Creating a cross-language application

Bridging Java and Native code using JNI

LTH School of Engineering at Campus Helsingborg

Bachelor Thesis:
Rickard Ingemansson
David Skog

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund

Abstract

This thesis is a report from a project where an application connecting Java and native code was made. The question answered by this report is how integration between Java and native code can be achieved. The application was made specifically for SAAB Training Systems who specialize in training simulations for military purposes. The application, which is called a driver was written partially in C++ and partially in Java code and was constructed using the JNI and JNA frameworks. The driver would on completion become a part of the WISE integration platform, a software suit made by SAAB Training Systems. The project was successful resulting in a functioning application and this report. The report will document the functionality and structure of the driver and also how JNI and JNA makes the implementation work. The experience of using JNI and JNA, both pitfalls and successes, are documented as well. Hopefully this can provide the reader with helpful information and aid any further development of the driver or any development integrating native code with Java.

Keywords:
Integration, Java, JNI, JNA, Driver

Sammanfattning

Den här avhandlingen är en rapport från ett projekt där en applikation som integrerar Java och nativekod skapades. Frågan som besvaras av den här rapporten är hur integrationen kan möjliggöras. Applikationen gjordes på uppdrag av SAAB Training Systems, ett företag som specialiserat sig på träningssimulationer för militärt bruk. Applikationen, kallad en driver, skrevs delvis i C++ och delvis i Java kod och konstruerades med hjälp av JNI- och JNA- ramverken. Drivern skulle när den var färding inkluderas i WISE, en integrationsplatform gjord av SAAB Training Systems. Projektet var lyckat och resulterade i en fungerande applikation samt den här rapporten. Rapporten dokumenterar driverns funktionalitet och struktur samt hur JNI och JNA gör detta möjligt. Erfarenheten av att använda JNI och JNA, såväl motgångar som framgångar, dokumenteras också. Förhoppningsvis kan detta ge läsaren information som kan hjälpa vid vidarutveckling av drivern såväl som annan integration mellan native och Java kod.

Nyckelord:
Integration, Java, JNI, JNA, Driver

Contents

Chapter 1

Introduction

Where does synchronization between different platform based computer programs take place, and how can this be achieved with minimal impact on the end users time and workload? SAAB Training Systems have developed an integration platform, WISE, which strives to do exactly that. So what is WISE and what was actually produced? The idea behind the WISE platform is to relieve customers from tedious and costly integration work between different APIs. There are integration platforms similar to WISE, on such platform is CORBA[5] which is developed by OMG. However, because SAAB Training Systems have developed WISE, other similar integration platforms will not be considered. WISE makes use of applications called drivers to integrate applications based on different code languages. A driver acts as a sort of interpreter between the two worlds that translates data structures from one side and makes it into something similar, understood by the other side.

This thesis is the documentation of how the programming frameworks JNI and JNA where used to make Java code talk to a computer application called WISE which is based on C and C++ code.

This was a thesis work done by two students from LTH Campus Helsingborg and the task was provided by SAAB Training Systems. SAAB Training Systems specializes in training simulations for military and civilian use. The thesis work was done at the SAAB Training Systems office located in Helsingborg Sweden. The driving question behind this thesis is how to make the integration work, but this is impossible to evaluate without going into the different behavior of C, C++ and Java. Because of this, when the differences between the languages affect the implementation of the driver they will also be documented. This is not, however, an attempt to map differences between the programming languages in general as that is far too big for the scope of the thesis.

1.1 Purpose

The purpose of the project, or thesis work, was to construct an application requested by SAAB Training Systems called the Java Driver. The application makes it possible for a Java application to connect and communicate with SAAB Training Systems system that is made in C and C++ code. The question that must be answered is how to construct such a program considering communications ways, system structure and marshalling between the Java and nativ code. The purpose of this thesis is to give an in depth explanation of the previous question of how integration between native code and Java is done by using the frameworks JNI and JNA and also to explain how the Java Driver works and is structured.

- Question: How can integration between Java and native code be achieved using the JNI and JNA frameworks?
- Purpose: Constructing a functioning marshalling driver able to connect Java applications with an existing program written in C and C++ code.

1.2 Scope

This is what the report covers

- The functionality of the software that was produced.
- How JNI and JNA where used to reach a working implementation.
- Some information about the target system WISE.

Demarcation

- Differences between C, C++ and Java. Although these differences have shaped the implementation at times covering them all would be a far too big undertaking for this report. The ones that have affected the design greatly will be mentioned.

1.3 Report structure

For anyone not familiar with the WISE system this report should be read from start to finish, the structure is intended to begin with the target system and the structural requirements of the driver before going into the details of the application.

- Chapter 1. The introduction gives the background to this project. What was investigated? What was produced? Why was this project done and what where the expected results.

- Chapter 2. The Process. This chapter explains how this thesis work was done from start to finish detailing information gathering, implementation, testing and documentation.
- Chapter 3. WISE explains the target system, the software that will incorporate the application produced during the course of this project.
- Chapter 4. Java Native Interface is the framework, or you could call it the tool set that is used to enable Java and native code to communicate. The third chapter gives a brief introduction to what JNI is and also explains JNA which is a further development of JNI. Any negative aspects of using the two are also documented here.
- Chapter 5. The Driver chapter is the main chapter of the thesis. It details the different classes that make up the application. The start gives an overview of how the system is laid out in design and the communication ways. The following sections document the classes written in C++ and Java.
- Chapter 6. This section details the process of marshalling data.
- Chapter 7. The discussion chapter is where we analyze problems, solutions and design of the driver application. Ideas for what could be done differently, problems that have affected the implementation and things to be aware of when doing a similar project are covered here.
- Chapter 8. Conclusion, this is the end of the thesis. The results of the project are evaluated.
- Chapter 9. Vocabulary, acronyms and words are explained here.
- Chapter 10. References, a list of references used during the project when searching for information.

1.4 Deliverables

At the end of the project the following was delivered to SAAB, the advisor and the examiner at LTH Campus Helsingborg

- Java Driver, SAAB only.
- Thesis, SAAB and LTH.

Chapter 2

The Process

This degree project was announced by SAAB Training Systems, they had a clear idea of what they wanted and so the foundation of the project had a clear goal of what was to be delivered at the end. SAAB had previously constructed similar software aiming at other APIs, a C# driver already existed and was given to us as a guideline if not a template for what was needed. For all the similarities that programming languages share there are still a lot of differences that need studying before heading into the stage of development. We choose to break the project down into few phases that we would repeat until the project was done. The best way of describing the process in words would probably be to call it iterative development where information gathering, implementation and testing where done during the entire length of the project. Each iteration would last one week with a target goal ending with testing and evaluation of the results. As the project progressed the amount of information gathering would give way to more implementation and testing which is to be expected.

