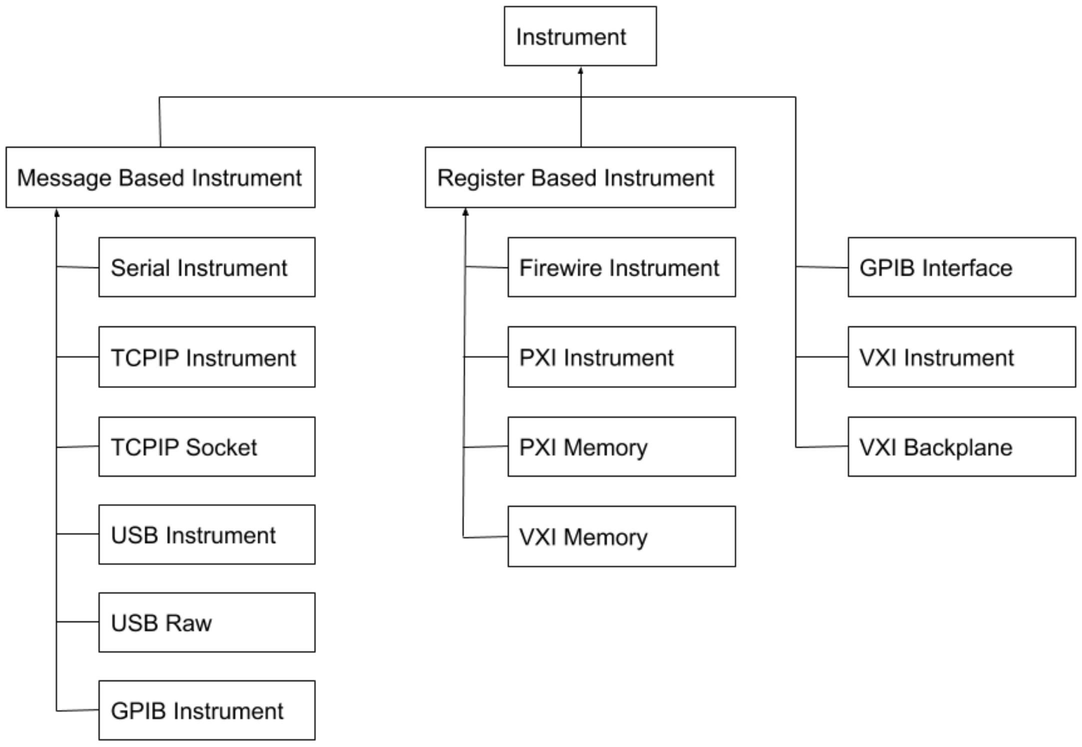


Figure 1: Instrument Hierarchy

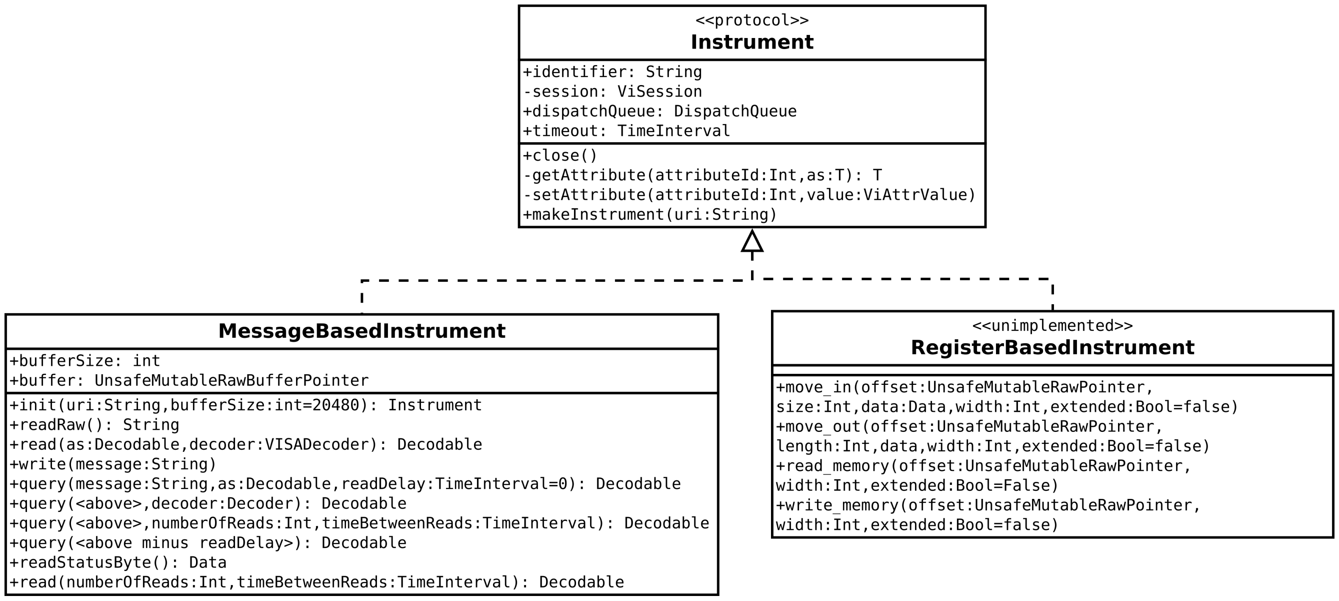
3.2 Instrument UML

Figure 2 shows the UML diagram representing the implementation details of the high level overarching protocols. The Instrument protocol is responsible for housing session related information, attribute operations and attributes common to all instruments like manufacturer model, and a dispatch queue responsible for multithreading scheduling. One thing to note about our UML diagram is that Decoder and Decodable are special types that are described further in the Technical Design section. In essence, methods that use Decodable and Decoder leverage Swift Protocol features that allow for powerful abstractions and templating of variables in functions. This allows SwiftVISA to have more type safety checks than PyVISA while at the same time reducing code repetition for each cast (i.e. `queryString()`, `queryInt()`, etc.) and allowing people who wish to extend the functionality of SwiftVISA's casting system to custom defined types. All functions are also designed to throw errors on failed operations based on the status code the C library returns. In order to ease convenience in this avenue, we also implemented a mapping from error code to error message so that when an error is thrown, a descriptive message is given to the end user as to what happened.

An additional abstraction made by our design is that when a user wishes to get something like a voltage reading from a device, the user would have to first initiate a write command indicating that they wish to read the voltage and then perform a read. Our architecture allows for users to instead use a query method that performs the write and read for them. There are also additional overloads on query that controls everything from the timing of operations sent to the device to even query multiple values.

