An Implementation of Java Native Interface (JNI) to Replace a Socket

Brandon Hessler
Penn State - World Campus
Enterprise Integration
SWENG568 Section 1
Program: Software Engineering M.S.

April 21, 2017

Contents

1 Project Introduction

The integration of applications is essential to the world we live in today. As programmers and software engineers we cannot effectively work or deliver the services necessary for our businesses or industries without integrating applications and systems together. With the world becoming more and more integrated and automated, the techniques that we learned in this course become more and more valuable.

On to my project. At work, I work on an application named TADA[1] that needs to integrate with another application, the AP[2] that controls signal attenuator switch boxes. Originally, we wanted my Java program to run the switch boxes outright but there was an issue of a lot of logic that went into the code for the program and not wanting to completely rewrite it. Plus, this issue was compounded by a classification issue that could arise in some situations: with that we decided on a connection between the two. Myself and the other programmer who integrated these three years ago did not know of an other ways to communicate between two programs other than sockets. We came up with the ICD[3] shown in appendix A. With these messages, I would send him one of these messages and his program (the AP) would either set the value I gave to him or respond with the requested values.

1.1 Business View of the Integration Needs

From the perspective of the business, these applications need to work together to provide the test engineers both information of the results of a test as well as let them control the attenuation of the signals during the test. With these tools integrated it will not only save money with the ability to have less people working during a test event, but will give greater figelity to the data collected. In my program, I am able to calculate what the result of the test is, but without the ability to collect the results straight from the AP during the test, we are unable to make the best test taking decisions that we can in real time. Instead of using sockets, the JNI[4] will allow us to not have the AP running during the test and instead will allow us to call the code already written in the AP from the TADA program.

2 Workflows or Process Flows Design (e.g., Integration logic across integrated applications)

The nice thing about using JNI in this context is that from my Java program I can do all of the initialization that is needed for the AP. There is no need to open up the AP as an application but we can instead use its functionalities as a background process of TADA. This means I do not have to make calls to the hardware directly, which would be hard since there are no Java drivers for that product so I would have to write them myself.

[1]Test Activity Data Aggregator
[2]Attenuator Program
[3]Interface Control Document
[4]Java Native Interface

2.1 Data Flow

The data flow for this integration can be seen in figure 1. This shows that TADA is making a call to the AP which will then in turn poll either of the attenuator switch boxes, and then return that data back to TADA, if asked. Some of the commands, like setting the Scan Rate (see message ID 15 in appendix A, are for the AP to regulate how often to tell the switch boxes to change. This flow of data may mean that the AP is not going to give that message to the switch boxes, but will give an entirely different one at the interval it was told to.

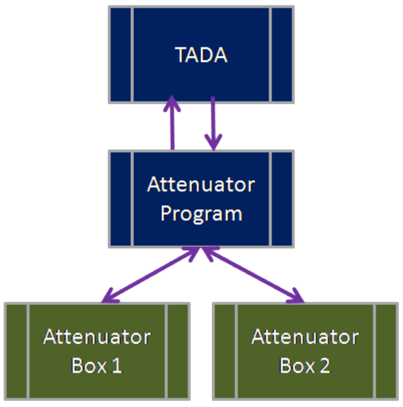


Figure 1: Shows the flow of data between TADA and the AP

2.2 Control Flow

Probably the most convenient feature of the JNI implementation is that the control will stay with the program that was originally designed to do so. The control will originate from TADA, but once a run is started, the AP will have control over the switch box hardware and will constantly be away of their values. From TADA I will be able to query those values whenever I need them from the AP. The control flow can be seen in figure 2.

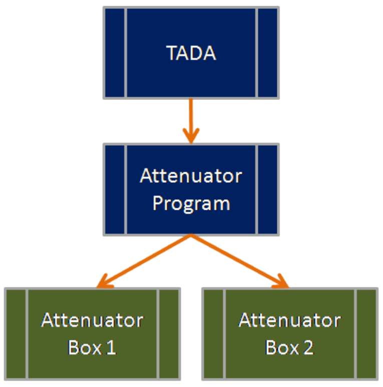


Figure 2: Shows the control flow TADA, the AP, and the switch box hardware

3 Integration Design (i.e., Approach to implementing the identified integration logic)

At a high level this integration should be fairly easy, remove the socket connection from my code in TADA and replace it with the JNI code. From The Breakfast Post website [Cannam, 2012] I looked at his step by step instructions on how to create JNI code and the C wrapper required to run it. Now all I needed to do was learn a bit of C and C++. The steps for the JNI and its wrapper seem to be fairly straightforward.

