デバッグの実験室より： Android のデバッグ

Hagen Patzke, Software Design Embedded Debugging, Lauterbach 社

Synopsis

Android は開発とデバッグのための優れたサポートが提供されています。高級言語（Java™）アプリケーションのデバッグは、Dalvik インタプリタのデバッグ・サポートで十分に対応されていますし、C/C++ で書かれたシステム・サービスなど、それ自体のプロセスで実行されるプラットフォームの「ネイティブな」部分も、内蔵のGNU デバッグ・サーバで比較的容易にデバッグできます。新しいプラットフォームに移植したい場合は、高機能なハードウェア支援デバッグ・ツールを使用し、低位のドライバとカーネル自体の初期デバッグを行います。すべて完璧、記事はおしまい。本当にそうでしょうか？
もし、あなたがシステム開発者で、前述の複数の領域にわたるバグを追跡しなければならないとしたら、ここからが現実の世界です。デバッグ支援を有効にしたとたん、姿を消してしまうバグを見つけなければならない場合もあるでしょう。あるいは、プロダクション・ステージやセキュア・プラットフォームを手掛けていて、ソフトウェア支援デバッグが使えないかもしれません。このような場合、状況は非常に厳しくなります。
この記事では、一般的な組み込みデバッグについて簡単に紹介し、さまざまな抽象化レベル、および Android デバッグの特殊な点について説明します。

プラットフォーム

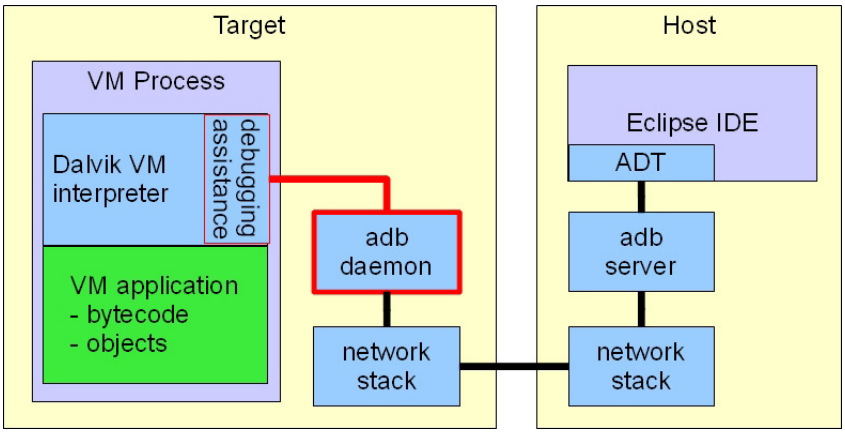
Android がこれほどの成功を収めた理由は、おそらく上から下まで完全に統合されたプラットフォームだからでしょう。きちんと動作し、使える機器を構築できるよう、何もかも綿密に定められています。さらに、Google™ がエミュレータと実際の機器の両方で「オープン・ソース」として公開したときは、ARM 実装がその機能を実世界に証明しました。
Android の中心は、専用に変更を加えたLinux 2.6 カーネルです。このカーネルの機能は多数ありますが、特に重要なのは、サービスと仮想マシン処理にマルチスレッディングを適用することです。Android の「システム」は、ネイティブ・コードと仮想マシン・プログラムの両方で構成されているからです。

Android のデバッグ機能のすべて

アプリケーション開発者は、非常に良い待遇を受けています。優れた SDK が提供され、アクティブなコミュニティがサポートを提供しているからです。また、無償のEclipse プラグインを使えば、自分の Java/Dalvik アプリケーションを開発できるだけでなく、Android デバッグ・ブリッジ（ADB）経由でJava ワイヤド・デバッグ・プロトコル（JWDP）の拡張バージョンを使用し、Dalvik 仮想マシン（VM）と機器上のデバッグ・デーモンの助けを借りることで、極めて効率的にデバッグを行うことができます。
「ネイティブ・コード」を書く、例えば、サービス・プロセスでアプリケーション・コードの重労働をするときも、同じことが言えます。GNU デバッグ・サーバをプロセスに接続し、デバッグすれば良いのです。

