Java and C/C++ Interoperability: Java Integration to Windows Event Log

Aleksandar Bulajic, Slobodan Jovanovic
Faculty of Information Technology, Metropolitan University, 11000 Belgrade, Serbia
aleksandar.bulajic.1145@fit.edu.rs, slobodan.jovanovic@fit.edu.rs

ABSTRACT
SUN Java provides a mechanism for integration with functions coded in C/C++ or assembler, and basically offers two frameworks: Java Native Interface (JNI), and Java Native Access (JNA). In this paper, each of these two frameworks are first described, and then JNI is demonstrated with an example code, which uses Java code to write to the Windows Event Log through code written in C/C++. Event Log is a standard Windows OS feature and is used to log messages. Java usually does not use Windows Event Log to log messages, but in cases when it is necessary, this example demonstrates how it can be achieved. The beginning of this paper is dedicated to the JNI framework and data mapping, and the JNI interface pointer that allows C/C++ functions to interact with the JNI environment. Interaction in this case is bidirectional, which means that Java code is able to interact with code written in C/C++ and C/C++ code is able to interact with Java code. This paper provides a detailed description of the source code and a Visual Studio project for creating a C/C++ DLL library that is able to write to the Windows Event Log, and also offers a detailed description of source code of an Eclipse Java project that writes to the Windows Event Log through the C/C++ DLL library.

Keywords: Java, Interoperability, JNI, C, C++, Event Log

1. INTRODUCTION

Exchanging data and calling functions that are coded and compiled in different languages is an old and very well-known issue. For example, Long ago, FORTRAN functions were called from COBOL to surpass limitations in the earlier COBOL versions related to the arithmetic libraries and were missing a square root function. This issue was solved when BM released COBOL/370, which is a version of COBOL85 that includes a collection of built-in mathematical functions for numeric and date-related operations, and the square root function is a part of it. Another need for smooth communication between routines compiled in different languages is for clients to maintain access to the native libraries and operating system (OS) routines. This is as necessary under Windows as it is under UNIX or Linux or any other operating systems. Not all languages libraries are as rich with functionality as C/C++ language libraries, and these languages need to be able to call functions compiled in C/C++ language or assembler to use native libraries.

SUN Java introduced different frameworks for integrating Java application with legacy code written in C/C++:

a. Java Native Interface (JNI)
b. Java native Access (JNA)
c. K Native Interface (KNI), a subset of JNI used in the Java 2 Micro Edition

These frameworks are Java specific solutions and implementations. The JNA framework is first released in 2007. The JNA is a new approach to the integration of Java and C/C++ and code. JNA enables Java application access to native shared libraries (DLLs on Windows) without writing anything but Java code. In this case there is no need for functions written in native code. Calling native libraries, such as, for example, Windows DLLs, shared libraries, is in this case directly called from Java by using Java native method invocation.

Microsoft introduced its own set of API calls for integrating C++ and Java languages and called this framework Raw Native Interface (RNI). The primary purpose of the RNI was integration between Microsoft Visual J++ and C/C++. After SUN and Microsoft suit, Microsoft stopped developing Visual J++ in 2001, and stopped distribution and maintenance in January 2004.

This paper gives a detailed description of the source code and a Visual Studio project for creating C/C++ DLL library that is able to write to the Windows Event Log. The paper also provides a detailed description of source code of an Eclipse Java project that writes to the Windows Event Log through C/C++ DLL library. This paper describes communications between two very popular programming languages, SUN Java and C/C++, and provides fully functional code examples for Java JNI and C/C++. The Java code is developed by using Eclipse Java EE IDE for Web Developers Helios Service Release 2. The C/C++ code was developed by using Visual Studio 2003 Professional.

2. LITERATURE REVIEW

Besides references [1] and [2], provided previously by Sun Microsystems and currently by Oracle, current literature offers papers and articles that more or less detailed elaborate original papers or suggesting a new approach. Cleary et al (1998) [3] proposed using of the Interface Definition Language (IDL) for high-performance scientific applications language interoperability. Even this proposal is related to the interoperability between different languages, this is quite another approach than approach presented in this paper. The IDL is offering communication
between different languages and operating systems and interface definitions to remote access. This kind of interoperability is characteristic of the middleware. The IDL is a complex and requires time to describe stubs and skeletons that are connecting client and server applications. The Object Management Group (OMG), developed interoperability standard Common Object Request Broker Architecture (CORBA) based on the using of the IDL. The CORBA standard is today widely replaced by Web Services interoperability standard [4].

This paper is demonstrating writing of the native methods in the C/C++ language and calling/accessing these methods, which are running on the same machine, from Java executable.

Per Bother [5] describes this approach, but do not provide complete working example. He rather shows API calls and simple examples of use.

Other papers are based on the experience collected by moving C applications to Java. [6][7] Primary focus in these papers is a strategy for translating C pointers to Java references, as well as importing libraries source code from C to Java.

Even the Java language has a rich set of libraries, moving low level functions and calls to the Java can be a challenge, especially in case when such functions are controlling local hardware. The best example is embedded Java used on the mobile devices and consumer electronics. Most significant difference between presented literature and this paper, apart from presenting implemented technology, is the completeness of the example, which is not only an example, but rather fully functional and very useful application that demonstrate using of the low-level system functionality that is not supported by Java native libraries, but functionality can be fully available to the Java executable through the calls to the native C/C++ libraries.

