OpenVMS Technical Journal V12

Calling OpenVMS native routines from Java

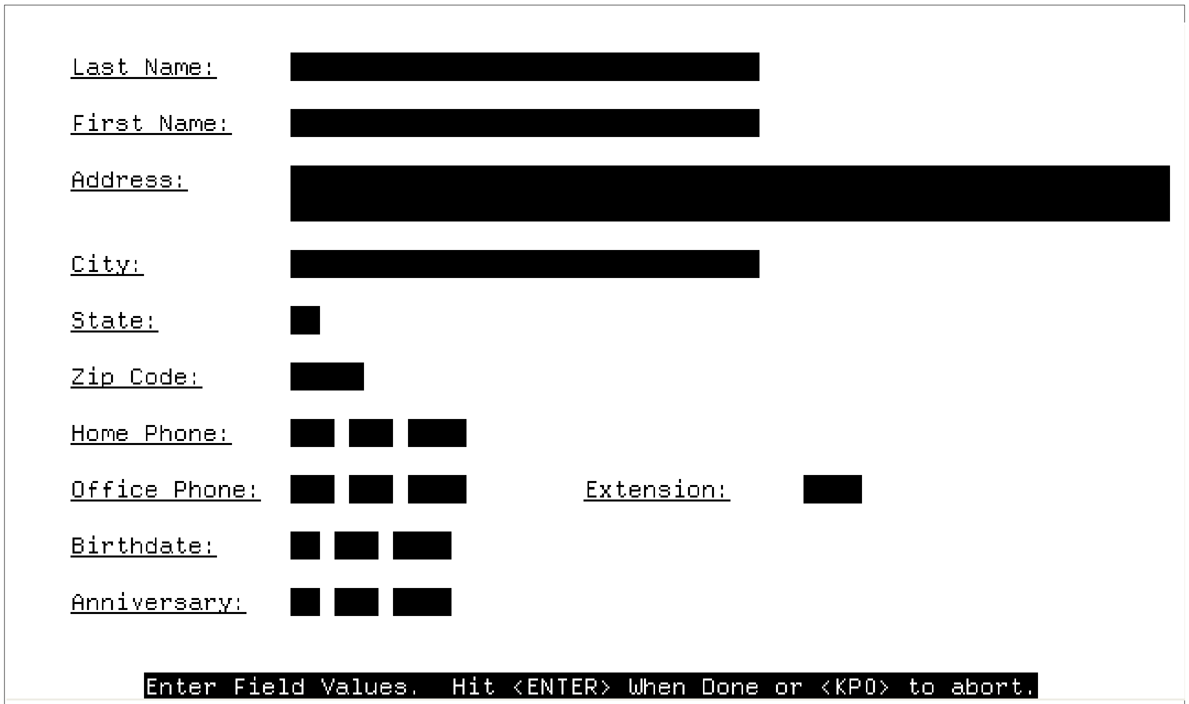
Tim E. Sneddon

Overview

Developing a Java application that calls native routines can be tedious. Along with the native routines and the Java application, there is also the need to develop a Java Native Interface (JNI) layer. This article presents a collection of tools and libraries that remove the need to develop the JNI layer—allowing Java to call native routines and manipulate data structures directly.

The “legacy” application

For this tutorial, a simple native application written in PL/I has been selected as the target of an “upgrade”. The application is an address/phone book program, called PHONE[1] (not to be confused with the OpenVMS utility of the same name). It uses the SMG$ API to draw the screen and an RMS index file to store each phone book record. Figure 1 shows the record entry screen.



```
Last Name:
First Name:
Address:
City:
State:
Zip Code:
Home Phone:
Office Phone:                    Extension:
Birthdate:
Anniversary:

     Enter Field Values.  Hit <ENTER> When Done or <KP0> to abort.
```

Figure 1. PHONE data entry screen

The intention is to be able to access the phone book data file from Java so a new GUI can be built. By using Java, the options for future development are expanded to allow development of either a Swing-based (GUI) interface or a web-based front end (using Tomcat). However, the most important requirement is to allow the original application to continue being used with little or no change.

Setting up the run-time library

In order for Java to call a native routine it must be in a shareable image. The first step is to identify the modules that are necessary for accessing the data file. In this case, it is simple: the DATABASE.PLI module contains all routines related to file access. From this module, the information about its entry points are as follows:

- OPEN_PHONEBOOK—This routine opens the phone book datafile.
- CLOSE_PHONEBOOK—This routine closes the phone book datafile.