1. Find methods and classes that are to be called in the C++ code
   Ensure calls needed to make in C++ are public.

2. Make corresponding classes or methods within the Java code
   These must be identified in the code as native. You then leave those methods empty since the code that will be called will be from the C++ code.

3. Create the JNI wrapper in C
   This will correlate the Java code and methods to the C++ code and methods. It seems that using JNI is more writing in C than it is in Java.

4. Test code

3.1 Architecture Model

As you can see in figure 3 the JNI process is a fairly straightforward way to cross the gap between Java and C++ code. The only middle process required is that of the JNI wrapper. Note that this is not the entire architecture of both systems as those are both irrelevant to this discussion and also I am unable to give that out in this venue. The AttenuatorDriver is part of the Attenuator Program and its code is not given here as it is the property of the Navy and I am not able to distribute it. In 3 you will notice the colors of the lines. The red path is that for the real system that we are going to integrate this at work, the blue is the one for this project, and the purple path is the one followed in either case.

3.2 Functional Model

In this functional model (figure 4) you can see the functions that are implemented. You will notice that I am replacing all of the socket calls that were previously made via the byte arrays in appendix A as well as the calls for the AP to connect to the attenuator switch boxes. These switch boxes connect via socket, using IP address and port, to the AP using always localhost as the address and ports 5150 and 5151 for boxes one and two, respectively. As you will see in the code soon, I am writing the functions to take in a port just in case those change out of necessity, but they will be hard coded in a *constants* file in case later we need to ask the user's input; it would be easier to do that later on the Java side then to go back into the JNI code.

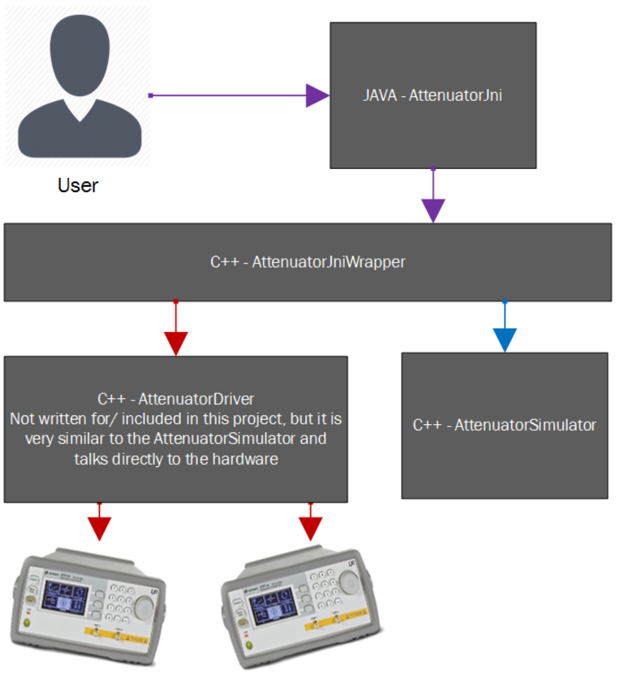
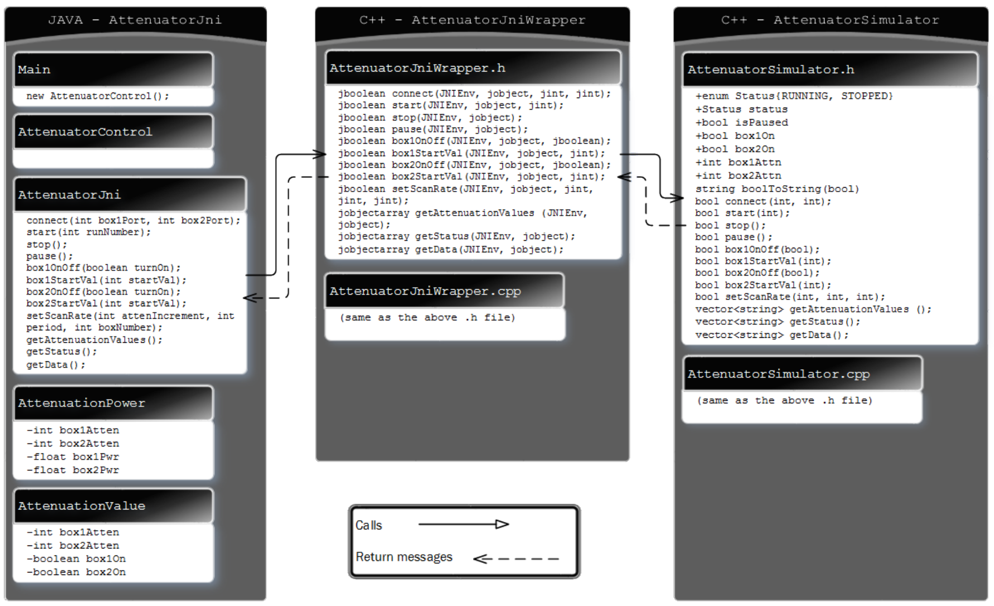

