



Creating and Porting NDK-Based Android Applications

It has become appallingly obvious that our technology has exceeded our humanity.

—Albert Einstein

Android applications can incorporate native code using the Native Development Kit (NDK) toolset. It allows developers to reuse legacy code, program for low-level hardware, and differentiate their applications by taking advantage of features otherwise not optimal or possible.

This chapter provides an in-depth introduction on how to create NDK-based applications for the Intel architecture. It also covers the cases of porting existing NDK-based applications. It discusses in-depth the differences between the Intel compiler and the default NDK compiler, and explains how to take full advantage of the Intel NDK environment.

JNI and NDK Introduction

JNI Introduction

We know that Java applications do not run directly on the hardware, but actually run in a virtual machine. The source code of an application is not compiled to get the hardware instructions, but is instead compiled to get the interpretation of a virtual machine to execute code. For example, Android applications run in the Dalvik virtual machine; its compiled code is executable code for the Dalvik virtual machine in DEX format. This feature means that Java runs on the virtual machine and ensures its cross-platform capability: that is its “compile once, run anywhere” feature. This cross-platform capability of Java causes it to be less connected to and limits its interaction with the local machine’s various internal components, making it difficult to use the local machine instructions to utilize the performance potential of the machine. It is difficult to take advantage of locally based instructions to run a huge existing software library, and thus functionality and performance are limited.

Is there a way to make Java code and native code software collaborate and share resources? The answer is yes—by using the Java Native Interface (JNI), which is an implementation method of a Java local operation. JNI is a Java platform defined as the Java standard to interact with the code on the local platform. (It is generally known as the *host platform*. But this chapter is for the mobile platform, and in order to distinguish it from the mobile cross-development host, we call it the *local platform*.) The so-called “interface” includes two directions—one is Java code to call native functions (methods), and the other is local application calls to the Java code. Relatively speaking, the former method is used more in Android application development. This chapter therefore focuses on the approach in which Java code calls the native functions.

The way Java calls native functions through JNI is to store the local method in the form of library files. For example, on a Windows platform, the files are in .DLL file format, and on a UNIX/Linux machine the files are in .SO file format. By an internal method of calling the local library file, Java can establish close contact with the local machine. This is called the *system-level approach* for various interfaces.

JNI usually has two usage scenarios: first, to be able to use legacy code (for example C/C++, Delphi, and other development tools); second, to more directly interact with the hardware for better performance. You will see some of this as you go through the chapter.

JNI general workflow is as follows: Java initiates calls so that the local function’s side code (such as a function written in C/C++) runs. This time the object is passed over from the Java side, and run at a local function’s completion. After finishing running a local function, the value of the result is returned to the Java code. Here JNI is an adapter, mapping the variables and functions (Java methods) between the Java language and the native compiled languages (such as C/C++). We know that Java and C/C++ are very different in function prototype definitions and variable types. In order to make the two match, JNI provides a `jni.h` file to complete the mapping between the two. This process is shown in Figure 7-1.

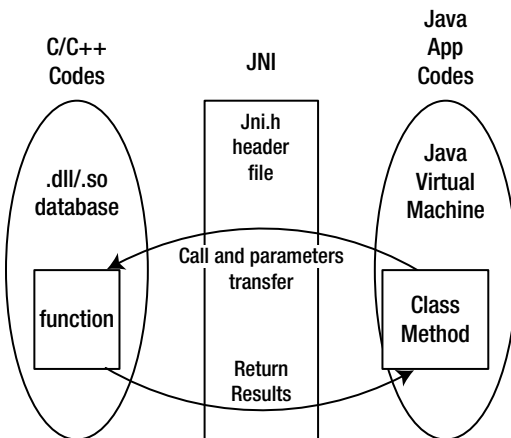


Figure 7-1. JNI General Workflow

The general framework of a C/C++ function call via a JNI and Java program (especially an Android application) is as follows:

1. The way of compiling native is declared in the Java class (C/C++ function).
2. The .java source code file containing the native method is compiled (build project in Android).
3. The javah command generates an .h file, which corresponds to the native method according to the .class files.
4. C/C++ methods are used to achieve the local method.
5. The recommended method for this step is first to copy the function prototypes into the .h file and then modify the function prototypes and add the function body. In this process, the following points should be noted:
 - The JNI function call must use the C function. If it is the C++ function, do not forget to add the extern C keyword.
 - The format of the method name should follow the following template: Java_package_class_method, namely the Java_package name classname and function method name.
6. The C or C++ file is compiled into a dynamic library (under Windows this is a .DLL file, under UNIX/Linux, it's a .SO file).

Use the `System.loadLibrary()` or `System.load()` method in the Java class to load the dynamic library generated.

These two functions are slightly different:

- `System.loadLibrary()`: Loads the default directory (for Windows, for example, this is `\System32`, `jre\bin`, and so on) under the local link library.
- `System.load()`: Depending on the local directory added to the cross-link library, you must use an absolute path.

In the first step, Java calls the native C/C++ function; the format is not the same for both C and C++. For example, for Java methods such as non-passing parameters and returning a String class, C and C++ code differs in the following ways:

C code:

```
Call function: (*env) -> <jni function> (env, <parameters>)
Return jstring: return (*env)->NewStringUTF(env, "XXX");
```

C++ code:

```
Call function: env -> <jni function> (<parameters>)
Return jstring: return env->NewStringUTF("XXX");
```

in which both Java String object NewStringUTF functions are generated by the C/C++ provided by the JNI.

Java Methods and Their Corresponding Relationship with the C Function Prototype Java

Recall that in order for Java programs to call a C/C++ function in the code framework, you use the `javah` command, which will generate the corresponding `.h` file for native methods according to the `.class` files. The `.h` file is generated in accordance with certain rules, so as to make the correct Java code find the corresponding C function to execute.

For example, for the following Java code for Android:

```
public class HelloJni extends Activity
```

```
1.  {
2.      public void onCreate(Bundle savedInstanceState)
3.      {
4.          TextView tv.setText(stringFromJNI() ); // Use C function Code
5.      }
6.      public native String  stringFromJNI();
7.  }
```

For the C function `stringFromJNI()` used in line 4, the function prototype in the `.h` file generated by `javah` is:

```
1.  JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_
stringFromJNI
2.      (JNIEnv *, jobject);
```

In this regard, C source code files for the definition of the function code are roughly:

```
1.      /*
2.      ...
3.      Signature: ()Ljava/lang/String;
4.      */
5.      jstring Java_com_example_hellojni_HelloJni_stringFromJNI
(JNIEnv* env, jobject this )
6.      {
7.          ...
8.          return (*env)->NewStringUTF(env, "...");
9.      }
```

From this code you can see that the function name is quite long, but still very regular, in full accordance with the naming convention: `java_package_class_method`. The `stringFromJNI()` method in `Hello.java` corresponds to the `Java_com_example_hellojni_HelloJni_stringFromJNI()` method in C/C++.

Notice the comment for Signature: `()Ljava/lang/String;`. The parentheses `()` of `()Ljava/lang/String;` indicate that the function parameter is empty, which means, beside the two parameters `JNIEnv *` and `jobject`, there are no other parameter. `JNIEnv *` and `jobject` are two parameters that all JNI functions must have, respectively, for the jni environment and for the corresponding Java class (or object) itself. `Ljava/lang/String;` indicates that the function's return value is a Java `String` object.

Java and C Data Type Mapping

As mentioned, Java and C/C++ variable types are very different. JNI provides a mechanism to complete the mapping between Java and C/C++. The correspondence between the main types is shown in Table 7-1.

Table 7-1. *Java to C Type Mapping*

Java Type	Native Type	Description
boolean	jboolean	C/C++ 8-bit integer
byte	jbyte	C/C++ unsigned 8-bit integer
char	jchar	C/C++ unsigned 16-bit integer
short	jshort	C/C++ signed 16-bit integer
int	jint	C/C++ signed 32-bit integer
long	jlong	C/C++ unsigned 64-bit integer
float	jfloat	C/C++ 32-bit floating point
double	jdouble	C/C++ 64-bit floating point
void	void	N/A
Object	jobject	Any Java object, or does not correspond to an object of java type
Class	jclass	Class object
String	jstring	String objects
Object[]	jobjectArray	The array of any object
Boolean[]	jbooleanArray	Boolean array
byte[]	jbyteArray	Array of bits
char[]	jcharArray	Character array
short[]	jshortArray	Short integer array

(continued)

Table 7-1. *(continued)*

Java Type	Native Type	Description
int[]	jintArray	Integer array
long[]	jlongArray	Long integer array
float[]	jfloatArray	Floating point array
double[]	jdoubleArray	Double floating point array

■ **Note** The correspondence between Java types and the local (C/C++) type.

When a Java parameter is passed, the idea of using C code is as follows:

- Basic types can be used directly; for example, `double` and `jdouble` can be interoperable. Basic types are the types listed from the line `boolean` through `void` in Table 7-1. In such a type, if the user passes a `boolean` parameter into the method, there is a local method called `jboolean` corresponding to the `boolean` type. Similarly, if the local methods return a `jint`, then an `int` is returned in Java.
- Java object usage. An `Object` object has `String` objects and a generic object. The two objects are handled a little differently.
- The `String` object. The `String` object passed over by the Java program is the corresponding `jstring` type in the local method. The `jstring` type and `char *` in C are different. So if you just use it as a `char *`, an error will occur. Therefore, you need to convert `jstring` into a `char *` in C/C++ prior to use. Here we use the `JNIEnv` method for conversion.
- The `Object` object. Use the following code to get the object handler the class:

```
jclass objectClass = (env)->FindClass("com/ostrichmyself/jni/Structure");
```

Then use the following code to take required domain handler of the class:

```
jfieldID str = (env)->GetFieldID(objectClass,"nameString","Ljava/lang/
String;");
jfieldID ival = (env)->GetFieldID(objectClass,"number","I");
```

Then use the following similar code to assign value to the incoming fields of the jobject object:

```
(env)->SetObjectField(theObjet, str, (env)->NewStringUTF("my name is D:"));
(env)->SetShortField(theObjet, ival, 10);
```

- If there is no incoming object, then C code can use the following code to generate the new object:


```
jobject myNewObjet = env->AllocObject(objectClass);
```
- Java array processing. For an array type, JNI provides some operable functions. For example, `GetObjectArrayElement` can take the incoming array and use `NewObjectArray` to create an array structure.
- The principle of resource release. Objects of C/C++ new or objects of `malloc` need to use the C/C++ to release memory.
- If the new object of the `JNIEnv` method is not used by Java, it must be released.
- To convert a string object from Java to get UTF by using `GetStringUTFChars`, you need to open the memory, and you must release the memory after you are finished using `char *`. The method to use is `ReleaseStringUTFChars`.

These are brief descriptions of type mapping when Java exchanges data with C/C++. For more information on Java and C/C++ data types, refer to related Java and JNI books, documentation, and examples.

NDK Introduction

From the previous description, you know that the Java code can visit local functions (such as C/C++) using JNI. To achieve this effect, you need development tools. There is a whole set of development tools based on the core Android SDK that you can use to cross-compile Java applications to applications that can run on the target Android device. Similarly, you need cross-development tools to compile the C/C++ code into applications that can run on an Android device. This tool is the Android Native Development Kit, or Android NDK.

Prior to the NDK, third-party applications on the Android platform were developed on a special Java-based Dalvik virtual machine. The native SDK allows developers to directly access the Android system resources and use traditional C or C++ programming languages to create applications. The application package file (.apk) can be directly embedded into the local library. In short, with the NDK, Android applications originally

run on a Dalvik virtual machine can now use native code languages like C/C++ for program execution. This provides the following benefits:

- Performance improvement. It uses native code to develop the part of the program that requires high performance and directly accesses the CPU and hardware.
- The ability to reuse existing native code.

Of course, compared to the Dalvik virtual machine, using native SDK programming also has some disadvantages, such as added program complexity, difficulty in guaranteeing compatibility, the inability to access the Framework API, more difficult debugging, decreased flexibility, and so on. In addition, access to JNI incurs some additional performance overhead.

In short, NDK application development has its pros and cons. You need to use the NDK at your own discretion. The best strategy is to use the NDK to develop parts of the application for which native code will improve performance.

The NDK includes the following major components:

- Tools and a build file generate the native code libraries from C/C++. This includes a series of NDK commands, including `javah` (use the `.class` files to generate the corresponding `.h` files), `gcc` (to be described later), and other commands. It also includes the `ndk-build` executable scripts, and so on, which are covered in detail in the following sessions.
- A consistent local library will be embedded in the application package (application package files, that is, `.apk` files), which can be deployed in Android devices.
- Support for some native system header files and libraries for all future Android platforms.

The process framework of the NDK application development is shown in Figure 7-2. An Android application consists of three parts: Android application files, Java native library files, and dynamic libraries. These three parts are generated from different sources through the respective generation paths. For an ordinary Android application, the Android SDK generates Android applications files and Java native library files. The Android NDK generates the dynamic library files (the file with the `.SO` extension) using non-native code (typically C source code files). Finally the Android application files, Java library files, and native dynamic libraries are installed on the target machine, and complete collaborative applications run.

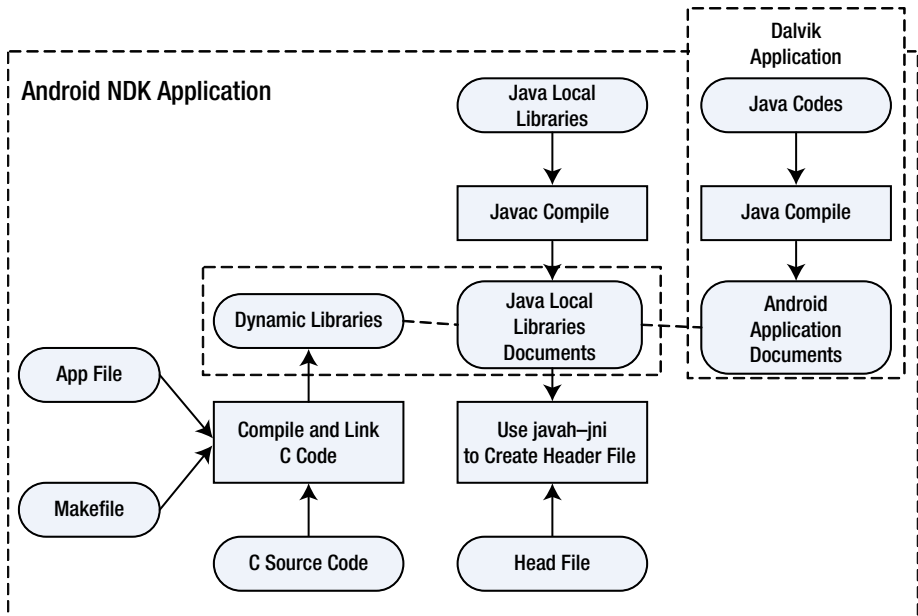


Figure 7-2. Flowchart of Android NDK Application Development

Applications projects developed by the NDK (referred to as NDK application projects) have components, as shown in Figure 7-3. In contrast to typical applications developed using the Android SDK, projects developed in the NDK add the Dalvik class code, manifest files, common resources, and also the JNI and a shared library generated by the NDK.