ストップ・モード・デバッグ: デバッガによる検査のため、プラットフォーム上のＯＳ及びアプリケーションは全て「デバッグモード」にサスペンドされます。通常、このモードでは、JTAG アクセスポートを使用する外部デバッガハードウェアを必要とします。
ラン・モード・デバッグ: デバッグされていないＯＳ及びタスクは実行状態のままになります。このモードでは、通常、ＯＳへの変更、或いは少なくともアプリケーションによる支援が必要となります。Android では、ホストとの接続を担う Android Debug Bridge Daemon(adbd)、ビルトイン Dalvik VM デバッグサポート及びネイティブ・プロセスのデバッグのための GNU デバッグサーバー(gdbserver)等の複数のデバッグ支援機能が用意されています。

このような「プラットフォーム支援の実行モード・デバッグ」だけで済むのであれば、問題はありません。コーディングとデバッグを楽しんでください！
Java のようにコンパイルされていない、しかし解釈済みの高級言語は、デバッグに仮想マシン（VM）インタプリタの助けが必要です。このことは、Android についても言えます。したがって物理的なマシンを止めれば、外付けのJava/Dalvik デバッガとデバッグしているアプリケーションとの通信リンクが途切れてしまいます。



仮想マシン (VM) アプリケーションの「支援型」デバッグ

外部のネイティブ・コードをデバッグする場合も同じです。オペレーティング・システム・カーネルを停止すれば、GNU デバッグ・サーバのプロセスも停止します。
あなたが、Android を新しいプラットフォームに移植するシステム開発者だとしましょう。あるいは、 ドライバ等の低位のコンポーネントとGUI 等の高位のコンポーネントを緊密に接続する新しい低位のサービスを構築しているとしましょう。システム・コンポーネントを1 つずつ別々にデバッグする間は、ほとんどうまくいきます。しかし、ネットワーク・スタックのようなコア・コンポーネントを変更する必要がある場合を考えてみてください。低位のドライバでハードウェア・ブレークポイントにヒットした瞬間、ホストとターゲットの間のすべてのネットワーク通信が切れ、ネットワーク・ベースのデバッグ支援は使えなくなってしまいます。フリーズした機器を接続して何が起こったかを調べたいなど、「事後」デバッグが必要な場合はどうでしょう。
この場合、「ネイティブ・デバッグ」と「Java デバッグ」の両方を同じハードウェア・デバッガに統合することで、複数の領域にまたがる見つけにくいバグを追跡、修正する必要があります。

デバッグとトレースの基本

デバッグのさまざまな「ビルディング・ブロック」を掘り下げる前に、組み込みプラットフォーム、つまり「ターゲット」上のデバッガの基本機能について、少し考えてみましょう。

デバッガとは、物理的なマシンの状態を示すスナップショットを、プログラマの意図を仮想的に表現した1 つ以上の抽象化レベルにマッピングするものです（息を吸うのを忘れないように）

しかし、はっきり言って重要なのは、プログラム・カウンタの値が0x123456 であることではなく、それがMP3 プレーヤのソース・コードの14 行目にあることです。プロセッサの3 番レジスタの値が0x1003 であることではなく、それがアプリケーションのステート・マシンの「playback_active」を意味することが極めて重要となる場合もあります。
一定の時間、プログラム実行を「トレース」し、例えば「プロファイリング」によってアプリケーションのどこを実行するのに時間がかかっているか、どんなバグが頻繁に起こっているかを調べることも有効です。このために「プログラム・トレース」、「システム・トレース」を行うハードウェアとソフトウェアがあります。いくつかのプラットフォームでは、プログラム・カウンタの値（や分岐）だけでなく、データ・メモリの変更も一定時間トレースできます。仮想マシンのプログラムは、仮想マシン（VM）インタプリタを実行する物理マシン用の単なる普通のデータなので、VM を持っている場合は、データ・トレース機能が非常に便利です。
一言で言うと、「デバッガ」のコア・タスクとは、生のメモリとレジスタ・データを、意味を成すものにマッピングすることです。一定の時間にわたって（選択した）システム状態の変化を記録することは「トレース」と呼ばれ、性能のボトルネックを特定したり、断続的なバグを見つけたりするのに役立ちます。