3. JAVA NATIVE INTERFACE (JNI)

The JNI is Java interface for calling and invoking native methods written in the languages different than Java. The JNI is also a framework that provides naming conventions and enables Java to call native method and to be called by native methods. The JNI is delivered as a part of the Java Development Kit (JDK) and is used for access to the native methods and native libraries, and as an interface to the applications written in different languages like, for example, C/C++ and assembly.

The JNI can be used to:

- Integrate Java application with code written in the C/C++ and assembly languages
- Integrate JVM to an application that is written in the C/C++ language

When Java code is executed on different hardware and operating systems, it requires access to the native libraries and methods to perform I/O operations, networking, graphics, access to the sound capabilities, or any other hardware- and operating system-specific functionality. The JNI provides sets of API calls, data types, and pointers and objects that are used to call native methods and libraries as well as enable native methods and libraries to call Java code and access Java Virtual Machine.

Native methods and libraries are written in a platform’s specific native languages, for example C/C++, and are dependent of the platform specific hardware and operating system characteristics. While code written in the Java language is portable between different hardware and software platform, native code is not portable and will not execute properly on a different platform. Java code portability is provided by Java Virtual Machine (JVM). JVM is interfaced between Java code and hardware and the operating system. If JVM does not exist on a particular operating system, then it is not possible to execute Java code under that operating system.

Even in cases when there is existing JVM under particular operating system, it is necessary for Java code to use native methods and libraries to execute specific operations. The most obvious reason is that the Java libraries lack what is necessary to support a particular functionality. An example would be a specific file operation or the execution of a graphic-intensive application when in direct access to the graphic processor through assembly code, which can significantly increase performances or provide access to the registries and memory locations that is not possible from Java code.

Another example of the necessity of using JNI is when Java is loading native libraries to avoid complex inter-process communications. Executing code in the same process is more efficient and uses less operative system resources. However, there are also pitfalls using JNI, some of which can be very serious, such as:

- Risk of losing portability
- The crashing of JVM because of native code
- Difficulties in debugging and reproducing errors
- Missing of automatic garbage collection for memory resources that are allocated and managed by native method or libraries

Java data types are different than C/C++ languages data types. When the native method or library is called from the Java code, the JNI first has to convert Java data types to the native method and libraries data types and vice versa. Primitive data types, such as int, float or char, are mapped straightforward to the corresponding C/C++ types. The following table describes mapping of primitive data types and is a slightly modified table from [1] that shows the mapping of types between Java and native code.
Table 1: Java Primitive Types and JNI Mapping

<table>
<thead>
<tr>
<th>Java Language Type</th>
<th>JNI Type</th>
<th>Description</th>
<th>Type signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>jboolean</td>
<td>unsigned 8 bits</td>
<td>Z</td>
</tr>
<tr>
<td>Byte</td>
<td>Jbyte</td>
<td>signed 8 bits</td>
<td>B</td>
</tr>
<tr>
<td>Char</td>
<td>Jchar</td>
<td>unsigned 16 bits</td>
<td>C</td>
</tr>
<tr>
<td>Short</td>
<td>Jshort</td>
<td>signed 16 bits</td>
<td>S</td>
</tr>
<tr>
<td>Int</td>
<td>Jint</td>
<td>signed 32 bits</td>
<td>I</td>
</tr>
<tr>
<td>Long</td>
<td>Jlong</td>
<td>signed 64 bits</td>
<td>J</td>
</tr>
<tr>
<td>Float</td>
<td>Jfloat</td>
<td>32 bits</td>
<td>F</td>
</tr>
<tr>
<td>Double</td>
<td>Jdouble</td>
<td>64 bits</td>
<td>D</td>
</tr>
</tbody>
</table>

The Java reference types, such as classes, instances, and arrays, are passed to the native methods as opaque references. Opaque references are C pointer types to the internal data structures in the JVM and all JNI reference types have type JObject. The following picture from [1] shows reference types and the type’s hierarchy:

Table 2: Java Reference Types Mapping and Hierarchy

The jni.h Java include file contains definitions of primitive and reference JNI data types. To understand what kind of issues can arise during data types exchange between Java code and native method code, we will use a simple example of string data type. A string data type in Java is defined as a java.lang.String type and it is a reference type, which means that java.lang.String represents an Object type. In Java language, all reference types are subclasses of Object type.

In C language, a string is represented as a pointer to an array of characters terminated by a NULL character. By using a C pointer, it is possible to directly modify array contents and even reallocate or change length of the array of characters. C characters are 8 bits long and do not support Unicode by default. In Java language, String is an object. This means that internal implementation is hidden inside of the String object. Java String is immutable, which means that it cannot be changed once it is initialized. If it is necessary to modify the Java String, then the whole String structure is created with modified contents. Java String is by default 16 bits long and supports Unicode. This example illustrate differences that must be handled properly in cases where objects need to be passed to the native method and the native method should be modified and returned back to Java code object references.

To be able to manipulate Java Strings successfully, the JNI provides set of methods that native methods can call to manipulate Java Strings. These methods are called through *JNIEnv, a pointer that is passed in each call to native methods. The *JNIEnv is an object and provides pointers to native methods and libraries that are used to access JVM. The jni.h contains the definition for JNI Native Interface_ structure that is passed by the *JNIEnv pointer.