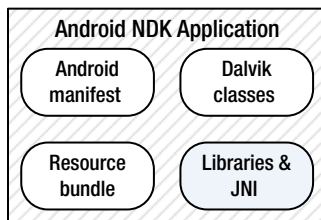


Figure 7-3. Application Components for an Android NDK Application

Android adds the NDK support in its key API version. Each version includes some new NDK features, simple C/C++, a compatible STL, hardware expansion, and so on. These features make Android more open and more powerful. The Android API and its corresponding relationship with the NDK are shown in Table 7-2.

Table 7-2. *Relationship Between Main Android API and NDK Version*

API Version	Supported NDK Version
API Level 3	Android 1.5 NDK 1
API Level 4	Android 1.6 NDK 2
API Level 7	Android 2.1 NDK 3
API Level 8	Android 2.2 NDK 4
API Level 9	Android 2.3 NDK 5
API Level 12	Android 3.1 NDK 6
API Level 14	Android 4.0.1 NDK 7
API Level 15	Android 4.0.3 NDK 8
API Level 16	Android 4.1 NDK 8b
API Level 16	Android 4.2 NDK 8d
API Level 18	Android 4.3 NDK 9b

■ **Tip** Each piece of native code generated using the Android NDK is given a matching Application Binary Interface (ABI). The ABI precisely defines how the application and its code interact with the system at runtime. The ABI can be roughly understood as similar to an ISA (instruction set architecture) in computer architecture.

A typical ABI contains the following information:

- Machine code the CPU instruction set should use.
- A runtime memory access ranking.
- The format of executable binary files (dynamic libraries, programs, and so on) as well as what type of content is allowed and supported.
- Different conventions used in passing data between the application code and systems (for example, when the function call registers and/or how to use the stack, alignment restrictions, and so on).
- Alignment and size limits of enumerated types, structure fields, and arrays.
- The available list of function symbols for application machine code at runtime usually comes from a very specific set of libraries. Each supported ABI is identified by a unique name.

Android currently supports the following ABI types:

- *armeabi*—This is the ABI name for the ARM CPU, which supports at least the ARMv5TE instruction set.
- *armeabi-v7a*—This is another ABI name for ARM-based CPUs; it extends the armeabi CPU instruction set extensions, such as Thumb-2 instruction set extensions and floating-point processing unit directives for vector floating-point hardware.
- *x86*—This is ABI name generally known for the support of x86 or IA-32 instruction set of the CPU. More specifically, its target is often referred to in the following sessions as i686 or Pentium Pro instruction set. Intel Atom processors belong to this ABI type.

These types have different compatibility. X86 is incompatible with armeabi and armeabi-v7a. The armeabi-v7a machine is compatible with armeabi, which means the armeabi framework instruction set can run on an armeabi-v7a machine, but not necessarily the other way around, because some ARMv5 and ARMv6 machines do not support armeabi-v7a code. Therefore, when you build the application, users should be chosen carefully according to their corresponding ABI machine type.

NDK Installation

Here we use NDK Windows environment as an example to illustrate the NDK software installation. The Windows NDK includes the following modules:

- Cygwin runs Linux commands in the Windows command line.
- Android NDK package, including `ndk-build` and other key commands, is the core of the NDK software; it compiles C/C++ files into .SO shared library files.
- CDT (C/C++ Development Tooling, C/C++ development tools) is an Eclipse plug-in and can compile C/C++ files into .SO shared library in Eclipse. This means you can use it to `ndk-build` replace the command-line commands.

The CDT module is not required, but does enable development in the familiar Eclipse IDE. The Cygwin module must be installed in the Windows environment, but is not required in the Linux environment. Of course, the entire development environment needs to support the Java development environment. The following sections explain the installation steps for each module separately.

Android NDK Installation

This section describes how to install the Android NDK:

1. Visit the Android NDK official web site at <http://developer.android.com/sdk/ndk/index.html> and download the latest NDK package, as shown in Figure 7-4. In this case, you click on the file `android-ndk-r8d-windows.zip` and download the files to the local directory.

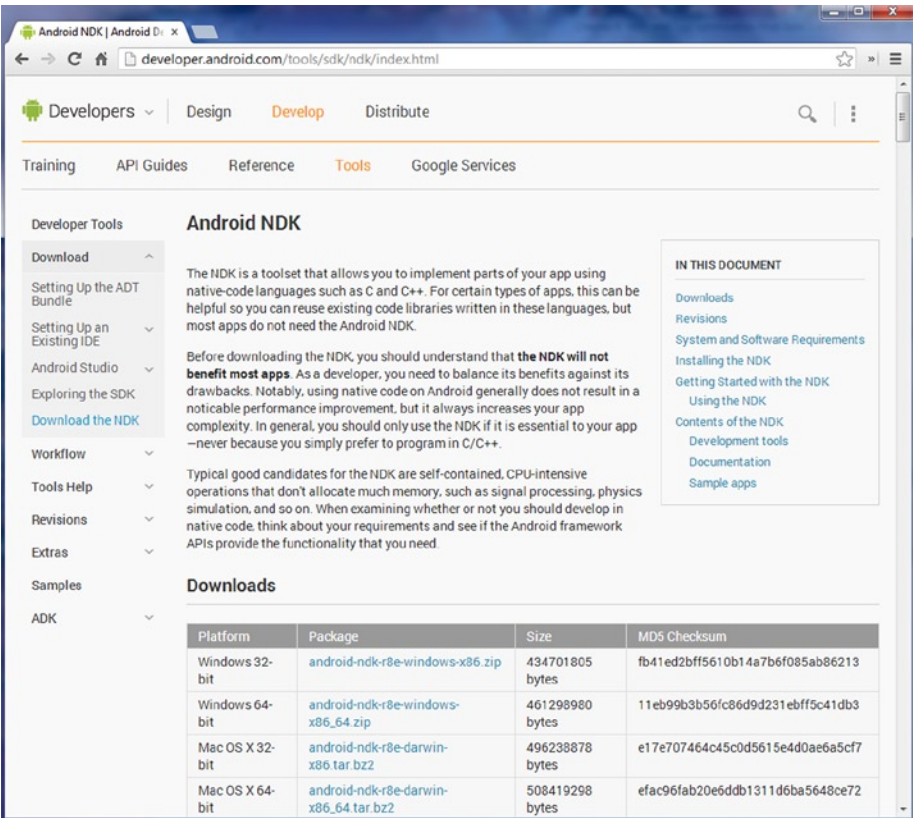


Figure 7-4. The NDK Package Download Page from the Android Official Web Site

2. Install the Android NDK.

Android NDK installation is relatively simple. All you need to do is to extract the downloaded `android-ndk-r4b-windows.zip` to a specified directory. In this case, we install Android NDK in the directory `D:\Android\android-ndk-r8d`. You need to remember this location, as it is required for the following configuration to set up the environment.

Install Cygwin

This section describes how to install Cygwin:

1. Visit Cygwin's official web site (<http://www.cygwin.com/>). Download the Cygwin software, as shown in Figure 7-5. Go to the download page, and then click on the `setup.exe` file to download and install packages.

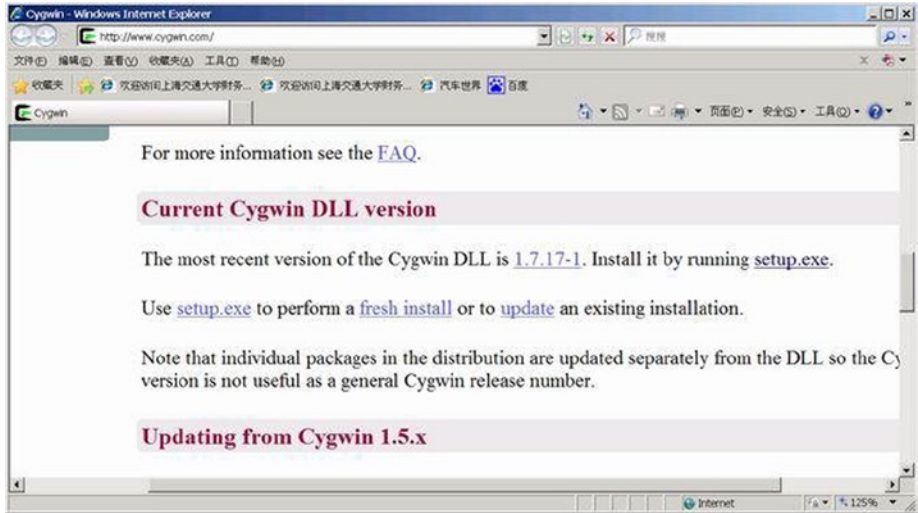


Figure 7-5. *Cygwin Download Page*

2. Double-click the downloaded `setup.exe` file to start the installation. The pop-up shown in Figure 7-6 appears.



Figure 7-6. *Cygwin Initial Install Window*

3. The installation mode selection box is shown in Figure 7-7. In this example, select Install from Internet mode.

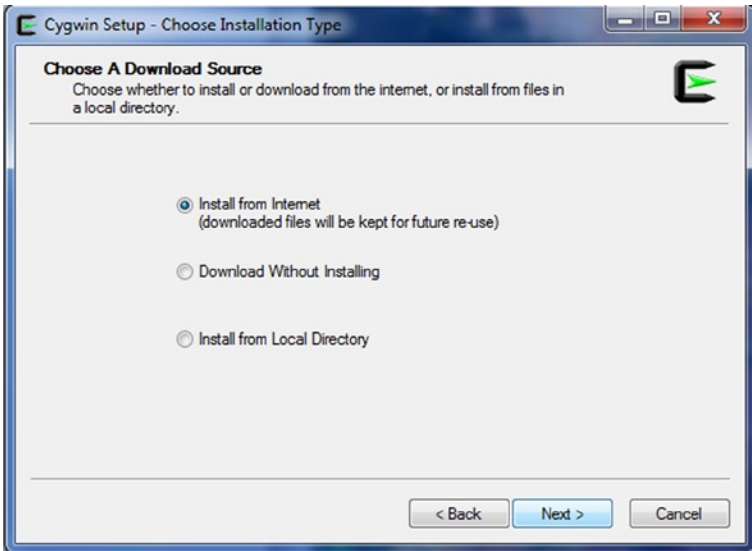


Figure 7-7. *Cygwin Install Mode Selection*

4. The display installation directory and user settings selection box is shown in Figure 7-8.

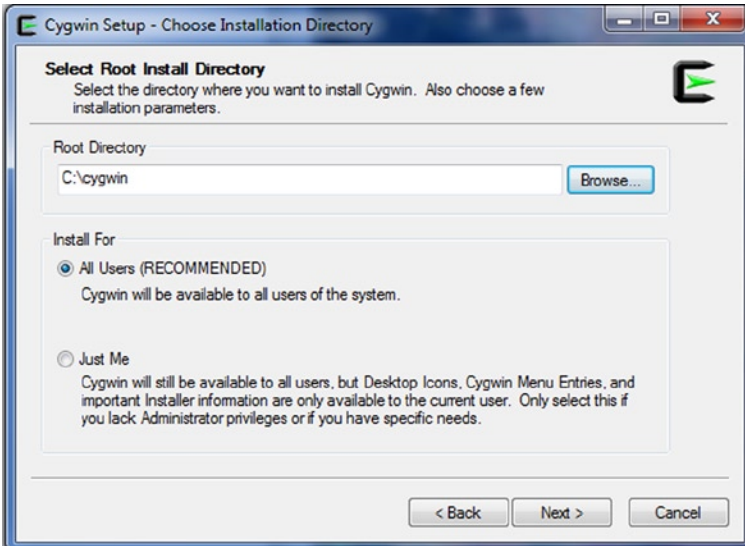


Figure 7-8. *Installation Directory and User Settings Selection*

5. You are next prompted to enter a temporary directory to store the downloaded files, as shown in Figure 7-9.

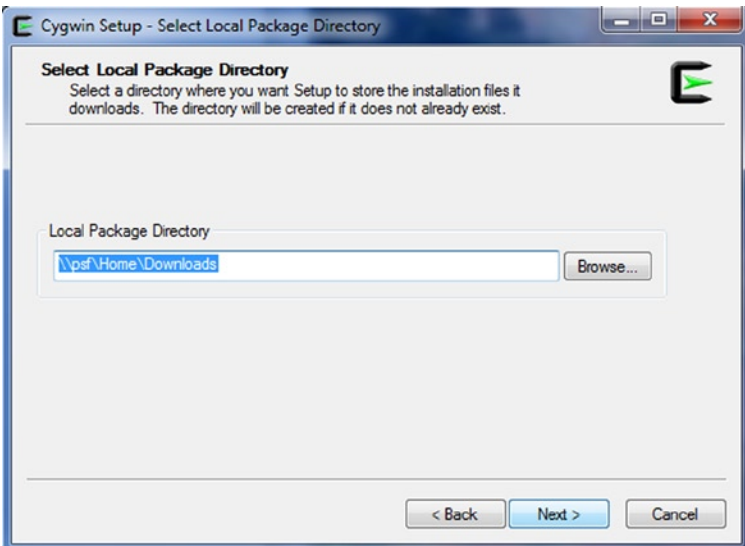


Figure 7-9. *Cygwin Temporary Directory Setting for Downloaded Files*

- Next you are prompted to select an Internet connection type, as shown in Figure 7-10. For this example, select Direct Connection.

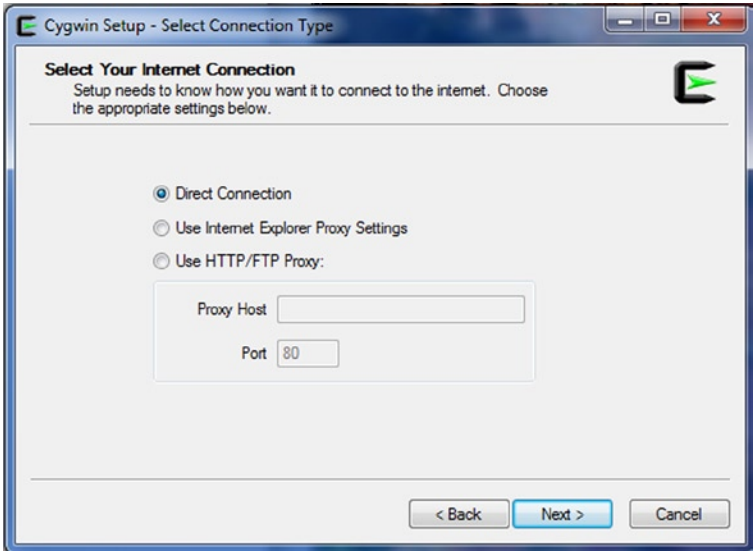


Figure 7-10. Cygwin Setup Internet Connection Type Selection

- You are now prompted to select a download mirror site, as shown in Figure 7-11.

