

Debugging the Android Software Platform on ARM-based Designs

By Achim Nohl, Principal Solutions Specialist, CoWare, Inc.

Bringing up Android on a new device cannot be done without touching the internals of the Android Software Stack. While the Java application layer remains to a large extent unmodified, work has to be done in the Java Native Layer, the Hardware Adaptation Layer and the underlying Linux and its drivers. Furthermore, device specific services and applications need to be integrated and tested along with the rest of Android. The vertical integration of a device feature across all those layers in the software stack is a fair challenge. Isolating and debugging one software module in a single layer can already be tough, but debugging multiple interacting modules across all layers is not supported by any debug framework and requires patience and many 'printf' trace messages.

CoWare's Virtual ARM Development Board for Android provides an Android-aware debug and analysis framework allowing for a deterministic and successive top-down debug approach. The Android-awareness is first noticeable by the ability to detect and trace ARM Linux operating system contexts such as interrupt handlers, kernel threads, drivers as well as user space processes. The ability to observe the scheduling of processes already gives a lot of insight into the overall system behavior. The figure below shows an online context trace of a system deadlock after a device resumes from a deep sleep suspend mode. While the device is waking up but not reacting on any keys or touch-screen, the system is trapped in a deadlock between a software interrupt (swi) handler and Android's input device reader process (InputDeviceRead). With this analysis it becomes easily obvious that no other important process such as the window manager are instructed to handle the key.

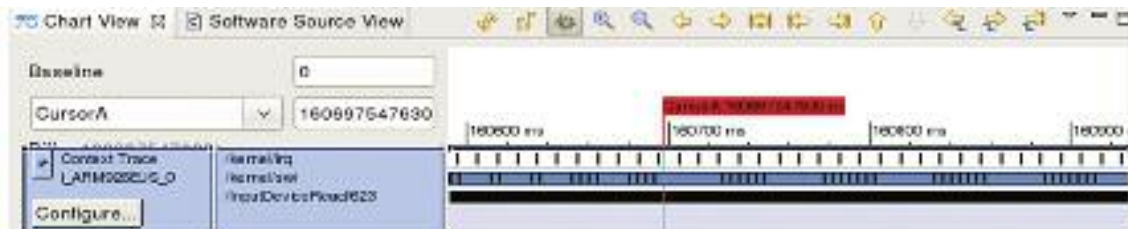


Figure 1: Context Trace of a System Deadlock after Resume from Deep Sleep

Supporting the top down debug flow, each process can be further investigated on the function up to the instruction level. Even memory accesses are traced which allows an effective analysis of behaviors across layers in the software stack. The figure below illustrates the function level interaction between the Android input device reader process and the kernel software interrupt. First of all it allows a good understanding which code gets actually executed. This is of tremendous help already to isolate the locations in the large 3rd party software code base that requires the developer's attention. Second, it allows understanding how software interacts across the various layers. In the figure we can observe how an Android middleware function "read_notify" triggers the kernel software interrupt "swi".

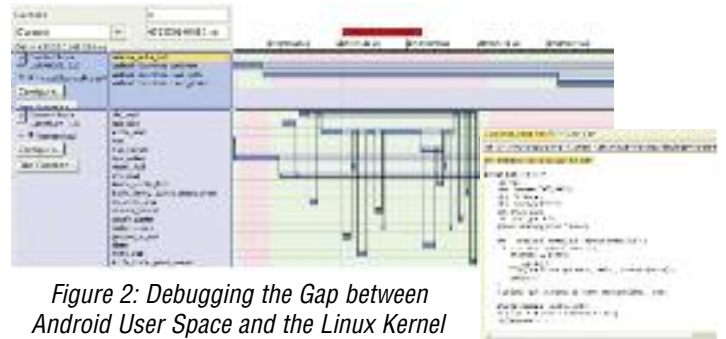


Figure 2: Debugging the Gap between Android User Space and the Linux Kernel

CoWare's analysis solution integrates the existing logging and tracing capabilities that exist in Android and Linux. Without instrumentation or changes in any parts of the software, the Android logger and Kernel debug messages are captured and visualized along with the previously introduced traces. This way,

developers can continue to use those facilities with the added value that all logs and traces are synchronized and can be easily correlated with the process and software function as shown in the figure 3 on the next page.