Not shown in the UML is the fact that each instrument class has various getters and setters for each attribute that corresponds to that device in the VISA architecture. These functions are meant to abstract away the need for a user to page through the VISA documentation to find an attribute name (For example, `VI_ATTR_MANF_NAME` to get the manufacturer name) and call `setAttribute()/getAttribute()` themselves from the instrument class. This also provides a nice mechanism to have in-IDE documentation about what an attribute represents. As there are 150 attributes and each has a getter and setter, the UML excludes those attributes from the attributes list.

One major thing to note in our design is that we had originally intended to implement attributes using a Swift feature called computed properties. This allows us to combine getters and setters in a way that makes the attributes look like regular variables rather than getter or setter methods to a person using SwiftVISA. However, Swift doesn't currently support throwing errors from computed attributes, which resulted in us having to write the methods for each getter and setter manually.


```
<<protocol>>
Instrument
+identifier: String
-session: ViSession
+dispatchQueue: DispatchQueue
+timeout: TimeInterval
+close()
-getAttribute(attributeId:Int,as:T): T
-setAttribute(attributeId:Int,value:ViAttrValue)
+makeInstrument(uri:String)
```

```
MessageBasedInstrument
+bufferSize: int
+buffer: UnsafeMutableRawBufferPointer
+init(uri:String,bufferSize:int=20480): Instrument
+readRaw(): String
+read(as:Decodable,decoder:VISADecoder): Decodable
+write(message:String)
+query(message:String,as:Decodable,readDelay:TimeInterval=0): Decodable
+query(<above>,decoder:Decoder): Decodable
+query(<above>,numberOfReads:Int,timeBetweenReads:TimeInterval): Decodable
+query(<above minus readDelay>): Decodable
+readStatusByte(): Data
+read(numberOfReads:Int,timeBetweenReads:TimeInterval): Decodable
```

```
<<unimplemented>>
RegisterBasedInstrument
+move_in(offset:UnsafeMutableRawPointer,
size:Int,data:Data,width:Int,extended:Bool=false)
+move_out(offset:UnsafeMutableRawPointer,
length:Int,data,width:Int,extended:Bool=false)
+read_memory(offset:UnsafeMutableRawPointer,
width:Int,extended:Bool=False)
+write_memory(offset:UnsafeMutableRawPointer,
width:Int,extended:Bool=false)
```

