NEWS 2011

LAUTERBACH DEVELOPMENT TOOLS

DEBUGGER, REAL-TIME TRACE, LOGIC ANALYZER

Always a Few Steps Ahead

Lauterbach has been developing tools for the embedded industry for over 30 years advocating this slogan. For most new debug technologies Lauterbach is the world leader and trend setter.

This has allowed us to gain the recognition of all the big semiconductor manufacturers. For many years, those involved in developing and implementing new technologies have favored collaboration with Lauterbach. This collaboration has inspired many ground breaking ideas to be transformed into advanced products.

In addition, Lauterbach is very customer focused. The desires and suggestions of our TRACE32 users provide a valuable contribution to our product development. In many cases, suggestions are put into practice immediately and are then included in the next released version of our debugger.

From this vantage point what trends does Lauterbach currently see? What technologies are soon to emerge in the market?

Android Debugging

Android debugging is certainly an important topic. Applications for mobile phones are increasingly being written architecture-independent for virtual machines (VM). Google's Android and its Dalvik VM are quite prevalent. Complex errors that will only appear with the interplay of application, virtual machine, operating system and the underlying hardware have to be debugged. To do this it is necessary to have transparency through all of the software layers, from the Java application down to the Linux hardware drivers.

At the request of some mobile phone manufacturers, Lauterbach started developing an API for *VM Debugging Awareness* in the middle of 2010. Android is used here as a reference platform. The aim is to provide an open interface that allows providers of open-source and closed-source VMs to adapt their products for debugging with TRACE32. For information on *VM Debug-* »

CONTENTS

ging Awareness and the current state of development, see the article “API for VM Debugging Awareness” on page 6.

Energy Profiling

Energy measurement for embedded systems has come more into focus with the increasing emphasis on global warming and “green” electronics systems. Every technical journal now contains many articles on battery-driven equipment and low-power microcontrollers. Prizes for innovation are increasingly being awarded for new technologies in this field.

However, in the mobile phone market standby and operating times have always been an important topic. For years, extensive energy reduction measures have been implemented in this area. But these measures only make sense if the software that controls an embedded system consistently uses all the energy-saving features of the hardware.

Since the beginning of 2006, Lauterbach tools have supported measuring arrangements that allow the simple comparison and analysis of the interplay between software and power consumption in an embedded system. This technology has also been available for the TRACE32 CombiProbe since mid 2010. For more information on “Energy Profiling with the CombiProbe”, see page 11.

Multicore Debugging

Although multicore chips have been used in embedded systems for ten years and Lauterbach has had debuggers for them since 2001, this is still a highly dynamic topic. The current calls for greater visibility into the internal system operation are ensuring the integration of new trace cells within the debug infrastructure of the chips.

Originally, trace information was only generated for the individual cores, whereas today there are many other trace sources:

a) Trace sources that make transfers on chip-internal buses visible:

- ARM CoreSight with the AMBA AHB Trace Macrocell (HTM)
- MCDS with the System Peripheral Bus (SPB) and the Local Memory Bus (LMB) for the TriCore from Infineon
- RAM Trace Port for chips from Texas Instruments
- DMA and FlexRay trace for NEXUS Power Architecture

b) Trace sources that generate trace information for chip-internal IP (Intellectual Property), such as special interrupt traces.

c) Trace sources that permit the output of software-generated trace information, such as:

- Instrumentation Trace Macrocell (ITM) for ARM CoreSight
- System Trace Macrocell (STM) for ARM CoreSight

The continuous development of the TRACE32 debugger ensures it is aware of these new trace sources and can provide easy configuration and a comprehensive analysis of the information provided.

Serial Trace Ports

Due to the extra trace data provided by this visibility into the internal chip processes, complex multicore chips and high-performance processors require more and more bandwidth and thus even faster trace ports.

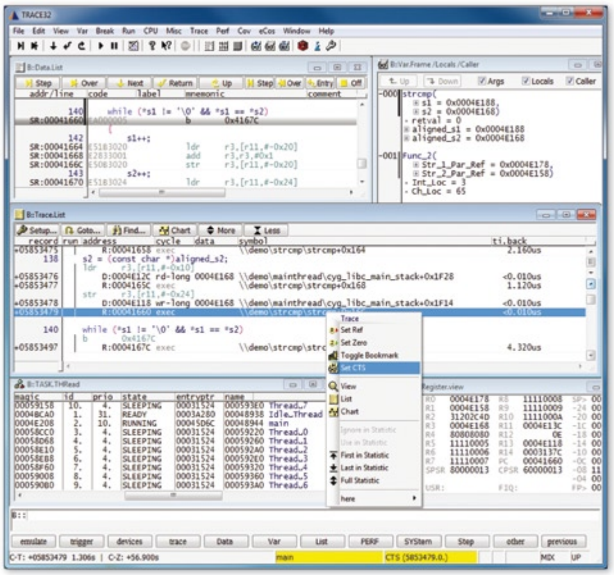
In response chip manufacturers have developed serial trace ports as an important innovation in the last few years. Hard-disk manufacturers, who have been using serial interfaces for high-speed data exchange with the PC for years, used this technology for the first time in 2008 to export trace information via ARM’s High Speed Serial Trace Port (HSSTP). At the same time, Lauterbach launched trace tools for this technology.

In the meantime, there are other processor families with serial trace interfaces. For current developments in this area, see the article “Serial Trace Port Usage Growing” on page 9.

Bigger Trace Memory

Fast trace interfaces with their high data rates inevitably require more trace memory. Without this, it is impossible to capture a sufficiently large program section for troubleshooting and the analysis of the time behaviour for an embedded system.