Key point for the method: gather information $->$ implement one part of the driver $->$ testing $->$ when satisfied, repeat.

The phases where defined to the following:

2.1 Information gathering

There are a few methods available when integrating Java with native code, the JNI and JNA frameworks are examples of this and there are also a couple of software bundles that claim to generate wrapping methods for native code to expose it to Java code and vice versa. We decided not to look into the later as it would probably not give enough freedom to our implementation. After the initial period just at the start of this project we decided we would use both JNI and JNA for our implementation,

JNA would be used when making function calls from Java to native code and JNI would be used when going from native to Java. An explanation to why we use both is given in the chapter 4.

2.2 Implementation

To help the information gathering process along the implementation was split down into small parts. The key parts of the driver where identified and given their own development phase. For example to launch the JVM or to marshall the different data types aren't dependent on each other and could be tackled separately. Once we where able to launch a Java environment with all that in entails and test it, we could move on to the next iteration focusing on another part. Each part of the implementation has it's own section in the report in chapter 5 "The Driver".

2.3 Testing

Initially testing was done without the WISE environment, which does provide testing tools. The Java data structures where given basic functionality and required only simple populating and retrieving tests. The testing done when calling Java functions from native code was slightly harder since any crashes on the Java side couldn't be debugged from the native side compiler. We chose to work around this using trace out prints in the Java code. Admittedly this is not a perfect solution as it is very time inefficient. The ending stage of testing was done using the WISE test tools towards a live and running instance of WISE.

2.4 Documentation

The documentation work ran parallel to the entire project. From the start of the project a structure for the report was made and day to day progress was kept in a "diary". The main focus of the diary would be problems that where solved along the way and also to keep a task list of what needed to be done. A lot of the writing ending up in this report will be of a technical kind since it will be used as documentation of the driver for SAAB to use. Naturally most of that work could not be done before the Java Driver was nearing completion.

Chapter 3

WISE

WISE is a SAAB Training Systems built software suit which purpose is to connect different computer applications into a common environment, SAAB describes WISE as an information infrastructure. The idea behind WISE is to relieve the end user of as much programming as possible, leaving them with more time to configure their environment and analyzing results instead of building an integration structure. The structure of a connectivity is to have a central backbone information model that keeps and stores data received by the connected external applications.

The applications communicate with WISE through drivers, a sort of bridge between the application and the native backbone. The driver translates the data flowing between the application and the information model, by doing so the need to change applications connecting to the common environment is eliminated. In the end this saves a lot of time, once a driver that handles for instance Java to native code has been written all Java based applications can connect to a WISE integration model. A WISE connectivity setup consists of one or more information models and a connectivity layer. This connectivity setup is then used by user applications.

A data manager is responsible for synchronizing data between its local database and any connected application. A driver such as the one created in this project works as a translator between the information model of an application and the rest of the system and so the driver is part of what is called the connectivity layer. An application is never aware of any protocol used to talk to the common environment, this is handled by the driver. The driver also handles the transformation of data to types compliant with the central hub and the applications information model.

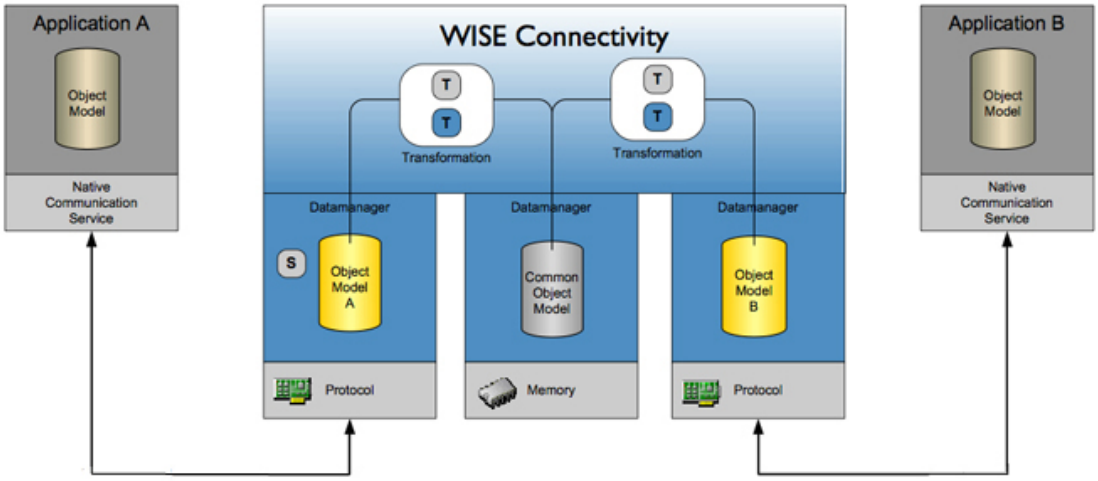

Figure 1.*A WISE integration model showing two applications connected to a central WISE database.*

Figure 1. illustrates a normal connectivity configuration and the connections between the different data models. Object model A and B have their own template and object database, they communicate with the common object model connecting the two. Between object model A and B the data is transformed to comply with the data stored in the common object model memory. This could be two Java applications, A and B, connected to the backbone database (the central Common Object Model) written in C++. The Protocol in the picture would then correspond to the driver developed during this project wich makes it possible for the two java applications to communicate with the native based backbone database. When this project was ongoing there were already existing marshalling drivers made for other programming languages. For example C#. Literature reference [4]

Chapter 4

Java Native Interface (JNI) and Java Native Access (JNA)

4.1 JNI

JNI is a framework for handling communication between native and Java code. It was developed by Sun and was introduced and supported since Java release 1.1. In the JavaDriver all communication going from native to Java code use JNI. JNI enables native code to access and use Java objects much in the same way that they are used by java code. It also supports launching of a JVM that native code can access and execute Java applications in.

JNI contains mapping between Java and native types as this table illustrates:

Native Type	*Java Language Type*	*Description*	*Type Signature*
unsigned char	jboolean	unsigned 8 bits	Z
unsigned short	jchar	unsigned 16 bits	C
short	jshort	signed 16 bits	S
long	jint	signed 32 bits	I
long long __int64	jlong	signed 64 bits	J
float	jfloat	32 bits	F
double	jdouble	64 bits	D

JNI uses an environment pointer to the JVM inorder to make communication between native and Java possible. The environment pointer is used to fetch id for Java classes, Java methods and Java variables. These ids are used to invoke the methods and variables. For more information on JNI see reference [3].

4.1.1 Datatypes provided by JNI

There are some datatypes provided by JNI that solves the problem with complex datatypes that are not mapped. A complex dataype in Java is represented as jobject in native code. To actually get the Java class type from the jobject the environment pointer to the JVM is used to fetch an instance of a jclass object. The jclass object is the representaion of the Java datatype and sent to methods that invoke the java object so the JVM knows wich .class file to use.