The JNI defines naming conventions when native methods are called from Java code, as well as the native methods’ names and signatures. To be able to call native methods and libraries, at least the JNI native method arguments have to be respected. Access to the native methods and libraries are two-step operations:

a. Locate and load native library
b. Locate native method that implements desired functionality

In the second step, JVM searches for the native method inside of the loaded native libraries. This step can be replaced by manually linking the native method via registering a function pointer with a class reference and method name and method descriptor. The following code snippet does a manual native method registration [1]:

```c
JNI Native Method nm;
nm.name = "g";
/* method descriptor assigned to signature field */
nm.signature = "() V";
nm.fnPtr = g_impl;
(*env)->Register Natives (env, cls, &nm, 1);
```

Manually linking native methods is more efficient than relying on a JVM search and linking. Native methods are named by following convention [2]:

a. The prefix Java_
b. A fully qualified class name
c. An underscore ("_") separator
d. A method name
e. For over loaded native methods, two underscores ("__") are followed by argument signature
The underscore ("_") separator is used as the substitute for the slash ("/") in a fully qualified class name. In the code example, the name of native method for logging events to Windows Event Log is named:

```c
Java_net_bulajic_log_SystemLogAbstract_logMsg:
JNIEXPORTvoidJNICALLJava_net_bulajic_log_SystemLogAbstract_logMsg
```

(JNIEnv *env, jobjectobj, jstringmessage, jinteventID, jinteventSeverity, jinteventCategory)

The net_bulajic_log_SystemLogAbstract isa fully qualified name of the Java class that will load and call this native method:

```c
Package net.bulajic.log;
Publicabstractclass SystemLogAbstract {};
```

The logMsg is a name of the native method. Before Java class is able to make a call to native method, it is necessary to load the native library. Loading is accomplished by using the command:

```
System.loadLibrary (<native-library-name>)
```

The following example loads "EVENTLOGJNI", that is stored in the EVENTLOGJNI.dll file:

```c
static {try {System.loadLibrary("EVENTLOGJNI");} catch (Throwable tr) {System.out.println(tr.getMessage() + " " + tr.toString());}}
```

Please note that System.loadLibrary ("EVENTLOGJNI") is inside of the static initializer to ensure that the native library is loaded only once per class. However, a native library can be loaded outside of the static initializer if necessary. Each native method has as the first argument a JNI interface pointer that is of type JNIEnv. The second argument depends on whether the native method is static or non-static. The second argument to static native methods is a reference to the Java class and if the native method is non-static, it is a reference to the object.

[2] There are two reasons why this kind of error handling is not implemented:

- Checking for all possible errors degrades native method performances
- There is often insufficient info about runtime types

The JNI does not check for programming errors in cases of passing NULL pointer or illegal argument types. [2] The huge number of C library functions does not check for programming errors and just cause runtime errors if invalid arguments are used. If the error checking was implemented in the libraries, it would lead to duplicate error checking, once in library and then again in a developer’s code. The developers are responsible for passing correct argument types to the JNI function; otherwise it can corrupt the system state or cause JVM to crash. [2] The native methods are able to rise and handle Java exceptions. Unhandled exceptions are propagated back to JVM. [2]

Most of the JNI functions return error code and throw exceptions in case of an error. The developer should always check the return value of the last JNI call to find out if an error occurred and then the developer can call a function ExceptionOccurred() to report a more detailed description of this kind of error. However, a developer should always call ExceptionOccurred () especially in cases when the native method invokes a Java method. In this case, this call ensures that an exception will not occur, and if one occurs, then a more detailed description is available.

By calling aExceptionOccurred () function, a developer can see if any exception has been raised. Also, in the case of asynchronous exceptions caused by a thread execution other than the current thread, the developer should call the function since it does not immediately affect the execution of native code. The native code can handle exceptions by [2]:

- Returning immediately and propagating exception to the Java code that called the native method
- Clear exception by calling a function ExceptionClear (), and then executing its own exception handling code.

"After an exception has been raised, the native code must first clear the exception before making other JNI calls. When there is a pending exception, the only JNI functions that are safe to call are ExceptionOccurred (), ExceptionDescribe (), and ExceptionClear (). The
ExceptionDescribe () function prints a debugging message about the pending exception.”[2]

The Java garbage collection is not able to free an object that is locally or globally referenced by native methods. The locally referenced objects are freed after the native method returns, but globally referenced objects must be explicitly freed. Locally referenced objects can be freed any time by native methods and JNI is not allowed to create extra local references. The only exceptions are references returned from native methods. Local references are only valid in the thread they are created in, and the native method must not pass it to another thread.

JVM creates a registry entry for each object reference that is sent or received from the native method. When the native method returns control to Java, the registry is deleted and garbage collection can free up referenced objects’ space. However, developers should explicitly free a local reference, especially in situations when [2]:

- A native method accesses a large Java object and creates a local reference and then executes additional computation operations before returning to a caller. Although this object is no longer used in the rest of the computation, it will prevent garbage collection to free that object,
- A native method loops through a large array of objects and creates a large number of local references. Because JVM needs to keep track of all local references, it can cause memory leaks and “out of memory” errors.

