From the Debug Research Labs: Debugging Android

Hagen Patzke, Software Design Embedded Debugging, Lauterbach GmbH

Synopsis

Android comes with good support for developing and debugging: High-level (Java™) application debugging is well covered by debug support in the Dalvik interpreter. Debugging “native” parts of the platform, like system services written in C/C++ that run in their own process, can be done with relative ease with an included GNU Debug Server. And if you want to port to a new platform, you can use mature hardware-assisted debug tools for the initial debugging of low-level drivers and the kernel itself. So all is perfect - end of article. Really?

Welcome to the other side, if you are a system developer who needs to track a bug that spans several of these worlds. Or if you need to find a bug that does not show itself as soon as debug assistance is enabled. Or perhaps you work with a production-stage or secure platform where no software-assisted debugging is allowed. Here, things can become really difficult.

This article gives a short introduction to embedded debugging in general, covering the different abstraction levels involved, and some special aspects of Android debugging.

The Platform

Android probably is such a success because it is a complete top-to-bottom integrated platform. Everything is well specified to build a working, useful device. Furthermore, when Google™ made it available as "open source", both an emulator and a real device, ARM™ implementation was provided that actually prove its function in the real world.

At Android’s heart is a specially adapted Linux 2.6 kernel. This is what makes it tick and among lots of other functions, it provides multi-threading for services and for virtual machine processes.

Native code and virtual machine programs together form the Android “system”.

Android Debug Inventory

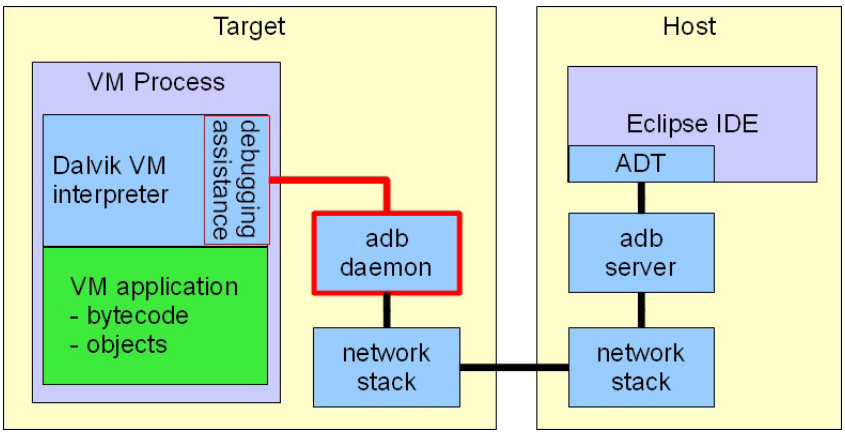
Application developers are pretty well catered for: a very good SDK is available and an active community provides support. With a free Eclipse plug-in you can not only develop your Java™/Dalvik application, but by using an extended version of the Java Wired Debug Protocol (JWDP) via the Android Debug Bridge (ADB), with aid from the Dalvik Virtual Machine (VM) and a debug daemon on the device, you can also quite efficiently debug it.

The same is true if you write “native code”, e.g. to do application code heavy-lifting in a service process. You can use a GNU Debug Server to attach to your process and debug it.

Stop-Mode Debugging: The platform with all operating system and application processes is suspended in “debug mode”; all state is frozen for inspection by the debugger. This usually requires external debug hardware, very often connected using a JTAG test access port.
Run-Mode Debugging: The platform executes the operating system and all tasks that are currently not debugged. This mode usually requires either changes to the operating system, or at least a helper application (a so-called “debug server”) executed on the platform. Android has several debug helpers: e.g. the Android Debug Bridge Daemon (adbd) for managing connections to the host, built-in Dalvik VM debug support, and a GNU Debug Server (gdbserver) for native processes.

If all you need is this type of “platform assisted run-mode debugging”, you will be fine. Happy coding and debugging!

High-level languages that are not compiled, but interpreted, like Java™, need help from their Virtual Machine (VM) interpreter for debugging. This is also true for Android and if you stop the physical machine, the communication link between the external Java/Dalvik debugger and the debugged application is disrupted.