ターゲットとホスト

PC では、アプリケーションや開発フレームワークを実行しているマシンでソフトウェア・デバッガも実行し、ほとんどのアプリケーション・デバッグを行います。ほとんどのオペレーティング・システムとPC アーキテクチャ自体が、このタイプの「ソフトウェア・ベースの」デバッグを積極的にサポートしています。

組み込みプラットフォームは違います。ほとんどのターゲット機器は、処理性能、メモリ、使用可能なインタフェースが限られています。したがって、開発やデバッグは通常、十分な性能を持った外付けの「ホスト」マシンで行われます。

ここで、あなたの組み込み機器のインフラ（ブート・ローダ、ドライバ、オペレーティング・システム）について考えてみましょう。何であれ、ソフトウェアをビルドするには、「ターゲット」機器のアーキテクチャ向けに「クロスコンパイル」する必要があります。PC の「ホスト」とターゲット・アーキテクチャが同じでないこともあります。そして、そのソフトウェアを組み込み機器にロードします。

これでやっとソフトウェアを起動し、デバッグすることができます。他が万事OK であれば、デバッグの対象はアプリケーションです。アプリケーション・コードの実行を開始する前に何か問題が生じれば、インフラを先にデバッグする必要があるでしょう。「プラットフォーム自体の上でデバッグする」という従来のPC の手法は、ここでは使えません。

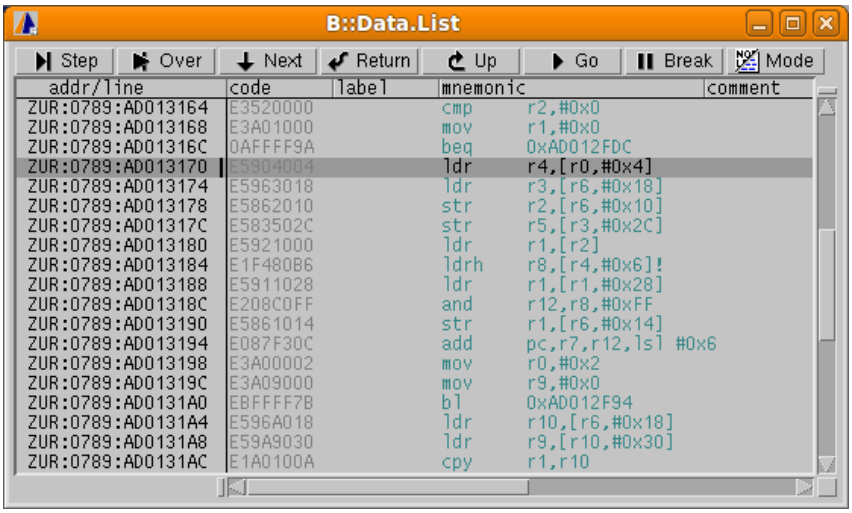
組み込み分野で、ソフトウェアをロードし、デバッグする標準的な方法は、ターゲットをJTAG（IEEE1149.1）などのハードウェア・インタフェースを介して、外付けのデバッグ・ハードウェアに接続することです。デバッグ・ハードウェアは、「ホスト」上で動作するグラフィカル・ユーザ・インタフェース（GUI）で制御します。GUI 経由、またはスクリプト言語を使ってメモリ・コンテンツを変更し、ソフトウェアをターゲット上に「ダウンロード」した後、システムとアプリケーションを実行、デバッグします。

ネイティブ・コードのデバッグ

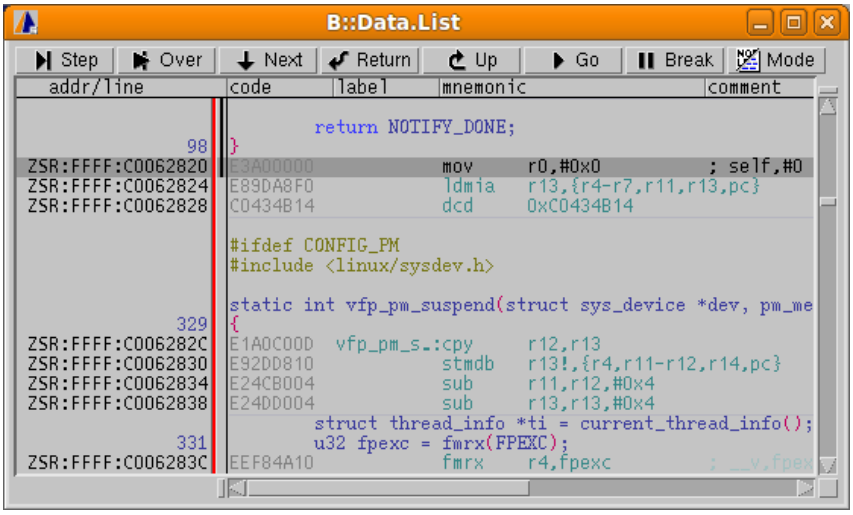
マシン・ステートの抽象化レベルへのマッピングを見てみましょう。最低レベルは、オンチップ信号と電圧です。1 つ上のレベルは、レジスタとメモリのビットです。もう1 つ上は、数（10 進数または16 進数で表現）です。次のレベルに、やっとアセンブリ言語命令が出てきます。ここで、支援なしのデバッグは終わりです。