Figure 2: UML Diagram for higher level protocols

4 Technical Design

4.1 Data Decoders

The UML diagram shown in Figure 3 demonstrates the design of the Data Decoder using Swift protocols. By using protocols, users of our framework can easily add conformance for VISADecodable to any type they may need. Further, by providing Decodable as a protocol, the user may implement multiple decoders for a single type for cases where instruments return different formats for the same type. Lastly, as a convenience, rather than requiring the user to pass in a decoder with each call to reading from an instrument, we added a requirement for VISADecodable, where it must provide a default decoder. When calling to the read function, the decoder may be omitted if the user wishes to use the default decoder.

By using generics in association with these protocols, we were able to make a single method for reading from an instrument. This function takes two parameters, the type that the message should be decoded to, and the decoder (optional) that should be used for decoding. Out of the box, our framework adds conformance to VISADecodable for five types, and provides a single (default) decoder for each. These types are String, Int, Double, Bool, and Data.

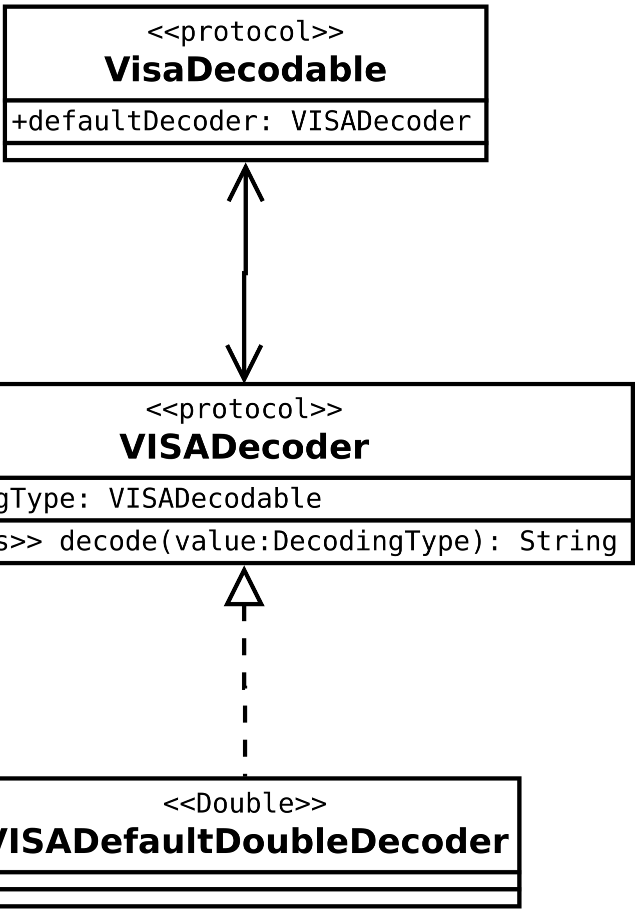

<<protocol>>
VisaDecodable
+defaultDecoder: VISADecoder
<<protocol>>
VISADecoder
+decodingType: VISADecodable
+<<throws>> decode(value:DecodingType): String
<<Double>>
VISADefaultDoubleDecoder

Figure 3: Data Decoder Design

4.2 Multi-threading

One of the important aspects of our project is that fact that it supports multi-threading. Without multi-threading, the entire program would be running on a single thread. This results in the GUI freezing because it has to wait for input/output communication each time, which could take only a couple milliseconds, or several seconds. This is not ideal because it slows down the use of the code and the GUI. If there is only one thread, then all of the instruments are on the same thread, and the program has to alternate reading and writing for each instrument, as depicted in Figure 4.