»


TRACE32
File Edit View Var Break Run CPU Misc Trace Perf Cov eCos Window Help
B::Data.List
Step Over Next Return Up Step Over Entry Off
B::Var.Frame /Locals /Caller
Up Down Args Locals Caller
B::Trace.List
Setup... Goto... Find... Chart More Less
Trace
Set Ref
Set Zero
Toggle Bookmark
Set CTS
View
List
Chart
Ignore in Statistic
Use in Statistic
First in Statistic
Last in Statistic
Full Statistic
here
B::TASK.THRead
Register.view
emulate trigger devices trace Data Var List PERF SYStem Step other previous

```
   addr/line  code       label   mnemonic          comment
         140      while (*s1 != '\0' && *s1 == *s2)
 SR:00041660 EA000005            b      0x4167C
                  {
         142          s1++;
 SR:00041664 E5183020            ldr    r3,[r11,#-0x20]
 SR:00041668 E2833001            add    r3,r3,#0x1
 SR:0004166C E5083020            str    r3,[r11,#-0x20]
         143          s2++;
 SR:00041670 E5183024            ldr    r3,[r11,#-0x24]
```

```
-000 strcmp(
        s1 = 0x0004E188,
        s2 = 0x0004E168)
      - retval = 0
      aligned_s1 = 0x0004E188
      aligned_s2 = 0x0004E168

-001 Func_2(
        Str_1_Par_Ref = 0x0004E178,
        Str_2_Par_Ref = 0x0004E158)
      - Int_Loc = 3
      - Ch_Loc = 65
```

```
    record run address      cycle   data     symbol                                       ti.back
 +05853475  |    R:00041658 exec             \\demo\strcmp\strcmp+0x164                    2.160us
       138    s2 = (const char *)aligned_s2;
            | ldr     r3,[r11,#-0x10]
 +05853476       D:0004E12C rd-long 0004E168 \\demo\mainthread\cyg_libc_main_stack+0x1F28  <0.010us
 +05853477       R:0004165C exec             \\demo\strcmp\strcmp+0x168                    1.120us
              str     r3,[r11,#-0x24]
 +05853478       D:0004E118 wr-long 0004E168 \\demo\mainthread\cyg_libc_main_stack+0x1F14  <0.010us
 +05853479       R:00041660 exec             \\demo\strcmp\strc                            <0.010us

       140    while (*s1 != '\0' && *s1 == *s2)
            | b       0x4167C
 +05853497       R:0004167C exec             \\demo\strcmp\strc                            4.320us
```

```
magic     id   prio  state     entryptr  name
00059158  10.   4.   SLEEPING  00031524  000593E0 Thread_7
0004BCA0   1.  31.   READY     0003A280  00048938 Idle_Thread
0004E208   2.  10.   RUNNING   00045D6C  00048944 main
00058CC0   3.   4.   SLEEPING  00031524  00059220 Thread_0
00058D68   4.   4.   SLEEPING  00031524  00059260 Thread_1
00058E10   5.   4.   SLEEPING  00031524  000592A0 Thread_2
00058EB8   6.   4.   SLEEPING  00031524  000592E0 Thread_3
00058F60   7.   4.   SLEEPING  00031524  00059320 Thread_4
00059008   8.   4.   SLEEPING  00031524  00059360 Thread_5
000590B0   9.   4.   SLEEPING  00031524  000593A0 Thread_6
```

```
R0   0004E178  R8   11110008  SP> 00
R1   0004E158  R9   11110009  -24 00
R2   31202C4D  R10  1111000A  -20 00
R3   0004E168  R11  0004E13C  -1C 00
R4   80808080  R12        0E  -18 00
R5   11110005  R13  0004E118  -14 00
R6   11110006  R14  0003137C  -10 00
R7   11110007  PC   00041660  -0C 00
SPSR 80000013  CPSR 60000013  -08 11
                              -04 00
USR:           FIQ:           FP> 00
```

```
B::
```

```
C-T: +05853479 1.306s | C-Z: +56.900s    main    CTS (5853479.0.)    MIX    UP
```

Fig. 1: Even demanding 4-GB trace memory analysis for Trace-based Debugging can be performed quickly.

However, providing more and more trace memory only makes sense if the necessary infrastructure for fast processing of the trace information is available. This applies particularly to demanding trace analysis functions such as *Trace-based Debugging* (see Fig. 1). The increasing capacity of SDRAM chips, fast PCs, and GB Ethernet interfaces enabled Lauterbach to launch the trace tool PowerTrace II with 4 GB memory in 2007.

In mid-2008, Lauterbach started developing a new method of trace recording and analysis, *Real-time Streaming*. This development was driven by customers' demands for long-term code coverage analysis, for comprehensive system runtime analyses and for a much longer trace recording time to locate infrequent errors.

The new feature of *Real-time Streaming* is that the trace data is transferred to the host while it is being recorded. The trace information is then analyzed on the host as soon as it is received. Optionally, the trace information can also be saved to the hard-disk while it is being analyzed.

Real-time Streaming works only if all processing steps for the trace data run at optimal speed. This applies to transfer and analysis, as well as the systematic search for trace information in a file saved to the hard-disk.

Conventional tracing also profits from many of the new speed optimizations. For example, there are plans to implement the trace-data compression (developed for *Real-time Streaming*) also for conventional tracing. For details on the trace-data compression, see page 10.

Trace-Based Debugging

Trace-based Debugging (also known as CTS = Context Tracking System) allows re-debuggging of a traced program section. TRACE32 makes this possible as it can reconstruct the state of the target system for each individual trace record in its PowerView GUI. This reconstruction includes the register and memory contents, variable states, source and task listing, stack-frame, and much more.

After choosing a starting point for Trace-based Debugging, all of the debug commands can be used. These commands are executed by TRACE32 based upon the reconstruction from the trace recording. Many users of Trace-based Debugging appreciate the fact that they can also step backwards or return to the function start.

Trace-based Debugging also provides a series of other useful functions:

- Trace display in high-level language with all local variables
- Runtime analyses and function call tree
- Reconstruction of the trace gaps that can occur if more trace data is being generated than can be exported via the trace port

www.lauterbach.com/cts.html

Outlook

In addition to the current trends, there are a large number of new developments in debug technology. When you browse through our 2011 newsletter, you will probably discover one or two of these that might help with your project. We will be demonstrating several of them live at the upcoming ESC Silicon Valley, May 2-5, in San Jose, and also at many other shows in the US throughout the year.