幸運なことに、現代的なコンパイラは「デバッグ情報」を出力し、アセンブリのロケーションを高級言語（C/C++ など）のソース・コードの行にマッピングします。他の「デバッグ・レコード」は、論理データ型をメモリ内のデータ・レイアウトにマッピングします。この付加的なデバッグ情報はプログラム・ファイルをかなり増大させますが、これなしでバグを探すのは「暗闇を手探りする」のと同じです。やはり、カレント・プログラム・カウンタを高級言語のソース・ファイル内の行にマッピングする必要があるでしょう。そうすれば初めて、物理的なプロセッサ・コアおよび任意の抽象化レベル上で実行される「ネイティブ・マシン」コード内で何が起こっているかを見ることができます。

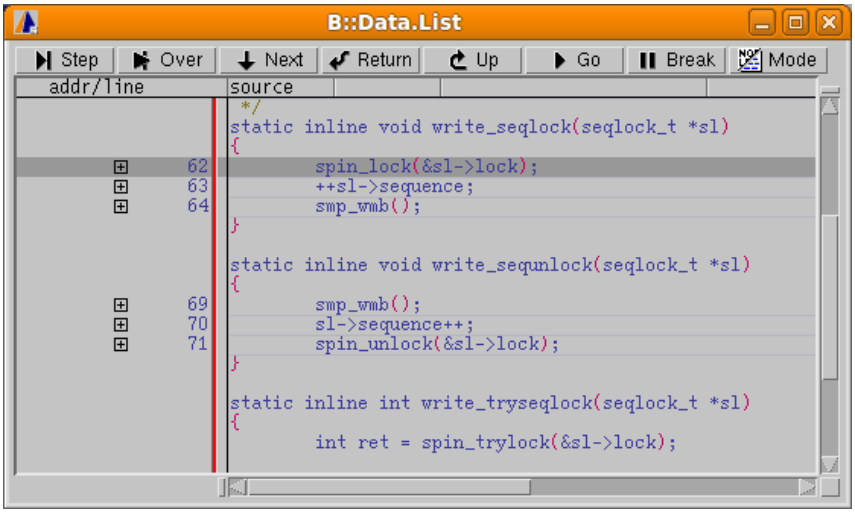
ネイティブ・コード・デバッガでは、何ができるのでしょうか？ 一言で言えば、使用可能なプロセッサ・コア上にあるプログラム実行の選択的な開始と停止、メモリ、コア、ペリフェラル・デバイス・レジスタの調査と操作、ブレークポイント（プログラムを所定のロケーションで停止）のセットと削除です。



数字（コード）とアセンブリ言語（ニモニック）の抽象化レベル



混合アセンブリ言語と高級（C）言語の抽象化レベル



高級（C）言語の抽象化レベル

カーネルのデバッグと OS 認識

オペレーティング・システム・カーネルの登場です。現代のプラットフォームは、マルチタスキングとマルチスレッディングをサポートしています。つまり、2 つ以上のプログラムやプロセスを同時に実行できます。これは、シングル・プロセッサ・コアでも同じです。オペレーティング・システムは、各プロセスに「タイム・スライス」を与え、次のプロセスに移ります。

アプリケーションが1 つだけなら簡単に見つけてデバッグできますが、今度は複数のアプリケーションが「同時に」動作しています。同じアプリケーションの複数のインスタンスが同時にアクティブで、そのうち1 つだけをデバッグしたいこともあるでしょう。

