Software Design Document For Integration to Fuel Truck Flow Meter Register via Java Native Interface on Windows Platforms

Version 8 (12/13/13)

Group Number:	DEC13-07
Client:	Oakland Corporation
Advisor:	Gurpur Prabhu
Group Members:	Jason Kaiser, Project Manager
	Yaze Wang, Tester
	Bryce Kvindlog, Developer
	Carl Garrett, Architect

Table of Contents

Table of Contents ____ 2
1. Introduction
1.1 Purpose ____ 4
1.2 Scope ____ 4

2. Requirements
2.1 Functional ____ 4
2.2 Non-Functional ____ 4

3. Architecture Design
3.1 Setup Diagram / Device Decomposition ____ 5
3.2 UML / Class Diagram ____ 8
3.3 Implementation Issues and Challenges ____ 9

4. Graphical User Interface Design ____ 10

5. Verification and Validation Plan
5.1 Prototyping ____ 11
5.2 Integration Test / Field Testing ____ 12
5.3 Mock API ____ 12
5.4 Black Box Testing ____ 12
5.5 White Box Testing ____ 12
5.6 Testing and Integration Results ____ 13

6. Standards ____ 16

A. Appendix
A.1 Operation Manual ____ 16
A.2 Project Plan ____ 18

Definitions	
Mock Object	A simulated software object which behaves as the real object in a controlled manner. This is typically used in testing software.
The Application	The implemented version of our project.
Flow Meter	The specific name of the flow meter is the LectroCount LCR-II. Its purpose is to measure and control the fuel being delivered.
API	An application programming interface is a protocol which is used as an interface by software components in order to communicate with each other.
JNI	Java Native Interface is a programming framework that enables Java code running in a Java Virtual Machine to call, and to be called by, native applications and libraries written in other languages.
Wrapper	A design pattern that translates one interface for a class into a desired compatible interface.
DLL	Dynamic link libraries are a collection of subroutines.

1. Introduction

1.1 Purpose

The purpose of this document is to describe the implementation of the Flow Meter Software described in the Project Plan. The Flow Meter Software is designed to create a user friendly computer graphical user interface for fuel truck driver to control and access the physical flow meter from the cab of the fuel truck. By providing a detailed coverage of the software application this document is intended to give a clear understanding of the project to the reader.

1.2 Scope

This document describes the implementation details of the Flow Meter Software. The software will consist of two major functions. First is to communicate with the physical flow meter in the truck and the printer for document printing. The second is to provide the flow meter user control and input through a computer graphical user interface. This document also overviews the hardware which the FlowMeter Software deals with and the functionality between all of the project's components.

2. Requirements

2.1 Functional

- The application shall be capable of communicating with and handling two flow meters.

- The application shall never communicate directly with the printer. The application shall send commands to the flow meter and the flow meter unit shall then print off the final receipt on the printer.

- The Flow meters shall connect to the application computer through a multiplexer which also connects to the printer.

- The application shall be designed to throw error messages in cases where failures occur.

2.2 Non-Functional

- For each flow meter, a single thread shall be used by the application to manage communication with that flow meter.

- Threads communicating with the flow meters shall not communicate at the same time.
- A singleton class shall manage the two threads communicating with each of the two flow meters to prevent the threads from interfering with one another.
- The application shall have both a 64 bit version and a 32 bit version.

3. Architecture

3.1 Setup Diagram / Device Decomposition

The system setup is very straightforward. The LectroCount LCR-II flow meter usually sits in the back of the truck. A connection cable will route from the LCR-II flow meter to the dual meter multiplexer. From the dual meter multiplexer the first connection cable runs into the truck cabin to the application computer. From the dual meter multiplexer the second connection cable runs to the printer. Through the connection cable, the LCR-II power cable is included and printer data cable as seen in Figure 1.