Visit us: Booth # 1922

New Supported Processors

New derivatives

Actel	**LA-7844 (Cortex-M)** • A2F060, A2F200, A2F500
AppliedMicro	**LA-7723 (PPC400)** • APM80186, APM821x1 • APM86290 **LA-7752 (PPC44x)** • PPC460SX
ARM	**LA-7843 (Cortex-A/R)** • Cortex-A15 • Cortex-A15 MPCore **LA-7844 (Cortex-M)** • Cortex-M4 • SC000, SC300
Atmel	**LA-7844 (Cortex-M)** • AT91SAM3S, AT91SAM3N **LA-3779 (AVR32)** • AT32UC3A/B/C/D/L
Broadcom	**LA-7760 (MIPS32)** • BCM3549/35230/4748 • BCM5354/5358/5331X • BCM6816/6328/6369 • BCM7407/7413/7420
Cavium	**LA-7761 (MIPS64)** • CN63XX
Ceva	**LA-3711 (CEVA-X)** • CEVA-X1643, CEVA-XC
Cortus	**LA-3778 (APS)** • APS3/B/BS/S
Cypress	**LA-7844 (Cortex-M)** • PSoC5
Faraday	**LA-7742 (ARM9)** • FA726TE
Freescale	**LA-7736 (MCS12X)** • MCS9S12GC/GN/Q **LA-7732 (ColdFire)** • MCF5301x, MCF5441x **LA-7845 (StarCore)** • MSC8156 **LA-7742 (ARM9)** • i.MX28 **LA-7843 (Cortex-A/R)** • i.MX53 **LA-7844 (Cortex-M)** • Kinetis
Freescale (Cont.)	**LA-7753 (MPC55xx/56xx)** • MPC5602D/P • MPC564XA/B/C/S • MPC567XF/R **LA-7729 (PowerQUICC II)** • MPC830X **LA-7764 (PowerQUICC III)** • P10xx, P20xx, P40xx • P3041 (2H/2011) • P5010, P5020 (2H/2011)
Fujitsu	**LA-7844 (Cortex-M)** • FM3
Infineon	**LA-7756 (TriCore)** • TC1182, TC1184 • TC1782, TC1782ED • TC1784, TC1784ED • TC1791, TC1791ED • TC1793, TC1793ED • TC1798, TC1798ED **LA-7759 (XC2000/C166S V2)** • XC22xxH/I/L/U • XC23xxC/D/E/S • XC27x2/x3/x7/x8 • XE16xFH/FU/FL
Intel®	**LA-3776 (Atom™/x86)** • E6xx, Z6xx, N470 • Core i3/i5/i7, Core2 Duo
Lantiq	**LA-7760 (MIPS32)** • XWAY xRX200
LSI	**LA-7765 (ARM11)** • StarPro2612, StarPro2716 **LA-7845 (StarCore)** • StarPro2612, StarPro2716
Marvell	**LA-7742 (ARM9)** • 88F6282, 88F6283, 88F6321 • 88F6322, 88F6323 **LA-7765 (ARM11)** • 88AP510-V6 **LA-7843 (Cortex-A/R)** • 88AP510-V7
MIPS	**LA-7760 (MIPS32)** • MIPS M14K, MIPS M14KC
Netlogic	**LA-7761 (MIPS64)** • XLR, XLS
NXP	**LA-7844 (Cortex-M)** • LPC11xx • EM773

New derivatives

Ralink	**LA-7760 (MIPS32)** • RT3052, RT3662
Renesas	**LA-3777 (78K0R / RL78)** • 78K0R / Hx3 / Lx3 / Ix3 • 78F804x, 78F805x • RL78 / G12, RL78 / G13 **LA-3786 (RX)** • RX610 / 6108 / 621 / 62N / 630
STMicro-electronics	**LA-7753 (MPC55xx/56xx)** • SPC560D/P, SPC56APxx • SPC564Axx, SPC56ELxx **LA-7844 (Cortex-M)** • STM32F100, STM32L15x
ST-Ericsson	**LA-7843 (Cortex-A/R)** • DB5500, DB8500
Tensilica	**LA-3760 (Xtensa)** • LX3
Texas Instruments	**LA-3713 (MSP430)** • MSP430xG461x • MSP430x20x1 / x2 / x3 **LA-7742 (ARM9)** • AM1707 / 1808 / 1810 **LA-7843 (Cortex-A/R)** • OMAP36xx **LA-7838 (TMS320C6x00)** • OMAP36xx
Toshiba	**LA-7742 (ARM9)** • TMPA900, TMPA910 **LA-7844 (Cortex-M)** • TMPM330, TMPM370
Trident	**LA-7760 (MIPS32)** • HiDTV PRO-QX
Wintegra	**LA-7760 (MIPS32)** • WinPath3, WinPath3-SL
Zoran	**LA-7760 (MIPS32)** • COACH 12

Tracing for Virtual Targets in Fast Models

Lauterbach has supported tracing for ARM Fast Models since November 2010.

To avoid having to wait for the first hardware prototypes before starting software development, software models of the hardware are often used. With Fast Models, ARM offers its customers a software package for programming models for ARM-based designs.

Since 2008, Lauterbach has supported the debugging of Fast Models over the CADI interface. It has now introduced support for the Model Trace Interface, which was introduced for Fast Models with Version 5.1. To prepare the trace information appropriately and buffer it in the virtual target, debugger manufacturers can load a separate trace plug-in. Fig. 2 shows an overview of the interplay of TRACE32 and Fast Models.

For detailed information on debugging virtual targets, see:

www.lauterbach.com/frontend.html

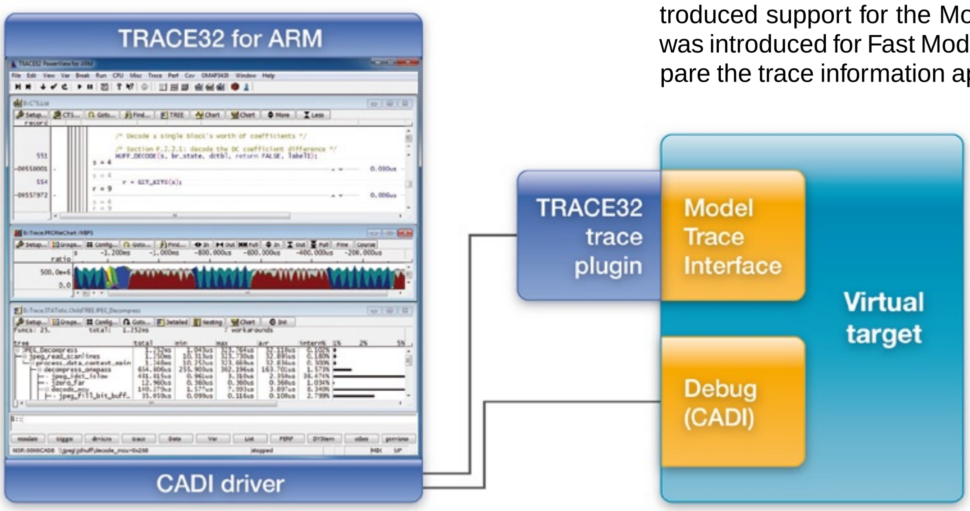


Fig. 2: TRACE32 supports the debugging and tracing of virtual targets.