Although Java documentation has provided enough details to show that they are aware of the garbage collection and memory leak issues, the best and safest approach is to ensure that all resources used in the native methods are released and the garbage collected, since Java and JVM cannot guarantee that JVM garbage collection will free all references and release memory space.

4. CREATING C/C++ HEADER FILES

When Java is integrated to C/C++, it is necessary to create C/C++ header files from the Java classes. Header files contain forward declaration of classes, methods, functions, variables, and identifiers. Forward declaration is an incomplete declaration that is not fully defined and one of the most important purposes of a forward declaration is the separation of declarations from source code and the reusability achieved through sharing header files between different source code files. The JNI tool javah.exe, delivered as part of the Java Development Kit (JDK) and located in the <jdk_installation_folder>\bin folder, can create C/C++ header files from the Java class.

To quickly determine what parameters can be specified to javah.exe, open the Command Prompt window and type javah. Press the Enter key; the following screen shot shows the javah.exe help screen:

Fig1: javah help screen

Now go to the Java project bin folder and type:

javah –jni <full-qualified –path-to-Java-class>

and press Enter. In this case, the Java project name is EventDLL.

The following screen shot shows how the SystemLogAbstract.h header file that is used in the EventLogJNI code example is generated:

Fig 2: javah generating SystemLogAbstract.h C/C++ header file

The command:

javah -jni net.bulajic.log.SystemLogAbstract

creates

net_bulajic_log_SystemLogAbstract.h

the C/C++ header file.

The net_bulajic_log_SystemLogAbstract.h renamed in the EventLogJNI C/C++ project and the path to the jni.h is also changed to point to the include folder in the EventLogJNI project where all Java JNI include files are located. The following are the contents of the SystemLogAbstract.h header file:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include "include\jni.h"
/* Header for class net_bulajic_log_SystemLogAbstract */
 ifndef Included_net_bulajic_log_SystemLogAbstract
 define Included_net_bulajic_log_SystemLogAbstract
 ifndef __cplusplus

To quickly determine what parameters can be specified to javah.exe, open the Command Prompt window and type javah. Press the Enter key; the following screen shot shows the javah.exe help screen:

Fig1: javah help screen

Now go to the Java project bin folder and type:

javah –jni <full-qualified –path-to-Java-class>

and press Enter. In this case, the Java project name is EventDLL.

The following screen shot shows how the SystemLogAbstract.h header file that is used in the EventLogJNI code example is generated:

Fig 2: javah generating SystemLogAbstract.h C/C++ header file

The command:

javah -jni net.bulajic.log.SystemLogAbstract

creates

net_bulajic_log_SystemLogAbstract.h

the C/C++ header file.

The net_bulajic_log_SystemLogAbstract.h renamed in the EventLogJNI C/C++ project and the path to the jni.h is also changed to point to the include folder in the EventLogJNI project where all Java JNI include files are located. The following are the contents of the SystemLogAbstract.h header file:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include "include\jni.h"
/* Header for class net_bulajic_log_SystemLogAbstract */
 ifndef Included_net_bulajic_log_SystemLogAbstract
 define Included_net_bulajic_log_SystemLogAbstract
 ifndef __cplusplus

To quickly determine what parameters can be specified to javah.exe, open the Command Prompt window and type javah. Press the Enter key; the following screen shot shows the javah.exe help screen:

Fig1: javah help screen

Now go to the Java project bin folder and type:

javah –jni <full-qualified –path-to-Java-class>

and press Enter. In this case, the Java project name is EventDLL.

The following screen shot shows how the SystemLogAbstract.h header file that is used in the EventLogJNI code example is generated:

Fig 2: javah generating SystemLogAbstract.h C/C++ header file

The command:

javah -jni net.bulajic.log.SystemLogAbstract

creates

net_bulajic_log_SystemLogAbstract.h

the C/C++ header file.

The net_bulajic_log_SystemLogAbstract.h renamed in the EventLogJNI C/C++ project and the path to the jni.h is also changed to point to the include folder in the EventLogJNI project where all Java JNI include files are located. The following are the contents of the SystemLogAbstract.h header file:

/* DO NOT EDIT THIS FILE - it is machine generated */
#include "include\jni.h"
/* Header for class net_bulajic_log_SystemLogAbstract */
 ifndef Included_net_bulajic_log_SystemLogAbstract
 define Included_net_bulajic_log_SystemLogAbstract
 ifndef __cplusplus