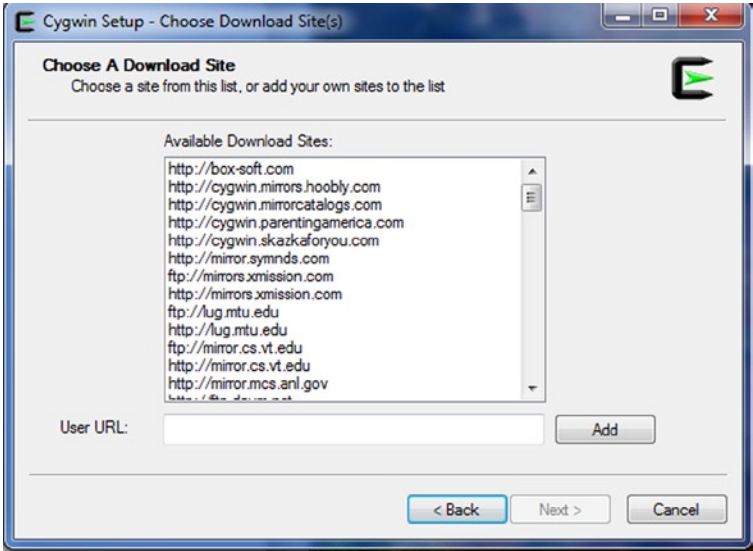
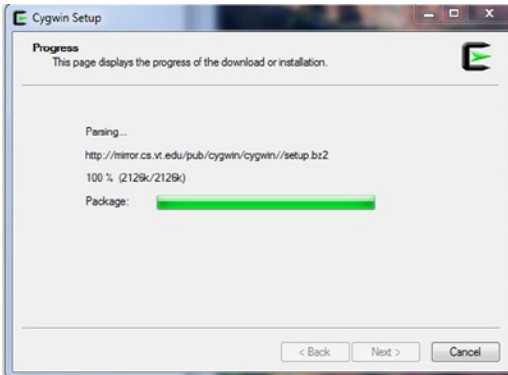
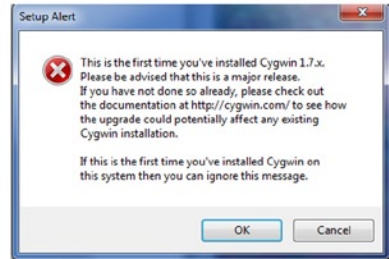


Figure 7-11. Cygwin Install: Prompt to Select Download Mirror Site

8. Start the download and install the basic parts, as shown in Figure 7-12(a). During the setup, a Setup alert will indicate that this is the first time you are installing Cygwin, as shown in Figure 7-12(b). Click OK to continue.



(a) Installation Package



(b) Setup Alert

Figure 7-12. Cygwin Installation Package Download and Install

9. Select the packages to install, as shown in Figure 7-13. The default is to install all of the packages.

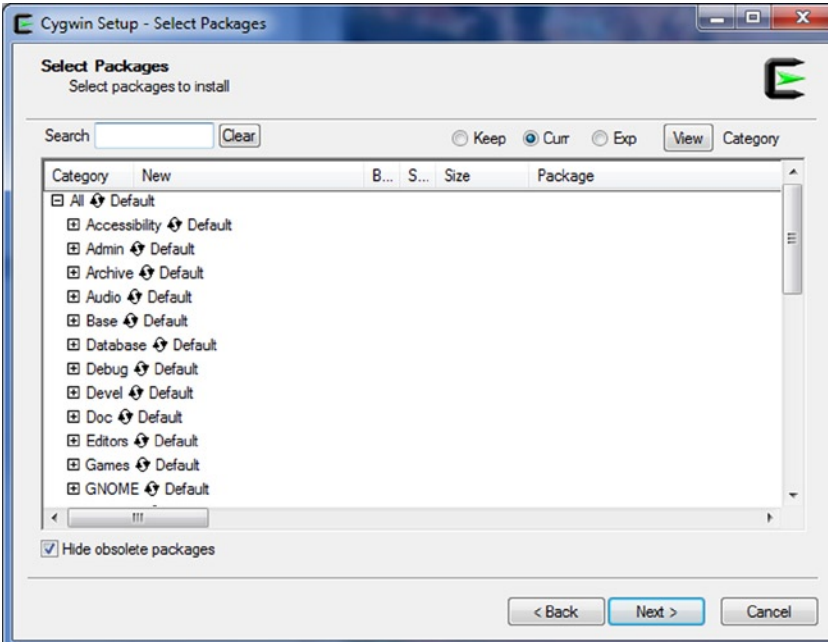


Figure 7-13. Cygwin Packages Install Selection

If you download all components, the total size is more than 3GB. This requires a very long time on normal broadband Internet speeds; it is actually not recommended to install all the components. You need to install the NDK Devel component and the Shells components, as shown in Figure 7-14.



Figure 7-14. *Cygwin Components Packages Required by NDK*

There are some tricks to the selection of Devel and Shells from the Install component packages. You can first click on the loop icon next to All; it will loop among Install, Default, and Uninstall. Set it to Uninstall State, and then click the loop icon next to the Devel and Shells entries so that it stays in the Install state. Finally, click Next to continue.

- 10. The contents of the selected components are displayed next, as shown in Figure 7-15.



Figure 7-15. *Dependency Reminder After Selecting Cygwin Component Package*

11. Start to download and install the selected components, as shown in Figure 7-16.

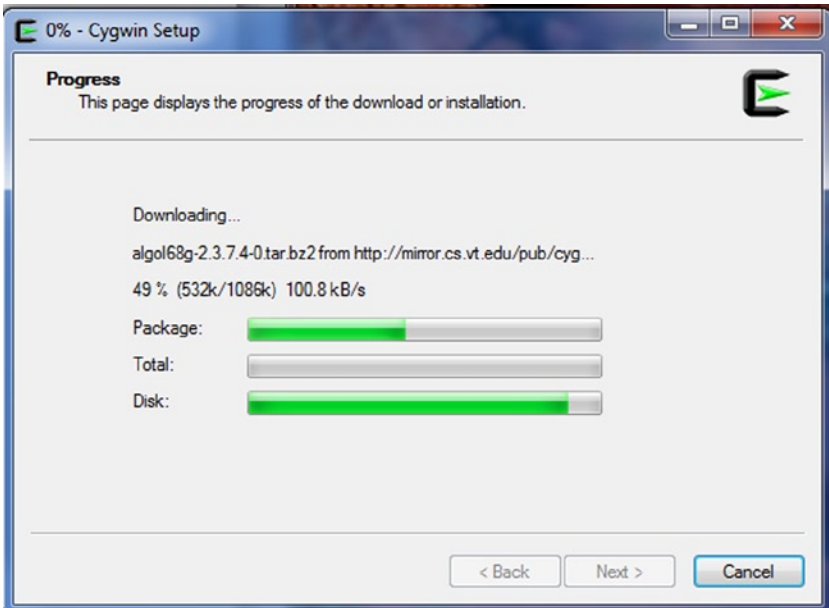
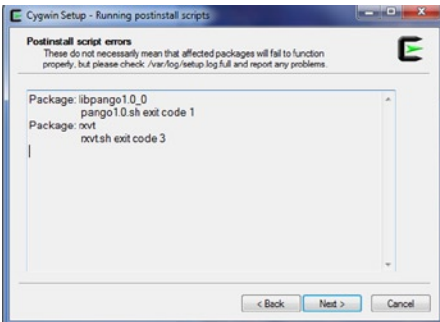
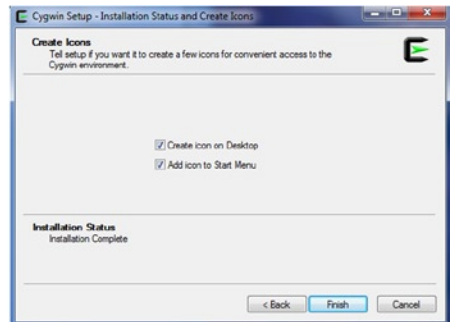


Figure 7-16. *Cygwin Download and Install Selected Components*

12. Installation is complete. Message boxes appear, as shown in Figure 7-17.



(a) Message Box after Cygwin Install



(b) Last Reminder Box

Figure 7-17. *Cygwin Reminder Boxes after Installation Is Complete*

13. Configure the Cygwin Windows path environment variable.

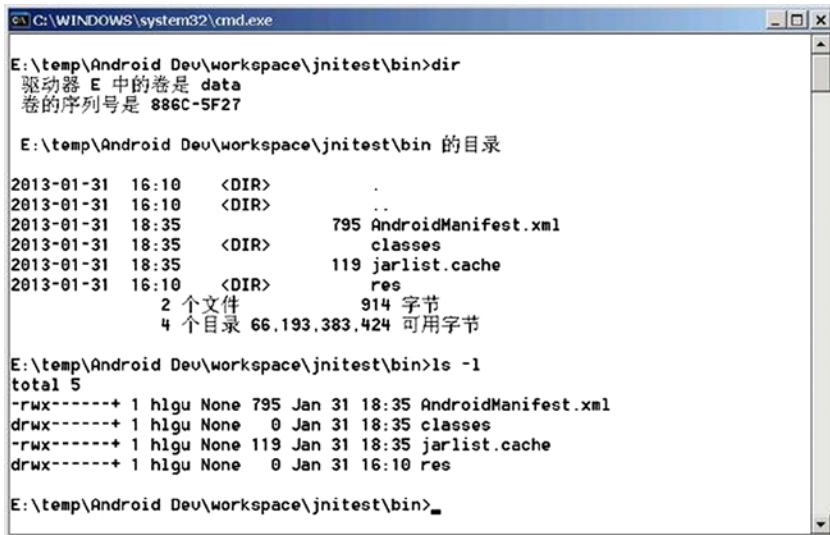
Follow these steps to add the NDK package installation directory and Cygwin bin directory to the path environment variable:

1. On the desktop, right-click My Computer and select the \Properties\Advanced\Environment Variables menu item.
2. Click System Variables in the PATH variable. Then click the Edit button in the dialog box of the [variable value] NDK package added after the installation directory, in the subdirectory build\tools\cygwin\bin.

For example, if the NDK is installed in the directory D:\Android\android-ndk-r8d and Cygwin is installed in the D:\cygwin directory, you add the path after the PATH variable, as follows:

```
PATH=...;D:\Android\android-ndk-r8d;D:\Android\android-ndk-r8d\build\tools;D:\cygwin\bin
```

After this configuration is successful, you can use the console command `cmd` under Linux commands. For example, Figure 7-18 shows a command-line window with the Windows `dir` command and the Linux `ls` command.



```
C:\WINDOWS\system32\cmd.exe
E:\temp\Android Dev\workspace\jnitest\bin>dir
驱动器 E 中的卷是 data
卷的序列号是 886C-5F27

E:\temp\Android Dev\workspace\jnitest\bin 的目录
2013-01-31 16:10 <DIR>      .
2013-01-31 16:10 <DIR>      ..
2013-01-31 18:35          795 AndroidManifest.xml
2013-01-31 18:35 <DIR>      classes
2013-01-31 18:35          119 jarlist.cache
2013-01-31 16:10 <DIR>      res
                2 个文件          914 字节
                4 个目录 66,193,383,424 可用字节

E:\temp\Android Dev\workspace\jnitest\bin>ls -l
total 5
-rwx-----+ 1 hlgu None 795 Jan 31 18:35 AndroidManifest.xml
drwx-----+ 1 hlgu None  0 Jan 31 18:35 classes
-rwx-----+ 1 hlgu None 119 Jan 31 18:35 jarlist.cache
drwx-----+ 1 hlgu None  0 Jan 31 16:10 res

E:\temp\Android Dev\workspace\jnitest\bin>
```

Figure 7-18. Command-Line Window after Installing the NDK

You configure Cygwin's internal environment variables for NDK as follows:

1. Before configuring the NDK Cygwin internal environment variables, you must run Cygwin at least once, otherwise the `\cygwin\home` directory will be empty. Click the Browse button in Windows Explorer and select the `mintty.exe` file under the `bin` subdirectory of the Cygwin installation directory (in this example, it is located at `D:\cygwin\bin`). The window is shown in Figure 7-19.

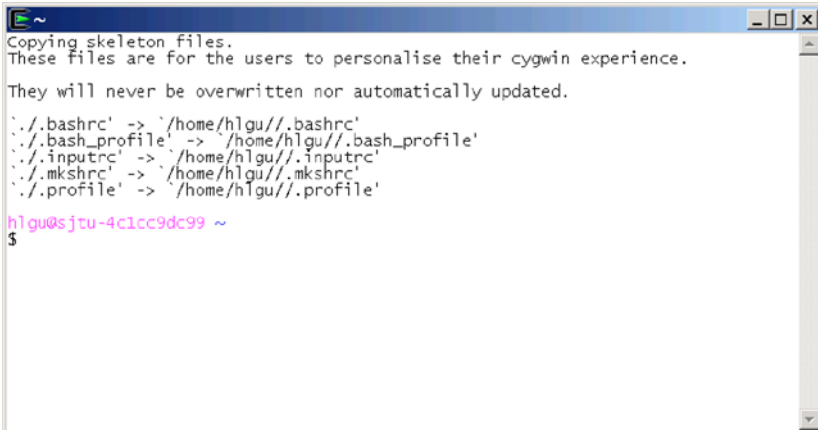


Figure 7-19. Initial Window when Starting Cygwin for the First Time

2. Then select the Windows menu `\programs\Cygwin\Cygwin terminal`. You can directly enter the Cygwin window, as shown in Figure 7-20.