API for VM Debugging Awareness

Since 2006, Lauterbach has supported the debugging of Java applications for the Java Virtual Machines J2ME CLDC, J2ME CDC and Kaffe. Since virtual machines are increasing in popularity, the number of providers is growing. Nowadays not all of these virtual machines are open-source. To enable VM providers and their customers to adapt debugging flexibly for their VM, Lauterbach has been working on a solution since mid-2010.

The Android Dalvik Virtual Machine implemented for ARM cores is used as a reference for the development of a VM API for stop-mode debugging.

Two Debug Worlds

For developers, Android is an open-source software stack consisting of the following components (see Fig. 3):

- A Linux kernel with its hardware drivers.
- Android Runtime with Dalvik Virtual Machine and a series of libraries: classic Java core libraries, Android-specific libraries, and libraries written in C/C++.
- Applications programmed in Java and their supporting Application Framework.

Software for Android is written in various languages:

- The Linux kernel, some libraries, and the Dalvik Virtual Machine are coded in C, C++, or Assembler.

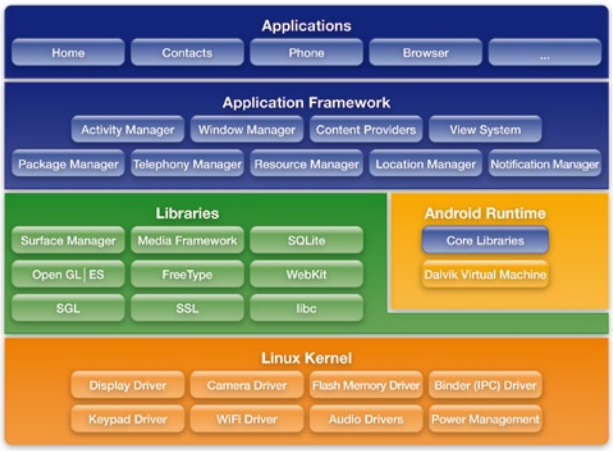


Fig. 3: The open-source Android software stack.

- VM applications and their supporting Application Framework are programmed in Java.

Each block of code is tested in its own separate debug world.

Debugging C/C++ and Assembler Code

The Android part coded in C/C++ and Assembler can be debugged on the target hardware over the JTAG interface in stop-mode. In stop-mode debugging, the TRACE32 debugger communicates directly with the processor of the Android hardware platform (see Fig. 4).

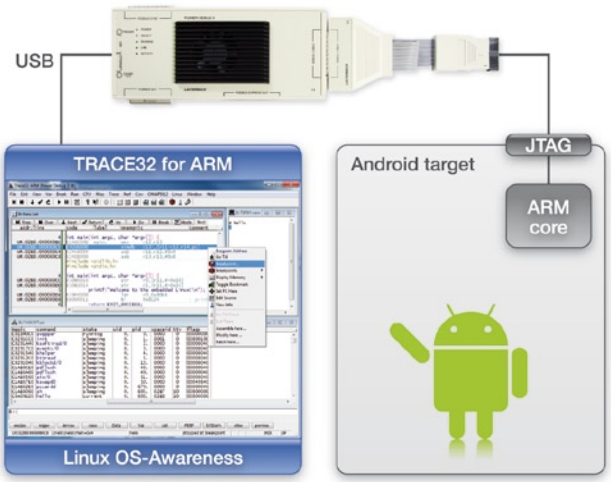


Fig. 4: In stop-mode debugging, the debugger communicates directly with the processor on the Android hardware platform.

A characteristic of stop-mode debugging is that when the processor is stopped for debugging, the whole Android system stops.

Stop-mode debugging has some big advantages:

- It needs only a functioning JTAG communication between the debugger and the processor.
- It needs no debug server on the target and is therefore very suitable for testing release software.
- It permits testing under real-time conditions and therefore enables efficient troubleshooting for problems that only occur in such conditions.

»

At present, stop-mode debugging does not support the debugging of VM applications such as on the Dalvik VM. Therefore transparent debugging through all of the software layers is not yet possible.

Debugging Java Code

Java code for Android is usually tested with the Android Development Tools (ADT) integrated into Eclipse. The adb server – adb stands for Android Debug Bridge – on the host communicates over USB or Ethernet with the adb daemon on the target (Fig. 5).

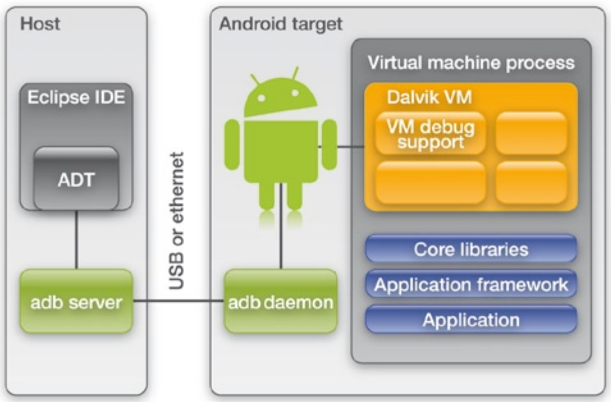


Fig. 5: The Android Development Tools (ADT) integrated in Eclipse for debugging Java code.

Prerequisites for debugging with ADT are VM applications specially compiled for debugging and Android debug support (adb daemon) running on the hardware platform.

Debugging Java code with ADT is comfortable. However, there are a few cases in which ADT cannot help you. These are:

- Errors that first occur with the release code.
- Errors that first occur when the Java application interacts with a service offered in C/C++ or a Linux hardware driver.
- Debugging following a communication breakdown between adb server and adb daemon.