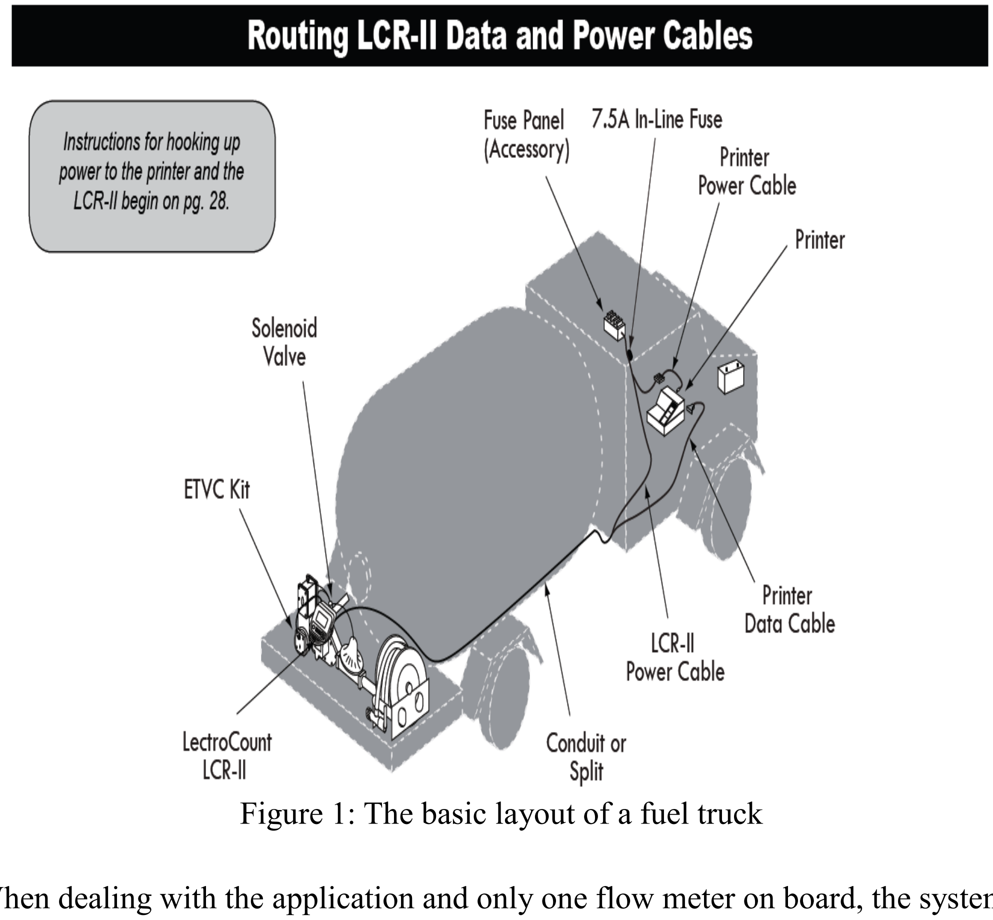

Figure 1: The basic layout of a fuel truck

When dealing with the application and only one flow meter on board, the system simply uses a Lap Pad Adapter to retrieve the data from the flow meter and share it with printer and the laptop computer. The connection will be similar to that shown in Figure 2; however, the Lap Pad will be replaced by a laptop in the implemented system.

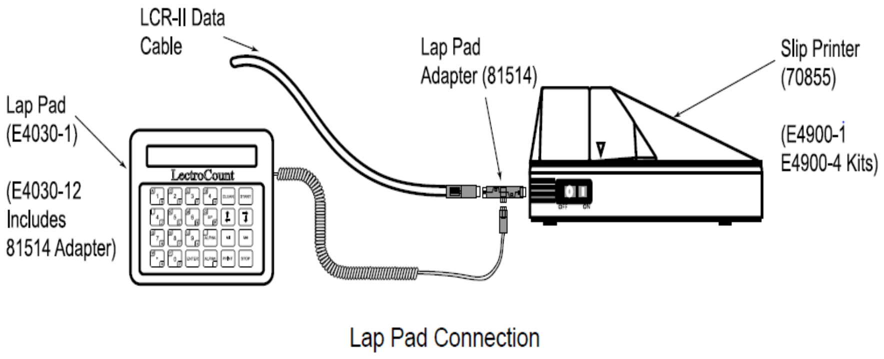

Figure 2: The connection between flow meter, on-board printer and computer device

When dealing with the application which includes two flow meters, the system requires the additional Dual Meter Multiplexer. As shown in Figure 3, this multiplexer is in charge of inter meter communication with both the printer and laptop. Since the multiplexer is not a weatherproof box, it must be installed in a location which it will not receive any damage from moisture. This location will be inside the cabin for our implementation of this project.

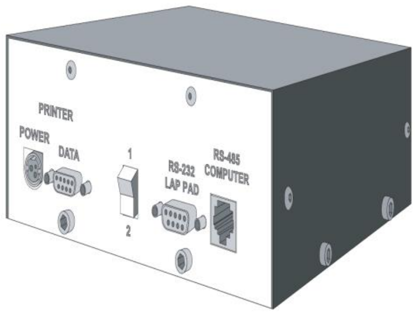

Figure 3: Multiplexer found onboard the fuel trucks.

