Java/JMDL communication with MDL applications

By Stanislav Sumbera

[Editor' Note: The arrival of MicroStation V8 and its support for Microsoft Visual Basic for Applications opens an entirely new set of dual-language communication issues beyond the scope of this article. Support for JMDL and MDL continues with the release of MicroStation V8.]

The C-style MDL language has been available to MicroStation developers for several years. Gigabytes of codes have been written worldwide in the form of utility applications (.ma files), shared libraries (.msl files), static libraries (.ml files) and resource files (.r) for graphic interfaces, command tables and many other functions. More importantly, this MDL code has been maintained, tested and refined over thousands of hours of use. There is also considerable native-code DLL written to perform time-critical tasks in MicroStation. Such code can be easily incorporated into MDL programs via the **dlm** set of functions.

On the other side, there is the new, modern, easy to learn, platform-independent and object-oriented language Java and its JMDL extension in MicroStation/J. Java enables rapid application development with many advantages for MicroStation programmers (pure Java classes from the third-party vendors, safety in memory management, scalability). However, Java or JMDL applications seem to have some deficiencies. For instance, they do not look and feel like MDL applications. The standard Java class library does not support some platform-dependent features rarely required by MicroStation, and their performance is low in situations where time-consuming operations need to be optimized.

MDL programmers had many questions when Java was first integrated into MicroStation/J: "How do we communicate between MDL and Java applications?" "How do we access low-level code from the Java environment?" and "How do we recall the Java code from MDL?"

The Java development kit delivered with MicroStation/J contains documented and supported interface specifications that allow programs written in other languages to be called up from the Java code that runs within the Java Virtual Machine. Programs in other languages are able to call up methods and access objects written in the Java language using the Java Native Interface (JNI).

Both MDL and Java have full, bi-directional access to the native code. But it would not be sufficient to make the communication easy if the virtual machines for interpreting these two different codes would run in separate process space. Fortunately, MDL runtime (**ustation.dll**) and Java/JMDL runtime (**javai.dll**) are mapped and run within the process address space of MicroStation (**ustation.exe**). That is why the DLL native code attached to the Java application and then to the MDL application will receive only one **DLL_PROCESS_ATTACH** message from the first one. Thus, we can speak about an in-process communication between MDL and Java/JMDL (Figure 1.)

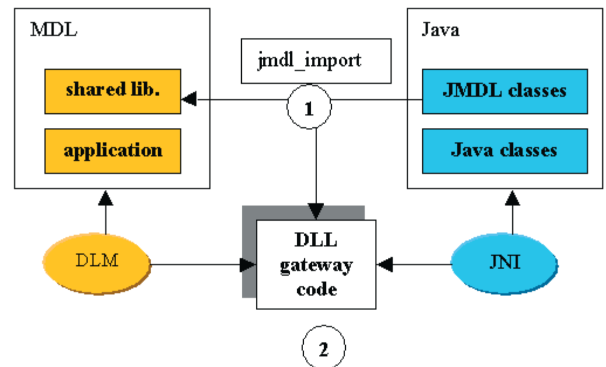


Figure 1. Types of communication between Java/JMDL and MDL

Use the jmdl_import keyword to call MDL or DLL

The easiest and most straightforward way of invoking a shared library written either in the MDL or a native code from a JMDL code is to use a mechanism that is very similar to the J/Direct technology. The **jmdl_import** keyword is a part of a native function declaration in which we specify the MDL or DLL library name and their exported functions which we are going to invoke. The jmdl method succeeding the **jmdl_import** statement must be declared as *static*.

An example of a JMDL method is **mdl_printPromp** which refers to a mdl shared library (**mdlPrompt**) and its function (**printPrompt**) :

```
jmdl_import
(mdl = "mdlPrompt", name = printPrompt)
public static native void
mdl_printPrompt(byte     *showText);
```

Similarly, in order to invoke a native function from DLL, you can use the dll = <name of the native code library> instead of mdl =...:

```
jmdl_import
(dll = "dllPrompt", name = printPrompt)
public static native void
dll_printPrompt(byte     *showText);
```

Using Java Native Interface to interact with MDL

To use the JNI for accessing the MDL code, we need an intermediate native layer for passing calls from Java to MDL and vice versa. This DLL "gateway" consists of the JNI interface to interact with Java on one side, and, on the other side, there must be an interface to communicate with the MDL code in a way of a dynamic link module.

For better understanding of the following codes and description of technology, Figure 2 is a diagram of calling between MDL and Java, with chapter numbers for particular areas of interaction.

