Volume 1, Issue 1

Summer, 2013

Golden Bits Tech Newsletter

Golden Bits Software ®

Going Native with Android NDK
How to call C/C++ code from your Android App

Highlights:

- Sample code can be downloaded from the Golden Bits Web site. Use this link:

http://www.docs-goldenbits.com/newsletters/goingnative-tictactoe-samplecode.zip

Golden Bits Software, Inc.
3525 Del Mar Heights Road, Suite 158
San Diego, CA 92130
858-259-3870 phone
858-259-7655 fax

Inside:

Introduction

Google implemented their Android application framework using the Java language; as a result, Android applications are written in Java. Why Java? There are a number of reasons, both legal and technical, but the end result is that if you want to create an Android application, you'll need to use Java. However, there are some cases where using Java is not the best approach. Why should you rewrite an existing (and well tested) C/C++ code base just for Android? What if you need more performance than what Java can offer? How can you use a common code base for Android and other platforms (iOS, desktop, and embedded)? How can you use a really good open source library written in C/C++?

Fortunately there is a way you can do all of this by using Android's Native Development Kit, otherwise known as Android NDK. This newsletter provides a primer on how to use the NDK and provides a sample application that you can download.

The sample application is a simple Tic-Tac-Toe Android application that uses a C/C++ code for the game logic. This sample application calls the C/C++ source using Android's Native Interface

Android NDK

The NDK provides a set of tools and methods to enable you to call your C/C++ code directly from a Java based Android application. Google tries to discourage using the NDK, and yes most of your Android application development will work with Java just fine, but if you need to use C/C++ code – then use it!! The NDK is a separate download that you install along with the Android SDK.

One of the key things the NDK does for you is handling the compiling and packaging your code for the different target processors that run Android, specifically ARMv5TE, ARMv7-A, x86, and MIPS. The processor architecture details are defined by an Application Binary Interface (ABI for short) that defines the low level processor details such as data alignment, CPU instruction set, calling conventions, and

Golden Bits Software ® is a software engineering firm providing consulting services in a wide range of diverse technologies.

Contact information:

Email: deang@goldenbits.com
Web: goldenbits.com

Going Native!!

What about application development support?

Yes, Google provides an Eclipse plugin called ADT, Motorola also provides a very nice Eclipse plugin called MOTODEV. Both of these plugins are feature packed and recommended.

more. Each supported processor type has an ABI short name that is used by the build system, the name are: armeabi, armeabi-v7a, x86, and mips.

The NDK uses the armeabi by default unless you specify additional processor types. To specify additional processor types, in the Application.mk file set the APP_ABI for each of the desired processor types. In the accompanying Tic-Tac-Toe sample application, APP_ABI is set to x86 and armeabi types. The NDK will build a library for each processor type under the lib sub directory in your project directory. When your application is deployed in an .apk file, all of the processor specific libraries are included. During the installation process the Android system will pick out the appropriate library automatically.

The NDK also provides a set of prebuilt libraries that are commonly used such as libc, libm, libz, and more. The actual mechanism that performs Java to C/C++ call is Google's implementation of the JNI standard, in fact the Java VM is an implementation created by Google. The NDK is available for Linux, Windows, and Mac for both 32 and 64 bit hosts, where the host is the system you are using to build, not the target Android device.

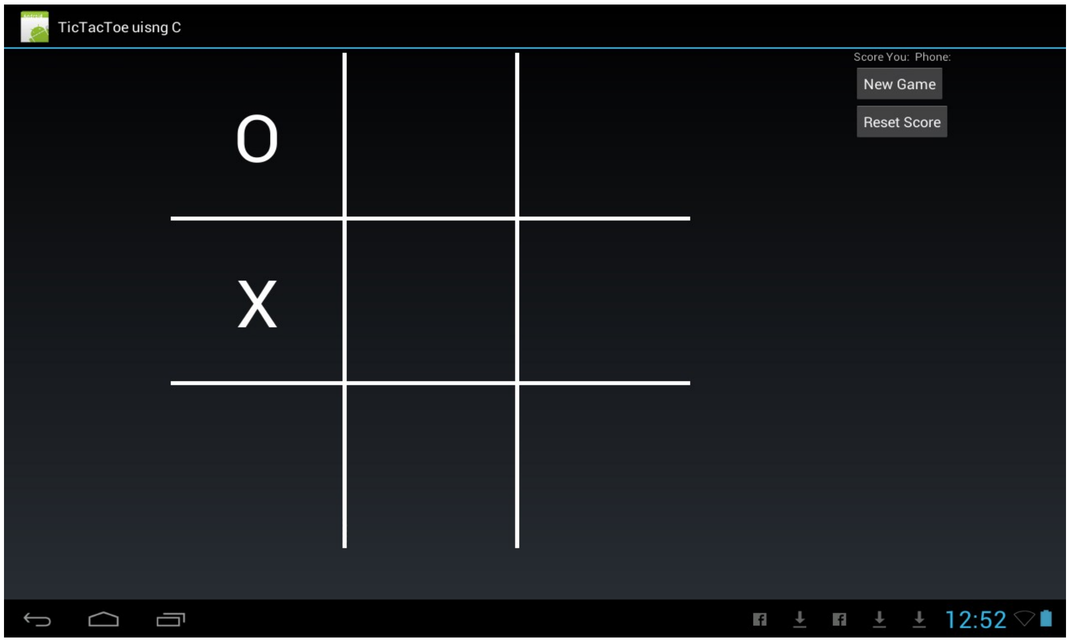
Installing the NDK is very simple. Just untar the NDK files under the main Android SDK directory, the sub directory will be android-ndk-r**X** when **X** is the NDK version level. The most recent version is '8e', so the directory will be 'android-ndk-r8e'. You will need to set the NDK environment variable to point to the directory where you untared the NDK. A great place to start learning the NDK is with the documentation that is located in the 'docs' directory of the NDK. Another great place to start learning the NDK is by examining the sample code under the samples directory.

