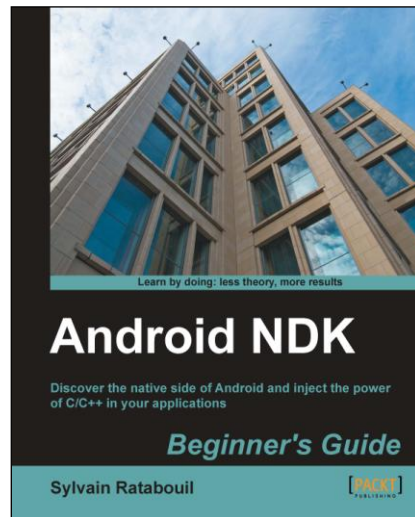




Android NDK Beginner's Guide

Sylvain Ratabouil



Chapter No. 11 "Debugging and Troubleshooting"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.11 "Debugging and Troubleshooting"

A synopsis of the book's content

Information on where to buy this book

About the Author

Sylvain Ratabouil is a confirmed IT consultant with experience in C++ and Java technologies. He worked for the space industry and got involved in aeronautic projects at Valtech Technologies where he now takes part in the Digital Revolution.

Sylvain earned the master's degree in IT from Paul Sabatier University in Toulouse and did M.Sc. in Computer Science from Liverpool University.

As a technology lover, he is passionate about mobile technologies and cannot live or sleep without his Android smartphone.

I would like to thank Steven Wilding for offering me to write this book; Sneha Harkut and Jovita Pinto for awaiting me with so much patience; Reshma Sundaresan, and Dayan Hyames for putting this book on the right track; Sarah Cullington for helping me finalizing this book; Dr. Frank Grützmacher, Marko Gargenta, and Robert Mitchell for all their helpful comments.

For More Information:

www.packtpub.com/android-ndk-beginners-guide/book

Android NDK

Beginner's Guide

The short history of computing machines has witnessed some major events, which forever transformed our usage of technology. From the first massive main frames to the democratization of personal computers, and then the interconnection of networks. Mobility is the next revolution. Like the primitive soup, all the ingredients are now gathered: an ubiquitous network, new social, professional and industrial usages, a powerful technology. A new period of innovation is blooming right now in front of our eyes. We can fear it or embrace it, but it is here, for good!

The mobile challenge

Today's mobile devices are the product of only a few years of evolution, from the first transportable phones to the new tiny high-tech monsters we have in our pocket. The technological time scale is definitely not the same as the human one.

Only a few years ago, surfing on the successful wave of its musical devices, Apple and its founder Steve Jobs combined the right hardware and the right soft ware at the right time not only to satisfy our needs, but to create new ones. We are now facing a new ecosystem looking for a balance between iOS, Windows Mobile, Blackberry, WebOS, and more importantly Android! The appetite of a new market could not let Google apathetic. Standing on the shoulder of this giant Internet, Android came into the show as the best alternative to the well established iPhones and other iPads. And it is quickly becoming the number one.

In this modern Eldorado, new usages or technically speaking, applications (*activities*, if you already are an Android adept) still have to be invented. This is the mobile challenge. And the dematerialized country of Android is the perfect place to look for. Android is (mostly) an open source operating system now supported by a large panel of mobile device manufacturers.

Portability among hardware and adaptability to the constrained resources of mobile devices: this is the real essence of the mobile challenge from a technical perspective. With Android, ones has to deal with multiple screen resolutions, various CPU and GPU speed or capabilities, memory limitations, and so on, which are not topics specific to this Linux-based system, (that is, Android) but can particularly be incommoding.

For More Information:

www.packtpub.com/android-ndk-beginners-guide/book

To ease portability, Google engineers packaged a virtual machine with a complete framework (the Android SDK) to run programs written in one of the most spread programming language nowadays: Java. Java, augmented with the Android framework, is really powerful. But first, Java is specific to Android. Apple's products are written for example in Objective C and can be combined with C and C++. And second, a Java virtual machine does not always give you enough capability to exploit the full power of mobile devices, even with just-in-time compilation enabled. Resources are limited on these devices and have to be carefully exploited to offer the best experience. This is where the Android Native Development Kit comes into place.

What This Book Covers

Chapter 1, Setting Up your Environment, covers the tools required to develop an application with the Android NDK. This chapter also covers how to set up a development environment, connect your Android device, and configure the Android emulator.

Chapter 2, Creating, Compiling, and Deploying Native Projects, we will compile, package, and deploy NDK samples and create our first Android Java/C hybrid project with NDK and Eclipse.

Chapter 3, Interfacing Java and C/C++ with JNI, presents how Java integrates and communicates with C/C++ through Java Native Interface.

Chapter 4, Calling Java Back from Native Code, we will call Java from C to achieve bidirectional communication and process graphic bitmaps natively.

Chapter 5, Writing a Fully-native Application, looks into the Android NDK application life-cycle. We will also write a fully native application to get rid of Java.

Chapter 6, Rendering Graphics with OpenGL ES, teaches how to display advanced 2D and 3D graphics at full speed with OpenGL ES. We will initialize display, load textures, draw sprites and allocate vertex and index buffers to display meshes.

Chapter 7, Playing Sound with OpenSL ES, adds a musical dimension to native applications with OpenSL ES, a unique feature provided only by the Android NDK. We will also record sounds and reproduce them on the speakers.

Chapter 8, Handling Input Devices and Sensors, covers how to interact with an Android device through its multi-touch screen. We will also see how to handle keyboard events natively and apprehend the world through sensors and turn a device into a game controller.

Chapter 9, Porting Existing Libraries to Android, we will compile the indispensable C/C++ frameworks, STL and Boost. We will also see how to enable exceptions and RunTime Type Information. And also port our own or third-party libraries to Android, such as, Irrlicht 3D engine and Box2D physics engine.

For More Information:

www.packtpub.com/android-ndk-beginners-guide/book

Chapter 10, Towards Professional Gaming, creates a running 3D game controlled with touches and sensors using Irrlicht and Box2D.

Chapter 11, Debugging and Troubleshooting, provides an in-depth analysis of the running application with NDK debug utility. We will also analyze crash dumps and profile the performance of our application.

For More Information:

www.packtpub.com/android-ndk-beginners-guide/book

11

Debugging and Troubleshooting

This introduction to the Android NDK would not be complete without approaching some more advanced topics: debugging and troubleshooting code. Indeed, C/C++ are complex languages that can fail in many ways.

I will not lie to you: NDK debugging features are rather rubbish yet. It is often more practical and fast to rely on simple log messages. This is why debugging is presented in this last chapter. But still, a debugger can save quite some time in complex programs or even worse... crashing programs! But even in that case, there exist alternative solutions.