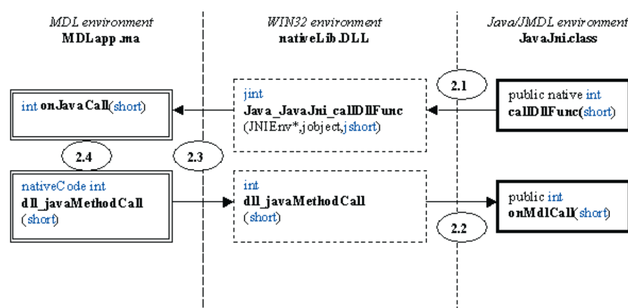


Figure 2. Function calls between Java and MDL.

Using Java Native Interface to invoke the DLL function

The JNI is quite a rich application interface that enables the interaction between the native code and Java. On the Win32 platform, such a native code is represented by a regular DLL with exported functions, which we are going to invoke. Through the JNI it is possible to:

- Call Java methods
- Call native methods
- Operate with a Java object
- Process exceptions
- Load classes and obtain their information

Suppose there is a class which will invoke a function in a native library called **nativeLib.dll**. First, Java must load the DLL library into memory and link to it by calling the **System.loadLibrary**. The only parameter of the method is a name of the desired dynamic library. The specified DLL must be stored in one of the system path directories or in the same directory containing a Java class file. The Java virtual machine will automatically append the appropriate extension according to the given platform, so you do not need to write "**.dll**" extension there. Here's an example of the method:

```
static {
// loading the library nativeLib.dll
System.loadLibrary("nativeLib");
}// declaration of a native function
public native void callDllFunc(short limit);
```

Then to implement the Java method in C++, you need to create a "Java to C++" mapping header file with function prototype of given method. The task could be done simply by a standard Java Development Kit (JDK) utility, **javah.exe** with the **-jni** command parameter and the name of the class that will be processed. The result is a declaration of native function placed in the header file named as the processed class. For instance, we can enter a command **javah - jni JavaJni**, and the header file "**JavaJni.h**" will contain a prototype for the function **callDllFunc** in the C++ code :

```
#include <jni.h>
extern "C" {
// declartion of native function
JNIEXPORT jint JNICALL Java_JavaJni_callDllFunc
(JNIEnv *, jobject, jshort);
}
```

The header file begins with an *include* directive for **jni.h.** In **Jni.h** there are, among other things, the data types and function prototypes of the JNI defined. The **JNIEXPORT** macro is defined in **jni_md.h** for the Win32 platform as **__declspec(dllexport)**. Such a function declaration means that you do not need an additional module-definition (.DEF) file of the exported functions. The JNICALL is expanded into **__stdcall**, which expresses the definition of the calling convention for Win32 API functions. The JNI imposes a naming style on the native methods through which the Java Virtual Machine links the Java calls to the native methods. So the name of the native language function that implements a native method consists of the Java keyword, followed by a package name (here we have a default package, hence, in our example, this part is omitted), then the class name **JavaJni** and, finally, the name of the native method **callDllFunc**. Between each name part is an underscore "_" separator. A question may come up about the number

of parameters in our **Java_JavaJni_callDllFunc** prototype where there are three arguments in total instead of only one. These two additional arguments are passed from Java to all native functions using the JNI. The first of them is the Java Environment interface pointer containing all the necessary function pointers for the native processing of parameters and objects passed from Java. The second one is, in this case, a reference to the current instance of the **JavaJni** object—an equivalent to "**this**" in Java. These two parameters make a gateway to call back the Java method but only in the current context of calling. The third one is a corresponding argument to **short** type passed from the Java code.

The example code **nativeLib.cpp** given below, in which we implement the Java native function with an only short parameter, is written in C ++ and displays the parameter value using the Win32 API:

```
#include "stdafx.h"
// header file generated by the javah utility
#include "JavaJni.h"
BOOL APIENTRY DllMain
(HANDLE hModule, DWORD
dwReason,LPVOID lpReserved){
   return TRUE; // dll entry point
}
JNIEXPORT jint JNICALL Java_JavaJni_callDllFunc
  (JNIEnv * env,jobject  jthis, jshort limit){
     char    message[255];
     sprintf(message,"Native Dll recieved value
%d",limit);
// displaying a message
     ::MessageBox(NULL,message,"nativeLib",MB_OK);
//  call to mdl will be explained soon
return (jint) dll_callMdlFunc(limit);
}
```

Calling the Java method from the native code

To invoke the Java non-static method from a native code, we need to do a little bit more of programming than before. First of all, we need to design a **JavaJni** class to implement the calling from the native code and compile it with **javac** utility to have a **JavaJni.class**. The class displays a simple dialog with a label item where will be reflected passed parameters from a native function :

```
import java.awt.*;
public class JavaJni extends Frame{
   Label   label    =
new Label ("Label",Label.CENTER);
   static JavaJni          javaObj   =
new JavaJni("Java dialog");
public JavaJni(String title){
// constructor to display a dialog box
    super(title);
    resize(150,100);
    setBackground(SystemColor.control);
    add(label); show();
}

public int onMdlCall(short limit){
// receiver  method of a MDL call via DLL
     label.setText
("Native call, value  = " +limit);
     return 1;
     }

public static void main(String[] args)   {
javaObj.show();
   }
}
```

Now you are prepared to implement calling from a native method. The necessary steps are:

1. Get a Java Virtual Machine (JVM) pointer.
The **JNI_GetCreatedJavaVMs** function from the JNI library will successfully process your request if the JVM has already been started. The returned pointer will be used in the second task.