extern "C" {
    #endif
    #undef net_bulajic_log_SystemLogAbstract_EVENTLOG_ERROR_TYPE
    #define net_bulajic_log_SystemLogAbstract_EVENTLOG_ERROR_TYPE 1L
    #undef net_bulajic_log_SystemLogAbstract_EVENTLOG_WARNING_TYPE
    #define net_bulajic_log_SystemLogAbstract_EVENTLOG_WARNING_TYPE 2L
    #undef net_bulajic_log_SystemLogAbstract_CAT_INPUT
    #define net_bulajic_log_SystemLogAbstract_CAT_INPUT 1L
    #undef net_bulajic_log_SystemLogAbstract_CAT_INTERNAL
    #define net_bulajic_log_SystemLogAbstract_CAT_INTERNAL 2L
    #undef net_bulajic_log_SystemLogAbstract_CAT_INST
    #define net_bulajic_log_SystemLogAbstract_CAT_INST 3L
    #undef net_bulajic_log_SystemLogAbstract_CAT_RES
    #define net_bulajic_log_SystemLogAbstract_CAT_RES 4L
    #undef net_bulajic_log_SystemLogAbstract_CAT_CONFIDENTIAL
    #define net_bulajic_log_SystemLogAbstract_CAT_CONFIDENTIAL 5L
    #undef net_bulajic_log_SystemLogAbstract_STR_INPUT
    #define net_bulajic_log_SystemLogAbstract_STR_INPUT 4096L
    #undef net_bulajic_log_SystemLogAbstract_STR_INTERNAL
    #define net_bulajic_log_SystemLogAbstract_STR_INTERNAL 8192L
    #undef net_bulajic_log_SystemLogAbstract_STR_INST
    #define net_bulajic_log_SystemLogAbstract_STR_INST 12288L
    #undef net_bulajic_log_SystemLogAbstract_STR_RES
    #define net_bulajic_log_SystemLogAbstract_STR_RES 16384L
    #undef net_bulajic_log_SystemLogAbstract_STR_CONFIDENTIAL
    #define net_bulajic_log_SystemLogAbstract_STR_CONFIDENTIAL 20480L
    
    #ifdef __cplusplus
    }
    #endif
    #endif
}

Figure SystemLogAbstract.h C/C++ header file

First, note that the SystemLogAbstract.h file includes a jni.h header file that contains data mapping types and all declarations, structures, and pointers related to the JVM that the native method should use to be able to access to the JVM:

#include "jni.h"

Then note that all public members of the SystemLogAbstract.java Java class are defined, as well as it is declared to call to the C/C++ native method that is defined in the SystemLogAbstract.java Java class:

/*
 * Class: net_bulajic_log_SystemLogAbstract
 * Method: logMsg
 * Signature: (Ljava/lang/String;III)V
 */
JNIEXPORT void JNICALL Java_net_bulajic_log_SystemLogAbstract_logMsg
(JNIEnv *, jobject, jstring, jint, jint, jint);

The warning at the beginning of the above header file:

/* DO NOT EDIT THIS FILE - it is machine generated */

should be respected, but this header file can be edited or manually created, although the developer should be very careful to not enter invalid information. In this particular example, we changed the line:

#include<jni.h>

to:

#include "jni.h"

The simple reason is to keep all Java JNI include files in the same location and separate them from the C/C++ include files.

5. JAVA INTEGRATION TO WINDOWS EVENT LOG

To describe how the JNI works and how it can be used, the next sections will use an example where Java code is written to the Windows Event Log. Windows Event Log is a standard log system developed by Microsoft and used by the Windows operating system to report events, messages, and errors written by operating system, libraries, and applications. The Event Log comes with predefined Application, Security, and System logs.

The Event Viewer is a Windows administration application used to monitor events recorded in the logs and also for manipulating and cleaning logs. Java developers very seldom use Windows Event Log, mostly because of portability issues and development complexity, but also because many do not know how to manipulate the Windows Event Log from Java code. However, sometimes customers who are used to Event Log logs require such functionality.

The following text describes all the necessary steps for creating a C/C++ DLL library that calls Windows
native functions to write to the Event Log, as well as Java code that calls DLL native methods and reports errors and messages. The attached code provides fully functional sources that can be directly used to report events to the Windows Event Log.

The attached code example has been built from the two different projects:

a. Java EventDLL project that has a `SystemLogAbstract.java` class that calls `logMsg` native method

b. C/C++ EventLogJNI project that creates a DLL library where the native method `logMsg` is stored.

The step in between these two steps, the creation of the C/C++ header file `SystemLogAbstract.h`, is described in details in the previous section. (See C/C++ Header Files section)

a. Java EventDLL project

The Java EventDLL project is created by using the Eclipse Java IDE version:

Fig 3: Eclipse Version

Please note that in this case the Java Enterprise Edition is not necessary. A simple Java project can do the job. The following screen shot shows the EventDLL project and project files:

Fig 4: EventDLL Eclipse Project Files

The public abstract class `SystemLogAbstract` class loads the Windows dll library:

```java
System.loadLibrary("EVENTLOGJNI");
```

and writes to the Windows Event Log through the C/C++ native method. Please note how the C/C++ native method is named:

```c
JNIXEXPORTvoidJNICALLJava_net_bulajic_log_SystemLogAbstract_logMsg
(JNIEnv *env, jobject obj, jstring message, jint eventID, jint eventSeverity, jint eventCategory)
```

A part of the native method name is the fully qualified name of the `SystemLogAbstract` class. See Section 2.2 Naming Conventions and Native Method Arguments for more details. The C/C++ native method is declared in the `SystemLogAbstract` class as:

```java
PublicnativevoidlogMsg(String msg, int eventID, int eventSeverity, int eventCategory);
```

Please note that pointer `JNIEnv *env` and `jobject obj, “this”` pointer, are not declared in the `logMsg` method. The JVM will send these two to the C/C++ native method automatically.

The `System.loadLibrary("EVENTLOGJNI")`; is discussed in more detail in Section 2.2 Naming Conventions and Native Method Arguments.