More specifically, we are going to discover how to do the following:

- ◆ Debug native code with **GDB**
- ◆ Interpret a **stack trace** dump
- ◆ Analyze program performances with **GProf**

Debugging with GDB

Because Android NDK is based on the GCC toolchain, Android NDK includes GDB, the GNU Debugger, to allow starting, pausing, examining, and altering a program. On Android and more generally on embedded devices, GDB is configured in client/server mode. The program runs on a device as a server and a remote client, the developer's workstation connects to it and sends debugging commands as for a local application.

GDB itself is a command-line utility and can be cumbersome to use manually. Hopefully, GDB is handled by most IDE and especially CDT. Thus, Eclipse can be used directly to add breakpoints and inspect a program, only if it has been properly configured before!

For More Information:

www.packtpub.com/android-ndk-beginners-guide/book

Indeed, Eclipse can insert breakpoints easily in Java as well as C/C++ source files by clicking in the gutter, to the text editor's left. Java breakpoints work out of the box thanks to the ADT plugin, which manages debugging through the Android Debug Bridge. This is not true for CDT which is naturally not *Android-aware*. Thus, inserting a breakpoint will just do nothing unless we manage to configure CDT to use the NDK's GDB, which itself needs to be bound to the native Android application to debug.

Debugger support has improved among NDK releases (for example, debugging purely native threads was not working before). Although it is getting more usable, in NDK R5 (and even R7), situation is far from perfect. But, it can still help! Let's see now concretely how to debug a native application.

Time for action – debugging DroidBlaster

Let's enable debugging mode in our application first:

1. The first important thing to do but really easy to forget is to activate the debugging flag in your Android project. This is done in the application manifest `AndroidManifest.xml`. Do not forget to use the appropriate SDK version for native code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <uses-sdk android:minSdkVersion="10"/>
    <application ...
        android:debuggable="true">
        ...
```

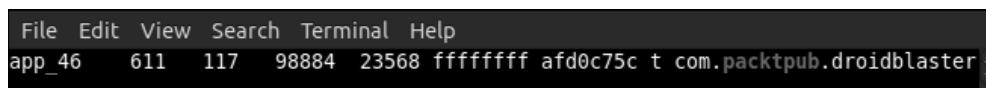
2. Enabling debug flag in manifest automatically activates debug mode in native code. However, `APP_OPTIM` flag also controls debug mode. If it has been manually set in `Android.mk`, then check that its value is set to `debug` (and not `release`) or simply remove it:

```
APP_OPTIM := debug
```

First, let's configure the GDB client that will connect to the device:

3. Recompile the project. Plug your device in or launch the emulator. Run and leave your application. Ensure the application is loaded and its PID available. You can check it by listing processes using the following command. One line should be returned:

```
$ adb shell ps | grep packtpub
```



```
File Edit View Search Terminal Help
app_46 611 117 98884 23568 ffffffff afd0c75c t com.packtpub.droidblaster
```

4. Open a terminal window and go to your project directory. Run the `ndk-gdb` command (located in `$ANDROID_NDK` folder, which should already be in your `$PATH`):

```
$ ndk-gdb
```

This command should return no message and create three files in `obj/local/armeabi`:

- `gdb.setup`: This is a configuration file generated for GDB client.
- `app_process`: This file is retrieved directly from your device. It is a system executable file (that is, **Zygote**, see Chapter 2, *Creating, Compiling, and Deploying Native Projects*), launched when system starts up and forked to start a new application. GDB needs this reference file to find its marks. It is in some way the binary entry point of your app.
- `libc.so`: This is also retrieved from your device. It is the Android standard C library (commonly referred as **bionic**) used by GDB to keep track of all the native threads created during runtime.

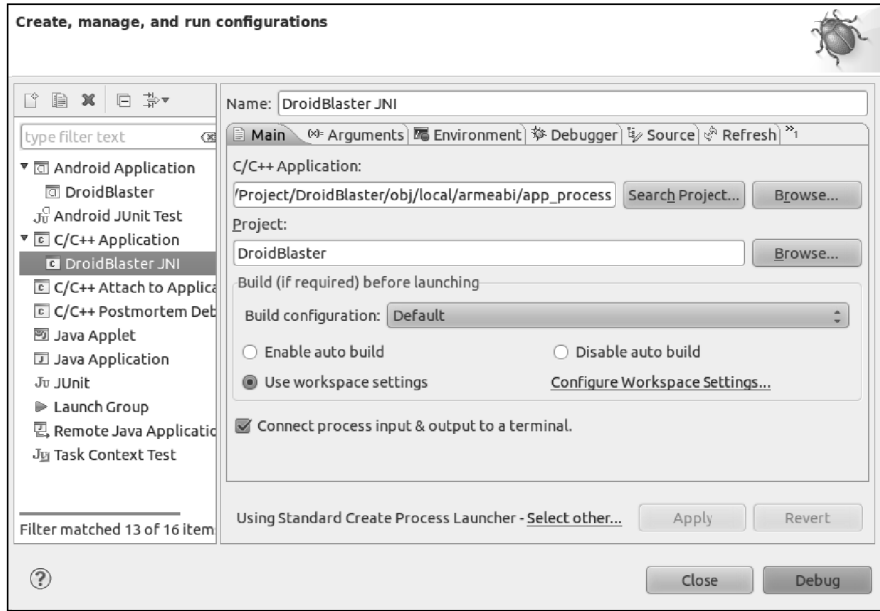


Append `-verbose` flag to have a detailed feedback on what `ndk-gdb` does. If `ndk-gdb` complains about an already running debug session, then re-execute `ndk-gdb` with the `-force` flag. Beware, some devices (especially HTC ones) do not work in debug mode unless they are rooted with a custom ROM (for example, they return a **corrupt installation** error).

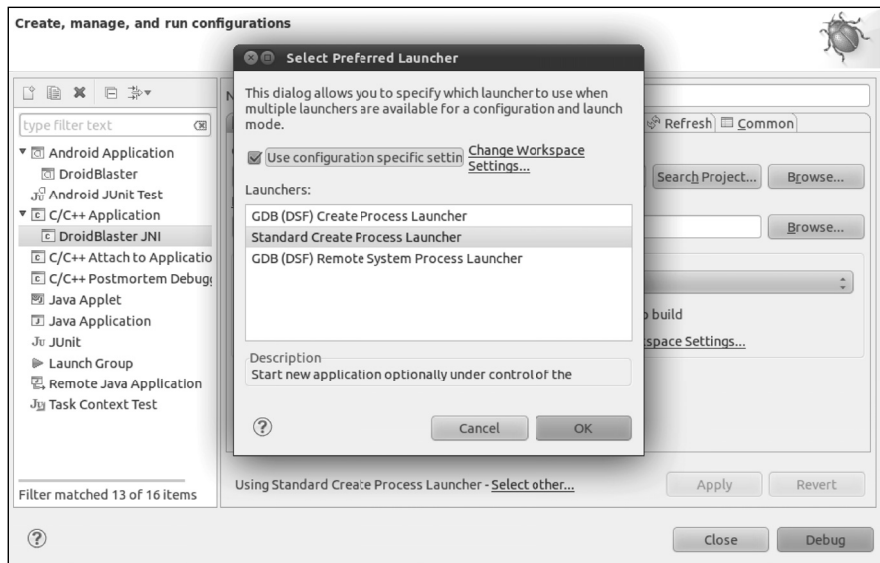
5. In your project directory, copy `obj/local/armeabi/gdb.setup` and name it `gdb2.setup`. Open it and remove the following line which requests GDB client to connect to the GDB server running on the device (to be performed by Eclipse itself):