User
JAVA - AttenuatorJni
C++ - AttenuatorJniWrapper
C++ - AttenuatorDriver
Not written for/ included in this project, but it is very similar to the AttenuatorSimulator and talks directly to the hardware
C++ - AttenuatorSimulator

Figure 3: The architecture model of the JNI bridge between the applications



JAVA - AttenuatorJni

Main
```
new AttenuatorControl();
```

AttenuatorControl

AttenuatorJni
```
connect(int box1Port, int box2Port);
start(int runNumber);
stop();
pause();
box1OnOff(boolean turnOn);
box1StartVal(int startVal);
box2OnOff(boolean turnOn);
box2StartVal(int startVal);
setScanRate(int attenIncrement, int
period, int boxNumber);
getAttenuationValues();
getStatus();
getData();
```

AttenuationPower
```
-int box1Atten
-int box2Atten
-float box1Pwr
-float box2Pwr
```

AttenuationValue
```
-int box1Atten
-int box2Atten
-boolean box1On
-boolean box2On
```

C++ - AttenuatorJniWrapper

AttenuatorJniWrapper.h
```
jboolean connect(JNIEnv, jobject, jint, jint);
jboolean start(JNIEnv, jobject, jint);
jboolean stop(JNIEnv, jobject);
jboolean pause(JNIEnv, jobject);
jboolean box1OnOff(JNIEnv, jobject, jboolean);
jboolean box1StartVal(JNIEnv, jobject, jint);
jboolean box2OnOff(JNIEnv, jobject, jboolean);
jboolean box2StartVal(JNIEnv, jobject, jint);
jboolean setScanRate(JNIEnv, jobject, jint,
jint, jint);
jobjectarray getAttenuationValues (JNIEnv,
jobject);
jobjectarray getStatus(JNIEnv, jobject);
jobjectarray getData(JNIEnv, jobject);
```

AttenuatorJniWrapper.cpp
```
(same as the above .h file)
```

C++ - AttenuatorSimulator

AttenuatorSimulator.h
```
+enum Status{RUNNING, STOPPED}
+Status status
+bool isPaused
+bool box1On
+bool box2On
+int box1Attn
+int box2Attn
string boolToString(bool)
bool connect(int, int);
bool start(int);
bool stop();
bool pause();
bool box1OnOff(bool);
bool box1StartVal(int);
bool box2OnOff(bool);
bool box2StartVal(int);
bool setScanRate(int, int, int);
vector<string> getAttenuationValues ();
vector<string> getStatus();
vector<string> getData();
```

AttenuatorSimulator.cpp
```
(same as the above .h file)
```

Figure 4: The functional model of the JNI bridge function calls

4 Implementation and Programming

4.1 Classes Used

With the code that I am writing for this section I will include the following files and their purposes.