VM Aware Stop-Mode Debugging

To enable thorough testing of an Android system from the Java application down to the Linux hardware driver under real-time conditions, Lauterbach is currently adding VM debugging awareness to its stop-mode debugging.

The JTAG debugger communicates directly with the processor on the Android hardware platform. The debugger can therefore access all system information after the processor stops. The “fine art” for the debugger is now to find the correct information and make it easy to understand for the user, abstracted from bits and bytes.

One abstraction level has given TRACE32 users the option of debugging operating system software even over several virtual address spaces. Another abstraction level, up to now independent of operating-system debugging, is Java debugging.

To debug applications running on VMs in systems like Android, where the VMs themselves are instantiated within the operating-system processes, operating-system debugging and Java debugging now have to be combined. To implement this new complexity, Lauterbach is developing a new, open, and easy-to-expand solution.

The Open Solution

In the future, stop-mode debugging from Lauterbach will support the following abstraction levels:

- High-level language debugging
- Target OS debugging awareness
- VM debugging awareness

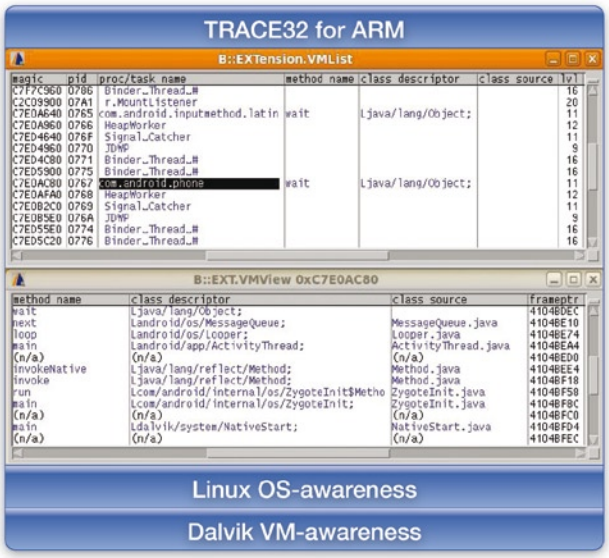
High-level language debugging is a fixed component of the TRACE32 software and is configured for a program with the loading of the symbol and debug information.

»

Dalvik Virtual Machine

Dalvik is the name of the virtual machine used in Android. The Dalvik Virtual Machine is a software model of a processor that executes byte code derived from Java. Virtual machines permit the writing of processor-independent software. If you switch to a new hardware platform, you only have to port the virtual machine.

Software compiled for a VM runs automatically on any platform to which this VM is ported.



```
magic    pid  proc/task name               method name class descriptor       class source lvl
C7F7C960 0786  Binder_Thread_#                                                               16
C2C09900 07A1  r.MountListener                                                               20
C7E0A640 0765 com.android.inputmethod.latin wait       Ljava/lang/Object;                    11
C7E0A960 0766  HeapWorker                                                                    12
C7ED4640 076F  Signal_Catcher                                                                11
C7ED4960 0770  JDWP                                                                           9
C7ED4C80 0771  Binder_Thread_#                                                               16
C7ED5900 0775  Binder_Thread_#                                                               16
C7E0AC80 0767 com.android.phone             wait       Ljava/lang/Object;                    11
C7E0AFA0 0768  HeapWorker                                                                    12
C7E0B2C0 0769  Signal_Catcher                                                                11
C7E0B5E0 076A  JDWP                                                                           9
C7ED55E0 0774  Binder_Thread_#                                                               16
C7ED5C20 0776  Binder_Thread_#                                                               16
```

```
method name   class descriptor                       class source        frameptr
wait          Ljava/lang/Object;                                         4104BDEC
next          Landroid/os/MessageQueue;              MessageQueue.java   4104BE10
loop          Landroid/os/Looper;                    Looper.java         4104BE74
main          Landroid/app/ActivityThread;           ActivityThread.java 4104BEA4
(n/a)         (n/a)                                  (n/a)               4104BED0
invokeNative  Ljava/lang/reflect/Method;             Method.java         4104BEE4
invoke        Ljava/lang/reflect/Method;             Method.java         4104BF18
run           Lcom/android/internal/os/ZygoteInit$Metho ZygoteInit.java  4104BF58
main          Lcom/android/internal/os/ZygoteInit;   ZygoteInit.java     4104BF8C
(n/a)         (n/a)                                  (n/a)               4104BFC0
main          Ldalvik/system/NativeStart;            NativeStart.java    4104BFD4
(n/a)         (n/a)                                  (n/a)               4104BFEC
```

Fig. 6: For the reference implementation, Linux OS-awareness and Dalvik VM-awareness have to be loaded in TRACE32.

Target-OS debugging awareness must always be configured by the TRACE32 user. There are example configurations available for all common operating systems. The RTOS API provides an option to be customized for proprietary operating systems.

VM debugging awareness is a fixed component of the TRACE32 software for J2ME CLDC, J2ME CDC and Kaffe. All other virtual machines have to be adapted individually with the VM API. A ready-to-use configuration is available for the very popular Android Dalvik VM.

The open solution, both for the operating system and for the virtual machine, enables providers of closed-source VMs to write a TRACE32 VM awareness for their product and offer it to their customers.

The Reference Implementation

To be able to debug thoroughly on an ARM-based Android target from the Java applications right down to the Linux hardware drivers, TRACE32 requires the following extensions (see Fig. 6):

- A Linux OS-awareness as provided by Lauterbach since 1998.
- A Dalvik VM-awareness, which can be downloaded from the Lauterbach homepage. This just has to be configured for the platform used.

www.lauterbach.com/vmandroid.html

It is now possible to identify and list all Java applications now being run (EXTension.VMList in Fig. 6) and to analyze and view the VM stack for a selected Java application (EXTension.VMView in Fig. 6).

The next step planned is to display the source code currently being run by the VM. The aim of the development is of course stop-mode debugging for VM applications with all the functions of a modern debugger.

New Supported RTOS	
DSP/BIOS for ARM	Q2/2011
OSEK/ORTI SMP	Q2/2011
T-Kernel for ARM	available
Windows Embedded Compact 7 for ARM	available
µC/OS-III for ARM	available

Extensions and New RTOS Versions

- TRACE32 scripts were adapted for Timesys embedded Linux.

