Chapter 5

Logging, Debugging, and Troubleshooting

In previous chapters, you explored the Android NDK build system and how to connect the native code to the Java application using the JNI technology. Needless to say, learning application development on a new platform involves much experimentation; it takes time to get things right. It is vital to gain the troubleshooting skills pertaining to Android platform before starting to experiment with the native APIs offered, as it can catalyze the learning phase greatly by helping you to spot problems quickly. Your existing troubleshooting skills may not directly apply since the development and execution of Android applications happens on two different machines. In this chapter you will explore logging, debugging, and troubleshooting tools and techniques including:

- An introduction to Android Logging framework
- Debugging native code through Eclipse and command line
- Analyzing stack traces from crashes
- Using CheckJNI mode to spot problems earlier
- Troubleshooting memory issues using libc and Valgrind
- Using strace to monitor native code execution

Logging

Logging is the most important part of troubleshooting, but it is tricky to achieve, especially on mobile platforms where the development and the execution of the application happen on two different machines. Android has an extensive logging framework that promotes system-wide centralized logging of information from both the Android system itself and the applications. A set of user-level applications is also provided to view and filter these logs, such as the logcat and Dalvik Debug Monitor Server (DDMS) tools.

O. Cinar, *Pro Android C++ with the NDK*

Framework

The Android logging framework is implemented as a kernel module known as the *logger*. The amount of information being logged on the platform at any given time makes the viewing and analysis of these log messages very difficult. In order to simplify this procedure, the Android logging framework groups the log messages into four separate log buffers:

- *Main*: Main application log messages
- *Events*: System events
- *Radio*: Radio-related log messages
- *System*: Low-level system debug messages for debugging

These four buffers are kept as pseudo-devices under the `/dev/log` system directory. Since input and output (I/O) operations on mobile platforms are very costly, the log messages are not saved in persistent storage; instead, they are kept in memory. In order to keep the memory utilization of the log messages under control, the logger module puts them in fixed-sized buffers. Main, radio, and system logs are kept as free-form text messages in 64KB log buffers. The event log messages carry additional information in binary format, so they are kept in a 256KB log buffer.

Native Logging APIs

Developers are not expected to directly interact with the logger kernel module. The Android runtime provides a set of API calls to allow both Java and the native code to easily send log messages to the logger kernel module. The logging API for the native code is exposed through the `android/log.h` header file. In order to use the logging functions, native code should include this header file first.

```
#include <android/log.h>
```

In addition to including the proper header file, the `Android.mk` file needs to be modified dynamically to link the native module with the log library. This is achieved through the use `LOCAL_LDLIBS` build system variable, as shown in Listing 5-1. This build system variable must be placed before the `include` statement for the shared library build fragment; otherwise, it will not have any affect.

Listing 5-1. Dynamically Linking the Native Module with Log Library

```
LOCAL_MODULE := hello-jni
...
LOCAL_LDLIBS += -llog
...
include $(BUILD_SHARED_LIBRARY)
```