Agenda
Agenda

What’s the NDK and why use it?
Native Code From Java (JNI)
Handling Java Objects in C (JNI)
Native Activity
The NDK C/C++ Libraries
Modify the Compiler Flags
Conclusion
The Android NDK
Android* Native Development Kit

**What is it?**
Build scripts/toolkit to incorporate native code in Android Apps via JNI

**Why Use it?**
For Performance intensive tasks (Signal-processing/complex algorithms)
Games Differentiated app that takes advantage of direct CPU/HW access (e.g. using SSE3 for optimization)
Software Code reuse
Android* Native Development Kit

Create Native Codes → Create Make File → Compile Native Code → Call it from Java → Debug with GDB

APP_ABI := x86

SDK API

- Can run in x86 directly
- Adjust UI resolution
- Adjust memory allocation for higher resolution

JNI

Native Libs

C Framework

Java Framework

Android* Applications

Recompile native codes
Recompile share library
Porting ARM assembly codes

NDK API

intel Software
The NDK Platform

Optional for C++ thanks to JNI_Onload
Installing the NDK

What must I do?

Installation is as simple as extracting the archived NDK toolchain for your platform. It provides:

- A build environment
- Android headers and libraries
- Documentation and samples (these are very useful)

To install the SDK, you can refer to the second course (IntelAcademic_AndroidIntel_02_Environment)

Can work with Eclipse

You can integrate it with eclipse ADT
Installing the NDK

Change Eclipse Preferences

![Image of Eclipse Preferences window with NDK location set to C:\Android\ndk]
Adding Native Code

What must I do?

The native code must belong to a folder named `jni` at the root of your Eclipse project.

You can create a file (`Android.mk`) that will contain the build directives.

To build your application for x86 platform, you must add a file (`Application.mk`) in the `jni` folder.
Adding Native Code

Quick How To

Standard Android* Project Structure

1. Create JNI folder for native sources

Native Sources - JNI Folder

2. Reuse or create native c/c++ sources

3. Create Android.mk with build rules

NDK will automatically create ABI lib folder.

Adding Native Code

Application.mk

Allows to target x86 platform

The file needs to contain the following line

    APP_ABI := x86

You can also specify multiple targets

    APP_ABI := armeabi armeabi-v7a x86
Adding Native Code

Application.mk

The NDK will generate optimized code for all the targets

← Build ARM v5...

← Build ARM v7a...

← Build ARM x86...
Adding Native Code

Take a look at the samples

The NDK provides a lot of interesting samples that cover some features that you can expect from JNI

<table>
<thead>
<tr>
<th>Sample App</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello-jni</td>
<td>Call a native function written in C from Java.</td>
</tr>
<tr>
<td>bitmap-plasma</td>
<td>Access an Android* Bitmap object from C.</td>
</tr>
<tr>
<td>san-angeles</td>
<td>EGL and OpenGL ES code in C.</td>
</tr>
<tr>
<td>hello-gl2</td>
<td>EGL setup in Java and OpenGL ES code in C.</td>
</tr>
<tr>
<td>native-activity</td>
<td>C only OpenGL sample (no Java, uses the NativeActivity class).</td>
</tr>
<tr>
<td>native-plasma</td>
<td>C only OpenGL sample (also uses the NativeActivity class).</td>
</tr>
</tbody>
</table>
Native Code From Java
# Native Code From Java

## The Primitive Types

When working with JNI, handling types between Java and C is not direct!

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Native Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>jboolean</td>
<td>unsigned 8 bits</td>
</tr>
<tr>
<td>byte</td>
<td>jbyte</td>
<td>signed 8 bits</td>
</tr>
<tr>
<td>char</td>
<td>jchar</td>
<td>unsigned 16 bits</td>
</tr>
<tr>
<td>short</td>
<td>jshort</td>
<td>signed 16 bits</td>
</tr>
<tr>
<td>int</td>
<td>jint</td>
<td>signed 32 bits</td>
</tr>
<tr>
<td>long</td>
<td>jlong</td>
<td>signed 64 bits</td>
</tr>
<tr>
<td>float</td>
<td>jfloat</td>
<td>32 bits</td>
</tr>
<tr>
<td>double</td>
<td>jdouble</td>
<td>64 bits</td>
</tr>
<tr>
<td>void</td>
<td>void</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Native Code From Java

The keyword 'native'

In your Java file, you need to declare the C method by using the keyword native

    public native String stringFromJNI();

When you compile your C files, you create a native shared library

    libMyLib.so

Your application must load the native library

    Static {
        System.loadLibrary("MyLib");
    }
Native Code From Java

The java file

Here is how the Java file should look like

```java
TextView tv = new TextView(this);
tv.setText( stringFromJNI() );
setContentView(tv);

/* A native method that is implemented by the
public native String stringFromJNI();

/* this is used to load the 'hello-jni' library on application
static {
    System.loadLibrary("hello-jni");
}
```
Native Code From Java

Use javah

Once your functions are defined in your Java file, javah can help you to generate the prototype of the C functions.

```
javah -d jni -classpath <SDK-location>/platforms/android-<version>/android.jar:bin/classes com.example.hellojni.HelloJni
```

This command creates a file named `com_example_hellojni_HelloJni.h` with the following methods:

```
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI(JNIEnv *, jobject, jint);
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI(JNIEnv *, jobject);
```
Native Code From Java

Create your .c file

In the case of HelloJNI, the .c file is already created for you but in your projects, just copy the prototypes of the functions from the generated header and add a body.
Handling Java Objects in C
Handling Java Objects in C

Access Object Methods
Accessing the methods of a Java object in native code can be done in an easy way.