1. Java Classes
   (a) Main.java - Launches the GUI[5]
   (b) AttenuatorControl.java - GUI for the control of the attenuator functions. Some of these will be automated later but this will be useful for testing.
   (c) AttenuatorJni.java - This will contain the JNI calls that will call the code from the other application
   (d) AttenuationPower.java - Makes it easier for the program to output good java values from the JNI call *getData()*
   (e) AttenuationValue.java - Makes it easier for the program to output good java values from the JNI call *getAttenuationValues()*
   (f) RegExFieldFormatRestrictor.java - This is a class that extends Document that makes it easy to apply a regular expression to a JTextField for input validation (or in this case restriction). A coworker and I wrote this at work.
2. C++ Classes
   (a) AttenuatorSimulator.cpp - Returns the appropriate values for the information given form the Java GUI
   (b) AttenuatorSimulator.h - Header File for above
   (c) AttenuatorJniWrapper.cpp - This is the C++ file containing the code of the JNI wrapper, this is the bridge between the C++ code and the Java code
   (d) AttenuatorJniWrapper.h - This is the header file for the JNI wrapper

For this section I will explain the code that I needed to write to accomplish the communication from Java to C++ without the use of sockets. I will be explaining them from the front to the back end of the code, starting from the user interface and working back to the simulated hardware program (written in C++) which in practice will communicate with the hardware itself.

4.2 Java Code

Starting with the Java element, I wrote a mock-up of the GUI. In the final product, many of these processes would not necessarily have a button associated with it, but for testing purposes it made sense to make a button for each of the function calls necessary. Figure 5 is a screenshot of that GUI. The full code written will not be shown here but is available with this document as are the classes *AttenuationPower.java* and *AttenuationValue.java* since that code is altogether standard.

[5]Graphical User Interface

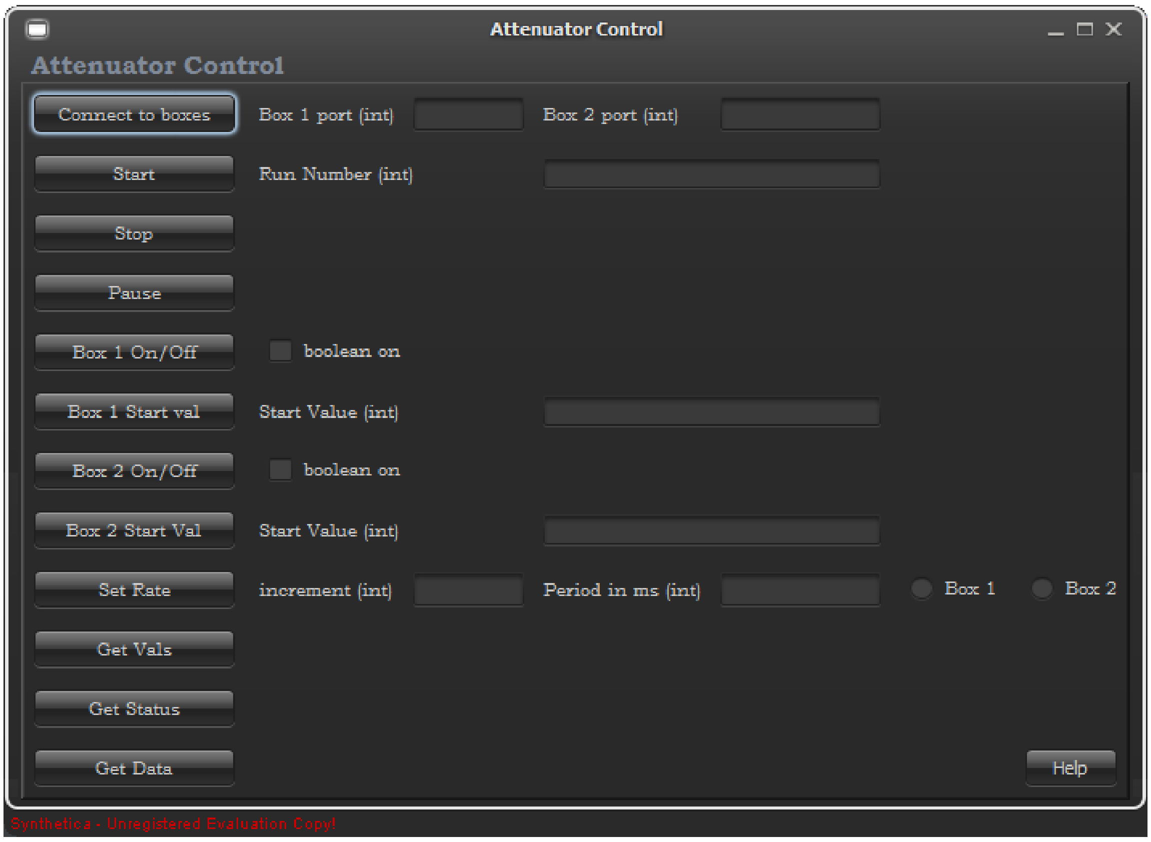
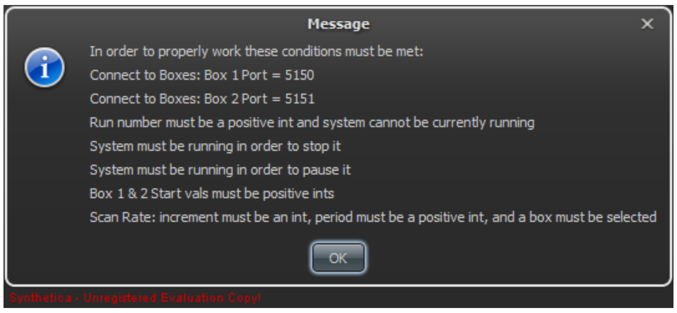