Support for multiple processor types, packaging and deploying, ease of use, debugging, and pre-build libraries are a few of the reasons why the NDK is a very powerful tool.

A simple Tic-Tac-Toe Android application is shown here. This application is intentionally very simple, the main purpose is to provide a quick start for using and learning the Android NDK.

You can download the sample application from this link:

http://www.docs-goldenbits.com/newsletters/goingnative-tictactoe-samplecode.zip



Going Native!!

Application Setup

For your Android application, your C/C++ source code and Android.mk file should go in the 'jni' directory under your project's main directory, but this is not a requirement for the source files. The Android.mk file is used by the NDK build script to find your native source files. You can modify the LOCAL_PATH variable in the Android.mk file to point to your C/C++ source files in a different directory. The sample Tic-Tac-Toe application does this, the Android.mk file points to the source files located in a common directory. The Application.mk file (also located in the jni sub directory) is used to specify additional build options, such as multiple processor types (mentioned previously) or to override the compiler options.

To build your native code, run the ndk-build script your project directory. By default the script will look for the Appliation.mk file in the jni sub directory. The ndk-build script is located in the NDK directory.

Calling C/C++ Code

The actual mechanism used to call C/C++ code is the Java JNI interface (JNI == Java Native Interface). Java JNI has been around since the beginning of Java itself, for Java applications it provides a way to call native code. The latest JNI version is 6.0.

So how do you actually call your native code? ***Show me the code!!*** Here's a quick tour of the Tic-Tac-Toe sample application below. In the sample Tic-Tac-Toe application, look at the Java file GameBoardView.java. A snippet of the code is shown below (Figure 1),

```
// Declare the native functions here
public native int  TicGetNextMove(byte UserSelectedSquare,
                                  GameMove NextMove);

public native int  TicGetSquareState(byte Square,
                                     SquareState SqState);

// Load the library as part of the class construction.
// make sure to load our native library
static
{
     System.loadLibrary("GameEngine");
}
```

Figure 1

A cyclist was stopped by Customs. "What's in the bags?", asked the officer, pointing to the cyclist's panniers. "Sand," said the cyclist. "Let me take a look.", said the cop. The cyclist did as he was told, emptied the bags, and proving they contained nothing but sand, refilled the bags and continued across the border.

A week later, the same thing happened and continued every week for a year until one day the cyclist with the sand bags failed to appear.

A few months later, the cop saw the cyclist living it up downtown. "You sure had us foxed", said the cop. "We knew you were smuggling something across the border. I won't say a word, but what was it you were smuggling?" ..... "Bicycles!" responded the cyclist.

Going Native!!

Once you have declared the native functions and loaded the library, actually calling the native functions is very easy. Just invoke the function directly with the appropriate arguments. In GameBoardView.java you can see how the `TicGetNextMove()` is called directly (Figure 2). On the C/C++ side of things, you will need to do a little more work but not

```
int retCode = TicGetNextMove(gameSq.mSquareNum, gameMv);
```

Figure 2

much. First wrapper functions need to be created, see AndroidWrapper.c in the sample application and in Figure 3.

```
jint
Java_com_goldenbits_tictactoe_GameBoardView_TicGetNextMove(JNIEnv *env,
                    jobject obj, jbyte CheckedSquare,
                    jobject GameMoveObj)
{
         // Your native C/C++ code goes here.
         // The actual entry points to your code are C functions.
         // You will call C++ code from here.
}
```

Figure 3

These wrapper functions will then call your native code. In the sample code you can see when the native method, TicGetNextMov() is called, the Java VM routes and invokes the Java_com_goldenbits_tictactoe_GameBoardView_TicGetNextMove() function in your C/C++ library. Why the funny function declaration in AndroidWrapper.c? The function declaration is part of the Java JNI specification; it enables the JNI interface to find the correct C/C++ function. The format of the native functions is:

```
extern "C" <return_type>
Java_<package_name>_<class_name>_<method_name>( JNIEnv* env, ... )
```

In our example, the class name is com.goldenbits.tictactoe.GameBoardView but the periods ('.') are replaces with the underline character ('_').