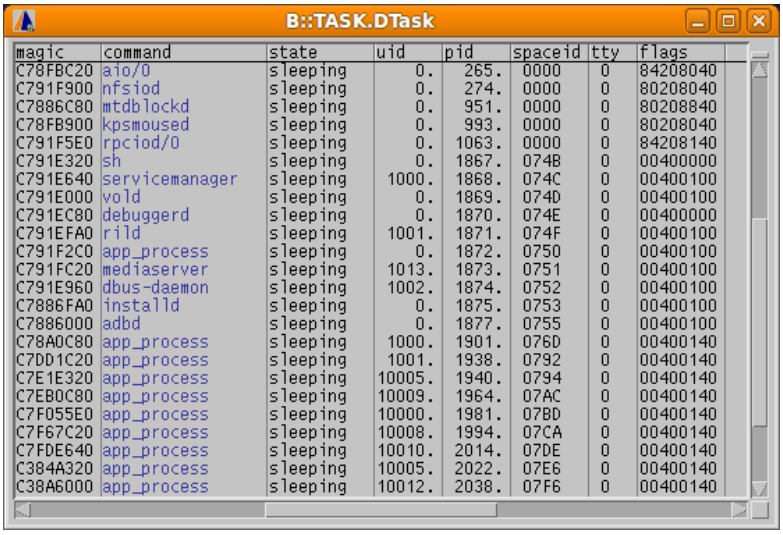
前述のように、純粋な「ネイティブ・コード・デバッグ」では、オペレーティング・システムのブート・コード、デバイス・ドライバ、タスク・スケジューラのような低位の構造体の問題を、容易に見つけることができます。

しかし、カレント・アプリケーション・プロセスがプログラム・メモリのどの部分を実行しているのか、インスタンス変数がどこにあるか（複数あるかもしれませんね）を知りたい場合は、デバッガがオペレーティング・システムを知っている必要があります。すなわち「OS 認識機能」を持つデバッガが必要なのです。

実は、この機能は、デバッガに簡単に付与できるものではありません。プロセッサ・コアや高級言語コンパイラ（C/C++ など）と異なり、オペレーティング・システム（OS）自体が、ほとんどのPC に搭載されているような固定された「既製の」ものではないからです。組み込み分野のOS は、競争力のある新製品を作るために積極的に変更し、調整する必要があるものです。

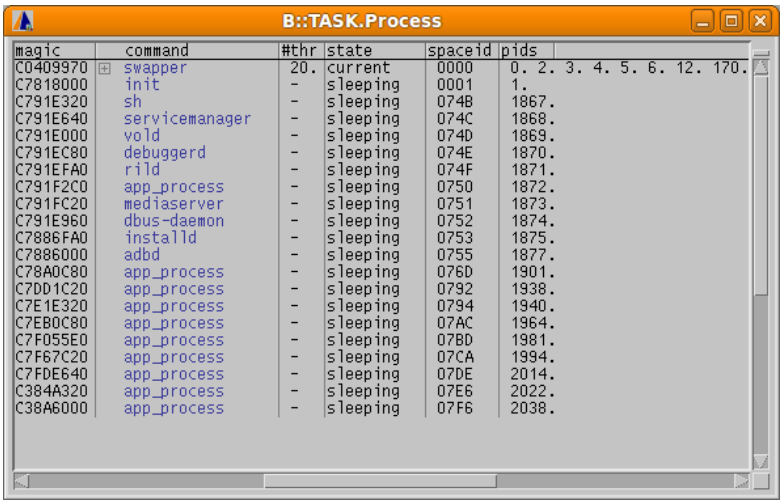
変更可能なOS は、デバッグにとって「動く標的」です。したがって、構成や調整によって、デバッガに自分の使っているOS バリアントに対する「OS 認識機能」を持たせる必要があります。

このことは、組み込みリアルタイム・オペレーティング・システム（RTOS）が普及すると明らかとなり、ローターバッハ社はそのニーズに応えて、Extension Development Kit（ EDK） を備えた「TRACE32Extension」メカニズムを実装しました。TRACE32 Extension を使えば、既存のOS 認識機能（Linux 用など）に調整を加えたり、必要に応じてユーザが自分で書いたりすることができます。