```
target remote :5039
```
6. In the Eclipse main menu, go to **Run | Debug Configurations...** and create a new Debug configuration in the **C/C++ Application** item called `DroidBlaster_JNI`. This configuration will start GDB client on your computer and connect to the GDB Server running on the device.
7. In the **Main** tab, set:
 - **Project** to your own project directory (for example, `DroidBlaster_Part8-3`).

- ❑ **C/C++ Application** to point to `obj/local/armeabi/app_process` using the **Browse** button (you can use either an absolute or a relative path).

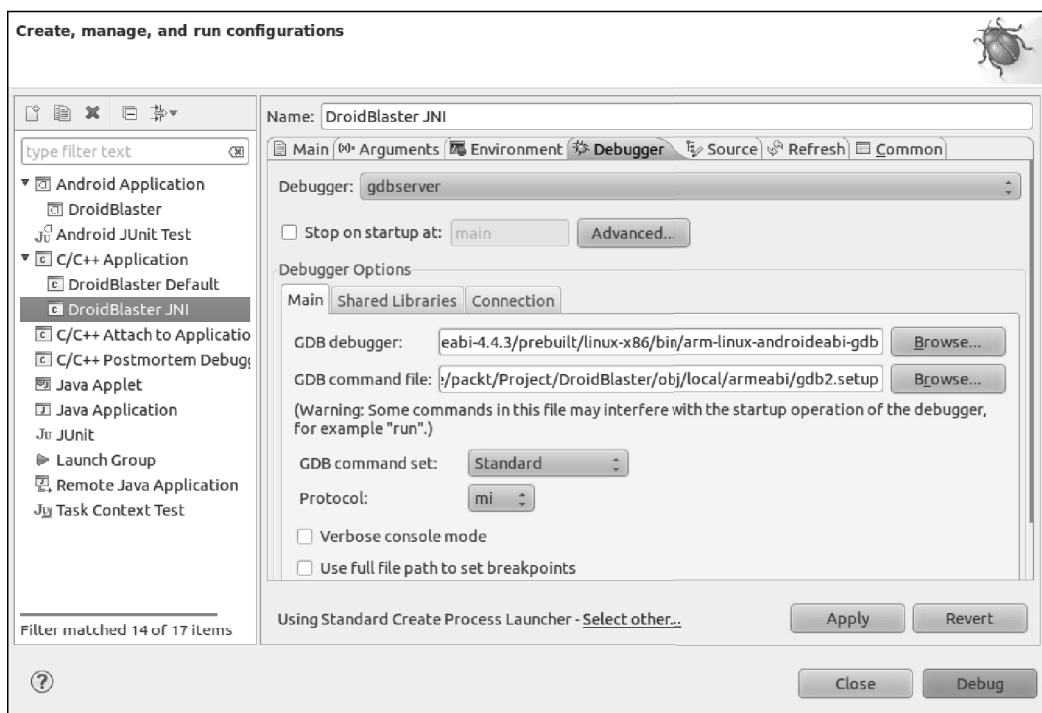


8. Switch launcher type to **Standard Create Process Launcher** using the link **Select other...** at the bottom of the window:

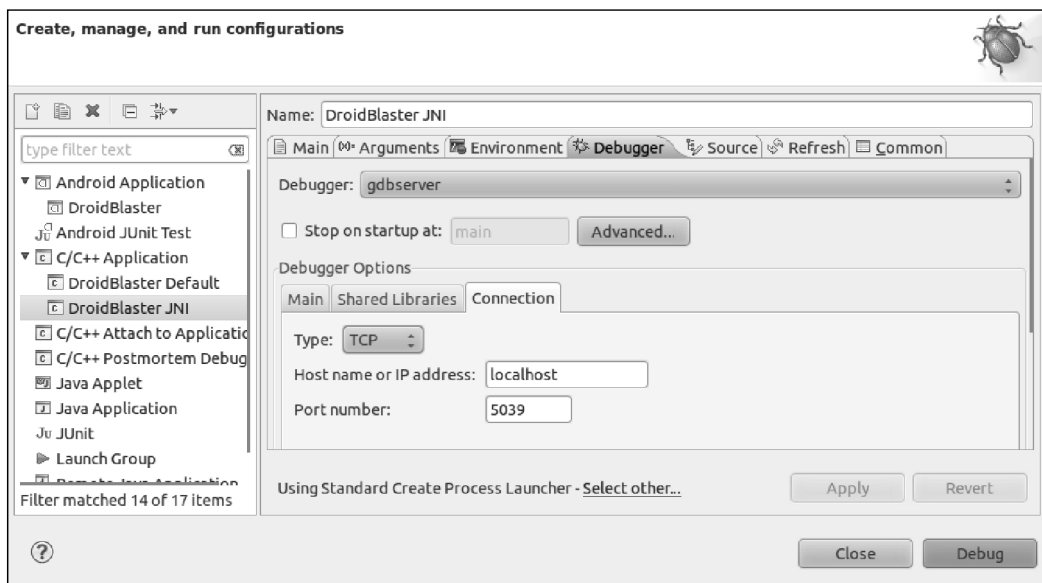


9. Go to the debugger file and set:

- ❑ **Debugger type** to **gdbserver**.
- ❑ **GDB debugger** to `${ANDROID_NDK}/toolchains/arm-linux-androideabi-4.4.3/prebuilt/linux-x86/bin/arm-linux-androideabi-gdb`.
- ❑ **GDB command file** to point to the `gdb2.setup` file located in `obj/local/armeabi/` (you can use either an absolute or a relative path).



10. Go to the **Connection** tab and set **Type** to **TCP**. Default value for **Host name or IP address** and **Port number** can be kept (**localhost** and **5039**).



Now, let's configure Eclipse to run GDB server on the device:

11. Make a copy of `$ANDROID_NDK/ndk-gdb` and open it with a text editor. Find the following line:

```
$GDBCLIENT -x `native_path $GDBSETUP`
```

Comment it because GDB client is going to be run by Eclipse itself:

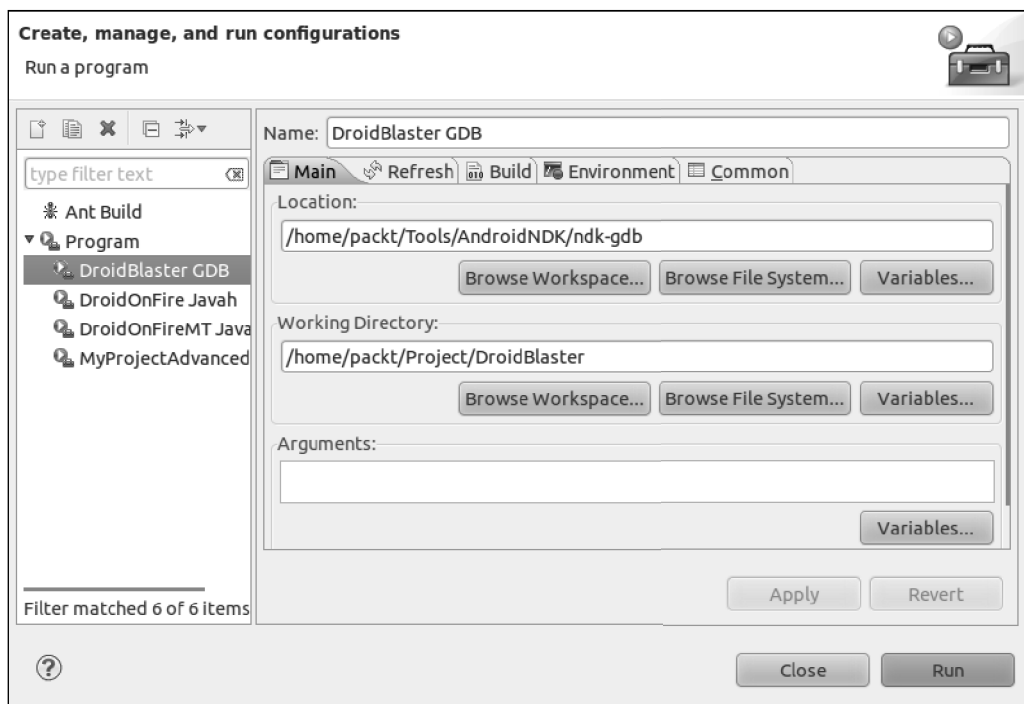
```
#$GDBCLIENT -x `native_path $GDBSETUP`
```

12. In the Eclipse main menu, go to **Run | External Tools | External Tools Configurations...** and create a new configuration `DroidBlaster_GDB`. This configuration will launch GDB server on the device.

13. In the **Main** tab, set:

- **Location** pointing to our modified `ndk-gdb` in `$ANDROID_NDK`. You can use **Variables...** button to define Android NDK location in a more generic way (that is, `${env_var:ANDROID_NDK}/ndk-gdb`).
- **Working directory** to your application directory location (for example, `${workspace_loc:/DroidBlaster_Part8-3}`)

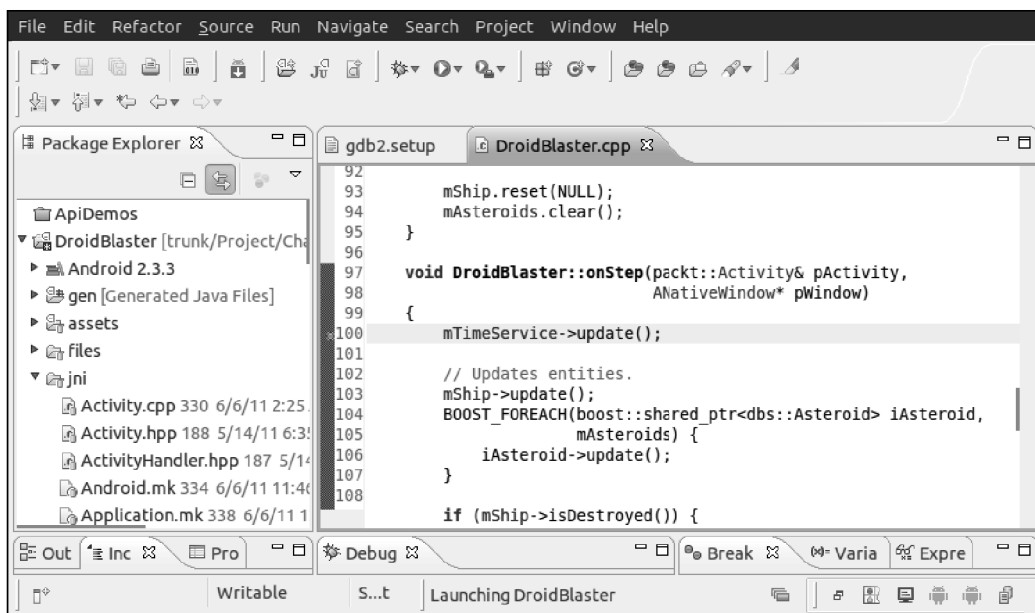
- ❑ Optionally, set the **Arguments** textbox:
- ❑ `--verbose`: To see in details what happens in the Eclipse console.
- ❑ `-force`: To kill automatically any previous session.
- ❑ `-start`: To let GDB Server start the application instead of getting attached to the application after it has been started. This option is interesting if you debug native code only and not Java but it can cause troubles with the emulator (such as to leave the back button).



We are done with configuration.

- 14.** Now, launch your application as usual (as shown in Chapter 2, *Creating, Compiling, and Deploying Native Projects*).
- 15.** Once application is started, launch the external tool configuration **DroidBlaster GDB** which is going to start GDB server on the device. GDB server receives debug commands sent by the remote GDB client and debugs your application *locally*.

16. Open `jni/DroidBlaster.cpp` and set a breakpoint on the first line of `onStep()` (`mTimeService->update()`) by double-clicking on the gutter on the text editor's left (or right-clicking and selecting **Toggle breakpoint**).

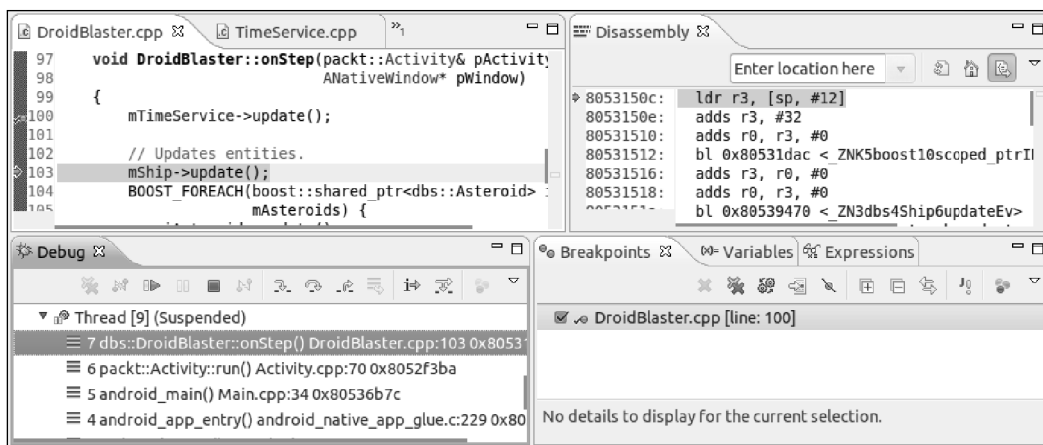


17. Finally, launch DroidBlaster JNI C/C++ application configuration to start GDB client. It relays debug commands from Eclipse CDT to GDB server over a socket connection. From the developer's point of view, this is almost like debugging a local application.

What just happened?

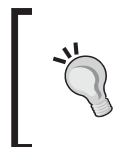
If set up properly, application freezes after a few seconds and Eclipse focuses into the *breakpointed* line. It is now possible to step into, step out, step over a line of code or resume application. For assembly-addict, an instruction stepping mode can also be activated.

Now, enjoy the benefit of this modern productivity tool, that is, a debugger. However, as you are going or maybe are already experiencing, beware that debugging on Android is rather slow (because it needs to communicate with the remote Android device) and somewhat unstable though it works well most of the time.



If the configuration process is a bit complicated and tricky, the same goes for the launch of a debug session. Remember the three necessary steps:

1. Start the Android application (whether from Eclipse or your device).
2. Then, launch GDB server on the device (that is, the `DroidBlaster_GDB` configuration here) to attach it to the application locally.
3. Finally, start GDB client on your computer (that is, the `DroidBlaster_JNI` configuration here) to allow CDT to communicate with the GDB server.
4. Optionally, start the GDB server with the `-start` flag to make it launch the application itself and omit the first step.



Beware `gdb2.setup` may be removed while cleaning your project directory. When debugging stops working, this should be the second thing to check, after making sure that `ndk-gdb` is up and running.

However, there is an annoying limitation about this procedure: we are interrupting the program while it is already running. So how to stop on a breakpoint in initialization code and debug it (for example in `jni/DroidBlaster.cpp` on `onActivate()`)? There are two solutions:

- ◆ Leave your application and launch the GDB client. Android does not manage memory as it is in Windows, Linux, or Mac OS X: it kills applications only when memory is needed. Processes are kept in memory even after user leaves. As your application is still running, GDB server remains started and you can quietly start your client debugger. Then, just start your application from your device (not from Eclipse, which would kill it).

- ◆ Take a pause when the application starts... in the Java code! However, from a fully native application, you will need to create a `src` folder for Java sources and add a new `Activity` class extending `NativeActivity`. Then you can put a breakpoint on a static initializer block.

Stack trace analysis

No need to lie. I know it happened. Do not be ashamed, it happened to all of us... your program crashed, without a reason! You think probably the device is getting old or Android is broken. We all made that reflection but ninety-nine percent of the time, we are the ones to blame!

Debuggers are the tremendous tool to look for problems in your code. But they work in *real time* when programs run. They assume you know where to look for. With problems that cannot be reproduced easily or that already happened, debuggers become sterile.

Hopefully, there is a solution: a few utilities embedded in the NDK help to analyse ARM stack traces. Let's see how they work.

Time for action – analysing a crash dump

1. Let's introduce a fatal bug in the code. Open `jni/DroidBlaster.cpp` and modify method `onActivate()` as follows:

```
...
    void DroidBlaster::onActivate() {
        ...
        mTimeService = NULL;
        return packt::STATUS_KO;
    }
...

```

2. Open the **LogCat** view (from **Window | Show View | Other...**) in Eclipse and then run the application. Not pretty for a candid Android developer! A crash dump appeared in the logs:

```
...
*** ** * ** * ** * ** * ** * ** * ** * ** * ** * ** * ** * ** * ** * ** * ** *
Build fingerprint: 'htc_wwe/htc_bravo/bravo:2.3.3/...
pid: 1723, tid: 1743 >>> com.packtpub.droidblaster <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0000000c
    r0 a9df2e71  r1 40815c8d  r2 7cb9c28d  r3 00000000
...

```

```

ip a3400000 sp 45102830 lr 00000016 pc 80410a2c cpsr 00000030
d0 6f466e6961476e6f d1 0000000400000390
...
scr 20000012
    #00 pc 00010a2c /data/data/com.packtpub.droidblaster/
lib/libdroidblaster.so
    #01 pc 00009fcc /data/data/com.packtpub.droidblaster/
lib/libdroidblaster.so
...
    #06 pc 00011618 /system/lib/libc.so
code around pc:
80410a0c 00017ad4 00000000 b084b510 9b019001
...
code around lr:
stack:
    451027f0 00000000
    451027f4 45102870
    451027f8 804110f5 /data/data/com.packtpub.droidblaster/lib/
libdroidblaster.so
...

```

This dump contains useful information about the current program state. First it describes the error that happened: a SIGSEGV, also known as a **segmentation fault**. If you look at the faulty address, that is, `0000000c`, you will see that it is close to NULL. This is an important hint!

Then we have information about ARM register states (`rx`, `dx`, `ip`, `sp`, `lr`, `pc`, and so on). But what we are interested in comes right after this: information about where the program was when it got interrupted. These lines are highlighted in the extract above and can be identified by the words `pc` written on the line and an hexadecimal number after it. The latter expresses the **Program Counter** location, that is, which instruction was executed when problem occurred. Note that this memory address is relative to the containing library. With this piece of information, we know exactly on which instruction problem occurred... in the binary code!

3. We need somehow to translate this binary address into something understandable to a normal human being. The first solution is to disassemble completely the `.so` library. Open a terminal window and go to your project directory. Then execute the **objdump** command located in the executable directory of the NDK toolchain:

```

$ $ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/
linux-x86/bin/arm-linux-androideabi-objdump -S
./obj/local/armeabi/libdroidblaster.so > ~/disassembler.dump

```


4. This command disassembles the library and outputs each assembler instruction and location accompanied with the source C/C++ code. Open the output file with a text editor and if you look carefully, you will find the same address than the one in the crash dump, next to `pc`:

```
...
void TimeService::update()
{
10a14: b510      push    {r4, lr}
10a16: b084      sub     sp, #16
10a18: 9001      str     r0, [sp, #4]
        double lcurrentTime = now();
10a1a: 9b01      ldr     r3, [sp, #4]
10a1c: 1c18      adds   r0, r3, #0
10a1e: f000 f81f   bl     10a60 <_
ZN5packt11TimeService3nowEv>
10a22: 1c03      adds   r3, r0, #0
10a24: 1c0c      adds   r4, r1, #0
10a26: 9302      str     r3, [sp, #8]
10a28: 9403      str     r4, [sp, #12]
        mElapsed = (lcurrentTime - mLastTime);
10a2a: 9b01      ldr     r3, [sp, #4]
10a2c: 68dc      ldr     r4, [r3, #12]
10a2e: 689b      ldr     r3, [r3, #8]
10a30: 9802      ldr     r0, [sp, #8]
10a32: 9903      ldr     r1, [sp, #12]
...

```

5. As you can see, problem seems to occur when executing `mService->update()` in `jni/TimeService.cpp` instruction because of the wrong object address inserted in step 1.
6. Disassembled dump file can become quite big. For this version of `droidblaster.so`, it should be around 3 MB. But it could become tenth MB, especially when libraries such as `Irrlicht` are involved! In addition, it needs to be regenerated each time library is updated.

Hoperfully, another utility named **addr2line**, located in the same directory as `objdump`, is available. Execute the following command with the `pc` address at the end, where `-f` shows function names, `-C` demangles them and `-e` indicates the input library:

```
$ $ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/
linux-x86/bin/arm-linux-androideabi-addr2line -f -C
-e ./obj/local/armeabi/libdroidblaster.so 00010a2c

```

This gives immediately the corresponding C/C++ instruction and its location in its source file:

```
File Edit View Search Terminal Help
packt::TimeService::update()
/home/packt/Project/DroidBlaster_Part8-3/jni/TimeService.cpp:25
```

7. Since version R6, Android NDK provides **ndk-stack** in its root directory. This utility does what we have done manually using an Android log dump. Coupled with the ADB, which is able to display Android logs while in real time, crashes can be analyzed without a move (except your eyes!).

Simply run the following command from a terminal window to decipher crash dumps automatically:

```
$ adb logcat | ndk-stack -sym ./obj/local/armeabi
***** Crash dump: *****
Build fingerprint: 'htc_wwe/htc_bravo/bravo:2.3.3/
GRI40/96875.1:user/release-keys'
pid: 1723, tid: 1743 >>> com.packtpub.droidblaster <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0000000c
Stack frame #00 pc 00010a2c /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine update in /home/
packt/Project/Chapter11/DroidBlaster_Part11/jni/TimeService.cpp:25
Stack frame #01 pc 00009fcc /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine onStep in /home/
packt/Project/Chapter11/DroidBlaster_Part11/jni/DroidBlaster.
cpp:53
Stack frame #02 pc 0000a348 /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine run in /home/packt/
Project/Chapter11/DroidBlaster_Part11/jni/EventLoop.cpp:49
Stack frame #03 pc 0000f994 /data/data/com.packtpub.
droidblaster/lib/libdroidblaster.so: Routine android_main in /
home/packt/Project/Chapter11/DroidBlaster_Part11/jni/Main.cpp:31
...
```

What just happened?

We have used ARM utilities embedded in the Android NDK to locate the origin of an application crash. These utilities constitute an inestimable help and should be considered as your first-aid kit when a bad crash happens.

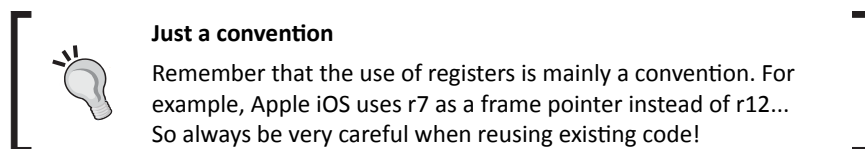
However, if they can help you finding the "where", it is another kettle of fish to find the "why". As you can see in the piece of code at step 4, understanding why `LDR` instruction (whose goal is to load in a register, some data from memory, constants, or other registers) fails is not trivial. This is where your programmer intuition (and possibly knowledge of assembly code) comes into play.

More on crash dumps

For general culture, let's linger briefly on what is provided in the LogCat crash dump. A crash dump is not dedicated only to overly talented developers or people seeing red-dressed girl in binary code, but also to those who have a minimum knowledge of assemblers and the way ARM processors work. The goal of this trace is to give as much information as possible on the current state of the program at the time it crashed:

- ◆ The first line gives the build fingerprint, which is a kind of an identifier indicating the device/Android release currently running. This information is interesting when analyzing dumps from various origins.
- ◆ The second line indicates the **PID**, process identifier, which uniquely identify an application on Unix system, and the **TID**, which is the thread identifier. It can be the same as the process identifier when crash occurs on the main thread.
- ◆ The third line shows the crash origin represented as a signal, here a classic segmentation fault (**SIGSEGV**).
- ◆ Then, processor's register values are dumped, where:
 - `rX`: This is an integer register.
 - `dX`: This is a floating point register.
 - `fp` (or `r11`): The Frame Pointer holds a fixed location on the stack during a routine call (in conjunction with the Stack Pointer).
 - `ip` (or `r12`): The **intra procedure call scratch register** may be used with some subroutine calls, for example, when the linker needs a veneer (a small piece of code) to aim at a different memory area when branching (a branch instruction to jump somewhere else in the memory requires an offset argument relative to current location, allowing a branching range of a few MB only, not the full memory).
 - `sp` (or `r13`): This is the **stack pointer**, which saves location of the top of the stack.
 - `lr` (or `r14`): The **link register** generally saves program counter's value temporarily to restore it later. A typical example of its use is a function call which jumps somewhere in the code and then go back to its previous location. Of course, several chained subroutine calls requires the link register to be stacked.

- `pc` (or `r15`): This represents the **program counter** which holds the address of next instruction to execute. Program counter is just incremented when executing a sequential code to fetch next instruction but is altered by branching instructions (if/else, a C/C++ function calls, and so on).
 - `cpsr`: The **Current Program Status Register** contains a few flags about the current processor working mode and some additional bit flags for condition codes (such as `N` for an operation which resulted in a negative value, `Z` for a 0 or equality result, and so on), interrupts, and instruction set (Thumb or ARM).
- ◆ Crash dump also contains a few memory words around PC (that is, the block of instructions around) and LR (for previous location).
 - ◆ Finally, a dump of the raw call stack is logged.



Performance analysis

If debugging tools are still imperfect, I have to advise you that profiling tools are rather immature... when they even work! Actually, there is no real official support from Google for memory or performance profiler, except in the emulator. This may change soon or later. But right now, those who like to tweak code and analyse each instruction may starve. This is particularly true when developing with a non-developer or non-rooted phone.

Hopefully, a few solutions exist and some are coming. Let's cite the following one:

- ◆ **Valgrind**: This is probably the most famous open source profiler which can monitor not only performance but also memory and cache usage. This utility is currently being ported to Android. With some tweaking, it is possible to make it work on a developer or rooted phone in ArmV7 mode. It is one of the best hopes for Android.
- ◆ **Android-NDK-Profiler**: This is a port of **Gprof** on Android. It is a simple and basic profiler which works by instrumenting and sampling code at runtime. It is the simplest solution to profile performance and does not require any specific hardware.
- ◆ **OProfile** is a system-wide profiler which inserts its code in the system kernel (which thus needs to be updated) to collect profiling data with a low overhead. It is more complicated to install and requires a developer or rooted phone to work but works quite well and does instrument code. It is a much better solution to profile code for free if you have proper hardware at your disposal.

- ◆ The commercial development suite **ARM DS-5** and its **StreamLine** performance analyzer may become an interesting option.
- ◆ Open GL ES Profilers from manufacturers: **Adreno Profiler** for Qualcomm, **PerfHUD ES** for NVidia and **PVRTune** for PowerVR. These profilers are hardware-specific. The choice depends on your phone. These tools are however essential to see what is happening under the GLES hood.

We are not going to evoke the emulator profiler here because of its inability to emulate programs properly at an effective speed (especially when using GLES). But know that it exists. Instead, we are now going to discover the interesting Android-NDK-Profiler, an alternative Gprof-based profiler ported on Android by Richard Quirk (see <http://quirkygba.blogspot.com/> for more information). Android-NDK-Profiler requires a device running at least Android Gingerbread.



Project DroidBlaster_Part8-3 can be used as a starting point for this part. The resulting project is provided with this book under the name DroidBlaster_Part11.

Time for action – running GProf

Let's try to profile our own application code:

1. Open a browser window and navigate to the Android-NDK-Profiler homepage at <http://code.google.com/p/android-ndk-profiler/>. Go to the **Downloads** section and save the latest release (3.1 at the time of writing) on your computer.
2. Unzip archive in `$ANDROID_NDK/sources/android-ndk-profiler`. This archive contains an Android Makefile and two libraries: one for Arm V5 and one for Arm V7.
3. Turn Android-NDK-Profiler into a full android module (see highlighted lines). The main missing point is the export of `prof.h` file that we are going to include in our code.

This Makefile uses the `$TARGET_ARCH_ABI` variable to select the right library version (Arm V5/V7) automatically according to what is defined in `Application.mk` (`APP_ABI= armeabi, armeabi-v7a`). It also filters some optimization options which could interfere with it (for Thumb as well as ARM code):

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