“Assisted” Virtual Machine (VM) application debugging

The same is true for external native code debugging - if you stop the operating system kernel also the provided GNU Debug Server process is halted.

You might be a system developer porting Android to a new platform. Or building new low-level services that interface closely with very low (device) and very high level (GUI) components. Most of the time debugging these individual system components separately will work well enough.

But imagine you need to change core components such as the network stack. As soon as you hit a hardware breakpoint in its low-level driver, you lose all network communication between host and target, rendering the network-based debug assistance inoperative. Or what if you need “post-mortem” debugging, i.e. hooking up to a “frozen” device to see what happened?

In this case you want both "native debugging" plus "Java debugging" integrated into the same hardware debugger, to track and shoot the difficult-to-find bugs that span the worlds.

Debugging and Tracing Basics

Before we delve into the different debug “building blocks”, let us think a moment about what the basic function of any debugger on an embedded platform or “target” is:

A debugger maps a snapshot of a physical machine's state to an abstraction on one or more levels that are a virtual representation of the programmer's intent. (Please continue breathing.)

Put bluntly, you are not thrilled about a program counter value of 0x123456, but want to see that this is in line fourteen of your MP3 player source code. You are also not interested about value 0x1003 in processor register number three, but it can be crucial to know that this means “playback_active” for the state machine of the application.

It is also useful to “trace” program execution over a period of time, e.g. for “profiling” to find out where the execution time is spent in an application and for infrequently occurring bugs. For this, “program trace” and “system trace” hard- and software are available. On some platforms you can not only trace program counter values (or branches) over time but also changes in data memory. As the program of a virtual machine is just ordinary data for the physical machine that runs the virtual machine (VM) interpreter, data trace capability is very useful if you have a VM.

To sum up, a “debugger”s core task is to map raw memory and register data to something that is meaningful for you. Recording (selected) system state changes over a period of time is called “tracing”, and this is useful for identifying performance bottlenecks or for finding intermittent bugs.

Target and Host

On a PC, most application debugging is done with software debuggers that run inside the very same machine that also runs the application and the development framework. Most operating systems and the PC architecture itself, actively support this type of "software-based" debugging.

Embedded platforms are different: Most target devices are constrained in processing power, memory and available interfaces. Therefore development and debugging for them usually take place on an external "host" machine that is powerful enough for the job.

Now think a moment about your embedded device infrastructure (boot loader, drivers, operating system). Whatever it is, you need to build the software, which often means “cross-compiling” for your “target” device architecture; more often than not the PC “host” and the target architecture are not the same. Then you need to load it onto the embedded device.

Finally, you can start your software and debug it. This can be an application, if everything else is OK, or you may need to debug the infrastructure first, because something went wrong before the start-application code could execute. The traditional PC approach of "debug on the platform itself" does not really work here.

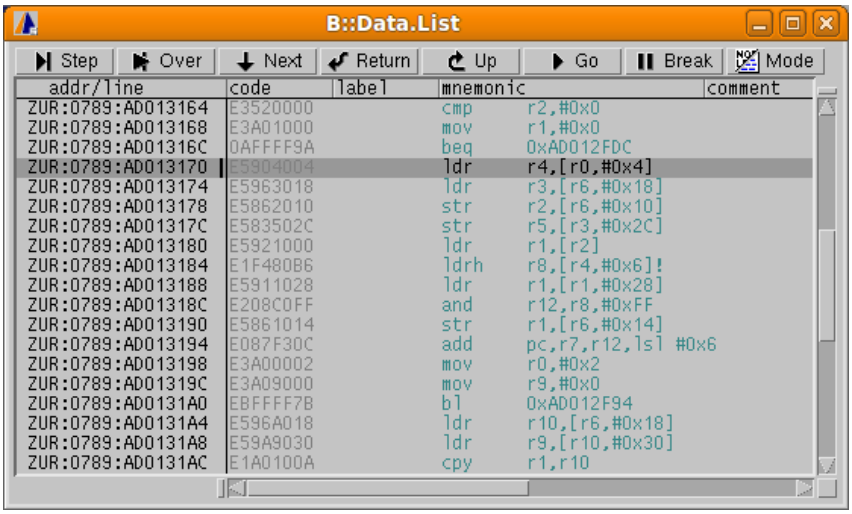
In the embedded world, one standard method to load and debug software is connecting the target via a hardware interface like JTAG (IEEE1149.1) to external debug hardware. This in turn is controlled by a graphical user interface (GUI) running on the "host". Via the GUI or using a script language you can change its memory content to “download” software onto your target and then execute and debug your system and application.