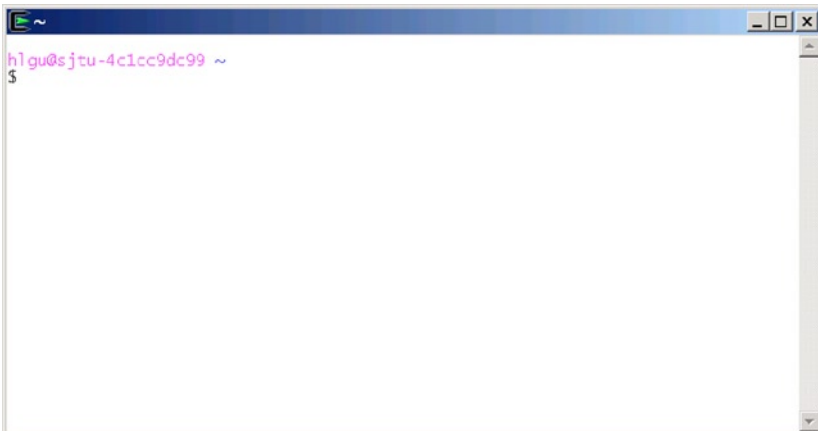


Figure 7-20. Cygwin Window if It Is Not Being Run for the First Time

This will create a username (in this case, the Windows logon username hlgw) subdirectory under `empty\cygwin\home` and generate several files in the directory.

```
D:\cygwin\home\hlgw>dir
2013-01-30  00:42                6,054 .bashrc
2013-01-30  00:52                  5 .bash_history
2013-01-30  01:09             1,559 .bash_profile
2013-01-30  00:42             1,919 .inputrc
2012-12-01  08:58             8,956 .mkshrc
2013-01-30  00:42             1,236 .profile
```

- Find `.bash_profile` in the installation directory `cygwin\home\<username>\` file. In this case, it is `D:\cygwin\home\hlgw\.Bash_profile`. To the end of the file, add the following code:

```
NDK=<android-ndk-r4b unzipped_NDK_folder>
export NDK
ANDROID_NDK_ROOT=<android-ndk-r4b unzipped_NDK_folder >
export ANDROID_NDK_ROOT
```

The line `<android-ndk-r4b unzipped_NDK_folder >` corresponds to the installation directory of the NDK package. (In this example, it's `D:\Android\android-ndk-r8d`.) Cygwin provides a directory-conversion mechanism. Add `/cygdrive/DRIVELETTER/` in front of the directory to refer to the designated directory in the drive. Here, `DRIVELETTER` is the driver letter of the directory. Consider this example:

```
NDK= /cygdrive/d/Android/android-ndk-r8d
export NDK
ANDROID_NDK_ROOT=/cygdrive/d/Android/android-ndk-r8d
export ANDROID_NDK_ROOT
```

- Determine whether the command can be run by testing the `make` command.

```
C:\Documents and Settings\hlgw>make -v
GNU Make 3.82.90
Built for i686-pc-cygwin
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

If you see this output, it means the make command is running normally. Make sure the version of make is 3.8.1 or above, because all examples in this session need v3.8.1 or above to be able to be compiled successfully.

Now you can test the gcc, g+, gcj, and gnat commands:

```
C:\Documents and Settings\hlg>gcc -v
Access denied.
C:\Documents and Settings\hlg>g++ -v
Access denied.
C:\Documents and Settings\hlg>gcj
Access denied
C:\Documents and Settings\hlg>gnat
Access denied.
```

If you get the Access denied message, you need to continue the following steps. Otherwise, the installation is completed successfully.

5. Under the bin directory of Cygwin, delete the gcc.exe, g++.exe, gcj.exe, and gnat.exe files.
6. Under the same directory, select the needed gcc, g++, gcj, and gnat files that match the version. For example, version 4 corresponds to gcc-4.exe, g++-4.exe, gcj-4.exe, and gnat-4.exe. Make copies of those files and rename the copied files gcc.exe, g++.exe, gcj.exe, and gnat.exe.
7. Now test again to see if gcc and the other commands can run:

```
C:\Documents and Settings\hlg> gcc -v
```

Using built-in specifications, you can see which commands are available:

```
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-pc-cygwin/4.5.3/lto-wrapper.exe
Target: i686-pc-cygwin
Configured with: /gnu/gcc/releases/respins/4.5.3-3/gcc4-4.5.3-3/src/gcc-4.5.3/co
nfigure --srcdir=/gnu/gcc/releases/respins/4.5.3-3/gcc4-4.5.3-3/src/gcc-4.5.3 --
prefix=/usr --exec-prefix=/usr --bindir=/usr/bin --sbindir=/usr/sbin
--libexecdi
r=/usr/lib --datadir=/usr/share --localstatedir=/var --sysconfdir=/etc
--dataroot
tdir=/usr/share --docdir=/usr/share/doc/gcc4 -C --datadir=/usr/share
--infodir=/
usr/share/info --mandir=/usr/share/man -v --with-gmp=/usr --with-mpfr=/usr
--ena
```

```

ble-bootstrap --enable-version-specific-runtime-libs --libexecdir=/usr/lib
--ena
ble-static --enable-shared --enable-shared-libgcc --disable-__cxa_atexit
--with-
gnu-ld --with-gnu-as --with-dwarf2 --disable-sjlj-exceptions --enable-
languages=
ada,c,c++,fortran,java,lto,objc,obj-c++ --enable-graphite --enable-lto
--enable-
java-awt=gtk --disable-symvers --enable-libjava --program-suffix=-4
--enable-lib
gomp --enable-libssp --enable-libada --enable-threads=posix --with-arch=i686
--w
ith-tune=generic --enable-libgcsj-sublibs CC=gcc-4 CXX=g++-4 CC_FOR_
TARGET=gcc-4
CXX_FOR_TARGET=g++-4 GNATMAKE_FOR_TARGET=gnatmake GNATBIND_FOR_
TARGET=gnatbind -
-with-ecj-jar=/usr/share/java/ecj.jar
Thread model: posix
gcc version 4.5.3 (GCC)

```

C:\Documents and Settings\hlg>g++ -v

Using built-in specifications, like gcc, you can see which commands are available:

```

COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-pc-cygwin/4.5.3/lto-wrapper.exe
Target: i686-pc-cygwin
Configured with: /gnu/gcc/releases/respins/4.5.3-3/gcc4-4.5.3-3/src/gcc-
4.5.3/co
nfigure --srcdir=/gnu/gcc/releases/respins/4.5.3-3/gcc4-4.5.3-3/src/gcc-
4.5.3 --
prefix=/usr --exec-prefix=/usr --bindir=/usr/bin --sbindir=/usr/sbin
--libexecdi
r=/usr/lib --datadir=/usr/share --localstatedir=/var --sysconfdir=/etc
--dataroo
tdir=/usr/share --docdir=/usr/share/doc/gcc4 -C --datadir=/usr/share
--infodir=/
usr/share/info --mandir=/usr/share/man -v --with-gmp=/usr --with-mpfr=/usr
--ena
ble-bootstrap --enable-version-specific-runtime-libs --libexecdir=/usr/lib
--ena
ble-static --enable-shared --enable-shared-libgcc --disable-__cxa_atexit
--with-
gnu-ld --with-gnu-as --with-dwarf2 --disable-sjlj-exceptions --enable-
languages=
ada,c,c++,fortran,java,lto,objc,obj-c++ --enable-graphite --enable-lto
--enable-

```



```
java-awt=gtk --disable-symvers --enable-libjava --program-suffix=-4
--enable-lib
gomp --enable-libssp --enable-libada --enable-threads=posix --with-arch=i686
--w
ith-tune=generic --enable-libgcj-sublibs CC=gcc-4 CXX=g++-4 CC_FOR_
TARGET=gcc-4
CXX_FOR_TARGET=g++-4 GNATMAKE_FOR_TARGET=gnatmake GNATBIND_FOR_
TARGET=gnatbind -
-with-ecj-jar=/usr/share/java/ecj.jar
Thread model: posix
gcc version 4.5.3 (GCC)
```

```
C:\Documents and Settings\hlgu>gcj
gcj: no input files
```

```
C:\Documents and Settings\hlgu>gnat
GNAT 4.5.3
Copyright 1996-2010, Free Software Foundation, Inc.
```

List of available commands

gnat bind	gnatbind
gnat chop	gnatchop
gnat clean	gnatclean
gnat compile	gnatmake -f -u -c
gnat check	gnatcheck
gnat sync	gnatsync
gnat elim	gnatelim
gnat find	gnatfind
gnat crunch	gnatkr
gnat link	gnatlink
gnat list	gnatls
gnat make	gnatmake
gnat metric	gnatmetric
gnat name	gnatname
gnat preprocess	gnatprep
gnat pretty	gnatpp
gnat stack	gnatstack
gnat stub	gnatstub
gnat xref	gnatxref

Commands find, list, metric, pretty, stack, stub and xref accept project file sw

itches -vPx, -Pprj and -Xnam=val

8. Finally, check out the NDK core command `ndk-build` script to see if it can run.