The event messages that are written to the Windows Event Log are created in the `SystemLogJNI`, a subclass of the `SystemLogAbstract` class. The method `SystemLogJNI.logMessage (...)` calls the `SystemLogAbstract.logMsg (...)` method, which is actually a call to the C/C++ native method:

```java
public void logMessage(String msg, int eventID, int severity, int cat) { try {switch (severity) {
case EVENTLOG_INFORMATION_TYPE :
    logMsg(msg, eventID, EVENTLOG_INFORMATION_TYPE, cat);
    break;

case EVENTLOG_WARNING_TYPE :
    logMsg(msg, eventID, EVENTLOG_WARNING_TYPE, cat);
    break;

case EVENTLOG_ERROR_TYPE :
    logMsg(msg, eventID, EVENTLOG_ERROR_TYPE, cat);
    break;
}
```
default:break;}}
  catch (Exception e) {System.err.println("Error in Event Log interface :logMessage(String, int, int, int):" +
e.toString());}}}

This is a pure Java code and there are no mappings to a native method argument or use of the JNI data mapping. The call to the native method and argument mapping is the job of the JNI and it is accomplished behind the scenes.

The EventLogTest.java source file is a JUnit test and contains test that uses a SystemLogJNI class to write to the Windows Event Log:

EventLogTest JUnit test class:

```java
public class EventLogTest {
    public static final int CAT_INPUT = 1;
    public static final int CAT_INTERNAL = 2;
    public static final int CAT_INST = 3;
    public static final int CAT_RES = 4;
    public static final int CAT_CONFIG = 5;
    public static final int STR_INPUT = 0x00001000;
    public static final int STR_INTERNAL = 0x00002000;
    public static final int STR_INST = 0x00003000;
    public static final int STR_RES = 0x00004000;
    public static final int STR_CONFIG = 0x00005000;
    public static final int EVENTLOG_ERROR_TYPE = 1;
    public static final int EVENTLOG_WARNING_TYPE = 2;
    public static final int EVENTLOG_INFORMATION_TYPE = 4;

    @Test
    public void testSystemLogAbstract() {
        try {
            System.out.println("testSystemLogAbstract: entering");
            SystemLogAbstract systemLogAbstract = (SystemLogJNI)
                    SystemLogAbstract.createInstance();
            systemLogAbstract.logMessage("testSystemLogAbstract message 1", STR_INPUT,
                    EVENTLOG_ERROR_TYPE, CAT_INPUT);
            systemLogAbstract.logMessage("testSystemLogAbstract message 2", STR_INTERNAL,
                    EVENTLOG_ERROR_TYPE, CAT_INTERNAL);
            systemLogAbstract.logMessage("testSystemLogAbstract message 3", STR_INST,
                    EVENTLOG_ERROR_TYPE, CAT_INST);
            systemLogAbstract.logMessage("testSystemLogAbstract message 4", STR_RES,
                    EVENTLOG_WARNING_TYPE, CAT_RES);
            systemLogAbstract.logMessage("testSystemLogAbstract message 5", STR_CONFIG,
                    EVENTLOG_INFORMATION_TYPE, CAT_CONFIG);
            assertNotNull(systemLogAbstract);
            System.out.println("testSystemLogAbstract: exiting");
        } catch (Exception e) {
            fail("Exception caught" +
e.toString());
        }
    }
}
```

The very first part of the test class is made up of constants that are used in the testSystemLogAbstract() method, declared as publicstaticfinal.

The line:

```java
SystemLogAbstract systemLogAbstract = (SystemLogJNI)
        SystemLogAbstract.createInstance();
```

instantiates systemLogAbstract as an instance of the (SystemLogJNI) class that is used in the rest of this class to call a SystemLogJNI.logMessage(...) method that will call a SystemLogAbstract.logMsg(...) method and write messages to the Windows Event Log. When the test is executed by right-clicking the mouse in the EclipseProject Explorer window on the EventLogTest test class, and then selecting RunAs and the selecting JUnit Test, it will write five messages in the Event Log named FIT:

Fig 5: FIT EventLog in the Windows 7 EventViewer

The above screen shot is the Event Log from the Windows 7 installation. In the event that the test is executed under Windows XP, it will look differently:
These five tests, consisting of one Information type event message, one Warning type event message, and three Error type messages, are easily visible in the FIT Event Log under both Windows operating systems. Each event has additional text messages attached that could explain, for example, the error’s cause and suggest further action.

How to create this text and connect it to the event message is explained in the section labeled “EventLogJNI.mc file”.

6. C/C++ EVENTLOGJNI PROJECT

The C/C++ project is a Visual Studio 2003 Win32 Console application:
a. **SystemLogAbstrac.h file**

The C/C++ header file `SystemLogAbstrac.h` has been described in detail in the previous section. (See Section 2.5 C/C++ Header Files.)

b. **EventLogJNI.mc file**

The `EventLogJNI.mc` is a message resource file that is not necessary for demonstrating the C/C++ and Java integration, but is necessary when messages written to Event Log should be formatted according to application-specific event-reporting requirements. This source file needs to be compiled by the `mc.exe` Microsoft compiler and compilation generates and updates the following files:

- `EventLogJNI.h`
- `EventLogJNI.rc`
- `MSG0001.bin`

All of the above files are used for Event Log message formatting and packed together with the native method in the same `EventLogJNI.dll` dll file.