Native Code Debugging

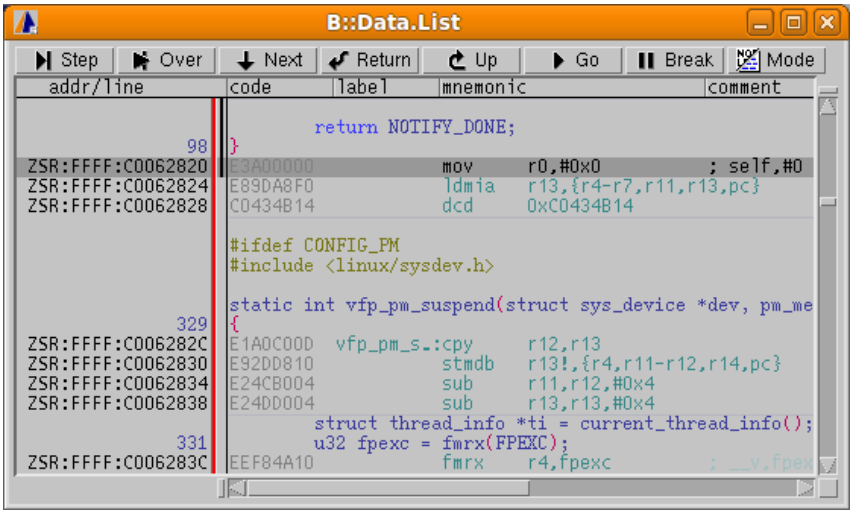
Let's look at the mapping of machine state to abstraction levels. Lowest are on-chip signal and voltage levels. One level up are bits in registers and in memory. Another level up you look at numbers (represented e.g. in decimal or “hex”). On the next level you can finally see assembly language instructions. This is where unaided debugging stops.

Luckily modern compilers emit "debug info", mapping assembly locations to high-level language (e.g. C/C++) source code lines. Other “debug records” map logical data types to the data layout in memory. This additional debug information makes program files pretty big, but without it bug-hunting is mostly "poking in the dark": you really want to map the current program counter to a line in a high-level language source file. Only then you can see what's going on in the “native machine” code running on your physical processor core, and on the abstraction level you desire.

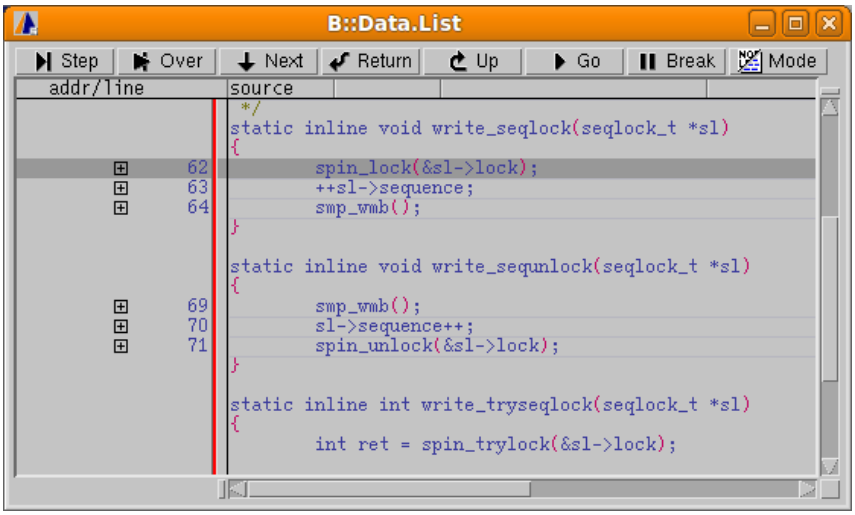
What can you do with a native code debugger? In short: selectively start and stop program execution on the available processor core(s), inspect and manipulate memory, core and peripheral device registers, and set and delete breakpoints (to stop your program at a predetermined location).



