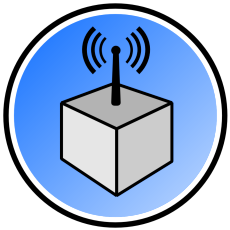


LittleBox
Solutions

Introduction to the Android NDK

James Puderer

May 18th, 2011

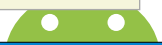
What

- The Android NDK allows you to include native code (C/C++) in your application
- NDK builds native code as a library that your application can call using JNI
- NDK includes all the scripts, Makefiles, toolchains and libraries needed to compile native C or C++ code
- **Generally not a good way to develop native applications**
  - Pre Android 2.3, NDK could not be used to develop native applications
  - Post Android 2.3 implements NativeActivity, but Services and Content providers cannot be accessed natively

NDK includes support for the following APIs:

- libc (C library) headers
- libm (math library) headers
- JNI interface headers
- libz (Zlib compression) headers
- liblog (Android logging) header
- OpenGL EoS 1.1 and OpenGL ES 2.0 (3D graphics libraries) headers
- libjnigraphics (Pixel buffer access) header (for Android 2.2 and above).
- A Minimal set of headers for C++ support
- OpenSL ES native audio libraries
- Android native application APIS

You should avoid linking against other libraries included in Android. If you *really* need something, link it statically or otherwise build it into your application.



Why

Good reasons to use the NDK:

- Native Libraries
- CPU Intensive
- IO Intensive

Marginal reasons to use the NDK:

- You prefer coding in C/C++

Examples:

- Codec encode/decode
- Graphics
- Sound
- Games - All of the above

How

1. Start with your project
2. Download and install NDK
3. Add JNI function stubs to your Java code
4. Add a 'jni' directory to your project
5. Generate header files for your native code
6. Write native code
7. Create Makefiles
8. Build native code
9. Build and install APK
10. Debug it (*you* can skip this, *your* code will be perfect)

Start with your Project

Our initial project for this example consists of a single Activity, and two buttons that do nothing:

```
public class PiNul extends Activity {
    Button pi_button, nul_button;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        pi_button = (Button) this.findViewById(R.id.pi_button);
        pi_button.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(PiNul.this, "Doing Nothing.", Toast.LENGTH_LONG).show();
            }
        });

        nul_button = (Button) this.findViewById(R.id.nul_button);
        ...
        });
    }
}
```

Download and Install NDK

- Download from here: http://developer.android.com/sdk/ndk
- Uncompress it somewhere, and add the NDK to your path

I added the following to my $HOME/.bashrc

```
# Add Android NDK
if [ -d "$HOME/tools/android-ndk-r5b" ] ; then
    PATH="$HOME/tools/android-ndk-r5b:$PATH"
fi
```

Add JNI Function Stubs

Add code in the Java application to load the native library, and create stubs for the native functions.

```
// Load our native library
static {
    System.loadLibrary("pinul");
}

// Function stubs for native code
private native int pi(int digits);
private native void nul();
```

For the purpose of making a simple example, we've added this code to the PiNul activity directly. We've also added some code to call the functions when the buttons are pressed (not shown).

Add a 'jni' Directory

```
$ cd $WORKSPACE/PiNul
$ mkdir jni
$ ls
AndroidManifest.xml  assets  bin  default.properties  gen  proguard.cfg  res  src
```

Generate Header Files

Use the *javah* utility to generate the header file for your native functions. The *javah* command operates on class files in your project, so you need to have recently built your project before running this command.

```
$ cd $WORKSPACE/PiNul/bin
$ javah -o ../jni/pinul.h ca.littlebox.misc.pinul.PiNul
$ cat ../jni/pinul.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class ca_littlebox_misc_pinul_PiNul */

...

JNIEXPORT jint JNICALL Java_ca_littlebox_misc_pinul_PiNul_pi
  (JNIEnv *, jobject, jint);
...
```

