+1 650-938-9328 ext. 802

WILDE ON MOBILE & EMBEDDED PORTING, INTEGRATING AND DEVELOPING DEVICE SOFTWARE

HOME NEW PROJECTS CONTACT ME

HOME » ALL » ANDROID NATIVE LIBRARIES FOR JAVA APPLICATIONS

ANDROID NATIVE LIBRARIES FOR JAVA APPLICATIONS

Posted by Charles Wilde on Sep 23, 2009 in All, Android, Mobile Application Development, Mobile Software | 1 comment

Part Five of a Series of Posts about Android Mobile Application Development.

In three prior blog posts, Developing An Android Mobile Application , Android Software Development Tools – What Do I Need? and Android Native Development on Ubuntu 9.04 (Jaunty Jackalope), I discussed our approach for choosing Android as the next platform for our company's mobile application, Aton Connect. Once I identified which platform was most suitable, I needed to decide on development tools that would allow us to develop our mobile application rapidly. Because the Aton Connect software architecture combines native and managed code, I am looking for ways to include native C/C++ code in our Android version.

In my fourth post, Android Native Development Using the Android Open Source Project, I mentioned that there are now two primary sets of tools that can be used to develop Android applications that include native C/C++ native code. The original, and still unsupported way, is to leverage the tools built into the Android Open Source Project to build Android applications.

The Android team has recently released the Android NDK or Android Native Development Kit which is the supported way to integrate native code into Android Java applications. The initial Android NDK is quite limited in scope, but does offer the ability to develop native code libraries on Windows.

In order for the NDK to operate on Windows, you must install the Cygwin command shell and emulator to allow the Linux based NDK to function. I chose to forgo this Windows operation and continue with the Android Open Source project on Linux because it provides much greater capability without the idiosyncrasies introduced by Cygwin.

The previous post was concerned with building a pure C++ Android application. A much more practical method is using Java for the GUI elements, and C/C++ for background support processing. Each language has its strengths. GUI activity is best done with the Java language supplied with the Android tool kit. C/C++ native code libraries can accomplish what is difficult to do in Java.

In this post, we will be working with the Android Open Source Project (OSP) on Ubuntu to build Android Java applications that utilize native code libraries. In a later post we will address the NDK, with its advantages and limitations.

A Simple Android Java Application That Uses A Native Code Library

The initial question to be resolved is how do we "glue" the native code library with the Java main application code? What kind of interfacing is required?

I have identified at least four mechanisms:

ABOUT CHARLES WILDE

Current Focus: Windows Phone 8, Windows 8 Store and Embedded

I provide porting, integration and mobile and embedded software development services and mobile apps to a wide variety of companies including Trimble (GPS), Trilliant (Energy), NavCom (GPS), Baxa (Medical Devices), KLM-Air France (Airlines), Nissan USA (Auto) and a host of others.

Connect with me on LinkedIn

Contact me Here

LOOKING FOR SOMETHING?

Search for:

Search

DON'T MISS THE NEXT POST!

You won't find these articles anywhere else. Sign up to get updates via email.

[nm-mc-form fid="5"]

CATEGORIES

- All
- Android
- iPhone
- Mobile Application Development
- Mobile Devices
- Mobile Software
- Native Code
- Windows 8
- Windows Embedded
- Windows Phone

- Java Native Access (JNA) This is a mechanism very similar to Windows managed language P/Invoke or Platform Invoke. JNA allows you to call directly into native functions using natural Java method invocations. The Java call looks just like it does in native code. Although this would be the best mechanism, it is not currently supported by the Android Dalvik Java Virtual Machine.
- Java Native Interface (JNI) This is one of the original Java/Native interface mechanisms available. It is the mechanism implemented in the Android Dalvik Virtual Machine. This mechanism is used extensively throughout the Android Open Source Project. In its raw form, developing JNI code interfaces can be time consuming and error prone due to the low level detail that must be considered in its use.
- Simplified Wrapper and Interface Generator (SWIG) This is a JNI "helper" that takes as input C/C++ interface definition files and outputs C/C++ JNI wrapper code and a collection of Java code files to be added to your Java application. SWIG implementations include versions for Microsoft Visual Studio and Eclipse.
- Javah This is a tool available in the JDK. It produces C header files from a Java class. These files provide the connective glue between your Java and C code.

Since JNA is not available with the current Android releases, I will use JNI. The Java classes generated by SWIG are not compatible with the Dalvik JVM, so I opt to use "javah" automation rather than hand generate the needed JNI code.