2. Get an interface pointer.
To obtain a valid interface pointer, a member function of the JVM pointer called **AttachCurrentThread** needs to be invoked. The function attaches the current thread to the JVM and returns the JNI interface. If the thread has already been attached, no operation is performed and only the valid interface pointer is returned.

3. Retrieve a class reference of the called method.
The JNI interface pointer received in the previous task includes all the necessary functions to operate with the JNI. One of them is **FindClass** which will realize the current task. The returned value is a representation of the searched class.

4. Retrieve a method and object identifier.
A method and object identifiers can be obtained from within a class. The identifier is requested when calling up the method. The methods of the JNI called **GetMethodID** and **GetStaticFieldID** accept the class object we received in the previous step, the method or object name of the Java code and its signature as parameters. To find out what signature is used for the given method, we need to run **javap** utility from JDK as follows: **javap -s JavaJni.** Then, in the commentary below the method, we will find its signature. For instance, the public native int **onMdlCall**(short) method has the "**(S)I**" signature. Similarly, we may find the signature for a static **javaObj** object - "**LJavaJni;**"

5. Obtain a requested instance of an object.
Because we designed the **JavaJni** object as a static one, we can use the **GetStaticObjectField** function to get a reference to it. The input parameters are represented by the class and object identifier. The returned value is represented by a reference to the static **javaObj** object.

6. Invoke the non-static method.
Now you know all the necessary gadgets to perform a call. There is a special function for each type of the calling method in Java. In this example, we have designed a method returning the integer type, hence the **CallIntMethod** function is the right one. The input arguments must be represented by the object reference from the previous step, then the method identifier and the parameter expected by the Java method, in our case represented by the **jshort** value.

Now add into the file **nativeLib.cpp** used before as an example of calling the Java method **onMdlCall** via native function **dll_javaMethodCall** :

```
int  JNICALL dll_javaMethodCall(short limit)
{
 JavaVM *jvm; /* denotes a Java VM */
 JNIEnv *env; /* point to java interface */
 jsize  nVMs;  /* number of created JVM, should
be 1  */

/* 1.*/ jint res =
JNI_GetCreatedJavaVMs(&jvm,(jsize)1,&nVMs);
/* 2.*/ res        =
jvm->AttachCurrentThread(&env,NULL);
/* 3.*/ jclass    cls   =
 env->FindClass("JavaJni");
/* 4.*/ jmethodID mid   = env->GetMethodID(cls,
"onMdlCall", "(S)I");
/* 4.*/ jfieldID  fid   =
 env->GetStaticFieldID(cls, "javaObj",
"LJavaJni;");
/* 5.*/ jobject   obj   =
 env->GetStaticObjectField(cls, fid);
/* 6.*/ return =
 env->CallIntMethod(obj, mid,limit);
}
```

The full interaction between Java and the native code has been achieved. You can call up native functions from the Java environment as well as Java methods from the native code. Now the task is to create a bridge in the native code for interaction with the MDL code.

Designing a native code to interact with MDL

A technology for communicating between the MDL code and DLL is well-known and is described in the MicroStation help texts. Functions from the DLL code can be imported into the MDL code by declaring a function with the nativeCode specification. The native code is able to invoke MDL as well, via the documented **dlmSystem_callMdlFunction** function.

Let's create a C++ file *"***MDLLib.cpp***"* to operate with MDL. A first function **dll_setMdlFuncCall** accepts the mdl function offset as a parameter, gets a MDL task descriptor and saves them into a global variable for later use. The second function **dll_callMdlFunc** will make use of the saved variables and invoke a particular offset function of the MDL task with one argument of the **short** type.