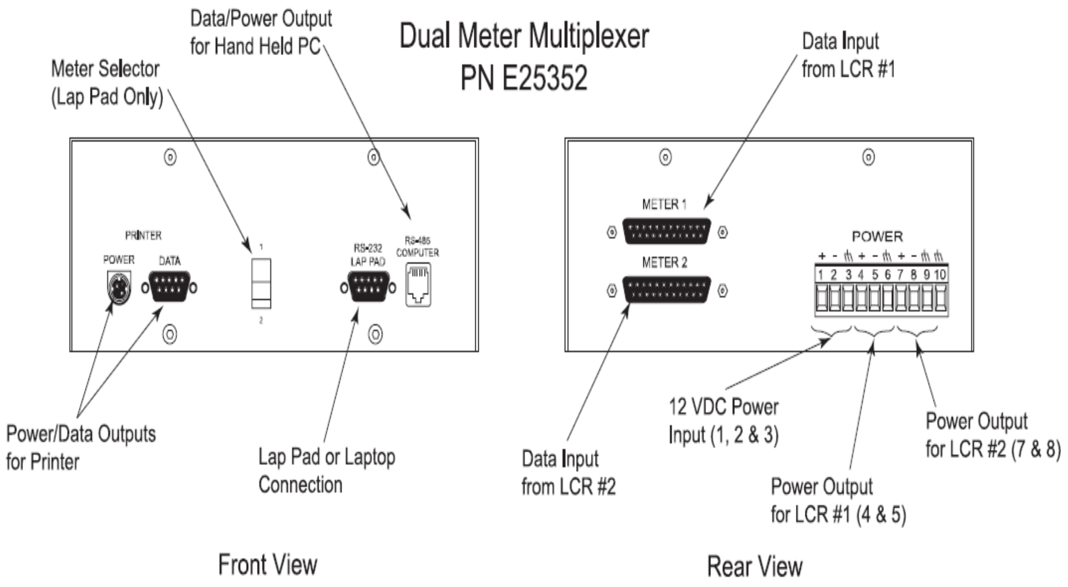
Dual Meter Multiplexer
PN E25352
Data/Power Output for Hand Held PC
Meter Selector (Lap Pad Only)
PRINTER
POWER
DATA
1
2
RS•232 LAP PAD
RS•485 COMPUTER
Power/Data Outputs for Printer
Lap Pad or Laptop Connection
Front View
Data Input from LCR #1
METER 1
METER 2
POWER
+ - ⏚ + - ⏚ + - ⏚ ⏚
1 2 3 4 5 6 7 8 9 10
Data Input from LCR #2
12 VDC Power Input (1, 2 & 3)
Power Output for LCR #1 (4 & 5)
Power Output for LCR #2 (7 & 8)
Rear View

Figure 4: Front and rear view of the multiplexer.

The setup connection with printer, as shown in Figure 5, will have the data power cable will be connected into the front side of the multiplexer. The opposite end of the power cable will be connected into the printer input.

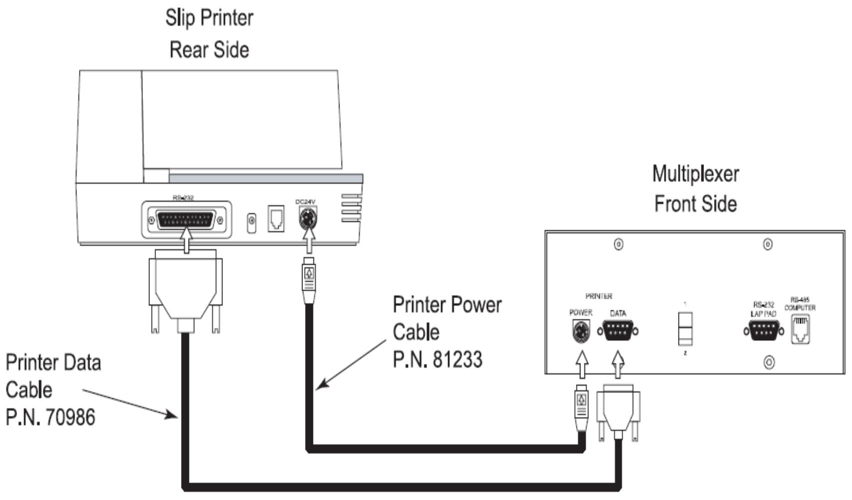

Figure 5: Diagram showing the connection between the printer and the multiplexer.

If using a laptop computer with HOST software to control or set up the registers, a 9 pin extension cable must be used, as shown in Figure 6. The MALE end of the cable will be connected into the front side of the multiplexer on the port labeled “RS-232 Lap Pad”. The other end of the cable will be connected into the RS-232 Port of the laptop computer.

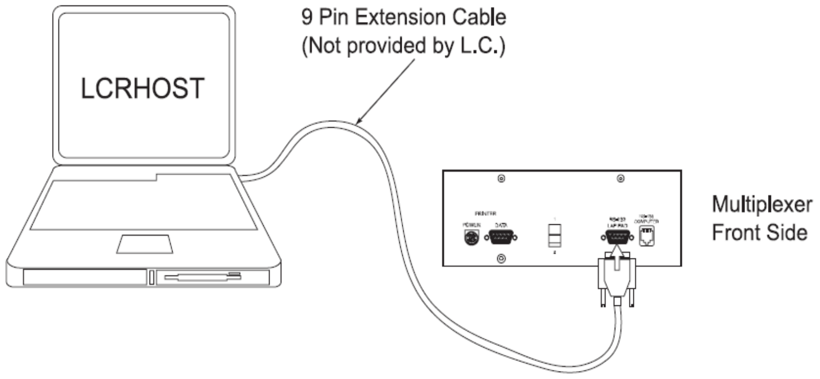

Figure 6: Diagram showing the connection between the multiplexer and the laptop.