4.1.2 Launching the JVM

As said the JNI handles the creation of the JVM. This is done in the native code since the Java User Application is created and launched from native code. The full path to the java class files must be provided. In this way the JVM finds the java classes when the native code requires it. The JVM.dll file must also be loaded in order to create JVM. To do this the LoadLibrary method located in Winbase.h is invoked and the full path to the JVM.dll is passed to it as an argument. It is important not to move JVM.dll to the project in VisualStudio but instead give the full path. Once the JVM.dll is loaded JNI can create the JVM. This is done by invoking the method JNI_CreateJavaVM. Arguments to this method are a pointer to the JVM and a pointer to an environment inside JVM.

4.1.3 Destroy the JVM

JNI also is responsible for destroying the JVM. This is done with the JVM pointer. To be able to destroy the JVM all threads but the main thread must first detach from the JVM, this is required by JNI and cannot be altered. One way of making sure this is done is to only attach threads as daemon threads. A daemon thread detached automatically when the JVM is destroyed so there is no need to wait until all threads are detached.

4.2 Java Native Access (JNA)

When the system makes a call from Java code to native it makes use of the JNA framework. JNA is simply easier to use than JNI for this purpose as it does not require extra code wrapping of the invocation calls nor does it need to load a .dll made of the native code. To use JNA, you must add the jna.jar[1] file as an external library to your Java project. When this is done the only thing needed in the Java code is to prefix a function call with the native word, like this:

[1] http://java.net/projects/jna/downloads/directory - 2011-06-13

```
native int foo(arguments);
```

This is strictly speaking about the JavaDriver implementation. In the native code, the JNI function RegisterNatives is used to map the functions between the Java and native environment. If this is not done the JNA call above will not work. For more information on JNI see reference [2].

4.3 Shortcomings of JNI and JNA

Although not explored much in this project as it lies outside the scope of the thesis, both JNI and JNA seem to work poorly when using a 64 bit JDK. At the start of the project several attempts where made to launch a JVM when linking JNI with a 64 bit jvm.dll but without success.
JNI currently does not support multiple JVMs to be launched in the same process[2].

[2]http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/invocation.html#wp16334 2011-06-13

Chapter 5

The Driver

5.1 Structure

In order to make WISE which is written in native code communicative with applications written in Java, a driver between the two programming language worlds must be present. The driver that has been developed contains objects written in both Java and C++. The driver lets Java code call and be called by native applications.

Figure 2. *Marshalling driver connecting WISE to a Java application.*

To at all be able to make calls between Native language and Java language there must be a framework dealing with this task. Two frameworks are used: Java Native Interface(JNI) and Java Native Access(JNA) which are described in chapter 3.

5.1.1 Communication ways

The classes that handle the communication between Java and Native are:

1. JavaDriver (Native)
2. JNISink (Native)
3. The java user application (Java)
4. CJavaWISEDriver (Java)

When WISE sends out a message the JavaDriver.cpp is called. JavaDriver.cpp then uses JNI to call the corresponding method in the Java user application. When the

Java application driver call WISE it uses the CJavaWISEDriverSink, which uses JNA to communicate with native code. The code communicating directly with WISE is strictly native. Hence the java part of the driver does not communicate directly with WISE but must instead go through C code and then C++ code. The Java class CJavaWiseDriverSink calls the class JNISink(native). The JNISink class has global methods written in C in its .cpp file. These are the methods that are called from Java. The JNISink calls a class to get a pointer to the correct JavaDriver.cpp since only the JavaDriver.cpp can communicate with WISE. When received pointer, global method in JNISink invokes WISE through the JavaDriver pointer.

Figure 3 illustrates the communication pathways of the different classes that make up the JavaDriver:

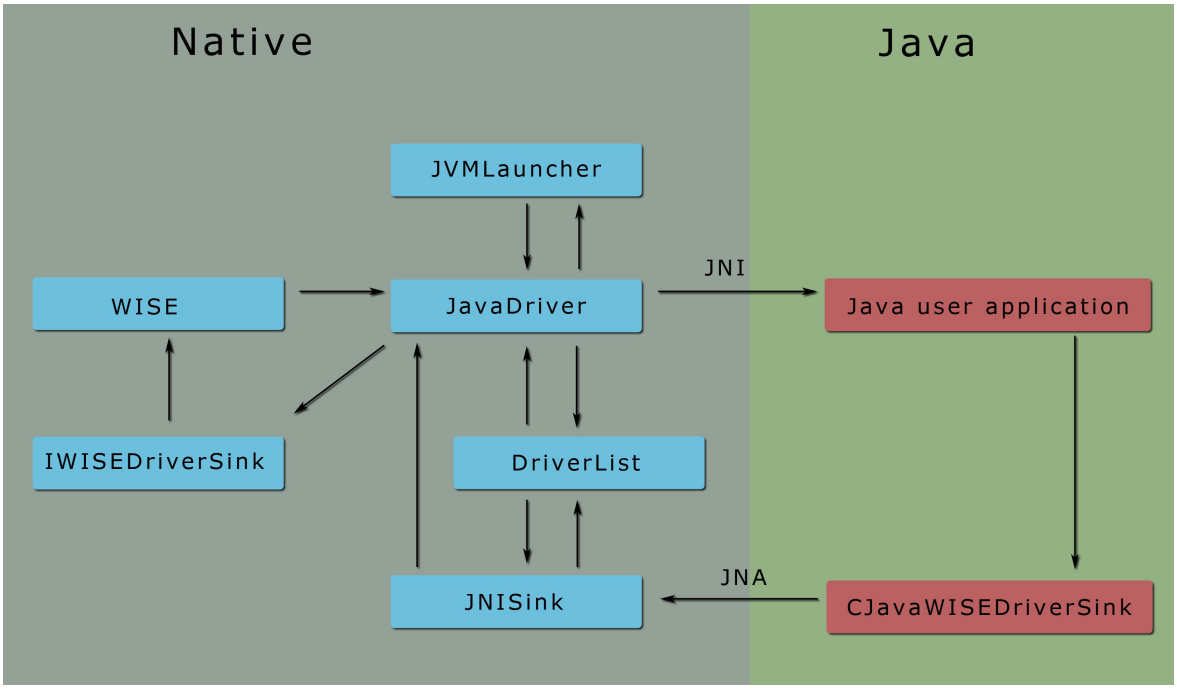

Figure 3. *The system structure including communication pathways.*

5.2 Java Part

The JavaDriver consists of a native code part written in C++ and a Java part. The Java part of the solution consists of data types mirroring their C++ counterparts. Apart from a brief explanation of these objects, any deviations from their WISE counterparts will also be detailed here. These differences are either due to design choice or differences in programming languages, in whichever case missing functionality will be highlighted for future development. A preexisting driver written in C# has also been used as a reference when creating the Java classes. Almost every Java class will be sorted into different packages. This is simply to stay away from the Default Package[1] directory from which nothing can be imported. Previously the Default Package held files that where hard to categorize but compared to having slightly vague resemblance to the package name, not being able to import classes is a much bigger problem.