Write Native Code

Create pinul.c in the 'jni' directory that define the functions we will use from our application.

```
#include <stdlib.h>
#include <android/log.h>
#include "pinul.h"

extern calculate_pi(int digits);

JNIEXPORT jint JNICALL Java_ca_littlebox_misc_pinul_PiNul_pi(
    JNIEnv *env, jobject jobj, jint digits) {
    return (jint) calculate_pi((int)digits);
}

// Le deréférencement d'un nul est nul.
JNIEXPORT void JNICALL Java_ca_littlebox_misc_pinul_PiNul_nul(
    JNIEnv *env, jobject jobj) {
    __android_log_write(ANDROID_LOG_DEBUG, "PiNul", "About to crash.");
    int *p = (int *) NULL;
    *p = 0xdeadd00d;
}
```

Create Makefiles

Create Android.mk file in the 'jni' directory:

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

LOCAL_MODULE := PiNul
LOCAL_CFLAGS := -g -W
LOCAL_LDLIBS := -llog
LOCAL_ARM_MODE := arm
LOCAL_SRC_FILES := pinul.c pi8.c

include $(BUILD_SHARED_LIBRARY)
```

Optionally, create the Application.mk file:

```
# Compile a fat binary for all supported ABIs
APP_ABI := armeabi armeabi-v7a
```

Build It

Compile the Native code:

```
$ cd $WORKSPACE/PiNul
$ ndk-build
Compile arm    : PiNul <= pinul.c
Compile arm    : PiNul <= pi8.c
SharedLibrary  : libPiNul.so
Install        : libPiNul.so => libs/armeabi/libPiNul.so
Compile arm    : PiNul <= pinul.c
Compile arm    : PiNul <= pi8.c
SharedLibrary  : libPiNul.so
Install        : libPiNul.so => libs/armeabi-v7a/libPiNul.so
```

- Build the project from Eclipse or the CLI.
- Run it.
- You're done! ...almost

Debug It

So... you have a bug after all.

First, you'll need to enable debugging

- Modify your manifest to set android:debuggable attribute to "true", then rebuild normally.
- This will automatically add the necessary debug flags to your build (try: *ndk-build V=1*)

Next, launch the application and start the gdb. You may need to add a delay to your program if it crashes shortly after launch.

```
ndk-gdb --verbose
```

The following will also launch the application for you.

```
ndk-gdb --verbose --start
```

Caveats

- No support inside Eclipse for NDK
- Supporting multiple architectures
- Not all JNI functions are created equal
- Native support for graphics and audio

No Support Inside Eclipse for NDK

Sadly, the ADT plugin for Eclipse doesn't know how to build using the NDK. Worse yet, it doesn't check to see if the NDK libraries have changed (requiring the APK to be rebuild and reinstalled).

It can also be annoying to switch back and forth during the test, debug, build cycle.

The best alternative is to do everything from the command line.