Figure 5: The graphical user interface of the test program

The only part of the GUI code I will show is the part that uses the RegExFieldFormatRestrictor and that is so you are not confused when you try to type wrong values in the JTextFields (Listing 1). I did get the code for the port number regular expression from StackOverflow [npinti, 2012]. The code that makes the JNI calls though is important to include here and is shown below in Listing 2. This shows that here we are making the *native* calls that will be calling the methods from the AttenuatorJniWrapper dll that is included in the Java project; the details of which I will explain now.



In order to properly work these conditions must be met:
Connect to Boxes: Box 1 Port = 5150
Connect to Boxes: Box 2 Port = 5151
Run number must be a positive int and system cannot be currently running
System must be running in order to stop it
System must be running in order to pause it
Box 1 & 2 Start vals must be positive ints
Scan Rate: increment must be an int, period must be a positive int, and a box must be selected

Figure 6: The help dialog that gives the tester hints on how to use this properly

Listing 1: AttenuatorControl RegEx

```
        tfRunNumber.setDocument(new
            RegExFieldFormatRestrictor("^[0-9]*[1-9][0-9]*$", false));
        tfBox1StartVal.setDocument(new
            RegExFieldFormatRestrictor("^[0-9]*[1-9][0-9]*$", false));
```

```
       tfBox2StartVal.setDocument(new
          RegExFieldFormatRestrictor("^[0-9]*[1-9][0-9]*$", false));
       tfBox1Port.setDocument(new RegExFieldFormatRestrictor(
              "^([0-9]{1,4}|[1-5][0-9]{4}|6[0-4][0-9]{3}|" +
                     "65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5])$",
                        false));
       tfBox2Port.setDocument(new RegExFieldFormatRestrictor(
              "^([0-9]{1,4}|[1-5][0-9]{4}|6[0-4][0-9]{3}|" +
                     "65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5])$",
                        false));
       tfRateIncrement.setDocument(new
          RegExFieldFormatRestrictor("-?[0-9]{0,10}", false));
       tfRatePeriod.setDocument(new
          RegExFieldFormatRestrictor("^[0-9]*[1-9][0-9]*$", false));
```