[1] For those with a PL/I compiler this program can be found in the examples directory for the Kednos VAX and Alpha PL/I compilers.

- GET_A_RECORD—This routine reads a record with the specified last name and returns a pointer to that record.
- GET_A_MONTH_RECORD—This routine reads a record for an event that occurs in the specified month. It returns a pointer to the record.
- GET_A_DATE_RECORD—This routine reads a record for an event that occurs on the specified date. It returns a pointer to the record.
- WRITE_A_RECORD—This routine writes the specified record into the phone book database.
- DELETE_A_RECORD—This routine deletes the specified record from the phone book database.

```
$ PLI PHONE
$ PLI DATABASE
$ PLI SCREEN
$ LINK PHONE,DATABASE,SCREEN
$ IF (F$SEARCH(“PHONE.DAT”) .EQS. “”) THEN -
$ CREATE/FDL=PHONE.DAT
```

Example 1. Original PHONE build procedure

This information is then used to build a shareable image symbol vector that details the entry points to the linker. Example 1 shows the original build procedure for the self-contained application, which is one single executable. Example 2 shows the new build procedure (changed sections are highlighted in yellow).

```
$ PLI PHONE
$ PLI DATABASE
$ PLI SCREEN
$ LINK/SHARE=DATABASE_RTL.EXE DATABASE,SYS$INPUT/OPTION
SYMBOL_VECTOR = ( -
 OPEN_PHONEBOOK = PROCEDURE, -
 CLOSE_PHONEBOOK = PROCEDURE, -
 GET_A_RECORD = PROCEDURE, -
 GET_A_MONTH_RECORD = PROCEDURE, -
 GET_A_DATE_RECORD = PROCEDURE, -
 WRITE_A_RECORD = PROCEDURE, -
 DELETE_A_RECORD = PROCEDURE -
 )
$DEFINE/NOLOG DATABASE_RTL SYS$DISK:[]DATABASE_RTL
$ LINK PHONE,SCREEN,SYS$INPUT/OPTION
DATABASE_RTL/SHARE
$ IF (F$SEARCH(“PHONE.DAT”) .EQS. “”) THEN -
$ CREATE/FDL=PHONE.DAT
```

Example 2. New PHONE build procedure (BUILD_PHONE.COM)

It is now possible to rebuild the application with the user interface in PHONE.EXE and the phone book file interface in DATABASE_RTL.EXE.

Building the Java layer

Once the RTL has been set up and shown to work correctly with the existing data file, the next step is to generate the Java definitions of the entry points and record structures. This is done using the Java back end for the SDL compiler.

SDL stands for Structure Definition Language. It is a language and compiler for taking language-independent record, constant, and entry-point definitions and generating language-specific include/header files.

Normally this header file would have to be built by hand. However, for PL/I developers there is a /SDL qualifier on the PL/I compiler that allows PL/I source modules to be translated to SDL. Example 3 demonstrates using the PL/I SDL generator and its output.

The newly generated SDL source file can then be processed by the Java back end to generate a Java module that defines the same information in a way that can be used by the J2VMS package. A severely trimmed example of the Java class generated from DATABASEDEF.SDL (shown in Example 2) can be seen in Example 4.

```
$ PLI/SDL=(MODULE=DATABASEDEF,OUTPUT=DATABASEDEF.SDL) DATABASE.PLI
$ TYPE DATABASEDEF.SDL
module DATABASEDEF;
entry "DELETE_A_RECORD" parameter(
 character length 65 reference
 ) returns boolean;
entry "WRITE_A_RECORD" parameter(
 pointer value
 );
entry "GET_A_DATE_RECORD" parameter(
 character length 5 reference,
 boolean value,
 boolean value
 ) returns pointer;
 ⋮
aggregate "ENTRY" union;
 "ENTRY_STRING" character length 266;
 "FIELDS" structure;
 "NAME" union;
 "FULL_NAME" character length 65;
 "PIECES" structure;
 "LAST" character length 32;
 "SPACE" character length 1;
 "FIRST" character length 32;
 end "PIECES";
 end "NAME";
 "ADDRESS1" character length 60;
 "ADDRESS2" character length 60;
 "CITY" character length 32;
 "STATE" character length 2;
 ⋮
end_module DATABASEDEF;
```

Example 3. Generating the SDL definitions from PL/I source (DATABASEDEF.SDL)

The two SDLJAVA_* logicals are required to give the Java back end information necessary for building the Java source file correctly. The SDLJAVA_PACKAGE logical details the Java package name giving the source module a place in the class hierarchy. SDLJAVA_LIBNAME defines the name of the run-time library (as required by LIB$FIND_IMAGE_SYMBOL) where J2VMS will locate the routines.