- OSEK/ORTI now ensures that NEXUS ownership trace messages are generated for task changes. This enables TRACE32 to make task-aware runtime measurements for the MPC55xx/MPC56xx, even if NEXUS generates no data trace messages.

The following version adaptations have been made or are planned:

- OSEck 4.0
- QNX 6.5.0
- Symbian^3 for ARM
- Symbian^4 planned for Q1/2011
- Windows CE6 for Atom™

Serial Trace Port Usage Growing

Faster, higher, stronger! Not only is this the motto of many sports – it has even been raised to a core principle in microelectronics. Ever faster clock speeds and a greater parallelization of processing steps have given us an astonishingly constant increase in processing speed for decades. It is no wonder that designers have also followed this motto for the transmission of trace information.

The trace interface, over which the processors deliver the detailed information on the operation of their inner processes, has struggled to keep up with the growing flood of information. For many developers of embedded systems it would be unthinkable to undertake a development without this important information, so all sorts of efforts have been made to increase the data throughput of the trace interface. For many years the increase in clock frequency and a greater bus-width at the trace port were an effective way of increasing data volumes.

However, these measures have their price. Not only does a wider trace port take up highly coveted package pins but poor signal quality at higher clock frequencies requires compensation on all signals from the trace bus. Thanks to the sophisticated algorithms of its AutoFocus technology, Lauterbach is able to ensure error-free recording of high-frequency trace signals.

As processor architectures continue to gain in speed and complexity through parallelization, the trace interfaces are starting to use a high-speed data transfer method that has been in use in other areas for a long time. A high-speed serial transmission is used in SATA, Fibre Channel, PCI Express, and USB3.0 (SuperSpeed USB). The extremely high data rates more than compensate for the disadvantage of only a few differential data lines.

The integration of high-speed serial interfaces on the chip is expensive and can initially cause problems. As just one example, the I/O pads have to be operated at a much higher speed. But with the increasing experience in the implementation of serial interfaces in the gigahertz range the knowledge gained can be used to solve many of the problems arising with the serial trace ports.

In 2008, ARM implemented this technology with its High Speed Serial Trace Port – HSSTP for short. This was quickly followed by AMCC with the Titan, Freescale with the QorIQ processors P4040 and P4080, as well as Marvell with the SETM3.

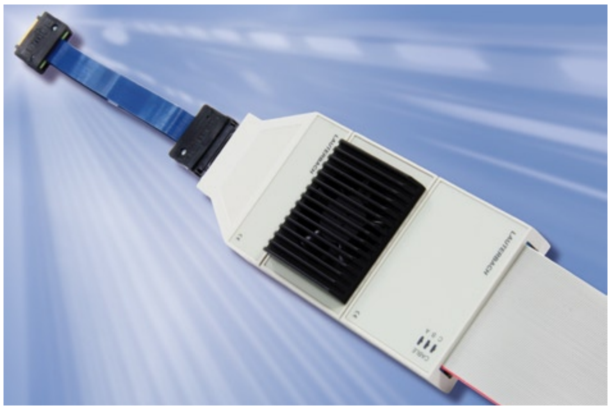


Fig. 7: Following firmware and software adaptations, a universal hardware supports the most varied protocols of serial trace interfaces.

Lauterbach had designed a hardware interface for the serial trace in 2008. A universal preprocessor was developed on the basis of the Aurora protocol. Only the firmware and software have to be changed to record any of the alternative protocols. This means that our system is already prepared for further variants of serial trace protocols.

Supported Serial Trace Ports

AMCC	APM83290 Program flow	2009
ARM-HSSTP	ETMv3, PTM, CoreSight ETMv3, CoreSight PTM Program flow, Data flow and Context-ID	2008
Freescale	NEXUS QorIQ P4040 and P4080 Branch Trace and Ownership Trace Messages, Data Write Messages	2010
Marvell-SETM3	CoreSight ETMv3 Program flow, Data flow and Context-ID	2009

Higher Transmission Rate for Real-Time Streaming

“Real-time Streaming” means transferring trace data to the host whilst it is being recorded and analyzing it there immediately. This requires the transmission of large volumes of data from the trace tool to the host, especially for CPU-intensive applications and multicore systems. To make TRACE32 fit for these application scenarios, the trace data is compressed by the trace tool, PowerTrace II, before being transferred to the host. This feature has been supported by the TRACE32 software since December 2010.

Real-time Streaming is currently implemented for the ARM trace protocols ETMv3 and PTM.

Hardware Compression

The maximum transmission rate to the host is still the bottle-neck for *Real-time Streaming*. Even with a peer-to-peer GB Ethernet interface between the trace tool and the host, the maximum is currently only about 500 MBit/s net. This maximum transmission rate has to be sufficient to transfer all data at the trace port without loss to the host.

To be able to estimate the actual data volume to be transmitted, it is important to know the conditions of *Real-time Streaming:*

1. The main applications for *Real-time Streaming* are code coverage and run-time measurements. For both functions, it is sufficient if only the program trace information is exported. To get a very accurate run-time measurement, cycle-accurate tracing can be enabled.

2. For a realistic estimate of the necessary data rate, you just have to consider the average load at the trace port. Peak loads at the trace port are intercepted by PowerTrace II, which can be considered as a large FIFO (up to 4 GB). Fig. 8 shows an overview of the average / maximum load at the trace port for Cortex cores. The application running on the Cortex core ultimately determines the actual load.

By implementing FPGA-based hardware compression in PowerTrace II, the transmission rate to the host was raised to 3.2 GBit/s.

Pure Long-Time Trace

If trace data is analyzed and also saved to the hard-disk during *Real-time Streaming*, Lauterbach considers this a *Long-time Trace*.

To provide long-time tracing for other trace protocols such as Nexus, Lauterbach is now offering pure streaming onto the hard-disk without simultaneous analysis. This means that trace recording of up to 1 tera-frames is possible for a 64-bit host operating system.