Listing 2: AttenuatorJni Java native code

```
package com.brandonhessler.Attenuator;

/**
 * Created by Cpl Hess on 4/15/2017.
 */
public class AttenuatorJni {

   public native boolean connect(int box1Port, int box2Port);
   public native boolean start(int runNumber);
   public native boolean stop();
   public native boolean pause();
   public native boolean box1OnOff(boolean turnOn);
   public native boolean box1StartVal(int startVal);
   public native boolean box2OnOff(boolean turnOn);
   public native boolean box2StartVal(int startVal);
   public native boolean setScanRate(int attenIncrement, int period, int
       boxNumber);

   //the values returned will be in the form of (int
       box1AttenuationValue,
   // int box2AttenuationValue, boolean box1On, boolean box2On)
   private native String[] getAttenuationValues();

   //the status returned will be in the form of (boolean runInProgress,
       boolean isPaused)
   public native String[] getStatus();

   //The data returned will be in the form of (float box1Power, float
       box2Power,
   // int box1AttenuationCurrentValue, int box2AttenuationCurrentValue
   private native String[] getData();

   static {
       System.loadLibrary("AttenuatorJniWrapper"); // Load native
          library at runtime
```

```
    }

    public AttenuatorJni() {
    }

    /**
     * This will use the JNI to get the data and will convert them into
         the correct data types
     * @return The string of all of the values
     * float box1Power,
     * float box2Power,
     * int box1CurrentAttenuationValue,
     * int box2CurrentAttenuationValue
     */
    public String getDataMessage() {
        return new AttenuationPower(new
            AttenuatorJni().getData()).makeString();
    }

    public String getAttenuationValueMessage() {
        return new AttenuationValue(new
            AttenuatorJni().getAttenuationValues()).makeString();
    }

    public String getStatusMessage() {
        StringBuilder stringBuilder = new StringBuilder();
        String[] strings = getStatus();
        boolean[] bools = new boolean[2];
        bools[0] = stringToBoolean(strings[0]);
        bools[1] = stringToBoolean(strings[1]);

        stringBuilder.append("The Run is currently ");
        if (!bools[0])
            stringBuilder.append("Not ");
        stringBuilder.append("Running");
        stringBuilder.append("\n");
        stringBuilder.append("And is currently ");
        if (!bools[1])
            stringBuilder.append("Not ");
        stringBuilder.append("Paused");
            return stringBuilder.toString();
    }

    private boolean stringToBoolean(String str) {
        return str.equalsIgnoreCase("true");
    }

}
```

4.3 C++ Code

The AttenuatorJniWrapper dll was created from the AttenuatorJniWrapper.cpp and AttenuatorJniWrapper.h files that I alluded to above. The AttenuatorJni-Wrapper.h file can be created or generated through the use of **javah**. The javah tool is executed via command line and turns a java file (in this case the Attenuator-Jni.class) into a header file for use in JNI. Notice I said the .class file, not the .java file, this was something that I did not realize and took me about five hours to figure out why my commands in the command line were not working. These files, along with the AttenuatorSimulator.lib file, form the bridge between the simulator and the Java code (and GUI). As you can see below in Listing 3 this code takes calls from the Java code, calls the C++ code (the attenuator simulator), then translates the C++ object types into Java objects that can be returned, then returns them to Java. Since all of the necessary information can be gained through looking at just the .cpp file, I will not clutter this paper further with the .h file.