```
$ cd $WORKSPACE/PiNul
$ android update project -p . -s
$ ant clean
$ ndk-build
$ ant install
$ ndk-gdb --verbose --start
```

Supporting Multiple Architectures

ARMv5TE - Will run on all ARM based Android devices (the official ones)

- Minimum requirement for Android
- Support for Thumb-1 instructions
- Lacks support for floating point acceleration

ARMv7-A - Will run on ARM Cortex-A Android phone

- Support for Thumb-2 instructions
- Support for FPU (VFPv4) instructions (floats not doubles!)
- Support for NEON (Advanced SIMD extensions) instructions. HW support for NEON is optional. Software must check at runtime.

x86 - Beta status

- Prebuild toolchain not included in the NDK

Existing Devices

Based on http://en.wikipedia.org/wiki/Comparison_of_Android_devices

ARMv5TE (ARM9, ARM10, ARM11) - Generally early or low end Android phones

- HTC Dream/G1, Magic, Hero, Aria
- Samsung M900 Moment
- Samsung Galaxy Ace
- Cherry Mobile Obrit, Cosmo
- ...

ARMv7-A (Cortex-A series) - Generally higher end Android phones

- Motorolo Droid, Milestone (OMAP3430)
- HTC Nexus One, Incredible S, Nexus S, Inspire
- Sony Ericsson Xperia Arc
- ...

x86 - Few official devices, mostly tablets or netbooks

- Unofficial community ports for a large number of devices (http://www.android-x86.org/)

Not all JNI Operations are Created Equal

In our trivial example, we didn't actually transfer much data. In a real application you would. To do so, we need to use some JNI operations to get and put data.

When transferring large blocks of data between Java and C, it is most efficient if you can directly access the underlying physical buffer.

You have a few options:

- **GetDirectBufferAddress() is optimal if you can use one of the java.nio.* types such as ByteBuffer or ShortBuffer**
    - Look at *VorbisDecoder.java* and *VorbisDecoder.cpp* in libgdx

- **Sadly, this doesn't work for if need to work with native Java types like short[] or byte[]**
  - Why would you need to? Well, the Android AudioTrack for example only accepts arrays of shorts
  - Converting from ShortBuffer to short[] is CPU intensive
  - In these cases use GetPrimitiveArrayCritical() & ReleasePrimitiveArrayCritical()
  - This **may** imply a copy, but is better than the alternative
  - A better solution would be to playback audio directly using the native audio libraries (OpenSL ES), but this is only available in Android 2.3

Here's a simple example that does both (for Badlogic Games' libgdx):

```
JNIEXPORT jint JNICALL
Java_com_badlogic_gdx_utils_BufferUtils_copyJni___3FLjava_nio_Buffer_2II
  (JNIEnv *env, jclass, jfloatArray src, jobject dst, jint numFloats,
jint offset )
{
   float* pDst = (float*)env->GetDirectBufferAddress( dst );
   float* pSrc = (float*)env->GetPrimitiveArrayCritical(src, 0);
   memcpy( pDst, pSrc + (offset << 2), numFloats << 2 );
   env->ReleasePrimitiveArrayCritical(src, pSrc, 0);
   return (int)pDst;
}
```

Mario Zechner from Badlogic Games has also written some excellent blog posts discussing the performance of direct buffer bulk operations in Android. Read these!

- **Libvorbis on Android**
  - http://www.badlogicgames.com/wordpress/?p=451
- **Libgdx, MD5 and direct Buffer Madness**
  - http://www.badlogicgames.com/wordpress/?p=904
- **Android direct Buffers revisited**
  - http://www.badlogicgames.com/wordpress/?p=1755
- **GetDirectBufferAddress & GetPrimitiveArrayCritical Order**
  - http://www.badlogicgames.com/wiki/index.php/GetDirectBufferAddress_%26_G

NDK Functionality by Android Release

not exhaustive

- NDK is available for Android 1.5 or later
- OpenGL ES 1.X support in Android 1.6 or later
- OpenGL ES 2.0 support in Android 2.0 or later
- android.graphics.Bitmap access in Android 2.2 or later
- GDB debugging is available only on production devices running Android 2.2 or later (threaded debugging available in Android 2.3)
- Native Audio (OpenSL ES) support added in Android 2.3
- Native Activity support added in Android 2.3 (with limitations)

References

- **Android NDK**
  http://developer.android.com/sdk/ndk
- **Google Developer Blog: Gingerbread NDK Awesomeness**
  http://android-developers.blogspot.com/2011/01/gingerbread-ndk-awesomeness.htm
- **Wikipedia: JNI (Good initial introduction)**
  http://en.wikipedia.org/wiki/Java_Native_Interface
- **JNI: Programmer's Guide and Specification**
  http://java.sun.com/docs/books/jni/
- **Wikipedia: Comparison of Android devices**
  http://en.wikipedia.org/wiki/Comparison_of_Android_devices
- **Badlogic Games**
  http://www.badlogicgames.com/wordpress/