The `EventLogJNI.mc` is a text file and should be configured before compiling by the Custom Build setup described in the following steps:

a. Right-click the mouse on the `EventLogJNI.mc` file in the Visual Studio Solution Explorer window
b. Select Properties
c. Select Custom Build Setup under Configuration Properties and edit the Command Line and Outputs values, and set them to the exact same values as shown in the following screen shot, and then click the OK button.

![Fig 11: EventLogJNI.mc Custom Build](image)

Now the custom build can be executed by right-clicking the mouse on the `EventLogJNI.mc`, and selecting Compile. The following is content from the `EventLogJNI.mc` file and this paper will not use more time to describe its syntax and purpose because it is outside of the scope of this text:

**EventLogJNI.mc File:**

```plaintext
MessageIdTypedef=DWORD ;
MessageId=0x1 Language=English INPUT
MessageId=0x2 Language=English INT
MessageId=0x3 Language=English INST
MessageId=0x4 Language=English RES
MessageId=0x5 Language=English IT620
MessageId=0x1000 SymbolicName=INPUT Language=English An input file has a formal error: %1
MessageId=0x2000 SymbolicName=INT Language=English Internal IT620 error: %1
MessageId=0x3000 SymbolicName=INST Language=English Installation problem: %1
MessageId=0x4000 SymbolicName=RES Language=English Resource problem: %1
MessageId=0x5000 SymbolicName=IT620 Language=English %1
```


The `source.cpp` file is the source file where the native method is stored. The following is the contents of this file:
and the native method for log events to the Windows Event Log file:

```c
#include<stdio.h>
#include<windows.h>
#include<malloc.h>
#include "SystemLogAbstract.h"

/**********************************************
***************************** This method is called
by the Java application through JNI to log event **
messages in the Windows Event Log
********************************-------------

 JNICALL Java_net_bulajic_log_SystemLogAbstract_logMsg
(JNIEnv *env, jobject obj, jstring message, jint eventID,
jint eventSeverity, jint eventCategory)
{
    const char *msg = env->GetStringUTFChars(message, 0);
    DWORD wEventID = (DWORD)eventID;
    WORD wEventSeverity = (WORD)eventSeverity;
    WORD wEventCategory = (WORD)eventCategory;
    HANDLE h = RegisterEventSource(NULL, "EventLogJNI");
    LPTSTR szMsg[2];
    szMsg[0] = (char *) msg;
    if (h != NULL) {
        FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER|
        FORMAT_MESSAGE_FROM_SYSTEM|
        FORMAT_MESSAGE_IGNORE_INSERTS, NULL, GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), (LPTSTR)&szMsg[1], 0, NULL);
        ReportEvent(h, wEventSeverity, wEventCategory, wEventID,
        NULL, 1, 0, (const char **)szMsg, NULL);
        LocalFree(szMsg[1]);
    }
    (VOID) DeregisterEventSource (h);
    env->ReleaseStringUTFChars(message, msg);
}
```

Figure source.cpp Native Method File

Section 2.2 Naming Conventions and Native Method Arguments contains a description of the native method naming convention and arguments. Here just the most important information is repeated:

```c
JNIEXPORT void JNICALL Java_net_bulajic_log_SystemLogAbstract_logMsg
(JNIEnv *env, jobject obj, jstring message, jint eventID, jint eventSeverity, jint eventCategory)
```

The first argument JNIEnv *env is the interface pointer, the pointer to JVM, and the native method uses this pointer to access JVM.

The second argument jobject obj is this pointer.

The third argument jstring message and the rest of the arguments jint eventID, jint eventSeverity, and jint eventCategory are used to describe the event message, event ID, and event severity and event category, and are the parameters required to call the ReportEvent function and report events to the Windows Event Log.

What is most interesting in this code from the viewpoint of developers is the integration between Java and native method. Even the Java JNI mapped Java types to the native method. While primitive types, such as jint, can be used directly, string type jstring cannot be used directly by native code and direct access to jstring would most likely crash JVM. The following code illustrates a possible JVM crash example:

```c
printf("%s", message);
```

The string in Java is an object type while the string in the C/C++ native method is a pointer to continuous character array. The string in Java is immutable, which means that it cannot be changed directly. If changes are necessary, then Java makes a new string copy and stores the changes. String in the C/C++ native method can be changed directly.

The Java JNI provides functions that are used to access Java strings and convert them to C/C++ native method strings. The following code line shows Java String conversion to C/C++ character array:

```c
const char *msg = env->GetStringUTFChars(message, 0);
```

In the above code, it is important to note that the JNIEnv *env and GetStringUTFChars functions are used to convert to C/C++ char *msg.