Listing 3: AttenuatorJniWrapper code

```
  //
// Created by Cpl Hess on 4/15/2017.
//
//#include <stdio.h>
//#include <stdlib.h>
#include "AttenuatorJniWrapper.h"
//#include <jni.h>
//#include "pch.h"
#include "AttenuatorSimulatorVS.h"

static SimulatedAttenuators::Sim library;


JNIEXPORT jboolean JNICALL
   Java_com_brandonhessler_Attenuator_AttenuatorJni_connect(JNIEnv *env,
   jobject obj, jint port1, jint port2) {
  jboolean jsuccess;
  int cPort1 = port1;
  int cPort2 = port2;
  bool success = library.connect(cPort1, cPort2);
  jsuccess = (jboolean)success;
  return jsuccess;
}

/*
* Class:     com_brandonhessler_Attenuator_AttenuatorJni
* Method:    start
* Signature: (I)Z
*/
JNIEXPORT jboolean JNICALL
   Java_com_brandonhessler_Attenuator_AttenuatorJni_start
(JNIEnv *env, jobject obj, jint jrunNumber){
  jboolean jsuccess;
  bool success;
  int runNum = jrunNumber;
```

```
   success = library.start(runNum);
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   stop
* Signature: ()Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_stop
(JNIEnv *env, jobject obj){
   jboolean jsuccess;
   bool success = library.stop();
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   pause
* Signature: ()Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_pause
(JNIEnv *env, jobject obj){
   jboolean jsuccess;
   bool success = library.pause();
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   box1OnOff
* Signature: (Z)Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_box1OnOff
(JNIEnv *env, jobject obj, jboolean jBool){
   bool on = jBool;
   jboolean jsuccess;
   bool success = library.box1OnOff(on);
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   box1StartVal
* Signature: (I)Z
```

```
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_box1StartVal
(JNIEnv *env, jobject obj, jint jStartVal){
   int startVal = (int)jStartVal;
   jboolean jsuccess;
   bool success = library.box1StartVal(startVal);
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   box2OnOff
* Signature: (Z)Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_box2OnOff
(JNIEnv *env, jobject obj, jboolean jBool){
   bool on = jBool;
   jboolean jsuccess;
   bool success = library.box2OnOff(on);
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   box2StartVal
* Signature: (I)Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_box2StartVal
(JNIEnv *env, jobject obj, jint jStartVal){
   int startVal = (int)jStartVal;
   jboolean jsuccess;
   bool success = library.box2StartVal(startVal);
   jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   setScanRate
* Signature: (III)Z
*/
JNIEXPORT jboolean JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_setScanRate
(JNIEnv *env, jobject obj, jint jIncrement, jint jPeriod, jint jBoxNum){
   int increment = (int)jIncrement;
   int period = (int)jPeriod;
   int boxNum = (int)jBoxNum;
   bool success = library.setScanRate(increment, period, boxNum);
   jboolean jsuccess = (jboolean)success;
   return jsuccess;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   getAttenuationValues
* Signature: ()[Ljava/lang/String;
*/
JNIEXPORT jobjectArray JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_getAttenuationValues
(JNIEnv *env, jobject obj){
   vector<string> strings = library.getAttenuationValues();

   jobjectArray vals = (jobjectArray)env->
      NewObjectArray(4, env->FindClass("java/lang/String"),
          env->NewStringUTF(""));

   for (unsigned long i = 0; i < 4; i++) {
      env->SetObjectArrayElement(vals, i,
          env->NewStringUTF(strings.at(i).c_str()));
   }
   return vals;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   getStatus
* Signature: ()[Z
*/
JNIEXPORT jobjectArray JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_getStatus
(JNIEnv *env, jobject obj){
   vector<string> strings = library.getStatus();

   jobjectArray vals = (jobjectArray)env->
      NewObjectArray(2, env->FindClass("java/lang/String"),
          env->NewStringUTF(""));

   for (unsigned long i = 0; i < 2; i++) {
      env->SetObjectArrayElement(vals, i,
          env->NewStringUTF(strings.at(i).c_str()));
   }
   return vals;
}

/*
* Class:    com_brandonhessler_Attenuator_AttenuatorJni
* Method:   getData
* Signature: ()[Ljava/lang/String;
*/
```

```
JNIEXPORT jobjectArray JNICALL
    Java_com_brandonhessler_Attenuator_AttenuatorJni_getData
(JNIEnv *env, jobject obj){
   vector<string> strings = library.getData();

   jobjectArray vals = (jobjectArray)env->
      NewObjectArray(4, env->FindClass("java/lang/String"),
          env->NewStringUTF(""));

   for (unsigned long i = 0; i < 4; i++) {
      env->SetObjectArrayElement(vals, i,
          env->NewStringUTF(strings.at(i).c_str()));
   }
   return vals;
}
```

To make the AttenuatorJniWrapper class work it needs to know what C++ calls to make, that is why I had to include the AttenuatorSimulator.lib file in the project (or at least link the project to it). That static library was created from the AttenuatorSimulator.h and .cpp files. These files have a few basic variables that do not do much other than verify that all of the calls coming from the Java code are working, the values given are valid, and the calls made are valid. I will only show a few lines of the AttenuatorSimulator.h file (Listing 4) since I declare a few things that are necessary to the .cpp file, but most everything of note is in the .cpp file.

Listing 4: AttenuatorSimulator global variables

```
   #pragma once
#include <stdio.h>
#include <string>
#include <vector>
using std::vector;
using std::string;
using std::to_string;

namespace SimulatedAttenuators {
   class Sim {
   public:
      enum Status {
         RUNNING, STOPPED
      };
      Status status = STOPPED;
      bool isPaused = false;
      bool box1On = false;
      bool box2On = false;
      int box1Attn = 0;
      int box2Attn = 0;
```

This .cpp file (Listing 5) shows the validation that the program would use before trying to send signals to the signal attenuator switch boxes such as to ensure that they are running before telling them to stop and only allowing them to pause their steps if they are already running. Otherwise, this class is returning false values that would usually be reserved for the actual values that the hardware is set to.