An excellent overview of how to use javah to integrate a native shared library with Java as provided by the Sun JDK is given in Chapter 2 of the book "*The Java Native Interface Programmer's Guide and Specification"* published by Sun Microsystems. This is available online at: http://java.sun.com/docs/books/jni/html/start.html.

Some variation from this description is need to accommodate the Android Dalvik Virtual Machine. The Android Dalvik Virtual Machine is separate from and independent of the Sun JDK. The basic process is to declare the Java form of your native function in your Java class source code file by using the keyword "native". You then use the "javac" compiler, a tool in the JDK, to create a Java class file. The "javah" tool in the JDK takes this Java class as input, and outputs a C++ header file. The C++ header file contains a function definition that is compatible with the Java and Dalvik virtual machines.

Note that I am using the J2SE compiler from the Sun JDK to generate a Java class file, not the Android Dalvik compiler. The Dalvik compiler generates a "dex" file which is not the same as a Java class file. The "javah" requires a class file, not a dex file as input. The header output by javah is compatible with the Dalvik Virtual Machine, even though the tools used to generate it were designed for the Java virtual machine in the Sun JDK.

First, let's write a simple Java application that calls a native shared library to multiply two numbers and display the result.

```
package com.MultPkg;
import android.app.Activity;
import android.util.Log;
import android.os.Bundle;
import android.widget.TextView;

public class Mult extends Activity {

    static {
        try {
             Log.i("JNI", "Trying to load libMult.so");
             System.loadLibrary("Mult");
         }
        catch (UnsatisfiedLinkError ule) {
            Log.e("JNI", "WARNING: Could not load libMult.so");
        }
    }

    // This declares the native function to be imported from the shared librar
    native private int mult(int a, int b);

```

Internal Details of a Native Code Library

Some thing to note here is the static function at the beginning of the class. Java calls this static function once when the class is instantiated. I use this as an opportunity to load the shared library containing the native shared library "libMult.so".

For the System.loadLibary method to locate the library, it must be placed into the proper sub folder which is "dex/lib/armeabi" where "dex" is the folder containing the Dalvik Java class file.   The script needed to build an APK package with the native shared library at the proper folder location is given below.

Below the static function is the line:

```
native private int mult(int a, int b);
```

This is the Java language declaration of the native function to be called from the Java application. Because the interface mechanism for Android Dalvik Java is JNI, rather than the more modern JNA, you must create some "glue" code in your native C++ program so that it can interface with the Dalvik JVM. This glue code registers native methods with the JVM, so the JVM will know how to link to the native code.

For Windows systems, the native code publishes or exports a description of the method interfaces as part of the shared library and the Windows OS provides the interface mapping. When using JNI however, the task of interface mapping is left to explicit code you must write into each native C++ module.

Automating The Development Of JNI Interfaces With The "javah" Tool

After compiling the Java application into a J2SE compatible class object file, the JDK tool "javah" can be used to convert this Java declaration into a native C++ declaration in a header file. The information in the native C++ declaration can be used in native C++ shared library source code to interface with Java via JNI.

For simple function calls, you might find it easier to work out the correct C++ interface from the description of JNI given in the book "*The Java Native Interface Programmer's Guide and Specification"* published by Sun Microsystems. For more complex interfaces, including those addressing structures, using the "javah" tool can speed your work and help you avoid errors.

To demonstrate the "javah" tool, create a text file named "Mult.java" containing the Java application given above. The Java language has the characteristic that the folder structure containing the Java code is influenced by the Java class name. For those used to other programming languages, this implied correlation between class name and folder structure can be a surprise.

In this case, the full class name is a combination of the package name "com.MultPkg" and the class name "Mult" yielding a full class name "com.MultPkg.Mult". This, in turn, corresponds to a folder tree and file name of "com/MultPkg/Mult.java". To take advantage of the Android Open Source Project build system, we will create a folder named "MyApps" in the "external" folder of the Android open source folder structure. The base folder of our "Mult" test application is then located at $ANDROID_HOME/external/myapps/mymult". The Java source file is then located at "$ANDROID_HOME/external/myapps/mymult/com/MultPkg/Mult.java"

To use the “javah” tool, we first compile the source code into a Java class using the compiler provided by the JDK. We cannot use the Android Dalvik compiler here, since it does not generate a regular Java class file, required as input to the “javah” tool. The following steps accomplish this.