Our goal
We want to call a Java method named `callBack` from the native code.
Handling Java Objects in C

The Java file

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    int n = 40;
    long t1 = new Date().getTime();
    long resultNDK = fibonacciJNI(n);
    long t2 = new Date().getTime();
    long resultSDK = fibonacci(n);
    long t3 = new Date().getTime();
    Log.i("results", "NDK: " + resultNDK + " in " + (t2 - t1) + "msas");
    Log.i("results", "SDK: " + resultSDK + " in " + (t3 - t2) + "msas");
    TextView tv = new TextView(this);
    tv.setText("Done");
    setContentView(tv);
}

public void callBack(int value){
    Log.i("results", "Value is " + value);
}

public native long fibonacciJNI(int n);
```
Handling Java Objects in C

The C file

```c
int fibonacci(int n)
{
    if(n<=1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

void callMethod(JNIEnv* env, jobject thiz, int val){
    jclass c = (*env)->GetObjectClass(env, thiz);
    jmethodID methID = (*env)->GetMethodID(env, c, "callBack", "(I)V");
    if (methID == 0)
        return;
    (*env)->CallVoidMethod(env, thiz, methID, val);
}

jlong Java_com_example_hellojni_HelloJni_fibonacciJNI(JNIEnv* env, jobject thiz, jint n)
{
    int val = fibonacci(n);
    callMethod(env, thiz, val);
    return val;
}
```
Handling Java Objects in C

Compile the c file

cedric@cedric-zenbook ~/Documents/workspace/Android/HelloJni $ ndk-build
Gdbserver : [x86-4.6] libs/x86/gdbserver
Gdbsetup  : libs/x86/gdb.setup
Compile x86 : hello-jni <= hello-jni.c
SharedLibrary : libhello-jni.so
Install     : libhello-jni.so => libs/x86/libhello-jni.so

Execute

We can see that using native code to compute an unoptimized Fibonacci number is way faster than in standard Java.
Handling Java Objects in C

Possibilities

JNI offers a set of features that let you manipulate Java objects in native code. You can:

- Get local or global references on Java objects
- Call Java methods
- Access fields
- Handle Java exceptions
- Etc.

More Informations

http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/functions.html
Native Activity
Native Activity

Definition
All the code is in the jni folder, no Java class
Entry point: android_main()
Event loop to process input data and messages

Want to know more?
Take a look at the Native Activity Sample in the NDK

Native activity is used in game development. It can be a solution when your application needs a lot of performance.
The NDK C/C++ Libraries
The Bionic C Library

Just include the headers in your C files and compile with ndk-build

- Lighter than standard GNU C Library
- Not POSIX compliant
- pthread support included, but limited
- No System-V IPCs
- Access to Android* system properties
Using another C++ library

By default, libstdc++ is used but some features are missing (Standard C++ Library support, C++ exceptions support, RTTI support)

Fortunately the NDK allows you to change the library

<table>
<thead>
<tr>
<th>library</th>
<th>system</th>
<th>gabi++</th>
<th>stlport</th>
<th>gnustl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

You can select the library you want to compile with in your Application.mk file
The NDK C++ Library

In the Application.mk

Postfix the runtime with _shared or _static

APP_STL := gnustdl_static

To enable exceptions and RTTI support

LOCAL_CPP_FEATURES += exceptions rtti

Google is very active on the NDK features development.
Try to stay in touch to be informed!
ARM Native Applications
ARM Native Applications

_libhoudini_ allows to run native ARM applications on x86 architecture

This feature is already installed on Intel phones and Intel Android tablets.

Keep in mind that compiling for a specific platform is always better. And it's very simple:

```
APP_ABI := x86
```
ARM* Native Applications

Memory alignment is different between ARM* and x86. Take a look at a simple structure:

```c
struct TestStruct {
    int mVar1;
    long long mVar2;
    int mVar3;
};
```

Even if x86 platforms can run ARM* native code thanks to libhoudini, it's always better to re-compile ARM* native code for x86.
Modify the Compiler Flags
Modify the Compiler Flags

You can change the compiler options in your Application.mk file to improve the performances

ifeq ($(TARGET_ARCH_ABI),x86)
    LOCAL_CFLAGS += -O3 -ffast-math -mtune=atom -msse3 -mfpmath=sse
else
    LOCAL_CFLAGS += ...
endif
You can use vectorization

SIMD instructions are available on Android IA platforms

**SSE3**
- Vector size: **128 bit**
- Data types:
  - 8, 16, 32, 64 bit integer
  - 32 and 64 bit float
- VL: 2, 4, 8, 16

On ARM, you can get vectorization through the ARM NEON instructions

When dealing with background computing, you can improve the performances of your application but also the battery life by using vectorization
Conclusion
Conclusion

Why should I use the NDK?
Improve the performances when you have background computation to perform
Vectorize your code!

When should I use the NDK?
Signal processing, Image processing, etc.
Video games
A good pattern

It can be interesting to use the NDK when your Java application can send data to the native code, let the native code process the data, send the data back to Java.
A bad pattern

When a lot of data transfer is needed in the same process, it's probably not a good idea to use JNI.
License Creative Commons - By 3.0

You are free:
to Share — to copy, distribute and transmit the work
to Remix — to adapt the work
to make commercial use of the work

Under the following conditions:
Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:
Waiver — Any of the above conditions can be waived if you get permission from the copyright holder.
Public Domain — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.

Other Rights — In no way are any of the following rights affected by the license:
Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
The author's moral rights;
Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

http://creativecommons.org/licenses/by/3.0/