```
C:\Documents and Settings\hlg>ndk-build
Android NDK: Your Android application project path contains spaces:
'C:/./ Settings/'
Android NDK: The Android NDK build cannot work here. Please move your
project to a different location.
D:\Android\android-ndk-r8d\build\core\build-local.mk:137: *** Android NDK:
Aborting. Stop.
```

If your output looks like this, it indicates that the Cygwin and NDK have been installed and configured successfully.

Install CDT

CDT is an Eclipse plug-in that compiles C code into .SO shared libraries. In fact, after installing the Cygwin and NDK module, you can compile C code into .SO shared libraries at the command line, which means the core component of Windows NDK is already installed. If you still like using the Eclipse IDE rather than a command-line compiler to compile the local library, you need to install the CDT module; otherwise, skip this step and move ahead to the NDK examples.

If you need to install CDT, use the following steps:

1. Visit Eclipse's official web site at <http://www.eclipse.org/cdt/downloads.php> to download the CDT package. As shown on the download page in Figure 7-21, you can click to download a version of the software. In this case, click `cdt-master-8.1.1.zip` to start the download.

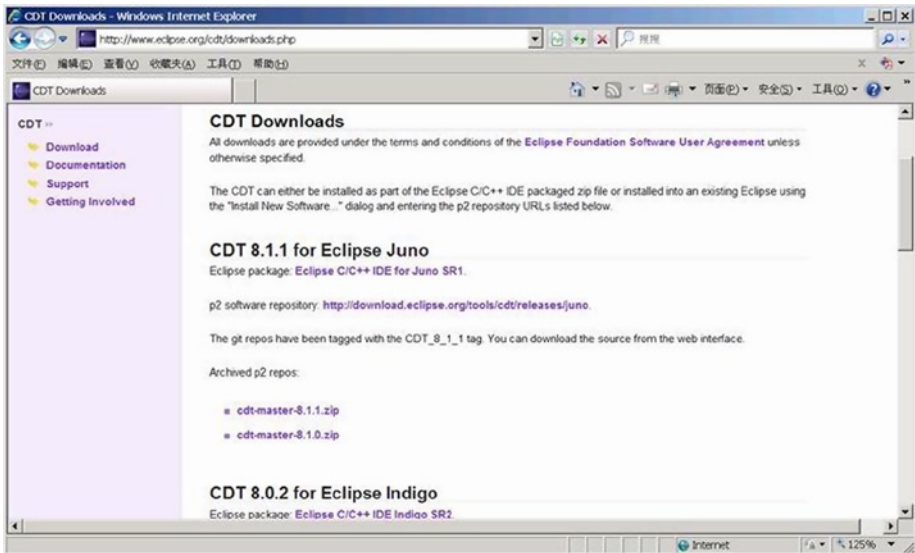


Figure 7-21. CDT Download Page

2. Start Eclipse. Select menu \HELP\Install new software and start to install CDT.
3. In the pop-up Install dialog box, click Add, as shown in Figure 7-22.

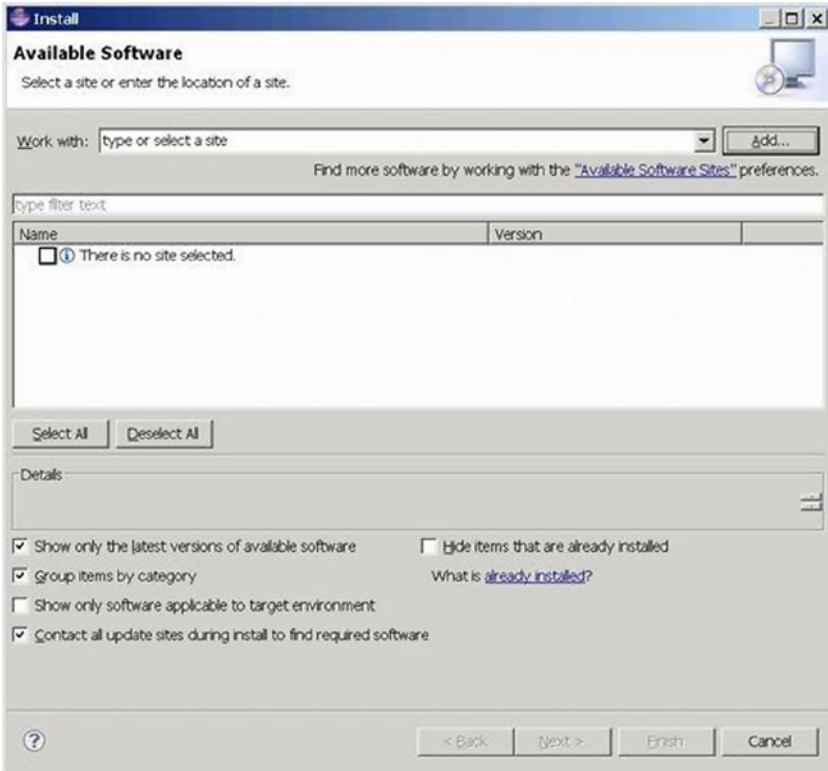


Figure 7-22. Eclipse Install Software Dialog Box

4. In the pop-up Add Repository dialog box, enter a name for Name and a software download web site address in Location. You can enter the local address or the Internet address. If you're using an Internet address, Eclipse will go to the Internet to download and install the package, while the local address will direct Eclipse to install the software from the local package. Enter the local address; then you can click the Archive button in the pop-up dialog box and enter the directory and filename for the downloaded cdt-master-8.1.1.zip file, as shown in Figure 7-23. If the file is downloaded from the Internet, the address is <http://download.eclipse.org/tools/cdt/releases/galileo/>.

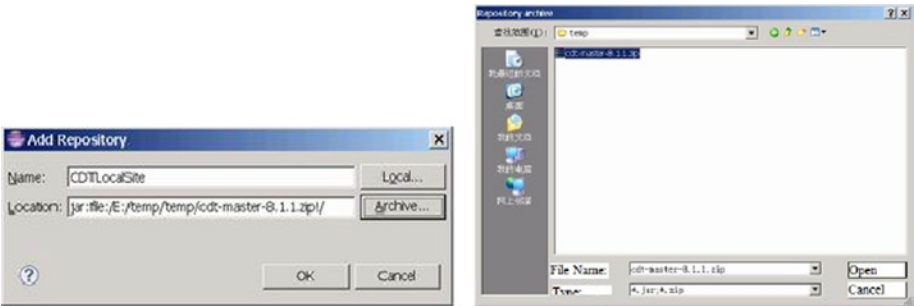


Figure 7-23. Dialog Box of Eclipse Software Update Install Address

5. After returning to the Install dialog box, click to select the software components that need to be installed, as shown in Figure 7-24.

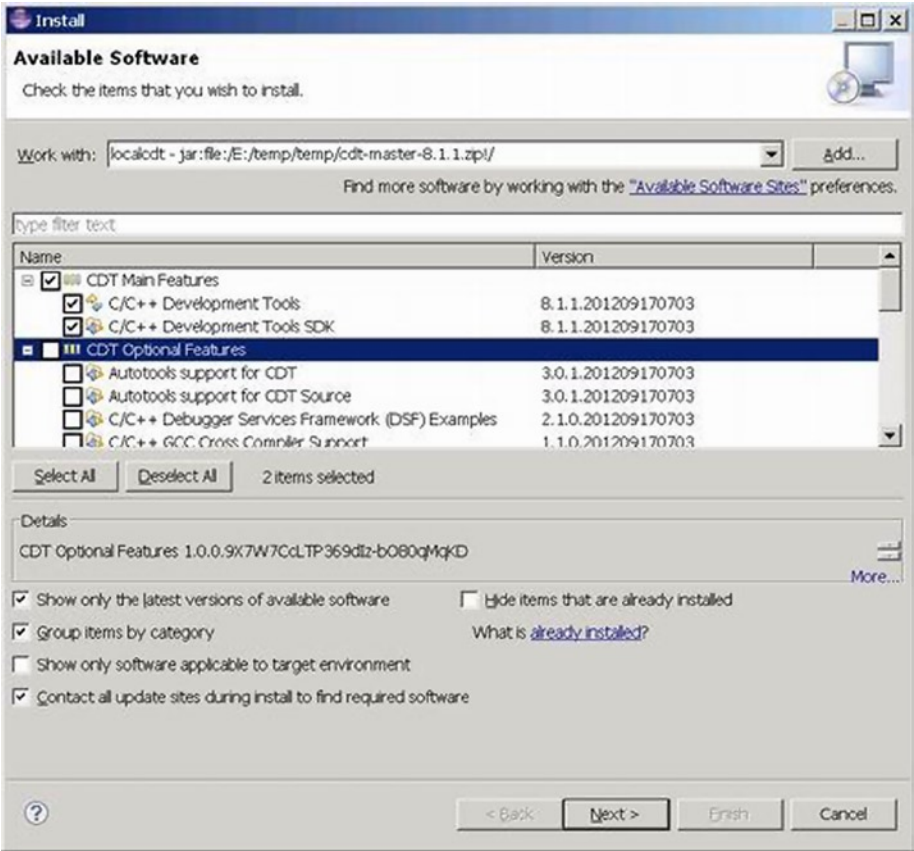


Figure 7-24. Selection Box for CDT for Components to Install

Of the components list, the CDT Main Feature is the required component. In this example, we only select this component.

6. A list of detailed information about CDT components to install is displayed, as shown in Figure 7-25.

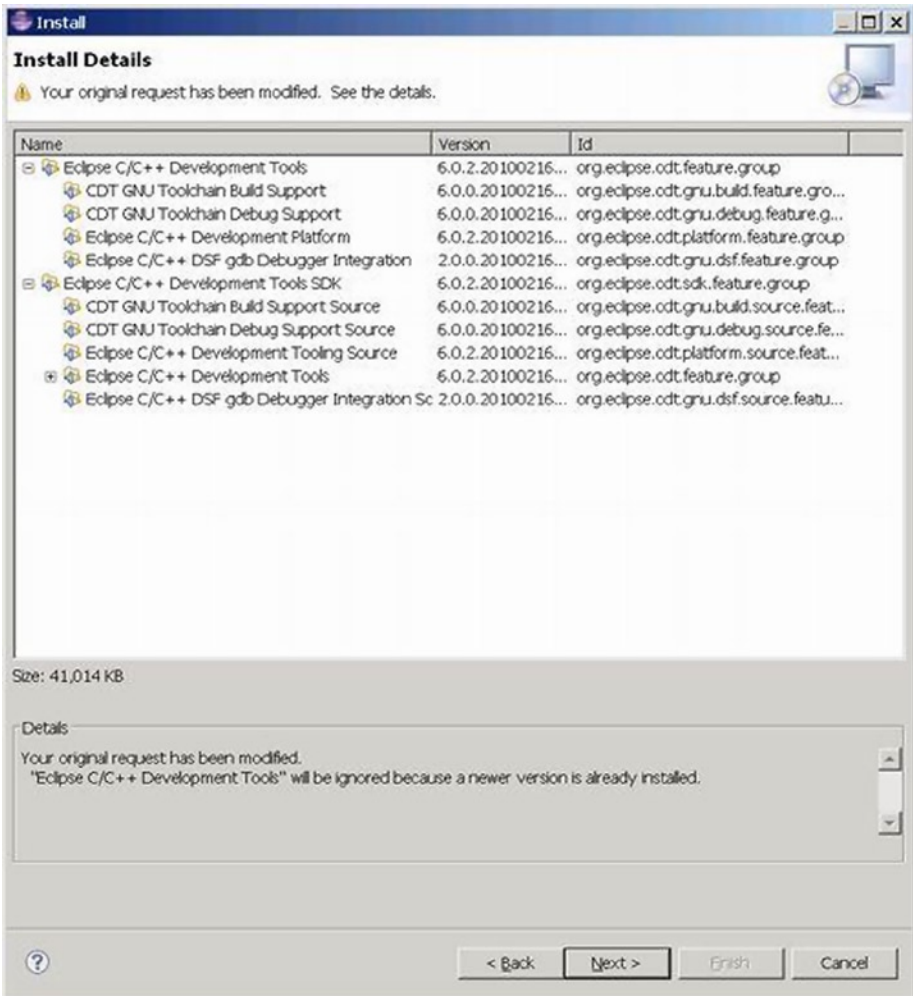


Figure 7-25. Detailed Information for CDT Component Installation

7. Review the licenses dialog box. Click “I accept the terms of the license agreement” to continue, as shown in Figure 7-26.



Figure 7-26. CDT License Review Window

8. The installation process starts, as shown in Figure 7-27.

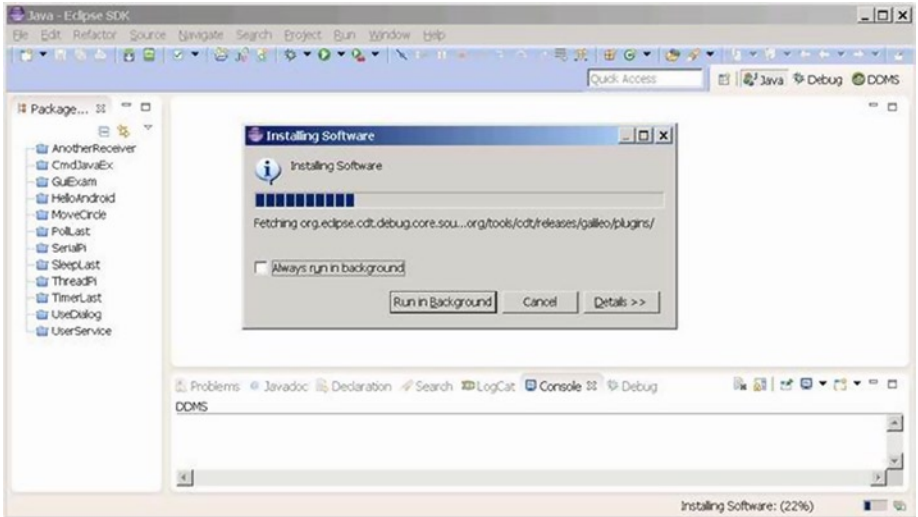


Figure 7-27. CDT Installation Progress

9. When the installation process is complete, restart Eclipse to complete the installation.

NDK Examples

This section includes an example to illustrate the use of JNI and NDK. As described previously, NDK can run from the command line and in the Eclipse IDE. We will use both methods to generate the same NDK application.

Using the Command-Line Method to Generate a Library File

The name of this example is `jnitest`, and it's a simple example to demonstrate the JNI code framework. The steps are outlined in the following sections.

Create an Android App Project

First, you need to create an Android app project, compile the code, and generate the `.apk` package. Create a project in Eclipse, and name the project `jnitest`. Choose Build SDK to support the x86 version of the API (in this case the Android 4.0.3), as shown in Figure 7-28. Finally, you generate the project.

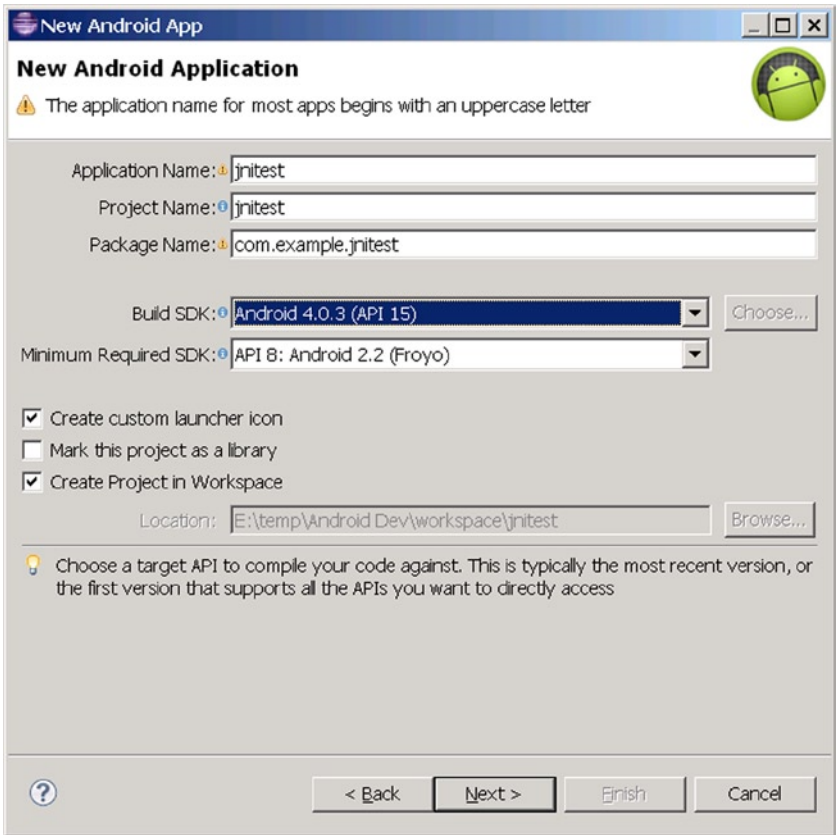


Figure 7-28. *jnitest Project Parameters Setup*

After the project is generated, the file structure is created as shown in Figure 7-29. Note the directory where the library file (in this case, `android.jar`) is located, because the following steps will use this parameter.

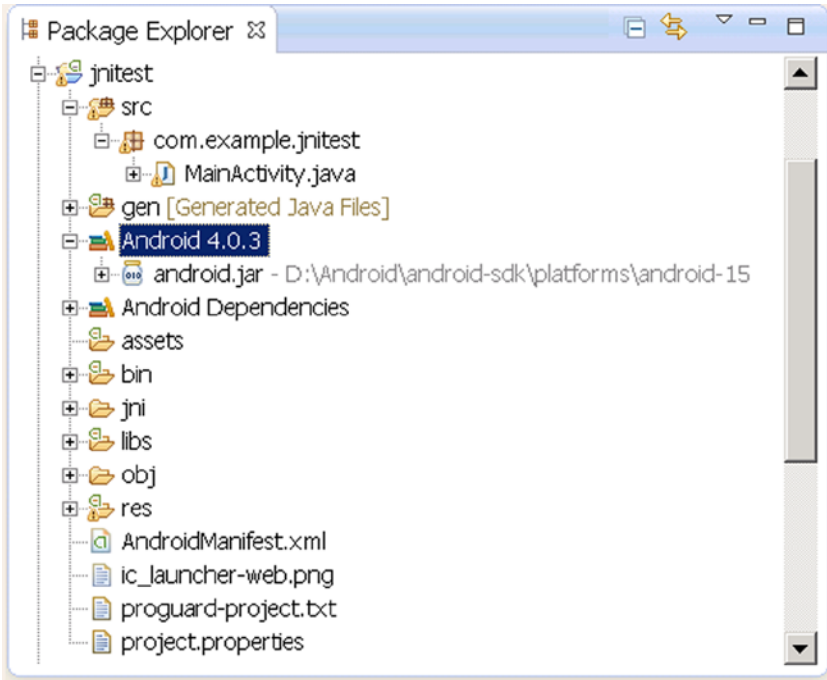


Figure 7-29. File Structure of *jnittest* Project

Modify the Java Files

Next you modify the Java files, creating code using a C function. In this case, the only Java file is `MainActivity.java`. You need to modify its code as follows:

```

1. package com.example.jnittest;
2. import android.app.Activity;
3. import android.widget.TextView;
4. import android.os.Bundle;
5. public class MainActivity extends Activity
6. {
7.     @Override
8.     public void onCreate(Bundle savedInstanceState)
9.     {
10.         super.onCreate(savedInstanceState);
11.         TextView tv = new TextView(this);
12.         tv.setText(stringFromJNI()); // stringFromJNI as a C
function
13.         setContentView(tv);
14.     }

```

```

15.     public native String stringFromJNI();
16.
17.         static {
18.             System.loadLibrary("jnittestmysharelib");
19.         }
20.     }

```

The code is very simple. In lines 11 through 13, you use a `TextView` to display a string returned from the `stringFromJNI()` function. But unlike the Android application discussed before, there is nowhere in the entire project that you can find the implementation code of this function. So where has the implementation of the function occurred? In line 15 you declare that the function is not a function written in Java, but is instead written by the local (native) libraries, which means the function is outside of Java. Since it's implemented in the local library, the question is what libraries? The answers are described in lines 17–20. The parameter of the static function `loadLibrary` of `System` class describes the name of the library. The library is a Linux shared library named `libjnittestmysharelib.so`. The application code declared in the static area will be executed before `Activity.onCreate`. The library will be loaded into memory when it's first used.

Interestingly, when the `loadLibrary` function loads the library name, it will automatically add the `lib` prefix before the parameters and the `.SO` suffix to the end. Of course, if the name of the library file specified by the parameter starts with `lib`, the function will not add the `lib` prefix to the filename.

Generate the Project in Eclipse

Only build (build), rather than run. This will compile the project, but the `.apk` file won't be deployed to the target machine.

When this step is completed, the corresponding `.class` files will be generated in the project directory called `bin\classes\com\example\jnittest`. This step must be completed before the next step, because the next step needs the appropriate `.class` files.

Create a Subdirectory in the Project Root Directory

Name this subdirectory `jni`. For example, if the project root directory is `E:\temp\AndroidDev\workspace\jnittest`, you can use the `md` command to create the `jni` subdirectory.