First start a terminal session and change directory to the base of our application:

```
cd $ANDROID_HOME/external/myapps/mymult
```

Porting Native Code to Android

then enter the Java compilation command:

```
javac com/MultPkg/Mult.java -classpath $ANDROID_HOME/out/host/common/obj/JAVA_L
```

Note the “-classpath” option that links classes from the Android Open Source Project into the compilation. The output of this compilation “Mult.class” is located in the same directory as the “Mult.java” file.

Next we use the “javah” tool to extract a C++ header file describing the required interface between the native C++ function and the Java application.

```
javah -jni com.MultPkg.Mult
```

In this case, the “javah” tool derives the location of the Java source file from the class name given in the command line. The class name “com.MultPkg.Mult” gets converted by “javah” into the file name “./com/MultPkg/Mult.java”. The output of the “javah” tool is a C++ header file located in the application base directory “./com_MultPkg_Mult.h”. This file contains the following text:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include
/* Header for class com_MultPkg_Mult */

#ifndef _Included_com_MultPkg_Mult
#define _Included_com_MultPkg_Mult
#ifdef __cplusplus
extern "C" {
```

The symbolic definitions “JNIEXPORT” and “JNICALL” can be ignored since they are defined to be empty in the “jni_md.h” machine dependent header file for Linux referenced by “jni.h”.

Compare this C++ header with the Java function declaration:

```
native private int mult(int a, int b);
```

Note that the C++ declaration includes four arguments, whereas the Java declaration has only two. The first C++ declaration argument has a type of “JNIEnv *” which is a pointer to the JNI interface function pointer table. JNI interface functions are used to create objects, access fields, and call methods.

The second argument of type “jobject” is a pointer to the Java object or class on which the method (function) was invoked. It can be used to look up data members or call other methods of the class.

The actual arguments in the function call follow the first two, so in this case we have four arguments to the native C++ function, the last two corresponding to the two arguments in the Java function.

Designing A Native Code Library That Is Compatible With Java

We are now ready to code the native C++ function "Mult.cpp", which looks like this:

```
#include

#include <string.h>
#include
#include

// LOG_TAG identifies a log message with a symbolic tag
#define LOG_TAG "mult"
#include "utils/Log.h"

// define our "mult" function here using information from the header generated
// note we add "static" to the definition and eliminate JNIEXPORT and JNICALL

static jint Java_com_MultPkg_Mult_mult (JNIEnv * env, jobject myobj, jint a, j
{
      return (jint)(a * b);
}

// now we have to register our "mult" function as a method with JNI so the JVM

// first step is to define an array of structures describing the methods we wa
// each structure in the array describes one method with the name of the metho
// a pointer to the native function that implements the method.
//
// The method signature describes the Java types of the arguments (if any) and
// string. This string is given in the header generated by the "javah" tool. I
// described by the line "* Signature: (II)I".

static const JNINativeMethod gMethods[] = {
    { "mult", "(II)I", (void*) Java_com_MultPkg_Mult_mult }
};

// the next step is registering the native method when the OnLoad event occurs

jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env = NULL;
    jint result = -1;
    jclass clazz;
    static const char* const strClassName = "com/MultPkg/Mult";

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failedn");
        return result;
    }

    if (env == NULL) {
        LOGE("ERROR: env is NULLn");
        return result;
    }

    /* find the class handle */
    clazz = env->FindClass(strClassName);
    if (clazz == NULL) {
        LOGE("Can't find class %sn", strClassName);
```

Although this is a bit more tedious than the native interface methods available in Windows, it better supports the multi-platform goal of Java.

Compiling And Building The Native Code Library

To compile this native code, we create a new folder $ANDROID_HOME/external/myapps/mymult/jni and create a source code file “Mult.cpp” containing the code given above. We add an “Android.mk” file to this folder as follows:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := samples

LOCAL_MODULE:= libMult

LOCAL_SRC_FILES:= Mult.cpp
```

To build the shared libraries from the “Mult.cpp” file using the “Android.mk” file, open a terminal window and enter these three commands:

```
cd $ANDROID_HOME/external/myapps/mymult/jni
source envsetup.sh
mm
```