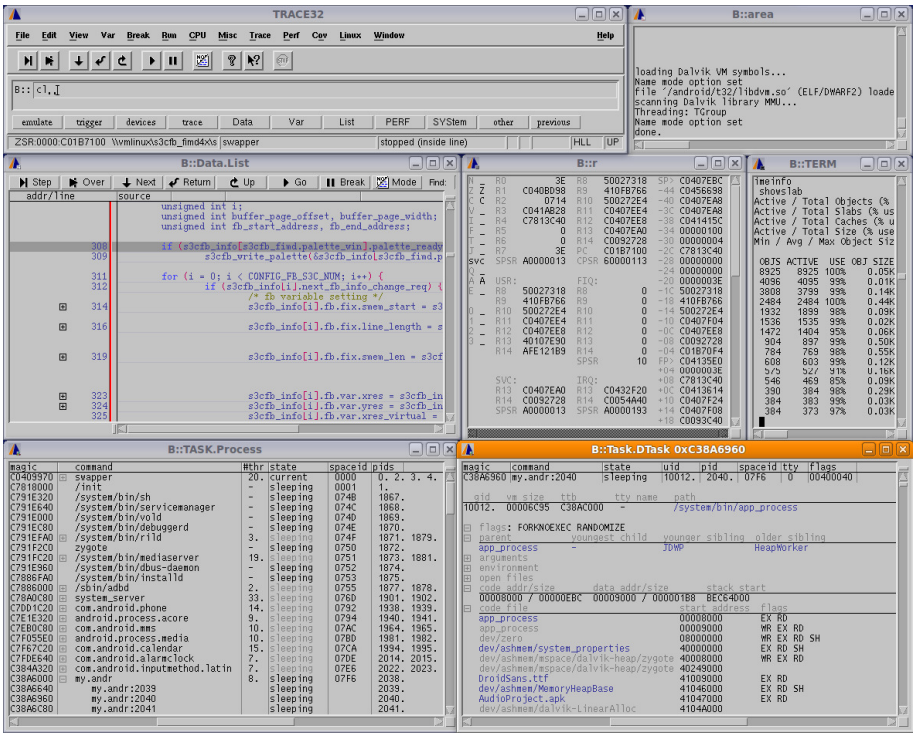


magic	command	state	uid	pid	spaceid	tty	flags
C78FBC20	aio/0	sleeping	0.	265.	0000	0	84208040
C791F900	nfsiod	sleeping	0.	274.	0000	0	80208040
C7886C80	mtdblockd	sleeping	0.	951.	0000	0	80208840
C78FB900	kpsmoused	sleeping	0.	993.	0000	0	80208040
C791F5E0	rpciod/0	sleeping	0.	1063.	0000	0	84208140
C791E320	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 カーネルのタスク



Linux のプロセス・ディスプレイ



VM アプリケーション名付きのプロセス・ディスプレイ

仮想マシン...

プロセッサの性能は、どんどん高まりました。これにより、ハードウェアすら抽象化することが可能となり、かつては大きなメインフレーム機のものだった仮想マシン（VM）が組み込み分野にも導入されました。

ネイティブのマシン・コードをプロセッサ上で直接実行するのではなく、仮想マシン（VM）でコードを実行する場合、別の「仮想」マシンをエミュレートするソフトウェアを実行することになります。

このような VM には、いくつかの利点があります。1 つは、VM 向けに書いたコードを、VM の実装を持っている他のどんな実プロセッサ上でも実行できることです。もう1 つは、VM アーキテクチャを特定のニーズ（スタック・ベース、レジスタ・ベースのマシン、セキュリティ向上など）に合わせて変更できることです。内部のVM 演算や命令コードも必要に応じて最適化できます。言い遅れましたが、実は、新世代の実マシンとしてハードウェアにVM を構築することすら可能です。

皆さんはおそらく、既に日常的にVM を使っています。例えば多くの携帯電話プロバイダは、SIM として JavaCard™ スマートカードを選択しています。さまざまなベンダのさまざまなプロセッサやハードウェアにスマートカード・ハードウェアを使用し、なおかつソフトウェア・アップデートを管理しやすいからです。また、解釈済みのJavaバイトコードは、ネイティブのマシン・コードより本質的にセキュアです。