5.2.1 The Sink

CJavaWISEDriverSink:
The CJavaWISEDriverSink is more commonly referred to as the Sink and will be called so in this chapter. The Sink contains all methods available to a user constructing a Java application. Just like the native version of the Sink all calls from the Java environment to WISE will be made using the Sink object. Looking at the different methods, all in all there are 238 of them although most are not unique. The major part of them are simply overloaded versions with different arguments. The Sink contains no logic operations, actually no code on the Java side does, it simply takes arguments and passes them on to the native code. The reason function calls made from Java to native code are wrapped is because the native call cannot return a WISE_RESULT. Here is an example of a function call:

```
public WISE_RESULT GetDatabaseType(DatabaseHandle hDatabase, ...){
     int result = getDatabaseType(hDatabase, ...);
     return new WISE_RESULT(result);
}
native int getDatabaseType(DatabaseHandle hDatabase, ...);
```

The GetDatabaseType function call does two things, it creates a WISE_RESULT for the Java environment and also calls the native function getDatabaseType which passes the arguments on to the native environment. For this to work the native

[1]the Default Package is the default directory all Java Classes are created in, its use is discouraged for larger projects since importing classes from the Default Package is impossible

function must be registered. This is done on the C++ "side" of the driver. More information on this can be found in the native side documentation under the JNISink section. To summarize, the Sink is the main tool for handling communication between a Java application and WISE. The Sink contains no logic operations. It simply calls functions that also exist in the native code and passes on arguments. Since data sent from Java is represented differently the next step is translating these structures to something understood by the C++ environment, this is done by the C++ class called "Marshall" and is covered in section 4.3.4 Marshall.

5.2.2 EnumFactory

The EnumFactory is a workaround aiding the Marshall class to create java type enums from native code. The reason for this is that enum classes in Java lack a public constructor, combined with not being able to pass objects by reference to native code it is impossible to update enum values that are sent from a Java application to WISE. This is solved by the EnumFactory.

5.2.3 dataTypes

In this chapter the different data types, or structures, are explained. As mentioned earlier they mirror their native counterparts as close as possible. For a user experienced in the WISE connectivity most of this chapter will be well known already. There is however a part dedicated to differences between the native and Java structures located at the end of each data structures own chapter. This chapter is divided into categories in the same way that the code is implemented, separating the different types. It is worth mentioning that Java does not allow operator overloading like C++ does and because of this those methods will not be found in the Java classes. The dataTypes package holds structures that do not share similarities with others. A Vec3 for instance has nothing in common with maps, lists or handles but is a standalone object with unique features and functions.

AttributeGroup

An attribute group is a collection of attributes held in map structures, the AttributeGroup has 16 private map attributes all together where every map has a handle as key. The attribute group is fully implemented.

AttributeGroupTemplate

The AttributeGroupTemplate holds one private attribute of each map where the key is a string. It also holds one private attribute of each list in the lists package. All

functions in the C# driver have been implemented in the Java version.

AttributeQuality

An attribute quality group holds values that are read and set by WISE to indicate the status of an object. There are five values in the form of enums set in an AttributeQuality object, these are: AttributeQuality -Limit, -Facility, -Status, -Code and -Type. The class contains a basic empty constructor, a copy constructor and set/get methods.

Blob

A Blob, or Binary Large Object, is a container holding data that cannot be handled or stored in any other way supported by WISE. The Java Blob accepts any object as argument to the function add or to the constructor. A Blob is created in the following way: the object received is transformed to a String object. An integer variable is set to the String.size(), a byte array is initiated with size corresponding to the integer value. The string objects method .getBytes is called which returns the string value as a stream of bytes and is stored in the byte array.

Deviations
The Java Blob stores data in a byte array instead of a MemoryStream object as in the C# driver. There is no exact match for a MemoryStream object in Java, if need be the structure could be changed to a java.io.ByteArrayInputStream which holds functionality closer to the MemoryStream class but also adds the need for IOExcption catching when used.

ValueUnion

The value union is a class that holds a variable of type java.lang.Object. It also contains get and set functions.

Vec3

A Vec3 object holds 3 coordinate attributes as double values. The purpose of the object is to serve as a coordinate value for objects in the WISE database. The Java Vec3 is fully implemented.

WISE_RESULT

Currently WISE_RESULT has two constructors taking either a long or integer value as argument. If the value is 1 then a WISE_OK is returned, else

WISE_ERROR is returned.

Deviations
The WISE_RESULT class is not fully implemented. It only holds the functionality described above.

WISEConstants

The WISEConstants class is simply a list of variables declared as final and static that hold a certain meaning. Most are an Integer value, some are Strings and two are longs. All constant values represented in the C# version of WISEConstants exist in the Java version.

Deviations
Apart from two constants, MessageCategoryAll and MessageCategoryProgress there should be no differences between the native and the Java implementation. The reason the two named constants differ is because they don't fit the span of an Integer. In WISE this is not a problem since it is possible to declare unsigned integers but Java does not support this.

WISEError

The WISEError class is used for error tracing. Whenever a Sink is used to create objects, events, populate attributes etc a WISE_RESULT containing a WISEError code is returned. The WISEError class holds these codes as static variables and a couple of functions to set and return values to the correct error code.

Deviations
The functions FormatMessageText(), GetErrorMessageText(), long/String GetFacility(), WISEErrorSeverity/String GetSeverity() and GetStatusCode() are not yet implemented.

WISEException

The WISEException class is the base class for the exceptions that WISE can throw.

Deviations
WISEException is not yet implemented.

WISEString

The WISEString is a wrapper class for a normal java string. The reason it has been created is to help marshalling of string values passed from Java to WISE and back. The problem lies in the fact that a java string has no set method. The value of the actual string is held in a char array but it is not possible to just populate the array with new values since a lot of other attributes in the string object need changing when the char array is changed.

Deviations
The WISEString is unique to the Java driver and has no representation in native code.

5.2.4 enums

General attributes: Enum types are used to describe a fixed set of constants like for instance the days of the week or any other structure that you know all possible values at the time of compilation. The Java enums all share a similar structure and anyone reasonably experienced in programming will recognize the structure of the classes. There are fourteen enum classes provided in the Java enums package, they are:

- AttributeQualityCode
- AttributeQualityCodeType
- AttributeQualityFacility
- AttributeQualityLimit
- AttributeQualityMode
- AttributeQualityStatus
- AttributeTimeMode
- DatabaseDistType
- DatabaseType
- DataType
- MessageCateggory
- OwnershipMode
- WISEErrorSeverity.