This will cause the shared library to be built from the “Mult.cpp” source code file. The build listing output on the terminal screen will look like this:

```
make: Entering directory `/home/cawilde/mydroid'
build/core/product_config.mk:261: WARNING: adding test OTA key
============================================
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=
TARGET_BUILD_TYPE=release
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=
============================================
target thumb C++: libMult <= /home/cawilde/mydroid/external/myapps/mymult/jni/
target SharedLib: libMult (out/target/product/generic/obj/SHARED_LIBRARIES/lib
target Non-prelinked: libMult (out/target/product/generic/symbols/system/lib/l
target Strip: libMult (out/target/product/generic/obj/lib/libMult.so)
Install: out/target/product/generic/system/lib/libMult.so
Finding NOTICE files: out/target/product/generic/obj/NOTICE_FILES/hash-timesta
Combining NOTICE files: out/target/product/generic/obj/NOTICE.html
```

```
gzip -c out/target/product/generic/obj/NOTICE.html > out/target/product/generi
make: Leaving directory `/home/cawilde/mydroid'
```

The build listing describes where the various types of build outputs are located. These are close to the end of the session:

```
target SharedLib: libMult (out/target/product/generic/obj/SHARED_LIBRARIES/libM
target Non-prelinked: libMult (out/target/product/generic/symbols/system/lib/li
target Strip: libMult (out/target/product/generic/obj/lib/libMult.so)
Install: out/target/product/generic/system/lib/libMult.so
```

The build outputs of interest are described by the lines:

```
Install: out/target/product/generic/system/lib/libMult.so  (no symbols) (5 KB)
target SharedLib: libMult (out/target/product/generic/obj/SHARED_LIBRARIES/libM
```

These lines mean the build outputs are located at $ANDROID_HOME at the paths given above.

Compiling And Building The Android Java Application That Uses The Native Code Library

Next, we build the Java package using the Android open source build system. As we described above in the "javah" tool instructions, the Mult.java file was placed at $ANDROID_HOME/external/myapps/mymult/com/MultPkg/Mult.java.

We add the required "AndroidManifest.xml" file in the folder $ANDROID_HOME/external/myapps/mymult as follows:

```
"1.0" encoding="utf-8"?>
"http://schemas.android.com/apk/res/android" package="com.MultPkg">
    "Mult">
        "Mult">

                "android.intent.action.MAIN"/>
                "android.intent.category.LAUNCHER"/>

```

And we add the "Android.mk" file in the folder $ANDROID_HOME/external/myapps/mymult as follows:

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := samples

LOCAL_PACKAGE_NAME := MultPkg

LOCAL_SRC_FILES := com/MultPkg/Mult.java
```

To build the Java application from the "Mult.java" file using the "Android.mk" file, open a terminal window and enter these three commands:

```
cd $ANDROID_HOME/external/myapps/mymult
source envsetup.sh
mm
```

This will cause the Java application package "MultPkg" to be built from the

"Mult.java" source code file. The build listing output on the terminal screen will look like this:

```
make: Entering directory `/home/cawilde/mydroid'
build/core/product_config.mk:261: WARNING: adding test OTA key
============================================
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=
TARGET_BUILD_TYPE=release
TARGET_ARCH=arm
```

The build output of interest is described by the line:

```
Install: out/target/product/generic/system/app/MultPkg.apk
```

This line means the build output is located at $ANDROID_HOME at the path given above.

Merging The Android Java Application And The Native Code Library Into An Installable Android Application

The "MultPkg.apk" at this point only contains the Java (Dalvik) DEX class, but not the native shared library "libMult.so". We use the following commands to add the shared library to the package at the correct folder location so that it can be successfully located by the "System.loadLibrary("Mult");" line in the Java application.

```
cd $ANDROID_HOME/external/myapps/mymult

mkdir bin
unzip $ANDROID_HOME/out/target/product/generic/system/app/MultPkg.apk -d bin
mkdir -p bin/lib/armeabi
cp $ANDROID_HOME/out/target/product/generic/system/lib/libMult.so bin/lib/armea
apkbuilder MultPkg.apk -v -rf bin
```

What this series of commands does is create a folder called "bin" in the "mult" application base directory. It then extracts into the "bin" folder the APK package created by the Java compilation, which includes the DEX class file, the AndroidManifest.xml file and meta-data. It then creates the subfolder at the correct location to allow the "System.loadLibrary("Mult");" command in the Java source file to locate the native library. The native library is copied into this subfolder "bin/lib /armeabi" and the package is rebuilt using the "apkbuilder" tool of the Android SDK. The APK file containing both the Java application and the native shared library is then located at $ANDROID_HOME/external/myapps/mymult/MultPkg.apk.

Installing The Android Java Application That Uses A Native Code Library

You can now load this complete MultPkg.apk package into the Android emulator to test it. Start a new terminal window and point it to $ANDROID_HOME/external /myapps/mymult.

You can determine the names of the emulator skins or AVD instances by using the command: "android list avds". If you have not set up your AVD, refer to this page for instructions: http://developer.android.com/guide/developing/tools/avd.html

You can then start the emulator with this command: emulator -avd where avd_name is the name you gave the AVD when you created it, and is listed by the command: "android list avds". Note that the ".avd" extension is not included as part of . Also note that the emulator command will not exit as long as the emulator is active. You will need to open another terminal emulator window to input additional commands while the emulator is running.

Once the Android emulator has booted up you can install the MultPkg application using the following procedure. Start another new terminal window and point it to $ANDROID_HOME/external/myapps/mymult.

Use the command "adb install –r MultPkg.apk" to install the Mult application. The terminal screen results will be similar to:

```
* daemon not running. starting it now *
* daemon started successfully *
98 KB/s (4369 bytes in 0.043s)
                 pkg: /data/local/tmp/MultPkg.apk
Success
```

You can now look at the emulator, click on the Menu button and observer the icon labeled "Mult" next to the "Messaging" icon. If you click on the Mult icon, it will bring up the Mult application and display the text line "Native shared library result of 3 * 6 is 18". This demonstrates the Java application successfully calling the native shared library with two arguments and printing the result from the shared library using Java code.

After installation onto the emulator, the native shared library is located at "/data /data/com.MultPkg/lib/libMult.so". The package is located at "/data /app/com.MultPkg.apk". The Java application is located at: "/data/dalvik-cache/data@app@com.MultPkg@classes.dex".

Debugging The Android Java Application That Uses A Native Code Library

So now how do we debug the application?

To debug the Java portion of the project, we can simply use the Eclipse development environment with the Android Development Tools. To do this, create a new Android project in Eclipse, and use the Eclipse workspace for storage of the project.   Use "Mult" as the Project name, Android 1.5 as the Build Target, use "Mult" as the Application name, "com.MultPkg" as the Package name, and "Mult" as the Activity to Create. Use 3 as the Min SDK Version.

Then display the auto-generated source code in the Eclipse Navigator located at Mult | src | com | MultPkg | Mult.java. In the Eclipse editor window, replace the default Java source code that was auto-generated with the source code located at "$ANDROID_HOME/external/myapps/mymult/com/MultPkg/Mult.java".

Next set a breakpoint on the line " int multiplicand = mult(3,6);". Use the Eclipse command "Run | Debug | Android Application". This will cause Mult.java to be compiled and loaded into the emulator. Execution will stop on the breakpoint you set.

Next, verify that the Mult application has been started on the Android emulator screen, and that the text area of the screen is blank. Then click on the Eclipse Resume (F8) button and verify that the message "Native shared library result of 3 * 6 is 18". " is now on the screen.

Note that Eclipse did not load the native shared library "libMult.so" into the correct location on the emulator, it was loaded when we installed the Mult application package. An alternative is to copy the "libMult.so" using the following command:

```
adb push $ANDROID_HOME/out/target/product/generic/system/lib/libMult.so /data/d
```

Debugging of the native library can be accomplished with the techniques described in the previous post in this series: Android Native Development Using the Android Open Source Project. The native library can be called from a native main application and debugged using GDB and gdbserver as described in that post. Note that the JNI interface code can be left as is in the library you are debugging, it will be ignored by the native C/C++ main application.

* * * * * * * *

Other posts in this series:

- Developing An Android Mobile Application
- Android Software Development Tools – What Do I Need?
- Android Native Development on Ubuntu 9.04 (Jaunty Jackalope)
- Android Native Development Using the Android Open Source Project
- Android Native Libraries for Java Applications
- Porting Native Code to Android: From Business Case to Coding
- Pure Native App or Java Shell for Android Market?
- Porting Applications Using the Android NDK
- Integrating the Android NDK C++ Native Support into Eclipse Using Sequoyah

With 20+ years as a top software and firmware developer, Charles Wilde has acquired a combination of proven business smarts, mobile development skills and device engineering expertise that is hard to match. Charles is available to consult with you and your team about native code development in Android, Windows Mobile or Windows CE. Wilde is the author of Porting Native Code to Android and can be reached at AtonMail (at) aton (dot) com. © 2010 Aton International, Inc.

.

1 COMMENT

- WIL *April 2, 2010*

  You literately SAVED my life! Thanks!

  REPLY

POST A REPLY

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

SUBMIT COMMENT