The MDLLib.cpp file :

```
#include "stdafx.h"
extern "C"{
// mdl communication include
#define      winNT
#include       "mssystem.fdf"
#include       "dlmsys.fdf"
}

// Pointer to Call back the MDL module
mdlDesc               *mdlDescP = NULL;
// offset   to MDL function
MdlFunctionP      funcOffset = 0L;


int __stdcall
dll_setMdlFuncCall(MdlFunctionP mdlFunc){
  if ((funcOffset = mdlFunc) && (mdlDescP =
mdlSystem_getCurrMdlDesc()))
      return true;
  return false;
}

// Invoker of  a MDL function
int __stdcall
dll_callMdlFunc(short limit){
if ((funcOffset) && (mdlDescP))
      return  dlmSystem_callMdlFunction
      (mdlDescP, funcOffset,limit);
      return false;
}
```

Finally, we need to create a module-definition file (.DEF extension) to provide the linker with information about exported functions about the **nativeLib.dll** to be linked.

The **Export.def** file exports two functions. The first one has just been described above, the second one is the Java method invoker function designed earlier in the article. Note that the previous JNI function of **Java_JNIgo_callDllFunc** is exported automatically.

```
LIBRARY      "nativeLib"
DESCRIPTION  'JAVA - MDL gateway library'

EXPORTS
    dll_setMdlFuncCall   @1    ; MDL function call back saver
    dll_javaMethodCall   @2    ; Java method invoker
```

Designing a MDL application to interact with DLL

Finally, design a MDL application to interact with the finished **nativeLib.dll**. First map the exported DLL function to the MDL import functions conventions. For native function references in MDL code, the linker **mlink.exe** needs the Dynamic Link Specification object file (.DLS extension for source and .DLO for the precompiled object). This object is treated as a static library and is linked with an MDL program. The MDL virtual machine automatically loads the specified DLL defined in DLO file when MDL application is loaded. Compare it to Java where the DLL must be explicitly loaded. The **import.dls** file reflects the just-created file **Export.def**:

```
%Version 0x700
%ModuleName nativeLib
%Functions
dll_setMdlFuncCall
dll_javaMethodCall
%EndFunctions
%End
```

The steps to design MDL code so it can call DLL native functions and also be called from the DLL are:

1. Add the native declarations to the global part of the **mdl** module corresponding to the function declarations in the **Import.dls** file.

```
nativeCode int dll_setMdlFuncCall(ULong);
nativeCode int dll_javaMethodCall(short);
```

2. Create a function **mdlOnJavaCall** to be called from the native DLL library **nativeLib.dll** by the function **Java_JavaJni_callDllFunc**, which has been called by the Java class **JavaJni** from the method **callDllFunc**.

```
int mdlOnJavaCall ( short limit){
     char message[255];
     sprintf(message,
"got request on %d,limit);
     mdlDialog_openMessageBox
   (DIALOGID_MsgBoxOK,message,

   DIALOGID_MediumInfoBox);
     // do some interations...
     return TRUE;
}
```

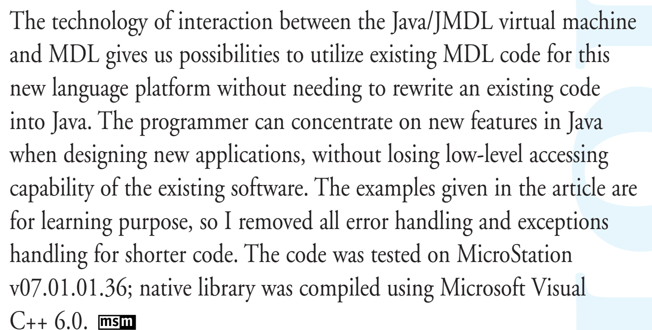
3. Set the function hook offset of **mdlOnJavaCall** for the native code in the main entry point of a MDL.

```
int main(int argc, char   *argv[]){
 //  set a hook for DLL
 dll_setMdlFuncCall((ULong) mdlOnJavaCall );
   return TRUE;
}
  break;
```

4. Finally, go the other way and make a call to the **nativeLib.dll** on some user input event, or whenever you like to call up the native function **dll_javaMethodCall**, which will invoke Java method **onMdlCall** from the class **JavaJni**.

```
case DITEM_MESSAGE_BUTTON:
// call a Java method via dll
dll_javaMethodCall(i);  break;
```

Final words



The technology of interaction between the Java/JMDL virtual machine and MDL gives us possibilities to utilize existing MDL code for this new language platform without needing to rewrite an existing code into Java. The programmer can concentrate on new features in Java when designing new applications, without losing low-level accessing capability of the existing software. The examples given in the article are for learning purpose, so I removed all error handling and exceptions handling for shorter code. The code was tested on MicroStation v07.01.01.36; native library was compiled using Microsoft Visual C++ 6.0. msm

Stanislav Sumbera is a MicroStation software developer for the Berit group (www.berit.com) specializing in raster data and seamless map visualization. The programming code in this article is offered to the MicroStation community "as is." You may contact Stanislav Sumbera at stanislav.sumbera@berit.cz.