Deviations WISEErrorSeverity stores its value as a long not an integer because some of the values used are beyond the limits of a signed integer, the unsigned prefix doesn't exist in Java. Apart from that there are no differences, all enum classes should have the same functionality as their original native code representation. However, the constructor of a Java enum class is always private. Combined with not being able to update objects sent from Java to native code (see chapter 6.1 "Pass by value/reference") directly this makes the use of enum classes slightly more problematic than other data types. To handle this problem a class unique to the Java driver was created called EnumFactory.

5.2.5 handles

General attributes: A handle in WISE is simply an ID tag for objects, events or attributes. For instance a DatabaseHandle is an ID to a specific database in WISE used when extracting or inserting values into the database. All handles are integers and have the usual getters and setters methods. The following handles are available in the Java package handles:

- AttributeHandle
- CellHandle
- ClassHandle
- DatabaseHande
- EventHandle
- LoggerHandle
- ObjectHandle
- PlaybackHandle
- ServiceHandle
- TransactionHandle
- TriggerHandle.

Deviations None, handles are only a name wrapper for an Integer with get/set methods.

5.2.6 interfaces

IJavaWISEDriver
The interface is implemented by any Java user application that connects to the WISE connectivity, if not the driver will not be launched and cannot function.

IJavaWISEDriverSink
The Interface of the Sink declares what methods must be implemented by the Sink. It ensures that all methods are covered and the comments found here are usually more detailed that the one found in the Sink. Any class using a Sink object to communicate with WISE from Java must implement the IJavaWISEDriverSink to make sure all method calls are being handled.

IJavaWISEStringCache
Similar to the IJavaWISEDriver the StringCache interface must be implemented by any Java user application to function.

5.2.7 lists

General attributes:
All lists in the Java driver inherits from the class java.util.ArrayList. The lists are defined using Java generics in the same way maps have been implemented to make sure the structure is robust and no casting needs to be done on return values. An ArrayList behaves similar to the Vector class except that the ArrayList is unsynchronized.There are ten lists in the list package of the Java driver, their name reflecting what they store. They are:

- ByteList
- DateList
- DoubleList
- GroupList
- GroupTemplateList
- IntList
- LongList
- ObjectHandleList
- StringList
- Vec3List

5.2.8 maps

General attributes:
All maps in the Java driver inherits/extends from the HashMap class with generic input values defined to avoid any need for casting return and input values. The naming convention is NameOfValueMap for all the maps. For instance a HashMap mapping IntLists will be named IntListMap and its interface would then look like:

```
public IntListMap extends HashMap<String, IntList>{}
```

There are a total of 38 map structures in the java maps package.

5.2.9 settings

General Attributes: DriverSettings is the only class in the settings package and it is not implemented. The intended function of the DriverSettings is as the name implies to collect desired settings for the driver at initiation.

Deviations Not yet implemented due to time shortage.

5.3 Native part

The native part of the implementation does not hold any special data structures but serves more like the brain of the implementation. All logic operations take place here. The native part consists of the following classes.

5.3.1 JavaDriver

JavaDriver is the class that communicates directly with WISE. All calls to and from WISE go through JavaDriver. The JavaDriver creates the instance of the Java user application. Several JavaDrivers shall be able to connect to WISE. It is therefore important that when a JavaDriver is created it registers itself so WISE communicates with the correct JavaDriver. In the JavaDriver constructor a call to a static class DriverList is made. JavaDriver registers itself in the DriverList class and receives a signed 32 bits representation of its address. The number is later used as an id when being sent to the Java user application. When the Java instance then calls WISE the id is sent as an argument so WISE knows which driver to communicate to. When the JavaDrivers destructor is called the DriverList is called once again but this time the JavaDriver unregisters itself.

When registered, the JavaDriver tries to launch the JVM and obtain an environment pointer to the JVM. For this it uses the JVMLauncher class. If the JVM is not already created by the another JavaDriver, JVM is created and an environment pointer to JVM is sent back to the JavaDriver. The obtained environment pointer to the JVM is used for all use of JNI. In this way different JavaDrivers do not collide in the JVM.

Next the JavaDriver creates an instance of the Java user application. This made with the JNI framework. Note that the driverId is passed to the Java constructor so the user Java application can communicate with WISE.

```
//Find the class
jclass javaDriverClass = env->FindClass("ChatDriver");

// Get the methodId for the constuctor in ChatDriver.java
jmethodID methodId = jenv->GetMethodID(javaClass, "<init>", "(I)V");

// Create a new instance of ChatDriver
javaInstance = jenv->NewObject(javaClass,methodId, driverId);
```

JavaDriver has nine methods. Each of this methods calls the corresponding method in the Java user application which implements an interface containing all the methods in JavaDriver minus isCorrectDriver. To make sure that the Java user application

that is to be created really implements the corresponding interface a method called isCorrectDriver is invoked. This method makes calls to Java reading the Java user application class and trying to receive a method id for each of the methods in the interface. If there happens to be a method that is not implemented the driver aborts the creation of the Java instance.

```
jmethodID method = env->GetMethodID(javaClass, "OnUninitialize",
"()LdataTypes/WISE_RESULT;");

if(method == NULL){return false;}
```

The above code is performed for every method that needs to be implemented in the Java user application. The first argument is the jclass instance of the object.

The methods in JavaDriver are called by WISE and are passed on to a Java user application that is part of the connectivity. The following functions exist in JavaDriver.cpp:

OnInitialize

This call is the first made by WISE to a Java user application, this method should handle everything required to get the application running. If specific driver settings are required these should be read here, the Java Virtual Machine should be launched, pointers to the JVM should be registered and passed to the user application. The calls to register native methods are made here as well and an instance of the user application is created.

OnUninitialize

Called after OnCloseDatabase, this is the final call before the connectivity is shut down. Any objects, locks, and resources should be freed and the java application should terminate.

OnCreateDatabase

The template database and application databases are initiated. The database structure is predefined in the WISE Connectivity Designer Edition (CoDE) where Objects, Events and their attributes are designed.

OnCloseDatabase

The template database is closed. Called by WISE just before the OnUninitialize call.

OnAddObject

When ever an object is added to the shared backbone database, WISE issues a call to the local data managers to add this item to theirs as well. That is if there is a relation defined between the object of the backbone and the local user application database. The relation of objects are pre-defined in the WISE Connectivity Design Edition program. In the user application inside the OnAddObject function call, code needs to be written to support extracting values from the backbone database. Objects are persistant in the database.

OnRemoveObject

When objects are removed from the backbone database this call is made by WISE, it tells the user application to delete the local version of the object as well.