A Few JNI, C/C++ Details

Every native function begins with two arguments, JNIEnv and jobject. JNIEnv is a pointer to the JNI interface itself, the set of pointers to functions that your native code calls to interact with the managed Java code. For example, when using JNI functions to access a class member variable, your native code will call `JNIEnv *(pEnv)->GetFieldID()` (Note the parens around `pEnv`). The second argument, `jobject`, will vary depending if the native function is static or non-static. A little clarification is necessary here; a native function is static if it is declared as static in the Java class, for example:

```
class SomeJavaClass {

  public static native void MyStaticFunction();
```

```
                }
```

From your Java code, the static function would be invoked by calling `SomeJavaClass.MyStaticFunction()`

When MyStaticFunction native function is called, `jobject` is a reference to the Java ***class*** not an instance of a Java class. A non-static function is declared as a member function of the class, for example:

```
class SomeJavaClass {

    public native void MyFunction();
}
```

In this example, the non-static function is called from an instance of the class, like this:

```
class SomeJavaClass  myClass = new SomeJavaClass();
myClass.MyFunction()
```

In this example, `jobject` is a reference to the `SomeJavaClass` ***instance***.

Java Type	Native Type	Description
boolean	jboolean	Unsigned 8 bits
byte	jbyte	Unsigned 8 bits
char	jchar	Unsigned 16 bits. NOTE: Java uses Unicode which is 16 bits. If your native code uses a different encoding such as ASCII or UTF-8, you will need to convert .
short	jshort	Signed 16 bits
int	jint	Signed 32 bits
long	jlong	Signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

Table 1

The remainder of the arguments map to their respective data types. Primitive data types are passed as values, Java objects, such as strings, arrays, or classes, are passes by reference to the native function. The primitive data types are show in Table 1.

References to objects.

Objects passed to the native functions have a local reference which means the object reference is valid during the execution of the native call, once the native call returns the local reference is no longer valid and will be cleared by the JVM's garbage collector. Thus, it is a very bad idea to store a reference to an object in your native code. However there may be scenarios where you want to insure the object is retained and not garbage collected. For example, maybe your native code returns an object that contains counters of some set of operations and you want to use a single object instead of returning a new object every time counters are requested. In this case you could set a global reference to this object and save off the reference in your native code. Just do not forget to free the global reference when the object is no longer needed.

Going Native!!

Access Object Fields

Reading and setting object fields is one of the common ways used to pass arguments to and from native code. Using Java objects to pass arguments is very simple and easy way to call native code. In the Tic-Tac-Toe sample application, two Java object are used to pass game information, GameMove and SquareState. Also in the Tic-Tac-Toe sample, look at the function TickHelper_SetByteValue in AndroidWrapper.c. This function uses three JNI functions , GetObjectClass(), GetFieldID(), SetByteField() to set the byte value of an object. JNI provides a set of assessor and setter functions that enable your native code to get and set member variables of your Java object.

Summary

Calling native C/C++ code is really not that difficult or mysterious and depending on your application, does have advantages. This is a quick overview of the Androids's NDK and JNI, enough to get you started. Download the sample Tic-TacT-oe application and give it a spin. It provides a great od starting point for ***Going Native.***

Resources

The NDK documentation and download can be found here: http://developer.android.com/tools/sdk/ndk/index.html

Android, really good tips, a must read: http://developer.android.com/training/articles/perf-jni.html

Google Groups: https://groups.google.com/group/android-ndk

Golden Bits Project Experience

Web. Enhanced and extended web monitoring system. Added new web services using Java servlets.

Embedded Devices/OS. Experience with embedded OS systems: Linux, TI DSP, and Threadx. Developed application and system level code including USB enhancements and embedded management system for a large blade enclosure. Ported embedded Linux driver code to Power PC 44GX and Broadcom 1255 processors.

WDM, NDIS Device Drivers. Developed a WDM and NDIS device driver for a prototype wireless system. Also developed USB wireless driver which presented the USB device as a network adapter to the host.

Device Drivers, Windows, Linux, Mac. Developed a variety of Windows, Linux, and Mac rivers to handle network packet inspection, USB devices, SCSI Fibre channel adapters, ethernet adapters, and job scheduling to a device.

Set Top Box. Helped port a STB to new Broadcom 7405 chip. Developed a script language and compiler used to code the television UI (guide menus, channel select).

SCSI Port driver for Fibre Channel. Designed the operating system layer for a SCSI storage driver (XP, Win2K, NT, Linux) for a fibre channel HBA (host bus adapter – PCI/SBUS card).