3.2 UML/Class Diagram

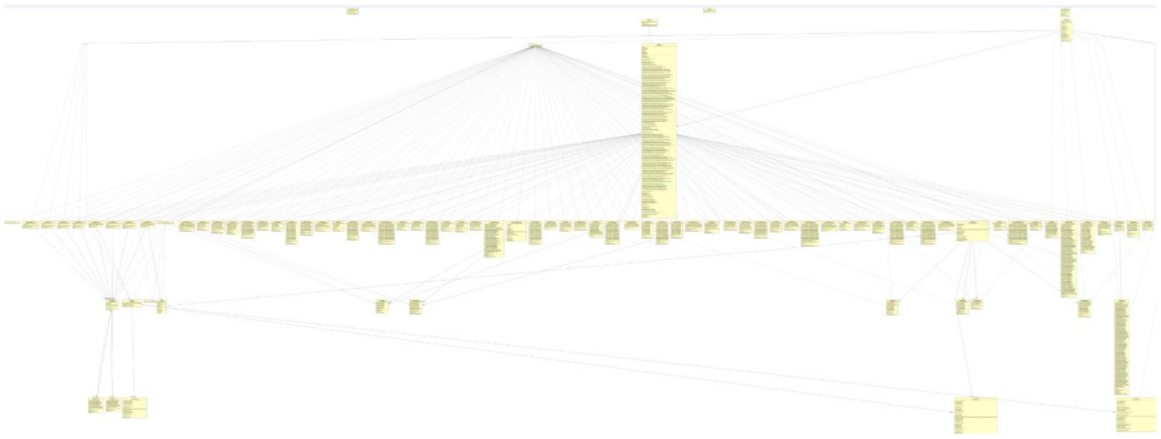

Figure 7: The Unified Modeling Language Class Diagram

Figure 7 shows how the application will use the library we are creating. It will enter through the FlowMeter Interface which is an interface designed to be a representation of the Flow Meter found on the truck. Eventually the interface is inherited and implemented by Flow meter classes that represent a refined or a Liquid Propane Flow Meter. These meters are able to communicate with the laptop by a C++ API from their maker Liquid Controls. This Liquid Controls API is able to interact with our program through native method calls from the native wrappers. The API uses numerous return variables and defined structures that we mirrored in Java so we could transfer and keep data. These wrappers use the base return type to set the return code of the function while the pointer returns from the API function is made into a java object and returned in a return object. These return objects all inherit the methods and data of the return class. Other functions require status variables and transmission variable to store data and transfer between C++ and java.

Note: Scalable version of this diagram can be found on the project's website.

3.3 Implementation Issues and Challenges

The main challenges for this project revolve around the need to use a C API library through Java. The required specification for this is called the Java Native Interface (JNI). JNI allows a Java program to call code written in C or C++ from Java methods. These C functions have access to operating system level functions that a Java program running on a virtual machine does not. This requires us to wrap a C API to use from a Java program.

The required API uses delegate functions (Function Prototypes) that are clearly defined and allowed in C but do not have an equivalent in Java. These functions must be defined as Listeners that can be called from a C function. This presents two problems as the object being called may not be in existence or the object being called may not be the object that the information was intended for.

The reason the functions are causing these problems is they are designed to be asynchronous while operating in a synchronous system. To solve this problem we have opted for the solution that requires listeners be called to carry on the workings of the programs. This may fail. The other solution that has been proposed is to use a system to wait on the asynchronous methods and merge them with the synchronous method while still in the C operating environment.

These problems are compounded by the API required to function with the external Flow Meter and will not work for tests without having a Flow Meter connected to the computer. This forces us to make a Mock API where we can control values returned from these functions for testing purposes. By using this API we can use black box testing to ensure that the wrapper functions properly before we try to do field tests. This helps because the project may fail if we can only find critical errors in field tests.

4. Graphical User Interface Design

Currently an older implementation of the application already exists and is being used by Oakland Corporation. Our group will not be making any changes to the GUI during our work on the application. Our purpose in modifying and updating the code is to bring the application up to standards and add additional functionality. The GUI window our portion of the application interfaces with is a part of a much larger application. Our focus is on a single section which can display a single flow meter connection or a dual flow meter connection. This section is a small part of a much larger application.