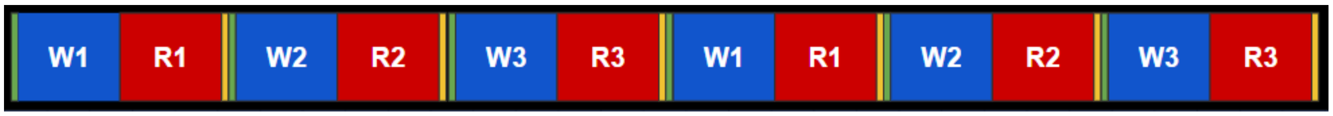


Figure 4: Single Thread Diagram

Our design has integrated multi-threading on a per-instrument basis as indicated by the dispatch queue owned by each instrument to schedule operations. By giving each instrument its own dispatch queue, they can all communicate at the same time. Each instrument can be on its own thread, as depicted in Figure 5. This allows the system to simultaneously read and write to multiple instruments. We also chose to make multi-threading opt-in so that it is an option to write code for a single thread for beginners. This gives advanced users more control over how the multi-threading works, but allows less advanced users to still use the code without having to learn about multi-threading.

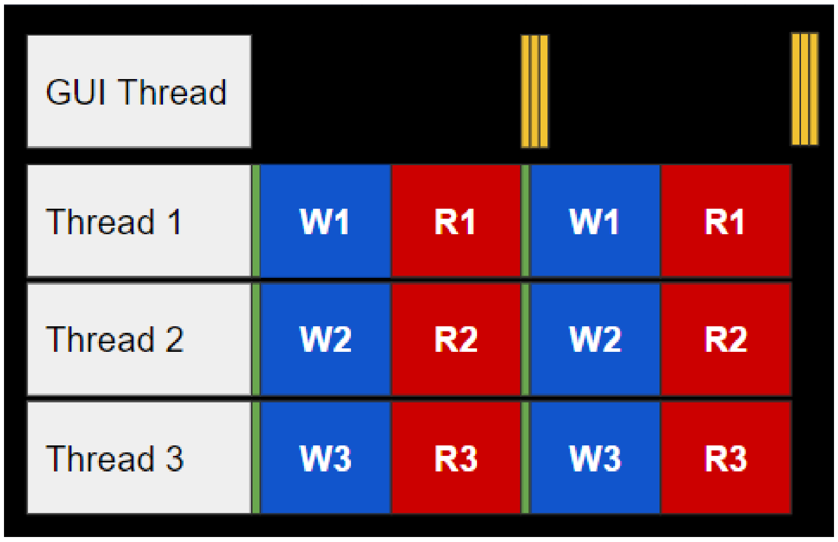


Figure 5: Multi-Threading Diagram

5 Quality Assurance

5.1 Unit Testing

We have quite a few unit tests in order to ensure that our functions are working properly. The goal is for this to have 100% coverage, but this is unfortunately not possibly for our project. Dr. Hildreth has provided us with a digital multimeter and a waveform generator, both of which are message based instruments (they communicate via strings). Through these, we can test functionality for our `MessageBasedInstrument` protocol and some of the more specific functions, but we are unable to test `RegisterBasedInstrument` since we do have a device of that type in our possession. However, since this project is open sourced on GitHub, other people who do have access to those specific instruments will be able to go in and test the functionality.

We considered having mock interfaces in order to test the classes and protocols that we do not have devices for, but we ultimately ruled this out. With the short time limit of 6 weeks (5 if you account for the fact that the last week is merely presentation and our code has to be complete before this), we were concerned that it would take more time to learn how to set up the mock interfaces than to fully implement the functionality that the client will actually use. As such, we have hunkered down to ensure that MessageBasedInstrument is thorough as opposed to guessing what to do for ResgisterBasedInstrument and the other, more specific, instruments.

5.2 User Acceptance Testing

Obviously user acceptance testing is important, because if Dr. Hildreth cannot use our code, then it is just about worthless. With this in mind, we have had him come and see our progression several times a week in order to keep him in the loop about the developments. This has already provided invaluable input to our team, as the first time we showed something to Dr. Hildreth, he told us we were going in the wrong direction and that it was not what he wanted. We were able to change courses early, as opposed to getting all the way to the end and having the wrong product. We have more of these meetings set so that Dr. Hildreth can continue to have input with the project as it comes to a close. Ultimately, we intend to sit down and help him use the framework so that he understands how everything works.