For detailed information on *Real-time Streaming* and *Long-time Trace*, go to the Lauterbach homepage at: **www.lauterbach.com/tracesinks.html**

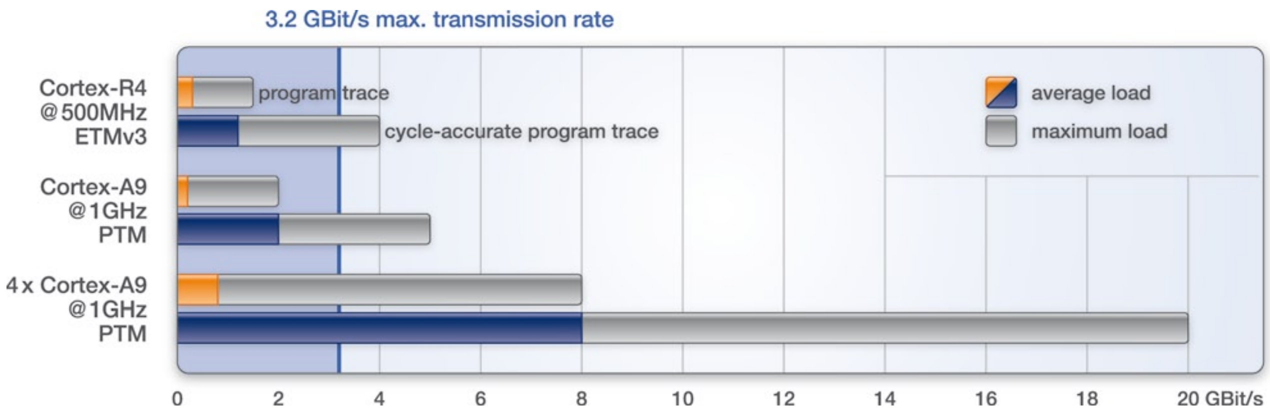


Fig. 8: A transmission rate of 3.2 GBit/s is usually enough to transfer program trace information to the host while it is being recorded.

Energy Profiling with the CombiProbe

The TRACE32 CombiProbe can now also be used for measuring the energy used by applications.

The following analyses are possible:

- The current/voltage profile at up to three measurement points can be displayed directly linked to the code running on the processor.
- The energy consumption of the entire system can be analysed for the individual functions.

Which part of a program uses the most energy? What influence does a program modification have on the energy requirements of an embedded system? These are the questions that can now be dealt with by the CombiProbe.

To determine the energy consumption for every point of the program, the following measurement data has to be collected:

- The program flow being exported via the trace port of the processor.
- The current and voltage profile measured at suitable measurement points on the target hardware.

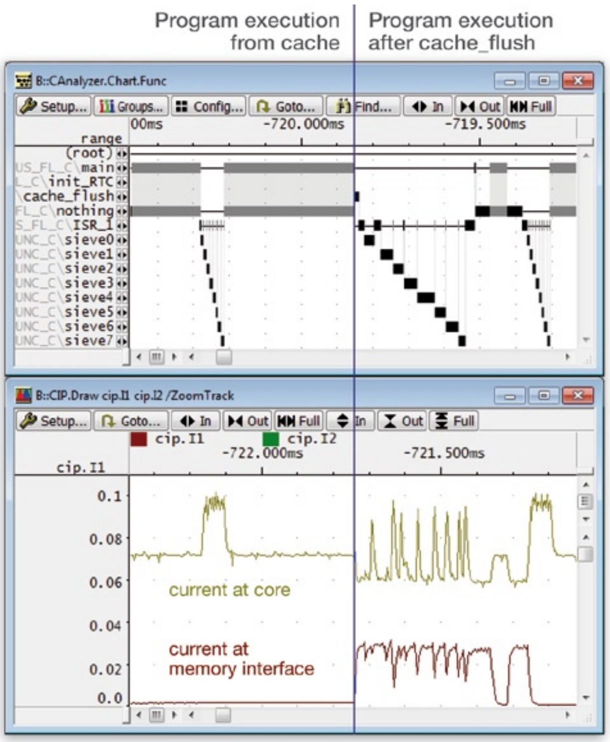
The current and voltage development for up to three power domains can now be identified by connecting a *TRACE32 Analog Probe* to the CombiProbe.

CombiProbe

The CombiProbe is a debug cable that also contains a 128 MB trace memory. The CombiProbe was specially developed for processors with a 4-bit trace port. Program flow recording is currently supported for the following trace protocols:

- ARM-ETMv3 in continuous mode (ARM)
- IFLOW Trace for PIC32 (Microchip)
- MCDS Trace for X-GOLD102 and X-GOLD110 (Infineon)

www.lauterbach.com/cobstm.html



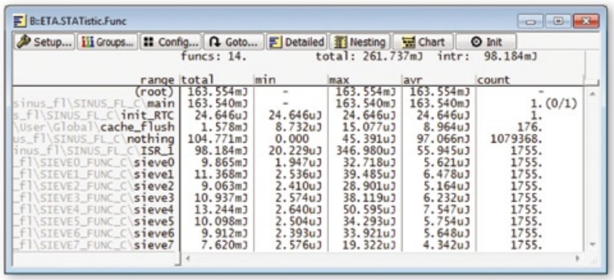
Program execution from cache
Program execution after cache_flush
B::CAnalyzer.Chart.Func
Setup... Groups... Config... Goto... Find... In Out Full
-720.000ms
-719.500ms
range
(root)
main
init_RTC
cache_flush
nothing
ISR_1
sieve0
sieve1
sieve2
sieve3
sieve4
sieve5
sieve6
sieve7
B::CIP.Draw cip.I1 cip.I2 /ZoomTrack
Setup... Goto... In Out Full In Out Full
cip.I1
cip.I2
-722.000ms
-721.500ms
0.1
0.08
0.06
0.04
0.02
0.0
current at core
current at memory interface

Fig. 9: A program section not running from the cache needs more time and uses more current.

Since all measurement data is time-stamped by the global timer of the CombiProbe, you can quickly and easily see the direct connection between executed program code and the power consumption as well as the voltage profile of the system.

Fig. 9 shows that a program section running from external memory instead of cache not only needs much more processing time but also uses more power at the external memory.

Fig. 10 shows the energy consumption as a statistical analysis.