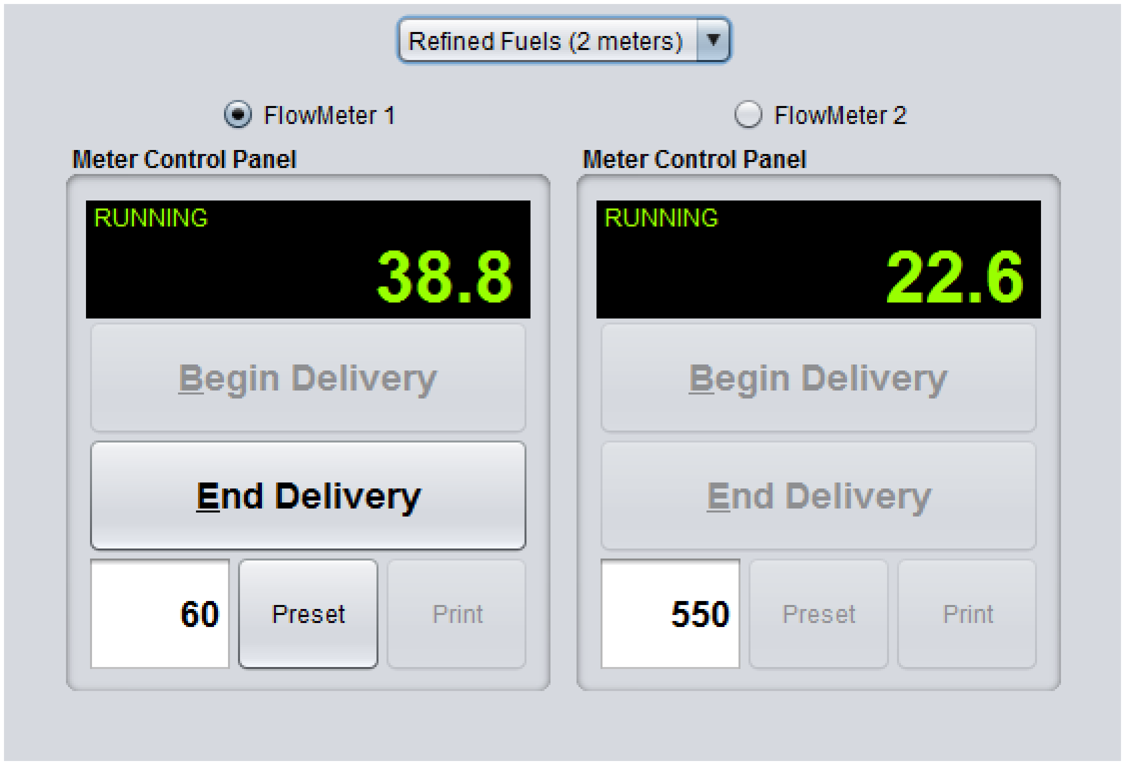

Figure 8: Screenshot of test GUI using the two fuel meter testing scenario.

During early development and prototyping a demo GUI application providing the same functionality as the GUI within the application itself will be used. The major difference and advantage is that this demo application does not require a flow meter to hooked up to it to function. The application is sent information in the same way that the flow meter would send it information and monitors the total fuel in the same way. This will allow simple and quick testing of the application. This demo application can be used to test a dual meter communication or a single meter communication. It can also provide a preset limit where the fuel will automatically stop when the number is reached just like the true interface. Eventually our team will have a fuel meter to work with as well as the opportunity to go on site. Images of the demo application can be seen above and below displaying both the single meter option and the dual meter option.

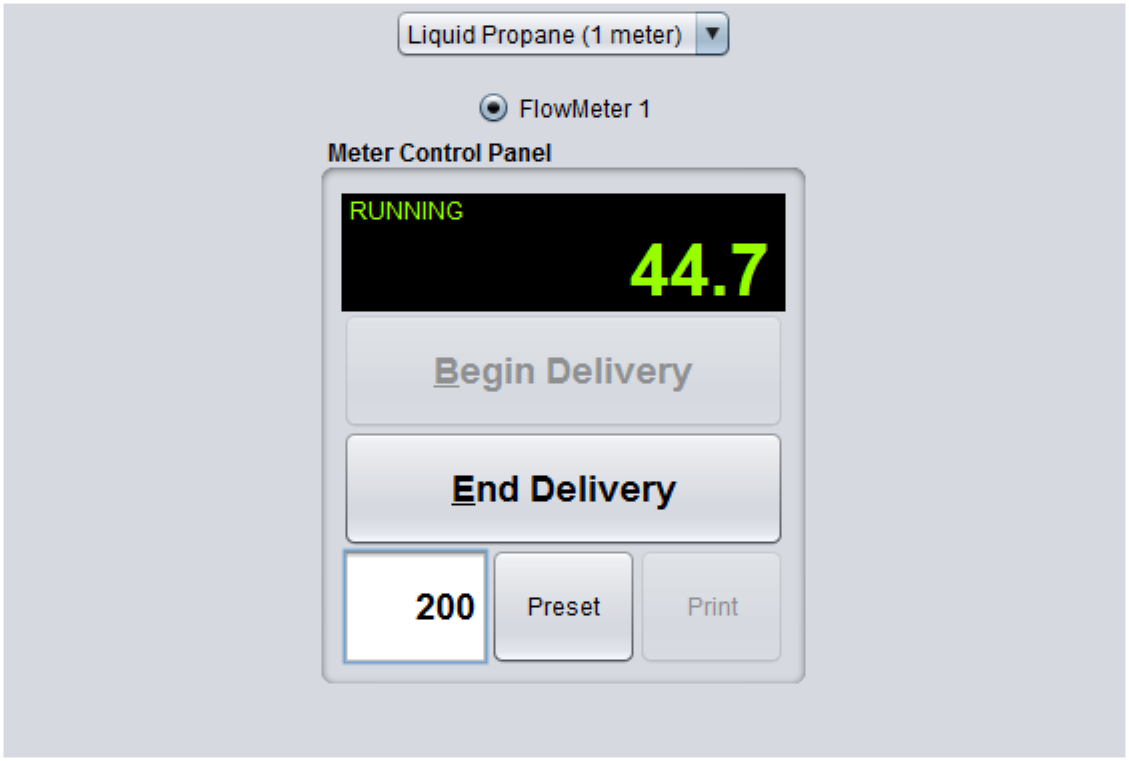

Figure 9: Screenshot of test GUI using the one fuel meter testing scenario.