OnSendEvent

When an event is received by the backbone database this call is made to pass the event on to the user applications. This call is very similar to the OnAddObject function call. Events are not persistent in the database.

OnUpdateAttribute

When attributes are updated in an object this call is made by WISE to instruct local database managers to do the same. The same conditions apply to this as in the OnAddObject call.

isCorrectDriver

This call is made when the system is initialized (in OnInitialize), it makes sure the Java user application implements IJavaWISEDriver. If not the process is terminated.

5.3.2 JVMLauncher

JVMLauncher is a static class that is used for creation and termination of the JVM and also providing JavaDrivers with environment pointers to the JVM.

Launch the Java Virtual Machine

When a driver is created it asks the JVMLauncher to create and launch the JVM. If the JVM is not already created the JVMLauncher will create and launch it. For this the JVMLauncher uses its method createVirtualJavaMachine. In order to launch the

JVM the jvm.dll must me loaded. A absolute path to the jvm.dll must be provided to the JVMLauncher class. When creating the JVM, the JVMLauncher obtains a pointer to the virtual machine and stores it in the static member variable JavaVM *jvm.

Providing pointers to environment inside JVM

The JavaDriver also asks the JVMLauncher for a pointer to the JVM to be used with JNI calls. A uniqe pointer of type JNIEnv is returned to the JavaDriver. This because calls from different JavaDrivers should not collide in the JVM. The JNIEnv variable is used in all JNI usage.

Destroying the JVM

As mentioned the JVMLauncher is also responsible for destroying the JVM. The JNI function call destroyJavaVM

5.3.3 JNISink

The JNISink is the native class who handles all calls from Java to native. In the JNISink.cpp file there are global methods written in C that receives the calls from Java. All the global methods have corresponding methods in CJavaWISEDriverSink. The global methods in JNISink are written in C because JNA does not support direct communication between Java and C++. Hence a function AClass::aMethod(someArgument); can not be called from Java. Upon receiving a call issued by a java instance of CJava-WISEDriver the arguments received by the global JNISink methods are handled by the Marshall class converting them to C++ objects that WISE can handle.

Since only the JavaDriver class can communicate with WISE, the global methods in the JNI sink must obtain a pointer to a JavaDriver and then use that pointer to invoke WISE. There can be several instances of JavaDriver connected to WISE and therefore it is crucial the the C methods in JNISink get a pointer to the correct JavaDriver. This problem is solved by passing the drivers id as an argument from CJavaWISEDriverSink to JNISink. In JNISink the driver id is passed to the static class DriverList which contains pointers to all instances of JavaDrivers connected to WISE. If the driver id is valid a pointer to the corresponding JavaDriver is returned to JNISink.

```
CJavaDriver *javaDriver = DriverList::getJavaDriver(driverId);

if(javaDriver == NULL)
     return 0;
```

The JNISink class has one member method. This member method is called registerNativeMethods. The registerNativeMethods takes a pointer to the JVM environment as an argument and uses this pointer to map the all methods in the CJavaWISWDriverSink with the global methods in the JNISink so when a call is issued from Java the corresponding method in native is called.

5.3.4 Marshall

The Marshall class translates data types sent back and forth between Java and the native code so that it can be used by both sides. The name of the class is taken from what in computer science is known as marshalling and can be described as the process of transforming the memory representation of an object to another format. The Marshall class is invoked by the JavaDriver class and the JNISink class. The Marshall class uses JNI to invoke all java objects. The pointer to the JVM environment sent to the marshalling method is crucial so that the marshalling method operates on the correct Java instances.

The method of marshalling between native and Java is basically all the same.

1. Get the class object so JNI know what class it is dealing with.
2. Get the id for a method you want to invoke or a variable you want set/get
3. Call the method or set/get the variable using the id and the object representation of the java instance.

The steps are used in all marshaling methods.

This is an example of a function in the Marshall class:

```
int Marshall::getInt(JNIEnv *env, ...)
{
// get the class
jclass jcClass = env->GetObjectClass(object);

// Get the id for the variable "variableName"
jfieldID iId = env->GetFieldID(jcClass, variableName, "I");


// Get the value of the Integer object and return it
return env->GetIntField(object, iId);
}
```

First a class object is created so that JNI knows in which class to look for the attribute or method. The Java instance is sent to the native side as an jobject. All complex data types are sent as a jobject. In the second line of code the method tries to obtain the id for the variable field in the Java class mirroring the name of the argument variableName using the function GetFieldID. The "I" is the signature of the variable which corresponds to the type of object it is, in this case an integer. For more information about signatures see the JNI/JNA chapter. Finally the value from the specific object is returned through GetIntField.

There are several different marshalling functions, the example above is a method used previously in this project. After a few iterations over the implementation the GetFieldID approach to changing variables has been left for what we believe to be a more robust approach. Through the GetMethodId call it is possible to access any method in a Java class and so get/set methods are used instead of GetFieldId to change values of object. Although the result is the same the later approach is more similar to how you would normally treat attributes in a class, the example above is similar to dealing with public attributes. Some of the old calling routines might still be left in the released version because of time constrains. The example below illustrates how values are returned and set in later implementations of the Marshall class:

```
/* Returns the double value from the object */
double Marshall::getDouble(JNIEnv *env, ...)
{
// get the class
jclass jcClass = env->GetObjectClass(object);

// Get the id for the method
jmethodID iId = env->GetMethodID(jcClass, methodName, signature);

//Get the value of the Double object and return it
return env->CallDoubleMethod(object, iId);
}
```

The Marshall class can be improved. There are many methods doing more or less the same thing. For example there is a marshall method for every map type. Instead there should be a generic method that takes all map types.

5.3.5 DriverList

The DriverList is a class that keeps a pointer to every JavaDriver instance. This because there can be several JavaDrivers connected to WISE and when a java call

invokes the native JNISink, the JNISink must know which JavaDriver to use. Therefore an id is given to the JavaDriver when registering towards DriverList and later passed on to Java.

The DriverList holds the JavaDriver pointers in a std::list. DriverList has methods to register new Driver, get a JavaDriver pointer and remove a JavaDriver.

When the JavaDriver is created it registers towards Driverlist. DriverList stores a pointer to the JavaDriver in a std::list and returns the JavaDriver adress represented as an unsigned 32 bits integer.

5.4 Setup of Visual Studio

The development environment used during the project is Visual Studio 2010 and 2008. This part describes the configuration of Visual Studio needed to run and develop the JavaDriver. The path names correspond to using a 32 bit JDK installation under a Windows 7 64 bit system.