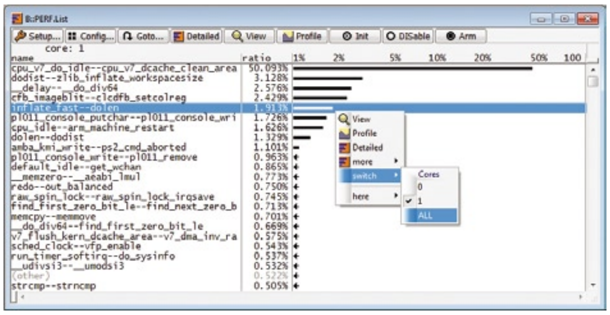
B::ETA.STATistic.Func
Setup... Groups... Config... Goto... Detailed Nesting Chart Init

funcs: 14.   total: 261.737mJ   intr: 98.184mJ

range	total	min	max	avr	count
(root)	163.554mJ	-	163.554mJ	163.554mJ	-
sinus_fl\SINUS_FL_C\main	163.540mJ	-	163.540mJ	163.540mJ	1.(0/1)
s_fl\SINUS_FL_C\init_RTC	24.646uJ	24.646uJ	24.646uJ	24.646uJ	1.
\User\Global\cache_flush	1.578mJ	8.732uJ	15.077uJ	8.964uJ	176.
us_fl\SINUS_FL_C\nothing	104.771mJ	0.000	45.391uJ	97.066nJ	1079368.
inus_fl\SINUS_FL_C\ISR_1	98.184mJ	20.229uJ	346.980uJ	55.945uJ	1755.
_fl\SIEVE0_FUNC_C\sieve0	9.865mJ	1.947uJ	32.718uJ	5.621uJ	1755.
_fl\SIEVE1_FUNC_C\sieve1	11.368mJ	2.536uJ	39.485uJ	6.478uJ	1755.
_fl\SIEVE2_FUNC_C\sieve2	9.063mJ	2.410uJ	28.901uJ	5.164uJ	1755.
_fl\SIEVE3_FUNC_C\sieve3	10.937mJ	2.574uJ	38.119uJ	6.232uJ	1755.
_fl\SIEVE4_FUNC_C\sieve4	13.244mJ	2.640uJ	50.595uJ	7.547uJ	1755.
_fl\SIEVE5_FUNC_C\sieve5	10.098mJ	2.504uJ	34.293uJ	5.754uJ	1755.
_fl\SIEVE6_FUNC_C\sieve6	9.912mJ	2.393uJ	33.921uJ	5.648uJ	1755.
_fl\SIEVE7_FUNC_C\sieve7	7.620mJ	2.576uJ	19.322uJ	4.342uJ	1755.

Fig. 10: The minimum, maximum, and average energy consumption of individual functions.

SMP Profiling



```
      core: 1
name                                      ratio    1%    2%     5%    10%   20%      50%   100
cpu_v7_do_idle--cpu_v7_dcache_clean_area  50.093%
dodist--zlib_inflate_workspacesize         3.128%
__delay--__do_div64                        2.576%
cfb_imageblit--clcdfb_setcolreg            2.429%
inflate_fast--dolen                        1.913%
pl011_console_putchar--pl011_console_wri   1.726%
cpu_idle--arm_machine_restart              1.626%
dolen--dodist                              1.329%
amba_kmi_write--ps2_cmd_aborted            1.101%
pl011_console_write--pl011_remove          0.963%
default_idle--get_wchan                    0.865%
__memzero--__aeabi_lmul                    0.773%
redo--out_balanced                         0.750%
raw_spin_lock--raw_spin_lock_irqsave       0.745%
find_first_zero_bit_le--find_next_zero_b   0.713%
memcpy--memmove                            0.701%
__do_div64--find_first_zero_bit_le         0.669%
v7_flush_kern_dcache_area--v7_dma_inv_ra   0.575%
sched_clock--vfp_enable                    0.543%
run_timer_softirq--do_sysinfo              0.537%
__udivsi3--__umodsi3                       0.532%
(other)                                    0.522%
strcmp--strncmp                            0.505%
```

Fig. 11: Sample-based profiling shows the ratio of individual code sections compared to the overall run-time as a percentage. The result can be displayed both for the individual cores of the SMP system (here for core 1) as well as for the total of all cores.

Sample-based profiling was completely reworked in 2010. Important innovations include a new operating concept for measuring, a self-calibrating sampling rate and an extension for SMP systems.

SMP Profiling for Functions

From October 2010, profiling data for SMP systems can now be collected. To create function level profiling, TRACE32 cyclically reads the program counters of the individual cores and saves them in a database. The profiling can then be shown for the individual cores and also as a total for all cores.

Since many chips provide the capability of reading the program counter whilst the processor is executing, this measurement can be made in real-time for the following architectures:

- **ARM / Cortex:** ARM11 MPCore, Cortex-A5 MPCore, Cortex-A9 MPCore, Cortex-A15 MPCore
- **MIPS32:** MIPS34K, MIPS1004K
- **MIPS64:** Broadcom BCM7420

If the chip's debug logic does not permit non-intrusive reading of this information, the individual cores will have to be briefly stopped periodically to obtain this information.

SMP Profiling for Tasks

To create a task profile, the task ID for the individual cores has to be read cyclically from the memory. Many chips allow the physical memory to be read at program run-time. If the on-chip debug logic supports this feature, the measurement can be made in real-time:

- **ARM / Cortex:** ARM11 MPCore, Cortex-A5 MPCore, Cortex-A9 MPCore, Cortex-A15 MPCore
- **Power Architecture:** MPC8641D, MPC8572, QorIQ

Otherwise, the individual cores have to be briefly stopped to read the data required.

Sample-Based Profiling

For Sample-based Profiling, the program counter or the variable containing the ID of the current task is periodically read. On the basis of this information, the ratio of a function or task compared to the overall run-time is shown as a percentage.

Symmetrical Multiprocessing (SMP)

A multicore chip consisting of identical cores can be configured as an SMP system. An SMP operating system distributes the pending processes (tasks) dynamically to individual cores at program run-time (but not before). For debugging SMP systems, a single TRACE32 instance is opened from which all cores are monitored.

WORLDWIDE BRANCHES

- USA
- Germany
- France
- UK
- Italy
- China
- Japan

Represented by experienced partners in all other countries

KEEP US INFORMED

If your address has changed or if you no longer want to be on our mailing list, please send us an e-mail to info_us@lauterbach.com.