Numerical (code) and Assembly Language (mnemonic) abstraction levels



Mixed Assembly Language and High-Level (C) Language abstraction level



High-Level (C) Language abstraction level

Kernel Debugging and OS Awareness

Enter the operating system kernel. Modern platforms support multi-tasking and multi-threading, i.e. more than one program or process can be executed at the same time. This is also true for single-processor cores, the operating system just gives every process a “time slice” and then switches to the next one.

Now you have not only one application that is easy to find and debug, but multiple applications running at the “same” time. Maybe even multiple instances of the same application are active at once of which you want to debug only one.

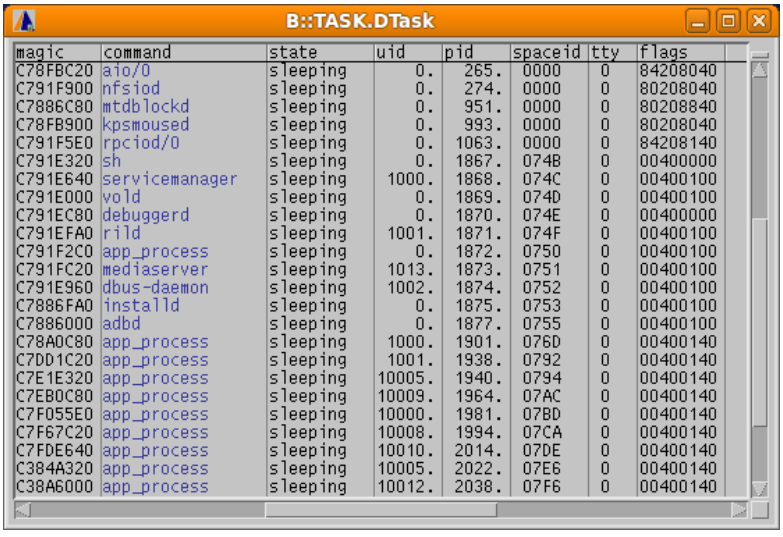
As we discussed, pure "native code debugging" nicely allows you to find problems in operating system boot code, device drivers, or in low-level constructs like the task scheduler.

However, if you want to know in which part of program memory your current application process executes and where its instance variables are (you may have more than one running, right?), you need the debugger to “know” the operating system. You need a debugger that has "OS awareness".

Well, for a debugger this is actually not quite so easy to provide. Unlike processor cores and high-level language compilers (C/C++, for example), the operating system (OS) itself might not be a fixed "off the shelf" entity like it is on most PCs. In the embedded world the OS is often something you need to actively change and adapt to build a new competitive product.

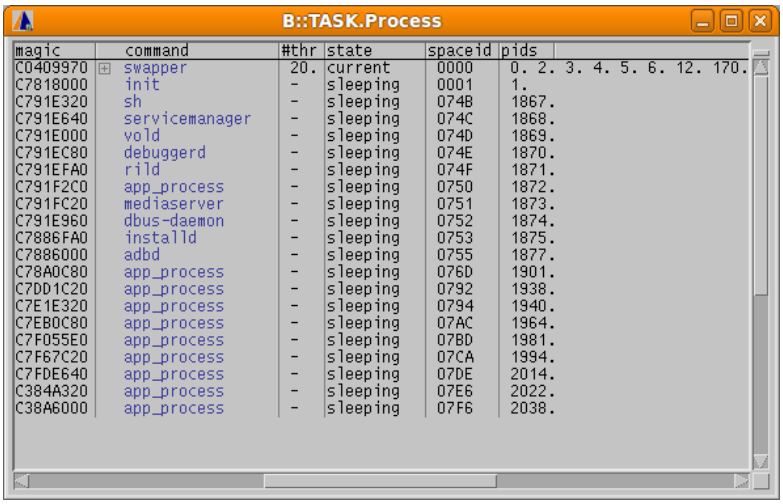
An OS you can change is a "moving target" for debugging, and this makes it necessary to configure and adapt the debugger "OS awareness" to your very own operating system variant.