5. Verification and Validation Plan

5.1 Prototyping

In an effort to guarantee that our development environment is setup correctly and to get an understanding of the process required for developing the JNI interface that will sit between a Java application and the Liquid Controls SDK we intend on developing a prototype. This prototype will be a partially implemented JNI wrapper that will include a collection of fully implemented operations to feel out what are going to be typical problems and how to avoid them.

5.2 Integration Testing / Field Testing

Since the beginning of our project our team has been aware that our client would provide our development team with an official flow meter device for integration testing, and field testing with one of Oakland client's fuel trucks. Since Oakland is renting a flow meter for our team to use, our team will only have 90 days to perform out integration testing. Similarly since field testing requires access to a fuel truck, we will only have a time or two to perform all of it.

5.3 Mock API

In a mitigation strategy to avoid the issue of not having access to integration or field tests for a majority of this project we decided to create a mock API object. A mock object is an object used to mimic the behavior of a real object in a controlled way. Since the Liquid Control libraries require interaction with a plugged in flow meter to access, a mock API would allow us to throw expected objects up through the wrapper and then test them for correctness in Java.

Our mock API would be implemented in C++ and would allow us to perform both our white box and black box testing in the absence of the required hardware. During the time we don't have access to the hardware we would simply have our wrapper point to our mock API DLL's (dynamic link libraries) instead of the Liquid Control's DLLs.

5.4 Black Box Testing

Black Box testing is the process of testing software codebase based on its specification and expected outcome of its interfaces, without having any knowledge how each function is actually implemented. Our project has a very clear cut partition where black box testing would be ideal and that would be our JNI wrapper. It's important that our wrapper is returning values according to the Liquid Controls SDK to assure that our FlowMeter implementation is fail-safe.

5.5 White Box Testing

White Box testing is the process of testing software codebase based on its implementation of its interfaces. It's during this process of testing that we will verify that our tests check for expected values among most branches of execution (code coverage). These types of tests can be performed within JUnit test cases.

5.6 Testing and Integration 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 those 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 maintains 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 preferred 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 client's 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.

6. Standards

Since our project is just a part of integration of existing client's software, all the standards we have to follow come from the client's software standards.

- Weight & Measure Handbook 44
  Inside this handbook, it states the federal law requires that the printers inside of the fuel delivery truck must be directly connected to flow meter device itself, so that no computer software can modify the fuel delivering volume and price.
- Printing Receipt Standards
  On the receipts which are printed by the onboard printer of the fuel truck, there must have the Start Time and Finish Time of the fuel delivery, the volume of fuel in gallon has be delivered, the client's name and the atmosphere temperature at the moment of delivering.

A. Appendix

A.1 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 in person.

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 client's office where all the hardware was configured.

A.2 Project Plan

Software Project Plan

For

Integration to Fuel Truck Flow Meter Register via Java Native Interface on Windows Platforms

Version 3.1 (12/13/13)

Group Number:	DEC13-07
Client:	Oakland Corporation
Advisor:	Gurpur Prabhu
Group Members:	Jason Kaiser
	Yaze Wang
	Bryce Kvindlog
	Carl Garrett

1. Project Overview

1.1 Problem Statement

Oakland Corporation is an agriculture business firm developing both accounting software and IT solutions. One of the major integrations that Oakland provides is a suite of tools for distributing agriculture based fuel. In addition to providing the hardware for this process Oakland has developed their own standalone accounting platform called Fuel Truck Point of Sale (FuelPOS). FuelPOS runs on a window's laptop from the cabin of the fuel truck where it provides recording functionality (tracking the amount of fuel that a client withdraws), up-to-date contract & pricing information, and existing customer account information. The FuelPOS system has become outdated since its creation over ten years ago. Oakland would like to see the application updated to current programming standards as well as US Weights and Measures standards.

It’s our team’s job to develop an improved implementation for the FlowMeter application that will handle modern software (Win 8/7/Vista 32-bit and 64-bit platforms) as well as environmental risks that work laptops are forced to endure in the cabin of fuel trucks. In addition to obvious improvements in code quality and performance that the application will receive, the project will be brought up to US Weights and Measures standards when printing fueling receipts. At the time of the creation of the current software regulations allowed the software to directly communicate with the printer in printing the receipt. New standards require that the printer only communicate with the meter with no communication from the laptop.

1.2 Operating Environment

The program running environment will be standard full-sized laptop running any of the Windows 8/7/Vista 32-bit and 64-bit platforms. Oakland Corporation provides support for the client's computers and installation of their systems.

When the fuel truck driver pulls into the fueling location, he or she will need to connect the fuel line from the truck to the fuel reservoir. Then, the driver will go to the cabin either set a preset fuel quantity to be delivered or simply begin the delivery to be stopped manually. Once the delivery is finished meter will detect the finished state and begin the receipt printing process. The printer, solely connected to the meter will the print off the receipt containing all required information to be then taken by the driver and turned over to the customer. This receipt must contain all legally required data pertaining to the delivery.