Using JNI:

- Right click the project, choose properties.
- Select the C++ tab.
- In the Additional Included Directories add the path to jni.h and jni_md.h
- Default path is c:\program files(x86)\Java\jdkx.x.x_xx\include for jni.h and c:\program files(x86)\Java\jdkx.x.x_xx\include\win32 for jni_md.h

If HINSTANCE and LoadLibrary cannot be found:
#include windows.h

Linking jvm.dll and jvm.lib to the project:
if this is not done the following error message will be received
"unresolved external symbol __imp__JNI_CreateJavaVM@12"

Adding jvm.dll:

- Right click the project, choose properties.
- Under configuration properties select the C++ tab.
- In the Additional Included Directories add the path to jvm.dll Default path is: c:\Program Files (x86)\Java\jdkx.x.x_xx\jre\bin\client
- Right click the project, choose properties.
- Under configuration properties choose the Debugging tab.
- In the Environment field add the the path to jvm.dll.

adding jvm.lib:

- Right click the project, choose properties.
- Under configuration properties chose the Linker tab.

- Click the Input tab.
- In the Additional Dependencies field add jvm.lib.

Adding MSVCR71.dll:
From bugs.sun.com: Java Bug #6509291 - "Launching java using the jvm.dll no longer works without msvcr71.dll in the system path" If an error message is displayed saying the MSVCR71.dll was not found it has to be added to the project.

- Right click the project, choose properties.
- Under configuration properties select the C++ tab.
- In the Additional Included Directories add the path to MSVCR71.dll
- Right click the project, choose properties.
- Under configuration properties choose the Debugging tab.
- In the Environment field add the the path to MSVCR71.dll

Chapter 6

Marshalling

The marshalling class Marshall is responsible for conversion between native datatypes and Java datatypes. It also contains a few methods for creating Java datatypes from native code. The class contains alot of methods and all will not be explained in this report instead a general description will be given.

The Marshall class is invoked by the JavaDriver class and the JNISink class. The Marshall class uses JNI to invoke all Java objects. A pointer to the JVM environment is sent to the marshalling methods so that they operate on the correct Java instance.

The method of marshaling between native and Java is basically all the same.

1. Get the class object so JNI knows what class it is dealing with.
2. Get the id for a method you want to invoke or a variable you want set/get
3. Call the method or set/get the variable using the id and the object representation of the java instance.

The Marashalling class is divided into four different groups of methods: Set methods, Get methods, Conversion methods and Creating methods.

6.1 Set methods

These methods populate a Java datatype with values from a native datatype. Arguments passed to the methods are generally a pointer to the Java environment, the Java datatype represented as a jobject and the native data type with the values. Return type is usually void. Here follows an example of the method setDoubleList which populates a Java data type called DoubleList which is an array of Double values. This Java instance is populated with values from a std::list< *double* >.

```
void Marshall::setDoubleList(JNIEnv *env, jobject list, ...)
{
```

```
    // Get the class
    jclass jcClass = env->GetObjectClass(list);

    // Get the id for the size method of Doublelist
    jmethodID sizeId = env->GetMethodID(jcClass, "size", "()I");

    // Get the size of the list
    int size = env->CallIntMethod(list, sizeId);

    // if the list isnt empty, empty it
    if(size != 0)
    {
     // Get the id for the clear method
     jmethodID clearId = env->GetMethodID(jcClass, "clear", "()V");

     // Invoke the clear method
     env->CallVoidMethod(list, clearId);
    }

    // Get the method id for the add method
    jmethodID addId = env->GetMethodID(jcClass, "add",
    "(Ljava/lang/Double;)Z");

    for(std::list<jdouble>::iterator iter = value.begin();
    iter != value.end(); iter++)
    {
         // Create a Double
         jobject doubleObject = Marshall::createJObjectDoubleArg
         (env, "java/lang/Double",  iter._Ptr->_Myval );

         // Add the Double to the list
         env->CallObjectMethod(list, addId, doubleObject);
    }
}
```

6.2 Get methods

These methods take a Java instance(jobject) as an argument and return a native data type with the values of the Java object. The method often invokes the get method of the Java object. Here is an example of a method that takes a Java object

Vec3(a dataype containing 3 doubles) as argument and extracts the values from it. It then creates a CWISEVec3 which is the corresponding native data type. Finally the method returns the native object holding the same values as the Java object.

```
CWISEVec3 Marshall::getVec3(JNIEnv *env, jobject object)
{
// Get the class
jclass jcClass = env->GetObjectClass(object);

// Get the ids for the variables
jmethodID getV1Id = env->GetMethodID(jcClass, "getV1", "()D");
jmethodID getV2Id = env->GetMethodID(jcClass, "getV2", "()D");
jmethodID getV3Id = env->GetMethodID(jcClass, "getV3", "()D");

// Get the double values
double v1 = env->CallDoubleMethod(object, getV1Id);
double v2 = env->CallDoubleMethod(object, getV2Id);
double v3 = env->CallDoubleMethod(object, getV3Id);

CWISEVec3 vec3(v1, v2, v3);

return vec3;
}
```

6.3 Conversion methods

These methods convert Java data types to native data types. The conversion methods differ from the get methods in such a way that they take values from an object and transfer them to a different kind of object. For example there are two methods that convert a jstring. One converts the jstring to a const char* and the other one to a std::wstring.

6.4 Creating methods

The creation methods create new jobjects and return them. They have different arguments that are used to set their value, for example createJOBjectIntegerArg creates a new jobject with a constructor accepting an integer.

Chapter 7

Discussion

During the course of this project some of the problems we have faced have stood out. It is either because they caused a lot of extra work and where hard to solve or because the reason they exist are because of differences in the programming languages. They are included here together with thoughts on improvements and possible different approaches to the design of the JavaDriver. Unless you have a lot of experience working with a system such as WISE or with JNI for that matter, it is almost certain that you somewhere along the way will find things that could be improved. The question is, as in every project, if time permits these changes.

7.1 Pass by value/reference

Something to be aware of when creating a cross platform program between Java and C++ and C#, and possibly many more than that, is that whereas C++ and C# allows pass by reference Java does not. This means that if an argument is passed from Java to native code, and that argument is updated and sent back nothing will actually be changed once returned. This is simply because what is passed to the native code is a copy of the original object, not a pointer or a reference to the actual object itself. Below are two examples of code, the first in Java and the second in C++ and an explanation of what happens:

```
Java:
public static void swap(String arg1, String arg2){
String temp = arg1;
arg1 = arg2;
arg2 = temp;
System.out.println(arg1 + " " + arg2);

}
```

```
public static void main(String[] args) {
String arg1 = "This should be last";
String arg2 = "This should be first";

swap(arg1, arg2);
System.out.println(arg1 + " " + arg2);

}

The out print inside the swap function reads:
This should be first This should be last.
The out print after the swap function is called in the main function reads:
This should be last This should be first.
The values where only updated locally.

C++:
public void swap(String& arg1, String& arg2) {
String temp = arg1;
arg1 = arg2;
arg2 = temp;
cout << arg1 + " " + arg2 << endl;
}

void main{
String arg1 = "This should be last";
String arg2 = "This should be first";

swap(arg1, arg2)
cout << arg1 + " " + arg2 << endl;
}
```

As expected the resulting output from both calls look the same: This should be first This should be last, the variables have been updated and stay changed outside the swap function call. The consequence of this is that instead of passing an argument from it's origin, through some sort of marshalling process and returning home updated, there needs to be a few more steps. This is a typical example of how the marshalling class works:

1. receive the item as a jobject.

