Interfacing to ASAPX from Nonstop Java

The purpose of this document is to outline the steps involved in utilizing the ASAPX API from a Java application running on a NonStop server. It focuses primarily on the technical mechanics of getting the various pieces of the infrastructure in place so that Java can communicate with ASAPX. This document does not attempt to discuss higher-level application design issues related to the use of ASAPX.

There are several steps involved in creating an interface between the Java environment and the ASAPX API. Though none of these steps are particularly difficult or time consuming, they must all be executed in order to successfully call ASAPX procedures from within a Java application. The steps are:

1) <u>Install NonStop Java on your NSK server</u>.

To do so, follow the instructions on the NonStop Java CD. Essentially this step involves uploading and unpacking the Java executable and documentation files. For purposes of this discussion, assume that Java is installed to the base OSS directory */usr/tandem/nssjava*, referred to throughout the rest of this document as *<javadir>*.

Note: Current versions of NonStop Java require hardware-based IEEE floating-point support by default. This is only available on S72000 and later processors. If you attempt to execute a Java application on an earlier processor, an error message will be displayed and the application will not be executed.

2) <u>Declare ASAPX API methods in Java</u>.

In order to call ASAPX procedures from Java, you will need to supply a C++ wrapper library as part of your application (see step 5 below), and call into this library using the Java Native Interface. As a result, when you write your Java code, you need to create a class that declares these wrapper library routines as *native*. While you can layer abstractions on top of this class, each of these declarations needs to match, as closely as possible, the corresponding ASAPX declaration. For example:

```
public static native short AsapRegister(String  DomainName,
                                        short   DomainNameLen,
                                        int[]   SegOffset,
                                        short[] ErrorDetail,
                                        short   SegmentID,
                                        int     SegmentBase,
                                        short   Version,
                                        String  AsapID,
                                        short   AsapIDLen,
                                        short   Flags,
                                        int     Timeout);

public static native short AsapUpdate  (int     SegOffset,
                                        short[] ErrorDetail,
                                        short   DataItem,
                                        long    Value,
                                        short   Math);
etc.
```

Note that you will also need to call the System.loadLibrary method during the static initialization of this class in order to load your wrapper library (created in Step 8 below). For example:

```
System.loadLibrary("MyArchive")
```

3) <u>Compile your Java application</u>.
Use the OSS *javac* utility to compile your application. For example, if your Java source code is contained in the file *MyApp.java*, the command would be:

```
javac MyApp.java
```

If the application compiles cleanly, a *.class* file will be created with the same name as the Java source file (e.g. in this case it would be *MyApp.class*).

4) <u>Generate a C++ header file from the Java class</u>.
As part of developing an ASAPX wrapper library in C++, you must create a C++ header file (i.e. a *.h* file) that contains the declarations of all the native methods you specified in your Java application. While you could attempt to write these declarations by hand, it is far easier (and safer) to use the OSS *javah* utility. To generate a C++ header file for the example above, you would enter:

```
javah –jni MyApp
```

The *javah* utility will open the *MyApp.class* file, extract all native method declarations from it, and generate the corresponding C++ header file (in this case named *MyApp.h*).

It is important that you use *javah* to generate the header file, rather than create it manually, for the following reasons:

- The names of your wrapper library methods will not be the same as the method names declared *native* in your Java source. The wrapper library method names are derived from these declarations, but will not match exactly. There are several rules used by the JVM in determining the actual method name to be called, but in general the alteration will consist of prefixing the declared method name with the keyword *Java_*, followed by the name of the class that contains the method. For example, assume that you have a Java class named *MyClass*, and declare a native method *AsapRegister* as follows:

  ```
  public class MyClass
  {
     public static native short AsapRegister(<remaining declaration>…)
  }
  ```

  The actual native method the JVM will call in this case is *Java_MyClass_AsapRegister*, not simply *AsapRegister*.

  The *javah* utility is aware of JVM native method naming rules and will generate the correct method names that the JVM will expect to find.

- There are type differences between Java and C++. The *javah* utility ensures that the C++ methods are declared with the correct parameter types for the version of Java running on your system (i.e. *javah* makes sure the C++ method is declared correctly to receive the data it's actually going to be passed).

- The JVM adds extra parameters to each method call beyond what is specified in the formal Java method declaration. This can include a pointer to the JVM environment, as well as a pointer to the current Java object. Because the number or types of these additional parameters could change at some point, *javah* should be used to ensure compatibility.

5) **Implement the C++ library methods declared above**.
This step involves writing the C++ wrapper library methods that were initially declared as *native* in your Java source code, and for which declarations were generated above using *javah*.

