Implementation / Testing Results

Implementation

The purpose of this section is to discuss how exactly our group had solved the problem given to us by our client. This section will bring up design decisions that we ended up using in the final version of the project, and reasoning for why we choose to use them.

Design - These are the core concepts that we used when designing the project to make things clean, extendable, and functional.

Structures - The LCR library includes many custom C-structures, many of which choose to use design patterns that only really make sense in C. Instead of trying to directly cloning these structures to classes on the Java side, we decided to improve them following Java paradigms instead of C paradigms. This means that our classes would instantiate status classes in our structures instead of returning bit fields or raw field numbers to the Java side.

Pointers - Pointers are obviously a very C-like concept that doesn't translate very well. Liquid Controls used pointers in several functions to return multiple objects from a call. In situations like this we had to build custom return type classes that included multiple members and returned those instead.

Enumerations - There are several places in the native LCR library where if we were to directly clone the signatures to the Java side, we would have provided a bunch of holes in our code where the programmer on the Java side could enter illegal values. This issue is incredibly clear in the portion of the project dealing with setting and getting fields. This process on the native side involves a C-like generics system for accessing variables stored on the device. Instead of providing an interface where the user could ask for a integer field then give the id of a string variable, we restrict the developers requests to fields that align with the return type of the method.

Exception Handling - *See Exception Handling in the Result section below.

Results - During our groups first meeting with the client we were given a run down of the overview of the project as well as a series of implementation constraints. Now that the implementation is complete we can discuss how our current codebase satisfies our clients requirements.

Exception Handling - Since the fuel delivery process is very volatile (it requires human interaction, an environment where things can stop working or unplug themselves, etc.) it was a requirement of this project to be able to perform in the

expected environment and react logically when unexpected events occur. Most of these statuses that the device tracks are returned with every native call. Instead of having all of our wrapper methods return this status as well, we instead throw an exception up to the Java level whenever something not normal happened. This allows our wrapper to be more intelligent by being reactive to problems and also prevents us from needing to create a custom return type class for every method call.

Printing - It was required that our project do all printing through the flow meter device using the LCR library instead of interfacing directly with the printer like our clients previous implementation. To meet this requirement we simply had to wrap the print methods available in the library. This involved properly handling unicode strings between the Java and C side.

Async Wrapper Access - Since the fuel truck setup can include a two flow meter configuration it was important that our wrapper could support being accessed asynchronously. To solve this problem we simply synchronize all access to the native wrapper from the Java side on a singleton class. This allows any number of meter implementations to hit the device at the same time without the fear of a deadlock.

Baud Rate - The previous implementation of this project ran at a baud rate of 9600. Our client hoped that we could have our implementation run at a higher speed and still maintain stability. We were able to run all of our tests at a baud rate of 19200 while maintaining stability.

64bit Runtime Environment - When our client first started this project 32bit systems were prominent. Since then the industry has started to shift toward 64bit systems. Because of this shift in computing environments our client required that our project operates in 64bit OS environments so that it will work on any new laptops he uses for his setups. Our solution to this problem is to compile our native wrapper in 32bit configuration and then require that the environment that is running the application (either 32bit or 64bit) to be running a 32bit JRE. We would have prefered having a native wrapper for both 32bit and 64bit, but the creators of the flow meter only provide a set of 32bit libraries, so we are stuck with a 32bit wrapper only.

Java 7 JDK - Java has come a long way since our client first started working on this project. He requested that our project not only compiles in the newest Java 7 JDK, but also takes advantage of the new classes and patterns. This wasn't a problem for our group seeing as we've all grown accustomed to Java 6 and 7 during our time as students (meaning we had no reason to go back to old design patterns for synchronization or anything).

Testing

The purpose of this section is to discuss the methods in which our team tested our codebase and the results of those tests.

Disconnected Device Test - Since we didn't have access to the flow meter device until the last 3 months of the project we needed other ways to verify that our framework communicates with the device library. These tests would use our wrapper to invoke all of the native methods we had access to, and verified that they failed successfully. Successful.