This became apparent when embedded Real-Time Operating Systems (RTOS) became popular, and in response Lauterbach implemented a "TRACE32 Extension" mechanism, complete with an Extension Development Kit (EDK). With this we can either adapt an existing OS awareness (e.g. for Linux), or a customer can write their own, if this is necessary.

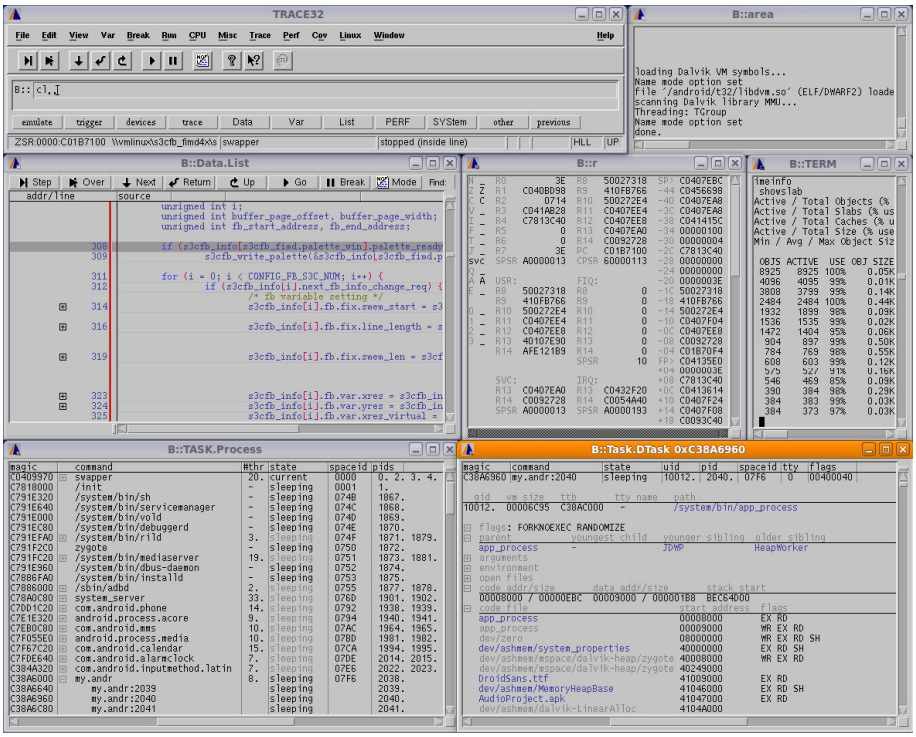


magic	command	state	uid	pid	spaceid	tty	flags
C78FBC20	aio/0	sleeping	0.	265.	0000	0	84208040
C791F900	nfsiod	sleeping	0.	274.	0000	0	80208040
C7886C80	mtdblockd	sleeping	0.	951.	0000	0	80208840
C78FB900	kpsmoused	sleeping	0.	993.	0000	0	80208040
C791F5E0	rpciod/0	sleeping	0.	1063.	0000	0	84208140
C791E320	sh	sleeping	0.	1867.	074B	0	00400000
C791E640	servicemanager	sleeping	1000.	1868.	074C	0	00400100
C791E000	vold	sleeping	0.	1869.	074D	0	00400100
C791EC80	debuggerd	sleeping	0.	1870.	074E	0	00400000
C791EFA0	rild	sleeping	1001.	1871.	074F	0	00400100
C791F2C0	app_process	sleeping	0.	1872.	0750	0	00400100
C791FC20	mediaserver	sleeping	1013.	1873.	0751	0	00400100
C791E960	dbus-daemon	sleeping	1002.	1874.	0752	0	00400100
C7886FA0	installd	sleeping	0.	1875.	0753	0	00400100
C7886000	adbd	sleeping	0.	1877.	0755	0	00400100
C78A0C80	app_process	sleeping	1000.	1901.	076D	0	00400140
C7DD1C20	app_process	sleeping	1001.	1938.	0792	0	00400140
C7E1E320	app_process	sleeping	10005.	1940.	0794	0	00400140
C7EB0C80	app_process	sleeping	10009.	1964.	07AC	0	00400140
C7F055E0	app_process	sleeping	10000.	1981.	07BD	0	00400140
C7F67C20	app_process	sleeping	10008.	1994.	07CA	0	00400140
C7FDE640	app_process	sleeping	10010.	2014.	07DE	0	00400140
C384A320	app_process	sleeping	10005.	2022.	07E6	0	00400140
C38A6000	app_process	sleeping	10012.	2038.	07F6	0	00400140

Linux Kernel Tasks



Linux Process Display



Process Display with VM Application Names

Virtual Machines...