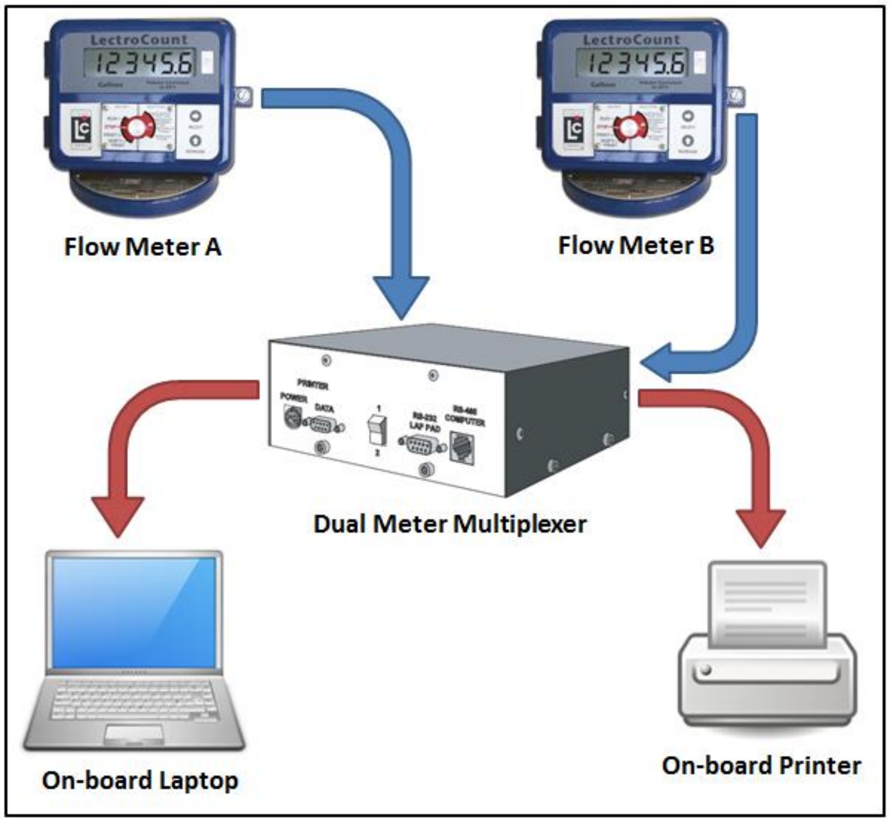

Figure 1.A Connection set up for the all relevant hardware equipment

As shown in Figure 1.A, the flow meter(s) will be directly connected to the multiplexer. Meanwhile in the driver cabin, the on-board laptop and on-board printer are also connected with the multiplexer. If the delivery truck does not have two meters then the multiplexer is not present and the laptop and printer connect into the same data cable leading to the meter.

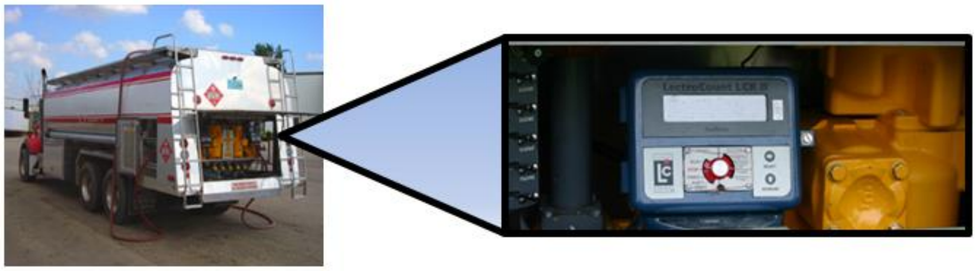

Figure 1.B The location of flow meter in a fuel truck

Figure 1.B shows the location of the flow meter in a standard fuel truck. As shown due to the location of the flow meter, it would be necessary for the driver to have the meter reading while he or she is in the driver cabin.

See Design Document for system diagram

2. Requirements / Specification

2.1 Functional Requirements

- The system shall provide error handling for all known situations which could result in a fault.

- The application shall handle the synchronized communication between both the diesel and propane meters on a refined fuel truck.

- The application shall print the ticket through the flow meter. The only connection to the printer shall be to the flow meter which will print the ticket when commands are received from methods within the application.

- The updated application shall be capable of communicating with the currently implemented GUI of Oakland.

2.2 Non-Functional Requirements

- The application shall support a connection to the flow meter with a connection speed higher than 9600 baud rate.

- The application shall compile at the JRE7-level. The application shall be brought up to date from its current level of 1.4.2 with the current library.

- The application shall be developed with a 64bit version and a 32bit version.

2.3 Constraints

- Software Runtime Environment

  The client's requirement is to use the latest Java version, being 1.7, to implement the code and that decides this project's development environment. Also, the final deliverable must be compiled as a JAR file to be used with existing client side application.

  Furthermore, the API of the Liquid Controls Flow Meter is written in C language and our client's application is in Java. In order to cope with that characteristic, the client gives us a FlowMeter Interface as guidance. The completed project needs to be able fulfill all the functionalities mentioned in the FlowMeter Interface.

- System Operating Environment

  Most of the user of FuelPOS will be using laptop computers with either Windows Vista, Windows 7, or Windows 8 operating system. So, the project has to be able to run smoothly under those systems.