Incidentally, the steps up to now make it possible to use the SDL definitions with a host of other languages including C, Fortran, Ada, and BLISS—so not only can you expose your code to Java, but it is available in the common language environment.

```
$ DEFINE/USER SDLJAVA_PACKAGE "com.kednos.jphone"
$ DEFINE/USER SDLJAVA_LIBNAME "DATABASE_RTL"
$ SDL/ALPHA/LANGUAGE=JAVA DATABASEDEF.SDL/VMS_DEVELOPMENT
$ TYPE DATABASEDEF.JAVA
//*****************************************************************************
// Created: 15-Dec-2008 11:24:14 by OpenVMS SDL EV2-3
// Source: 05-DEC-2008 22:59:24 SYSPROG:[TSNEDDON.SCRATCH.SWING]DATABASEDEF.SDL
//*****************************************************************************
package com.kednos.jphone;
import vs.VMSparam;
import vs.SystemCall;
import vs.FieldDescriptor;
public class DATABASEDEF { // IDENT
private static SystemCall nullclass;
private static final String libname = "DATABASE_RTL";
private static SystemCall delete_a_record_return;
public static int delete_a_record(VMSparam[] args) {
 if (delete_a_record_return == nullclass) {
 delete_a_record_return = new SystemCall("DELETE_A_RECORD",libname);
 }
 return delete_a_record_return.call(args);
}
 ⋮
public static final int S_ENTRY = 266;
public static final FieldDescriptor entry_string = new FieldDescriptor(0,0,0,0);
public static final FieldDescriptor ENTRY_STRING = entry_string;
public static final int S_ENTRY_STRING = 266;
public static class _0 extends FieldDescriptor { // FIELDS
public _0() { super(0,0,0,0); }
 ⋮
public static final _0 fields = new _0();
public static final _0 FIELDS = fields;
public static final int S_FIELDS = 266;
}
```

Example 4. Generating Java class from SDL (DATABASEDEF.JAVA)

Wrapping it all up in a Java jacket

Now that the relevant modules have been shifted into a run-time library and the Java declarations have been generated, it is time to build the Java application. Figure 2 shows a screen shot of the Java equivalent of the data-entry screen shown in Figure 1. However, the focus of this article is the use of the J2VMS package and not the Swing toolkit, so the remainder of this article will focus on the interface to the phone book data file.

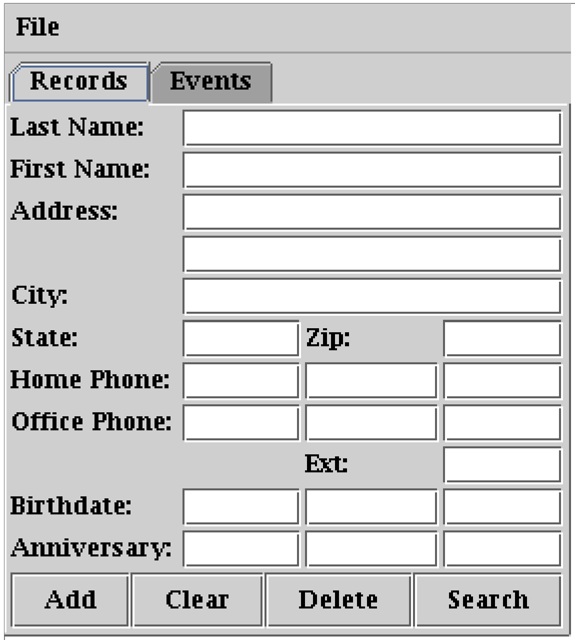


Figure 2. jPhone data entry screen

J2VMS is a Java package that provides a collection of classes that help Java feel more like a native OpenVMS language. It provides support for easily calling native routines and manipulating native data structures, and provides an interface to common OpenVMS argument-passing mechanisms.