Processors have become more and more powerful. This has made it possible to abstract even the hardware–Virtual Machines (VM), formerly a domain of big mainframes, arrived in the embedded world.

If you run code in a Virtual Machine (VM), instead of executing native machine code directly on a processor, you run a piece of software that emulates another “virtual” machine.

Such a VM has several advantages: the code written for it can be executed on any (other) real processor for which you have an implementation of the VM. Also you can change and tune the VM architecture for specific needs (e.g. stack vs. register based machine, higher security, etc.). Then you can optimize the internal VM operations and instruction code as needed. Last but not least you could actually consider building the VM as a new-generation real machine in hardware.

You probably already use a VM on a daily basis: many cell phone providers chose to use JavaCard™ smart cards as their SIM. They now can use smart card hardware with different processors and hardware, and from different vendors, but still keep software updates manageable. Interpreted Java™ bytecode is also inherently more secure than native machine code.

Of course VMs also have drawbacks. One of them is speed: the VM itself is a software program that has to interpret a stream of data as VM instructions. This is slower than directly executing native machine code on the processor core, and if the VM implementation is not "correct", you might even get security problems.

Luckily for us, people will always manage to introduce bugs, this is also true for VM code. So there is a need for virtual machine (VM) debugging, which has its own advantages and problems.

...and VM Debugging

One option is adding debug support directly to the Virtual Machine (VM) itself. Then e.g. a special “Debug Interpreter” can handle the debugging requests. Android is an example for such an implementation, and this usually works well for pure Java™ application debugging. But what do you do if you can't use this, because the bug does not show when the “VM Debug Interpreter” is active? Or if you have to debug interactions between VM and Linux kernel?

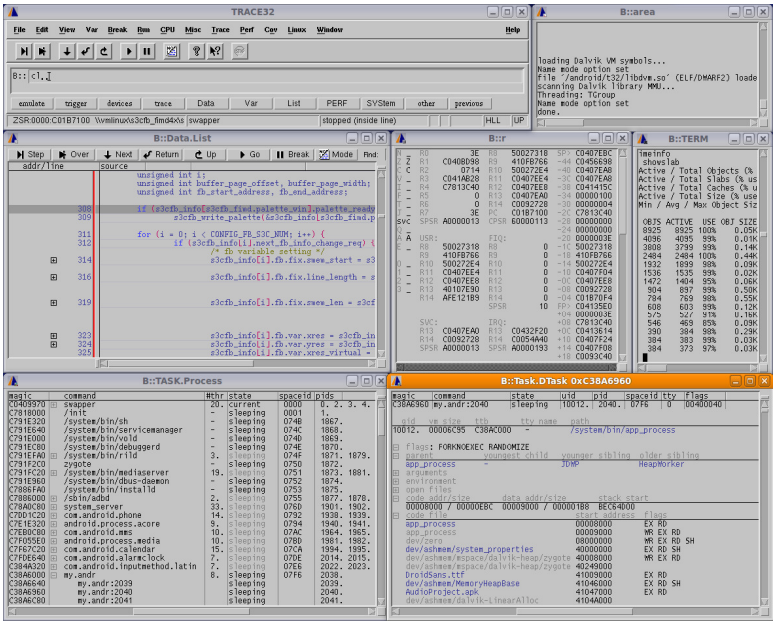
In this case, you debug in “stop-mode”. To find out which program currently runs in the Virtual Machine and which values its variables and objects have, you first need to read the memory content of the real machine. Then the debugger must find, analyze and interpret the data structures of the VM itself and of the application's object data to give a good “VM abstraction level” view of the system and its state.

One of the challenges for VM debugging is (like for operating system kernels) that any VM itself is "just a piece of software" which can and will be adapted to the needs of the final product. Any major change of the VM code and its data structures must be matched by the debug tools, or the interpreted information will be useless.