```
E:\temp\Android Dev\workspace\jnittest>mkdir jni
```

Then test whether the directory has been built:

```

E:\temp\Android Dev\workspace\jnittest>dir
...
2013-02-01  00:45    <DIR>          jni

```

Create a C Interface File

The so-called C interface file is the C function prototype that works with the local (external) function. Specific to this case are the C function prototypes of the `stringFromJNI` function. You declare that you need to use the prototype of the external function, but it is in Java format: you need to change it to C format building C-JNI interface file. This step can be done with the `javah` command. The command format is:

```
$ javah -classpath <directory of jar and .class documents>
-d <directory of .h documents> <the package + class name of class>
```

Command parameters are described here:

- `-classpath`: Represents the classpath
- `-d ...`: Represents the storage directory for the generated header file
- `<class name>`: The complete `.class` classname of a native function being used, which consists of “the package + class name of class” component.

For this example, follow these steps:

1. Enter the root directory from the command line (in this example, it's `E:\temp\Android Dev\workspace\jnitest`).
2. Then run the following command:

```
E:> javah -classpath "D:\Android\android-sdk\platforms\android-15\android.jar";bin/classes com.example.jnitest.MainActivity
```

In this example, the `stringFromJNI`'s class of the native function used is `MainActivity`, and the resulting file after compiling this class is `MainActivity.class`, which is located in the root directory of the project `bin \classes\com\example` directory. The first line of the source code file of its class `MainActivity.java` shows where the package of the class is:

```
package com.example.jnitest;
```

In the previous command, `class name = package name.Class name` (be careful not to use the `.class` suffix), `-classpath` first needs to explain the Java library path of the entire package (in this case the library file is `android.jar`; its location is shown in Figure 7-30, namely `D:\Android\android-sdk\platforms\android-15\android.jar`). `-classpath` also needs to explain the target class (`MainActivity.class`) directory. In this case, it is in the `bin\classes` directory, under `bin\classes\com\example\ MainActivity.class` (both are separated by semicolons).

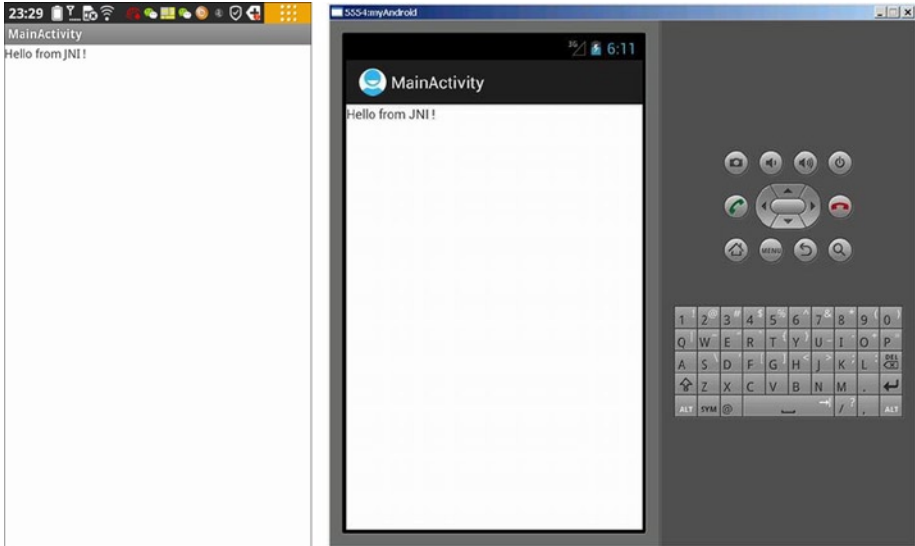


Figure 7-30. *jnittest Application Running Interface*

After the previous steps, the `.h` file is generated in the current directory (the project root directory). The file defines the C language function interface.

You can test the output of the previous steps:

```
E:\temp\Android Dev\workspace\jnittest>dir
...
2013-01-31  22:00          3,556 com_example_jnittest_MainActivity.h
```

It is apparent that a new `.h` file has been generated. The document reads as follows:

```
1.      /* DO NOT EDIT THIS FILE - it is machine generated */
2.      #include <jni.h>
3.      /* Header for class com_example_jnittest_MainActivity */
4.
5.      #ifndef _Included_com_example_jnittest_MainActivity
6.      #define _Included_com_example_jnittest_MainActivity
7.      #ifdef __cplusplus
8.      extern "C" {
9.      #endif
10.     #undef com_example_jnittest_MainActivity_MODE_PRIVATE
11.     #define com_example_jnittest_MainActivity_MODE_PRIVATE 0L
12.     #undef com_example_jnittest_MainActivity_MODE_WORLD_READABLE
13.     #define com_example_jnittest_MainActivity_MODE_WORLD_READABLE 1L
14.     #undef com_example_jnittest_MainActivity_MODE_WORLD_WRITEABLE
15.     #define com_example_jnittest_MainActivity_MODE_WORLD_WRITEABLE 2L
16.     #undef com_example_jnittest_MainActivity_MODE_APPEND
```

```

17. #define com_example_jnitest_MainActivity_MODE_APPEND 32768L
18. #undef com_example_jnitest_MainActivity_MODE_MULTI_PROCESS
19. #define com_example_jnitest_MainActivity_MODE_MULTI_PROCESS 4L
20. #undef com_example_jnitest_MainActivity_BIND_AUTO_CREATE
21. #define com_example_jnitest_MainActivity_BIND_AUTO_CREATE 1L
22. #undef com_example_jnitest_MainActivity_BIND_DEBUG_UNBIND
23. #define com_example_jnitest_MainActivity_BIND_DEBUG_UNBIND 2L
24. #undef com_example_jnitest_MainActivity_BIND_NOT_FOREGROUND
25. #define com_example_jnitest_MainActivity_BIND_NOT_FOREGROUND 4L
26. #undef com_example_jnitest_MainActivity_BIND_ABOVE_CLIENT
27. #define com_example_jnitest_MainActivity_BIND_ABOVE_CLIENT 8L
28. #undef com_example_jnitest_MainActivity_BIND_ALLOW_OOM_MANAGEMENT
29. #define com_example_jnitest_MainActivity_BIND_ALLOW_OOM_MANAGEMENT
16L
30. #undef com_example_jnitest_MainActivity_BIND_WAIVE_PRIORITY
31. #define com_example_jnitest_MainActivity_BIND_WAIVE_PRIORITY 32L
32. #undef com_example_jnitest_MainActivity_BIND_IMPORTANT
33. #define com_example_jnitest_MainActivity_BIND_IMPORTANT 64L
34. #undef com_example_jnitest_MainActivity_BIND_ADJUST_WITH_ACTIVITY
35. #define com_example_jnitest_MainActivity_BIND_ADJUST_WITH_ACTIVITY
128L
36. #undef com_example_jnitest_MainActivity_CONTEXT_INCLUDE_CODE
37. #define com_example_jnitest_MainActivity_CONTEXT_INCLUDE_CODE 1L
38. #undef com_example_jnitest_MainActivity_CONTEXT_IGNORE_SECURITY
39. #define com_example_jnitest_MainActivity_CONTEXT_IGNORE_SECURITY 2L
40. #undef com_example_jnitest_MainActivity_CONTEXT_RESTRICTED
41. #define com_example_jnitest_MainActivity_CONTEXT_RESTRICTED 4L
42. #undef com_example_jnitest_MainActivity_RESULT_CANCELED
43. #define com_example_jnitest_MainActivity_RESULT_CANCELED 0L
44. #undef com_example_jnitest_MainActivity_RESULT_OK
45. #define com_example_jnitest_MainActivity_RESULT_OK -1L
46. #undef com_example_jnitest_MainActivity_RESULT_FIRST_USER
47. #define com_example_jnitest_MainActivity_RESULT_FIRST_USER 1L
48. #undef com_example_jnitest_MainActivity_DEFAULT_KEYS_DISABLE
49. #define com_example_jnitest_MainActivity_DEFAULT_KEYS_DISABLE 0L
50. #undef com_example_jnitest_MainActivity_DEFAULT_KEYS_DIALER
51. #define com_example_jnitest_MainActivity_DEFAULT_KEYS_DIALER 1L
52. #undef com_example_jnitest_MainActivity_DEFAULT_KEYS_SHORTCUT
53. #define com_example_jnitest_MainActivity_DEFAULT_KEYS_SHORTCUT 2L
54. #undef com_example_jnitest_MainActivity_DEFAULT_KEYS_SEARCH_LOCAL
55. #define com_example_jnitest_MainActivity_DEFAULT_KEYS_SEARCH_LOCAL
3L
56. #undef com_example_jnitest_MainActivity_DEFAULT_KEYS_SEARCH_GLOBAL
57. #define com_example_jnitest_MainActivity_DEFAULT_KEYS_SEARCH_GLOBAL
4L

```

```

58.      /*
59.       * Class:      com_example_jnittest_MainActivity
60.       * Method:     stringFromJNI
61.       * Signature: ()Ljava/lang/String;
62.       */
63.      JNIEXPORT jstring JNICALL Java_com_example_jnittest_MainActivity_
stringFromJNI
64.          (JNIEnv *, jobject);
65.
66.      #ifdef __cplusplus
67.      }
68.      #endif
69.      #endif

```

In the previous code, pay special attention to lines 63–64, which are C function prototypes of a local function `stringFromJNI`.

Compile the Corresponding. C File

This is the true realization of a local function (`stringFromJNI`). The source code file is obtained by modifying the `.h` file, according to the previous steps.

Create a new `.C` file under the `jni` subdirectory in the project. The filename can be created randomly. In this case, it is named `jnittestccode.c`. The contents are as follows:

```

1.      #include <string.h>
2.      #include <jni.h>
3.      jstring Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv*
env, jobject this )
4.      {
5.          return (*env)->NewStringUTF(env, "Hello from JNI !");
// Newly added code
6.      }

```

The previous code defines the function implementation and is very simple. Line 3 is the Java code used in the prototype definition of the function `stringFromJNI`. It is basically a copy of the corresponding content of the `.h` file obtained from the previous steps (lines 63–64 of `com_example_jnittest_MainActivity.h`), and slightly modified to make the point. The prototype formats of this function are fixed—`JNIEnv* env` and `jobject this` are inherent parameters of JNI. Because the parameter of the `stringFromJNI` function is empty, there are only two parameters in the generated C function. The role of the code in line 5 is to return the string "Hello fromJNI!" as the return value.

The code in line 2 is the header file that contains the JNI function, which is required for any functions that use JNI. As it relates to the `string` function, line 1 contains the corresponding header file in this case. After you complete the previous steps, the `.h` file has no further use and can be deleted.

Create the NDK Makefile File in the jni Directory

These documents mainly include the `Android.mk` and `Application.mk` files, where `Android.mk` is required. However, if you use the default configuration of the application, you do not need `Application.mk`. The four specific steps are as follows:

1. Create a new `Android.mk` text file in the `jni` directory in the project. This file tells the compiler about some requirements, such as which C files to compile, the filename for compiled code, and so on. Enter the following:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := jnittestmysharelib
LOCAL_SRC_FILES := jnittestccode.c
include $(BUILD_SHARED_LIBRARY)
```

The file contents are explained next.

Line 3 represents the generated `.SO` filename (identifying each module described in your `Android.mk` file). It must be consistent with parameter values of the `System.loadLibrary` function in the Java code. This name must be unique and may not contain any spaces.

■ **Note** The build system automatically generates the appropriate prefix and suffix. In other words, if one is the shared library module named `jnittestmysharelib`, then a `libjnittestmysharelib.so` file will be generated. If you name the library `libhello-jni`, the compiler will not add the `lib` prefix and will generate `libhello-jni.so` too.

The `LOCAL_SRC_FILES` variable in line 4 must contain the C or C++ source code files to be compiled and packaged into modules. The previous steps create a C filename.

■ **Note** Users do not have to list the header files and include files here, because the compiler will automatically identify the dependent files for you. Just list source code files that are directly passed to the compiler. In addition, the default extension name of C++ source files is `.CPP`. It is possible to specify a different extension name, as long as you define the `LOCAL_DEFAULT_CPP_EXTENSION` variable. Don't forget the small dot at the start (`.cxx`, rather than `cxx`).

The code in Lines 3 through 4 is very important and must be modified for each NDK application based on their actual configuration. The contents of the other lines can be copied from the previous example.

2. Create an `Application.mk` text file in the `jni` directory in the project. This file tells the compiler the specific settings for this application. Enter the following:

```
APP_ABI := x86
```

This file is very simple. You use the object code generated by the application instructions for the x86 architecture, so you can run the application on Intel Atom machines. For `APP_ABI` parameters, use `x86`, `armeabi`, or `armeabi-v7a`.

3. Next, compile the `.c` file to the `.SO` shared library file.

Go to project root directory (where `AndroidManifest.xml` is located) and run the `ndk-build` command:

```
E:\temp\Android Dev\workspace\jnitest>ndk-build
D:/Android/android-ndk-r8d/build/core/add-application.mk:128: Android NDK:
WARNI
NG: APP_PLATFORM android-14 is larger than android:minSdkVersion 8 in
./AndroidM
anifest.xml
"Compile x86 : jnitestmysharelib <= jnitestccode.c
SharedLibrary : libjnitestmysharelib.so
Install : libjnitestmysharelib.so => libs/x86/libjnitestmysharelib.so
```

The previous command will add two subdirectories (`libs` and `obj`) in the project. Include an execution version of the `.SO` file (the command execution information prompt file named `libjnitestmysharelib.so`) under the `obj` directory, and it will eventually put the final version under the `libs` directory.

If the previous steps do not define the `Application.mk` file of the specified ABI, using the `ndk-build` command will generate object code of the ARM architecture (`armeabi`). If you must generate the x86 architecture instructions, you can also use the `ndk-build APP_ABI = x86` command to remedy the situation. The architecture of the object code generated by this command is still x86.

4. Deployment: run the project.

After you complete this step, you are almost ready to deploy and run the project. The application running on the interface on the target device is shown in Figure 7-30.

Generating a Library File in the IDE

Recall from the steps described in the previous section the process of compiling the C files into the dynamic library `.SO` files that can be run on the Android target device. You run the `ndk-build` command in the command line to complete the process. In fact, you can also complete this step within the Eclipse IDE.

When generating the library files in the IDE, the code in the first four steps are exactly the same as in the previous section. You just have to compile the .C files into .SO shared library files instead. This is explained in detail as follows:

1. Compile the .C file into the .SO shared library file. Right-click on the project name, and select Build Path, Configure Build Path. In the pop-up dialog box, select the Builders branch. Then click the New button in the dialog box. Double-click Program in the prompt dialog box. This process is shown in Figure 7-31.