The following table, copied from [2], contains a summary of JNI string functions:

<table>
<thead>
<tr>
<th>JNI Function</th>
<th>Description</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStringChars</td>
<td>Obtains or releases a pointer to the contents of a string in Unicode format. May return a copy of the string.</td>
<td>JDK1.1</td>
</tr>
<tr>
<td>ReleaseStringChars</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetStringUTFChars</td>
<td>Obtains or releases a pointer to the contents of a string in UTF-8 format. May return a copy of the string.</td>
<td>JDK1.1</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>JDK 1.1</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>GetStringLength</td>
<td>Returns the number of Unicode characters in the string.</td>
<td></td>
</tr>
<tr>
<td>GetStringUTFLength</td>
<td>Returns the number of bytes needed (not including the trailing 0) to represent a string in the UTF-8 format.</td>
<td></td>
</tr>
<tr>
<td>NewString</td>
<td>Creates a java.lang.String instance that contains the same sequence of characters as the given Unicode C string.</td>
<td>JDK 1.1</td>
</tr>
<tr>
<td>NewStringUTF</td>
<td>Creates a java.lang.String instance that contains the same sequence of characters as the given UTF-8 encoded C string.</td>
<td>JDK 1.1</td>
</tr>
<tr>
<td>GetStringCritical, ReleaseStringCritical</td>
<td>Obtains a pointer to the contents of a string in Unicode format. May return a copy of the string. Native code must not block between a pair of Get/ReleaseStringCritical calls.</td>
<td>Java 2 SDK 1.2</td>
</tr>
<tr>
<td>GetStringRegion, SetStringRegion</td>
<td>Copies the contents of a string to or from a preallocated C buffer in the Unicode format.</td>
<td>Java 2 SDK 1.2</td>
</tr>
<tr>
<td>GetStringUTFRegion, SetStringUTFRegion</td>
<td>Copies the content of a string to or from a pre-allocated C buffer in the UTF-8 format.</td>
<td>Java 2 SDK 1.2</td>
</tr>
</tbody>
</table>

The Java String function \texttt{constchar *msg = env-GetStringUTFChars (message,0)} converts Java String to a C/C++ pointer to character array, and then the rest of the code directly converts jint to DWORD/C/C++ types:

\begin{verbatim}
DWORDwEventID = (DWORD) eventID;
WORDwEventSeverity = (WORD) eventSeverity;
WORDwEventCategory = (WORD) eventCategory;
\end{verbatim}

Although it is a pure C/C++ native method system call, it is interesting to mention and explain a RegisterEventSource function:

\begin{verbatim}
HANDLEh  = RegisterEventSource(NULL, "EventLogJNI");
\end{verbatim}

This C++ native method system function returns HANDLEh to Windows Event Log. This handle is used in the function ReportEvent to write events to the correct Event Log:

\begin{verbatim}
ReportEvent (h, wEventSeverity, wEventCategory, wEventID, NULL, 1, 0, (constchar **)szMsg, NULL);
\end{verbatim}

In case that EventLogJNI does not exist, then the event will be written in the Event Log Application log. The attached code in the zip file includes an EventLogJNI.reg file in the code/reg folder that contains entries for Windows registry database. By double clicking on the reg file, the Windows registry will be updated and new FIT log will be created.

Before executing the registry update, please edit the path to CategoryMessageFile and the EventMessageFile to point to the place where you copied the EventLogJNI.dll file. Please note that the path should contain the backslash character “\”. This character is a Windows escape character and should be preceded by the escape character “\\”. That is why in the path exist two backslash “\\" characters.

Please note also that the native code releases allocated memory.

\begin{verbatim}
Local Free (szMsg[1]); (VOID) DeregisterEventSource (h);
\end{verbatim}

7. CONCLUSION

The JNI is a Java specific framework for integrating functions written in the C/C++ or assembler languages to Java code. This framework enables Java programmers to access the operating system specific system functions and native libraries and use functionalities that are not provided by standard Java libraries. Another reason for using the JNI is to improve performance. For example, mathematic calculations or direct access to the graphical processor can significantly improve performance in cases when the Java code is replaced by system functions and native methods calls.

Calling system functions and native methods through JNI is not error-free and there are important disadvantages to consider. The most unpleasant issues are:

- Risk of losing portability
- Crashing of JVM because of native code errors
- Difficulties to debug or reproduce errors
- Missing of automatic garbage collection for memory resources allocated and managed by native method or libraries
Memory leaks

The JVM can crash because of an error in the native code. An example of what would cause this error is an attempt to access the Java string directly from C/C++ code without using JNI string functions first to transform the Java string to a C/C++ character array. Correcting memory leaks can be daunting and an expensive and time-consuming task. However, the JNI is an important part of JVM and Java standards require that JNI exists in each particular implementation of JVM.

This paper provides a detailed description of the source code and a Visual Studio project for creating a C/C++ DLL library that is able to write to the Windows Event Log. The paper also provides a detailed description of the source code for an Eclipse Java project that writes to the Windows Event Log through the C/C++ DLL library. This paper further describes communications between two very popular programming languages—SUN Java and C/C++—and provides fully functional code examples for Java JNI and C/C++. The Java code is developed by using Eclipse Java EE IDE for Web Developers Helios Service Release 2. The C/C++ code is developed by using Visual Studio 2003 Professional.

The JNI framework is especially important in programming for PDA devices, mobile phones, and consumer electronics. In many cases, accessing a device's native library is the only way to get full control of the device's hardware and gain access to its rich set of built-in functionality.

REFERENCES


[7] Martin, Johannes (2001), “Strategies for Migration from C to Java”, University of Victoria, Department of Computer Science, Victoria, BC, Canada