Listing 5: AttenuatorSimulator end code

```
   #include "AttenuatorSimulatorVS.h"
#include <iostream>

namespace SimulatedAttenuators {

   string Sim::boolToString(bool b) {
      return b ? "true" : "false";
   }

   bool Sim::connect(int port1, int port2) {
      status = STOPPED;
      return port1 == 5150 && port2 == 5151;

   }

   bool Sim::start(int runNumber) {
      if (runNumber > 0 && status == STOPPED) {
         status = RUNNING;
         return true;
      }
      else
         return false;

   }

   bool Sim::stop() {
      if (status == RUNNING) {
         status = STOPPED;
         return true;
      }
      else
         return false;
   }

   bool Sim::pause() {
      if (status == RUNNING) {
         isPaused = !isPaused;
         return true;
      }
      else
         return false;
   }

   bool Sim::box1OnOff(bool turnOn) {
      if (turnOn != box1On) {
         box1On = turnOn;
         return true;
      }
      else
         return false;
   }
```

```
bool Sim::box1StartVal(int startVal) {
   if (startVal > 0 && startVal < 256) {
      box1Attn = startVal;
      return true;
   }
   else
      return false;
}

bool Sim::box2OnOff(bool turnOn) {
   if (turnOn != box2On) {
      box2On = turnOn;
      return true;
   }
   else
      return false;
}

bool Sim::box2StartVal(int startVal) {
   if (startVal > 0 && startVal < 256) {
      box2Attn = startVal;
      return true;
   }
   else
      return false;
}

bool Sim::setScanRate(int attenIncrement, int period, int boxNumber) {
   return period > 0 && (boxNumber == 1 || boxNumber == 2);
}

vector<string> Sim::getAttenuationValues() {
   vector<string> vals(4);
   vals[0] = to_string(box1Attn);
   vals[1] = boolToString(box1On);
   vals[2] = to_string(box2Attn);
   vals[3] = boolToString(box2On);
   return vals;
}

vector<string> Sim::getStatus() {
   vector<string> statusVector(2);
   if (status == RUNNING)
      statusVector[0] = "true";
   else
      statusVector[0] = "false";

   if (isPaused)
      statusVector[1] = "true";
   else
      statusVector[1] = "false";
```

```
    return statusVector;
  }

  vector<string> Sim::getData() {
    vector<string> vals(4);
    vals[0] = to_string(box1Attn);
    vals[1] = to_string(box2Attn);
    vals[2] = to_string(3.14159f);
    vals[3] = to_string(2.71828f);
    return vals;
  }
}
```

5 Bibliography

References

[Cannam, 2012] Cannam, C. (2012). Wrapping a c++ library with jni. `https://thebreakfastpost.com/2012/01/21/wrapping-a-c-library-with-jni-introduction/`.

[npinti, 2012] npinti (2012). Stakoverflow - regex to validate port number. `http://stackoverflow.com/questions/12968093/regex-to-validate-port-number`.

Appendices

A Previous ICD

Interface Control Document (ICD) For Test Activity Data Aggregator (TADA) and Attenuator Program (AP)

From TADA to AP

Message	Data Type	Variable
Start Command	int	ID = 1
	int	RunNumber

Message	Data Type	Variable
Stop Command	int	ID = 2

Message	Data Type	Variable
Pause Command	Int	ID = 4

Message	Data Type	Variable
Box 1 Command	Int	ID = 11
	boolean	Pwr1On

Message	Data Type	Variable
Box 1 Command	Int	ID = 12
	Int	dBAttenuationStartValue

Message	Data Type	Variable
Box 2 Command	int	ID = 13
	boolean	Pwr2On

Message	Data Type	Variable
Box 2 Command	Int	ID = 14
	Int	dBAttenuationStartValue

Message	Data Type	Variable
Scan Rate Command	Int	ID = 15
	Int	dBAttenuationIncrement
	Int	PeriodInSeconds
	Int	Box1/Box2

1 is box 1, 2 is box 2, all others thrown out

Message	Data Type	Variable
GetAttenuationValues	int	ID = 21

Message	Data Type	Variable
GetStatus	int	ID = 22

Message	Data Type	Variable
GetData	Int	ID = 23

From AP to TADA

Message	Data Type	Variable
AttenuationValues	int	ID = 101
	int	Box1AttenValue
	int	Box2AttenValue
	boolean	Box1On
	boolean	Box2On

Message	Data Type	Variable
Status	int	ID = 102
	boolean	RunInProgres
	boolean	Pause

Message	Data Type	Variable
Data	int	ID = 103
	float	Box1Pwr
	float	Box2Pwr
	int	Box1AttenuationCurrentValue
	int	Box2AttenuationCurrentValue