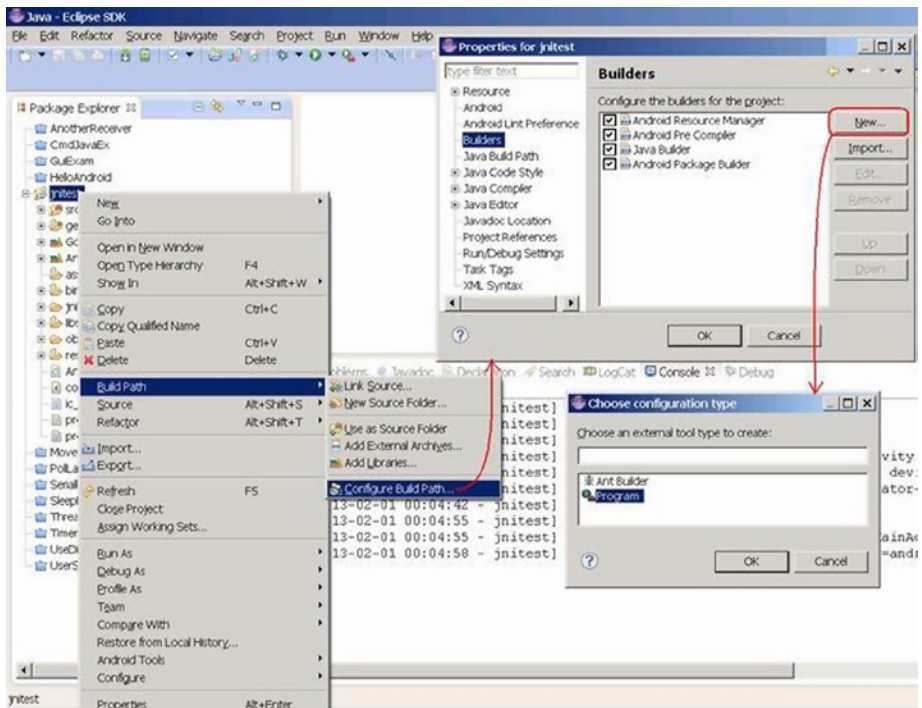


Figure 7-31. Enter Parameters Settings for the Interface of Compiling C Code in Eclipse

2. In the Edit Configuration dialog box, enter the following for the Main tab settings:
 - **Location:** The path to the Cygwin bash.exe.
 - **Working Directory:** The bin directory of Cygwin.

- *Arguments:*

```
--login -c "cd '/cygdrive/E/temp/Android Dev/workspace/jnittest' && $ANDROID_NDK_ROOT/ndk-build"
```

where E/temp/Android Dev/workspace/jnittest is the letter and path for the project. The entire setting is shown in Figure 7-32.

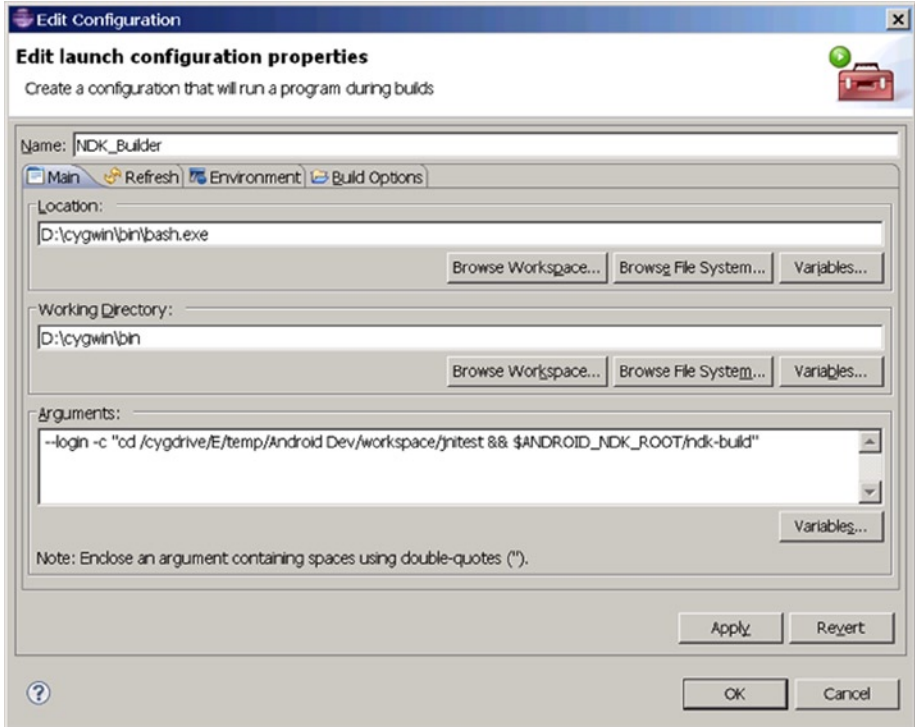


Figure 7-32. Main Tab Setting in the Edit Configuration Window

3. Then configure the Refresh tab, ensuring that these items are selected—The Entire Workspace and Recursively Include Sub-Folders—as shown in Figure 7-33.

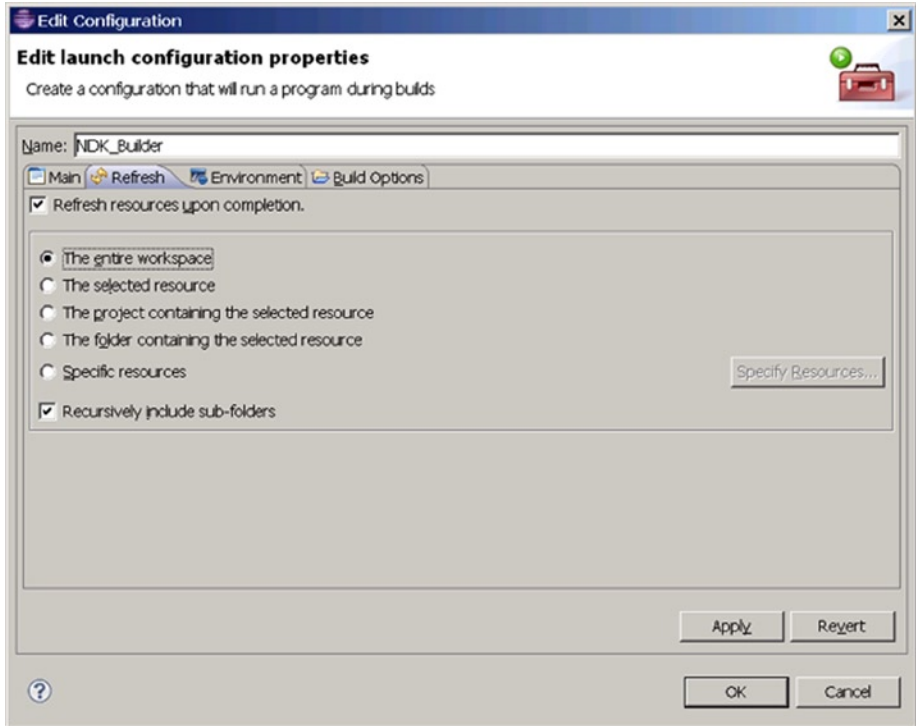


Figure 7-33. *Edit Configuration Window Refresh Tab Settings*

4. Reconfigure the Build Options tab. Check the During Auto Builds and Specify Working Set of Relevant Resources items, as shown in Figure 7-34.

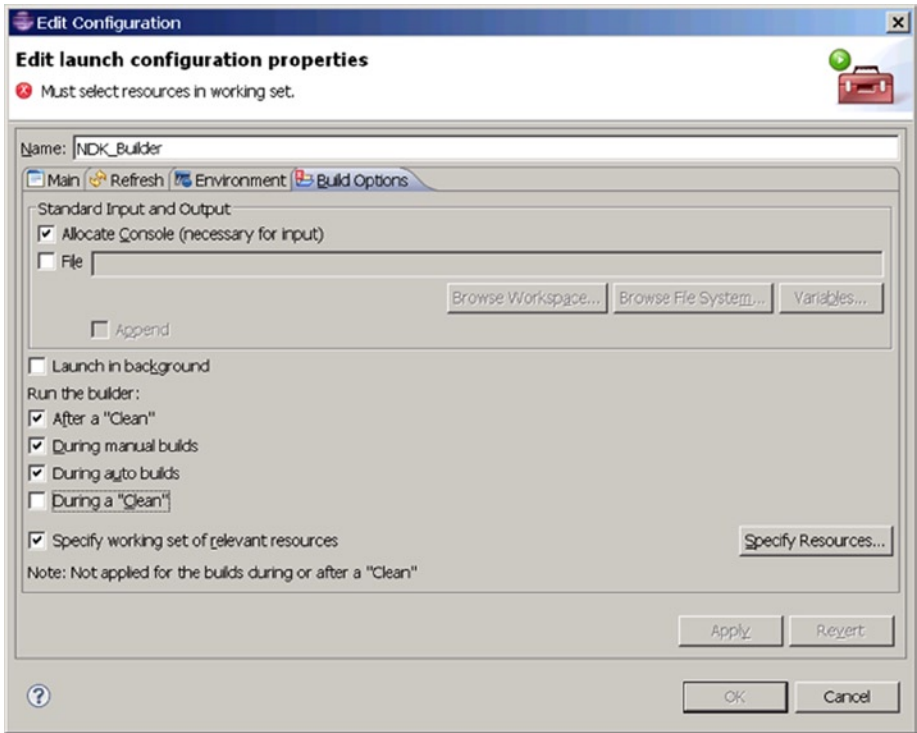


Figure 7-34. *Edit Configuration Window Build Options Tab Settings*

5. Click on the Specify Resources button. In the Edit Working Set dialog box, select the jni directory, as shown in Figure 7-35.

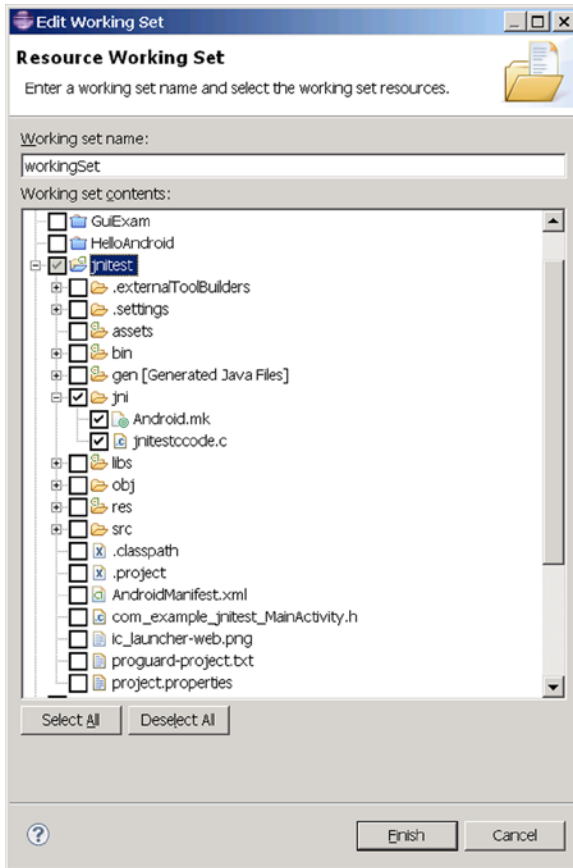


Figure 7-35. Select Source Code Directories Where Related Files Are Located

6. When the previous steps are correctly configured, the configuration is saved. It will automatically compile C-related code under the `jni` directory and output the corresponding .SO library files to the project's `libs` directory. The `libs` directory is created automatically. In the Console window you can see the output information for the build, as follows:

```
/cygdrive/d/Android/android-ndk-r8d/build/core/add-application.
mk:128: Android NDK: WARNING: APP_PLATFORM android-14 is larger than
android:minSdkVersion 8 in ./AndroidManifest.xml
Cygwin      : Generating dependency file converter script
Compile x86 : jnitestmysharelib <= jnitestccode.c
SharedLibrary : libjnitestmysharelib.so
Install     : libjnitestmysharelib.so => libs/x86/libjnitestmysharelib.so
```

Workflow Analysis for NDK Application Development

The process of generating an NDK project described previously works naturally to achieve the C library integration with Java. In the final step, you compile .C files into the .SO shared library files. The intermediate version of the libraries is placed into the obj directory, and the final version is placed into the libs directory. The project file structure is then created, as shown in Figure 7-36.



Figure 7-36. The *jnitest* Project Structure after NDK Library Files Generation

The shared library .SO files are in the directory of the project in the host machine and will be packed in the generated .apk file. The .apk file is essentially a compressed file. You can use compression software like WinRAR to view its contents. For this example, you can find the .apk file in the bin subdirectory of the project directory. Open it with WinRAR, and show the file structure.

The content of the lib subdirectory of .apk is a clone of the lib subdirectory of the project. In Figure 7-36 the generated .SO file is shown in the lib\x86 subdirectory.

When .apk is deployed to the target machine, it will be unpacked, in which case the .SO files will be placed in the /data/dat/XXX/lib directory, where XXX is the application package name. For example, for the previous example, the directory is /data/data/com.example.jnittest/lib. You can view the file structure of the target machine under the Eclipse DDMS; the file structure of the example is shown in Figure 7-37.

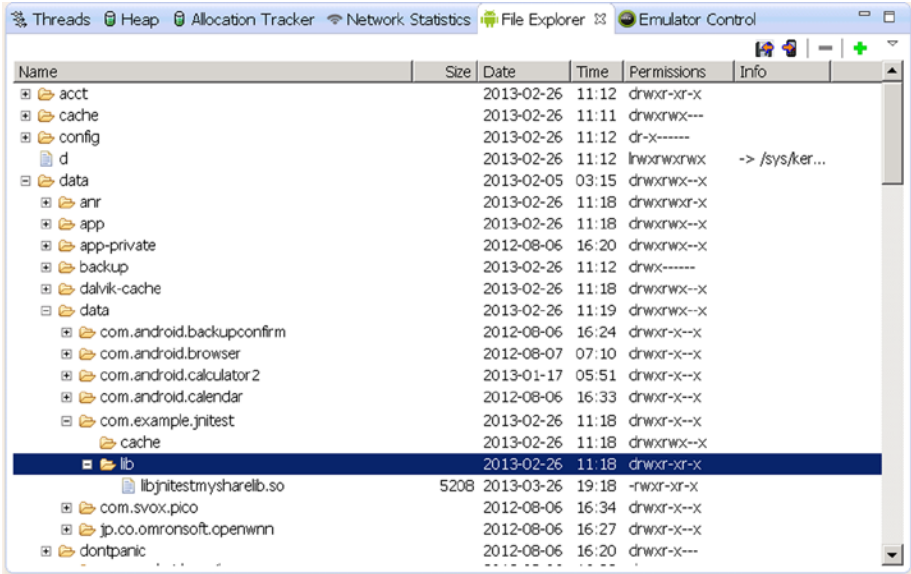


Figure 7-37. The jnittest Project Structure after NDK Library Files Generation

In Figure 7-37, you can find the .SO library file under the /data/data/XXX/lib directory, such that when the application is running, the System.loadLibrary function can be loaded into memory to run. Here you see the .SO file in a graphical display of DDMS. Interested readers can try it on the command line, using the adb shell command to view the corresponding contents in the target file directory.