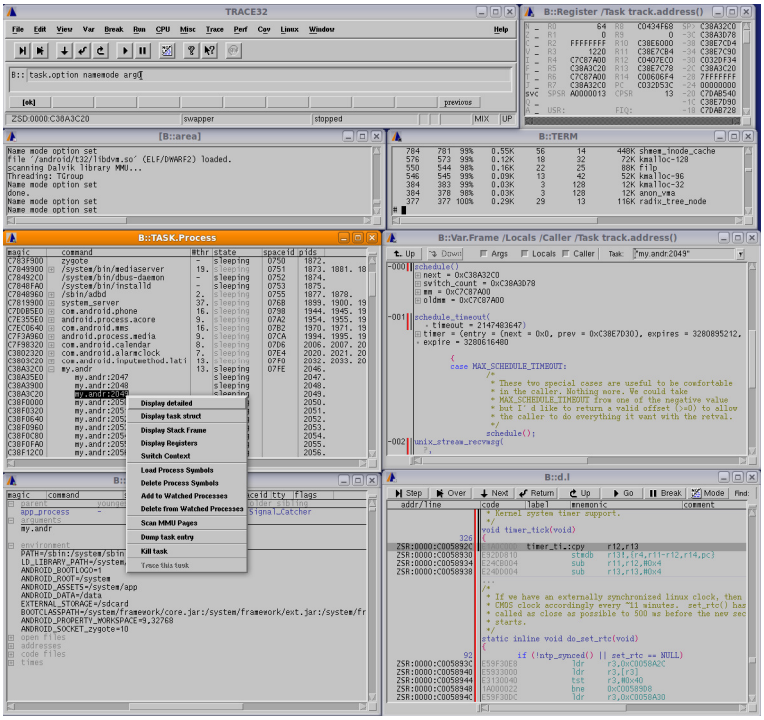
Android is an excellent example of this: the application developer writes code in Java, but the Dalvik bytecode generated from its class files could never run in a standard Java™ VM. Any standard Java™ debug tool will therefore not be able to display anything meaningful.

For the processor itself, Dalvik VM “program code” is just data. Therefore a “program trace” that works well for native code is not useful for VM tracing or profiling. For this task you need data trace capabilities (or some very clever trickery).

Providing VM debug support without any help from the target is very difficult: at Lauterbach we are currently researching unassisted and integrated Native/VM debugging. Part of the solution will be a “TRACE32 VM Awareness Extension” that can be adapted by a customer to any changes in the VM. This will make generic VM support possible.



Sample Android Debug Session



Android Debug Session with OS Awareness Menu

Boards

Our South Korean partner MDS Technology Co., LTD provided us with MEP-6410(M6R2) ARM11™ Reference Boards and with an Android operating system port.

At the heart of the MEP-6410(M6R2) is a powerful ARM1176JZF-S™ in a Samsung S3C6410 SoC. This ARM11 Reference Board features a 3.5 inch 320x480 LCD with touch screen optimized for Android, 128MB NAND FLASH and 128MB RAM, Ethernet interfaces, camera, USB, UART and plenty of other useful hardware for development. Also it includes a connector for JTAG debugging.

If you are on a more constrained budget, I suggest you check out one of the available BeagleBoard kits (e.g. EBVbeagle). Here you get an ARM™ Cortex™ A8 plus DSP within a TI™ OMAP™3530. No Ethernet or LCD are on the board itself, and depending on its revision you may have to do some Android adaptation work, but it comes for very little money.

Conclusion and Outlook

Available today are the very first steps of Virtual Machine (VM) debug support. For Dalvik VM debugging on Android, knowledge about Linux processes is essential, as each VM application runs in its own process. Our Linux Awareness interprets memory structures to give you an idea what the Linux kernel was doing at the moment the system was halted, and it can now extract and display Dalvik VM application names.

Research and development are in progress to bring “VM awareness" to Lauterbach's TRACE32 range of debugging tools. Android on ARM™ will be the first platform with VM awareness, and in a few months you can expect a ramp-up of Native/VM debug capabilities.

Published: IQ Magazine Volume 9, Number 1, 2010