もちろん VM にも欠点はあります。1 つは速度です。VM 自体がソフトウェア・プログラムとして、データ・ストリームをVM 命令として解釈する必要があります。これは、ネイティブのマシン・コードをプロセッサ・コア上で直接実行するよりも低速で、VM の実装が「適切」でなければ、セキュリティ上の問題が生じる恐れもあります。

我が社にとって幸運なことに、人間は必ずバグを入れてしまうものであり、それはVM コードについても言えます。ですから、仮想マシン（VM）のデバッグに対するニーズがあり、デバッグ上の利点と問題点があります。

...そして仮想マシンのデバッグ

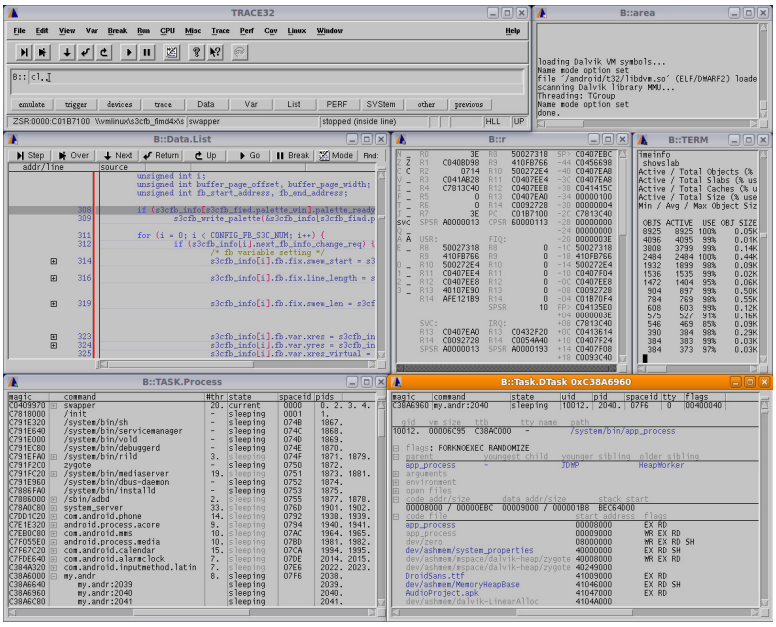
1 つの方法は、仮想マシン（VM）自体に直接デバッグ・サポートを追加することです。そうすれば、専用の「デバッグ・インタプリタ」などがデバッグ要求を処理できます。Android は、そのような実装の例であり、通常、純粋なJava™ アプリケーションのデバッグについては有効です。しかし、「VM デバッグ・インタプリタ」がアクティブなときにはバグが姿を見せず、これを使えない場合はどうしましょう？あるいは、VM とLinuxカーネルの相互作用をデバッグしなければならない場合は、どうでしょう？

この場合は、「ストップモード」でデバッグします。仮想マシン内で、現在どのプログラムが実行されているかを知り、変数とオブジェクトが何の値を持っているかを知るには、まず、実マシンのメモリ・コンテンツを読み出す必要があります。次にデバッガが、VM 自体およびアプリケーションのオブジェクト・データのデータ構造を発見、分析、解釈し、システムとその状態の良好な「VM 抽象化レベル」表示を提供せねばなりません。