To give the run-time library a Java "feel", it is hidden behind the class `com.kednos.jphone.Database`. Each of the native routines is given an equivalent method in the `Database` class. These are shown in Table 1. To make manipulating the phone book data record easier, the `Database` class extends the J2VMS class `vs.VmsStruct`. This means that the instance of `Database` becomes the record buffer also. Each of the methods operates on this buffer.

Java method	Native routine
Database	OPEN_PHONEBOOK
finalize	CLOSE_PHONEBOOK
getRecord	GET_A_RECORD
getMonthRecord	GET_A_MONTH_RECORD
getDateRecord	GET_A_DATE_RECORD
deleteRecord	DELETE_A_RECORD
writeRecord	WRITE_A_RECORD

Manipulating the record is facilitated via the inherited `get` and `put` methods from `vs.VmsStruct`. It would be possible to go one step further and create a set of get/set methods that manipulate each of the record fields. However, it is a reasonable amount of extra work for little (if any) gain. Example 5 demonstrates the field `ADDRESS1` being updated with the contents of the `address1` object (a text input field from the GUI). The object `db` is an instance of the `Database` class.

```
db.put(DATABASEDEF.FIELDS.ADDRESS1,
      DATABASEDEF.FIELDS.S_ADDRESS1, address1.getText(), ' ');
```

Example 5. Updating the ADDRESS1 field (Database.java)

To prevent problems with threading, the instance of the DATABASEDEF class (the SDL-generated "header" file) is declared static and used as a mutex. In Example 6, the code for the method `getRecord` is shown. It demonstrates the use of the `rtl` object as a mutex. The file access statements used in the native PL/I routines rely on the internal I/O buffers maintained by the PL/I run-time library. PL/I assumes that applications are single threaded, which conflicts with the Java environment. By using the mutex, the RTL will only be called by one thread at a time. To ensure that the active record is not lost, it is copied into the storage allocated by the `Database` class using the routine LIB$ LIB$MOVC3 before the mutex is released.

Conclusion

In conclusion, by using the PL/I SDL, SDL tools, and the J2VMS package, the task of writing a Java application that makes use of native routines and data structures has been greatly simplified. The work in creating a custom JNI layer has been completely avoided. Instead, all calls to native code and data manipulation are done from Java, making the application easier to understand and maintain. Using these tools allows more time to be spent on writing the new application, rather than working on the existing part that already works.

```
public class Database
      extends VmsStruct
{

// Own storage
//
private static boolean        isOpen = false;

private LibRoutines           lib = new LibRoutines();
private static DATABASEDEF    rtl = new DATABASEDEF();
:
public boolean getRecord(String name,
                         boolean first)
{
 int              pointer;
 boolean                result = false;

 /* The native run-time library we rely on for accessing the
 * phone book file is not thread-friendly and makes use of
 * PL/I internal file buffers. To prevent acess problems
 * we synchronize access to 'rtl'.
 */
 synchronized(rtl)
 {
      pointer = rtl.get_a_record(new VMSparam[] {
                         new ByRef(trunc(name,
                                     DATABASEDEF.FIELDS.S_NAME)),
                         new ByVal(first ? 1 : 0)
                         });

      if (pointer != 0)
      {
       /* To make sure we don't lose the record we've just
       * fetched (through a call from another instance) we
       * copy the record buffer into our own internal
       * storage.
       */
       lib.lib$movc3(new VMSparam[] {
                   new ByRef(DATABASEDEF.S_ENTRY),
                   new ByVal(pointer),
                   new ByRef(getTarget())
                   });

       result = true;
      }
 }

 return(result);
}
:
}
```

Example 6. Fetching a record (Database.java)

For more information

Tim Sneddon can be contacted via email at tsneddon@kednos.com.

For additional information, including the full source of the example application and kits for all software mentioned in this article, go to:

- www.kednos.com/kednos/Integration

For further information on creating shareable images, see the *HP OpenVMS Linker Utility Manual* at:

- http://h71000.www7.hp.com/doc/83final/4548/4548PRO.HTML

For more information regarding Java, see:

- http://h18012.www1.hp.com/java/alpha/
- www.kednos.com/kednos/Integration/Java