In addition, if you run the jnittest application in an emulator (in this case the target machine is a virtual machine), you'll see the following output in the Eclipse Logcat window:

- 07-10 05:43:08.579: E/Trace(6263): error opening trace file: No such file or directory (2)
- 07-10 05:43:08.729: D/dalvikvm(6263): Trying to load lib /data/data/com.example.jnittest/lib/libjnittestmysharelib.so 0x411e8b30
- 07-10 05:43:08.838: D/dalvikvm(6263): Added shared lib /data/data/com.example.jnittest/lib/libjnittestmysharelib.so 0x411e8b30
- 07-10 05:43:08.838: D/dalvikvm(6263): No JNI_OnLoad found in /data/data/com.example.jnittest/lib/libjnittestmysharelib.so 0x411e8b30, skipping init

```

5. 07-10 05:43:11.773: I/Choreographer(6263): Skipped 143 frames!
The application may be doing too much work on its main thread.
6. 07-10 05:43:12.097: D/gralloc_goldfish(6263): Emulator without GPU
emulation detected.

```

Lines 2–3 are reminders about the .SO shared library loaded into the application.

NDK Compiler Optimization

From the previous example, you can see that the NDK tool’s core role is to compile the source code into the .SO library file that can run on an Android machine. The .SO library file is placed into the lib subdirectory of the project directory, so that when you use Eclipse to deploy applications, you can deploy the library files to the appropriate location on a target device, and the application can run using the library function.

■ **Note** The nature of the NDK application is to establish a code framework that complies with the JNI standard. This will enable Java applications to use a local function beyond the scope of the virtual machine.

The key NDK command used to compile the source code into a .SO library file is `ndk-build`. It’s not actually a separate command, but an executable script. It calls the `make` command in the GNU cross-development tools to compile a project, and `make` calls, for example, to the `gcc` compiler to compile the source code to complete the whole process, as shown in Figure 7-38. Of course, you can also directly use .SO shared libraries developed by third parties already in Android applications, thus avoiding the need to write your own library (function code).

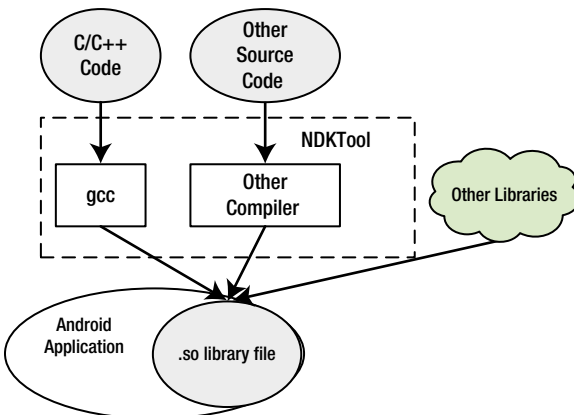


Figure 7-38. The Working Mechanism of NDK Tools

As Figure 7-38 shows, core GNU compiler gcc is the core tool in the NDK to complete C/C++ source code compilation. gcc is the standard compiler of Linux, and it can compile and link C, C++, Object-C, FORTRAN, and other source code on the local machine. In fact, the gcc compiler can not only do local compiling, but can also cross-compiling. This feature has been used by the Android NDK and other embedded development tools. In compiler usage, gcc cross-compiling is compatible with native compiling; that is, command parameters and switches of locally compiled code can essentially be ported without modification to cross-compiling code. Therefore, the gcc compiling method described next is generic for both local and cross-compiling.

In **Chapter 9: Performance Optimizations for Android Applications on x86**, we will discuss compiler optimizations in greater detail (that is, how some optimizations can be done automatically by the compiler). For systems based on Intel x86 architecture processors, in addition to the GNU gcc compiler, Intel C/C++ compiler is also a good tool. Relatively speaking, because the Intel C/C++ compiler fully utilizes the features of the Intel processors, the code optimization results will be better. For Android NDK, both Intel C/C++ compiler and gcc can complete the C/C++ code compilation. Currently, the Intel C/C++ compiler provides the appropriate usage mechanisms. Ordinary users need a professional license, while gcc is open sourced, free software and is more readily available. The following section uses gcc as an experimental tool to explain how to perform C/C++ module compiler optimization for Android applications.

The gcc optimization is controlled by the optimization options of the compiler switches. Some of these options are machine-independent, and some are associated with the machine. Here we will discuss some important options. For machine-related options, we will describe only the ones that are relevant to Intel processors.

Machine-Independent Compiler Switch Options

The machine-independent options for the gcc compiler switches are the `-Ox` options, which correspond to different optimization levels. The details are as follows.

-O or -O1

Level 1 optimization, which is the default level of optimization, uses the `-O` option. The compiler tries to reduce code size and execution time. For large functions, it needs to spend more compiling time and use a large amount of memory resources for optimizing compiling.

When the `-O` option is not used, the compiler's goal is to reduce the overhead of compiling, so that results can be debugged quickly. In this compilation mode, statements are independent. By inserting a breakpoint interrupt program run between the two statements, a user can reassign variables or modify the program counter to jump to other currently executing statements, so you can precisely control the running process. The user can also get results when they want to debug. In addition, if the `-O` option is not used, only declared variables of a register can have register allocation.

When you specify the `-O` option, the `-fthread-jumps` and `-fdefer-pop` options are turned on. On a machine with a delay slot, the `-fdelayed-branch` option is turned on. Even for machines that support debugging without a frame pointer, the `-fomit-frame-pointer` option is turned on. Some machines may also activate other options.

-O2

Optimizes even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. When compared to -O, this option increases compilation time and the performance of the generated code.

-O3

Optimizes yet more. The option -O3 turns on all optimizations specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize, -fvect-cost-model, -ftree-partial-pre, and -fipa-cp-clone options.

-O0

Reduces compilation time and makes debugging produce the expected results. This is the default.

An automatic inline function is often used as a function optimization measure. C99 (C language ISO standard developed in 1999) and C++ both support the `inline` keyword. The `inline` function is a reflection of thinking of using inline space in exchange for time. The compiler does not compile an inline-described function into a function, but directly expands the code for the function body, thereby eliminating the function call, returning the `call ret` instruction and the parameter's push instruction execution. For example, in the following function:

```
inline long factorial (int i)
{
    return factorial_table[i];
}
```

all occurrences of the `factorial ()` call are replaced with the `factorial_table []` array references.

When in the optimizing state, some compilers will treat that function as an inline function even if the function does not use inline instructions. It does this only if appropriate in the circumstances (such as the body of the function code is relatively short and the definition is in the header file), in exchange for execution time.

Loop unrolling is a classic speed optimization method and is considered by many compilers as the automatic optimization strategy. For example, the following loop code needs to loop 100 cycles:

```
for (i = 0; i < 100; i++)
{
    do_stuff(i);
}
```

In all 100 cycles, at the end of each cycle, the cycle conditions have to be checked to do a comparative judgment. By using a loop-unrolling strategy, the code can be transformed as follows:

```
for (i = 0; i < 100; )
{
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
}
```

As you can see, the new code reduces the comparison instruction from 100 to 10 times, and the time used on conditions comparison can be reduced by 90 percent.

Both methods described previously will increase the optimization of the object code. This is a typical space for time-optimization ideas.

Intel Processor-Related Compiler Switch Options

The `m` option of `gcc` is defined for the Intel i386 and x86 - 64 processors family. The main command options are explained in Table 7-3.

Table 7-3. Intel Processor-Related gcc Switch Options

Switch Options	Note	Description
-march=cpu-type		Generated code for the specified type of CPU. CPU type can be i386, i486, i586, Pentium, i686, Pentium 4, and so on
-mtune=cpu-type		
-msse		
-msse2		
-msse3		
-mssse3	gxx-4.3 new addition	
-msse4.1	gcc-4.3 new addition	The compiler automatic vectorization. Use or not use MMX, SSE, SSE2 instructions. For example, -msse represents programming into instruction, and -mno-sse means not programmed into the SSE instruction
-msse4.2	gcc-4.3 new addition	
-msse4	Include 4.1, 4.2 ,gcc-4.3 new addition	
-mmmx		
-mno-sse		
-mno-sse2		
-mno-mmx		
-m32		Generated 32/64 machine code
-m64		

In Table 7-3, -march is the CPU type of the machine, and -mtune is the CPU type that the compiler wants to optimize (by default it is the same as with -march). The -march option is “tight constraint,” and -mtune is “loose constraint.” The -mtune option can provide backward compatibility.

Compiler optimization options with -march = i686, -mtune = pentium4 is optimized for the Pentium 4 processor, but can be run on any i686 as well.

For -mtune = pentium-mmx compiled procedures, the Pentium 4 processor can be run.

-march=cpu-type

This option will generate cpu-type instructions that specify the type of machine. The -mtune = cpu-type option is available only for optimizing code generated for cpu-type. By contrast, -march = cpu-type generates code not run on non-gcc for the specified type of processor, which means that -march = cpu-type implies the -mtune = cpu-type option.

The cpu-type option values that are related to Intel processors are listed in Table 7-4.

Table 7-4. *The Main Optional Value of -march Parameters of gcc for cpu-type*

cpu-type Value	Description
native	This selects the CPU to generate code at compilation time by determining the processor type of the compiling machine. Using -march=native enables all instruction subsets supported by the local machine (hence the result might not run on different machines). Using -mtune=native produces code optimized for the local machine under the constraints of the selected instruction set.
i386	Original Intel i386 CPU.
i486	Intel i486 CPU. (No scheduling is implemented for this chip.)
i586	Intel Pentium CPU with no MMX support.
pentium	
pentium-mmx	Intel Pentium MMX CPU, based on Pentium core with MMX instruction set support.
pentiumpro	Intel Pentium Pro CPU.
i686	When used with -march, the Pentium Pro instruction set is used, so the code runs on all i686 family chips. When used with -mtune, it has the same meaning as “generic.”
pentium2	Intel Pentium II CPU, based on Pentium Pro core with MMX instruction set support.
pentium3	Intel Pentium III CPU, based on Pentium Pro core with MMX and SSE instruction set support.
pentium3m	
pentium-m	Intel Pentium M; low-power version of Intel Pentium III CPU with MMX, SSE, and SSE2 instruction set support. Used by Centrino notebooks.
pentium4	Intel Pentium 4 CPU with MMX, SSE and SSE2 instruction set support.
pentium4m	
prescott	Improved version of Intel Pentium 4 CPU with MMX, SSE, SSE2, and SSE3 instruction set support.
nocona	Improved version of Intel Pentium 4 CPU with 64-bit extensions, MMX, SSE, SSE2, and SSE3 instruction set support.
core2	Intel Core 2 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, and SSSE3 instruction set support.

(continued)

Table 7-4. *(continued)*

cpu-type Value	Description
corei7	Intel Core i7 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, and SSE4.2 instruction set support.
corei7-avx	Intel Core i7 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES, and PCLMUL instruction set support.
core-avx-i	Intel Core CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES, PCLMUL, FSGSBASE, RDRND, and F16C instruction set support.
atom	Intel Atom CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, and SSSE3 instruction set support.

Traditional gcc is a local compiler. These command options can be added to gcc to control gcc compiler options. For example, say you have an `int_sin.c` file.

```
$ gcc int_sin.c
```

The previous command uses the `-O1` optimization level (default level) and will compile `int_sin.c` into an executable file, called `a.out` by default.

```
$ gcc int_sin.c -o sinnorm
```

The previous command uses the `-O1` optimization level (default level) to compile `int_sin.c` into an executable file; the executable filename is specified as `sinnorm`.

```
$ gcc int_cos.c -fPIC -shared -o coslib.so
```

The previous command uses the `-O1` optimization level (default level) to compile `int_cos.c` into a shared library file called `coslib.so`. Unlike the previous source code files compiled into an executable program, this command requires that the source code file `int_cos.c` not contain the main function.

```
$ gcc -O0 int_sin.c
```

The previous command compiles `int_sin.c` into the executable file with the default filename. The compiler does not perform any optimization.

```
$ gcc -O3 int_sin.c
```

The previous command uses the highest optimization level `-O3` to compile the `int_sin.c` file to the executable file with the default filename.

```
$ gcc -msse int_sin.c
```

The previous command compiles `int_sin.c` into an executable file using SSE instructions.

```
$ gcc -mno-sse int_sin.c
```

The previous command compiles `int_sin.c` into an executable file without any SSE instructions.

```
$ gcc -mtune=atom int_sin.c
```

The previous command compiles `int_sin.c` into an executable file that can use the Intel Atom processor instructions.

From the previous example compiled by gcc locally, you have some experience using the compiler switch options for the gcc compiler optimizations. For the gcc native compiler, the gcc command can be used directly in the switch options to achieve compiler optimization. However, from the previous example, you know that the NDK does not directly use the gcc command. Then how do you set the gcc compiler switch option to achieve the NDK optimization?

Recall that using the NDK example, you used the `ndk-build` command to compile C/C++ source code; the command first needed to read the makefile `Android.mk`. This file actually contains the gcc command options. `Android.mk` uses `LOCAL_CFLAGS` to control and complete the gcc command options. The `ndk-build` command will pass `LOCAL_CFLAGS` runtime values to gcc, as its command option to run the gcc command. `LOCAL_CFLAGS` passes the values to gcc and uses them as the command option to run gcc command.

For example, you amended `Android.mk` as follows:

```
1. LOCAL_PATH := $(call my-dir)
2. include $(CLEAR_VARS)
3. LOCAL_MODULE      := jnittestmysharelib
4. LOCAL_SRC_FILES   := jnittestccode.c
5. LOCAL_CFLAGS      := -O3
6. include $(BUILD_SHARED_LIBRARY)
```

Line 5 is newly added. It sets the `LOCAL_CFLAGS` variable script.

When you execute the `ndk-build` command, which is equivalent to adding a `gcc -O3` command option. It instructs gcc to compile the C source code at the highest optimization level O3. Similarly, if you edit the line 5 to:

```
LOCAL_CFLAGS      := -msse3
```

You instruct gcc to compile C source code into object code using SSE3 instructions.

Interested readers can set `LOCAL_CFLAGS` to a different value, comparing the target library file size and content differences. Note that the previous example `jnittest` C code is very simple and does not involve complex tasks. As a result, the size or content of the library files are not very different when compiled from different `LOCAL_CFLAGS` values.

So, can there ever be a significant difference in the size or content of the library file? In fact, the answer is yes. In this regard, we will give practical examples in the following chapters.

Overview

With this chapter behind you, you should have a comprehensive knowledge of the Android native development kit and understand how it can be used to create Android applications for the Intel platform. We also covered the Intel C++ compiler and its options. It is important to remember that the Intel C++ compiler is just one of the possible compilers that can be used for Intel Android applications. We talked at length about the Java native interface that exists to interact with your NDK applications, and how it operates. We also covered various code samples to best explain the various basic optimizations that exist for the Intel C++ compiler.