Your C++ source code will need to include the header file you created in step 4; for example:

```
#include "MyApp.h"
```

You will also need to include the ASAPX header file *asapx.h*:

```
#include <asapx.h>
```

You then need to supply the body for each of these methods. Essentially the goal of each will be the same: to take the information passed in from Java, call the corresponding ASAPX library method, and return the results back to Java. You can obviously also add any additional features here that you feel are useful.

A few key points to keep in mind:

- When you call the actual ASAPX library methods (e.g. *ASAP_REGISTER_*), you are calling into a native NSK library. Thus you should have a general understanding of NSK libraries themselves, the NSK memory model, etc.

- Most of the ASAPX library methods contain output parameters, in addition to returning a value. These output parameters must be communicated back to the Java application, and doing so will require that your C++ library code utilize the JVM environment pointer parameter supplied when Java initially called your C++ method. The JVM environment pointer will allow your C++ library code to access and update objects within the Java run time environment, which is the only mechanism available to communicate output parameters back to the Java method that initiated the call.

6) **Compile the C++ library source**.
Use the OSS *c89* utility for this step, and be sure to specify the *–cg*, *-Wversion2*, and *-Wextensions* flags. *–cg* causes the file to be compiled and not linked, and includes symbolic debugging information. The *–Wversion2* flag enables use of all current C++ features, some of which are required for JNI. The *-Wextensions* flag enables HP-specific C++ language extensions; these are required for interfacing to ASAPX and are used in the *asapx.h* header file. For example, if your C++ source file name is *MyApp.cpp*, you would use the following command to compile it:

```
c89 –cg –Wversion2 –Wextensions MyApp.cpp
```

If the source file compiles cleanly, a *.o* object file will be created (e.g. in this case, *MyApp.o*).

7) **Link your object library with the ASAPX object library**.
At this stage, you have written all of the necessary code to make it possible to call ASAPX from Java. However, you now need to create the correct library files so that the JVM can find the C++ wrapper method and actual ASAPX library routines at run time. The first step in this process is to create a re-linkable native NSK library that contains both your wrapper library and the ASAPX library *asapxsro*. To do so, use the OSS *nld* utility. For example, if your library object file is named *MyApp.o*, the command to create a new library named *Combined.o* which contains both your library and the ASAPX library would be:

```
nld MyApp.o asapxsro –r –o Combined.o
```

The *–r* flag causes *nld* to create a re-linkable library rather than an executable object file, which is what the JVM build process requires.

8) Create a library archive file of your re-linkable library.
In order for the JVM to call your library at run time, that library must be linked to the Java executable. The build process that accomplishes this (referenced in step 11) expects your library to be contained in an archive. To create this archive, use the OSS *ar* utility with the *–rcv* flags. For example, if the library file containing a combination of your C++ wrapper library and the ASAPX library file is named *Combined.o*, the command would be:

```
ar -rcv MyArchive.a Combined.o
```

This would create the library archive file *MyArchive.a*. The *–rcv* flags to *ar* cause a new archive file to be created if necessary, suppress output of a message stating that the file has been created, and displays the name of each file added to the archive (which in this case would be limited to *Combined.o*).

9) Move the library archive file to the *<javadir>/lib* directory.
The build process for the JVM will automatically link any library archives contained in the *<javadir>/lib* directory into the JVM; thus you need to copy the library archive created in step 8 to this directory. For example:

```
cp MyArchive.a <javadir>/lib
```

Note: You may need root (SUPER.SUPER) authority in order to copy a file to the *<javadir>/lib* directory.

10) Save a copy of your existing Java Virtual Machine (Java executable).
This is located in the *<javadir>/bin/oss/posix_threads* directory. Copy the file *java* to another location; you can fall back to this in the event the rebuild fails.

Note: You may need root (SUPER.SUPER) authority in order to copy the Java executable.

11) Rebuild the Java Virtual Machine.
Now that everything is prepared, you need to rebuild the Java Virtual Machine to link in the library archive you created. To do so, go to the *<javadir>/install* directory. Then enter the command *make*. This will cause the Java Virtual Machine to be rebuilt, and it will contain your library archive.

Note: You may need root (SUPER.SUPER) authority in order to rebuild the JVM.

12) Turn on the RUNNAMED attribute of the new JVM.
In order to utilize the ASAPX API, your application's Java process must have a name. By default, the *java* executable does not run with a process name. You must enable this option using the OSS *nld* utility. To do so, go to the *<javadir>/bin/oss/posix_threads* directory, and enter:

```
nld -change RUNNAMED ON java
```

You are now able to execute your Java application and interface to ASAPX.

Note: You may need root (SUPER.SUPER) authority in order to turn on the RUNNAMED attribute of the new JVM.

Ongoing Maintenance:

If you subsequently make any changes to the native declarations in your Java application, you will need to re-execute this sequence of steps beginning with step 2.

If you do not make changes in your Java application, but wish to update your C++ wrapper library, you will need to re-execute this sequence of steps beginning with step 5.