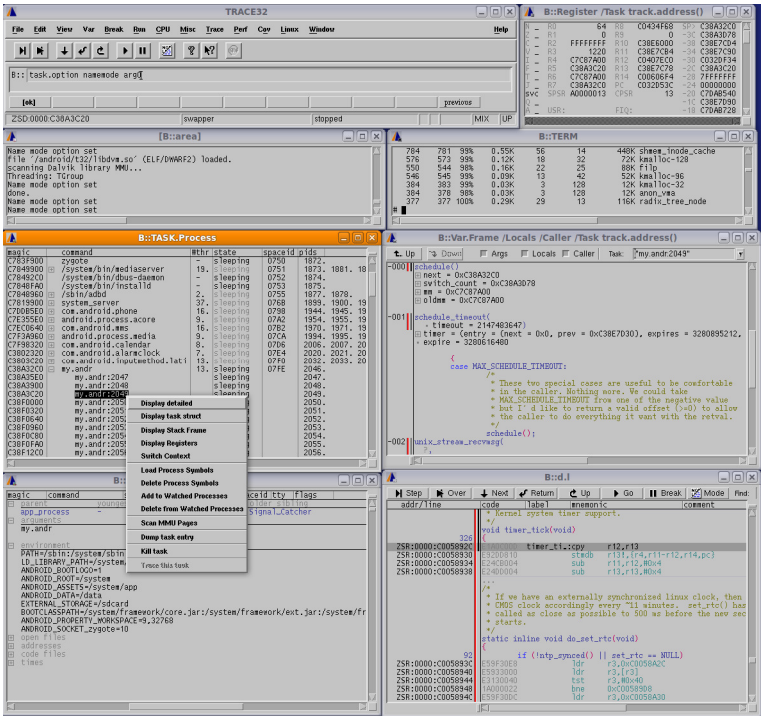
VM デバッグの問題の1 つは、（オペレーティング・システム・カーネルと同様）どんなVM も「単なる1 つのソフトウェア」であり、最終製品のニーズに合わせて変更されるということです。VM コードとそのデータ構造に大きな変更があれば、デバッグ・ツールもそれに対応させねばなりません。さもなければ、解釈した情報は使えません。

Android は、この好例です。アプリケーション開発者はJava でコードを書きますが、そのクラス・ファイルから生成されたDalvik バイトコードが、標準的なJava™ VM で動作することは決してありません。したがって標準的なJava デバッグ・ツールは、何も有意義なものを表示できないことになります。

プロセッサ自体について言えば、Dalvik VM の「プログラム・コード」は、単なるデータです。したがって、ネイティブ・コードに有効な「プログラム・トレース」はVM のトレースやプロファイリングに使えません。このタスクには、データ・トレース機能（または、非常に巧妙な工夫）が必要です。ターゲットから何の支援も得ずにVM デバッグ・サポートを提供することは、非常に困難です。ローターバッハでは現在、支援なしの内蔵ネイティブ/VM デバッグについて研究中です。ソリューションの一部は、VM の変更に応じてメーカが調整を加えることのできる「TRACE32 VM Awareness Extension」となる予定です。これが、汎用的なVM サポートを可能にするでしょう。



Android デバッグ・ セッションのサンプル



OS 認識メニューのある Android デバッグ・セッション

ボード

当社のパートナーである韓国のMDS Technology 社は、MEP-6410（M6R2）ARM11™ リファレンス・ボードをAndroid オペレーティング・システム・ポート付きで提供してくれました。

MEP-6410（ M6R2 ） の中心は、Samsung S3C6410 SoCに組み込まれた高性能なARM1176JZFS™ です。このARM11リファレンス・ボードは、3.5 インチ（240x320）のLCD を装備し、Android に最適化したタッチ・スクリーン、128MB のNAND FLASH、128MB のRAM、イーサネット・インタフェース、カメラ、USB、UART、他多数の開発に有用なハードウェアが付属しています。JTAG デバッグ用の端子も含まれます。

予算がさらに限られている場合は、BeagleBoardキット（EBVbeagle など）の検討をお勧めします。これには、ARM®Cortex™-A8 とDSP を含むTIOMAP 3530 が使用されています。デバッグ端子なし、イーサネットやLCD はボード自体に搭載、リビジョンによってはAndroid に調整を加える必要があるかもしれませんが、コストは非常に少なくて済みます。

結論と展望

現在、提供されているのは、非常に初歩的な仮想マシン（VM）デバッグ・サポートです。Android 上でDalvik VM をデバッグするには、Linux プロセスに関する知識が不可欠です。なぜなら各VM アプリケーションが、それ自体のプロセス内で動作するからです。当社のLinux Awareness は、メモリ構造を解釈し、システム・ホールト時にLinux カーネルが何をしていたかのアイデアを与え、現在ではDalvik VM のアプリケーション名を抽出、表示することもできます。
ローターバッハでは、TRACE32 デバッグ・ツール製品群に「VM 認識機能」を持たせるよう、現在、研究開発が進んでいます。Android on ARM は、VM 認識機能を持つ初のプラットフォームとなるでしょう。数カ月内に、ネイティブ/VM デバッグ機能の量産も期待されています。

Published: IQ Magazine Volume 9, Number 1, 2010