5.3 User Documentation

Documentation is very important for our project as it will not be completely and utterly finished and good to go out into the world at the end of field session. Dr. Hildreth has expressed interest in having another field session team next summer to further develop this project. Between this and the fact that it is an open-sourced project on GitHub, having documentation is necessary. Without it, anybody picking up where we left off will spend more time trying to figure out what we did than making any meaningful contribution.

Within our code, we are using an equivalent to JavaDocs, which can be exported into a page and have specific descriptions for each function. We are also creating a wiki page using Markdown on GitHub alongside our project to provide documentation similar to what PyVISA has. This will provide guides and usage examples. We also intend to have some higher level tutorials and explanations in order to fully convey what we have created and are passing off to Dr. Hildreth.

6 Results

6.1 Unimplemented Features & Future Work

6.1.1 Register Based Instruments

Currently SwiftVISA only works with Message Based instruments, which communicate using SCPI strings. Other types of instruments do exist (most notably Register Based instruments, which work by moving blocks of memory.) These should be incorporated into the framework. As of now, the framework has skeleton classes for these instruments, but their initializers produce fatal errors. Further, the protocol RegisterBasedInstrument is defined, but does not have any of the needed requirements. The first SwiftVISA team was unable to implement these instruments because they did not have access to any register based instruments. A future team should implement these instruments if they have access to them.

6.1.2 SwiftVISA Backend

SwiftVISA uses the NI-VISA C binary-compiled framework for the time being. This comes with several drawbacks:

1. Because the NI-VISA C binary framework was compiled for Intel processors, it is not possible to run the framework on iOS and iPadOS because they use ARM processors.
2. The library is unable to use the Swift Package Manager (SPM), because SPM does not support the use of binary compiled frameworks. Instead we are forced to create a Cocoa Framework. In order to be able to use SPM, we would need the source code of NI-VISA which costs $500 and is $150 per distribution (which would be impossible to track due to our project being open source).
3. In order to work with events in NI-VISA, you must make use of callbacks. Callbacks are imported into Swift as @convention(c) closures, which do not support capturing values, making them a pain to use because you cannot access 'self'. By writing the backend in Swift, we would not have to worry about making closures be @convention(c).

A future team should write an open-source replacement for the NI-VISA C framework in Swift. Doing so would allow the whole project to use SPM and be fully platform-independent. This would mean that SwiftVISA could run on any system that supports Swift such as Linux. It would even allow for SwiftVISA to run on devices that NI-VISA does not support such as iOS.

6.1.3 Full NI-VISA Implementation

SwiftVISA does not yet support the full range of features that NI-VISA's C framework supports. A future team should finish implementing all of the functionality. Notably:

- Events are not implemented.
- Non message based instruments (including instruments that are not register based either) are not implemented.
- Many memory-based operations are not implemented.
- Only PyVISA instrument types are added, not the full range of NI-VISA supported instruments.
- Not all attributes are implemented.

6.1.4 Documentation

- Make a script that compiles the Jazzy docs and uploads them to the GitHub Pages repository.
- Move the docs away from the GitHub wiki (which is incredibly rudimentary) onto the GitHub pages (would require some web design).

6.2 Lessons Learned

- One of the important lessons we learned is that this whole project functions significantly easier on Macs than it does on Windows computers. Since Swift is native to MacOS, it is easier to program this on a Mac. However, 3/4 of our team do not have Mac computers, so we had to borrow from our client so that we could more efficiently develop and test our project.
- Having the whole group work together in one room was helpful for working out problems and bouncing ideas off of each other. It was also good for keeping everyone up to speed on everything else happening since we could do in-person SCRUM meetings.
- However, some things do not require having the whole group present, and someone might work better at 3AM and someone else at 8AM. Being able to split up the workload so that everyone is maximizing productivity really helps get the work done and not feel so dreary.
- Early design decisions can have a large impact in the usability and readability of code. How we broke up instruments and decodable types into protocols allowed us to leverage strong language features to achieve this.

7 Appendices

Since our project is open sourced on GitHub, we utilized the GitHub wiki pages and our own Jazzy documentation to provide explanations of product installations, development environment, coding conventions, acceptance tests, as well as examples of how to use our framework. We provided this alongside all of the code on GitHub so that it remains together rather than be separated into this document. This can all be found on our GitHub wiki.