Connection Unit Tests - The first tests that we did to verify that we were capable of communicating with the client's hardware (and letting us know that our device configuration worked) was to open a connection to the flow meter(s), call a function to verify that the connection was successful, and then closing the connection. Successful.

Automated Deliveries - We created a series of unit tests to replicate the sequence of actions involved in fuel delivery. Since there are several methods that a driver typically deliveries fuel to a buyer, we did our best to create a separate test scenario to encompass the drivers experience for each.

> Preset Delivery - Delivery where the driver sets a preset volume to be pumped, and then lets the device deliver exactly that much. Successful.
>
> Manual Stop Delivery - Delivery where the driver watches the volume either through the application or the meter's display and manually switches the flow to stop. Successful.
>
> No Flow Stop Delivery - Delivery where the driver starts the flow of product, then pauses it after some period of time with the "No Flow" setting enabled. With the "No Flow" setting enabled, the delivery will end after a set interval of time. Successful.

Test GUI Tests - A GUI meant to represent the clients existing software was given to us at the beginning of this project. This GUI is capable of interfacing with the device in the same exact way that our clients software does. We used this GUI to connect to the device and test a series of features by hand. This included testing both the preset and manual stop deliveries pretending that we were the fuel truck drivers. These tests were probably the most important because the things that we were capable of doing in the test GUI are exactly what the fuel truck drivers will be doing. Successful.

Set/Get Field Tests - Most of our implementation, as well as most extended use of the flow meters capability, is based on the concept of getting and setting of fields stored on

the device. Since each field has an associated primitive type we designed a unit test for each of the primitive type to verify that we are capable of both getting and setting fields of that type on the device. These tests were verified in both an automated fashion as well as using the devices monitor to check things were successful. Successful.

Operation Manual

Setup System

This section will discuss the hardware configuration of our system.

Demo System

This section will discuss the setup process of the demo. We decided that the best way of preparing a demo was to record footage of a delivery happening on a computer that had everything successful setup (both hardware and software). Since the hardware configuration required for our test environment costs our client several thousand dollars, it wasn't a very likely idea that we would be able to setup and demo the actual system inperson.

Demo of the preset delivery

- Driver selects to use a single device and starts a connection with a single device.
- Driver enters a preset amount of volume to be pumped.
- Driver starts the delivery causing fuel to start pumping.
- The UI checks on the amount pumped by the flow meter every fraction of a second.
- The preset amount of delivered fuel is met and the device prints a receipt.

Demo of the manual delivery

- Driver selects to use a single device and starts a connection with a single device.
- Driver starts the delivery causing fuel to start pumping.
- The UI checks on the amount pumped by the flow meter every fraction of a second.
- The driver decides that enough fuel has been pumped and presses the button to end the delivery.
- The device prints a receipt of the transaction.

GUI App - This is the application our client gave us for this project. You can see the driver using the application throughout the entire demo to talk to the device. This application is written in Java Swing and is reliant on a Java interface for talking to the device(s). We had taken his application that had a test implementation at the end of the interface (simply faked interactions, no device communication) and replaced it with our own implementation. The interactions you are seeing during the demo are actual interactions with the device through our implementation.

FRAPS - Because we were incapable of performing a live demo for our presentation we had to record a demo of this delivery in advanced. FRAPS is a screen recording software used by many software developers to record demonstrations. We simply install the application and instructed it to record the office desktop while we performed the demo. It would have been more ideal if the application would have only recorded the contents of the window instead of the entire desktop.

Test System

This section will discuss how we had gone and tested our final system. These topics discussed are broad because the details are already covered in the testing results part of the final document.

JUNIT - We used JUNIT as the testing bed for all of our unit tests (both automated and manual). It was easiest for us to manage as well as run a series of tests when they fit the JUNIT format. These unit tests could be designed, committed to Git, and run either as a one-off or a collection all from Eclipse IDE.

Test GUI - Our client gave us a test application that mimics that software that the fuel truck drivers have access to. We used this application to test our implementation in a fashion similar to how the truck drivers will. These tests were performed in the clients office where all the hardware was configured.