2. get what class it belongs to.
3. get a method for changing the value.
4. create a similar object but as a C++ compatible type.
5. send the C++ type object to WISE to be updated.
6. receive the return value and through the method from *3 set this value to the jobject.
7. return the jobject to the Java user application.

7.2 Calling superclass method through JNI

When attempting to call Java functions from native code it is important to get the signatures right or the call will fail. Usually this will not be a problem since all that is required is to look in the Java code for arguments and return values, or even better use the javap -s -p < *className* > command to expose signatures. But when calling methods that a class inherits from a superclass it is not always so easy to see what the signature is. As an example, take the ByteMap in the map package of the JavaDriver. It extends HashMap¡String, Byte¿. If native code invokes the get() method a Byte is not returned but instead an Object since this is the implementation of the superclass. Because of this we must overload the get() function locally in every map so that it returns a Byte. So to conclude, when calling methods inherited from a superclass the return signature can be hard to spot.

7.3 Changes to code, effects

Many errors that occur when using registering of native methods are because of faulty signatures. Whenever a change is made to a data structure, a change of return value or renaming of a method this also impacts the signature of the native function. Extra care should be taken if big changes need to be made, in the C++ code it is very much recommended to make region comments to keep related function calls together. Any changes made on the Java side are likely to bring changes to the native code.

7.4 Relaunching the JVM

Unfortunately relaunching of the JVM, and by extension the JavaDriver seems to be unsupported by JNI at this time of writing. This affects the driver when a java application has been launched and then stopped. So far everything works as intended,

the java applications successfully connects to WISE, they run and can exchange data with the central backbone database and when the connectivity is stopped all drivers unload and the connectivity stops. Logically this would be the time to let void all references to the JVM and shut it down so that next time it's launched no references exist to the previous JVM and also we're not stuck with a running JVM that cannot be accessed.
However, the second time the connectivity is run the JVM fails to load resulting in a failed launch. WISE does not hang or crash it just aborts the launch. After a considerable amount of time spent debugging this and searching for information on the topic both from Sun and ORACLE the only thing that can be said is that relaunching the JVM is currently not working. According to documentation from Sun launching multiple JVMs from the same process using the JNI invocation interface is not supported. This is a matter of interpretation, most likely it means that you can't launch more than one at the same time but it could also refer to sequential launching and termination of the JVM. As a final attempt to get clarification on the matter the ORACLE support was contacted via e-mail but at the time of writing no response has been received.

Chapter 8

Conclusion

At the end of this project a driver application was delivered to SAAB which was able to translate the data flow between Java applications and WISE. To show a functioning, running connectivity a simple chat application was also constructed using the WISE Connectivity Design Edition. This Java based chat client could send and receive message events from a central data manager as is intended with any application connecting to WISE, and this served as the final display of functionality at a presentation at SAAB. This shows that JNI is a viable option when integrating native and Java code, we found it particularly easy when invoking Java methods using native code. The JNA framework is very simple to use for calling native code from java and works well when using a 32-bit JDK.

Have the question and purpose of the project been met?

- Question: How can integration between Java and native code be achieved using the JNI and JNA frameworks?

  Answer: JNI can be used to launch a JVM and pass an environment pointer for the native code to use. This enables native code to invoke methods written in Java code thus solving the native to Java way communication. By using JNI to register native functions they become accessible to Java code when using JNA. This solves the Java to native way of communication. The structure chosen for the JavaDriver application resulted in a working driver proving that the chosen structure works.

- Purpose: Constructing a functioning marshalling driver able to connect Java applications with an existing program written in C and C++ code.

  Result: The structure chosen for the JavaDriver using JNI and JNA for communication resulted in a functioning application. The driver was able to run a small test application that could send and receive message event to and from the

central backbone database thus proving the marshalling of data and invocation of methods worked.

8.1 Future development

Looking outside the implementation itself a thing worth investigating is the compatibility of the JavaDriver when using older JDKs. JNI was introduced in JDK 1.1 but some features where not supported such as detaching the main thread from the VM. In 1.2 this is supported however none of the versions support unloading the VM. This is from the official documentation "The Java 2 SDK still does not support VM unloading, however. DestroyJavaVM always returns an error code." [1]. Also using non official implementations of the JVM would probably be of interest. It would also be interesting to further examine the compatibility of the Java Driver when using a 64-bit JDK.

[1] from ORACLE guide to JNI, source 3

Chapter 9

Vocabulary

1. WISE - Widely Integrated Systems Environment, an information infrastructure developed by SAAB. WISE is used to integrate different applications into a common environment.
2. JNI - Java Native Interface, an interface that enables Java applications to communicate with code written in another language.
3. JNA - Java Native Access provides simplified access to native library methods, in this project it's used when calling native functions from Java.
4. Native and Native code - When Native code is mentioned or "the native side" etc the C and C++ side of the drivers code is implied.
5. JVM - Java Virtual Machine.
6. API - Application Programming Interface, a particular set of rules and specifications that software programs can follow to communicate with each other.
7. Driver - A driver is a computer program that works to translate the language of one computer application to another.
8. JavaDriver - The name of the driver produced in this project.
9. Marshalling driver - Another name for the Driver constructed in this project, same meaning as Driver.
10. Java user application - Any sort of application, written in Java code, that connect to the WISE connectivity and implements the IWISEJavaSink interface.
11. Marshalling - Conversion of data types between Java and native code.

Chapter 10

References

The following references have been used during the project. We have used the developers resources when looking up information on JNI, JNA, WISE and CORBA. Reference 1 is not referenced in the text since it has been used as programming reference literature during the implementation of the driver.

1. The Java(TM) Native Interface: Programmer's Guide and Specification Author: Sheng Liang ISBN10: 0201325772 Also available as a pdf document at: http://java.sun.com/docs/books/jni/download/jni.pdf -2011-06-13

2. JNA - Java native access source
   http://jna.java.net/ -2011-06-13

3. JNI - ORACLE documentation on JNI
   http://download.oracle.com/javase/1.4.2/docs/guide/jni/ -2011-06-13

4. WISE documentation: CoDE User's Guide CoRE User's Guide TestTool User's Guide WISE Connectivity SDK - Developer's Guide

   These documents are internal documentation belonging to SAAB and are not available to the public.

5. CORBA - CORBA official web page http://www.corba.org/