```
TARGET_thumb_release_CFLAGS := $(filter-out -ffunction-  
sections,$(TARGET_thumb_release_CFLAGS))
```

```

TARGET_thumb_release_CFLAGS := $(filter-out -fomit-frame-
pointer,$(TARGET_thumb_release_CFLAGS))
TARGET_CFLAGS := $(filter-out -ffunction-sections,$(TARGET_
CFLAGS))

# include libandprof.a in the build
include $(CLEAR_VARS)
LOCAL_MODULE := andprof
LOCAL_SRC_FILES := $(TARGET_ARCH_ABI)/libandprof.a
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/
include $(PREBUILT_STATIC_LIBRARY)

```

- 4.** Android-NDK-Profiler can now be included in a normal native library. Let's append it to `DroidBlaster_Part8-3` (you can use any other version you want).

Add the optimization filter like done in profiler's own Makefile. Since compilation is done in thumb mode by default, keep only related lines. Then include `-pg` parameter which inserts additional instruction necessary to the profiler. Finally, include profiler module as usual:

```

LOCAL_PATH := $(call my-dir)

TARGET_thumb_release_CFLAGS := $(filter-out -ffunction-
sections,$(TARGET_thumb_release_CFLAGS))
TARGET_thumb_release_CFLAGS := $(filter-out -fomit-frame-
pointer,$(TARGET_thumb_release_CFLAGS))
TARGET_CFLAGS := $(filter-out -ffunction-sections,$(TARGET_
CFLAGS))

include $(CLEAR_VARS)

LS_CPP=$(subst $(1)/,,$(wildcard $(1)/*.cpp))
LOCAL_CFLAGS := -DRAPIDXML_NO_EXCEPTIONS -pg
LOCAL_MODULE := droidblaster
LOCAL_SRC_FILES := $(call LS_CPP,$(LOCAL_PATH))
LOCAL_LDLIBS := -landroid -llog -lEGL -lGLESv1_CM -lOpenSLES

LOCAL_STATIC_LIBRARIES := android_native_app_glue png andprof

include $(BUILD_SHARED_LIBRARY)

$(call import-module,android/native_app_glue)
$(call import-module,libpng)
$(call import-module,android-ndk-profiler)

```

5. To run the profiler, we need to include a profiler start up and shut down function in the code. Open `jni/Main.cpp` and insert them at the beginning and end of `android_main()`. Set sample frequency to 6000 thanks to a predefined environment variable `CPUPROFILE_FREQUENCY`:

```
...
#include <cstdlib>
#include <prof.h>

void android_main(struct android_app* pApplication)
{
    setenv("CPUPROFILE_FREQUENCY", "60000", 1);
    monstartup("droidblaster.so");

    // Run game services and event loop.
    ...
    lEventLoop.run(&lDroidBlaster, &lInputService);

    moncleanup();
}
```

6. Finally, allow application to write on a storage in `AndroidManifest.xml`:

```
<?xml xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.packtpub.droidblaster" android:versionCode="1"
    android:versionName="1.0">
    ...
    <uses-permission
        android:name="android.permission.WRITE_EXTERNAL_
STORAGE"/>
</manifest>
```

7. Recompile `DroidBlaster` project. It now includes all the necessary instructions to start profiler and generate profiling information.
8. Run project on a device. Log messages are generated between profiler startup and shutdown. Make sure application completely dies by pressing the back button, a pause being not sufficient:

```
INFO/threaded_app(3553): Start: 0x97270
INFO/PROFILING(3553): Profile droidblaster.so 80400000-8043d000: 0
INFO/PROFILING(3553): 0: parent: carrying on
INFO/PACKT(3553): Creating GraphicsService
...
```

```

INFO/PACKT(3553): Exiting event loop
INFO/PROFILING(3553): parent: moncleanup called
INFO/PROFILING(3553): 1: parent: done profiling
INFO/PROFILING(3553): writing gmon.out
INFO/PROFILING(3598): child: finished monitoring
INFO/PACKT(3553): Destructing DroidBlaster

```

9. After application is terminated, retrieve file `gmon.out` generated in the `/sdcard` folder of your device (depending on your device, storage may be mounted in another directory) and save it in your project directory. Do not forget to activate USB Mass Storage mode to see files from your computer.

10. From a terminal window located in your project directory where `gmon.out` is saved, open a terminal and run `gprof` analyser located beside NDK ARM toolchain binaries:

```

$ ANDROID_NDK/toolchains/arm-linux-androideabi-4.4.3/prebuilt/
linux-x86/bin/arm-linux-androideabi-gprof obj/local/armeabi/
libdroidblaster.so

```

This command generates a textual output that you can redirect to a file. It contains all profiling results. The first part (flat profile) is the consolidated result with top functions which seem to take time. The second part is the *raw* index from which the first part is calculated:

Flat profile:

Each sample counts as 1.66667e-05 seconds.

%	cumulative	self	self	total		name
time	seconds	seconds	calls	us/call	us/call	
18.64	0.00	0.00				png_read_
						filter_row
13.56	0.00	0.00	15847	0.01	0.02	packt::Graph
						icsService::update()
10.17	0.00	0.00	15847	0.01	0.01	packt::Graph
						icsSprite::draw(float)
10.17	0.00	0.00	1	100.00	566.67	packt::EventLoop::run(...)
8.47	0.00	0.00	15847	0.01	0.03	dbs::DroidBlaster::onStep()
5.08	0.00	0.00	15847	0.00	0.00	packt::GraphicsTileMap::draw()
...						

index	% time	self	children	called	name
					<spontaneous>
[1]	57.6	0.00	0.00		android_main [1]
		0.00	0.00	1/1	
		packt::EventLoop::run(...) [2]			
		0.00	0.00	1/1	packt::EventLoop:
		:EventLoop(android_app*) [469]			
		0.00	0.00	1/1	packt::Sensor::Se
		nsor(packt::EventLoop&, int) [466]			
		0.00	0.00	1/1	packt::TimeServic
		e::TimeService() [433]			
		0.00	0.00	1/1	packt::GraphicsSe
		rvice::GraphicsService(...) [456]			
		...			

What just happened?

We have compiled Android-NDK-Profiler project as an NDK module and appended it to our own project. We turned profiling on with the help of two exported methods `monstartup()` and `moncleanup()`. The profiling result is written to `gmon.out` file on the SD Card (thus requiring write access) that can be parsed by the NDK `gprof` utility.

The output file contains a summary for each function hit by the sampler: the flat profile. More specifically, it indicates the following:

- ◆ `index`: This identifies an entry in the index computed from and written after the flat profile.
- ◆ `% time`: This represents the fragment of time spent in the function compared to the total program execution times.
- ◆ `cumulative seconds`: This is the accumulated total time spent in the function and all the function above in the table (using `self seconds`).
- ◆ `self seconds`: This is the accumulated total time spent in the function itself over its multiple execution.
- ◆ `calls`: This represents the total number of calls to a function. This is the only information which is really accurate.
- ◆ `self s/call`: This is the average time spent in one execution of the function. This column depends on sample hits and is not reliable.
- ◆ `total s/call`: This is the same as `self s/call` but cumulated with the time spent in sub-functions too. This column is also depends on sample hits.

Note that functions in which no *apparent* time is spent (which does not mean they are never called) are not mentioned unless `-z` is appended to command-line options.

How it works

To profile a piece code, GCC compiler instruments your code when option `-pg` is appended to compilation options. Instrumentation relies on a routine named `mcount()` (more formerly `__gnu_mcount_nc()`) which is inserted at the beginning of each function to gather information about its caller and compute call count indicator. The role of Android-NDK-Profiler here is to implement this routine which is not provided by the Android NDK.

More advanced profiling information is extracted by sampling the PC counter at constant intervals (100hz by default), in order to detect which function the program is currently running (and derive the call stack). From a theoretical point of view, the more a function takes time to run, the bigger is the probability that a sample hits it.

To do so, Android-NDK-Profiler creates a separate thread to collect timing information and a new fork process to interrupt native code and record samples. To do so, it requires the ability to attach to a parent process which only works from Android 2.3 Gingerbread. Thus, if you see the following message in Android logs, profiling information will not get collected accurately:

```
INFO/PROFILING(3588): child: could not attach 3584
```

GProf is a mature (not to say antic) tool which has limitations. First, GProf instrumentation is intrusive. It affects performance and potentially cache usage which result in perturbations. Moreover, it does not measure time spent in I/O which is often a good place to look for bottlenecks and does not handle recursion. Finally, because it uses sampling and makes some assumption about code (for example, a function is assumed to use more or less the same time to run for each call), GProf does not give very accurate results and needs many samples to increase accuracy. This makes it difficult to analyze results properly, when they are not misleading.

Although it is far from perfect, GProf is still easy to set up and can be a good start in profiling.


ARM, thumb, and NEON

Compiled native C/C++ code on current Android ARM devices follows an **Application Binary Interface (ABI)**. An ABI specifies the binary code format (instruction set, calling conventions, and so on). GCC translates code into this binary format. ABIs are thus strongly related to processors. The target ABI can be selected in the `Application.mk` file with the property `APP_ABI`. There exist four main ABIs supported on Android:

- ◆ **thumb**: This is the default option which should be compatible with all ARM devices. Thumb is a special instruction set which encodes instructions on 16-bit instead of 32 to improve code size (useful for devices with constrained memory). The instruction set is severely restricted compared to ArmEABI.


- ◆ **armeabi** (Or Arm v5): This should run on all ARM devices. Instructions are encoded on 32-bit but may be more concise than Thumb code. Arm v5 does not support advanced extensions like floating point acceleration and is thus slower than Arm v7.
- ◆ **armeabi-v7a**: This supports extensions such as Thumb-2 (similar to Thumb but with additional 32-bit instructions) and VFP plus some optional extensions such as NEON. Code compiled for Arm V7 will not run on Arm V5 processors.
- ◆ **x86**: This is for *PC-like* architectures (that is, Intel/AMD). There is no official device that existed at the time this book was written but an unofficial open source initiative exists.

It is possible to compile code, for example, for Arm V5 and V7 at the same time, the most appropriate binaries are selected at installation time.

 Android provides a `cpu-features.h` API (with `android_getCpuFamily()` and `android_getCpuFeatures()` methods) to detect available features on the host device at runtime. It helps in detecting the CPU (ARM, X86) and its capabilities (ArmV7 support, NEON, VFP).

Performance is one of the main criteria to develop with the Android NDK. To achieve this, ARM created a SIMD instruction set (acronym Single Instruction Multiple Data, that is, process several data in parallel with one instruction) called NEON which has been introduced along with the VFP (the floating point accelerated unit).

NEON is not available on all chips (for example, Nvidia Tegra 2 does not support it) but is quite popular in intensive multimedia application. They are also a good way to compensate the weak VFP unit of some processors (for example, Cortex-A8).

 NEON code can be written in a separate assembler file, in a dedicated `asm volatile` block with assembler instructions or in a C/C++ file or as intrinsics (NEON instructions encapsulated in a GCC C routine). Intrinsics should be used with much care as GCC is often unable to generate efficient machine code (or requires lots of *tricky* hints). Writing real assembler code is generally advised.

NEON and modern processors are not easy to master. The Internet is full of examples to get inspiration from. For example, have a look at code.google.com/p/math-neon/ for an example of math library implemented with NEON. Reference technical documentation can be found on the ARM website at <http://infocenter.arm.com/>.

Summary

In this last chapter, we have seen advanced techniques to troubleshoot bugs and performance issues. More specifically, we have debugged our code with the native code debugger, which is slow and complex to set up but is a real life saver.

We have also executed NDK Arm utilities to decipher crash dumps. They are the ultimate solution when a crash already occurred.

Finally, we have profiled our code to analyze performances with GProf. This solution is limited but can give an interesting overview.

With these tools in hand, you are now ready to venture out into the NDK jungle. And if you are adventurous, you can dive head first in ARM assembler to improve performances drastically . However, beware this is useful only when targeting the right pieces of code (the famous 20%!). Do not forget that optimizing a bad algorithm will never make it good, and a good algorithm even without optimization can make a huge difference.

Where to buy this book

You can buy Android NDK Beginner's Guide from the Packt Publishing website:
<http://www.packtpub.com/android-ndk-beginners-guide/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



For More Information:
www.packtpub.com/android-ndk-beginners-guide/book