  Due to the nature of fuel trucks operation, vibration of equipment potentially causing damage or disconnection will often happen between different devices. Having a robust error handling and emergency file saving feature is essential for this project. The client would like to see the fueling data is retractable even under the worst case scenario such as sudden power loss.

2.4 Verify/Validation

- Lab Tests

  To verify and validate that our modules function as intended, JUnit tests will be written and run on the modules. These tests will be largely be automated to ensure the quality of the testing. Some functions may require running the program and using human checks to verify if a function or module is working, such as in the case of the print mechanism. For these a suite of test will be run in the lab using the setup on the fuel trucks. These tests can encompass a test that does not require fuel to flow through the meter.

- Field Tests

  The optimum time is allowed to run tests on fuel trucks setup to deliver liquid propane and refined fuels. These tests will be more geared towards verifying that the information received from the meter is handled properly. Allowing the verification that the data delivered to the GUI application is consistent and valid.

2.5 Deliverables

- JNI Wrapper (x86 and x64 DLL)

  Software interface that allows the Java application to access the functionality of Liquid Controls SDK which is written in C. Since the application may be executed on either a 32 or 64 bit system, interfaces must be provided for both environments.

- Liquid Controls SDK (x86 and x64 DLL)

Compiled library binaries written in C to access the functionality of the Liquid Controls LCR-II flow meter.

- FlowMeter Implementation

  The FlowMeter software implementation is the focal point of our project. It's in this deliverable that we will design the process for performing expected operations on the flow meter device. The FlowMeter Interface is the socket that the Java GUI application uses to communicate with the hardware device (polling fuel amounts, printing receipts, etc.).

3. Implementation Plan

3.1 Proposed Solution

After evaluating our given functional specifications, we crafted many possible solutions to solving the problem at hand, eventually trimming it down to two. The implementation of the interface that is found between the Java GUI and the flow meter devices on the fuel truck is where we get to be creative in our design. There are two methods in which we could implement the FlowMeter Interface:

- Required (Features that are independent of the solutions below)
  - The FlowMeter implementation must match the FlowMeter Interface allowing the existing Java GUI to interact with the Liquid Controls LCR-II flow meter in a predictable fashion.
  - Redesign the interactions of the print operations so that the computer has no control over the content of the ticket. The content of the ticket must be decided solely by the flow meter devices themselves.
- Native Heavy Design
  - Write a JNI wrapper for the flow meter device so that any device operation can be performed in Java, and then use these Java method calls to implement the given FlowMeter Interface.

    Strengths
    - Performance. C/C++ is native code and performs quite a bit faster than Java based code. In addition to the change in platform performance, JNI only has to be called once per interface method call.

- Device Code will be performed on the platform that the creator of the device expected.

Weaknesses

- No one in the company knows C++ fluently. Our implementation would be far more important to our client if we write it in a language that they understand easily and can extend.

- Java Heavy Design
  - Implement the given interface methods in native C++ code and then simply have the java interface make a 1 line call to these methods.

    Strengths

    - Expandability
    - Developers have greater experience with Java (both our team and theirs)

    Weaknesses

    - Java applications do not deliver as good performance as native code.

Due to our limitation of time in this project we are forced to choose a method of implementation and go forth with it. We believe that by choosing the Java-Heavy design we are able to gain important traits such as extensibility in a modern programming language. Also if the implementation proved to be not efficient enough for the client's needs, then we have mitigation plans evolved for switching from one design to the other. This should prove quite easy because we would be able to decide the structure of the device operations that need to be called from the SDK. This structure of SDK calls could then be directly transferred to a native level library.

3.2 Risks

Risk	Mitigation Strategy
Performance of new implementation is worse than existing implementation	There will be consistent testing and prototyping throughout the development process to ensure a higher performance than the existing implementation.
Our implementation does not function properly	We will be testing with a flow meter and

once placed on a real fuel truck.	equipment from the fuel trucks during development.
Wrapping API does not result in minimum speed requirements	A prototype with the wrapper function will be developed early on in the project for testing and revision.
Implementing the communication via Bluetooth results in the application failing to function correctly	A prototype involving the Bluetooth implementation will be developed in the project for testing and revision.

4. Project Management

4.1 Resources

- People
  - Dan Oakland (doakland@oaklandcorp.com) - Initial developer of the application
  - Aaron Shaw (ashaw@oaklandcorp.com) - IT Director
  - Professor Gurpur Prabhu (prabhu@iastate.edu) - Project Advisor
- Equipment
  - Liquid Controls LCR-II flow meter register
  - Blackbox (multiplexer to handle multiple flow meters)
- Software
  - Java Runtime Environment 7
  - Eclipse IDE for Java
  - Visual Studios 2012 for C++
  - Git Version Control Software

4.2 Work Breakdown and Schedule

A task schedule and breakdown is available with a preliminary schedule in the attached document. Some tasks such as the breakdown for the revisions of this document and the Design Document may change. The modules listed might alter based on the results of the prototyping.

These items are a guideline and are more suited to finding the direction of the project which will be assigned and altered. This schedule was purposefully made with excess time in development and items such as fixing bugs that may be removed, but are being scheduled to allow for risk mitigation.

Note: Full version of the WBS is available on the project's site.